NightView User's Guide

0890395-220
March 2001

The information contained in this document is believed to be correct at the time of publication. It is subject to change without notice. Concurrent Computer Corporation makes no warranties, expressed or implied, concerning the information contained in this document.

To report an error or comment on a specific portion of the manual, photocopy the page in question and mark the correction or comment on the copy. Mail the copy (and any additional comments) to Concurrent Computer Corporation, 2881 Gateway Drive, Pompano Beach, FL 33069-4324. Mark the envelope **"Attention: Publications Department**." This publication may not be reproduced for any other reason in any form without written permission of the publisher.

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original Release -- July 1992	010	NightView 1.1
Previous Release -- August 2000	210	NightView 5.2
Current Release -- March 2001	220	NightView 5.3

Preface

General Information

NightView is a general purpose source-level program debugger. Some of the features make it useful for debugging systems of real-time programs, but it can also be used to debug a single ordinary program.

NightView can debug programs written in multiple languages. Ada, C, C++ and Fortran are supported.

NightView can debug multiple processes on the local system or on different hosts.

NightView has been designed to be as flexible as possible. The NightView command interpreter includes macro processing so that you can write your own NightView commands.

You communicate with NightView with one of three user interfaces. The command-line interface is useful when no advanced terminal capabilities are present. A simple full-screen interface is available for ASCII terminals. The graphical user interface provides the most functionality.

NightView is supported on systems running PowerMAX OS™. See the Hardware Prerequisites section of the *NightView Release Notes* associated with your particular version for a list of supported systems. The NightView user interfaces are supported on Intel® systems running Red Hat® Linux® 6.1 or later. The Intel/Red Hat interfaces can be used to debug processes on PowerPC/PowerMAX OS.

Scope of Manual

This document is the user manual for the NightView debugger. It is intended for anyone using NightView, regardless of their previous level of experience with debuggers. This manual describes how to use NightView, by way of tutorial and reference guide. There is also material for system administrators.

Structure of Manual

The manual begins with the short tutorials, Chapter 1 [A Quick Start] on page 1-1 and Chapter 2 [A Quick Start - GUI] on page 2-1, giving you just enough information to get you started. For more complete tutorials, see Chapter 4 [Tutorial] on page 4-1 and Chapter 5 [Tutorial - GUI] on page 5-1.

The next section describes the major concepts you will need to understand in order to get the best use out of NightView. See Chapter 3 [Concepts] on page 3-1.

More detailed information about the NightView commands is found in Chapter 7 [Command-Line Interface] on page 7-1.

The next chapter describes a simple full-screen interface to NightView. See Chapter 8 [Simple Full-Screen Interface] on page 8-1.

The next chapter describes the graphical user interface for NightView. See Chapter 9 [Graphical User Interface] on page 9-1.

This manual also contains several appendixes that may not be of interest to all users, such as an implementation overview. A glossary of terms related to NightView and a quick reference guide are also provided.

Syntax Notation

The following notation is used throughout this guide:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms and comments in code may also appear in *italic*.
`list bold`	User input appears in **`list bold`** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **`list bold`** type.
`list`	Operating system and program output such as prompts and messages and listings of files and programs appears in `list` type. Keywords also appear in `list` type.
emphasis	Words or phrases that require extra emphasis use *emphasis* type.
window	Keyboard sequences and window features such as push buttons, radio buttons, menu items, labels, and titles appear in window type.
`[ ]`	Brackets enclose command options and arguments that are optional. Mutually exclusive choices are separated by the pipe (|) character. You do not type the brackets (or the pipe character) if you choose to specify such options or arguments.
`{ }`	Braces enclose mutually exclusive choices separated by the pipe (|) character, where one choice must be selected. You do not type the braces or the pipe character with the choice.
`...`	An ellipsis follows an item that can be repeated.
`::=`	This symbol means *is defined as* in Backus-Naur Form (BNF).

Related Publications

0800032	*PowerPC Microprocessor Family: The Programming Environments*
0890161	*The C Programming Language*
0890240	*Hf77 FORTRAN Reference Manual*

0890288	*HAPSE Reference Manual*
0890300	*X Window System User's Guide*
0890380	*OSF/Motif Documentation Set*
0890382	*UNIX® System V AT&T C++ Language System Release 2.1*
0890398	*NightTrace Manual*
0890429	*System Administration Volume 1*
0890460	*Compilation Systems Volume 2 (Concepts)*
0890475	*NightView Pocket Reference*
0890497	*C++ Reference Manual*
0890516	*MAXAda Reference Manual*
0891019	*Harris C Reference Manual*
0891055	*Élan License Manager™ Release Notes*

Contents

Chapter 4 Tutorial

Chapter 5 Tutorial - GUI

1
A Quick Start

This chapter is for people who want to start using the command-line version of the debugger before reading the whole manual. You may also be interested in the graphical-user-interface (GUI) version of this chapter in Chapter 2 [A Quick Start - GUI] on page 2-1. There is a more thorough tutorial in Chapter 4 [Tutorial] on page 4-1.

If you are familiar with the GNU debugger, gdb™, you should have very few problems with NightView. The commands are almost all identical. The biggest difference between NightView and other debuggers is how you tell NightView what program to debug and how you start that program.

If you get any errors, the error message tells which section of the manual can help you determine what went wrong. At any time, you can ask the debugger to display help on an error message by mentioning that section's name as the argument to the **`help`** command (see “help” on page 7-111).

The rest of this chapter goes through a sample debug session on a small program. Feel free to dive right into the debugger. If you get into trouble, use the **`help`** command to get out of it.

Sample Program

This section lists the program used as an example through the remainder of the chapter. The program does not have any bugs in it; it will be used to show how to run a program, set breakpoints, look at variables, etc. You can copy this file from **`/usr/lib/NightView/fact.c`** into your own directory. The following program is in the file **`fact.c`**:

```
 1   #include <stdio.h>
 2
 3   static int factorial(x)
 4      int x;
 5   {
 6      if (x <= 1) {
 7         return 1;
 8      } else {
 9         return x * factorial(x-1);
10      }
11   }
12
13   void
14   main(argc, argv)
15      int argc;
16      char ** argv;
17   {
18      int i, errors;
19      for (i = 1; i < argc; ++i) {
20         long xl;
21         int x;
22         int answer;
23         char * ends = NULL;
24         xl = strtol(argv[i], &ends, 10);
25         x = (int)xl;
26         answer = factorial(x);
27         printf("factorial(%d) == %d\n", x, answer);
28      }
29      exit(0);
30   }
```

The remainder of this chapter assumes that you compiled **`fact.c`** and put the resulting executable in **`fact`**:

```
cc -g -o fact fact.c
```

Starting Up

You can start NightView with or without a program name. If you start it with a program name, NightView offers you the chance to debug the program in a dialogue shell (see "Dialogues" on page 3-4). If you start NightView without a program name or you want to debug another program, you must execute the program with the **`run`** command (see "run" on page 7-30) in a dialogue shell.

Below is an example of starting up the debugger with a program name. Note that throughout the quick start, the version and the link time might not match exactly for your version of NightView. Also, some of the shell output and other messages may not come out exactly as shown. Some messages might not appear, or additional messages might appear, depending on your environment.

```
$ nview -nogui ./fact
NightView debugger - Version 5.1, linked Thu Jan 13 10:24:51 EST 2000
Copyright (C) 2000, Concurrent Computer Corporation

In case of confusion, type "help"
```

Note that you invoked NightView with a program name argument **`./fact`**. NightView responded with information about the debugger.

Now NightView will prompt you for information about running the program.

```
Do you want to debug program './fact'? y
Type in the arguments you want to supply to program './fact'.
Arguments: 7
New process: local:2347 parent pid: 2340
Process local:2347 is executing /users/bob/fact.
Reading symbols from /users/bob/fact...done
Executable file set to /users/bob/fact
/usr/lib/NightView/ReadyToDebug
$ /usr/lib/NightView /ReadyToDebug
$./fact 7
(local)
```

NightView requested information about the program and its arguments and you complied.

NightView always runs a special program, **`/usr/lib/NightView/ReadyToDebug`**. This program helps NightView synchronize with the shell. That's why you see that line in the output. You might see only one echo of **`/usr/lib/NightView/ReadyToDebug`**, depending on how quickly the dialogue shell starts. The dollar signs ("$") are prompts from the shell.

NightView automatically created a dialogue named `local`; it also displayed the string `local` as the prompt, showing that by default, commands apply to that dialogue (or the processes running in that dialogue).

The debugger waited for the new program to get started. Because sending input to a dialogue is just like typing commands to a shell (the dialogue is really running the same shell program you normally use), this caused the **`fact`** program to be executed with the single argument **`7`**.

If the **`fact`** program had required input, you would have used the **`!`** command to send the input to the program. See “!” on page 7-27.

When the dialogue executed the program, NightView got control and informed you that a new process was just started in dialogue `local` and told you that the process id was 2347.

Because this is the only program running in dialogue `local`, you do not have to do anything special to cause any commands you type to refer to this process; the default qualifier is already set to `local`, so commands will automatically apply to the one process running there.

Getting Help

Next you will enter a bogus command. Note that throughout this section, the help text and display size may not exactly match your NightView session.

```
(local) foo
Error: Unrecognizable command "foo". [E-command_proc003]
```

NightView responded to the bogus command with an error message and an error code ([E-command_proc003]).

Now get NightView to tell you more about the error message.

```
(local) help
E-command_proc003:
Unrecognizable command "string".

STRING is not a valid NightView command.  See "Summary of
Commands".
```

You typed **help** without any arguments to see more information about the error message. NightView showed the extended error information.

In the command-line and and simple screen interfaces, online help is available only for error messages. Consult a printed manual or view the online help with NightView's graphical user interface or with **nhelp(1)**.

If you are familiar with **gdb**, the remainder of this chapter will be fairly boring because (once you get the program started) NightView and **gdb** look very much alike (at least for all the commands demonstrated in this simple example).

Setting a Breakpoint

You will now use the **list** command to look at the source.

```
(local) l 1
1    | #include <stdio.h>
2    |
3    | static int factorial(x)
4    |    int x;
5    | {
6  * |    if (x <= 1) {
7  * |       return 1;
8    |    } else {
9  * |       return x * factorial(x-1);}
10   |    }
(local)
```

You told the **list** command (abbreviated to **l** in this example) to list at line 1.

You now decide where you want to set a breakpoint. An interesting spot in this program is the `return` statement in the recursive routine `factorial` where it is about to start backing out of the recursive calls.

```
(local) b 7
local:2347 Breakpoint 1 set at fact.c:7
(local)
```

The `return` was on line 7, so you used the **`breakpoint`** command (abbreviated to **`b`**) to set a breakpoint on line 7.

Complete descriptions of the commands you used here appear in “list” on page 7-58 and “breakpoint” on page 7-79.

Finishing up

Now run the program until it reaches the breakpoint.

```
(local) c
local:2347: at Breakpoint 1, 0x100026fc in factorial(int
x = 1) at fact.c line 7
7  B=|       return 1;
(local)
```

You used the **`continue`** command (abbreviated to **`c`**) without any arguments. This told the program to start running. It ran until it hit the breakpoint that you had set on line 7. Note that your process ID and addresses will differ.

Now look at the call stack.

```
(local) bt
#0  0x100026fc  in factorial(int x = 1) at fact.c line 7
#1  0x1000271c  in factorial(int x = 2) at fact.c line 9
#2  0x1000271c  in factorial(int x = 3) at fact.c line 9
#3  0x1000271c  in factorial(int x = 4) at fact.c line 9
#4  0x1000271c  in factorial(int x = 5) at fact.c line 9
#5  0x1000271c  in factorial(int x = 6) at fact.c line 9
#6  0x1000271c  in factorial(int x = 7) at fact.c line 9
#7  0x10002784  in main(int argc = 2, char **argv =
0x2ff7eaec)
                 at fact.c line 26
(local)
```

You used the **`bt`** (**`backtrace`**) command to display the call stack. You saw all the expected recursive calls (see “backtrace” on page 7-65).

Now look at the value of the variable `x`.

```
(local) p x
$1: x = 1
(local)
```

You used the **`p`** (**`print`**) command to print the variable `x`, verifying that it was equal to 1.

Now finish running the program.

```
(local) c
factorial(7) == 5040
Process local:2347 is about to exit normally
```

```
#0  0x100027ac  in main(int argc = 2, unsigned char
**argv = 0x2ff7eaec)
                at fact.c line 29
29 <>|    exit(0);
(local)
```

You used the **c** (**continue**) command to allow the process to run to completion.

Exit from NightView.

```
(local) q
Kill all processes being debugged? y
You are now leaving NightView...
Process local:2347 exited normally
Dialogue local has exited.
$
```

Finally you typed **q** (**quit**) to leave the debugger. The **fact** program had not fully exited, so NightView prompted, asking if the program should be killed. You responded with **y**, and the sample session ended. The commands used in this section appear in "continue" on page 7-97, "backtrace" on page 7-65, "print" on page 7-66, and "quit" on page 7-17.

2
A Quick Start - GUI

This chapter is for people who want to start using the graphical-user-interface (GUI) version of the debugger before reading the whole manual. You may also be interested in the command-line version of this chapter in Chapter 1 [A Quick Start] on page 1-1. There is a more thorough tutorial in Chapter 5 [Tutorial - GUI] on page 5-1.

In this manual, the words click, drag, press, and select always refer to mouse button 1.

This entire manual is available through the online help system built into the debugger. If you get any errors, the error message tells which section of the manual can help you determine what went wrong. At any time, you can ask the debugger to display any section of the manual by clicking on the Help menu or using the H mnemonic. See "Help Menu" on page 9-3. Click on the Table of Contents menu item or use the n mnemonic. NightView puts up a Help Window that displays the table of contents for the manual. See "Help Window" on page 9-50. You can read this manual section by clicking on A Quick Start - GUI.

The rest of this chapter goes through a sample debug session on a small program. Feel free to dive right into the debugger. If you get into trouble, use the Help menu to get out of it.

Sample Program - GUI

This section lists the program used as an example through the remainder of the chapter. The program does not have any bugs in it; it will be used to show how to run a program, set breakpoints, look at variables, etc. You can copy this file from **`/usr/lib/NightView/fact.c`** into your own directory. The following program is in the file **`fact.c`**:

```
 1   #include <stdio.h>
 2
 3   static int factorial(x)
 4      int x;
 5   {
 6      if (x <= 1) {
 7         return 1;
 8      } else {
 9         return x * factorial(x-1);
10      }
11   }
12
13   void
14   main(argc, argv)
15      int argc;
16      char ** argv;
17   {
18      int i, errors;
19      for (i = 1; i < argc; ++i) {
20         long xl;
21         int x;
22         int answer;
23         char * ends = NULL;
24         xl = strtol(argv[i], &ends, 10);
25         x = (int)xl;
26         answer = factorial(x);
27         printf("factorial(%d) == %d\n", x, answer);
28      }
29      exit(0);
30   }
```

The remainder of this chapter assumes that you compiled **`fact.c`** and put the resulting executable in **`fact`**:

```
cc -g -o fact fact.c
```

Starting Up - GUI

You can start NightView with or without a program name. If you start it with a program name, NightView offers you the chance to debug the program in a dialogue shell (see "Dialogues" on page 3-4). If you start NightView without a program name or you want to debug another program, you must execute the program in the dialogue I/O area (see "Dialogue I/O Area" on page 9-17). (The dialogue I/O area is labeled Dialogue I/O: Run your programs in this shell.)

Below is an example of starting up the debugger with a program name. Note that throughout the quick start, the version and the link time might not match exactly for your version of NightView. Also, some of the messages might not come out exactly as shown. Some messages might not appear, or additional messages might appear, depending on your environment.

```
$ nview ./fact
```

NightView displays the Dialogue Window and a dialog box. See “Dialogue Window” on page 9-16 and “Program Arguments Dialog Box” on page 9-19. The dialog box says the following:

```
To debug program './fact', enter any command-line
arguments you want to supply to the program and press OK.

Press Cancel if you do not want to debug program
'./fact'.
```

Enter the number 7 as an argument and click on the OK button.

The dialogue I/O area displays the following information:

```
/usr/lib/NightView-release/ReadyToDebug
$ /usr/lib/NightView-release/ReadyToDebug
$ ./fact 7
```

NightView always runs a special program, **`/usr/lib/NightView-`***release*`/ReadyToDebug`** (*release* is the NightView release level). This program helps NightView synchronize with the shell. That's why you see that line in the output. You might see only one echo of `/usr/lib/NightView-`*release*`/ReadyToDebug`, depending on how quickly the dialogue shell starts. The dollar signs (``$ '') are prompts from the shell.

When NightView started, it automatically created a dialogue named `local`; it also displayed the string `local` as the qualifier, showing that by default, commands apply to that dialogue (or the processes running in that dialogue).

Your answers to the dialog box sent the line `./fact 7` to the `local` dialogue and caused the debugger to wait for the new program to get started. Because sending input to a dialogue is just like typing commands to a shell (the dialogue is really running the same shell program you normally use), this caused the **`fact`** program to be executed with the single argument 7.

If the **`fact`** program had required input, you would have typed the input into the dialogue I/O area.

NightView puts up a Debug Window (see “Debug Window” on page 9-20). The debug message area (see “Debug Message Area” on page 9-28) contains a message like the following:

```
New process:  local:2347     parent pid: 2340
Process local:2347 is executing /users/bob/fact.
Reading symbols from /users/bob/fact...done
Executable file set to
/users/bob/fact
Switched to process local:2347.
```

When the dialogue executed the program, NightView got control and informed you that a new process was just started in dialogue `local` and told you that the process id was 2347.

The debug identification area displays the program name **`fact`**. See “Debug Identification Area” on page 9-28. The debug source file name is **`fact.c`**. See “Debug Source

File Name" on page 9-29. The debug status area shows Stopped for exec. See "Debug Status Area" on page 9-29. The source code from file `fact.c` appears in the debug source display, centered around `main`. See "Debug Source Display" on page 9-31.

Getting Help - GUI

Next you will enter a bogus command. Note that throughout this section, the help text and display size may not exactly match your NightView session.

The debug command area is labeled Command:. Click in the debug command area (see "Debug Command Area" on page 9-35) and issue the following command:

```
foo
```

Press Return to enter the command.

NightView responded to the bogus command with the following message and error code:

```
Error:  Unrecognizable command "foo". [E-command_proc003]
```

Now get NightView to tell you more about the error message. Click on the Help menu or use the H mnemonic. See "Help Menu" on page 9-3. Click on the On Last Error menu item or use the E mnemonic. NightView puts up a Help Window that displays the following extended error information:

E-command_proc003

MESSAGE

ERROR: Unrecognizable command "*string*".

EXPLANATION

string is not a valid NightView command. See Summary of Commands.

Next, dismiss the Help Window by selecting Exit from the File menu. See "Help Window" on page 9-50.

Next you will read about the **`list`** command. Click on the Help menu or use the H mnemonic. See "Help Menu" on page 9-3. Click on the On Commands menu item or use the m mnemonic. NightView puts up the following Help Window with a menu of NightView commands.

Summary of Commands

This section gives a summary of all the commands in NightView. The table is organized alphabetically by command. The abbreviations for the commands are included with the corresponding commands, rather than alphabetically.

Also, remember that you can abbreviate commands by using a unique prefix.

!

Pass input to a dialogue.

agentpoint

Insert a call to a debug agent at a given location.

(etc.)

Most of the information would not fit on your display. The Help Window showed this by having only a small thumb or slider on the vertical scroll bar. Scroll down to the **list** command by moving the thumb or by clicking on the arrow heads of the vertical scroll bar. Click on the **list** command. NightView displayed the following Help Window with information about the **list** command.

list

List a source file. This command has many forms, which are summarized below.

list *where-spec*

List ten lines centered on the line specified by *where-spec*.

list *where-spec1*, *where-spec2*

List the lines beginning with *where-spec1* up to and including the *where-spec2* line.

(etc.)

To see more about the **list** command, you could move the thumb or click on the arrow heads of the vertical scroll bar. However, rather than reading more, you make the Help Window go away by selecting Exit from the File menu.

Setting a Breakpoint - GUI

You now decide where you want to set a breakpoint. An interesting spot in this program is the `return` statement in the recursive routine `factorial` where it is about to start backing out of the recursive calls.

Click on the line with the return statement (line 7) in the debug source display. Then click on the Breakpoint debug command button.

The `return` was on line 7, so you clicked on that line, then clicked on the Breakpoint debug command button to set a breakpoint on that line. The source line decoration beside line 7 is now a B for breakpoint. See “breakpoint” on page 7-79 and “Source Line Decorations” on page 7-63.

NightView responds with:

```
local:2347 Breakpoint 1 set at fact.c:7
```

Finishing up - GUI

Now you want to run the program until it reaches the breakpoint. Click on the Resume button. See "Debug Command Buttons" on page 9-32.

Clicking on Resume told the program to start running. It ran until it hit the breakpoint that you had set on line 7. The source line decoration beside line 7 is now a B=.

NightView responds with:

```
local:2347: at Breakpoint 1, 0x100026fc in factorial(int
x = 1) at fact.c line 7
```

Note that your process ID and addresses will differ. Now look at the call stack. Click in the debug command area and issue the following command:

```
bt
```

You used the **bt** (**backtrace**) command to display the call stack. See "backtrace" on page 7-65. You saw all the following expected recursive calls in the debug message area. See "Debug Message Area" on page 9-28. Note that the output may scroll in the debug message area.

```
#0  0x100026fc  in factorial(int x = 1) at fact.c line 7
#1  0x1000271c  in factorial(int x = 2) at fact.c line 9
#2  0x1000271c  in factorial(int x = 3) at fact.c line 9
#3  0x1000271c  in factorial(int x = 4) at fact.c line 9
#4  0x1000271c  in factorial(int x = 5) at fact.c line 9
#5  0x1000271c  in factorial(int x = 6) at fact.c line 9
#6  0x1000271c  in factorial(int x = 7) at fact.c line 9
#7  0x10002784  in main(int argc = 2, unsigned char **
argv = 0x2ff7eaec) at fact.c line 26
```

Now look at the value of the variable x. Drag the mouse pointer over the variable x anywhere it appears in the source display. Click on the Print button. See "Debug Command Buttons" on page 9-32.

NightView showed that the value of x was equal to 1. You saw the following output in the debug message area.

```
$1: x = 1
```

Now finish running the program. Click on the Resume button. See "Debug Command Buttons" on page 9-32.

This allowed the process to run to completion. NightView showed the call to exit(0) in the debug source display and displayed the following message in the debug message area.

```
Process local:2347 is about to exit normally
```

NightView displays the following message in the dialogue I/O area. See "Dialogue I/O Area" on page 9-17.

```
factorial(7) == 5040
```

Exit from NightView by selecting the dialogue NightView menu or debug NightView menu. See “Dialogue NightView Menu” on page 9-16 or “Debug NightView Menu” on page 9-20. Click on NightView or use the N mnemonic. Click on the Exit (Quit NightView) menu item or use the X mnemonic.

NightView responds with a warning dialog box. The warning dialog box says:

```
Kill all processes being debugged?
```

Finally you click on the OK button to leave the debugger. The **`fact`** program had not fully exited, so NightView prompted, asking if the program should be killed. You responded by clicking OK, and the sample session ended.

3
Concepts

This section describes concepts you will need to understand in order to use the debugger effectively.

Many of the concepts described in this section are also defined in the glossary. The glossary is an alphabetical list of the concepts — the description here is organized hierarchically.

Debugging

The term *debugger* is actually a misnomer. A debugger does not remove bugs from your program. Instead, it is a tool to help you monitor and examine your program so that you can find the bugs and remove them yourself.

A debugger primarily lets you do two things:

1. start and stop the execution of your program; and,
2. examine and alter the contents of the program's memory.

There are many ways to do these things, so there are lots of debugger commands. Also, some of the commands control the debugger itself.

NightView is a symbolic debugger. That means that you can talk about your program using the same high-level language constructs that you use when you write programs. You can refer to variables, expressions and procedures as they appear in your program source. You can also refer to source files and line numbers within those files. For example, you can tell your program to stop at a particular line. In order to use the symbolic capabilities of the debugger, you must compile and link your program with options that tell the compiler and linker to save the symbolic information along with your program.

Sometimes, you want to be able to debug at a lower level, referring to machine language instructions and registers. NightView lets you do that, too.

Accessing Files

During the course of debugging, NightView will likely have to access a number of files: executable files for programs being debugged, source files for those programs, and possibly object and library files. Those files must all reside, or be accessible from, the system on which NightView is executing.

If you are debugging processes running on some other system, you will probably want to have some of that system's files mounted via NFS™ on the system running NightView. Furthermore, your debugging will probably go much easier if the pathnames to those files (especially the executables) are the same on both systems. This will allow NightView to find the executable files automatically most of the time. See "Finding Your Program" on page 3-8. If the pathnames of the executable files are different, you can use the **`translate-object-file`** command to tell how to translate the names. See "translate-object-file" on page 7-21.

Programs and Processes

It is necessary to distinguish between a *program* and a *process*. A *program* is something that you write, compile and link to form a program file. A *process* is an instance of execution of a program. There may be several processes running the same program.

Multiple Processes

The most typical use for NightView is debugging a single program running as a single process, but NightView can also be used to debug an *application* consisting of multiple processes, so the debugger has ways to describe multiple processes. If you come to a section of the manual that describes multiple processes, and you are only debugging one process, you can usually just ignore the parts about multiple processes.

You may inadvertently create multiple processes, even though you only want to debug one. This may happen if your program *forks*. For example, your program may call **`system`**. This call works by using the **`fork`** service to create another process, which then runs a shell. A process created this way is called a *child process*. Because NightView has the capability of debugging child processes, you are notified when this happens. If you don't want to debug the child process, then you should **`detach`** from it, which allows it to run without further interference from the debugger. See "detach" on page 7-32. If you know in advance that you don't want to debug any child processes, you can use the **`set-children`** command to specify this. See "set-children" on page 7-41.

If you use pipelines in the dialogue shell, or invoke shell scripts which call many other programs, you are likely to get multiple processes which you are not interested in debugging. (Dialogues are described in a later section, see "Dialogues" on page 3-4.) Again, if you don't want to debug those other processes, you should detach from them.

Another way to determine which processes are debugged is to use **debug** and **nodebug**, which let you describe which processes you want to debug by their program names. See "nodebug" on page 7-20.

Families

One of the handy things NightView lets you do is group processes together into *families*. You do this by giving the family a name and telling the debugger what processes are in

that family. For example, you might have several processes executing the same program, and you might want to set a breakpoint at the same source line in all of them. You could define a family containing all of the processes and then use that family name with the **`breakpoint`** command. See "family" on page 7-40.

Attaching

Sometimes you want to debug a process that is already running, rather than starting up a new process running the same program. You can do this with the **`attach`** command (see "attach" on page 7-32).

In order to attach to a process, you must know its process identifier (or PID). You can get a list of running processes and their PIDs by running the **`ps(1)`** program. You can use the **`shell`** command (see "shell" on page 7-112) to run **`ps(1)`**. If you want to attach to a process running on another machine, you may have to use the remote shell command (`/usr/ucb/rsh`) to run **`ps(1)`** on the right machine.

Once you have attached to a process, you can debug it in the same way you would debug a process started normally from a dialogue. An attached process is debugged using `/proc` (see "Debugger Mechanisms" on page 3-17).

For the security restrictions on **`attach`**, see "Attach Permissions" on page 3-35.

If the process to which you attach is stopped (<CONTROL Z> stops a foreground process in most shells), then the attach will not take effect until the process is continued from the shell.

Detaching

Detaching a process is the inverse of attaching one. When you detach a process it starts running independently of the debugger. Nothing it does will get the debugger's attention. Any children it forks will also be ignored by the debugger. You have to explicitly attach to the process again to make the debugger notice it.

Detaching from an exited or terminated process completely removes the process from the system. See "Exited and Terminated Processes" on page 3-16. Detaching from or killing a pseudo-process associated with a core file (see "Core Files" on page 3-4) is the only way to make that pseudo-process go away.

Detaching from a process causes NightView to forget all the eventpoint settings and other information it remembers about the process.

NightView typically uses some memory in the debugged process. If you detach and re-attach repeatedly, NightView will eventually be unable to find memory where it needs it in the process. See Appendix E [Implementation Overview] on page E-1. See also "set-patch-area-size" on page 7-50.

Core Files

A core file is a snapshot image of a process created by the system when the process aborts (typical reasons for creating a core file include referencing an address outside the memory allocated to the process, dividing by zero, floating-point exceptions, etc). NightView allows you to debug core files as well as processes (see "core-file" on page 7-34). Since a core file is not actually a running process, all you can do is look at it. None of the commands which require a running process will work on core files (for example, you cannot **`continue`** a core file and you cannot evaluate any expression containing a function call).

If a core file is from a process that used dynamic linking, the core file must be debugged on the same system where the process was running, otherwise information from the libraries may not match the core file.

Qualifiers

If you are not debugging multiple processes, you will probably never need to worry about command qualifiers, but for multiprocess debugging, they are essential. A qualifier is used to restrict a command so it operates only on specific processes. There is always a default qualifier in effect, but any command may be given an explicit qualifier.

Most qualified commands act as though the command was specified once for each process (for instance, the **`breakpoint`** command sets a separate breakpoint in each of the processes specified in its qualifier).

Some commands treat the qualifier in special ways, and other commands ignore the qualifier. Any special treatment is described in the section on each command.

Qualifiers are specified as a prefix on the command. The complete description may be found in "Command Syntax" on page 7-1 and "Qualifier Specifiers" on page 7-10.

Dialogues

Dialogues are one of the most important (and unique) concepts in NightView. Essentially, a *dialogue* is just an ordinary shell where you run commands as you would normally run them in the shell (in fact, you are running your normal shell), but in a dialogue, you have the opportunity to debug any or all of the programs you run in the dialogue shell. Most debuggers have special commands to tell the debugger which program to debug and what arguments to give it. In NightView, the way to debug a program is to run it within a dialogue shell. This means you can debug a program that is a member of a pipe, or is invoked by some other program, and you can run the program in the debugger using the exact same invocation you would normally use outside the debugger. For instance, if your programs run under the control of the Frequency-Based Scheduler, you could invoke **`rtutil`** or NightSim™ from your dialogue.

The environment variable `NIGHTVIEW_ENV` is set to `1` within a dialogue shell. This allows you to alter the behavior of programs and scripts running in the dialogue shell.

For example, you may wish to avoid running some programs in a shell initialization file when the shell is a dialogue shell.

Once the shell is started, you can change directory, set environment variables, or set **`ulimit(2)`** parameters just like a normal shell. Any processes you start in the dialogue will automatically be debugged, except for programs in the standard directories such as `/bin`. You may alter this default behavior using the **`debug`** and **`nodebug`** commands. See “debug” on page 7-20 and “nodebug” on page 7-20.

When you start a program in a dialogue shell, the debugger prints a message describing the new process that just started in the dialogue. The information printed includes the program name, the arguments it received on startup and the process identifier (PID). This new process is stopped immediately prior to executing any code. At this point you can decide what to do with the process (set breakpoints, etc.) and tell it to continue, or detach from it and let it run without being debugged.

At startup, NightView provides an initial dialogue named `local`. This initial dialogue shell inherits the current working directory and environment variables in existence at the time you started the debugger.

You may create additional dialogues at any time (see “login” on page 7-18). Multiple dialogues allow you to debug distributed systems of processes running on different computers. Each dialogue has a name. Unless you specify otherwise, the name of a dialogue is the host name of the system to which it is connected. You may use dialogue names in command qualifiers to tell NightView to which system you wish to talk, such as, when you want to run a command in a particular dialogue.

Dialogue I/O

You send input to a dialogue shell or to a program you are debugging in the dialogue by using the **`!`** command (see “!” on page 7-27) or the **`run`** command (see “run” on page 7-30). The qualifier on the command determines which dialogue receives the input data. In the graphical user interface, you can send input to a dialogue with the dialogue I/O area (see “Dialogue I/O Area” on page 9-17) for that dialogue.

Since each dialogue is a separate shell, the programs running in separate dialogues may generate output at any time. In the command-line interface, it would be confusing to have these print at any time. Instead, all the output generated by each dialogue shell and the programs running in it is logged by NightView. You can control this log using the **`set-show`** command (see “set-show” on page 7-28), and you can review the log with the **`show`** command (see “show” on page 7-29). In the graphical user interface, dialogue output goes to the dialogue I/O area for that dialogue.

Real-Time Debugging

By running NightView on a development system and starting a dialogue on a real-time system you are debugging, you can minimize the impact of the debugger on the real-time system. Most of the debugger runs on the development system, and only a NightView control program and the dialogue shell run on the real-time system. You can also control the CPU, memory, and other resource allocations of debugger processes to help minimize

the impact of the debugger on critical resources. See "Remote Dialogues" on page 3-6.

Monitorpoints provide a means of monitoring the value of variables in your program without stopping it. See "Monitorpoints" on page 3-10.

You may also want to use the debug agent mechanism in addition to `/proc`. See "Debugger Mechanisms" on page 3-17. The debug agent allows you to manipulate your process while it is running.

NightTrace™ is another tool you may find useful in debugging real-time programs. It allows you to gather performance information and record limited amounts of data with minimal overhead. NightView provides facilities for using NightTrace from within the debugger; see "Tracepoints" on page 3-11.

Remote Dialogues

A *remote dialogue* is a shell, controlled by NightView, running on a system other than the one on which NightView was initially invoked. We refer to the system where NightView was invoked as the "local system", while the system where the remote dialogue shell is running is referred to as the "target" or "remote system".

You may need to use a remote dialogue if:

- you need to debug programs running on multiple target systems simultaneously;
- your application uses most of the system's CPU or memory resources, leaving insufficient resources for NightView;
- the source files for your programs are not accessible on the target system;
- you do not wish to install all of NightView on the target system, perhaps to conserve disk space on the target;
- you need to reduce network traffic on the target system by eliminating NightView's GUI overhead;
- you need to reduce disk loading on the target system by eliminating NightView's reading of source and object files.

When you use a remote dialogue, the NightView user interface runs on the local system, while another process, named NightView.p, runs on the remote system to access and control the processes you are debugging. The following activities are performed on the local system in this case:

- all user interaction, including command input/output and window manipulation and updating;
- reading source and object files, including reading and interpreting debug information in your program;
- evaluation of expressions in commands such as **`print`** and **`x`**, except that retrieving data from a debugged process (such as variable values) is performed on the remote system.

The activities performed on the remote system are limited to storing and retrieving data to and from a debugged process, controlling execution of a debugged process, and supplying target-dependent information to the local system portion of NightView. Additionally, NightView sometimes runs the C compiler on the target system to generate code for eventpoints. See “Eventpoints” on page 3-8.

You may wish to control how the target system allocates resources to NightView.p and the dialogue shell, both to prevent them from interfering with your application and to ensure that they get sufficient resources to give adequate response in NightView. You can control the allocation of CPU and memory resources as well as the scheduling policy and priority through either the **`login`** command or the remote login dialog. See “login” on page 7-18. See “Remote Login Dialog Box” on page 9-45.

Note that the parameters you specify for the remote dialogue will be inherited by processes you execute within that dialogue shell. You may wish to use the **`run(1)`** shell command when you run your application in the dialogue shell.

There are some things you need to be aware of when you use a remote dialogue. Because source files and debug information are read on the local system, those files (or copies of them) need to be accessible on the local system. This is particularly true of the executable program file, because that is where the debug information resides. When a debugged process `exec`s a new program, NightView attempts to determine the location of the executable program file. See “Finding Your Program” on page 3-8. With a remote dialogue, NightView assumes that the pathname of the executable program file is the same (or locates identical files) on both systems. If this is not true, then NightView is not able to read debug information for that program until you specify the correct pathname with the **`symbol-file`** command or use object filename translations. See “symbol-file” on page 7-33. Also, see “translate-object-file” on page 7-21.

You may need to configure your local and remote systems to be able to use NightView remote dialogues. See Appendix A [System Resource Requirements] on page A-1 for more information about configuring systems for NightView.

Creating a new dialogue involves logging into a system (see “login” on page 7-18). You may login again as yourself, or as another user (subject to a password check). When a dialogue is created, it executes your login shell (or, more accurately, the login shell of the user whom you logged in as).

Logging in runs your `.profile` or other initialization file appropriate to your normal login shell. The environment variable `NIGHTVIEW_ENV` is set to the name of the local system (that is, the one you are logging in *from*) during the shell initialization. Your `.profile` should avoid reading from the standard input if `NIGHTVIEW_ENV` has a non-empty value.

ReadyToDebug

The program **`/usr/lib/NightView-release/ReadyToDebug`** is a special program used by NightView to synchronize with the dialogue shell (*release* is the NightView release level). You will probably see this program name echoed when a dialogue shell starts up. When NightView sees this program run, it knows that the shell is through with any initialization. NightView then considers any new processes that run in the shell to be candidates for debugging. This allows the initialization to take place

without debugging the programs that might run during that time.

Finding Your Program

When a program is started up from a dialogue, NightView is notified that a new program is executing, but there is currently no way for NightView to find out exactly what program is running.

NightView tries to guess the name of your program by looking at the arguments, the current working directory, and the PATH environment variable of the program. Usually, these correctly identify the program name, but not always. Then NightView can't tell what the program name is. Also, sometimes NightView may guess wrong.

NightView prints a message with the name of the program when the program starts up. If this name is wrong, then you will need to tell NightView the name of the program by using the **`exec-file`** command. See "exec-file" on page 7-35.

Most shells already do this correctly, so you will rarely need to worry about it. The problem sometimes occurs in programs that run other programs.

Controlling Your Program

NightView provides many ways to control the execution of a program you are debugging.

Eventpoints

An eventpoint is a generic term which includes breakpoints, patchpoints, monitorpoints, agentpoints, tracepoints, and watchpoints. All of these are different ways to debug or modify the behavior of your program, and all of them are assigned unique numbers by the debugger when you create them. These numbers are unique across all processes. For example, if you use a command qualifier to set a breakpoint in many processes at once, each breakpoint in each process is assigned a unique eventpoint number.

Agentpoints, breakpoints, monitorpoints, patchpoints and tracepoints are *inserted eventpoints*. They are implemented by inserting code into your process. A watchpoint is not an inserted eventpoint. This difference is mostly transparent to the user, but it does cause some minor differences in behavior. Those differences are noted where appropriate.

NightView allows you to set conditions on eventpoints, so the action associated with the eventpoint is taken only if the condition is satisfied. For inserted eventpoints, the condition is an arbitrary expression in the language of the routine where the eventpoint is set (in other words, if you set a conditional eventpoint in a Fortran subroutine, you would write the conditional expression in Fortran). NightView actually compiles the conditional expressions and patches them into the program, so evaluating the condition does not

require the debugger to take control. This means that setting a conditional eventpoint only adds the overhead required to evaluate the condition and the program will run at almost full speed until the condition is satisfied. See “condition” on page 7-89. However, a condition on a watchpoint is evaluated in the debugger. For watchpoints, the language of the expression is determined by your language setting. See “set-language” on page 7-44. Because watchpoint conditions are always evaluated in the global scope, if your language setting is `auto`, NightView evaluates the condition in the language of the main program.

You can also specify an ignore count for an eventpoint. This means you must execute past the eventpoint a certain number of times before it might be taken. The ignore count is checked prior to the condition, so if you have both ignore counts and conditions, the condition will not be checked until the ignore count is down to zero. See “ignore” on page 7-93. Like conditions, the code to implement ignore counts is patched into the program for inserted eventpoints, so the program will execute at nearly full speed until the ignore count reaches zero. An ignore count on a watchpoint is evaluated in the debugger.

There are several commands to manipulate eventpoints, but not every type of manipulation makes sense for every type of eventpoint. Deleting, disabling, enabling, and attaching ignore counts and conditions works for all types of eventpoints. See “Manipulating Eventpoints” on page 7-77.

Table 3-1. Eventpoint Summary

	Action	Code is inserted	May have commands
agentpoint	call the debug agent	**X**	
breakpoint	stop the process when the breakpoint is reached	**X**	**X**
monitorpoint	display the value of expressions in the monitorpoint window	**X**	**X**
patchpoint	execute an expression or modify the flow of the program	**X**	
tracepoint	record an event when execution reaches the tracepoint	**X**	
watchpoint	stop the process when the process reads or writes a variable in memory		**X**

Inserted eventpoints evaluate their conditions and ignore counts at full program speed, and may be manipulated while the process is running. Watchpoint conditions and ignore counts are evaluated in the debugger. Watchpoints may be enabled and disabled only while the process is stopped.

Breakpoints

A breakpoint is one of the most frequently used features of a debugger. You can set a breakpoint at any place in a program you are debugging, and when execution reaches that point, the program will stop. You may then use the debugger to examine the current values of variables, set additional breakpoints, etc. See "breakpoint" on page 7-79.

You may also specify an arbitrary set of debugger commands to execute each time a breakpoint is hit (if it is a conditional breakpoint, that means only when the condition is satisfied). See "commands" on page 7-89.

Monitorpoints

If you are debugging a real-time program, you may wish to monitor the value of one or more variables without interrupting the execution of your program. Monitorpoints allow you to do this. A monitorpoint is code inserted at a specified location by the debugger that will save the value of one or more expressions, which you specify. Because the expressions are evaluated by the program within a specific context, the expressions may reference local stack variables and machine registers and may call functions in your program. The saved values are then periodically displayed by NightView in a Monitor Window (see "Monitor Window" on page 3-27). You can set a monitorpoint using the **`monitorpoint`** command. See "monitorpoint" on page 7-84.

Note that the expressions you specify are evaluated *every time* execution passes the location of the monitorpoint (unless the monitorpoint is disabled or has a condition or an ignore count). However, NightView may not display every value saved by the monitorpoint. If the monitorpoint location is executed more frequently than NightView can update the Monitor Window, you will miss seeing some of the values evaluated by the monitorpoint.

Note that there may be some delay between the time that NightView reads the values saved by a monitorpoint and the time the values appear on your display. Therefore, values sampled by *different* monitorpoints cannot reliably be related in time. However, you may be sure that all the values sampled by a *single* monitorpoint were all evaluated at the same time.

Patchpoints

During the course of debugging, you may find a small error you would like to fix, but you would also like to continue debugging the program without recompiling and relinking. The **`patchpoint`** command (see "patchpoint" on page 7-80) allows you to patch in a change to the memory image of the process and continue running. (Note that it does *not* change the disk copy of the program file; recompiling and relinking is the only way to make a permanent change.)

A patchpoint can cause an expression (including function calls) to be evaluated, modify a variable, or cause the program to branch to a new location.

The **`load`** command (see "load" on page 7-75) provides the ability to make larger scale changes by loading in whole object files. This feature may be used to replace defective routines, or to load custom designed debugging routines that can do things like verify complex data structures, or search through linked lists.

Tracepoints

The manual for the NightTrace tool describes a library that may be used to generate trace records by calling trace routines in your program. If you didn't initially build a program with trace calls, (or you did, but decided later additional trace calls were necessary) the **`tracepoint`** command (see “tracepoint” on page 7-83) may be used to patch in tracepoints. The values traced may then be examined with the **`ntrace`** tool. For more information on NightTrace, see **`ntrace(1)`**.

Because the program runs at full speed through a tracepoint, you can use tracepoints in real-time applications where breakpoints are unacceptable.

One significant difference between a tracepoint and a monitorpoint is that values recorded by a tracepoint are all available for later analysis; values will not be "lost" because of delays in displaying, as they may with a monitorpoint. Another difference is that tracepoints provide a reliable means of relating values of expressions at different points of execution to the times those values were evaluated. Monitorpoints do not. However, monitorpoints have the advantage of displaying information as it is happening, whereas tracepoint data may be analyzed only after execution is finished.

Agentpoints

NightView allows you to control when the overhead of debugging occurs relative to your program's execution, if you modify the program slightly to insert calls to a special Debug Agent (see “Debug Agent” on page 3-17). This can be useful for some real-time programs. NightView can insert the necessary code in your process for you, using the **`agentpoint`** command. See “agentpoint” on page 7-87. These inserted calls are called agentpoints.

Watchpoints

A watchpoint stops your program when a particular area of memory is read or written. This is most useful in determining when a variable (or other program element) is being changed to a "bad" value during execution. You could set a watchpoint on the variable, and then the program would stop whenever the variable is modified. Watchpoints are set with the **`watchpoint`** command. See “watchpoint” on page 7-95.

Often you know what the bad value is. If so, you can set a condition on the watchpoint so that the program will stop only when the variable is changed to the bad value. The condition is evaluated *after* the instruction that triggers the watchpoint has completed. NightView provides two process-local convenience variables, `$is` and `$was`, that are useful in watchpoint conditional expressions. See “Convenience Variables” on page 3-31. `$is` contains the value of the variable (or other program element) after the instruction that triggers the watchpoint has completed. `$was` contains the value of the variable before the instruction began.

A watchpoint condition is evaluated relative to the global scope of your program. The language of the condition is controlled by your current language setting. If the setting is `auto`, then the condition is evaluated in the language of the main program.

Unlike other eventpoints, a watchpoint is not associated with a code location. A watchpoint is not an *inserted eventpoint*. See “Eventpoints” on page 3-8.

NightView supports only one watchpoint at a time for each process.

A watchpoint can be set only on a program element in memory, not in a register. You should be careful about setting a watchpoint on a variable on the stack, because the watchpoint probably will not be meaningful once the routine that contains the variable returns.

For watchpoint restart information, NightView always uses the same address that it calculates when you originally create the watchpoint. Note that the specific address may or may not be interesting in another run of your program, depending on exactly what your program does. For example, a variable on the heap may always be allocated in the same place each time your program runs, or it may be allocated at a different address depending on when it is allocated, what other allocations are done, timing of external events, etc. You may need to delete a watchpoint that was created by restarting and create a different watchpoint. See "Restarting a Program" on page 3-14.

When you have a watchpoint set, your process does not incur any performance penalty until it references the addresses being watched. When that happens, NightView gets control. The mechanism NightView uses for watchpoints always watches exactly 8 bytes on an 8-byte boundary. If the variable you are watching is not 8 bytes long and aligned on an 8-byte boundary, then NightView automatically resumes the process for accesses outside the variable, although there is still considerable overhead associated with stopping and resuming the process, so the process does not run at full speed in this case. If the variable extends outside the aligned 8-byte range, because it is not aligned or because it is bigger than 8 bytes, then only the smallest address bytes of the variable are watched.

Once any watchpoint has been set in a process, NightView must control calls to `getcontext` and `setcontext` for that process, which adds overhead to these calls. Note that returning from a signal handler implicitly calls `setcontext`.

Because watchpoints are not inserted eventpoints, the debugger evaluates any ignore count and condition, so the ignore count and condition are not evaluated at full program speed. See "Eventpoints" on page 3-8.

A watchpoint is not triggered if the variable is accessed by other processes through shared memory (unless they are also being debugged and have watchpoints set) or if the variable is accessed through I/O using direct memory access (DMA), such as a low-level `read` from disk. A watchpoint is not triggered by accesses during a system service call.

NightView has no good way to handle watchpoint traps from the `lwarx` and `stwcx` instructions. (These instructions are typically used in very tight loops to synchronize with other processes.) If NightView gets a watchpoint trap from one of these instructions, it stops the process and gives you an error message. If this happens, you should set a breakpoint nearby, disable the watchpoint (see "disable" on page 7-91), let the process run to the breakpoint, and then enable the watchpoint (see "enable" on page 7-92).

Signals

Usually, your process is stopped and the debugger gets control if the process receives a signal. Signals may be generated by error conditions (such as dividing by zero or trying to write to a write-protected location). Other signals may be generated under program control (the program can request the system to send it a SIGALRM periodically, or another program may explicitly send a signal with the **`kill(2)`** system service).

Several ways in which to handle a signal are described in the **handle** command (see "handle" on page 7-105).

In addition, you may use the debugger to explicitly send a signal to a process (see "signal" on page 7-104). This is useful if you need to test the signal handler code in a program (however, the debugger itself uses SIGTRAP, so it should not be used in any of your code).

If you specify `nostop`, `noprint`, and `pass` for a signal, then the system will deliver the signal to the process normally and bypass the debugger. This avoids any performance penalty to your program if it makes frequent use of signals.

Signals may cause somewhat different behavior when you are single-stepping your program (see "Controlling Execution" on page 7-97). If a signal occurs while you are single-stepping, NightView's reaction depends on whether you specified `stop` or `nostop` and `pass` or `nopass` in the **handle** command (see "handle" on page 7-105). The four possible combinations are explained below.

`nostop`, `pass`

> The single-step operation continues, but the signal will be passed to the program. If you have a signal handler in your program, it will be executed *without* single-stepping. When the handler finishes executing, single-stepping will be resumed until it is complete or another signal occurs.

`nostop`, `nopass`

> The signal has no effect (other than temporarily interrupting execution). The single-step operation continues until it is completed or another signal occurs.

`stop`, `pass`

> The single-step operation is terminated and the process is stopped. If you issue another single-step command or a **continue** command, or a **resume** command with no argument, the signal is passed on to the process when it resumes execution.

`stop`, `nopass`

> The single-step operation is terminated and the process is stopped. The signal is discarded.

Some signals can have additional information passed to the signal handler via **siginfo(5)**. However, NightView has no mechanism for the user to specify this information, so signals sent to the process using the **signal** or **resume** commands will have no associated **siginfo(5)** information.

If a process stops with a signal that has associated **siginfo(5)** information, that information is preserved by NightView whenever possible. If you specified `pass` for that signal and you continue execution using the **continue** command or the **resume** command with no argument, the **siginfo(5)** information will be delivered to the process along with the signal. However, no **siginfo(5)** information is ever delivered if you explicitly specify a signal number on the **signal** or **resume** commands.

Restarting a Program

Restarting execution of a program under NightView is different than in many other debuggers, because instead of being executed directly by the debugger, programs are executed from a dialogue shell, or by other programs. The typical way you restart a program is to invoke it again in the dialogue shell. See "run" on page 7-30.

When NightView recognizes that a program is being run again, it automatically applies the same eventpoints, and other information, to the new instance of the program. NightView considers two programs to be the same if they have the same full pathname.

This method of restarting programs was chosen because of NightView's multi-process nature. You may actually want to debug multiple copies of the same program, and in that case you may or may not want to have the same eventpoints set in each copy. However, if you are debugging just one instance of one program, you can easily restart its execution without having to manually duplicate your eventpoint settings.

Occasionally you may wish to run a program again and again without stopping when it `exec`s or when it exits. For instance, if a program sometimes dies with a signal, you could run it repeatedly until the signal occurs and then examine where it occurred. To avoid having the process stop when it `exec`s, put a **`resume`** command (see "resume" on page 7-98) inside an **`on program`** command (see "on program" on page 7-36), like this:

```
on program yourprogram do
    resume
end on program
```

The **`resume`** command will not actually take effect until after the process has been initialized, so **`on program`** and **`on restart`** commands that set eventpoints and otherwise modify the process work as expected. Note that the process does actually stop when it `exec`s, but the **`resume`** command tells it to start running again as soon as NightView is finished initializing it.

To avoid having the process stop when it exits, use the **`set-exit`** command. See "set-exit" on page 7-42. These two mechanisms, in combination, allow you to run a program repeatedly and only stop it if it hits a breakpoint or a watchpoint or gets a signal.

The following sections describe the details of how restarting works. Most users will not need to know these details. The normal automatic mechanism handles most situations.

Restart Mechanism

At certain times in the execution of a program, NightView takes a *checkpoint* on that program. A checkpoint saves information about the eventpoints, signal disposition, etc. This information is called the *restart information*. Each checkpoint replaces the previous restart information.

The restart information is stored as a sequence of commands associated with your program name via an **`on restart`** command. See "on restart" on page 7-38. The commands restore the eventpoints and other information in the new program.

Each time you execute a program, NightView checks to see if an **`on restart`** command matches your program. If one matches, NightView executes the sequence of

commands associated with your program.

NightView takes a checkpoint on a process when:

- It is about to exit, terminate with a signal, or be killed by NightView.
- It is about to `exec` a new program.
- You enter a **`checkpoint`** command. See “checkpoint” on page 7-39.

It is not possible to turn off checkpoints. However, you can control whether restart information is applied. See “set-restart” on page 7-49.

Note that if you have a program that has not yet taken a checkpoint and you start a new instance of that program, then no restart information is applied to the new instance because there is none for that program.

You can save restart information to a file. See “info on restart” on page 7-128. This allows you to save the information across debug sessions. Or, you can edit the file to change the restart information. In either case, you would then **`source`** the file to restore the restart information. See “source” on page 7-113.

Restart Information

This section describes the restart information saved during a checkpoint.

- Any memory reservations made with the **`mreserve`** command. See “mreserve” on page 7-43.
- Eventpoints, including any names, conditions, ignore counts and commands associated with each eventpoint. See “Eventpoints” on page 3-8.
- Directory search path. See “directory” on page 7-60.
- Child disposition. See “set-children” on page 7-41.
- Signal and exception disposition. See “handle” on page 7-105.
- Display list. See “display” on page 7-72.
- Symbol file. See “symbol-file” on page 7-33.
- Default language. See “set-language” on page 7-44.
- Whether or not the process will stop before exiting. See “set-exit” on page 7-42.
- The interest level threshold, the interest level for `inline`, `justlines`, and `nodebug`, and any explicit interest levels for subprograms. See “interest” on page 7-51.

Restart Macros

If an **`on restart`** command is created by a checkpoint, then in addition to commands to restore eventpoints and other program information, there are two macros: `restart_begin_hook`, at the beginning of the commands, and

`restart_end_hook` at the end of the commands. Both macros are called with the name of the program being restarted as an argument.

These macros let you customize restart processing. The initial definition of these macros is

```
define restart_begin_hook(program_name) apply on program
define restart_end_hook(program_name) echo
```

This means that **`on program`** commands will be applied before any restart processing, and nothing will be done afterwards. (`restart_end_hook` is defined as **`echo`** because there is no way to make an empty macro.)

You can define these macros to be anything you wish. See "Defining and Using Macros" on page 7-134. For example, you could define `restart_begin_hook` to be **`echo`** to disable the **`on program`** processing. See "on program" on page 7-36.

Exited and Terminated Processes

When a process terminates normally, it flushes its I/O buffers, closes any open files, then calls the exit service. By default, NightView automatically arranges for a process to stop when it calls the **`exit`** system service. (You may alter this behavior with the **`set-exit`** command. See "set-exit" on page 7-42.) When a process terminates abnormally, it receives a signal, which causes the process to stop and NightView to get control. Thus, you may always examine a program that is about to exit or terminate abnormally. The process will still exist, so you can examine memory and registers.

If you continue execution of a process in one of these states, the process will cease to exist and NightView will forget about all the eventpoints set in that process. The PID for that process will be removed from all families (see "Families" on page 3-2) in which it appears. Detaching from such a process has the same effect (see "Detaching" on page 3-3).

Process States

A process is normally in one of two states; it is either *running*, or it is *stopped*. A process is said to be stopped when it gets a signal (and it is being debugged) or it hits a breakpoint or watchpoint (meaning that the point of execution reached the breakpoint or the watchpoint was triggered, and all the conditions on the breakpoint or watchpoint were satisfied). When it is stopped, the debugger has control. The debugger may continue to execute commands attached to that breakpoint or watchpoint, but once the debugger initially gets control, the process is considered to be stopped. (This is not the same type of stop as job control in the C shell or the Korn shell.)

Some debugger commands require the process to be stopped. It is meaningful to examine or modify stack locations or variables only if the process is stopped. Monitorpoints and tracepoints provide ways to examine variables without stopping a process. See "Monitorpoints" on page 3-10. See "Tracepoints" on page 3-11. The first *inserted eventpoint* in a process must be set while the process is stopped. See "Eventpoints" on

page 3-8. A watchpoint may be enabled or disabled only when the process is stopped. See “Watchpoints” on page 3-11.

In addition to being stopped or running, a process may be exiting or terminated, or it may be a pseudo-process associated with a core file. A pseudo-process cannot be continued. Continuing an exiting or terminated process causes the process to cease existence.

Debugger Mechanisms

NightView has two mechanisms it uses to interact with and control your program: /proc and the debug agent. These are described in subsections below.

/proc

The primary debugger mechanism is called `/proc` (or *procfs*), which is a file system that allows one program (such as NightView) to control the execution of another program. NightView uses `/proc` whenever you start up a program in a Dialogue (see “Dialogues” on page 3-4) or attach to a running process (see “Attaching” on page 3-3).

The `/proc` mechanism provides for comprehensive control of a process, including control over what happens when your program is about to get a signal. `/proc` can read and write static variables while the process is executing, but it cannot read or modify stack variables or registers unless the process is stopped. See “Operations While the Process Is Executing” on page 3-18.

Debug Agent

Another mechanism NightView can use is called a *debug agent*. A debug agent is a code subsystem that executes as part of your process and communicates with NightView through shared memory. The debug agent contains a subroutine that, when called, performs an operation on behalf of NightView. When NightView needs to perform an operation using the debug agent, it sends a message to the agent and waits for the agent to reply.

The debug agent mechanism allows NightView to examine and control your program while it is running. Because you control where in your program the debug agent is called, it can be a less intrusive means of debugging your program. However, the debug agent, by itself, does not provide comprehensive control of your program; you cannot get control when your program gets a signal, for example.

The debug agent was originally created for operating systems that do not support `/proc`. The advantages of using the debug agent with `/proc` are much smaller because `/proc` gives you the ability to read and modify memory while the process is running. The only advantages of using the debug agent are: 1) the program has greater control of exactly when the overhead of performing debugger operations occurs, and 2) there is no restriction on how the first eventpoint must be set. See “Operations While the Process Is

Executing" on page 3-18. A watchpoint may not be set or deleted using the debug agent.

NightView allows you to use multiple mechanisms when debugging a single process, by allowing you to add a debug agent to your program while you are debugging it with `/proc`. This gives you the advantages of both methods: comprehensive control over your program, along with control over debugger overhead. See "Using /proc and the Debug Agent Together" on page 3-19.

You control where the call to the debug agent is placed in your application, by placing an agentpoint in your process. See "Using /proc and the Debug Agent Together" on page 3-19. To effectively use the debug agent, you will need to choose this location carefully; the guidelines that follow will help you do this.

First, the debug agent executes as part of your process, so it has some effect on the performance of your application. The debug agent is very fast and efficient, though, so the impact should be minimal. Nevertheless, we recommend you avoid placing the debug agent call in a time-critical location. See "Debug Agent Performance" on page F-1 for information about the performance of the debug agent.

Second, the debug agent call must occur fairly frequently, at least a few times a second, to ensure reasonable response time from NightView. Each debug agent call does at most one NightView operation (such as read or write a memory location), to keep the overhead per debug-agent call as small as possible. A given NightView command may require several such operations, each of which requires that the debug agent be called. If your application uses the Frequency-Based Scheduler, a good place to call the agent is usually right before the call to `fbswait`. You may include multiple calls to the debug agent in a program, if you wish.

Operations While the Process Is Executing

This section lists what you can do with either `/proc` or the debug agent while the process is executing (i.e., running).

- Examine and modify statically-allocated variables. This includes `static` and global variables in C, and `COMMON` variables and variables with the `SAVE` attribute in Fortran. It does not include variables allocated to registers or the stack.
- Examine and modify absolute memory locations. This includes accessing memory referenced by a pointer variable, if the pointer variable is accessible as noted above.
- Evaluate expressions involving the above items. See "Expression Evaluation" on page 3-20. Note that a function call is not allowed.

  For the purposes of establishing the scope and meaning of variable names, and also the language for the expression, NightView uses the location where the process was last stopped to determine the context of the expression (see "Context" on page 3-24). You can use the special forms NightView provides to change this context, if you want to access variables local to a procedure, for instance. See "Special Expression Syntax" on page 7-4. However, note that the forms that refer to specific stack frames are not

allowed while the process is running, because the state of the stack is indeterminate.

- Examine, modify, and disassemble executable code.
- Create, manipulate, and destroy *inserted eventpoints* (agentpoints, breakpoints, monitorpoints, patchpoints and tracepoints). See “Eventpoints” on page 3-8. These types of eventpoints may be enabled and disabled, have conditions added or removed, and have ignore counts modified. You may modify the commands attached to breakpoints, monitorpoints and watchpoints. You may also get information about any type of eventpoint. See “Manipulating Eventpoints” on page 7-77.

  Enabling or disabling watchpoints requires the process to be stopped. Any of the other operations may be performed on watchpoints while the process is executing. However, since, by default, watchpoints are enabled when created, and disabled when destroyed, you cannot ordinarily create or destroy a watchpoint while the process is executing. See “Watchpoints” on page 3-11.

  There are two rules about manipulating eventpoints while your process is running with `/proc`:

  - The *first inserted eventpoint* within a particular text region must be set while the process is stopped. A text region is either your program or the dynamic libraries it references.
  - The first *monitorpoint* must be set while the process is stopped, regardless of whether other eventpoints have been set in that region. See “Monitorpoints” on page 3-10.

  This is necessary because NightView needs to do special processing when the first eventpoint is created within a text region, or when the first monitorpoint is created. That special processing requires the process to be stopped.

  These restrictions do not apply to the debug agent, which handles the special processing in a different way.

While the process is executing, you may not use forms of commands that depend on knowing the program counter or the value of any machine register. See “Predefined Convenience Variables” on page 7-6.

Note that monitorpoints and tracepoints also provide ways of monitoring your program without stopping it. See “Real-Time Debugging” on page 3-5.

Using /proc and the Debug Agent Together

Using the debug agent together with `/proc` is easy. To debug a program this way, you first get control of it with `/proc`, by either running the program in a dialogue shell or using the **`attach`** command. See “Dialogues” on page 3-4, “Attaching” on page 3-3, and “attach” on page 7-32.

Once you have control of your process, you can use the **`agentpoint`** command to insert the call to the debug agent in your program. An *agentpoint* is a type of eventpoint (see

"Eventpoints" on page 3-8). You can create multiple agentpoints in your process; you might want to do this if you cannot find one single place that will be executed sufficiently frequently.

For a description of the operations you can do using the debug agent, see "Operations While the Process Is Executing" on page 3-18.

While your process is executing, NightView uses the debug agent for reading and writing memory. If the process stops, either because you ask NightView to stop it, or because of a breakpoint, watchpoint, or signal, NightView automatically switches to using `/proc` for all access to the process. When you resume execution again, NightView automatically switches back to using the debug agent.

Examining Your Program

If you specify running processes in the qualifier of a command which requires stopped processes, you get a warning message about each running process, but the command executes normally on any of the stopped processes in the qualifier.

Expression Evaluation

Because NightView is a symbolic debugger supporting multiple languages, you are allowed to evaluate expressions written in different languages, but this does not mean you have access to all the features of each language. (Specific language syntax is not described here; consult the reference manuals for the language for that information.)

One important point to note is that the debugger may not always precisely follow the language semantics when evaluating an expression. In particular, the results of a floating-point expression evaluated by the debugger may not be bit for bit identical to the results the same expression would give if it were compiled and executed in your program. See "Special Expression Syntax" on page 7-4.

A program written in multiple languages may define identical names for different global objects. NightView looks first for the name as defined in the language of the current context (see "Context" on page 3-24). If there is no current context, it uses the current language setting to determine which symbols to look at first (see "set-language" on page 7-44).

The debugger can evaluate arithmetic or logical expressions (essentially anything that may appear on the right hand side of an assignment). The debugger cannot declare new variables.

In general, the debugger cannot execute statements, it can only evaluate expressions. However, for Ada and Fortran, the concept of an expression is extended to assignment.

In some ways the debugger is more flexible than the compiler. The debugger usually allows you to evaluate expressions or assign new values to variables without the type checking done by the compiler. Unless the expression simply makes no sense, the debugger will evaluate it.

Ada Expressions

Remember that the debugger handles expressions (plus assignment and procedure calls), not executable statements. You must leave off the trailing semicolon for an Ada assignment or procedure call.

Most Ada expression forms are supported, but there are some restrictions and limitations, summarized in the list below.

- Data types

  All data types are supported, with a few exceptions:

  - Task types are not fully supported as a data type. They are treated simply as an address.
  - Access to subprogram is not supported.

- Type conversions are supported as defined for the Ada language, and using the same syntax as that of the language (i.e. *type_mark*`(`*expression*`)`), with certain exceptions and additions. As defined by the language, conversions involving numeric types convert the value of the expression, not the representation. For example, `float(1)` would return `1.0`. NightView allows conversions from a value of any type to any target type, not just those cases allowed by the Ada language. Note that NightView does *not* perform representation changes when converting to or from derived or convertible array types with differing representations. Conversions involving non-numeric types are performed by simply interpreting the left justified bit pattern of the value as the value of the target type with the corresponding left justified bit pattern. Note that, if the target type is smaller than the source value, the rightmost bits of the converted value are indeterminate.

- NightView treats user-defined character types (i.e., enumerations which have character literals as enumeration values) strictly as enumerations, not as a character type. The chief effect of this is that you cannot use string-literal notation (e.g., `"abc"`) to form arrays of these types. In NightView, string literals are always interpreted as arrays of the built-in type `character`.

- Aggregate values, such as `(a => 1, b => 2)`, are not supported. Other expressions that yield aggregate values are allowed.

- Subprogram calls

  A NightView expression can contain subprogram calls (either functions or procedures), provided that the arguments are either scalar types, statically-sized record types, or arrays. Note that this excludes subprograms with a formal argument that is an unconstrained record with discriminants, but unconstrained arrays are supported. Functions that return arrays or records are supported.

  Overloaded operators and functions are supported in NightView with help from the user to select the correct function. See “Overloading” on page 3-23.

- Attributes

  Subprograms that rename attributes are not supported.

The following attributes are not supported: `'callable`, `'count`, `'key`, `'lock`, `'shm_id`, `'terminated`, and `'unlock`.

The `'fore` and `'aft` attributes of fixed-point types may not give correct results.

Other attributes are supported in such commands as **`print`** and **`set`**, but they cannot be used in monitorpoint, patchpoint, or tracepoint expressions, nor in an eventpoint conditional expression.

One attribute, `'self`, is supported as a language addition in the debugger. When used on a tagged type object or access to a tagged type object, the `'self` attribute returns the same object with the type set to the actual type of the real object as determined from the run time type information provided by the compiler.

- The catenation operator, &, is not implemented.
- Logical operations (e.g., the `and` operator) on arrays are not supported.
- Relational operations that require ordering (e.g., `<`) are not supported for all arrays; they are supported only for arrays of character. Equality operations (`=` and `/=`) are supported for all arrays.

&*variable* may be used as a synonym for *variable*`'address`.

Any exceptions raised in a monitorpoint, a patchpoint, or a tracepoint, or in an eventpoint conditional expression are propagated to the program.

C Expressions

All C expressions are supported.

The debugger supports array slices in expressions using the following syntax:

array_name[*l*..*u*]

where *l* is the lower bound and *u* is the upper bound. The *array_name* may be any expression that denotes either an array object or a pointer. The type of an array slice is an array whose bounds are the values of *l* and *u*, respectively.

C++ Expressions

Most C++ expressions are supported, with a few exceptions noted below.

The debugger supports array slices in C++. See also "C Expressions" on page 3-22.

In function calls and assignments, the debugger copies an object by copying the bytes of the object. No copy constructor or user-defined assignment operator is called.

These C++ features are not supported:

- Exceptions.
- Templates.

Operator and function overloading is supported with additional input from the user used

to select the desired function. See “Overloading” on page 3-23.

A special case form of the `dynamic_cast<>` function is supported. You may use `dynamic_cast<>`, spelled exactly this way (with no type name given as a template argument inside the `<>`). This form of dynamic casting will cast an object or a pointer to the actual type of that object as determined by run time type information provided by the compiler.

Fortran Expressions

All Fortran expressions are supported.

Fortran subroutines are treated as if they were functions with no return value. Fortran assignments are supported except for Concurrent Fortran array assignments.

The debugger cannot execute statements of any kind (except assignments and procedure calls), including Fortran I/O statements.

Overloading

Overloading of functions, procedures, and operators is allowed in location specifiers and expressions in the Ada and C++ language modes. See “set-language” on page 7-44. Overloading means that more than one entity with the same name is visible at the same point in the program. NightView will call the appropriate routine if it has enough context to determine there is only one choice, otherwise you will need to provide NightView with additional information in the form of special syntax added to the expression or location specifier where the overloaded name is used.

This is typically a two step process. You run the command once and get an error which displays the possible choices. Then you run the command again with additional syntax to request the specific candidate number from that list.

The special syntax used to request candidates from the list is described in “Selecting Overloaded Entities” on page 7-2. Overloaded names are supported in language expressions (see “Expression Evaluation” on page 3-20) and location specifiers (see “Location Specifiers” on page 7-9), and the same syntax is used for both.

The **`set-overload`** command (see “set-overload” on page 7-54) may also be used to make NightView automatically generate overload candidate lists by turning on either of the two separate overload modes for routine names and language operators. This automates the first step of the two step process. The special syntax may be used to request overload candidate information for a single function or operator even when the corresponding overload mode is off.

If overloading is on, NightView interprets overloaded entities according to the current language. If overloading is off, NightView uses the built-in meaning of all operators, if possible, and interprets all function and procedure calls as referring to one function or procedure it arbitrarily picks from the list of candidates. If operator overloading is off and the built-in operator does not make sense in the context in which it is used, NightView gives an error.

If overloading is on, but a unique meaning for an overloaded operator or routine cannot

be determined, NightView gives an error that includes the list of the possible overload candidates. You may then run the command again, adding the syntax to select the correct candidate.

The numbers assigned to the choices are unique for the specific context (see "Context" on page 3-24) where the expression or location specifier appears. If, for example the 5th item in a list of choices refers to a particular instance of the overloaded function `funcname` when you are stopped at one point in your program, you may not assume the 5th item will refer to that same instance when you are stopped at a different location.

The one number you can rely on is 1 for overloaded operators. The built in language operator is always number 1, and any user or library defined operators have numbers greater than 1.

Program Counter

When a process is stopped, it has stopped at one specific place in the program, which is the address of the next instruction to be executed. This place is where the program counter points. Different machines have different sets of registers, but the program counter is always referred to as `$pc`.

If the currently selected frame is not the most recently called frame, then the `$cpc` register points to the instruction that made the call and the `$pc` register points to the place where execution will return after the call. In the most recently called frame, `$cpc` and `$pc` point to the same place.

Context

The location pointed to by `$cpc` implies a specific context for evaluating expressions. `$cpc` is located in some procedure (or routine, or function — the terms are used interchangeably throughout this document). This procedure was coded in some language (Ada, C, C++, Fortran, or assembler). By default, the language of the routine containing the `$cpc` is the language used to evaluate any expressions.

Another component of the context is the current stack frame (see "Current Frame" on page 3-25). It establishes which instance of a given local variable you are actually referring to in an expression. NightView provides special syntax (see "Special Expression Syntax" on page 7-4) for referencing variables in other contexts besides the current one.

Scope

Most languages have scoping rules, with local variables visible only in inner blocks and more widely visible variables in outer blocks. Often the same name is used for different variables in different scopes. Just as the `$cpc` is located in a particular routine, it is also located in a particular block of the routine. The variables that are directly visible to the debugger are determined by the language rules and current block nesting structure of the

program at that point.

When debugging, you may need to look at other variables which would normally not be visible by the strict language rules. NightView makes every effort to make any additional variables visible for use in expressions (as long as the names do not conflict). If you cannot reference a variable due to a naming conflict, NightView provides special syntax (see "Special Expression Syntax" on page 7-4) for referencing variables visible in other scopes.

Stack

When a process stops, it not only stops at a particular program counter, but it also has a current stack. The stack is used to hold local variables and return address information for each routine. As a routine calls another routine, new entries (called *frames*) are made on the stack. The stack can be examined to show the routines which were called to get to the current routine using the **`backtrace`** command (see "backtrace" on page 7-65).

The debugger assigns numbers to each frame. The most recent frame is always frame zero.

In a program with multiple threads or Ada tasks, each thread or task has its own stack. See "select-context" on page 7-110.

Frames corresponding to uninteresting subprograms are not numbered and they are not shown in a backtrace. See "Interesting Subprograms" on page 3-27.

Current Frame

When a process stops, the current frame is initially the stack frame associated with the most recently called routine (where `$cpc` points). This frame contains the local variables for that routine, and these variables may be referenced in expressions you evaluate. Each frame also contains the return address indicating the specific point in the older routine where the `$pc` will be located when the current frame returns.

You may wish to examine the variables in one of the routines that called the current routine. To do that, you may use the **`up`** command ("up" on page 7-109) or the **`frame`** command ("frame" on page 7-108) to change the current frame. As you move up the stack (towards older routines, or in the same direction a return will go), the new stack frame becomes the *current frame*. Any variables referenced are now evaluated in the context of this new frame and new `$cpc` indicated by the called frame.

NightView also provides special syntax in expressions as an alternative to using the **`up`** or **`frame`** commands. See "Special Expression Syntax" on page 7-4.

Registers

Each stack frame also contains locations where registers are saved while in one routine so they can be restored when returning to the calling routine. As the current frame is moved,

the debugger notices which registers will be saved and restored. If you look at registers using the **`info registers`** command, or examine local variables which are being kept in registers, you see the values as they will be restored when the process finally returns to that frame. Referencing a specific register using the predefined convenience variable also refers to the register relative to the current frame.

When examining a variable allocated to a register, you must be aware that the variable may exist in that register for only a short time. Therefore, the contents of the register may not accurately reflect the value of the variable. See "Optimization" on page 3-33 for more information.

Inline Subprograms

Ada and C++ programs can have inline subprograms. The code for these subprograms is expanded directly into the calling program rather than being called with a transfer of control. There is usually a time savings, sometimes at a cost in the size of the code.

NightView generally treats inline subprogram calls the same as non-inline calls. Although an inline call does not create a stack frame, NightView creates a frame for it to match the semantics of the language and to simplify the model of debugging. You can use the usual commands to move up and down the stack frames and view variables within each frame. See "Current Frame" on page 3-25.

You can use single step commands to step into inline subprograms, to step over them, or to finish them. See "step" on page 7-99, "next" on page 7-100, and "finish" on page 7-102.

NOTE

If you step to a source line, and the instructions corresponding to that line begin with an inline call, NightView positions you at the beginning of the inline subprogram, rather than on the line with the call.

If you set an eventpoint within an inline subprogram, NightView modifies each instance of the subprogram. If there are a lot of calls to the subprogram, this may take a long time. If execution is stopped in an inline subprogram and you set an eventpoint using the default location specifier (which corresponds to `$pc`), the location specifier refers only to *that particular instance* of the inline subprogram as opposed to all instances. See "Location Specifiers" on page 7-9.

You can set an interest level for individual inline subprograms. The interest level applies to all instances of an inline. You can also set an interest level to avoid seeing any inline subprograms. See "Interesting Subprograms" on page 3-27. This may be desirable depending on how your program uses inline subprograms.

You may not call an inline subprogram in an expression, unless the compiler has created an out-of-line instance of the subprogram. See "Expression Evaluation" on page 3-20.

Interesting Subprograms

NightView considers some subprograms to be *interesting* and the rest to be *uninteresting*. NightView avoids showing you uninteresting subprograms. Single-step commands do not normally stop in an uninteresting subprogram. See “step” on page 7-99. A stack walkback does not display frames corresponding to uninteresting subprograms. See “Stack” on page 3-25.

In general, subprograms compiled with debug information are usually interesting and the rest are usually uninteresting. NightView gives you control over which subprograms are considered interesting by using the **`interest`** command. See “interest” on page 7-51.

Each process has a current *interest level threshold*. The default threshold is `0`. NightView uses rules to decide on the interest level of a subprogram. If the interest level of the subprogram is greater than or equal to the interest level threshold, then the subprogram is considered to be interesting.

NightView uses these rules, in order, to determine the interest level for a subprogram:

1. The interest level may be specified for that subprogram with the **`interest`** command.
2. If the subprogram is an inline subprogram, the value of the `inline` interest level is compared to the interest level threshold. If the `inline` interest level is less than the interest level threshold, then the interest level for the subprogram is the minimum value. Otherwise, continue with the next rule.
3. The interest level may be recorded in the debug information for that subprogram by the compiler. Some compilers have a way of designating an interest level in the source.
4. If the subprogram has debug information, but no explicit interest level, the interest level is `0`.
5. If the subprogram has line number information, but no other debug information, the interest level is the value of the `justlines` interest level for that process.
6. If the subprogram has no debug information at all, the interest level is the value of the `nodebug` interest level for that process.

In some situations there may be no interesting subprograms on the stack. In that case, the most recently called subprogram is considered interesting.

You can make all subprograms interesting by setting the interest level threshold to the minimum value.

Monitor Window

The Monitor Window shows the values of expressions being monitored by monitorpoints (see “Monitorpoints” on page 3-10). When you set a monitorpoint (see “monitorpoint”

on page 7-84), the Monitor Window is created if it does not already exist, and the expressions associated with that monitorpoint are automatically displayed in the Monitor Window. The values in the window are updated approximately once a second to show the values computed the last time each monitorpoint was executed.

The **`mcontrol`** command (see "mcontrol" on page 7-86) controls the monitorpoint display. You can remove monitorpoint items from the display window (and add them back in later). You can change the rate at which the window updates take place, and you can stop updates completely, then start them again later. You can also turn the Monitor Window off to remove it from your screen, then restore it later.

Note that interrupting the debugger implicitly causes the Monitor Window to stop updating. See "Interrupting the Debugger" on page 3-30.

The Monitor Window is not available in the command-line interface of the debugger. You must use either the simple full-screen interface (see Chapter 8 [Simple Full-Screen Interface] on page 8-1) or the graphical user interface (see Chapter 9 [Graphical User Interface] on page 9-1) in order to take advantage of monitorpoints.

The monitored items are displayed in the Monitor Window using built-in information about the precision of the data type to decide how many columns to use for the value. You have some control over this by using the format codes on the print command.

You also have some control over the layout of the items in the window. New items are added across a line, from left to right, until there is not enough space remaining on the line to add the current item. Then a new line is started. If you remove some items (by using **`mcontrol`** `nodisplay` or by removing the monitorpoints), the remaining items are shifted left and up to pack the display. If you then add the items back, they are added at the end of the display (*not* in their original positions).

By default, each item is displayed with an identification string, a *stale data indicator*, then the value itself laid out left to right. The stale data indicator can be turned on and off via **`mcontrol`**. There are 3 possible states that this indicator can denote:

Updated

> The monitorpoint location was executed and values were saved since the last time NightView updated the display. Note that the location may have been executed many times in between successive display updates. The displayed value represents the value as it existed the last time the monitorpoint location was executed.

Not executed

> Execution has not reached the monitorpoint location since the last time NightView updated the display. This may happen if that location is executed infrequently, if the process gets suspended for some reason, or if the process is stopped by a signal or breakpoint. The displayed value still represents the value as it existed the last time the monitorpoint location was executed.

Executed but not sampled

> Execution reached the monitorpoint location, but no values were saved because of an ignore count or unsatisfied condition. In this case, the displayed value is not necessarily the same as the value of the expression the last time the monitorpoint location was executed.

The actual form of the stale data indicator depends on the interface being used. See "Monitor Window - Simple Full-Screen" on page 8-2. See "Monitor Window - GUI" on page 9-48.

Errors

NightView error messages always have this format:

severity: *text* [*error-message-id*]

The *severity* can be one of:

`Caution`

Usually just an informational message. It is not serious.

`Warning`

A little more serious, but NightView tries to finish the current command as you requested.

`Error`

A serious error. This level of error terminates the current command. It also terminates a command stream. See "Command Streams" on page 3-29.

`Abort`

So serious that NightView cannot continue running. This does not usually indicate that you have done something wrong; either there is a system problem or there is a bug in NightView.

The *text* is a brief explanation of the problem.

The *error-message-id* is a section name you can use with the **`help`** command to find out more about the error and possibly how to fix it. An *error-message-id* begins with `E-`.

NOTE

Some libraries used by NightView, such as the X Window System™, issue their own error messages in certain circumstances. These error messages *do not* follow the format described above. You can recognize these messages because they do not have the [*error-message-id*] appended to the message.

Command Streams

A command stream is a set of commands that the debugger executes sequentially. There

are three kinds of command streams:

- Interactive command streams. These are commands entered directly by the user.
- A file of commands being read by the **`source`** command is also a command stream. Execution of the **`source`** command suspends execution of the command stream it appears in and creates a new one that endures until the file is exhausted.
- Event-driven command streams. For example, commands attached to a breakpoint are an event-driven command stream. Each instance of hitting a breakpoint creates a new command stream; the stream terminates when the commands attached to the breakpoint are finished. These non-interactive command streams always operate with safety level set to `unsafe` (see "set-safety" on page 7-49).

The debugger may interleave the execution of two or more command streams. For instance, it may execute some of the commands attached to one breakpoint, then execute some of the commands attached to a different breakpoint (on behalf of a different process), then execute more of the commands attached to the first breakpoint.

The debugger stops executing a command stream if it encounters a serious error (such as an unknown command, or a badly formed command). A less severe error (such as a warning about a process not being stopped) simply generates an error message, but the debugger continues to execute the remaining commands. If a serious error terminates a command stream, and that command stream was created by another command stream, then the older command stream is also terminated. This goes on until the interactive command stream is reached. The interactive command stream is not terminated.

Interrupting the Debugger

The shell interrupt character (normally <CONTROL C>) does not terminate NightView. Instead, it terminates whatever command is currently executing, if any. You may wish to use it if you accidentally ask NightView to print a large quantity of information you don't want. To type <CONTROL C>, press the `c` key while holding down the control key.

In the graphical user interface, you can interrupt the debugger by clicking the Interrupt button in any of the major windows. See Chapter 9 [Graphical User Interface] on page 9-1. See "Debug Interrupt Button" on page 9-34.

If you interrupt the debugger, all command streams except the standard input stream are terminated. The standard input stream is interrupted, but not terminated, so it will prompt for the next command immediately.

Furthermore, any output from debugged processes is temporarily halted (it is still buffered, but not displayed) until after you enter the next debugger command. This gives you a chance to type a command without interference from the debugger or the debugged processes. See "Dialogue I/O" on page 3-5 for more information about controlling the output from debugged processes.

Interrupting the debugger stops the Monitor Window from updating. See "Monitor

Window” on page 3-27.

Macros

A *macro* is a named set of text, possibly with arguments, that can be substituted later in any NightView command. When you define a macro, you specify its name, the names of the formal arguments, and the text to be substituted. The text to be substituted is called the *body* of the macro.

When you reference the macro in a NightView command, you again specify its name, along with the actual arguments. Actual arguments are the text you want substituted for the references to the formal arguments in the macro body. See “Defining and Using Macros” on page 7-134 for details on how to define and reference macros.

Macro expansion, the process of replacing the reference to a macro with its body, is simply a textual substitution. Very little analysis is performed on the substituted text, so macros can be a very powerful facility. Furthermore, a macro reference is expanded only when it is needed.

Macros provide a way for you to extend the set of NightView commands. They also provide a way to define shortcuts for things frequently used in commands or expressions.

Convenience Variables

NightView provides an unlimited number of convenience variables. These are variables you can assign values and reference in expressions, but they are managed by the debugger, not stored in your program. You don't have to declare these variables, just assign to them. They remember the data type and value last assigned to them.

There are two kinds of convenience variables — global and process local. Variables are global by default, but by using the **`set-local`** command (“set-local” on page 7-50) you can make a variable local to a process. Once you declare a variable name process local, each process maintains a separate copy of that convenience variable (a variable cannot be local in one process, but shared among all other processes). It is possible to imagine other types of scoping for convenience variables (such as breakpoint local or dialogue local), but process local and global are the only kinds currently implemented.

Because conditions on *inserted eventpoints* and the expressions associated with monitor-points, patchpoints, and tracepoints are compiled code executed in the process being debugged, references to convenience variables in these expressions always treat the convenience variable as a constant, using the value the variable had at the time the expression was defined. On the other hand, the *commands* associated with a breakpoint or watchpoint, and conditions attached to *watchpoints*, are always executed by the debugger, so a convenience variable referenced in a command gets the value at the time the command or condition is evaluated.

Logging

Each dialogue retains a buffer showing the output generated by the programs run in that dialogue shell. This output may also be logged to a file (see "set-show" on page 7-28).

In addition to the output log for each dialogue, you may log the commands you type, or the entire debug session (see "set-log" on page 7-44).

Value History

NightView keeps the results of the **`print`** command (see "print" on page 7-66) on a value history list. There is only one list for all the processes, and all printed values go on this list regardless of the process. You can review this history (see "info history" on page 7-124), or use previous history values in new expressions (see "Special Expression Syntax" on page 7-4).

Command History

NightView keeps a record of the commands you enter during a debugging session. There are mechanisms in the simple full-screen interface and in the graphical user interface to retrieve any of these commands, edit them, and re-enter them if desired. See "Editing Commands in the Simple Full-Screen Interface" on page 8-2. See "GUI Command History" on page 9-12.

NightView does not add a command to the command history if it is the same as the previous command. Empty lines are never added. Commands are added only from interactive command streams. See "Command Streams" on page 3-29.

Initialization Files

When the debugger starts up, it looks for a file named `.NightViewrc` in the current working directory. If it can not find one there, it looks for `$home/.NightViewrc`. The file, if found, is then automatically executed as though it appeared as an argument to the **`source`** command (see "source" on page 7-113).

You can specify other initialization files, and you may disable the automatic execution of the default initialization files, using options on the NightView command line. See Chapter 6 [Invoking NightView] on page 6-1.

Optimization

The problems of debugging optimized code are described in *Compilation Systems Volume 2 (Concepts)*.

These are the most common problems, but there are others:

- Machine language code may be moved around so that it does not correspond line for line to the source code in your program.
- Variables may not have the values you expect. The most common reason for this is that the value of the variable is not needed at the current location in your program and the register storing the value of the variable has been reused for another value.

In general, you must be alert to the possibility that the compiler has changed things in your program.

Concurrent compilers generate debugging information at high optimization levels because it is more useful than to have nothing; however, the debug information is often inadequate to describe an optimized program. (Future compilers may generate more accurate debug information.) So, be careful and consult the appropriate manual for details.

Debugging Ada Programs

Ada programs employ several concepts that are different from C, C++ and Fortran programs. NightView provides methods to assist in debugging programs that utilize these concepts.

Packages

Ada packages come in two parts: the specification, which gives the visible interface, and the body, which contains the details. NightView knows what source file to display depending on the execution context. For the Ada user, what is displayed is the body. If the unit specification is of interest the **`list`** command with the `'specification` modifier on the unit name may be used. (The modifier may be abbreviated.) See “list” on page 7-58.

An Ada unit name may be used to specify a location for those NightView commands that need a location specifier. See “Location Specifiers” on page 7-9. For example, locations are required for commands that manage eventpoints and the **`list`** command. All Ada unit names recorded in the debug table may be listed with the **`info functions`** command.

With Ada programs, declarations are elaborated in linear order. The elaboration of a declaration brings the item into existence, then evaluates and assigns any initial value to it. Elaboration occurs before any statements are executed. If the program has just started,

you can step into the elaboration code of library-level units with the **step** command. See "step" on page 7-99.

Exception Handling

Ada exception handling provides a method to catch and handle program errors. Each unit may have exception handlers. Exceptions which occur in a unit without appropriate handling code are propagated to the invoking unit. The unwinding process may be complex, therefore NightView provides several mechanisms to assist in debugging. The **handle** /exception command specifies whether to stop execution and notify the user that an exception has occurred. See "handle" on page 7-105.

Multithreaded Programs

NightView gives you facilities for debugging threads, Ada tasks, and Lightweight Processes. A *Lightweight Process* (LWP) is a distinct thread of control managed by the operating system. Ada tasks are serviced by LWPs, as are threads created by the threads library. See "Programming with the Threads Library" in the *PowerMAX OS Programming Guide*.

When a process containing multiple Ada tasks, threads, or LWPs stops, the operating system will choose one LWP to represent the process. This is the execution state that NightView will present to you by default. Whatever task or thread was being serviced by that LWP is the task or thread you will be viewing. To see other tasks, threads, or LWPs, use the **select-context** command (see "select-context" on page 7-110). This command allows you to select the Ada task, thread, or LWP whose context you wish to view.

The **select-context** command allows you to view the context of an Ada task or thread whether or not it is currently being serviced by an LWP. If the task or thread is currently being executed by an LWP, the **select-context** command automatically selects the context of that LWP.

It is important to note that NightView does **not** allow you to control the execution of a task, thread, or LWP independently of the others in that process. When you resume execution (see "resume" on page 7-98), all LWPs are allowed to execute, and they may service any of the threads or Ada tasks that are available to run. If you issue a single-step command (see "step" on page 7-99), the selected task, thread, or LWP will be stepped according to the command, but the other LWPs may also execute one or more instructions — they are not restricted to stepping the current line or instruction.

Each time your process stops, NightView automatically sets the current context to the context of the LWP that caused the process to stop. You may then use the **select-context** command to change the context.

Note that an LWP attached to a user-level interrupt cannot be stopped and continues to run when the other LWPs are stopped. See "User-Level Interrupts" on page 3-37.

Using NightView with Other Tools

NightView normally communicates with other programs via KoalaTalk. For example, other tools can start a debug session for a program, using NightView as a debug server.

This functionality is available only in the graphical user interface.

If you want to disable this mode, use **`-noktalk`** (see Chapter 6 [Invoking NightView] on page 6-1 or set the `useKoalaTalk` resource to `False`. (See Appendix D [GUI Customization] on page D-1.)

Limitations and Warnings

Setuid Programs

Setuid and setgid programs can be run in a dialogue shell. If you are the superuser or the owner of the setuid program, you may also debug the program. Otherwise, NightView issues a warning message telling you that it has automatically detached from the process and the program runs without being debugged. In this case, you also cannot debug any child processes of such a program.

One or more privileges may be associated with a program. These behave in a manner similar to setuid programs. If you run NightView with a privilege set that includes the privileges associated with the program you are attempting to debug, NightView is able to debug it. Otherwise, NightView automatically detaches from the process and the program runs without being debugged.

Note that programs run using the **`shell`** command (see “shell” on page 7-112) are not controlled by the debugger and so may run setuid.

Attach Permissions

You are only allowed to attach to processes running as the same user as the dialogue specified on the **`attach`** command. More precisely, the dialogue's effective UID must be the same as the real and saved UID of the process you want to attach, and the dialogue's effective GID must be the same as the real and saved GID of the process you want to attach.

An exception to the above rule is made for the superuser or users with `P_DACREAD` and `P_DACWRITE` privileges.. These users are allowed to attach to any process.

Frequency-Based Scheduler

When a process running under control of the Frequency-Based Scheduler (FBS) hits a breakpoint or watchpoint, or receives a signal that is handled by the debugger, the FBS stops running. This means that other processes under control of the same FBS will no longer be scheduled. Any other processes that are currently running will continue to run, but once they do an **`fbswait(2)`** call, they will not start running again until the FBS is restarted (it is as if the clock running the scheduler was stopped).

If you continue the stopped process, it will resume running, but once it executes an **`fbswait(2)`** call, it will also go to sleep and not wake up until the scheduler is restarted.

It is your responsibility to start the scheduler running again. This can be done via the **`resume`** command of the **`rtcp(1)`** program (perhaps using NightView's **`shell`** command), from the **`rtutil(1)`** program, or by clicking `Resume` in NightSim.

NightTrace Monitor

The **`tracepoint`** command (see "tracepoint" on page 7-83) can be used to trace variables in a process. Tracing only works if the **`ntraceud(1)`** monitor program has been started prior to adding tracepoints to the process. It is the responsibility of the user to make sure that the monitor is started (it may be started from within NightView using the **`shell`** command, see "shell" on page 7-112).

Memory Mapped I/O

Special purpose programs often attach to regions of memory mapped to I/O space. This memory is sometimes very sensitive to the size of reads and writes (often requiring an 8-bit or 16-bit reference). With the `/proc` or debug agent mechanisms, the debugger may access memory using 8-bit, 16-bit, or 32-bit references. See "/proc" on page 3-17. See also "Debug Agent" on page 3-17. This means you should probably avoid referencing I/O mapped memory unless the size of access does not matter.

Be especially careful of printing pointers to strings (e.g., variables declared to be (char *) in C or C++), because the debugger automatically dereferences these variables to print the referenced string.

Note that accesses made by tracepoints, monitorpoints, and patchpoints will be made according to the natural data type of the variable accessed, so those accesses should normally work correctly.

Blocking Interrupts

If you are debugging a program containing sections of code that raise IPL and block interrupts, you can easily get a CPU hung or crash the system by attempting to single step through this code (or by hitting a breakpoint or watchpoint in a section of code which

executes with IPL raised). In particular, the trace library routines do this, so do not try to single step through them.

User-Level Interrupts

Debugging a process that attaches to a user-level interrupt requires special care. There are certain restrictions you must obey and certain actions you must take to ensure correct operation. Note that this refers to user-level code attached directly to a hardware interrupt, not an ordinary signal handler, which requires no special treatment to debug.

You must never set a breakpoint or an agentpoint in any code that might be executed by the interrupt routine. If a user-level interrupt routine hits a NightView breakpoint, it will almost certainly crash the system. You *may*, however, set monitorpoints, patchpoints, and tracepoints, but be certain that none of the expressions associated with these eventpoints perform any actions not allowed by user-level interrupt code. See "Eventpoints" on page 3-8. Note that you can set a breakpoint or agentpoint in the process as long as you ensure they are not executed while servicing a user-level interrupt. You may set watchpoints in a process with user-level interrupts, however the watchpoint will not be in effect within the user-level interrupt routine.

You may attach to a process that has attached to user-level interrupts only if there is at least one Lightweight Process that is not attached to an interrupt. See "Multithreaded Programs" on page 3-34.

If you set an eventpoint in code that will be executed while servicing a user-level interrupt, you must make sure that all memory referenced by the eventpoint is locked in physical memory. NightView allocates memory regions where it places the code and data for eventpoints, so those regions must be locked in memory as well. You may either call the **`mlockall(3C)`** service, specifying `MCL_CURRENT`, **after** you have set all the eventpoints that will be executed by user interrupt code, or you may call **`mlockall(3C)`** and specify `MCL_FUTURE`.

If your process has attached an LWP to a user-level interrupt but also has other LWPs not attached to an interrupt, then the non-interrupt LWPs can be stopped by NightView, either using the **`stop`** command (see "stop" on page 7-103), by hitting a breakpoint or watchpoint (see "Breakpoints" on page 3-10 and "Watchpoints" on page 3-11), or by receiving a signal (see "Signals" on page 3-12). NightView indicates that the process has stopped, but the LWPs serving user-level interrupts continue to run and service interrupts. Only the LWPs not attached to an interrupt are stopped.

If you use the **`select-context`** command (see "select-context" on page 7-110) to examine the state of an LWP attached to an interrupt, the context will not be consistent. The registers will probably reflect the values they had when the LWP called the **`ienable(2)`** service. PowerMAX OS does not allow you to stop an LWP attached to a user-level interrupt.

Debugging with Shared Libraries

NightView provides the ability to debug programs that reference shared libraries, but there are a few things you need to know to use this effectively. This section describes

how NightView interacts with shared libraries.

Shared libraries are a mechanism that allows many programs to share libraries of common code without duplicating that code in each executable file. The executable files for those programs contain the names of the shared-library files referenced by that program. These references must be *resolved* before the program can reference data or functions in the libraries. When the program first starts executing, a routine called the *dynamic linker* gets control and resolves references to shared libraries.

However, NightView gets control of a process **before** the dynamic linker executes. This is useful for NightView, but not very useful for you the user, because until the dynamic linker runs, you cannot reference any of the data or functions in the shared libraries. For instance, you could not set a breakpoint in a function residing in a shared library.

Therefore, when NightView detects that the process references shared libraries, it lets the dynamic linker execute before giving you control of the process. This allows you to debug the entire program, without needing to know which parts reside in which shared library.

One consequence of this action, however, concerns signals. If your process should receive a signal while the dynamic linker is running, NightView will detect it and give you an error message. You will not be able to reference the shared-library parts of your program, and most likely the process will not be able to continue executing properly. One source of such a signal is the dynamic linker itself. If it cannot find one or more of the shared-library files referenced by the program, it will abort the process with a signal.

Some programs require more flexibility in their use of shared libraries. These programs call the **`dlopen(3X)`** service to load a shared library when it is needed. Because this happens after the program has initialized, NightView is unaware that a new shared library has been brought into the program's address space.

However, it is easy to make NightView aware of any dynamically loaded libraries at any time. Once your program has loaded a library or libraries using `dlopen`, you can use the **`exec-file`** command to force NightView to reexamine the list of shared libraries referenced by the program. See "exec-file" on page 7-35. After your program has called `dlopen`, enter the following command:

`exec-file` *program-name*

where *program-name* is the name of the program you are running (the one that calls `dlopen`). NightView updates its database of shared libraries, and you can then reference data and procedures in the dynamically loaded libraries.

You can issue this **`exec-file`** command as often as you wish. If your program loads several libraries at various points during its execution, you may want to issue the **`exec-file`** command several times.

4
Tutorial

This is the tutorial for the command-line version of NightView. NightView's command-line interface runs on all terminals. For more information about the command-line interface, see Chapter 7 [Command-Line Interface] on page 7-1. You may also be interested in the graphical-user-interface (GUI) version of this chapter in Chapter 5 [Tutorial - GUI] on page 5-1. There is a much shorter tutorial in Chapter 1 [A Quick Start] on page 1-1.

About the Tutorial

This tutorial shows only the most common debugger commands and features. It expects you to know the basics about processes and signals, but you do not need to know about NightView and debugging concepts.

The supplied tutorial program spawns a child process. The parent writes a message to stdout, sleeps, sends signal SIGUSR1 to the child, and loops. The child writes a message to stdout when it receives the signal.

Become familiar with the tutorial's source code; see Chapter G [Tutorial Files] on page G-1 or the files under the `/usr/lib/NightView/Tutorial` directory. The source files are:

C	Fortran	Ada	
`msg.h`	`msg.i`	`-`	Defines constants
`main.c`	`main.f`	`main.a`	Forks a child and calls other routines
`parent.c`	`parent.f`	`parent.a`	Sends signals to the child
`child.c`	`child.f`	`child.a`	Receives signals from the parent

This tutorial takes at least two hours to do. Each section must be performed in order.

Exercises in this tutorial tell you to do things and ask you questions. Make the most of this tutorial and the manual; follow the steps below:

1. Look up the information.
2. Try to figure out the answer on your own.

3. Apply the provided solution. (**Warning:** Type the solutions exactly as they appear or your results may differ from those provided in later steps of the tutorial. Do not type anything until you see the words "you should enter" in the tutorial.)

You do not need to follow cross references in this tutorial **unless** you are explicitly told to read them.

This tutorial often displays process IDs. Your process IDs will probably differ from those shown. Also, the tutorial displays hexadecimal addresses. The addresses for your program may differ from those shown. Additionally, the line breaks in your output may differ from those shown because the lengths of displayed data items may vary.

The code produced when you create your program may vary slightly from the programs used to prepare this tutorial. In particular, the line shown as the return address from a subprogram may be different from what is shown here.

Some messages might not appear, or additional messages might appear, depending on your environment.

Creating a Program

NightView is mainly used with executables that contain debug information. To create such a program, compile source files with a particular option, and link edit them.

Exercise:

Create a directory named `nview` where you can create files for this tutorial, and move into that directory.

Solution:

You should enter:

```
$ mkdir nview
$ cd nview
```

Note: do not enter the `$`. It is part of the shell prompt.

Exercise:

Use the manual to find out what compiler option is necessary for debugging. (Hint: use the index.)

Solution:

From the index, **Compiling** has this information. The **`-g`** compiler option puts debug information into an executable.

Exercise:

Decide what language program you want to debug. Do *not* copy the source files from the `/usr/lib/NightView/Tutorial` directory, just compile and link these files. Make the `msg` program contain debug information. What command or commands did you use?

Solution:

For C, you should enter:

```
$ cc  -g  -o msg /usr/lib/NightView/Tutorial/*.c
```

For Fortran, you should enter:

```
$ f77  -g  -o msg /usr/lib/NightView/Tutorial/*.f
```

For MAXAda, you should enter:

```
$ /usr/ada/bin/a.mkenv  -g
$ /usr/ada/bin/a.path  -I obsolescent
$ /usr/ada/bin/a.intro /usr/lib/NightView/Tutorial/*.a
$ /usr/ada/bin/a.partition  -create active  -main main
msg
$ /usr/ada/bin/a.build  -v msg
$ /usr/ada/bin/a.rmenv .
```

For HAPSE Ada, you should enter:

```
$ /usr/hapse/bin/a.mklib  -g  -f
$ /usr/hapse/bin/a.make  -v  -lib .  -o msg main \
     -f /usr/lib/NightView/Tutorial/*.a
$ /usr/hapse/bin/a.rmlib
```

You should now have a `msg` program with debug information in your `nview` directory. Note that for this tutorial, the source files should *not* be in this directory.

Starting NightView

You are ready to start up NightView *without* the graphical-user interface.

Exercise:

Read how to invoke the command-line interface of NightView. (You can find this information in the manual, on the man page, or by invoking **`nview`** with the **`-help`** option.) Start up the command-line interface of NightView.

Solution:

In the index, **Starting the debugger**, **Invoking the debugger**, and **`nview`, invoking** have this information. See Chapter 6 [Invoking NightView] on page 6-1. You should enter one of:

```
$ nview -nogui
$ nview -nog
```

Note that in this tutorial `msg` does not appear on the **`nview`** invocation line.

NightView responds with:

```
$ nview -nogui
NightView debugger - Version 5.1, linked Mon Jan  17
13:57:27 EST 2000
Copyright (C) 2000, Concurrent Computer Corporation

In case of confusion, type "help"

Use the 'run' command to run your program under the
debugger
(local)
```

These messages include NightView version information, copyright information, help information, and the command prompt, `(local)`. Your version number and date may differ. You will use online help later in this tutorial.

A dialogue contains a shell where you run shell commands and debug running programs. Each dialogue has a name; the default dialogue is `local`. The default qualifier is also `local`. The default command prompt is the qualifier in parentheses. For information about dialogues, see "Dialogues" on page 3-4. For information about qualifiers, see "Qualifiers" on page 3-4. For information about prompts, see "Command Syntax" on page 7-1.

In the command-line interface, NightView sometimes displays the command prompt before it completes its output display. You think NightView may have some undisplayed output.

Exercise:

To see the undisplayed output, wait a moment, press `Space`, then press `Return`.

NightView responds with:

```
/usr/lib/NightView-release/ReadyToDebug
$ /usr/lib/NightView-release/ReadyToDebug
$ (local)
```

NightView runs the `ReadyToDebug` program and your executable in the dialogue shell. You might see only one echo of `/usr/lib/NightView-`*release*`/ReadyToDebug`, depending on how quickly the dialogue shell starts (*release* is the NightView release level). For information about `ReadyToDebug`, see "ReadyToDebug" on page 3-7. Note that in this tutorial the dialogue shell prompt is ''$ ''. Yours may differ.

Getting General and Error Help

This tutorial expects you to look up information in the NightView manual. For the command-line and simple screen interfaces, online help is available only for error messages. For general help you need to read the printed manual or consult the online help via NightView's graphical user interface or via **`nhelp(1)`**. When this tutorial refers to another section of the manual, use one of those methods to read the section.

Exercise:

Try to use the non-existent "foo" command.

Solution:

You should enter:

```
(local) foo
```

Note: do not enter the `(local)`. It is part of the command prompt.

NightView responds with:

```
Error: Unrecognizable command "foo". [E-command_proc003]
(local)
```

Exercise:

Now, invoke help without any arguments.

Solution:

You should enter one of:

```
(local) help
(local) he
```

NightView displays additional information about your most recent error and prints a new command prompt.

Note that **`he`** is not an official abbreviation for the **`help`** command; however, you may abbreviate NightView commands and some keywords to the shortest unambiguous prefix. For more information, see "Command Syntax" on page 7-1. You cannot abbreviate file names, symbolic names, or NightView construct names.

Exercise:

Once again, invoke help without any arguments.

Solution:

You should enter one of:

```
(local) help
(local) he
```

Note that NightView does not redisplay the extended error information; it assumes that you have already read that information. If there had been earlier errors, NightView would display help for the next most recent error now. However, there are no earlier errors, so NightView gives an error message indicating that.

NightView responds with the command prompt.

Starting Your Program

Most NightView commands operate on existing processes in a running program. Because you did not specify a program when you started the debugger, there haven't been any processes to debug. You must start `msg` now to debug it and to use most of the rest of the NightView commands in this tutorial.

Exercise:

Read about the **`run`** command. Use it to start the `msg` program and have the program wait for debugging.

Solution:

You should enter one of:

```
(local) run ./msg
(local) ru ./msg
```

The preceding "./" is a safety precaution. Avoid accidentally debugging the wrong program by always providing some path information.

NightView responds with:

```
./msg
New process:  local:15625     parent pid: 16428
Process local:15625 is executing /users/bob/nview/msg.
Reading symbols from /users/bob/nview/msg...done
Executable file set to
/users/bob/nview/msg
(local)
```

If `msg` was dynamically linked, NightView also displays the following messages:

```
Program was dynamically linked.
Dynamic linking completed.
The file "file" does not contain symbolic debug
information,
only external symbols will be visible.
```

The long message *may* not appear.

NightView shows the process ID (PID) of the new process and its parent process, the path where your executable exists, and another local dialogue prompt. Your PIDs and the path where your executable exists will probably differ from those in this tutorial. For information about processes, see "Programs and Processes" on page 3-2.

Note that by appending an ampersand (&) to the **`run`** command, you could have started your program in the background of the dialogue shell. This is generally a good idea because it gives you the flexibility to debug multiple programs in one NightView session; however, in this tutorial, you will be supplying the program with input, so the program needs to be running in the foreground.

Note also that although this tutorial does not ask you to do so, you can use the **`run`** command to rerun a program.

Debugging All Child Processes

By default, NightView debugs child processes only when they have called **`exec(2)`**. In the `msg` program, the child process never calls `exec`. To be able to debug this child process, you must use the **`set-children`** command *before* `msg` forks the child process. Also, you have to issue the **`set-children`** command *after* the **`run`** command so the **`set-children`** command can be applied to existing processes.

Exercise:

Read about the **`set-children`** command. Issue the **`set-children`** command so that the child process in `msg` can be debugged.

Solution:

You should enter one of:

```
(local) set-children all
(local) set-c a
```

Handling Signals

By default, signals stop execution under the debugger. In the `msg` program, the parent process sends signal SIGUSR1 to the child process. It then sleeps as a crude way of synchronizing the sending and receiving of signals. Having execution stop because of this signal is not desirable in this case.

Exercise:

Read about the **`handle`** command. Use it to adjust the default handling of the SIGUSR1 signal so that the process does not stop.

Solution:

You should enter one of:

```
(local) handle SIGUSR1 nostop
(local) ha usr1 nos
```

NightView responds with:

```
Signal handling complete
(local)
```

Note: you had to issue the **handle** command *after* the **run** command so the **handle** command could be applied to existing processes.

Listing the Source

You probably want to look at the source files before debugging them.

Exercise:

Read about the **list** command. Notice all the syntax variations for this command, and use one of them to examine the source file where `main` is defined.

Solution:

You should enter one of:

```
(local) list main.c:1     (for the C program)
(local) l main.c:1        (for the C program)
(local) list main.f:1     (for the Fortran program)
(local) l main.f:1        (for the Fortran program)
(local) list main.a:1     (for the Ada program)
(local) l main.a:1        (for the Ada program)
(local) list main
(local) l main
(local) list
(local) l
```

NightView responds by displaying ten numbered source lines. (You will see a different ten source lines depending on how you ran the **list** command.) Executable lines have an asterisk (*) source line decoration beside the line numbers. For more information about source line decorations, see "Source Line Decorations" on page 7-63.

The **list** command is repeatable. Press Return.

Now you see the next ten lines of the source file.

Keep pressing Return until you get an end of file message.

Exercise:

List the source file so the display is centered around line 16.

Solution:

You should enter one of:

```
(local) list main.c:16      (for the C program)
(local) l main.c:16         (for the C program)
(local) list main.f:16      (for the Fortran program)
(local) l main.f:16         (for the Fortran program)
(local) list main.a:16      (for the Ada program)
(local) l main.a:16         (for the Ada program)
(local) list 16
(local) l 16
```

NightView responds by listing the lines.

Setting the First Breakpoints

A breakpoint is set on the executable statement where you want program execution suspended. The program stops at the breakpoint *before* it executes the instruction where the breakpoint is set.

Exercise:

Read about the **`breakpoint`** command. Set a separate breakpoint to stop at each of the following places:

- The line that prompts for the number of signals to send
- The call to `child_routine`
- The comment before the call to `parent_routine`

Solution:

For the C program, this part of your debug session should look something like this:

```
(local) b 18
local:15625 Breakpoint 1 set at main.c:18
(local) b 25
local:15625 Breakpoint 2 set at main.c:25
(local) b 30
local:15625 Breakpoint 3 set at main.c:30
```

For the Fortran program, this part of your debug session should look something like this:

```
(local) b 15
local:15625 Breakpoint 1 set at main.f:15
```

```
(local) b 21
local:15625 Breakpoint 2 set at main.f:21
(local) b 23
local:15625 Breakpoint 3 set at main.f:23
```

For the Ada program, this part of your debug session should look something like this:

```
(local) b 18
local:15625 Breakpoint 1 set at main.a:18
(local) b 25
local:15625 Breakpoint 2 set at main.a:25
(local) b 27
local:15625 Breakpoint 3 set at main.a:27
```

Note that the preceding examples could have spelled out the **`breakpoint`** command. NightView gives each breakpoint an ordinal identification number beginning at 1. By default, breakpoints are set in the current list file, `main.c`, `main.f`, or `main.a` in this tutorial.

Note that you can put breakpoints only on executable statements. NightView did not give you an error for attempting to put a breakpoint on a comment line. Instead, it put the breakpoint on the executable statement that immediately follows the comment line.

Listing a Breakpoint

NightView changes the list display when you set a breakpoint.

Exercise:

Issue the **`list`** command that will relist the current lines.

Solution:

You should enter one of:

```
(local) list =
(local) l =
```

NightView redisplays the ten lines you were viewing. Note that if you are displaying a line with a breakpoint on it, that line now has a `B` (for breakpoint) source line decoration.

Continuing Execution

To make use of the breakpoints you set, you must allow the `msg` program to execute up to the statement with the breakpoint.

Exercise:

Read about the **`continue`** command. Use it to continue program execution up to the statement with the breakpoint.

Solution:

You should enter one of:

```
(local) continue
(local) c
```

NightView displays the statement with the breakpoint. Note that the source line decoration is now a `B=`. The `B` still indicates a breakpoint, and the `=` indicates that execution is stopped there.

Not Entering Functions

Execution is stopped at the line that prompts for the number of signals to send. You don't want to enter the code for the output statement (or function) because it is part of the library, not part of your program.

Exercise:

Read about the **`next`** command. Use it to skip over the output statement (or function).

Solution:

You should enter one of:

```
(local) next
(local) n
```

The `msg` program writes the prompt "How many signals should the parent send the child?". NightView displays the next line. The `=` source line decoration shows that execution is stopped there.

Entering Input

You must respond to the `msg` program prompt "How many signals should the parent send the child?". By default, NightView interprets all input as debugger commands.

Exercise:

Assume that you want to send ten signals. See what problems arise when you simply enter the number ten.

Solution:

You should enter:

```
(local) 10
```

NightView responds with an error message.

Exercise:

Read about the `!` command. Use it to make NightView understand that the 10 is data for the `msg` program. (For information about Dialogue I/O, see "Dialogue I/O" on page 3-5 and "!" on page 7-27.)

Solution:

You should enter:

```
(local) !10
```

NightView responds with:

```
(local)
```

As described in "Starting NightView" on page 4-3, NightView sometimes has output that does not appear until you press Return.

Press Space, then press Return to see your input echoed.

NightView responds with:

```
10
(local)
```

Creating Families

Naming a process or process group has the following advantages over specifying PIDs.

- Mnemonic names are often easier to remember and type than numeric PIDs.
- You can group PIDs with a single name so that qualified NightView commands act only on the processes in the group.
- You can write generic NightView command files that use process names instead of specific PIDs.

In this tutorial, you will want to issue some NightView commands that pertain only to the parent process and others that pertain only to the child process.

Exercise:

Read about the **`family`** command. Use it to give the name `parent` to all processes that currently exist in your program. (There is only one process so far.)

Solution:

You should enter one of:

```
(local) family parent all
(local) fa parent all
```

Note that to name only the parent process, you had to issue this command *before* NightView executes the `fork` in the `msg` program. Note also that at this point, the `all` argument represents only one process, the parent process. Later you will see it represent multiple processes.

You will use the `parent` family name later in the tutorial.

Continuing Execution Again

Before you can examine aspects of `parent_routine` and `child_routine`, you must get NightView to stop at the calls to these routines.

Exercise:

Continue executing the program so it stops at the next statement with a breakpoint.

Solution:

You should enter one of:

```
(local) continue
(local) c
```

For the C program, NightView responds with:

```
New process:  local:13504     parent pid: 15625
#0  0x10002838  in main() at main.c line 20
20 < |    pid = fork();
local:15625: at Breakpoint 3, 0x1000284c in main() at
main.c line 31
31 B=|       parent_routine( pid, total_sig );
(local)
```

For the Fortran program, NightView responds with:

```
New process:  local:13504     parent pid: 15625
#0  0x100038e4  in main() at main.f line 17
```

```
17 < |       pid = fork()
local:15625: at Breakpoint 3, 0x10003904 in main() at
main.f line 24
24 B=|          call parent_routine( pid )
(local)
```

For the Ada program, NightView responds with:

```
New process:  local:13504     parent pid: 15625
#0  0x10010bc8  in main() at main.a line 21
21 < |   pid := posix_1003_1.fork;
local:15625: at Breakpoint 3, 0x10010bdc in main() at
main.a line 28
28 B=|       parent_routine( pid, total_sig );
(local)
```

The first few lines are from the child process. They show that you are currently calling `fork`. The `<` source line decoration indicates that this line made a subprogram call which has not yet returned. The subprogram that implements `fork` is hidden. NightView usually does not show you system library routines. See "Interesting Subprograms" on page 3-27.

In this example, the child process has process ID 13504, and the parent process has process ID 15625. Note that your process IDs will differ. Note also that after the `fork`, only the parent process continued execution; the child process is still at the `fork`.

The source line decoration in the parent process is now a `B=`. The `B` still indicates a breakpoint and the `=` indicates that execution is stopped there.

Creating Families Again

In this tutorial, you will want to issue some NightView commands that pertain only to the parent process and others that pertain only to the child process.

Exercise:

Use the **`family`** command to give the name `child` to only the newly forked child process.

Solution:

You should enter one of:

```
(local) family child all - parent
(local) fa child all - parent
```

At this time, the `all` argument consists of both the parent and child PIDs. In section "Creating Families" on page 4-12, you created the `parent` family so it consists of only the parent PID. Subtraction leaves only the child PID in the `child` family.

You will use the `child` family name later in the tutorial.

Note that to name only the child process, you had to issue this command *after* NightView executes the `fork` in the `msg` program.

Catching up the Child Process

To individually manipulate the parent and child processes, you must qualify your debugger commands.

Exercise:

Read about qualifiers. Get the child process to continue execution up to the breakpoint on the call to `child_routine` (line 25 in `main.c`, line 21 in `main.f`, and line 25 in `main.a`).

Solution:

You should enter one of:

```
(local) (child) continue
(local) (child) c
```

For the C program, NightView displays:

```
local:13504: at Breakpoint 5, 0x10002840 in main() at
main.c line 25
25 B=|       child_routine( total_sig );
(local)
```

For the Fortran program, NightView displays:

```
local:13504: at Breakpoint 4, 0x100038fc in main() at
main.f line 21
21 B=|          call child_routine()
(local)
```

For the Ada program, NightView displays:

```
local:13504: at Breakpoint 4, 0x10010bd0 in main() at
main.a line 25
25 B=|       child_routine( total_sig );
(local)
```

This breakpoint in the child corresponds to breakpoint 2 in the parent. Inherited eventpoints get new identifiers. The order of eventpoint numbers in the child is unpredictable, so you might see a breakpoint number of 4, 5, or 6.

Note that you could have qualified the command with the child's process ID number instead of the `child` family name.

Verifying Data Values

You want to look at the value of variables in the `msg` program.

Exercise:

Read about the **`print`** command. Use it to check that the `total_sig` variable has the value 10.

Solution:

You should enter one of:

```
(local) print total_sig
(local) p total_sig
```

NightView responds with:

```
Process local:15625:
$1: total_sig = 10
Process local:13504:
$2: total_sig = 10
```

By default, the 10 is printed in decimal. NightView keeps a history of printed values. The `$1` means that this is the first value in this history. For more information about the printed value history, see "Value History" on page 3-32.

Note that if you had looked at the `total_sig` variable *after* its last use, you might have seen gibberish. This happens when the location holding a value gets overwritten. For more information, see "Optimization" on page 3-33. In the Fortran program, `total_sig` was put in COMMON so you could consistently see its value in the tutorial.

NightView displays values for both processes because there are multiple processes in the default qualifier `local`.

Entering Functions

At this point, the parent process is about to run `parent_routine`, and the child process is about to run `child_routine`.

Exercise:

Read about the **`step`** command. Use it to simultaneously enter both routines.

Solution:

You should enter one of:

```
(local) step
(local) s
```

Note that if you had wanted to enter a routine in only one process, you would have had to qualify the **step** command. (For information about qualifiers, see “Qualifiers” on page 3-4.)

In all the following output descriptions, NightView displays the line you stepped to. The = source line decoration indicates that execution is stopped there.

For the C program, NightView displays:

```
#0  0x10002884  in child_routine(int total_sig = 10) at
child.c line 14
#0  0x10002944  in parent_routine(pid_t child_pid =
13504, int total_sig = 10)
                  at parent.c line 11
14 = |    signal( SIGUSR1, signal_handler );
11 = |    int isec = 2;
(local)
```

Line 14 is from the child process. Line 11 is from the parent process.

For the Fortran program, NightView displays:

```
#0  0x1000393c  in child_routine() at child.f line 17
17 = |       ireturn = csignal( SIGUSR1, signal_handler,
-1  )
#0  0x10003a48  in parent_routine(INTEGER child_pid /
13504 / )
                  at parent.f line 15
15 = |       do 10 sig_ct = 1, total_sig
(local)
```

Line 17 is from the child process. Line 15 is from the parent process.

For the Ada program, NightView displays:

```
#0  0x100108fc  in child_routine(total_sig : IN integer =
10) at child.a line 26
#0  0x10010578  in parent_routine(child_pid : IN pid_t =
13504,
                  total_sig : IN integer = 10) at
parent.a line 6
26 = | procedure child_routine( total_sig : integer ) is
6  = | procedure  parent_routine( child_pid :
posix_1003_1.pid_t; total_sig : integer )  is
(local)
```

Line 26 is from the child process. Line 6 is from the parent process.

NightView tells you when a **step** command takes you into (or out of) a subprogram call. The lines that begin with `#0` announce that you have entered `child_routine` in the child process and `parent_routine` in the parent process.

Note that the order of the lines displayed may vary.

Examining the Stack Frames

It is often helpful to see how you got to a certain point in a program.

Exercise:

Read about the **`backtrace`** command. Use it to display the list of currently active stack frames.

Solution:

You should enter one of:

```
(local) backtrace
(local) bt
```

For the C program, NightView responds with:

```
Backtrace for process local:13504
#0  0x10002884  in child_routine(int total_sig = 10) at
child.c line 14
#1  0x10002848  in main() at main.c line 25
Backtrace for process local:15625
#0  0x10002944  in parent_routine(pid_t child_pid =
13504, int total_sig = 10)
                  at parent.c line 11
#1  0x10002854  in main() at main.c line 31
(local)
```

For the Fortran program, NightView responds:

```
Backtrace for process local:13504
#0  0x1000393c  in child_routine() at child.f line 17
#1  0x10003900  in main() at main.f line 21
Backtrace for process local:15625
#0  0x10003a48  in parent_routine(INTEGER child_pid /
13504 / )
                  at parent.f line 15
#1  0x10003910  in main() at main.f line 24
(local)
```

For the Ada program, NightView responds:

```
Backtrace for process local:13504
#0  0x100108fc  in child_routine(total_sig : IN integer =
10) at child.a line 26
#1  0x10010bd8  in main() at main.a line 25
#2  0x10022750  in <anonymous>()
Backtrace for process local:15625
#0  0x10010578  in parent_routine(child_pid : IN pid_t =
13504,
                  total_sig : IN integer = 10) at
parent.a line 6
```

```
#1  0x10010be4  in main() at main.a line 28
#2  0x10022750  in <anonymous>()
(local)
```

Note the order of the displayed lines may vary.

On lines labeled `#0`, NightView shows its location within the current routine. On lines labeled `#1`, NightView shows the location of the call to the current routine within the calling routine.

In the Ada program, stack frame #2 is from the library level elaboration routine, which has no name.

Moving in the Stack Frames

You may want to move among the stack frames to examine and modify variables, run functions, etc., in other frames. For example, suppose that you want to examine the value of local variable `tracefile` in `main`.

Exercise:

Read about the **`up`** command. Qualify the **`up`** command so the current stack frame of the parent process is `main`.

Solution:

You should enter:

```
(local) (parent) up
```

For the C program, NightView responds with:

```
Output for process local:15625
#1  0x10002854  in main() at main.c line 31
31 B<|      parent_routine( pid, total_sig );
(local)
```

For the Fortran program, NightView responds with:

```
Output for process local:15625
#1  0x10003910  in main() at main.f line 24
24 B<|       call parent_routine( pid )
(local)
```

For the Ada program, NightView responds with:

```
Output for process local:15625
#1  0x10010be4  in main() at main.a line 28
28 B<|       parent_routine( pid, total_sig );
(local)
```

The < source line decoration indicates that this line made a subprogram call which has not yet returned.

Note that you could have qualified the command with the parent's process ID number instead of the `parent` family name.

Verifying Data Values in Other Stack Frames

From `main`, you can examine local variables, run functions, etc.

Exercise:

Qualify a **`print`** command so it displays the value of local variable `tracefile` in `main` only for the parent process.

Solution:

You should enter one of:

```
(local) (parent) print tracefile
(local) (parent) p tracefile
```

For the C program, NightView responds with:

```
$3: tracefile = 0x30003100 "msg_file"
(local)
```

For the Fortran and Ada programs, NightView responds with:

```
$3: tracefile = "msg_file"
(local)
```

Note that you could have qualified the command with the parent's process ID number instead of the `parent` family name.

Returning to a Stack Frame

You want to return to `parent_routine`.

Exercise:

Read about the **`down`** command. Qualify the **`down`** command so the current stack frame of the parent process is `parent_routine`.

Solution:

You should enter one of:

```
(local) (parent) down
(local) (parent) do
```

For the C program, NightView responds with:

```
Output for process local:15625
#0  0x10002944  in parent_routine(pid_t child_pid =
13504, int total_sig = 10)
                  at parent.c line 11
11 = |    int isec = 2;
(local)
```

For the Fortran program, NightView responds with:

```
Output for process local:15625
#0  0x10003a48  in parent_routine(INTEGER child_pid /
13504 / )
                  at parent.f line 15
15 = |       do 10 sig_ct = 1, total_sig
(local)
```

For the Ada program, NightView responds with:

```
Output for process local:15625
#0  0x10010578  in parent_routine(child_pid : IN pid_t =
13504,
                  total_sig : IN integer = 10) at
parent.a line 6
6  = | procedure  parent_routine( child_pid :
posix_1003_1.pid_t; total_sig : integer )  is
(local)
```

Note: it is not meaningful to do a **`down`** without doing an **`up`** *first* (as you did in section “Moving in the Stack Frames” on page 4-19).

Resuming Execution

You want to continue the execution of the child process so that it will get signals as soon as they are sent by the parent process. The **`continue`** command can do this, but it ties up the debugger’s input mechanism while waiting for the process. You don’t want to wait.

Exercise:

Read about the **`resume`** command. Qualify the **`resume`** command to resume execution of the *child* process without the waiting that occurs with the **`continue`** command.

Solution:

You should enter one of:

```
(local) (child) resume
(local) (child) res
```

Note that you could have qualified the command with the child's process ID number instead of the `child` family name.

Setting the Default Qualifier

As described in "Starting NightView" on page 4-3, the default qualifier is `local`, which means that unqualified commands affect all processes. It is cumbersome to keep qualifying commands that operate on a subset of these processes. The rest of the commands in this tutorial apply only to the parent process.

Exercise:

Read about the **`set-qualifier`** command. Use it to tell NightView that the default qualifier for the remaining commands is the family that consists of only the parent process.

Solution:

You should enter one of:

```
(local) set-qualifier parent
(local) set-q parent
```

NightView changes the prompt to your new qualifier, `parent`.

Removing a Breakpoint

Breakpoint 1 (set in "Setting the First Breakpoints" on page 4-9) is no longer needed.

Exercise:

Read about the **`delete`** command. Use it to remove breakpoint 1.

Solution:

You should enter one of:

```
(parent) delete 1
(parent) d 1
```

Setting Conditional Breakpoints

It is often useful to suspend execution conditionally.

Exercise:

Read about the **`breakpoint`** command. Set a breakpoint on the line that displays how long the parent is sleeping in `parent_routine`; the breakpoint should suspend execution when the value of `isec` equals the value of `total_sig`.

Solution:

For the C program, you should enter one of:

```
(parent) breakpoint 16 if isec == total_sig
(parent) b 16 if isec == total_sig
```

For the Fortran program, you should enter one of:

```
(parent) breakpoint 16 if isec .eq. total_sig
(parent) b 16 if isec .eq. total_sig
```

For the Ada program, you should enter one of:

```
(parent) breakpoint 15 if isec = total_sig
(parent) b 15 if isec = total_sig
```

For the C program, NightView responds with:

```
local:15625 Breakpoint 7 set at parent.c:16
```

For the Fortran program, NightView responds with:

```
local:15625 Breakpoint 7 set at parent.f:16
```

For the Ada program, NightView responds with:

```
local:15625 Breakpoint 7 set at parent.a:15
```

Attaching an Ignore Count to a Breakpoint

Sometimes you won't want to monitor each iteration of a loop. For example, assume that a loop runs many times, and somewhere during the loop an error occurs. You could ignore the first half of the loop values to determine in which half of the iterations the error occurred.

Exercise:

Read about the **ignore** command. Set a **breakpoint** command on the line that displays how long the parent is sleeping in `parent_routine`. NightView has a predefined name for the most-recently set breakpoint. For more information about this name, see "Eventpoint Specifiers" on page 7-12. Use this name on an **ignore** command on this line in `parent_routine`; ignore the next five iterations.

Solution:

For the C and Fortran programs, you should enter:

```
(parent) breakpoint 16
(parent) ignore . 5
```

or

```
(parent) b 16
(parent) ig . 5
```

For the Ada program, you should enter:

```
(parent) breakpoint 15
(parent) ignore . 5
```

or

```
(parent) b 15
(parent) ig . 5
```

For the C program, NightView responds with:

```
local:15625 Breakpoint 8 set at parent.c:16
Will ignore next 5 crossings of Breakpoint #8 in
local:15625.
```

For the Fortran program, NightView responds with:

```
local:15625 Breakpoint 8 set at parent.f:16
Will ignore next 5 crossings of Breakpoint #8 in
local:15625.
```

For the Ada program, NightView responds with:

```
local:15625 Breakpoint 8 set at parent.a:15
Will ignore next 5 crossings of Breakpoint #8 in
local:15625.
```

Attaching Commands to a Breakpoint

You can attach arbitrary NightView commands to a breakpoint. They run when that particular breakpoint is hit.

Exercise:

Read about the **`commands`** command. Attach a command stream that prints out the value of `total_sig` only when you hit the breakpoint you set in the previous step. Note: use the NightView predefined name for the most-recently set breakpoint.

Solution:

You should enter one of:

```
(parent) commands .
(parent) com .
```

NightView responds with:

```
Type commands for when the breakpoints are hit, one per
line.
End with a line saying just "end".
>
```

You should enter:

```
> print total_sig
> end
```

or

```
> p total_sig
> end
```

Automatically Printing Variables

You can create a list of one or more variables to be printed each time execution stops.

Exercise:

Read about the **`display`** command. Use a **`display`** command to display the value of the `sig_ct` variable.

Solution:

You should enter one of:

```
(parent) display sig_ct
(parent) disp sig_ct
```

Note that this **`display`** command runs every time execution stops, and the **`print`** command from “Attaching Commands to a Breakpoint” on page 4-24 runs only when execution stops at a specific breakpoint.

Watching Inter-Process Communication

You already resumed the execution of the child process, so NightView gave you a prompt and did not wait for the child process.

Exercise:

Now continue execution for the parent process.

Solution:

You should enter one of:

```
(parent) continue
(parent) c
```

NightView responds with something like the following:

```
 1.  Parent sleeping for 2 seconds
 2.  Parent sleeping for 2 seconds
Child got ordinal signal #1
 3.  Parent sleeping for 2 seconds
Process local:13504 received SIGUSR1
Child got ordinal signal #2
 4.  Parent sleeping for 2 seconds
Process local:13504 received SIGUSR1
Child got ordinal signal #3
 5.  Parent sleeping for 2 seconds
Process local:13504 received SIGUSR1
Child got ordinal signal #4
Process local:13504 received SIGUSR1
Child got ordinal signal #5
local:15625: at Breakpoint 8, 0x10002950 in parent_routine(
                  pid_t child_pid = 13504, int total_sig = 10)
                  at parent.c line 16
16 B=|       printf( "%d.  Parent sleeping for %d seconds\n", sig_ct, isec );
1: sig_ct = 6
(parent)
```

Note the order of the displayed lines may vary. For the Fortran and Ada programs, NightView prints the argument or arguments to `parent_routine` differently.

Because of the **`ignore`** command on breakpoint 8, the parent process sent only five out of ten signals to the child process before the breakpoint was hit. The source code is written so that the lines that begin with a number come from the parent process, and the lines that begin with the word "Child" come from the child process. The lines that mention signal SIGUSR1 appear because the **`handle`** command is implicitly set to **`print`** and explicitly set to `nostop`. Two lines show where execution stopped; these lines will differ depending on your programming language. Another line shows the value of `sig_ct` because of the **`display`** command.

Note that the **`print total_sig`** output did not appear because NightView returned your prompt before the commands in the **`commands`** command stream completed their output.

Exercise:

To see the **`print total_sig`** output, enter a space *and* Return.

WARNING

If you press `Return` *without* the space, you will repeat the **`continue`** command.)

NightView responds with the following:

```
$4: total_sig = 10
(parent)
```

Patching Your Program

You just watched the parent process sleep for 2 seconds between sending signals to the child process. Look at how this is done in the source.

Exercise:

List the source file for the `parent_routine` so the display is centered around line 13.

Solution:

You should enter one of:

`(parent)` **`list parent.c:13`**	(for the C program)
`(parent)` **`l parent.c:13`**	(for the C program)
`(parent)` **`list parent.f:13`**	(for the Fortran program)
`(parent)` **`l parent.f:13`**	(for the Fortran program)
`(parent)` **`list parent.a:13`**	(for the Ada program)
`(parent)` **`l parent.a:13`**	(for the Ada program)

You will notice that the variable `isec` always has the value 2. Instead, you could vary the sleep interval `isec` by assigning it a value from 1 through 3, based on the signal count `sig_ct`. Hint: In C the `%` operator, in Fortran the `mod` function, and in Ada the `rem` operator may be useful.

Exercise:

Read about the **`patchpoint`** command. In the parent process, *on* the line that displays how long the parent is sleeping, patch in the assignment expression just described.

Solution:

For the C program, you should enter:

```
(parent) patchpoint at 16 eval isec = sig_ct % 3 + 1
```

For the Fortran program, you should enter:

```
(parent) patchpoint at 16 eval isec = mod( sig_ct, 3 ) + 1
```

For the Ada program, you should enter:

```
(parent) patchpoint at 15 eval isec := sig_ct rem 3 + 1
```

For the C program, NightView responds with the following:

```
local:15625 Patchpoint 9 set at parent.c:16
```

For the Fortran program, NightView responds with the following:

```
local:15625 Patchpoint 9 set at parent.f:16
```

For the Ada program, NightView responds with the following:

```
local:15625 Patchpoint 9 set at parent.a:15
```

Disabling a Breakpoint

You want to run `msg` to completion without stopping at breakpoint 8.

Exercise:

Read about the **`disable`** command. Use it to disable breakpoint 8 (set in section "Attaching an Ignore Count to a Breakpoint" on page 4-23).

Solution:

You should enter one of:

```
(parent) disable 8
(parent) disa 8
```

Examining Eventpoints

An *eventpoint* is a generic term which includes breakpoints, patchpoints, monitorpoints, agentpoints, and tracepoints. You want to examine the locations, associated commands, and statistics related to the eventpoints you have set in `msg`.

Exercise:

Read about the **`info eventpoint`** command. Use it to examine all eventpoints.

Solution:

You should enter one of:

```
(parent) (local) info eventpoint
```

```
(parent) (local) i ev
(parent) (all) info eventpoint
(parent) (all) i ev
```

For the C program, NightView responds with the following:

```
Eventpoints for process local:15625:

ID  Typ Enb       Where
--- --- --- -------------------
  2  B   Y  main.c:25
  3  B   Y  main.c:30
    #crossings=1  #hits=1
  7  B   Y  parent.c:16
    only if  isec == total_sig
    #crossings=6
  8  B   N  parent.c:16
    #crossings=6  #hits=1
    commands:
       print total_sig
  9  P   Y  parent.c:16
    eval =  isec = sig_ct % 3 + 1

Eventpoints for process local:13504:

ID  Typ Enb       Where
--- --- --- -------------------
  4  B   Y  main.c:18
    #crossings=1  #hits=1
  5  B   Y  main.c:25
    #crossings=1  #hits=1
  6  B   Y  main.c:30
(parent)
```

For the Fortran program, NightView responds with the following:

```
Eventpoints for process local:15625:

ID  Typ Enb       Where
--- --- --- -------------------
  2  B   Y  main.f:21
  3  B   Y  main.f:23
    #crossings=1  #hits=1
  7  B   Y  parent.f:16
    only if  isec .eq. total_sig
    #crossings=6
  8  B   N  parent.f:16
    #crossings=6  #hits=1
    commands:
       print total_sig
  9  P   Y  parent.f:16
    eval =  isec = mod( sig_ct, 3 ) + 1

Eventpoints for process local:13504:
```

```
ID  Typ Enb        Where
--- --- --- -------------------
  4  B   Y  main.f:21
    #crossings=1  #hits=1
  5  B   Y  main.f:23
  6  B   Y  main.f:15
    #crossings=1  #hits=1
(parent)
```

For the Ada program, NightView responds with the following:

```
Eventpoints for process local:15625:

ID  Typ Enb        Where
--- --- --- -------------------
  2  B   Y  main.a:25
  3  B   Y  main.a:27
    #crossings=1  #hits=1
  7  B   Y  parent.a:15
    only if  isec = total_sig
    #crossings=6
  8  B   N  parent.a:15
    #crossings=6  #hits=1
    commands:
       print total_sig
  9  P   Y  parent.a:15
    eval =  isec := sig_ct rem 3 + 1

Eventpoints for process local:13504:

ID  Typ Enb        Where
--- --- --- -------------------
  4  B   Y  main.a:25
    #crossings=1  #hits=1
  5  B   Y  main.a:27
  6  B   Y  main.a:18
    #crossings=1  #hits=1
(parent)
```

NightView displays all eventpoints for process `local:15625` followed by the eventpoints for process `local:13504`.

Breakpoints 1, 2, and 3 were set in "Setting the First Breakpoints" on page 4-9. Breakpoint 1 has no entry because it was deleted in "Removing a Breakpoint" on page 4-22. Breakpoints 2 and 3 are still enabled. Breakpoint 3 has been crossed once and hit once. This means that its line has been executed once and stopped on once.

When the child process was forked, it inherited the parent process's breakpoints. The child's breakpoints 4, 5, and 6 correspond to the parent's breakpoints 1, 2, and 3. The order of the eventpoint numbers for inherited eventpoints is not necessarily the same as in the parent.

Breakpoint 7 was set in "Setting Conditional Breakpoints" on page 4-23 and is still enabled; note that NightView displays the condition on this breakpoint. This breakpoint

has been crossed six times without being hit. This means that the line has been executed six times, but the condition has not been true yet.

Breakpoint 8 was set in "Attaching an Ignore Count to a Breakpoint" on page 4-23 and was disabled in "Disabling a Breakpoint" on page 4-28; note that NightView displays the commands (`print total_sig`) attached to this breakpoint. This breakpoint has been crossed six times and has been hit only once. This means that the line has been executed six times, but the **`ignore`** command has prevented the breakpoint from being hit more than once.

Patchpoint 9 was set in "Patching Your Program" on page 4-27 and is still enabled; note that NightView displays the expression associated with this patchpoint. This patchpoint has not been crossed or hit yet so these statistics are omitted from the display.

Continuing to Completion

There's nothing else to look at, so you decide to run `msg` to completion.

Exercise:

Use the **`continue`** command to continue execution.

Solution:

You should enter one of:

```
(parent) continue
(parent) c
```

NightView responds with something like this:

```
6.  Parent sleeping for 1 seconds
7.  Parent sleeping for 2 seconds
Process local:13504 received SIGUSR1
Child got ordinal signal #6
8.  Parent sleeping for 3 seconds
Process local:13504 received SIGUSR1
Child got ordinal signal #7
9.  Parent sleeping for 1 seconds
Process local:13504 received SIGUSR1
Child got ordinal signal #8
10.  Parent sleeping for 2 seconds
Process local:13504 received SIGUSR1
Child got ordinal signal #9
Process local:13504 received SIGUSR1
Child got ordinal signal #10
Process local:15625 is about to exit normally
Process local:13504 is about to exit normally
#1  0x1000285c  in main() at main.c line 34
#1  0x1000285c  in main() at main.c line 34
```

```
34 <>|    exit( 0 );
34 <>|    exit( 0 );
--> Undisplayed items:
  1:  (print) sig_ct
(parent)
```

Note the order of the displayed lines may vary.

The source code is written so that the lines that begin with a number come from the parent process, and the lines that begin with the word "Child" come from the child process. Note that the sleep interval varies from 1 through 3 because of the patch you made. The lines that mention signal SIGUSR1 appear because the **`handle`** command is implicitly set to **`print`** and explicitly set to `nostop`.

The last two lines say that `sig_ct` is not displayed. This message appears because of the **`display`** command and because the `sig_ct` variable is not visible at this point in the parent process. For the Fortran program, `sig_ct` is displayed, because it is still available.

Leaving the Debugger

The tutorial is over.

Exercise:

Read about the **`quit`** command. Use it to leave the debugger.

Solution:

You should enter one of:

```
(parent) quit
(parent) q
```

Neither process has completely exited, so NightView asks the following question:

```
Kill all processes being debugged?
```

Exercise:

Make the processes go away.

Solution:

You should respond:

```
Kill all processes being debugged? y
```

NightView responds with:

```
You are now leaving NightView...
Process local:13504 exited normally
Process local:15625 exited normally
Dialogue local has exited.
```

5
Tutorial - GUI

This is the tutorial for the graphical user interface (GUI) version of NightView. NightView's graphical user interface runs only on X servers. For more information about the graphical user interface, see Chapter 9 [Graphical User Interface] on page 9-1. You may also be interested in the command-line version of this chapter in Chapter 4 [Tutorial] on page 4-1. There is a much shorter tutorial in Chapter 2 [A Quick Start - GUI] on page 2-1.

About the Tutorial - GUI

This tutorial shows only the most common debugger commands and features. It expects you to know the basics about window system concepts, processes, and signals, but you do not need to know about NightView and debugging concepts.

The supplied tutorial program spawns a child process. The parent writes a message to stdout, sleeps, sends signal SIGUSR1 to the child, and loops. The child writes a message to stdout when it receives the signal.

Become familiar with the tutorial's source code; see Chapter G [Tutorial Files] on page G-1 or the files under the `/usr/lib/NightView/Tutorial` directory. The source files are:

C	**Fortran**	**Ada**	
`msg.h`	`msg.i`	`-`	Defines constants
`main.c`	`main.f`	`main.a`	Forks a child and calls other routines
`parent.c`	`parent.f`	`parent.a`	Sends signals to the child
`child.c`	`child.f`	`child.a`	Receives signals from the parent

This tutorial takes at least two hours to do. Each section must be performed in order. If you do not have two hours, you may want to see Chapter 2 [A Quick Start - GUI] on page 2-1.

Exercises in this tutorial tell you to do things and ask you questions. Make the most of this tutorial and the manual; follow the steps below:

1. Look up the information.
2. Try to figure out the answer on your own.
3. Apply the provided solution in the correct window. (**Warning:** Perform the solutions exactly as indicated, or your results may differ from those

provided in later steps of the tutorial. Do not do anything until you see the words "you should" in the tutorial.)

You do not need to follow cross references in this tutorial **unless** you are explicitly told to read them.

Sometimes NightView displays a status so briefly that it seems to flash before being replaced by another status. This tutorial documents only the last status displayed.

This tutorial often displays process IDs. Your process IDs will probably differ from those shown. Also, the tutorial displays hexadecimal addresses. The addresses for your program may differ from those shown. Additionally, the line breaks in your output may differ from those shown because the lengths of displayed data items may vary.

The code produced when you create your program may vary slightly from the programs used to prepare this tutorial. In particular, the line shown as the return address from a subprogram may be different from what is shown here.

Some messages might not appear, or additional messages might appear, depending on your environment.

Some of the shortened commands that appear in this tutorial are not official abbreviations for NightView commands; however, you may abbreviate NightView commands and some keywords to the shortest unambiguous prefix. For more information, see "Command Syntax" on page 7-1. You cannot abbreviate file names, symbolic names, or NightView construct names.

Field names that begin with the word "dialogue" are part of the Dialogue Window. Field names that begin with the word "debug" are part of the Debug Window.

You could run this entire tutorial with commands and operations from the keyboard. However, to reduce confusion, use the mouse whenever possible during this tutorial. Use mouse button 1 when you are told to click, drag, and select.

Creating a Program - GUI

NightView is mainly used with executables that contain debug information. To create such a program, compile source files with a particular option, and link edit them.

Exercise:

Create a directory named `nview` where you can create files for this tutorial, and move into that directory.

Solution:

You should enter:

```
$ mkdir nview
$ cd nview
```

Note: do not enter the `$`. It is part of the shell prompt.

Exercise:

Use the manual to find out what compiler option is necessary for debugging. (Hint: use the index.)

Solution:

From the index, **Compiling** has this information. The **`-g`** compiler option puts debug information into an executable.

Exercise:

Decide what language program you want to debug. Do *not* copy the source files from the `/usr/lib/NightView/Tutorial` directory, just compile and link these files. Make the `msg` program contain debug information. What command or commands did you use?

Solution:

For C, you should enter:

```
$ cc  -g  -o msg /usr/lib/NightView/Tutorial/*.c
```

For Fortran, you should enter:

```
$ f77  -g  -o msg /usr/lib/NightView/Tutorial/*.f
```

For MAXAda, you should enter:

```
$ /usr/ada/bin/a.mkenv  -g
$ /usr/ada/bin/a.path  -I obsolescent
$ /usr/ada/bin/a.intro /usr/lib/NightView/Tutorial/*.a
$ /usr/ada/bin/a.partition  -create active  -main main
msg
$ /usr/ada/bin/a.build  -v msg
$ /usr/ada/bin/a.rmenv .
```

For HAPSE Ada, you should enter:

```
$ /usr/hapse/bin/a.mklib  -g  -f
$ /usr/hapse/bin/a.make  -v  -lib .  -o msg main \
     -f /usr/lib/NightView/Tutorial/*.a
$ /usr/hapse/bin/a.rmlib
```

You should now have a `msg` program with debug information in your `nview` directory. Note that for this tutorial, the source files should *not* be in this directory.

Starting NightView - GUI

You are ready to start up NightView *with* the graphical user interface.

Exercise:

Read how to invoke the graphical user interface of NightView. (You can find this information in the manual, on the man page, or by invoking **`nview`** with the **`-help`** option.) Start up the graphical user interface of NightView.

Solution:

In the index, **Starting the debugger**, **Invoking the debugger**, and **`nview`, invoking** have this information. See Chapter 6 [Invoking NightView] on page 6-1. You should enter:

```
$ nview
```

Note that in this tutorial `msg` does not appear on the **`nview`** invocation line.

NightView responds by displaying a Dialogue Window.

A Dialogue Window is used to control a NightView dialogue and for input and output with the dialogue shell. A dialogue contains a shell where you run shell commands and debug running programs. Each dialogue has a name; the default dialogue is `local`. The dialogue qualifier area shows the command qualifier, for this dialogue, `local`. For information about dialogues, see "Dialogues" on page 3-4. For information about Dialogue Windows, see "Dialogue Window" on page 9-16. For information about command qualifiers, see "Qualifiers" on page 3-4.

The dialogue I/O area displays:

```
/usr/lib/NightView-release/ReadyToDebug
$ /usr/lib/NightView-release/ReadyToDebug
$
```

NightView runs the **`ReadyToDebug`** program automatically as part of initialization. You might see only one echo of `/usr/lib/NightView-`*release*`/ReadyToDebug`, depending on how quickly the dialogue shell starts (*release* is the NightView release level). For information about **`ReadyToDebug`**, see "ReadyToDebug" on page 3-7. Note that in this tutorial the dialogue shell prompt is ''$ ''. Yours may differ.

Getting General and Error Help - GUI

This tutorial expects you to look up information in the NightView manual. You may read the hard copy or the similar online manual. The online manual is accessible through each major window's Help menu.

Exercise:

Try to use the non-existent "foo" command.

Solution:

In the dialogue command area, you should enter:

```
foo
```

and press Return.

NightView displays in the dialogue message area:

```
Error: Unrecognizable command "foo". [E-command_proc003]
```

Exercise:

Read about this error message.

Solution:

In the Dialogue Window, you should click on the Help menu and select On Last Error.

The Help Window displays additional information about your most recent error.

Read the information. Note that Summary of Commands appears highlighted.

Exercise:

Read about getting information about all NightView commands.

Solution:

In the Help Window, you should click on Summary of Commands.

The Help Window displays a list of NightView commands with each command highlighted. The vertical and horizontal scroll bars next to the help display let you examine the rest of the help text.

Exercise:

Read about the menu bar in the Dialogue Window.

Solution:

In the Dialogue Window, you should click on the Help menu and select On Context.

NightView changes your pointer to a modified question mark.

Click on the menu bar of the Dialogue Window.

NightView restores your pointer. The Help Window displays information about the Dialogue Window menu bar.

When this tutorial asks you to read about buttons, use this same procedure.

For now, you are finished using help.

Exercise:

Close the Help Window.

Solution:

In the Help Window, you should click on the File menu and select Exit. (The Help Window is running a separate program, so only that program will exit. NightView will still be running.)

The Help window goes away.

This tutorial discusses the Help menu again in "Debugging All Child Processes - GUI" on page 5-8. For more information about help, see "GUI Online Help" on page 9-2.

Starting Your Program - GUI

Most NightView features operate on existing processes in a running program. Because you did not specify a program when you started the debugger, there haven't been any processes to debug. You must start `msg` now to debug it and to use most of the rest of the NightView features in this tutorial.

Exercise:

Start the `msg` program, and have it wait for debugging.

Solution:

In the dialogue I/O area, you should enter:

```
./msg
```

and press Return.

The preceding "./" is a safety precaution. Avoid accidentally debugging the wrong program by always providing some path information.

NightView displays the Principal Debug Window. (You can create other Debug Windows, but you won't do that for this tutorial.)

The debug identification area shows that `msg` is the executable program the process is running.

The debug message area shows:

```
New Process:  local:  15625  parent pid:  17882
Process local:15625 is executing /users/bob/nview/msg.
Reading symbols from /users/bob/nview/msg...done
Executable file set to
/users/bob/nview/msg
Switched to process local:15625.
```

If `msg` was dynamically linked, NightView also displays the following messages:

```
Program was dynamically linked.
Dynamic linking completed.
The file "file" does not contain symbolic debug
information,
only external symbols will be visible.
```

The long message *may* not appear.

NightView shows the process ID (PID) of the new process and the path where your executable exists. Your PID and the path where your executable exists will probably differ from those in this tutorial. For information about processes, see "Programs and Processes" on page 3-2.

The message `Switched to process local:15625.` indicates that this process is the *currently displayed process*.

The debug source file name field shows the name of the source file that is being displayed in the debug source display, `main.c`, `main.f`, or `main.a`.

In the debug source display, NightView displays numbered source lines. Executable lines have an asterisk (`*`) source line decoration beside the line numbers. For more information about source line decorations, see "Source Line Decorations" on page 7-63. The vertical and horizontal scroll bars next to the debug source display let you examine the rest of the source file.

The debug status area shows the status `Stopped for exec`. This means that the process has just **`exec(2)`**'ed a new program image.

The debug qualifier area shows the qualifier, `local:15625`.

The debug group area has an entry for this process. The entry shows the qualifier, `local:15625`, the name of the program this process is running, `msg`, and the status of the process, `Stopped for exec`. See "Debug Group Area" on page 9-35.

The `Switch To` button and the buttons below the label `Switch To Stopped Process` are disabled (dimmed) because there is only one process present at this time.

The Dialogue Window lists an entry for process 15625 and says the process is running `msg`.

Note that by appending an ampersand (`&`) to the `./msg`, you could have started your program in the background of the dialogue shell. This is generally a good idea because it gives you the flexibility to debug multiple programs in one NightView session; however, in this tutorial, you will be supplying the program with input, so the program needs to be running in the foreground.

Note also that although this tutorial does not ask you to do so, you can rerun a program by invoking it again.

Debugging All Child Processes - GUI

By default, NightView debugs child processes only when they have called **`exec(2)`**. In the `msg` program, the child process never calls `exec`. To be able to debug this child process, you must use the **`set-children`** command *before* `msg` forks the child process. Also, you have to issue the **`set-children`** command *after* invoking `./msg` so the **`set-children`** command can be applied to existing processes.

Exercise:

Read about the **`set-children`** command.

Solution:

You should click on the Help menu of either window and select On Commands. Scroll down to the **`set-children`** command. Click on the highlighted text. Read the information that the Help Window displays about the **`set-children`** command.

Exercise:

Use the File menu to close the Help Window.

Solution:

In the Help Window, you should click on the File menu and select Exit.

The Help Window goes away.

When this tutorial asks you to read about commands, use this same procedure.

Exercise:

Issue the **`set-children`** command so that the child process in `msg` can be debugged.

Solution:

In the debug command area, you should enter one of:

```
set-children all
set-c a
```

and press Return.

NightView echoes this command in the debug message area.

Handling Signals - GUI

By default, signals stop execution under the debugger. In the `msg` program, the parent process sends signal SIGUSR1 to the child process. It then sleeps as a crude way of synchronizing the sending and receiving of signals. Having execution stop because of this signal is not desirable in this case.

Exercise:

Read about the **`handle`** command. Use it to adjust the default handling of the SIGUSR1 signal so that the process does not stop.

Solution:

In the debug command area, you should enter one of the following:

```
handle SIGUSR1 nostop
ha usr1 nos
```

and press Return.

NightView echoes this command and displays in the debug message area:

```
Signal handling complete
```

Note: you had to issue the **`handle`** command *after* invoking `./msg` so the **`handle`** command could be applied to existing processes.

Setting the First Breakpoints - GUI

A breakpoint is set on the executable statement where you want program execution suspended. The program stops at the breakpoint *before* it executes the instruction where the breakpoint is set.

Exercise:

Read about the Breakpoint debug command button in the Debug Window. Set a separate breakpoint to stop at each of the following places:

- The line that prompts for the number of signals to send
- The call to `child_routine`
- The comment before the call to `parent_routine`

Solution:

You should alternate between clicking on a prospective breakpoint line in the debug source display and clicking on the Breakpoint debug command button. Pause between each click so that NightView can respond.

For the C program, the lines are 18, 25, and 30. NightView displays the following information in the debug message area.

```
local:15625 Breakpoint 1 set at main.c:18
local:15625 Breakpoint 2 set at main.c:25
local:15625 Breakpoint 3 set at main.c:30
```

For the Fortran program, the lines are 15, 21, and 23. NightView displays the following information in the debug message area.

```
local:15625 Breakpoint 1 set at main.f:15
local:15625 Breakpoint 2 set at main.f:21
local:15625 Breakpoint 3 set at main.f:23
```

For the Ada program, the lines are 18, 25, and 27. NightView displays the following information in the debug message area.

```
local:15625 Breakpoint 1 set at main.a:18
local:15625 Breakpoint 2 set at main.a:25
local:15625 Breakpoint 3 set at main.a:27
```

An *eventpoint* is a generic term which includes breakpoints, patchpoints, monitorpoints, agentpoints, and tracepoints. NightView gives each eventpoint an ordinal identification number beginning at 1.

Note that you can put breakpoints only on executable statements. NightView did not give you an error for attempting to put a breakpoint on a comment line. Instead, it put the breakpoint on the executable statement that immediately follows the comment line. However, the message in the debug message area has the number of the line you clicked on.

NightView changes the debug source display when you set a breakpoint. Note that each line with a breakpoint on it now has a B (for breakpoint) source line decoration.

Continuing Execution - GUI

To make use of the breakpoints you set, you must allow the msg program to execute up to the statement with the breakpoint.

Exercise:

Read about the Resume debug command button in the Debug Window. Use it to continue program execution up to the statement with the breakpoint.

Solution:

In the Debug Window, you should click on the Resume button.

The debug status area shows the status Stopped at breakpoint 1. This means that the process hit breakpoint number 1. The debug group area shows the same status.

NightView changes the source line decoration on the statement with the breakpoint to B=. The B still indicates a breakpoint, and the = indicates that execution is stopped there.

For the C program, NightView displays the following in the debug message area:

```
local:15625: at Breakpoint 1, 0x10002818 in main() at
main.c line 18
```

For the Fortran program, NightView displays the following in the debug message area:

```
local:15625: at Breakpoint 1, 0x10003878 in main() at
main.f line 15
```

For the Ada program, NightView displays the following in the debug message area:

```
local:15625: at Breakpoint 1, 0x10010b18 in main() at
main.a line 18
```

Not Entering Functions - GUI

Execution is stopped at the line that prompts for the number of signals to send. You don't want to enter the code for the output statement (or function) because it is part of the library, not part of your program.

Exercise:

Read about the Next debug command button in the Debug Window. Use it to skip over the output statement (or function).

Solution:

In the Debug Window, you should click on the Next button.

The msg program writes the prompt "How many signals should the parent send the child?" in the dialogue I/O area.

In the debug source display, NightView changes the source line decoration of the next line to =, which shows that execution is stopped there.

The debug status area and the debug group area show the status Stopped after step. This means that the process has finished a stepping command.

Entering Input - GUI

You must respond to the `msg` program prompt "How many signals should the parent send the child?".

Exercise:

Send ten signals.

Solution:

In the dialogue I/O area, you should enter:

```
10
```

and press Return.

Continuing Execution Again - GUI

Before you can examine aspects of `parent_routine` and `child_routine`, you must get NightView to stop at the calls to these routines.

Exercise:

Continue executing the program so it stops at the next statement with a breakpoint.

Solution:

In the Debug Window, you should click on the Resume debug command button.

The debug status area and the debug group area show the status Stopped at breakpoint 3. This means that the process hit breakpoint number 3.

For the C program, NightView displays the following in the debug message area:

```
local:15625: at Breakpoint 3, 0x1000284c in main() at
main.c line 31
```

For the Fortran program, NightView displays the following in the debug message area:

```
local:15625: at Breakpoint 3, 0x10003904 in main() at
main.f line 24
```

For the Ada program, NightView displays the following in the debug message area:

```
local:15625: at Breakpoint 3, 0x10010bdc  in main() at
main.a line 28
```

The source line decoration is now a `B=`. The `B` still indicates a breakpoint, and the `=` indicates that execution is stopped there.

The debug group area has a new entry for the child process. The child process is the one with the status New Process.

The Switch To button and the buttons below the label Switch To Stopped Process are now enabled (not dimmed).

You would like to view the child process in the Debug Window.

Exercise:

Read about the debug group area. Switch to the child process.

Solution:

In the process list of the debug group area, you should click on the entry for the child process. Then you should click on the Switch To button.

Now the Debug Window is displaying the child process.

The debug identification area still shows that `msg` is the executable program the process in this window is running. (The child is executing the same program as the parent.) The qualifier specifier field now shows the qualifier of the child process.

For the C program, the debug message area shows:

```
Switched to process local:13504.
New process:  local:13504     parent pid: 15625
#0  0x10002838  in main() at main.c line 20
```

For the Fortran program, the debug message area shows:

```
Switched to process local:13504.
New process:  local:13504     parent pid: 15625
#0  0x100038e4  in main() at main.f line 17
```

For the Ada program, the debug message area shows:

```
Switched to process local:13504.
New process:  local:13504     parent pid: 15625
#0  0x10010bc8  in main() at main.a line 23
```

In this example, the child process has process ID 13504, and the parent process has process ID 15625. Note that your process IDs will differ. Note also that after the `fork`, only the parent process continued execution; the child process is still at the `fork`.

The debug source display shows the main program because execution is stopped in a routine (**`fork(2)`**) which is hidden because it is uninteresting. NightView usually does not show you system library routines. See “Interesting Subprograms” on page 3-27. The `<` source line decoration indicates that this line made a subprogram call which has not yet returned.

The debug status area shows the status `New process`. This means that the process has just been created by a **`fork(2)`** call in the parent process. The process is stopped. See "Multiple Processes" on page 3-2.

The debug qualifier area shows the qualifier, `local:13504`.

The Dialogue Window lists entries for processes 15625 and 13504.

Catching up the Child Process - GUI

Exercise:

Get the **child** process to continue execution up to the breakpoint on the call to `child_routine` (line 25 in `main.c`, line 21 in `main.f`, and line 25 in `main.a`).

Solution:

In the Debug Window, with the *child* as the currently displayed process, you should click on the `Resume` debug command button.

For the C program, NightView displays in the debug message area:

```
local:13504: at Breakpoint 5, 0x10002840 in main() at
main.c line 25
```

For the Fortran program, NightView displays in the debug message area:

```
local:13504: at Breakpoint 4, 0x100038fc in main() at
main.f line 21
```

For the Ada program, NightView displays in the debug message area:

```
local:13504: at Breakpoint 4, 0x10010bd0 in main() at
main.a line 25
```

The debug source file name is `main.c`, `main.f`, or `main.a`.

NightView puts a `B=` source line decoration in the debug source display on line 25 for the C and Ada programs and line 21 for the Fortran program.

The debug status area and the debug group area show the status `Stopped at breakpoint 5`. This means that the process hit breakpoint number 5. Breakpoint 5 in the child corresponds to breakpoint 2 in the parent. Inherited eventpoints get new identifiers, but the order of the eventpoint identifiers is unpredictable, so your breakpoint may have a different number.

Verifying Data Values - GUI

You want to look at the value of variables in the `msg` program.

Exercise:

Read about the Print debug command button in the Debug Window. Use it to check that the `total_sig` variable has the value 10.

Solution:

In the debug source display of the Debug Window, start at one side of any instance of the `total_sig` variable, hold down mouse button 1, drag it across the entire variable name, and release. (Alternatively, you could double click on the variable name where it appears surrounded by spaces.) Only the variable name should be highlighted. Click on the Print button.

NightView displays in the debug message area:

```
$1: total_sig = 10
```

The Print button always prints integers in decimal. NightView keeps a history of printed values. The `$1` means that this is the first value in this history. For more information about the printed value history, see "Value History" on page 3-32.

Note that if you had looked at the `total_sig` variable *after* its last use, you might have seen gibberish. This happens when the location holding a value gets overwritten. For more information, see "Optimization" on page 3-33. In the Fortran program, `total_sig` was put in COMMON so you could consistently see its value in the tutorial.

Listing the Source - GUI

You want to look at the source code for `child_routine`.

Exercise:

Read about the Debug Window's Source menu's List Function/Unit... item in "Debug Source Menu" on page 9-22. With the **parent** as the currently displayed process, use this item to display the source code for `child_routine`.

Solution:

You should switch to the **parent** process by clicking on the parent process's entry in the debug group area process list and then clicking on the Switch To button. Then you should click on the Source menu, and select List Function/Unit....

After pressing Switch To, Switched to process local:15625 appears in the debug message area. The debug status area shows `Stopped at breakpoint 3`. The debug source display shows that execution is stopped at the call to `parent_routine`.

After clicking in the Source menu, NightView puts up the Select a Function/Unit dialog box.

Exercise:

Read about the Search button in the Select a Function/Unit dialog box. Use it to search for `child_routine`.

Solution:

In the Select a Function/Unit dialog box, you should enter `child_routine` as the regular expression, and click on the Search button. (For more information about regular expressions, see "Regular Expressions" on page 7-12.)

NightView finds the `child_routine` function and puts it in the list.

Exercise:

Read about the OK button in the Select a Function/Unit dialog box. Use it to change the debug source display.

Solution:

In the Select a Function/Unit dialog box, you should click on the OK button.

NightView closes the Select a Function/Unit dialog box.

NightView changes the debug source file name to `child.c`, `child.f`, or `child.a`, and the debug source display shows the source code.

Entering Functions - GUI

At this point, the parent process is about to run `parent_routine`, and the child process is about to run `child_routine`.

Exercise:

Change to group process mode.

Read "Group Process Mode" on page 9-14.

Solution:

From the debug menu bar, you should select Group Process Mode from the View menu. The debug qualifier area displays [Group Mode]. NightView displays this message in the debug message area: Changed to group process mode.

Exercise:

Read about the Step debug command button. Use the Step button to *simultaneously* enter both routines.

Solution:

In the debug command button area, you should click on the Step button.

Because both the parent and child processes are listed in the debug group area of this Debug Window, and the Debug Window is in group process mode, the Step button causes both processes to step.

For the C program, NightView displays in the debug message area:

```
#0  0x10002884  in child_routine(int total_sig = 10) at
child.c line 14
#0  0x10002944  in parent_routine(pid_t child_pid =
13504, int total_sig = 10)
                  at parent.c line 11
```

For the Fortran program, NightView displays in the debug message area:

```
#0  0x1000393c  in child_routine() at child.f line 17
#0  0x10003a48  in parent_routine(INTEGER child_pid /
13504 / )
                  at parent.f line 15
```

For the Ada program, NightView displays in the debug message area:

```
#0  0x100108fc  in child_routine(total_sig : IN integer =
10) at child.a line 26
#0  0x10010578  in parent_routine(child_pid : IN pid_t =
13504,
                  total_sig : IN integer = 10) at
parent.a line 6
```

NightView tells you when a step command takes you into (or out of) a subprogram call. The lines that begin with #0 announce that you have entered child_routine in the child process and parent_routine in the parent process.

Note that the order of the lines displayed may vary.

Both the State: fields in the debug group area show the status Stopped after step. This means that the processes have finished a stepping command. The debug status area shows the same status for the parent process.

NightView changes the debug source file name to parent.c, parent.f, or parent.a, and the debug source display shows the source code.

Line 11 of `parent.c`, line 15 of `parent.f`, or line 6 of `parent.a` in the debug source display has the = source line decoration, which indicates that execution is stopped there.

Examining the Stack Frames - GUI

It is often helpful to see how you got to a certain point in a program.

Exercise:

Read about the **`backtrace`** command. Use it to display the list of currently active stack frames for **both** processes.

Solution:

In the debug command area, you should enter one of:

```
backtrace
bt
```

and press Return.

NightView echoes this command in the debug message area.

For the C program, NightView displays in the debug message area:

```
Backtrace for process local:15625
#0  0x10002944  in parent_routine(pid_t child_pid =
13504, int total_sig = 10)
                  at parent.c line 11
#1  0x10002854  in main() at main.c line 31
Backtrace for process local:13504
#0  0x10002884  in child_routine(int total_sig = 10) at
child.c line 14
#1  0x10002848  in main() at main.c line 25
```

For the Fortran program, NightView displays in the debug message area:

```
Backtrace for process local:15625
#0  0x10003a48  in parent_routine(INTEGER child_pid /
13504 / )
                  at parent.f line 15
#1  0x10003910  in main() at main.f line 24
Backtrace for process local:13504
#0  0x1000393c  in child_routine() at child.f line 17
#1  0x10003900  in main() at main.f line 21
```

For the Ada program, NightView displays in the debug message area:

```
Backtrace for process local:15625
#0  0x10010578  in parent_routine(child_pid : IN pid_t =
13504,
```

```
                    total_sig : IN integer = 10) at
parent.a line 6
#1  0x10010be4  in main() at main.a line 28
#2  0x10022750  in <anonymous>()
Backtrace for process local:13504
#0  0x100108fc  in child_routine(total_sig : IN integer =
10) at child.a line 26
#1  0x10010bd8  in main() at main.a line 25
#2  0x10022750  in <anonymous>()
```

On lines labeled `#0`, NightView shows its location within the current routine. On lines labeled `#1`, NightView shows the location of the call to the current routine within the calling routine.

In the Ada program, stack frame #2 is from the library level elaboration routine, which has no name.

Moving in the Stack Frames - GUI

You may want to move among the stack frames to examine and modify variables, run functions, etc., in other frames. For example, suppose that you want to examine the value of local variable `tracefile` in `main`.

Exercise:

Change back to single process mode and make sure the **parent** process is the currently displayed process.

Solution:

From the debug menu bar, you should select Single Process Mode from the View menu. The debug qualifier area displays the qualifier for the currently selected process. NightView displays these messages in the debug message area:

```
Changed to single process mode.
Switched to process local:15625.
```

If the parent process is not the currently displayed process, you should switch to it.

Exercise:

Read about the **up** command. Use the **up** command to make the current stack frame of the **parent** process be `main`.

Solution:

In the debug command area, you should enter:

up

and press Return.

NightView echoes this command in the debug message area.

NightView changes the debug source file name to `main.c`, `main.f`, or `main.a`, and the debug source display shows the source code.

For the C program, NightView displays in the debug message area:

```
Output for process local:15625
#1  0x10002854  in main() at main.c line 31
```

For the Fortran program, NightView displays in the debug message area:

```
Output for process local:15625
#1  0x10003910  in main() at main.f line 24
```

For the Ada program, NightView displays in the debug message area:

```
Output for process local:15625
#1  0x10010be4  in main() at main.a line 28
```

The > source line decoration in the debug source display indicates that execution will resume there when the called routine returns. This source line decoration appears on line 34 of `main.c`, line 27 of `main.f`, and line 31 of `main.a`.

The < source line decoration in the debug source display indicates that this line made a subprogram call which has not yet returned. This source line decoration appears on line 31 of `main.c`, line 24 of `main.f`, and line 28 of `main.a`.

Verifying Data Values in Other Stack Frames - GUI

From `main`, you can examine local variables, run functions, etc.

Exercise:

Use the Print debug command button to display the value of local variable `tracefile` in `main` for the parent process.

Solution:

In the debug source display, start at one side of any instance of the `tracefile` variable, hold down mouse button 1, drag it across the entire variable name, and release. (Alternatively, you could double click on the variable name. Note that this does not work in the C source because double clicking would highlight text delimited by spaces; in this case, it would highlight the * with the variable name.) Only the variable name should be highlighted. Click on the Print button.

For the C program, NightView displays in the debug message area:

```
$2: tracefile = 0x30003100 "msg_file"
```

For the Fortran and Ada programs, NightView displays in the debug message area:

```
$2: tracefile = "msg_file"
```

Returning to a Stack Frame - GUI

You want to return to `parent_routine`.

Exercise:

Read about the **`down`** command. Use the **`down`** command to make the current stack frame of the parent process be `parent_routine`.

Solution:

In the debug command area, you should enter one of:

down
do

and press Return.

NightView echoes this command in the debug message area.

For the C program, NightView displays in the debug message area:

```
Output for process local:15625
#0  0x10002944  in parent_routine(pid_t child_pid =
13504, int total_sig = 10)
                  at parent.c line 11
```

For the Fortran program, NightView displays in the debug message area:

```
Output for process local:15625
#0  0x10003a48  in parent_routine(INTEGER child_pid /
13504 / )
                  at parent.f line 15
```

For the Ada program, NightView displays in the debug message area:

```
Output for process local:15625
#0  0x10010578  in parent_routine(child_pid : IN pid_t =
13504,
                  total_sig : IN integer = 10) at
parent.a line 6
```

NightView changes the debug source file name to `parent.c`, `parent.f`, or `parent.a`, and the debug source display shows the source code.

The = source line decoration in the debug source display indicates that execution stopped there. This source line decoration appears on line 11 of `parent.c`, line 15 of `parent.f`, and line 6 of `parent.a`.

Note: it is not meaningful to do a **down** without doing an **up** *first* (as you did in section "Moving in the Stack Frames - GUI" on page 5-19).

Resuming Execution - GUI

You want to continue the execution of the child process so that it will get signals as soon as they are sent by the parent process.

Exercise:

Use the Resume debug command button to resume execution of the **child** process.

Solution:

You should switch to the **child** process by clicking on the child process's entry in the debug group area process list and then clicking on the Switch To button. Then you should click on the Resume button.

After pressing Switch To, the debug source file shown is `child.c`, `child.f` or `child.a`. Switched to process local:13504 appears in the debug message area.

After pressing Resume, NightView disables (dims) most of the debug command buttons.

The debug status area and the debug group area show the status Running. This means that the process is currently executing.

Exercise:

The remainder of this tutorial does not deal with the child process directly. Arrange for the remaining commands to affect only the parent process.

Solution:

You should switch to the **parent** process by clicking on the parent process's entry in the debug group area process list and then clicking on the Switch To button.

After clicking Switch To, the debug source file name, source display area and status are changed to their values for the parent. The debug message area shows `Switched to local:15625`. The buttons that were dimmed for the child process are no longer dimmed.

Removing a Breakpoint - GUI

Breakpoint 1 (set in "Setting the First Breakpoints - GUI" on page 5-9) is no longer needed.

Exercise:

Read about the Debug Window's Eventpoint menu's Summarize/Change... item in "Debug Eventpoint Menu" on page 9-24. Use this item to remove breakpoint 1.

Solution:

You should click on the Eventpoint menu. Select Summarize/Change....

NightView displays the debug eventpoint summarize/change dialog box.

Three eventpoints appear in the eventpoint list. NightView displays the following message below the eventpoint list: 3 eventpoints were found.

Exercise:

Read about the Delete button in "Debug Eventpoint Summarize/Change Dialog Box" on page 9-42. Use it to delete the breakpoint.

Solution:

You should select breakpoint 1 from the eventpoint list, and click on the Delete button.

NightView puts up a warning dialog box.

Exercise:

Read the message in the warning dialog box, allow the delete to proceed, and make the dialog box go away.

Solution:

In the warning dialog box, you should click on the OK button.

NightView closes the warning dialog box and deletes the breakpoint from the eventpoint list.

NightView displays the following message below the eventpoint list: Deleted 1 eventpoint: 1.

You have finished removing this breakpoint.

Exercise:

Make the debug eventpoint summarize/change dialog box go away.

Solution:

In the debug eventpoint summarize/change dialog box, you should click on the Close button.

NightView closes the window.

Setting Conditional Breakpoints - GUI

It is often useful to suspend execution conditionally.

Exercise:

Read about the Debug Window's Eventpoint menu's Set Breakpoint... item in "Debug Eventpoint Menu" on page 9-24. Use this feature to set a breakpoint on the line that displays how long the parent is sleeping in `parent_routine`; the breakpoint should suspend execution when the value of `isec` equals the value of `total_sig`.

Solution:

In the debug source display, you should click on the line. For `parent.c` and `parent.f`, it is line 16. For `parent.a`, it is line 15. You should click on the Eventpoint menu. Select Set Breakpoint....

NightView displays the breakpoint dialog box.

Do *not* press Return after you enter the following text.

For the C program, you should enter in the condition text input area:

```
isec == total_sig
```

For the Fortran program, you should enter in the condition text input area:

```
isec .eq. total_sig
```

For the Ada program, you should enter in the condition text input area:

```
isec = total_sig
```

You are ready to finish setting the conditional breakpoint.

Exercise:

Save your changes and make the breakpoint dialog box go away.

Solution:

In the breakpoint dialog box, you should click on the OK button.

NightView closes the breakpoint dialog box.

For the C program, NightView displays in the debug message area:

```
local:15625 Breakpoint 7 set at parent.c:16
```

For the Fortran program, NightView displays in the debug message area:

```
local:15625 Breakpoint 7 set at parent.f:16
```

For the Ada program, NightView displays in the debug message area:

```
local:15625 Breakpoint 7 set at parent.a:15
```

The indicated line gets a B source line decoration in the debug source display.

Attaching an Ignore Count to a Breakpoint - GUI

Sometimes you won't want to monitor each iteration of a loop. For example, assume that a loop runs many times, and somewhere during the loop an error occurs. You could ignore the first half of the loop values to determine in which half of the iterations the error occurred.

Exercise:

Set a breakpoint on the line that displays how long the parent is sleeping in `parent_routine`, ignoring the next five iterations.

Solution:

In the debug source display, you should click on the line. For `parent.c` and `parent.f`, it is line 16. For `parent.a`, it is line 15. You should click on the Eventpoint menu. Select Set Breakpoint….

NightView displays the breakpoint dialog box.

Enter 5 in the ignore count text input area. Do *not* press Return.

You are ready to finish attaching an ignore count to a breakpoint.

Exercise:

Save your changes and make the breakpoint dialog box go away.

Solution:

In the breakpoint dialog box, you should click on the OK button.

NightView closes the breakpoint dialog box.

For the C program, NightView displays in the debug message area:

```
local:15625 Breakpoint 8 set at parent.c:16
```

For the Fortran program, NightView displays in the debug message area:

```
local:15625 Breakpoint 8 set at parent.f:16
```

For the Ada program, NightView displays in the debug message area:

```
local:15625 Breakpoint 8 set at parent.a:15
```

Attaching Commands to a Breakpoint - GUI

You can attach arbitrary NightView commands to a breakpoint. They run when that particular breakpoint is hit.

Exercise:

Attach a command stream that prints out the value of `total_sig` only when you hit the breakpoint you set in the previous step (set in "Attaching an Ignore Count to a Breakpoint - GUI" on page 5-25).

Solution:

You should click on the Eventpoint menu. Select Summarize/Change….

NightView displays the debug eventpoint summarize/change dialog box.

Exercise:

Read about the Change… button in "Debug Eventpoint Summarize/Change Dialog Box" on page 9-42. Use it to add commands to this breakpoint.

Solution:

Notice that some of the buttons are disabled (dimmed), because you have not yet selected an eventpoint from the eventpoint list. Select breakpoint 8 from the eventpoint list, which will enable the buttons, and click on the Change… button.

NightView displays the breakpoint dialog box.

Note that 5 is in the ignore count text input area from "Attaching an Ignore Count to a Breakpoint - GUI" on page 5-25.

Do *not* press Return after you enter the following text.

In the commands text input area, you should enter one of:

```
print total_sig
p total_sig
```

Exercise:

In the breakpoint dialog box, save your changes and make the dialog box go away.

Solution:

In the breakpoint dialog box, you should click on the OK button.

NightView closes the breakpoint dialog box.

Exercise:

Make the debug eventpoint summarize/change dialog box go away.

Solution:

In the debug eventpoint summarize/change dialog box, you should click on the Close button.

NightView closes the window.

Automatically Printing Variables - GUI

You can create a list of one or more variables to be printed each time execution stops.

Exercise:

Read about the **`display`** command. Use a **`display`** command to display the value of the `sig_ct` variable.

Solution:

In the debug command area, you should enter one of:

```
display sig_ct
disp sig_ct
```

and press Return.

NightView echoes this command in the debug message area.

Note that this **`display`** command runs every time execution stops, and the **`print`** command from “Attaching Commands to a Breakpoint - GUI” on page 5-26 runs only when execution stops at a specific breakpoint.

Watching Inter-Process Communication - GUI

You already resumed the execution of the child process, so NightView did not wait for the child process.

Exercise:

Now continue execution for the **parent** process.

Solution:

In the Debug Window, you should click on the Resume button.

In the dialogue I/O area, NightView responds with something like the following:

```
 1.  Parent sleeping for 2 seconds
 2.  Parent sleeping for 2 seconds
Child got ordinal signal #1
 3.  Parent sleeping for 2 seconds
Child got ordinal signal #2
 4.  Parent sleeping for 2 seconds
Child got ordinal signal #3
 5.  Parent sleeping for 2 seconds
Child got ordinal signal #4
Child got ordinal signal #5
```

Because of the ignore count on breakpoint 8, the parent process sent only five out of ten signals to the child process before the breakpoint was hit. The source code is written so that the lines that begin with a number come from the parent process, and the lines that begin with the word "Child" come from the child process.

The debug status area and the debug group area show the status Stopped at breakpoint 8. This means that the process hit breakpoint number 8.

For the C program, NightView displays something like the following in the debug message area:

```
local:15625: at Breakpoint 8, 0x10002950 in
parent_routine(
                  pid_t child_pid = 13504, int total_sig
= 10)
                   at parent.c line 16
1: sig_ct = 6
$3: total_sig = 10
```

For the Fortran program, NightView displays something like the following in the debug message area:

```
local:15625: at Breakpoint 8, 0x105d0 in parent_routine(
                   INTEGER child_pid / 13504 / ) at
parent.f line 16
1: sig_ct = 6
$3: total_sig = 10
```

For the Ada program, NightView displays something like the following in the debug message area:

```
local:15625: at Breakpoint 8, 0x30324 in parent_routine(
                  child_pid : IN integer = 13504,
                  total_sig : IN integer = 10) at
parent.a line 15
1: sig_ct = 6
$3: total_sig = 10
```

Initial lines show where execution stopped. One line shows the value of `sig_ct` because of the **`display`** command. Another line shows the value of `total_sig` from the **`print`** command attached to breakpoint 8.

Note that the order of the displayed lines may vary.

Patching Your Program - GUI

You just watched the parent process sleep for 2 seconds between sending signals to the child process. Look at how this is done in the source.

You will notice that the variable `isec` always has the value 2. Instead, you could vary the sleep interval `isec` by assigning it a value from 1 through 3, based on the signal count `sig_ct`. Hint: in C the `%` operator, in Fortran the `mod` function, and in Ada the `rem` operator may be useful.

Exercise:

Read about the Debug Window's Eventpoint menu's Set Patchpoint... item in "Debug Eventpoint Menu" on page 9-24. In the **parent** process, *on* the line that displays how long the parent is sleeping, patch in the assignment expression just described.

Solution:

You should click on the Eventpoint menu. Select Set Patchpoint....

NightView displays the patchpoint dialog box.

Do *not* press Return after you enter the following text.

For the C program, you should enter in the evaluate text input area:

```
isec = sig_ct % 3 + 1
```

For the Fortran program, you should enter in the evaluate text input area:

```
isec = mod( sig_ct, 3 ) + 1
```

For the Ada program, you should enter in the evaluate text input area:

```
isec := sig_ct rem 3 + 1
```

You are ready to finish patching your program.

Exercise:

Save your changes and make the patchpoint dialog box go away.

Exercise:

In the patchpoint dialog box, you should click on the OK button.

NightView closes the patchpoint dialog box.

Note that the line in the debug source display with a patchpoint on it now has a BP= (for breakpoint, patchpoint, and execution stopped here) source line decoration.

For the C program, NightView displays in the debug message area:

```
local:15625 Patchpoint 9 set at parent.c:16
```

For the Fortran program, NightView displays in the debug message area:

```
local:15625 Patchpoint 9 set at parent.f:16
```

For the Ada program, NightView displays in the debug message area:

```
local:15625 Patchpoint 9 set at parent.a:15
```

Disabling a Breakpoint - GUI

You want to run `msg` to completion without stopping at breakpoint 8.

Exercise:

Disable breakpoint 8 (set in section "Attaching an Ignore Count to a Breakpoint - GUI" on page 5-25).

Solution:

You should click on the Eventpoint menu. Select Summarize/Change....

NightView displays the debug eventpoint summarize/change dialog box.

Exercise:

Read about the Disable button in "Debug Eventpoint Summarize/Change Dialog Box" on page 9-42. Use it to disable the breakpoint.

Solution:

Select breakpoint 8 from the eventpoint list, and click on the Disable button.

The eventpoint list shows that breakpoint 8 is disabled. NightView also displays the following message below the eventpoint list: Disabled 1 eventpoint: 8.

Exercise:

Make the debug eventpoint summarize/change dialog box go away.

Solution:

In the debug eventpoint summarize/change dialog box, you should click on the Close button.

NightView closes the debug eventpoint summarize/change dialog box.

Examining Eventpoints - GUI

You want to examine the types, locations, and statuses of the eventpoints you have set in msg.

Exercise:

Change to group process mode.

Solution:

From the debug menu bar, you should select Group Process Mode from the View menu. The debug qualifier area displays [Group Mode]. NightView displays this message in the debug message area: Changed to group process mode.

NightView displays in the debug message area:

```
Process local:13504 received SIGUSR1
Process local:13504 received SIGUSR1
Process local:13504 received SIGUSR1
Process local:13504 received SIGUSR1
Process local:13504 received SIGUSR1
```

The lines that mention signal SIGUSR1 appear because the **handle** command is implicitly set to **print** and explicitly set to nostop. These messages were saved while the Debug Window was in single process mode with another process as the currently displayed process; now that the Debug Window is in group process mode, messages from all processes are displayed, including any saved messages.

Exercise:

Examine all eventpoints.

Solution:

You should click on the Eventpoint menu. Select Summarize/Change....

NightView displays the debug eventpoint summarize/change dialog box.

For the C program, NightView displays in the eventpoint list:

```
EvptID   Type   Enabled     Process - Address
------   ----   --------    ----------------------
     2    B     Enabled     local:15625 at main.c:25
     3    B     Enabled     local:15625 at main.c:30
     7    B     Enabled     local:15625 at parent.c:16
     8    B     Disabled    local:15625 at parent.c:16
     9    P     Enabled     local:15625 at parent.c:16
     4    B     Enabled     local:13504 at main.c:18
     5    B     Enabled     local:13504 at main.c:25
     6    B     Enabled     local:13504 at main.c:30
```

For the Fortran program, NightView displays in the eventpoint list:

```
EvptID   Type   Enabled     Process - Address
------   ----   --------    ----------------------
     2    B     Enabled     local:15625 at main.f:21
     3    B     Enabled     local:15625 at main.f:23
     7    B     Enabled     local:15625 at parent.f:16
     8    B     Disabled    local:15625 at parent.f:16
     9    P     Enabled     local:15625 at parent.f:16
     4    B     Enabled     local:13504 at main.f:21
     5    B     Enabled     local:13504 at main.f:23
     6    B     Enabled     local:13504 at main.f:15
```

For the Ada program, NightView displays in the eventpoint list:

```
EvptID   Type   Enabled     Process - Address
------   ----   --------    ----------------------
     2    B     Enabled     local:15625 at main.a:25
     3    B     Enabled     local:15625 at main.a:27
     7    B     Enabled     local:15625 at parent.a:15
     8    B     Disabled    local:15625 at parent.a:15
     9    P     Enabled     local:15625 at parent.a:15
     4    B     Enabled     local:13504 at main.a:25
     5    B     Enabled     local:13504 at main.a:27
     6    B     Enabled     local:13504 at main.a:18
```

NightView displays all eventpoints for process `local:15625` followed by the eventpoints for process `local:13504`.

Breakpoints 1, 2, and 3 were set in "Setting the First Breakpoints - GUI" on page 5-9. Breakpoint 1 has no entry because it was deleted in "Removing a Breakpoint - GUI" on page 5-23. Breakpoints 2 and 3 are still enabled.

When the child process was forked, it inherited the parent process's breakpoints. The child's breakpoints 4, 5, and 6 correspond to the parent's breakpoints 1, 2, and 3. The order of the eventpoint numbers for inherited eventpoints is not necessarily the same as in the parent.

Breakpoint 7 was set in “Setting Conditional Breakpoints - GUI” on page 5-24 and is still enabled.

Breakpoint 8 was set in “Attaching an Ignore Count to a Breakpoint - GUI” on page 5-25 and was disabled in “Disabling a Breakpoint - GUI” on page 5-30.

Patchpoint 9 was set in “Patching Your Program - GUI” on page 5-29 and is still enabled.

Exercise:

Make the debug eventpoint summarize/change dialog box go away.

Solution:

In the debug eventpoint summarize/change dialog box, you should click on the Close button.

NightView closes the debug eventpoint summarize/change dialog box.

Exercise:

Change back to single process mode and make sure the **parent** process is the currently displayed process.

Solution:

From the debug menu bar, you should select Single Process Mode from the View menu. The debug qualifier area displays the qualifier for the currently selected process. NightView displays these messages in the debug message area:

```
Changed to single process mode.
Switched to process local:15625.
```

If the parent process is not the currently displayed process, you should switch to it.

Continuing to Completion - GUI

There’s nothing else to look at, so you decide to run `msg` to completion.

Exercise:

Continue execution of `msg`.

Solution:

In the Debug Window, you should click on the Resume button.

NightView displays in the dialogue I/O area:

```
6.  Parent sleeping for 1 seconds
7.  Parent sleeping for 2 seconds
Child got ordinal signal #6
8.  Parent sleeping for 3 seconds
Child got ordinal signal #7
9.  Parent sleeping for 1 seconds
Child got ordinal signal #8
10.  Parent sleeping for 2 seconds
Child got ordinal signal #9
Child got ordinal signal #10
```

The source code is written so that the lines that begin with a number come from the parent process, and the lines that begin with the word "Child" come from the child process. Note that the sleep interval varies from 1 through 3 because of the patch you made in "Patching Your Program - GUI" on page 5-29.

Note the order of the displayed lines may vary.

The debug source display shows the main program, at the call to `exit`.

The debug status area and the debug group area show the status About to exit. This means that the process has called the exit system service. See "Exited and Terminated Processes" on page 3-16. The debug group area shows the same status for the child.

For the C and Ada programs, NightView displays in the debug message area:

```
Process local:15625 is about to exit normally
--> Undisplayed items:
  1:  (print) sig_ct
```

The last two lines say that `sig_ct` is not displayed. This message appears because of the **`display`** command and because the `sig_ct` variable is not visible at this point in the parent process.

For the Fortran program, the variable sig_ct is still available, so it is displayed:

```
1: sig_ct = 11
```

Leaving the Debugger - GUI

The tutorial is over.

Exercise:

Read about the Debug Window's NightView menu. Use it to leave the debugger.

Solution:

You should click on the NightView menu of any window. Select Exit (Quit NightView).

Neither process has completely exited, so NightView puts up a warning dialog box, asking the following question:

```
Kill all processes being debugged?
```

Exercise:

Make the processes go away.

Solution:

In the warning dialog box, you should click on the OK button.

All windows are removed.

6
Invoking NightView

This section describes how to start a NightView session.

You can start NightView without any arguments at all, if you wish. The arguments available on the NightView command line control the initial state of the debugger, and optionally allow you to specify the first program to be debugged. The command line to invoke NightView looks like this:

nview [-editor *program***] [-help] [-ktalk] [-nogui]**
[-noktalk] [-nolocal] [-nx] [-prompt *string***]**
[-safety *safe-mode***] [-simplescreen] [-version]**
[-*Xoption* **...] [-x** *command-file***] [-xeditor]**
[*program-name* **[***corefile-name***]]**

-editor *program*

Use *program* to edit source files. (See "Edit" on page 9-23.) This option is valid only in the graphical user interface.

-help

Causes NightView to print its command line syntax followed by a brief description of each option and then exit with code 0.

-ktalk

Allows NightView to communicate with other tools via KoalaTalk. (See "Using NightView with Other Tools" on page 3-35.) This is the default mode of operation. Use **-noktalk** to disable this mode. This option is valid only in the graphical user interface.

-nogui

Prevents NightView from automatically invoking the graphical user interface. See Chapter 9 [Graphical User Interface] on page 9-1.

-noktalk

Prevents NightView from being used as a debug server via KoalaTalk. (See "Using NightView with Other Tools" on page 3-35.) This option is valid only in the graphical user interface.

-nolocal

Prevents NightView from starting a dialogue on the local system. See "Dialogues" on page 3-4. In the graphical user interface, if **-nolocal** is used, NightView pops up a Remote Login Dialog Box (see "Remote Login Dialog Box" on page 9-45). (**-nolocal** is implied on Intel/Red Hat Linux)

`-nx`

Prevents NightView from reading commands from the default initialization file. See "Initialization Files" on page 3-32.

`-prompt` *string*

Sets NightView's initial prompt string to *string*. See "set-prompt" on page 7-47.

`-safety` *safe-mode*

Sets the initial safety level to *safe-mode*, which can be `forbid`, `verify`, or `unsafe`. The default level is `verify`. This controls the debugger's response to dangerous commands. See "set-safety" on page 7-49.

`-simplescreen`

Directs NightView to use a simple full-screen interface. This option implies **`-nogui`**. See Chapter 8 [Simple Full-Screen Interface] on page 8-1.

`-version`

Causes NightView to display its current version and then exit with code 0.

-*Xoption*

Any standard X Toolkit command line option (see **`X(1)`**). These options are allowed only when using the graphical user interface.

`-x` *command-file*

Directs NightView to read commands from *command-file* before reading commands from the default initialization file or from standard input. You may supply more than one **`-x`** option if you like; the files are read in the order of their appearance on the command line.

`-xeditor`

Use this option if the program specified by **`-editor`** communicates with X directly (see **`X(1)`**). For example, if **`nedit(1)`** is specified by the **`-editor`** option, you should specify **`-xeditor`**. However, if **`vi(1)`** is specified as the editor, you should not use this option because **`vi`** must run from within an **`xterm(1)`**. This option is valid only in the graphical user interface.

program-name

If no *corefile-name* argument is specified, then NightView will prompt you for arguments to supply to *program-name* and start debugging that program. If you inadvertently specified the *program-name* argument, you will have the opportunity to cancel its effect when you are prompted for arguments.

corefile-name

When you supply both *program-name* and *corefile-name* arguments, NightView starts out by creating a pseudo-process for the given core file, using the given *program-name* as the executable image for that core file. See "Core Files" on page 3-4 and "core-file" on page 7-34.

All options may be abbreviated to unique prefixes. For example,

```
nview -si
```

invokes NightView with the simple full-screen interface.

If the environment variable DISPLAY is set, or the standard X Toolkit command line option **-display** is used, then NightView communicates through a graphical user interface. In this case, other standard X Toolkit command line options are also allowed, e.g., **-xrm** *resourcestring*. See Chapter 9 [Graphical User Interface] on page 9-1.

NightView must be run with the Élan License Manager. If your site has multiple license servers, and you need to indicate a server on a particular system, you can set the environment variable POWERWORKS_ELMHOST to the name of the server's system before invoking NightView. For more information about the license manager, see the *Élan License Manager Release Notes*.

All NightView command line options are case-insensitive. However, note that X Toolkit options are case-sensitive.

When NightView starts execution, it first attempts to read commands from any files specified in **-x** options. It then looks for any initialization files to read (see “Initialization Files” on page 3-32), unless the **-nx** option was specified. When those files have all been processed, NightView reads commands from standard input until it encounters the end of the file or the **quit** command is executed (see “quit” on page 7-17).

7
Command-Line Interface

This chapter describes how to interact with NightView through commands.

In some cases, this may be your only means of directing the debugger's actions. If you are using the graphical user interface (see Chapter 9 [Graphical User Interface] on page 9-1), however, commands are only one of several ways to control the debugger and your programs.

Command Syntax

This section describes the general syntax and conventions of NightView commands. Most commands have three parts. A qualifier appears first (in parentheses) and is used to restrict the command to a certain set of processes or dialogues. Next comes the keyword indicating which command is to be executed. The command arguments follow as the third part. In general, you must separate syntactic items (like keywords and argument values) with white space, unless they are separated by punctuation characters. White space consists of one or more blank or tab characters. These rules may be different within expressions, where the rules of the programming language apply.

Some commands apply to individual processes, others apply to dialogues. The *qualifier* is a prefix that determines the dialogues and/or processes to which the following command applies. A qualifier is simply a list of dialogues and/or processes enclosed in parentheses. If a command applies only to dialogues, and the qualifier includes specific processes, the command applies to the dialogues containing the processes. If a command applies only to processes, but the qualifier includes dialogues, the command applies to all processes in those dialogues. If a command affects neither dialogues nor processes, the qualifier is ignored. You can set a default qualifier that will be applied when you don't provide one. For more information on the syntax and operation of qualifiers, see “Qualifier Specifiers” on page 7-10.

On startup, NightView provides you with a dialogue, `local`, for debugging on the local machine. The initial default qualifier is set to `all` to indicate all dialogues and processes.

After the qualifier, if any, all commands start with a *keyword*, which may be abbreviated to the shortest unambiguous prefix. Many frequently used commands also have special abbreviations. Most commands have one or more *arguments*; some arguments are also keywords, while others are information you supply. A keyword argument can usually be abbreviated if it is unambiguous; any exceptions to this rule are noted in the section describing the command. Both command and argument keywords are case-insensitive; they can be entered in either upper or lower case. You cannot abbreviate file names, symbolic names, or NightView construct names.

Commands are terminated by the end of the input line.

If you enter a line interactively consisting solely of a newline, NightView will usually repeat the previous command. This is explained more fully later; see "Repeating Commands" on page 7-15.

You can include comment lines with your commands. A comment line starts with the # character, which must be the first non-blank character on the line, and terminates at the end of the input line. Comments are most useful when you write debugger source files or macros (see "Defining and Using Macros" on page 7-134 and "source" on page 7-113).

NightView prompts you for input. The format of the prompt may be controlled by the **`set-prompt`** command (see "set-prompt" on page 7-47). The default prompt includes the names of all the dialogues in the default qualifier and looks like this:

```
(local)
```

Some NightView commands require multiple lines of input. For these commands, the command-line and simple full-screen interfaces change the prompt to `>` to remind you that you are entering a multi-line command.

```
>
```

To terminate NightView, use the **`quit`** command, which can be abbreviated **`q`** (see "quit" on page 7-17).

The subsections below explain some common syntactic constructs that are used in a variety of NightView commands.

Selecting Overloaded Entities

For general information about function and operator overloading, see "Overloading" on page 3-23.

The special overloading syntax used in both expressions and location specifiers is always introduced by a number sign character (#) used as a suffix directly following the entity (an operator in an expression or a function or procedure name). The # is followed by additional information indicating the specific kind of overload request. There are three forms of # syntax:

1. #?

   A number sign followed by a question mark is a query. It always makes the command it appears in fail, but the error message shows all the possible choices for overloading the name or operator (even if there is only 1 choice). The choices will be numbered starting at 1, and the number may be used to select the specific function.

2. ##

   Two number signs in a row act just as if **`set-overload`** were on for that one name. If there is only one possible choice, it is used; if there are multiple choices, the command fails and the error message shows the list.

3. #<digits>

   A number sign followed by a number is the syntax used to pick a specific overloaded function or operator from the list printed in the error message.

In C++, the function call and subscript operators don't appear in a single location, but are "spread out" with arguments or subscripts between the parenthesis or brackets. In these cases the final bracket or parenthesis is the character which should be suffixed with the #. For example:

```
function#5(12,3)
```

This picks the 5th instance of the name `function` from a list of overloaded functions.

```
object(12,3)#5
```

This, on the other hand, picks the 5th version of an overloaded `operator()` function call operator applied to the `object` variable.

The following example shows a use of the overloaded "+" operator in Ada. The `#?` is first used to do a query, then the desired operator is selected with `#1` when the expression is evaluated again.

```
(local) print a +#? b
Warning: local:5865 Cannot evaluate argument expression:
Reason follows [E-print_cmd007]
Unresolved overloaded functions or operators:

#1   native language operator +
#2   interval_timer.a:294
     FUNCTION "+"(l : IN time, r : IN time)
        RETURN time
#3   interval_timer.a:328
     FUNCTION "+"(l : IN time, r : IN integer)
        RETURN time
#4   interval_timer.a:375
     FUNCTION "+"(l : IN time, r : IN time)
        RETURN long_float
#5   interval_timer.a:391
     FUNCTION "+"(l : IN time, r : IN time)
        RETURN float
#6   interval_timer.a:407
     FUNCTION "+"(l : IN time, r : IN time)
        RETURN duration
(local) print a+#1 b
$1: a +#1 b = 11
```

The following example shows that the **`set-overload`** command may be used to turn on automatic overloading, in which case you will see the same error message without needing the `#?` syntax.

```
(local) set-overload operator=on
Overload mode set to operator=on routine=off
(local) print a + b
Warning: local:5865 Cannot evaluate argument expression:
```

```
Reason follows [E-print_cmd007]

#1   native language operator +
#2   interval_timer.a:294
     FUNCTION "+"(l : IN time, r : IN time)
        RETURN time
```

etc...

Overloaded procedures may also be referenced with similar syntax.

```
(local) set ada.text_io.put#?("Hello world")
Warning: local:5865 Unable to evaluate expression
" ada.text_io.put#?("Hello world")": Problem follows [E-
set_cmd007]
Unresolved overloaded functions or operators:
#1   phase2/predefined/text_io_b.pp:1247
     PROCEDURE text_io.put(file : IN file_ptr, item : IN
character)
#2   phase2/predefined/text_io_b.pp:1269
     PROCEDURE text_io.put(item : IN character)
#3   phase2/predefined/text_io_b.pp:1469
     PROCEDURE text_io.put(file : IN file_ptr, item : IN
string)
#4   phase2/predefined/text_io_b.pp:1491
     PROCEDURE text_io.put(item : IN string)
(local) set ada.text_io.put#4("Hello world")
```

Special Expression Syntax

For general information about expression evaluation, see "Expression Evaluation" on page 3-20. In addition to the standard language syntax, NightView offers a special syntax for referencing convenience variables and variables from other scopes or stack frames.

The special constructs all start with '$' as shown in the following table.

Table 7-1. Special '$' Constructs

$

A simple '$' by itself is a special convenience variable which always refers to the last value history entry (see "print" on page 7-66). See "Value History" on page 3-32.

$$

The name '$$' refers to the value history entry immediately prior to '$'. See "Value History" on page 3-32.

$*number*

A '$' followed by a number refers to that number entry in the value history. See "Value History" on page 3-32.

`${-`*number*`}`

A '$' followed by a negative number enclosed in braces refers to value history entries prior to the most recent one. '${-0}' is a complicated way to refer to the same thing as '$', and '${-1}' is the same as '$$'. This syntax is useful when you want to reference values farther back than -1. See "Value History" on page 3-32.

$*identifier*

This is the standard syntax for convenience variables. Many names are predefined (for instance, all the machine registers may be referenced using predefined convenience variables). See "Convenience Variables" on page 3-31, and "Predefined Convenience Variables" on page 7-6.

`${`*file*`:`*line expression*`}`

This syntax is used to evaluate the expression in the context specified by the given file and line number. This is most useful for referencing static variables which are not visible in the current context. If you reference a local stack or register variable from some other context, the results are not defined.

`${+`*number*`:`*routine expression*`}`

This syntax is used to go up the stack (see "up" on page 7-109) until you see *number* previous occurrences of *routine* relative to the current frame. (It does not matter what the current routine name is, this construct always backs up the frame first, then starts looking for frames associated with the given routine.) The given *expression* is then evaluated in that context. For example, '${+1:fred x}' refers to the variable named 'x' in the first routine named `fred` above the current routine.

`${+`*number expression*`}`

This syntax simply refers to previous stack frames, regardless of the routine name. The immediately previous frame is '+1'.

`${-`*number*`:`*routine expression*`}`

This syntax is useful only if you have changed your current frame with the **`up`** command. This allows you to refer to frames down the stack and is analogous to the version above which uses the '+' syntax.

`${-`*number expression*`}`

This is also analogous to the corresponding '+' syntax, but refers to frames down, rather than up the stack.

`${=`*number expression*`}`

This syntax evaluates the expression in the context of the given absolute frame number, regardless of the current frame. You can determine absolute frame numbers by using the **`backtrace`** command (see "backtrace" on page 7-65).

`${*`*frame-addr expression*`}`

This syntax uses *frame-addr*, which must be a numeric constant, as an absolute frame address. It evaluates *expression* in the context of this frame address, regard-

less of the current frame. If there is no frame on the current stack with this address, the results are undefined.

You may wish to use this form in **display** expressions (see "display" on page 7-72) to refer to a specific stack frame regardless of where it appears relative to the current frame. You can use the **info frame** command (see "info frame" on page 7-122) to get the frame address for any stack frame.

The above constructs may be used freely in any language expression. This means they may be nested (in case you want to do something like back up the stack frame, then shift to a different local scope in that routine). Because different frames may be associated with routines in different languages, the expressions evaluated in any given context may be expressions in different languages. This might not always make sense because different languages support different data types. If NightView cannot figure out how to evaluate a mixed language expression, it returns an error.

If you use any of these constructs in a conditional expression for an *inserted eventpoint* (breakpoint, agentpoint, monitorpoint, patchpoint or tracepoint), or in a monitorpoint, patchpoint or tracepoint expression, they are evaluated at the time you establish the expression, not when the expression is evaluated within the eventpoint. This is because the eventpoint expressions are compiled into your program by the debugger, and these constructs must be evaluated at compile time.

In the rare case of a user program which contains variables that have a '$' in their name, the user program variable is always referenced in preference to the convenience variable.

Predefined Convenience Variables

You may create any number of convenience variables simply by assigning values to new names, but some variables are predefined and have special values. The '$' and '$$' variables have already been documented (see "Special Expression Syntax" on page 7-4). The following special variables are all automatically defined on a per process basis.

Table 7-2. Predefined Convenience Variables

$_

This variable holds the address of the last item dumped with the **x** command (see "x" on page 7-68). It is also set by the eventpoint status commands to the address of the last eventpoint listed, and the **info line** command to the address of the first executable instruction in the line. If you were dumping words, it holds the address of the last word. If you were dumping bytes, it holds the address of the last byte, etc. See "x" on page 7-68, "info eventpoint" on page 7-115, "info breakpoint" on page 7-116, "info tracepoint" on page 7-117, "info patchpoint" on page 7-118, "info monitorpoint" on page 7-119, "info agentpoint" on page 7-120, and "info line" on page 7-133.

$__

This variable holds the contents of the last item printed by the x command. If you were dumping words, it holds the last word. If you were dumping bytes, it holds the last byte, etc.

`$pc`

This variable provides access to the program counter. This is a machine register, but every machine has a `$pc`, so this name is common to all machines. When a program is stopped, `$pc` is the location where it stopped. On any given machine, `$pc` may not map directly onto a specific machine register (RISC machines often have multiple program counters), but it always represents the address at which the program stopped. See “Program Counter” on page 3-24.

`$cpc`

`$cpc` is similar to `$pc`. In frame 0, if there are no hidden frames below frame 0 (because of uninteresting subprograms), `$cpc` has the same value as `$pc`. See “Interesting Subprograms” on page 3-27. In other frames (including frame 0 if there are hidden frames below it), `$cpc` is the address of the instruction that is currently executing. In most cases, this is the call instruction that caused the frame immediately below the current frame to be created. For the frame immediately above a signal-handler stack frame, `$cpc` is the address of the instruction that was executing when the signal occurred.

`$sp`

Most machines have a stack pointer. The stack pointer is always called `$sp`.

`$fp`

Most machines either have a frame pointer, or have an implicit frame pointer derived from information in the program. The `$fp` variable always represents the frame address (even if it is not a specific hardware register), and local variables are always described with some offset from the frame pointer (see “info address” on page 7-131).

`$is`

`$is` is defined when a watchpoint is triggered. See “Watchpoints” on page 3-11. `$is` is the value of the variable being watched after the instruction that causes the trigger has completed.

`$was`

`$was` is defined when a watchpoint is triggered, before the condition is evaluated. See “Watchpoints” on page 3-11. `$was` is the value of the variable being watched before the instruction that causes the trigger has begun.

PowerPC Registers

The PowerPC machines are based on the IBM/Motorola PowerPC 604™ architecture (see *PowerPC Microprocessor Family: The Programming Environments* for architectural details). See “info registers” on page 7-124.

In addition to the common register definitions for stack pointer, frame pointer, and program counter, the PowerPC machines support the registers shown in the following table.

Table 7-3. PowerPC Registers

`$r0` through `$r31`

These names map onto the 32 general purpose registers (note that `$sp` is the same as `$r1`, and `$fp` will typically be either `$r1` or `$r2`, depending on the kind of code generated by the compiler).

`$f0` through `$f31`

These names map onto the 32 floating-point registers. The floating point registers on the PowerPC always hold double precision format values.

`$lr`

The link register.

`$ctr`

The counter register.

`$cr`

The condition register.

`$crf0` through `$crf7`

These names map onto the eight individual condition fields contained in the condition register `$cr`.

`$fpscr`

The floating point status and condition register.

`$xer`

The integer exception register.

`$srr0` through `$srr1`

The exception state save and restore register. (The `$srr0` register is the same as the `$pc` register).

`$mq`

The `$mq` register does not really exist on the machine, and the compilers will not generate references to it, but for backward compatibility with older architectures, it is emulated by the operating system, and you can refer to it in the debugger.

`$dabr`

The data address breakpoint register. NightView uses this register to implement watchpoints. See "Watchpoints" on page 3-11. Users should not modify this register.

Note that the floating point registers are not normally displayed by the **`info registers`** command. If you want to display all the floating-point registers, you can do so with the following command:

```
info registers f.*
```

The Power Hawk 700 Series supports additional registers:

`$v0` through `$v31`

Vector registers. To change the value of these registers with the debugger, see "vector-set" on page 7-76.

`$vrsave`

This register describes which vector registers are in use by the program. This is used by the operating system when context-switching. Avoid modifying this register.

`$vscr`

Vector status and control register.

Location Specifiers

A *location-spec* is used in various commands to specify a location in the executable program. It can be any of the following:

function_name or *unit_name*[`'specification`|`'body`]

specifies the beginning of the named function or Ada unit. Note that `'specification` and `'body` are meaningful only with an Ada unit. If a unit name is specified and neither `'specification` nor `'body` are given, then `'body` is assumed. `'specification` and `'body` may be abbreviated to unique prefixes.

file_name`:`*line_number*

specifies the first instruction generated for the given line in the given file

file_name`:`*function_name*

specifies the beginning of the specified function declared in the given file (this is required for `static` functions that are not globally visible).

line_number

specifies the first instruction generated for the given line in the current file

line_number`:`*unit_name*[`'specification`|`'body`]

specifies an Ada unit name, which may be specified as a fully expanded unit name, preceded by the line number in the source file. If neither `'specification` nor `'body` are given, then `'body` is assumed. `'specification` and `'body` may be abbreviated to unique prefixes.

Note that when specifying a line number and a *unit name* as a location specifier that the line number comes *first*; whereas when specifying a *filename* with a line number, the line number is *last*.

**expression*

specifies the address given by *expression*

If a location specifier is omitted, then the location used is the next instruction to be executed in the current stack frame. See "Current Frame" on page 3-25.

Function names always refer to the location of the first instruction following any prologue code (the *prologue* is code that allocates local stack space, saves the return address, etc.). To refer to the actual entry point of a function, use the **expression* form and write an expression that evaluates to the function entry point address (in C language mode, this would look like `*&function`).

NOTE

A location specifier may sometimes designate multiple locations; for instance, a line number within an Ada procedure that has been expanded inline several times will designate every location where that procedure was expanded. If such a location specifier is used to set an eventpoint (see "Manipulating Eventpoints" on page 7-77), NightView will set the eventpoint at each of the corresponding locations. An eventpoint set at multiple locations is still considered to be a single eventpoint. If you wish to set an eventpoint at some subset of the locations that are implied by a particular location specifier, the **`info line`** command may be used to determine the locations corresponding to the particular location specifier. The `*expression` form of location specifier may then be used to designate the proper location.

Wherever a *file_name* appears, it may be enclosed in double quotes. This is necessary if the *file_name* contains special characters.

Wherever a function name appears in a location specifier, it may also appear with an overloading suffix to distinguish between multiple functions with the same name (see "Selecting Overloaded Entities" on page 7-2). The names of operator functions in Ada or C++ may also be used as function names. In Ada, the operator name must appear in double quotes, and in C++ the keyword `operator` should be followed by the operator name (the same syntax used to declare operator functions in the language). Because the function name form of operator functions is always used in location specifiers, the only **`set-overload`** mode which affects location specifiers is the *routine* mode (see "set-overload" on page 7-54).

All commands that accept a *location-spec* argument allow the keyword **`at`** to precede the *location-spec*. In most cases, the **`at`** keyword is optional, but a few commands require it to be present. The syntax of each command indicates whether the keyword is required or optional.

Qualifier Specifiers

Qualifiers are used to apply NightView commands to specific processes or dialogues. A

qualifier is simply a list of *qualifier specifiers*, each specifier representing one or more processes or dialogues. You can supply a qualifier explicitly, in parentheses as a prefix to the command, or implicitly, by using the **`set-qualifier`** command (see "set-qualifier" on page 7-46). In a prefix qualifier, the list of specifiers is separated by either blanks or tabs.

Each *qualifier specifier* can be any one of the following items:

family-name

> A name given by you to a set of processes, called a *family.* See "family" on page 7-40.

dialogue-name

> The name of a dialogue in your NightView session. This is usually the name of the system on which the dialogue is running, but you may also specify a different name (see "login" on page 7-18). In contexts where the qualifier is being used to specify a set of processes, a *dialogue-name* refers to all the processes being debugged in that dialogue.

PID

> The numeric value of the process ID of one of the processes being debugged by NightView. You can use this form only if the process ID is unique among all the processes being debugged. This may not be true if you have multiple dialogues, but it is always true if you have only one dialogue.

dialogue-name:PID

> This is how you specify a particular process when processes in different dialogues have the same process ID.

`all`

> This keyword designates all processes or dialogues known to NightView.

`auto`

> This keyword designates the one process that is currently stopped and has been stopped for the longest time. You may want to specify **`auto`** as your default qualifier if you want to work on only one process at a time (see "set-qualifier" on page 7-46). NightView gives you an error message if you use **`auto`** when there are no processes stopped.

Note that, because a qualifier specifier can be either a family name or a dialogue name, you cannot have a dialogue with the same name as a process family.

In general, the specifiers in a qualifier are not *evaluated* until a qualified command requests the information. A qualifier is evaluated when a command qualified by it needs the information; that is, when the command is applied to the processes or dialogues in the qualifier. Certain NightView commands ignore their qualifier, so they do not request evaluation of the specifiers in the qualifier. This has several effects on you:

- A *family-name* appearing in a qualifier may remain undefined until a command requires evaluation of the qualifier. You may also change the defini-

tion of a *family-name* currently in use in a qualifier; such a change will affect the next command that evaluates that qualifier.

- Evaluating a *dialogue-name* yields all the processes in the dialogue at the time of the evaluation. Since evaluation is generally delayed until the last possible moment, using a *dialogue-name* is usually a good way to reference all the currently-existing processes in a dialogue.
- The specifiers `all` and **`auto`** are evaluated at the time a command is actually executed.

Eventpoint Specifiers

Eventpoints may be grouped together and assigned a name (see "name" on page 7-78). In addition, the name '.' is a reserved name that always refers to the set of eventpoints most recently created by a single command. Eventpoint numbers and eventpoint names are the two different types of *eventpoint specifiers.* Eventpoint specifiers that refer only to breakpoints may also be called *breakpoint specifiers* (*tracepoint specifiers*, *patchpoint specifiers*, *agentpoint specifiers*, *monitorpoint specifiers*, and *watchpoint specifiers* are similarly defined).

Regular Expressions

A *regexp* is used by many of the commands to specify a pattern used to match against a set of names (like variable names or register names in the **`info`** commands). Regular expressions may be case-sensitive or case-insensitive depending on the **`set-search`** command (see "set-search" on page 7-54).

Regular expressions are similar to wildcard patterns, but are more flexible. Regular expressions and wildcard patterns are used for different things in the debugger (see "Wildcard Patterns" on page 7-14). The descriptions of the commands tell if they take a regular expression or a wildcard pattern.

The regular expression syntax recognized is similar to that recognized by many other common tools, but the details (as always) vary somewhat.

Table 7-4. Regular Expressions

`.`

A dot matches any character except a newline.

`*`

A star matches zero or more occurrences of the preceding regular expression. For example, **`.*`** matches zero or more of any character except a newline.

`+`

A plus matches one or more occurrences of the preceding regular expression.

`{m}`

Matches exactly *m* occurrences of the preceding regular expression.

`{m,}`

Matches *m* or more occurrences of the preceding regular expression.

`{m,n}`

Matches from *m* to *n* occurrences of the preceding regular expression.

`^`

A caret matches at the beginning of a string.

`$`

A dollar sign matches at the end of a string.

`( )`

Parentheses are used to group regular expressions.

`[ ]`

Brackets define a set of characters, any one of which will match. Within the brackets, additional special characters are recognized:

^

If the first character inside the brackets is a caret, then the set of characters matched will be the inverse of the set specified by the remaining characters in the brackets.

-

A range of characters may be matched by specifying the starting and ending characters in the range separated by a dash.

To define a set that includes a **-** character, specify the dash as the first or last character in the set.

Any other character matches itself.

To literally match one of the special characters defined above, use a backslash (\) character in front of it (to literally match a backslash, use two of them (\\)).

The *m* and *n* match counts above must be positive integers less than 256.

Most commands that use regular expressions do not require the use of '^' and '$' because they implicitly assume that an *anchored* match is called for. Other commands (such as the **forward-search** and **reverse-search** commands) assume that only a partial match is called for (and does not imply an *anchored* match). The description of each command that uses regular expressions specifies whether or not it implicitly assumes its regular expressions are to be anchored.

If you do not need the full expressive power of regular expressions, you can just use a

normal string.

Examples:

```
r[1-5]
```

This example matches the strings 'r1', 'r2', 'r3', 'r4', and 'r5'. This might be a good expression to match register names.

```
child_pid
```

This example matches only the string 'child_pid'. This might be a good expression to match a program variable name.

Wildcard Patterns

Wildcard patterns are used by the commands **debug**, **nodebug** and **on program**. See "debug" on page 7-20, "nodebug" on page 7-20, and "on program" on page 7-36.

Wildcard patterns are similar to regular expressions, but are usually more convenient for representing file names. See "Regular Expressions" on page 7-12.

If the wildcard pattern starts with a `/`, it is assumed to be a pattern that must match a complete absolute path name. Otherwise the pattern is matched against the rightmost (trailing) components of the program name. Patterns are always matched to component boundaries. Spaces and tabs are not allowed in wildcard patterns.

Wildcards are similar to wildcards in **sh**.

Table 7-5. Wildcard Patterns

`*`

Matches zero or more characters (but does not match a `/`).

`{[`*chars*`]}`

Matches any of the characters in the set. A dash (`-`) can be used to separate a range of characters and a leading bang (`!`) matches any characters except the ones in the set (but not a `/`).

`?`

Matches any single character (except a `/`).

Any other character matches itself.

Unlike **sh**, a **`*`** matches a leading dot (`.`) in a file name.

If you do not need the full expressive power of wildcards, you can just use the file name.

Examples:

```
/bin/*
```

This matches any file in the directory **/bin**.

```
test*
```

This matches any file that begins with the letters **test**, in any directory.

```
*.c
```

This matches any source file that ends with **.c**, in any directory. This might be a good expression to match file names.

```
/usr/bob/myprog
```

This matches only the file **/usr/bob/myprog**.

Repeating Commands

A line typed from an interactive terminal consisting solely of a newline (no other characters, including blanks) generally causes NightView to repeat the previous command. Note that the blank line must come from an interactive device; a blank line in a macro or in a disk file read by the **source** command does not cause repetition. The command that gets repeated may, however, come from a macro.

Not all commands can be repeated in this manner. In general, commands whose result would not be any different when repeated will not repeat. Typing a blank line after a non-repeating command has no effect; it acts the same as a comment. If the description of a command does not say it is repeatable, then it isn't.

A few commands, such as **list** or **x**, alter their behavior slightly when repeated: instead of exactly repeating the command, they typically repeat the action on a different set of data. These differences in behavior are documented in the description of the command.

In the following examples, assume all commands were entered interactively.

```
(local) list  func:20
(local)
(local)
```

In this example, lines 16-25 (approximately) of function **func** would be listed by the **list** command. Repeating this command lists the next set of 10 lines, lines 26-35. Note that **list** is one of the commands whose behavior changes when it is repeated.

```
(local) define mac(ln) as
>       list func:@ln
>       end define
(local) @mac(20)
(local)
(local)
```

This example is equivalent to the previous one. It demonstrates that the repeated command may come from a macro.

```
(local) define mac(vn) as
>          x/20x @vn
>          echo
>       end define
(local) @mac(xstruct)
(local)
(local)
```

This example demonstrates how to write a macro that does not repeat at all. Since **`echo`** is a non-repeating command, entering a blank line after the **`@mac(xstruct)`** line does nothing.

Replying to Debugger Questions

This section describes how to respond when the debugger asks you a question.

Certain forms of some debugger commands are considered unsafe and will check the debugger's safety-level (see "set-safety" on page 7-49) before executing. When the safety-level is `verify`, these commands will ask a question of the user and wait for verification. The possible responses to the question are always "yes" and "no" (case insensitive). These responses may be abbreviated to their first letter if desired. The response must be terminated by a carriage return.

A "yes" response indicates that the unsafe action is to be performed.

A "no" response indicates that the unsafe action is *not* to be performed.

In the graphical user interface, the debugger pops up a warning dialog box. See "Warning Dialog Box" on page 9-15.

Controlling the Debugger

This section describes how to exit NightView, and the commands used to control debugged processes and your interaction with them.

Quitting NightView

quit

Stop everything. Exit the debugger.

`quit`

Abbreviation: **`q`**

This command terminates the debugger. If the safety level (see "set-safety" on page 7-49) is `forbid`, you will not be allowed to quit unless there are no processes being debugged. In other safety levels, any active processes will be killed when you quit. If the safety level is `verify`, you will be prompted for confirmation before quitting causes any debugged processes to be killed (see "Replying to Debugger Questions" on page 7-16).

The processes killed include all active processes started in any dialogue shell and not explicitly detached. NightView detaches from any processes that are being controlled but are not being debugged by you because of a **nodebug** command. See "Detaching" on page 3-3. See "nodebug" on page 7-20.

Processes started using the **`shell`** command are independent of the debugger, and are not affected by a **`quit`**.

Managing Dialogues

A *dialogue* is an interaction with a particular host system for the purpose of debugging one or more processes on that system under a particular user name. You may have as many dialogues as you wish; there can even be more than one dialogue with a particular host system. Dialogues are described in more detail in the Concepts chapter (see "Dialogues" on page 3-4).

login

Login to a new dialogue shell.

login [/conditional] [/popup] [name=*dialogue name*]
[user=*login name*] [*others ...*] *machine*

NOTE

If present, the options /conditional and /popup must appear before the machine name and before any keywords.

The **login** command takes many keyword parameters. The most commonly used are:

/conditional

Ignore this **login** command if a dialogue with this name already exists. This is useful from macros (see "Defining and Using Macros" on page 7-134) and for other programs that communicate with NightView.

/popup

Pop up the Remote Login Dialog Box (see "Remote Login Dialog Box" on page 9-45) initialized with the machine name and the values of the name= and user= keywords. No other keywords are allowed with this option. This option is meaningful only in the graphical user interface.

name=*dialogue name*

Give this parameter to specify a name for the dialogue you are creating. If you leave it off, the dialogue name is the same as the name of the machine running the dialogue. To run multiple dialogue shells on the same machine you must give them unique names. No dialogue name may be the same as a family name (see "family" on page 7-40). A dialogue name must start with an alphabetic character and may be followed by any number of alphabetic, numeric, or underscore characters.

user=*login name*

Login as this user. Normally your current user name is used, but you may login as any user.

machine

Specify the machine where the programs to be debugged are located and the dialogue shell will run. This is a required parameter. It may be a host name, with or without domain qualification, or it may be an IP address.

The following parameters are less frequently used, but are provided to allow you to control the execution environment of the remote dialogue.

`nice=`*nice value*

The dialogue normally runs with normal interactive priority. A positive nice value lowers the priority (makes other programs seem more important). You must have special privileges to specify a negative nice value.

`cpu=`*cpu list*

Set the CPU bias for the dialogue.

`memory=`*flags*

Control what sort of memory (local or global) will be used for the dialogue.

`priority=`*value*

Specify the priority of the remote dialogue processes. The scheduling policy determines what values may be specified for the priority.

`scheduling=`*sched_keywords*

Control the scheduling policy that will be used for the dialogue. The allowed keywords are: **`sched_fifo`**, **`fifo`**, **`sched_rr`**, **`rr`**, **`sched_other`**, and **`other`**.

`quantum=`*time*

Control the time slice quantum size for the process.

The `cpu`, `memory`, `scheduling`, `priority`, and `quantum` parameters all accept the same arguments as the corresponding options on the **`run(1)`** command — see the man page for details.

Any programs started in the dialogue shell will inherit all the above parameters. The **`run(1)`** command can control all these parameters, and may be used within the dialogue shell to debug programs and change the parameters.

When you use the **`login`** command you are asked for a password. See “Remote Dialogues” on page 3-6 for a general discussion of how to use remote dialogues.

Example:

```
(afamily) login fred
To begin a remote debug session on 'fred', enter the
password for user 'wilma'.
Password:  enter wilma's password
(afamily) login user=barney name=fredII fred
To begin a remote debug session on 'fred', enter the
```

```
password for user 'barney'.
Password:  enter barney's password
(afamily)
```

The above example shows the creation of two new dialogues. The first **login** command starts a dialogue on a machine named `fred` and logs in as the current user (`wilma` in this example). This dialogue is named `fred`, because no explicit name was given.

The second creates a dialogue on machine `fred` named `fredII`. In this case the user logged into `fred` is `barney`.

The **login** command is creating a new dialogue, so the qualifier has no effect on this command.

debug

Specify names for programs you wish to debug.

debug *pattern* `...`

pattern

A wildcard pattern matching the name of a program to be debugged. Spaces and tabs are not allowed in *pattern*. See "Wildcard Patterns" on page 7-14.

This command and its inverse (see "nodebug" on page 7-20) allow you to control which programs get debugged. The list of programs applies to the individual dialogues specified in the **debug** command qualifier (different dialogues may have different lists of programs to be debugged).

The **debug** and **nodebug** commands work by remembering the list of **debug** and **nodebug** commands. When a new file needs to be checked to see if it should be debugged, the name is first compared to the pattern in the most recent command, then the pattern in the next most recent command, and so on.

The first pattern that matches the file name determines what to do with the associated process. If the matching pattern is on a **debug** command, then the process will be debugged. If it was on a **nodebug** command, then the process will not be debugged.

The pattern ***** matches everything, so the list of patterns is always reset when ***** appears as an argument. Since each dialogue always starts with either **debug *** or **nodebug *** in the list, it is impossible to pick a file name that does not match at some point in the list.

The default pattern list for a dialogue is:

```
nodebug /usr/ccs/lib/* /usr/ccs/bin/* /sbin/* /usr/sbin/* /bin/*
       /usr/bin/* /usr/ucb/* /usr/bin/X11/* /usr/lib/*
debug *
```

To print the list of **debug** and **nodebug** patterns, see "info dialogue" on page 7-126.

nodebug

Specify names for programs you do not wish to debug.

nodebug *pattern* **...**

pattern

A wildcard pattern matching the name of a program to avoid debugging.

This command is typically used in combination with the **debug** command to control which programs are debugged in a dialogue. The complete syntax of wildcards and the algorithm used to match files is described in the **debug** command (see "debug" on page 7-20).

Example:

```
(afamily) nodebug *
(afamily) debug x*
```

This example uses **nodebug *** to turn off all debugging. It then uses **debug** to turn on debugging for any programs started where the basename begins with the letter **x**.

Note that even if one command is not debugged, its children may be debugged. To avoid debugging a command as well as any children, you must use the **detach** command (see "detach" on page 7-32).

To print the list of **debug** and **nodebug** patterns, see "info dialogue" on page 7-126.

translate-object-file

Translate object filenames for a remote dialogue.

translate-object-file [*from* [*to*]]

Abbreviation: **xl**

from

The filename or filename prefix as seen by the remote system.

to

The filename or filename prefix as seen by the local system.

If both *from* and *to* are present, a translation is added. If only *from* is present, the translation exactly matching *from* is removed. If neither is present, all translations are removed.

NOTE

from and *to* are *not* wildcard patterns or regular expressions. See "Wildcard Patterns" on page 7-14. See "Regular Expressions" on page 7-12.

The **translate-object-file** command manages translations for object filenames

for each dialogue in the qualifier. Translations are useful when:

- An object file is visible from both systems, but its position in the file system is different. For example, **/usr** on system `fred` may be mounted as **/fred/usr** on the local system.
- An object file is not visible from the local system, but you have a copy of the file. For example, you might have a development directory from which the image on the remote system is created.
- The object file on the remote system has been stripped, but you have a copy with debugging information.

Object filenames from **exec-file** and **load** commands are subject to object filename translation. See "exec-file" on page 7-35. See "load" on page 7-75. Dynamic library names are also subject to object filename translation. See "Debugging with Shared Libraries" on page 3-37. Object filenames from **symbol-file** commands are *not* subject to object filename translation. See "symbol-file" on page 7-33.

NightView attempts to match translations to the initial characters of the filename. Filename component boundaries are not treated as a special case. If you want to match to component boundaries, include slashes in the strings. NightView tries all translations that match the strings, beginning with the longest matching translation, until it finds a translated filename with the same text segment contents as the executing program. If no file is found with the same text segment contents, NightView gives a warning and uses the first translation that matched the object filename.

If an **exec-file** command fails because you don't have any translations or the translations are wrong, you can re-issue the **exec-file** command again after fixing the translations.

NightView automatically supplies a default set of translations when a remote dialogue is created. The default set is made by inspecting the local system mount table and by considering the set of cross-development environments on the local system. In many cases, these translations are sufficient; additional translations are not necessary.

Suppose the object files that exist on the remote system under the directory **/wilma/pebbles** exist on the local system under the directory **pebbles** (relative to your current working directory).

Examples:

```
(fred) xl /wilma/pebbles/ pebbles/
```

This command translates any object filename beginning with the string **/wilma/pebbles/** to the same filename with **/wilma/pebbles/** replaced by **pebbles/**. For example, **/wilma/pebbles/hair** becomes **pebbles/hair**. Note that **pebbles/hair** will be evaluated relative to NightView's current working directory. See "pwd" on page 7-56.

Suppose the object files that exist on the remote system under **/betty** exist on the local system under **/barney**. However, the files under **/betty** whose name begins with **bam** should be found under **/dino**.

```
(fred) xl /betty/ /barney/
(fred) xl /betty/bam /dino/bam
```

These commands translate any object filename beginning with the string **/betty/** to the same filename with **/betty/** replaced by **/barney/** and any object filename beginning with the string **/betty/bam** to the same filename with **/betty/bam** replaced by **/dino/bam**. NightView picks **/betty/bam** in preference to **/betty/** because **/betty/bam** is longer. For example,

/betty/dress becomes **/barney/dress**
/betty/bambam becomes **/dino/bambam**
/betty/bambino becomes **/dino/bambino**

A good place to put a **translate-object-file** command is in an **on dialogue** command in your **.NightViewrc** file. See "on dialogue" on page 7-24. Also, see "Initialization Files" on page 3-32.

Example:

```
(all) on dialogue fred.* do
>        xl /usr/ /fred/usr/
>     end on dialogue
```

This command translates the directory **/usr** on the remote system to the directory **/usr/fred** on the local system, for dialogues whose name begins with **fred**.

logout

Terminate a dialogue.

logout

The **logout** command terminates any dialogues named in the command qualifier. If your safety-level is `unsafe` then *all* processes being debugged in the dialogues are killed (see "set-safety" on page 7-49). If your safety-level is `verify` then you are prompted for confirmation before the logout causes any debugged processes to be killed (see "Replying to Debugger Questions" on page 7-16). If your safety-level is `forbid`, then the logout does not occur. If you want any processes to continue running, you must **detach** them prior to using **logout** (see "detach" on page 7-32). NightView detaches from any processes that are being controlled but are not being debugged by you because of a **nodebug** command. See "Detaching" on page 3-3. Also, see "nodebug" on page 7-20.

If the dialogue shell is still running at **logout** time, it is killed (you may send an exit command to the shell to terminate it normally prior to logging out).

Example:

```
(adialogue) detach
(adialogue) !exit
(adialogue) logout
```

The example shows how to avoid having any processes killed. The **detach** command allows all processes in the dialogue to continue running independently of the debugger. The **!exit** command sends an exit command to the dialogue shell to terminate it normally, then the **logout** command terminates the debugger dialogue.

on dialogue

Specify debugger commands to be executed when a dialogue is created.

`on dialogue` [*regexp*]

`on dialogue` *regexp command*

`on dialogue` *regexp* **`do`**

regexp

A regular expression to match against the names of newly created dialogues. See "Regular Expressions" on page 7-12.

command

A debugger command to be executed when a new dialogue whose name matches *regexp* is created.

In the third form of the **`on dialogue`** command, the debugger commands to be executed must begin on the line following the **`do`** keyword. The list of debugger commands to execute is terminated when a line containing only the words **`end on dialogue`** is encountered.

The **`on dialogue`** command allows a user-specified sequence of one or more debugger commands to be executed immediately after creating a new dialogue within NightView. When a new dialogue is created, the list of all **`on dialogue`** regular expressions is checked to see if any of them match the name of the new dialogue. The most recently specified **`on dialogue`** command whose regular expression matches the dialogue name will have its commands executed.

In its first form (given only a regular expression), the **`on dialogue`** command will remove any commands that were associated with the given regular expression. If no regular expression is given, then *all* previously defined **`on dialogue`** commands are removed. If your safety level is set to `forbid`, you are not allowed to remove all **`on dialogue`** commands. If your safety level is set to `verify`, NightView requests verification before removing all **`on dialogue`** commands. See "set-safety" on page 7-49.

In its second and third forms, the **`on dialogue`** command will associate a sequence of one or more user-specified debugger commands with the given regular expression. Macro invocations are *not* expanded when reading the commands to associate with the regular expression.

If dialogue `local` is started up automatically by NightView, then it will exist *before* any commands in your `.NightViewrc` file are read. In this case, NightView automatically runs the **`on dialogue`** command after all the initialization files have been processed. See "apply on dialogue" on page 7-25. See "Initialization Files" on page 3-32.

The default qualifier for all commands associated with the given regular expression will be the newly created dialogue.

The commands specified by **`on dialogue`** are event-triggered commands: they have an implied safety level (which may be different from the safety level that was set using **`set-safety`**).

If you wish to list all **on dialogue** commands, or see which **on dialogue** commands would be executed for a particular dialogue name, you should use the **info on dialogue** command.

Example:

```
(local)on dialogue ben.*  nodebug /usr/bin/*
```

After issuing the above command, if we now create a new dialogue named `ben_hur`, then we will automatically set it up so that programs residing in the directory named **/usr/bin** are not debugged by NightView.

Now suppose we do the following:

```
(local) on dialogue .*jerry do
>           nodebug /usr/remote/*
>           nodebug /usr/local/*
> end on dialogue
```

At this point, if we create another dialogue named `ben_n_jerry`, then this newly created dialogue will automatically be set up so that programs residing in the directories **/usr/remote** and **/usr/local** are not debugged by NightView. Note that even though the name `ben_n_jerry` also matches the regular expression `ben.*`, this dialogue *will* try to debug programs that reside in the directory **/usr/bin**. This is because **on dialogue** regular expressions are matched in reverse-chronological order (most recent first), and only the first match found is used.

```
(local) info on dialogue ben_n_jerry
on dialogue .*jerry do
          nodebug /usr/remote/*
          nodebug /usr/local/*
end on dialogue
```

If we were to now issue the command:

```
(local) on dialogue .*jerry
```

Then this would remove `.*jerry` (and its associated commands) from the debuggers **on dialogue** command list. Now, if we create yet another dialogue named `benny_and_jerry`, then this third dialogue will *not* automatically debug programs that reside in the directory **/usr/bin**, but it *will* debug programs that reside in **/usr/remote** and **/usr/local** (just like the first one did).

```
(local) info on dialogue benny_and_jerry
on dialogue ben.* do
nodebug /usr/bin/*
end on dialogue
```

apply on dialogue

Execute **on dialogue** commands for existing dialogues.

apply on dialogue

The **apply on dialogue** command allows **on dialogue** commands to be

executed for existing dialogues. See "on dialogue" on page 7-24. For each dialogue specified by the qualifier, the **`on dialogue`** commands which would match the name of the dialogue are immediately executed on behalf of the dialogue.

When the debugger automatically creates a `local` dialogue, it does an **`on dialogue`** command with a qualifier of `(local)` after processing all the initialization files. See "Initialization Files" on page 3-32. Because dialogue `local` exists *before* the customization commands in the user's **`.NightViewrc`** file are interpreted by the debugger, the **`on dialogue`** command by itself cannot initialize the environment for dialogue local (since it only applies to dialogues that will be created *after* the **`apply on dialogue`** command is issued). The automatic **`on dialogue`** executes any **`on dialogue`** commands that refer to dialogue `local`.

Dialogue Input and Output

Because each dialogue is a separate shell, each dialogue has its own input and output streams. NightView has several options for sending input to dialogues and managing the output data generated by the dialogue shell and the programs being run within it.

!

Pass input to a dialogue.

! [*input line*]

input line

If *input line* is specified, it is passed to the dialogue (or dialogues) determined by the command qualifier.

If *input line* is not specified, then this command switches to a special dialogue input mode.

If the qualifier for this command specifies more than one dialogue, then the same input data is sent to all the dialogues. This can make sense if you are doing something like debugging two versions of the same program and you want to see where they diverge. It is up to you to insure that the input is sensible to all the dialogues (or that the command qualifier only refers to one dialogue).

When you use the **!** command without an *input line* argument to switch to dialogue input mode, everything you type goes to the specified dialogues. Nothing you type is treated as a debugger command until a special terminator string is recognized. The default terminator string is ``-." (note that this is not the same as the ``~." used by **rlogin(1)** or **cu(1)**). See "set-terminator" on page 7-48, for information on how to change the terminator string.

The **!** command without an *input line* argument cannot be used inside a macro (see "Defining and Using Macros" on page 7-134), nor can it be used in the graphical or full-screen user interfaces.

Macros are *not* expanded when reading the input (or arguments) to this command.

This command does not care if it is talking to the dialogue shell or to a program running in the shell. If you start a program that requests input, you can pass the input to it using this command.

See "Repeating Commands" on page 7-15.

Example:

```
(afamily) !pwd
(afamily) !
PATH=/extra/progs:$PATH
ulimit -m 200
ulimit -d 100
ulimit -s 100
```

```
-.
(afamily)
```

The first line just sends a **pwd** command to the dialogue. The second switches to dialogue input mode and then several lines of input are sent directly to the dialogue to set up environment variables and limits on the amount of memory subsequent processes will be allowed to use. The final "**-.**" switches back to normal command input mode.

Note that if you just want to send a program name to the shell and wait for that program to start, you may want to use the **run** command instead. See "run" on page 7-30.

set-show

Control where dialogue output goes.

set-show [silent | notify=*mode* | continuous=*mode*]
[log[=*filename*]] [buffer=*number*]

silent

Just buffer the dialogue output, do not display it. The **show** command may be used to see what has accumulated (see "show" on page 7-29).

notify=*mode*

Do not display the dialogue output, but do print a notice when output first becomes available.

continuous=*mode*

Display dialogue output when it is generated.

The notify and continuous modes both accept one of the following keyword arguments:

immediate

In immediate mode the notification or actual output is displayed as soon as output becomes available.

atprompt

In the atprompt mode, the output is displayed only when the debugger is not requesting input. This is typically immediately prior to printing a new prompt to request additional commands, but it also prints output when the debugger is waiting for some event and has not yet prompted for new input.

Additional parameters on the **set-show** command control logging to a file and the size of the internal buffer.

log[=*filename*]

The log parameter without the =*filename* option turns off logging to a file and resumes buffering a limited amount of output in memory. When a file name is

specified, the output from the dialogue is logged to that file until the log parameter is changed.

`buffer=`*number*

The `buffer` parameter is used to set the size of the buffer holding all the most recent output from the dialogue. The default size is 10240 (10K bytes). When the buffer fills up, the oldest output is discarded. When logging to a file, this parameter does not have any effect — a log file may grow until disk space is exhausted.

This command only logs the output from dialogues. It does not log debugger commands, nor does it directly log the input to a dialogue; however, the input will normally be echoed by the system, so it will be logged as output from the dialogue.

To log the entire debug session, see "set-log" on page 7-44.

Each dialogue starts off in the default mode:

```
(all) set-show buffer=10240 continuous=atprompt
```

show

Control dialogue output.

`show` [*number* | `all` | `none`] [| *shell-command*]

number

The number of old output lines you wish to see again.

`all`

Specifying `all` instead of a number means show all the buffered output from the dialogue shell.

`none`

The `none` keyword is used to tell the debugger you are not interested in any of the buffered output. It pretends you have already seen any data currently in the buffer.

| *shell-command*

You may use a vertical bar (shell pipe operator) to request the output be sent to an arbitrary shell command, rather than being displayed. You may use this to run the output through a pager or filter of some kind.

The debugger always internally buffers output generated by dialogues. The **`show`** command displays any buffered output from a dialogue which you have not yet seen. The *number* or `all` arguments tell the debugger to display that many lines of previous output in addition to the new output (so the total number of lines displayed may be greater than *number*). The **`set-show`** command is used to control when dialogue output is printed without a specific request via the **`show`** command (see "set-show" on page 7-28).

Managing Processes

run

Run a program in a dialogue and wait for NightView to start debugging it.

run *input line*

input line

The shell command that will start a program (or programs) to debug.

This command is very similar to the **!** command (see "!" on page 7-27): it sends the specified *input line* to the dialogue shell (or shells) specified by the qualifier. The difference between **run** and **!** is that **run** waits for a new process to be debugged in one of the dialogues specified by the qualifier.

NOTE

Even if the qualifier specifies multiple dialogues, the **run** command terminates as soon as one new process has started.

The **run** command does not check the given *input line* for validity; it simply passes it unchanged to the dialogue shell, just like the **!** command. If it does not start a new process to be debugged, then **run** will just continue waiting forever (or until you type <CONTROL C>). If you issue a **run** command that starts more than one program, **run** will only wait until one of them starts up and is noticed by NightView. The other programs will start up and be debugged, but you probably won't know about them until after you have entered the next command.

If you just want to send input to a program that is reading from the shell's input terminal, or you want to start up a program or programs without waiting for them, just use the **!** command.

If you want to run the same program again, use the **run** command again. See "Restarting a Program" on page 3-14. If you want multiple programs to run concurrently, end the shell commands with & (ampersand). (You can't do this if your program expects input from you.)

set-notify

Control how you are notified of events.

set-notify [silent | continuous=*mode*]

silent

Only report events when explicitly requested.

`continuous=`*mode*

Display events when they happen.

The `continuous` mode accepts one of the following keyword arguments:

`immediate`

In immediate mode the notification is displayed as soon as the event happens.

`atprompt`

In the atprompt mode, the notification is displayed only when the debugger is not requesting input. This is typically immediately prior to printing a new prompt to request additional commands, but it also prints notifications when the debugger is waiting for some event and has not yet prompted for new input.

This command controls how the debugger tells you what is happening to the processes you are debugging. Individual processes may be set to notify you in different ways (using the command qualifier).

Events that might cause notification include hitting a breakpoint or watchpoint, getting a signal (but see “handle” on page 7-105), or ’`exec`’ing a new program. New processes to be debugged also cause notification, but this notification is controlled by the notification setting of the parent of the new process. Processes created directly by the dialogue shell always cause notification in the default notify mode. When a process exits, you will be notified by the process' dialogue (but see “show” on page 7-29 and “set-show” on page 7-28).

The output generated by any commands attached to a breakpoint (or watchpoint) or any automatic display expressions is also controlled by **`set-notify`**. If you set notify mode to `silent` for a process, all debugger output associated with that process will be buffered up and saved until you ask to see it.

Any change to the notify mode of a process takes place immediately, so changing the mode from `silent` to `continuous` may also result in large amounts of accumulated event notifications and other buffered output being generated.

The **`notify`** command (see “notify” on page 7-31) can be used to explicitly request notification of any events that have been saved up (this is the only way to find out about events that have happened in a process where the notify mode is `silent`).

If no arguments are given to the `notify` command, then the current notify mode of each process in the qualifier is printed.

The default notify mode is:

```
(all) set-notify continuous=atprompt
```

notify

Ask about pending event notifications.

`notify`

If you have been suppressing event notification on certain processes (see "set-notify" on page 7-30), the `notify` command may be used to request any notifications that have not yet been printed. It only tells you about pending events in the processes specified by the command qualifier.

attach

Attach the debugger to a process that is already running.

`attach` `PID`

PID

The process ID of the running process.

This command allows a program to be debugged even if it was not started from a debugger dialogue shell (see "Attaching" on page 3-3). The qualifier on this command must specify a single dialogue indicating which machine is running the specified PID. An error is reported if the qualifier implies multiple dialogues. It is also an error to attempt to attach to a program already being debugged, or to attach any of the processes required to run the debugger.

Since the program to which you are attaching is already running independently of the debugger, you will not be able to send it input through the normal dialogue input mechanism (see "!" on page 7-27) or see the output it generates (the input and output for the process remain connected to the same streams they were connected to prior to the **`attach`**).

Once you attach to a process, any future children it forks will also be debugged. See "set-children" on page 7-41. Children created prior to the attach must be explicitly attached if you want to debug them.

See "Attach Permissions" on page 3-35 for a description of what processes you are allowed to attach.

detach

Stop debugging a list of processes.

`detach`

The **`detach`** command terminates the debugger's connection to all the processes named in the command qualifier. Any breakpoints, monitorpoints, or watchpoints set in those processes are removed, but patchpoints, tracepoints, and agentpoints remain if they are enabled when you execute the **`detach`** command. See "breakpoint" on page 7-79, "patchpoint" on page 7-80, "monitorpoint" on page 7-84, "agentpoint" on page 7-87, "tracepoint" on page 7-83, and "watchpoint" on page 7-95.

The processes are allowed to continue running normally and the debugger will not be notified of any subsequent events that occur in those processes. If any of the processes fork or exec new programs, the debugger will not see them.

When the safety level is `unsafe` (see "set-safety" on page 7-49), detaching a process

that was stopped while evaluating a debugger expression containing a function call aborts any expression evaluation in progress. This returns the process to the state it was in when you asked to evaluate the expression. At `verify` safety level, it asks first, and at safety level `forbid`, it refuses to let you detach the process.

For another way of avoiding debugging certain processes, see "nodebug" on page 7-20. Also, see "set-children" on page 7-41.

kill

Terminate a list of processes.

`kill`

The **`kill`** command terminates all the processes named in the command qualifier.

In the Graphical User Interface, if you use a 'Kill' button (as opposed to manually typing the **`kill`** command) the debugger will check your safety level (see "set-safety" on page 7-49) before permitting you to kill the desired processes. If your safety level is `forbid` then you will *not* be permitted to kill the selected processes. If your safety level is `verify` then you will be prompted for verification (see "Warning Dialog Box" on page 9-15). If your safety level is `unsafe` then the processes are terminated with no questions asked.

symbol-file

Establish the file containing symbolic information for a program.

`symbol-file` *program-name*

program-name

This must be the name of an executable file corresponding to the programs running in the specified processes. It should contain symbolic debug information for the program.

If *program-name* is a relative pathname, it is interpreted relative to NightView's current working directory.

program-name is *not* subject to object filename translations. See "translate-object-file" on page 7-21.

A *symbol file* is an executable file from which NightView obtains information about symbols in a program being debugged. Normally, the symbol file is the same as the program's executable file, but it may be different if, for example, you are debugging a stripped program (see **`strip(1)`**). In this case, you need to specify an unstripped version of the program in the **`symbol-file`** command, if you want to access information symbolically.

The **`symbol-file`** command is applied to each process in the qualifier. You should make sure that each of those processes is running the same program; otherwise, you may get unpredictable results from the debugger when you examine variables or memory.

Note: If you have not specified a symbol file for a process, NightView attempts to obtain the information from the executable file (see "exec-file" on page 7-35).

In some situations, an object filename translation is more appropriate than a **symbol-file** command. See "translate-object-file" on page 7-21.

core-file

Create a pseudo-process for debugging an aborted program's core image file.

core-file *corefile-name* [exec-file=*program-name*]

corefile-name

The name of a core file.

If *corefile-name* is a relative pathname, it is interpreted relative to NightView's current working directory.

exec-file=*program-name*

Specifies the name of the executable program that created the given core file.

If *program-name* is a relative pathname, it is interpreted relative to NightView's current working directory.

A *core file* is a copy of a process's memory made when a process is terminated abnormally. You can examine these core files using NightView by specifying the core file name in the **core-file** command. NightView responds with a process ID (PID) corresponding to a newly-created *pseudo-process*. This is not a real executing process; a pseudo-process is merely a mechanism for dealing with core files in NightView. The PID NightView assigns does not correspond to any running process, but you can use it in qualifiers, and you can also include it in process families using the **family** command. See "family" on page 7-40.

The qualifier for the **core-file** command is used only to determine with which dialogue the pseudo-process should be associated. (Among other things, this determines the type of machine that created the core file.) Thus, the qualifier should specify exactly one dialogue; otherwise, NightView issues an error message and refuses to honor the command.

If you specify the exec-file=*program-name* option, it is equivalent to executing an **exec-file** command (see "exec-file" on page 7-35) on the pseudo-process created by the **core-file** command. This is seldom required, since NightView attempts to determine the location of the executable program from information saved in the core file (see "Finding Your Program" on page 3-8). If NightView is unable to correctly determine the executable program, you will need to specify the exec-file=*program-name* option or use the **exec-file** command to specify the name of the executable program.

When debugging a core file, NightView uses the executable program file for two purposes. NightView uses this file to obtain symbolic information about variables and procedures in your program, just as it does when debugging normal processes. For core files, NightView also must use this file to obtain the contents of read-only memory, including the machine instructions of the program. If NightView is unable to locate the

executable program, then you will only be able to examine writable memory by absolute address. You can specify the file, or files, NightView should use by specifying the `exec-file=`*program-name* option or by using the **`exec-file`** and **`symbol-file`** commands (see "exec-file" on page 7-35 and "symbol-file" on page 7-33).

Note that, unlike other debuggers, NightView allows you to examine the core file of a process at the same time you are executing the program that produced the core file. This allows you to try executing your program again to try to find the problem, while still accessing information from the core file. For instance, you may find from the core file that a certain global variable has an incorrect value. You could then run the program again, stopping it at interesting points to check the value of that global variable. By using an appropriate qualifier, you can easily print out the values of variables in both the running program and the core file for easy comparison.

exec-file

Specify the location of the executable file corresponding to a process.

`exec-file` *program-name*

program-name

Specifies the file containing the executable program corresponding to the specified processes.

If *program-name* is a relative pathname, it is interpreted relative to Night-View's current working directory.

program-name is subject to object filename translations. See "translate-object-file" on page 7-21.

This command tells NightView where to find the executable file corresponding to the processes specified by the qualifier. Obviously, you should ensure that all those processes are, in fact, running the same program; otherwise, you may get strange behavior. (NOTE: NightView does not do this verification for you because the processes may be executing different copies of the same program on several different systems. NightView would not be able to tell that these were the same program.)

You usually use this command in conjunction with the **`core-file`** command (see "core-file" on page 7-34). You may also need to use it if NightView is unable to determine the executable file corresponding to a new process being debugged. See "Finding Your Program" on page 3-8.

If you do not explicitly specify a symbol file for a process (see "symbol-file" on page 7-33), NightView uses the executable file. Since the symbolic information is usually contained in the executable file anyway, this is most often what you want. You can specify the executable file and symbol file in any order for a given process.

When a new executable file is specified, any **`on program`** commands that match the new file name are executed. See "on program" on page 7-36.

Examples:

```
(local) core-file ./mycore
New process:  local:65536
```

```
/users/bob/mycore
was last modified on Wed Nov 18 17:48:38 1992
Core file indicates the executable file is
/users/bob/myprog
Executable file set to
/users/bob/myprog
Pseudo-process assigned PID 65536
Process 65536 terminated with SIGQUIT
(local) family mycore 65536
(local) (mycore) exec-file ./stripped_prog
(local) (mycore) symbol-file ./full_prog
```

The first command creates a new pseudo-process for the file **mycore** in NightView's current directory. NightView assigns this pseudo-process PID number 65536. The **family** command then gives the name **mycore** to this pseudo-process. The **exec-file** command then establishes the file **stripped_prog** as the executable file for that process, while the **symbol-file** command establishes **full_prog** as the name of the symbol file.

on program

Specify debugger commands to be executed when a program is 'exec'ed.

on program [*pattern*]

on program *pattern command*

on program *pattern* **do**

pattern

A wildcard pattern to match against the executable file names of newly 'exec'ed programs. See "Wildcard Patterns" on page 7-14.

command

A debugger command to be executed when a new program whose executable file name matches *pattern* is 'exec'ed.

In the third form of the **on program** command, the debugger commands to be executed must begin on the line following the **do** keyword. The list of debugger commands to execute is terminated when a line containing only the words **end on program** is encountered.

The **on program** command allows a user-specified sequence of one or more debugger commands to be executed immediately after 'exec'ing a program that is being debugged by NightView. When a debugged process performs an 'exec' (or the **exec-file** command is used to change the location of the executable file name), the list of **on program** patterns for that process's controlling dialogue is checked to see if any of the patterns match the executable file name of the program that was just 'exec'ed. The most recently specified **on program** command whose pattern matches the executable file name of the newly 'exec'ed program will have its commands executed.

on program processing is related to **on restart** processing. When a program

execs (or the **exec-file** command is used), NightView first checks the **on restart** patterns. See "on restart" on page 7-38. If a match is found, then the commands associated with the matching pattern are executed. In this case, no **on program** patterns are checked. However, **on restart** commands created by a checkpoint always begin with a call to the macro restart_begin_hook. The initial definition of this macro invokes the **apply on program** command. So, by default, **on program** patterns are checked and matching commands are run *before* the **on restart** commands are run. See "Restarting a Program" on page 3-14.

If no match is found in the **on restart** patterns, then NightView checks the **on program** patterns.

In its first form (given only a pattern), the **on program** command will remove any commands that were associated with the given pattern for each dialogue specified in the qualifier. If no pattern is given, then *all* previously defined **on program** commands are removed from each dialogue specified in the qualifier. If your safety level is set to forbid, you are not allowed to remove all **on program** commands. If your safety level is set to verify, NightView requests verification before removing all **on program** commands. See "set-safety" on page 7-49.

In its second and third forms, the **on program** command will associate a sequence of one or more user-specified debugger commands with the given pattern for each dialogue specified by the qualifier. Macro invocations are *not* expanded when reading the commands to associate with the pattern.

The default qualifier for all commands associated with the given pattern will be the process performing the 'exec'.

The commands specified by **on program** are event-triggered commands: they have an implied safety level (which may be different from the safety level that was set using **set-safety**), and may be terminated automatically if they resume execution of the 'exec'ing process. See "Command Streams" on page 3-29.

If you wish to list all **on program** commands, or see which **on program** commands would be executed for a particular program name, you should use the **info on program** command.

Example:

```
(local)on program ren*  break main.c:24
```

After issuing the above command, if we now run a program in dialogue local named ren_n_stimpy, then we will automatically set a breakpoint in it at line 24 of the file **main.c**.

Now suppose we do the following:

```
(local)on program *stimpy do
>           handle 5 noprint nostop
>           handle 6 noprint nopass
> end on program
```

At this point, if we run ren_n_stimpy again, then this newly 'exec'ed program will handle signals 5 and 6 in the specified manner. Note that even though the name ren_n_stimpy also matches the pattern ren* that a breakpoint will *not* automatically be set at line 24 of **main.c** in this new invocation of ren_n_stimpy. This is because

on program patterns are matched in reverse-chronological order (most recent first), and only the first match found is used.

```
(local) info on program ren_n_stimpy
on program *stimpy do
          handle 5 noprint nostop
          handle 6 noprint nopass
end on program
```

If we were to now issue the command:

```
(local)on program *stimpy
```

Then this would remove `*stimpy` (and its associated commands) from the **on program** list for dialogue `local`. Now, if we run `ren_n_stimpy` a third time, then this third invocation will automatically have a breakpoint set at line 24 of **main.c** (just like the first one did).

```
(local) info on program ren_n_stimpy
on program ren* do
break main.c:24
end on program
```

apply on program

Execute **on program** commands for existing processes.

apply on program

The **apply on program** command allows **on program** commands to be executed for existing processes. (See "on program" on page 7-36). For each process specified by the qualifier, the **on program** commands which would match the executable file name of the process are immediately executed on behalf of the process.

Example:

Suppose I want to set a breakpoint at the subroutine named `main` in all programs both new and old that are debugged in dialogue `local`. Using the **on program** and **apply on program** commands, this could be accomplished as follows:

```
(local) on program *      b  main
(local) apply on program
```

on restart

Specify debugger commands to be executed when a program is restarted.

on restart [*pattern*]

on restart *pattern command*

on restart *pattern* **do**

pattern

A wildcard pattern to match against the executable file names of newly `exec`ed programs. See "Wildcard Patterns" on page 7-14.

command

A debugger command to be executed when a new program whose executable file name matches *pattern* is `exec`ed.

In the third form of the **`on restart`** command, the debugger commands to be executed must begin on the line following the **`do`** keyword. The list of debugger commands to execute is terminated when a line containing only the words **`end on restart`** is encountered.

The **`on restart`** command is primarily intended to be used internally by the debugger as part of the restart processing. See "Restarting a Program" on page 3-14. You may use **`on restart`** explicitly, if desired, but you should be wary of conflicts with the debugger's use. The debugger creates **`on restart`** commands as a result of a checkpoint.

`on restart` is virtually identical to **`on program`** in form and function. See "on program" on page 7-36 for a description of the parameters and functionality of these commands. That section also describes the interaction of these two commands.

If you wish to list all **`on restart`** commands, or see which **`on restart`** commands would be executed for a particular program name, use the **`info on restart`** command. See "info on restart" on page 7-128.

checkpoint

Take a restart checkpoint now.

`checkpoint`

The **`checkpoint`** command saves restart information for the program running in each process in the qualifier.

In most cases, you do not need to use the **`checkpoint`** command, because checkpoints are taken automatically at certain times. See "Restarting a Program" on page 3-14. **`checkpoint`** gives you a way to explicitly take a checkpoint at a time you choose. Note that any later checkpoints (either explicit or automatic) will replace the restart information.

Example:

In this example, you are debugging a complex program. You know some good places to set breakpoints, and you know that you need some more to find the bug, but are not sure yet where they should be. You set your known breakpoints, take a checkpoint, and save the restart information to a file. Then you experiment with some different breakpoints.

```
(local) # set known good breakpoints
(local) breakpoint fred.c:123
set other known breakpoints ...
```

```
(local) checkpoint
(local) info on restart output=restart_info

(local) # now try experimental breakpoints
(local) breakpoint pebbles.c:456
set other experimental breakpoints ...
```

You decide to start the program again and want only the known breakpoints. You kill your process, which takes a checkpoint, including the experimental breakpoints. Then you **source** the file containing the restart information. The restart information is replaced with only the known breakpoints. When you restart your program, only the known breakpoints are restored.

```
(local) kill
(local) source restart_info
restart program
```

family

Give a name to a family of one or more processes.

family *family-name* [[-] *qualifier-spec*] ...

family-name

The family name to be defined. This must not be the same as the name of any dialogue you currently have. The family-name must consist only of alphanumeric characters and underscores and must begin with an alphabetic character. The family-name may be of arbitrary length.

qualifier-spec

Identifies one or more processes to be included or excluded in the family named by *family-name*. See "Qualifier Specifiers" on page 7-10.

The total set of processes is accumulated by scanning the *qualifier-spec* arguments left to right. An argument is added to the set unless it is preceded by a '-', in which case it is subtracted from the set accumulated so far.

If no *qualifier-spec* is included, then this command removes any previous definition of the *family-name*. If your safety level is set to `forbid`, you are not allowed to remove the definition of a *family-name* that is present in the default qualifier. If your safety level is set to `verify`, NightView requests verification before removing such a definition. See "set-safety" on page 7-49.

If one or more *qualifier-spec* arguments are supplied, they are immediately evaluated (see "Qualifier Specifiers" on page 7-10) and the *family-name* is defined as the list of processes indicated by those arguments. Evaluation of the arguments has the following implications:

- Any *family-name* appearing in the argument list must be defined. Subsequent changes made to the definition of that *family-name* will have no

effect on the processes implied by the *family-name* being defined in the **`family`** command.

- The processes denoted by any *dialogue-name* appearing in the argument list are just those that exist at the time the **`family`** command is executed.
- The argument `all` denotes only those processes that exist at the time the **`family`** command is executed.
- The argument **`auto`** denotes the process that has been stopped the longest at the time the **`family`** command is executed.

Any qualifier applied to this command has no effect.

Note that you may use a *family-name* in a qualifier before it is actually defined, but you must define the *family-name* before executing any command that needs to know what the *family-name* refers to.

Examples:

```
(local) family fam1 12 25 18
(local) family fam2 fam1 99
(local) family fam1 fam1 16
```

The first command gives the name *fam1* to the processes identified by PIDs 12, 18, and 25. The second command gives the name *fam2* to the three processes in *fam1* plus process 99. The third command extends the definition of *fam1* to include process 16; thus *fam1* is a synonym for four processes: 12, 16, 18, and 25. Note that extending *fam1* has no effect on *fam2*, which still consists of processes 12, 18, 25, and 99.

Using the families defined in the previous examples, the use of a minus sign on arguments can be illustrated by the following examples:

```
(local) family fam3 fam1 fam2 -12
(local) family fam3 fam1 -12 fam2
```

The first command defines *fam3* to be the processes 16, 18, 25, and 99. In contrast, the second command defines *fam3* to be the processes 12, 16, 18, 25, and 99. In this case, the argument `-12` removed process 12 from the set accumulated from *fam1*, but the `fam2` argument adds that process back in. In general, it is a good idea to put all the subtracted arguments at the end of the list.

set-children

Control whether children should be debugged.

`set-children` `{ all [ resume ] | exec | none }`

`all`

Debug all children. If the optional keyword `resume` is specified, then a child process is `resume`d automatically after NightView has prepared it for debugging. This is useful if your program creates many child processes that you want to debug, but all you need to do is inherit the eventpoints and debug settings from the parent process. See “Multiple Processes” on page 3-2.

`exec`

Debug children only when they have called **`exec(2)`** (that is, when they are running a different program). The program name is checked against the debug/nodebug list for the controlling dialogue to see if the program should be debugged. See "debug" on page 7-20. This is the default setting for direct children of the dialogue shell and processes debugged with the **`attach`** command. See "attach" on page 7-32.

`none`

Ignore all children.

Sometimes you are not interested in the child processes of the process you are debugging. For example, your program may make many calls to **`system(3)`** which you are not interested in debugging. The **`set-children`** command gives you a way of controlling which children will be debugged without having to detach from each one individually. See "detach" on page 7-32.

The **`set-children`** command applies to future children of the processes specified by the qualifier. Existing children are not affected.

This mode is inherited by future children.

set-exit

Control whether a process stops before exiting.

`set-exit` `[stop | nostop]`

`stop`

The process will stop if the **`exit`** system service is called.

`nostop`

The process will *not* stop before exiting.

The **`set-exit`** command controls whether the processes specified by the qualifier will stop before exiting. The default state for a process is to stop before exiting. See "Exited and Terminated Processes" on page 3-16.

If no arguments are specified to the command, the command prints the current state for each process in the qualifier. If an argument is specified, the command changes the state of each process in the qualifier accordingly and then prints the new state.

Note that the **`set-exit`** mode is inherited by a child process if a process forks. Note also that the mode persists for the entire life of the process, even across an `exec` system call, until modified by another **`set-exit`** command. In the case of an `exec`, an **`on program`** or **`on restart`** command might specify a **`set-exit`** command that changes the mode. See "on program" on page 7-36 and "on restart" on page 7-38. See also "Restarting a Program" on page 3-14.

If you also want a process to automatically resume execution after an `exec`, put a **`resume`** command in an **`on program`** specification. See "resume" on page 7-98 and

"on program" on page 7-36.

mreserve

Reserve a region of memory in a process.

mreserve start=*address* {length=*bytes* | end=*address*}

start=*address*

Specify the start address of the region.

length=*bytes*

Specify the length of the region in bytes.

end=*address*

Specify the end address of the region.

The start=*address* parameter is required. You must specify either a length or an end address.

The **mreserve** command reserves a region of memory for each process specified by the qualifier. This means that NightView will not allocate space for patch areas in that region. See Appendix E [Implementation Overview] on page E-1.

This command does not directly affect the process. It is only an indication to NightView to avoid placing patch areas in the specified region, presumably because your program will be using that region later in its execution.

mreserve only affects *future* allocations. You should reserve memory before using any commands that allocate space in the process, including eventpoint commands, the **load** command, or any command with an expression that involves a function call. See "Eventpoints" on page 3-8. See "load" on page 7-75. See "Expression Evaluation" on page 3-20.

You should exercise some caution with this command. It is possible to reserve memory in such a way that NightView cannot function.

For convenience, you are allowed to specify reservations that overlap or contain existing regions in your process.

Memory reservations are printed as part of the **info memory** command. See "info memory" on page 7-126.

Memory reservations are remembered as part of the restart information. See "Restart Information" on page 3-15. During restart, memory reservations are applied before any commands that would allocate space in the process.

Setting Modes

set-log

Log session to file.

`set-log` *keyword filename*

keyword

The *keyword* parameter must be one of the following:

`all`

Log entire session (commands as well as the output generated by commands).

`commands`

Log just commands typed.

`close`

Close a log file.

filename

Name of the log file.

This command starts logging the debugger session to a file. If the file already exists, the log information is appended to it. You may log just the commands (by using the `commands` keyword) or the entire session (`all` keyword) to a file (if the named file is already an open log file, specifying a different keyword simply changes the mode of the log). You may open multiple log files (although more than one of each type of log would be rather redundant).

The `close` keyword is used to close the log associated with the file. (See "info log" on page 7-115).

The qualifier does not have any effect on this command. Any logs are global to the debug session.

Note that this command logs everything that happens during the debug session (essentially, everything you see on your terminal). The **`set-show`** command may be used to log output from a single dialogue (see "set-show" on page 7-28).

set-language

Establish a default language context for variables and expressions.

`set-language` {`ada | auto | c | c++ | fortran`}

`ada`

Indicates that the default language should be Ada.

`auto`

Indicates that the default language should be determined automatically.

`c`

Indicates that the default language should be C.

`c++`

Indicates that the default language should be C++.

`fortran`

Indicates that the default language should be Fortran.

The arguments to this command can be in any mixture of upper and lower case.

For each process specified by the qualifier, **`set-language`** sets the default language used to interpret expressions and variables in commands. If a default language has not been established, or if the default has been set to **`auto`**, NightView decides the language in one of two ways. If the object file contains DWARF, then it contains the language information. Otherwise, NightView infers the language from the extension (the last few characters) of the source file name associated with the frame selected when the expression or variable is mentioned. The following extensions are recognized:

`.a`

The language is assumed to be Ada.

`.c`

The language is assumed to be C.

`.C`

The language is assumed to be C++.

`.f`

The language is assumed to be Fortran.

`.s`

Although this indicates an assembler source file, NightView uses the C language for such files. C expressions include nearly all the operators allowed by the assembler, plus much more.

The language determines the meaning of operators and constants in expressions; determines the syntax of some kinds of expressions (e.g., C type casts); controls the visibility of variable names; and controls the significance of case (upper versus lower) in variable names. The language also controls the formatting of output from the **`print`** command (see “print” on page 7-66), especially the way the type of an expression is

indicated.

set-qualifier

Specify the default list of processes or dialogues that will be affected by subsequent commands which accept qualifiers.

`set-qualifier` [*qualifier-spec* ...]

qualifier-spec

Specifies a process or dialogue to be included in the default qualifier list (see "Qualifier Specifiers" on page 7-10). Any family names in the *qualifier-spec* are evaluated at the time of each command, not at the time of **`set-qualifier`**.

If no argument is specified, the default qualifier is set to null, meaning that a qualifier must be supplied to subsequent commands that require qualification.

set-history

Specify the number of items to be kept in the value history list.

`set-history` *count*

count

The number of items to be kept in the value history.

The qualifier is ignored on this command. The default history list size is 1000. If more history items than that are created, the oldest ones are discarded. No matter how many items are in the list, each new history item gets the next highest number.

set-limits

Specify limits on the number of array elements, string characters, or program addresses printed when examining program data.

`set-limits` {`array=`*number* | `string=`*number* | `addresses=`*number*} ...

`array=`*number*

The `array` keyword parameter specifies the maximum number of array elements to be printed. If you want unlimited output, specify zero as the limit.

`string=`*number*

The `string` keyword parameter specifies the maximum number of characters of a string to be printed. If you want unlimited output, specify zero as the limit.

`addresses=`*number*

The `addresses` keyword parameter specifies the maximum number of addresses to be printed for a particular location (See "Location Specifiers" on page 7-9). If you want unlimited output, specify zero as the limit.

The `array`, `string`, and `addresses` keywords may be specified in any order.

The qualifier is ignored on this command. The limits set by **`set-limits`** apply to all output of variables or expressions or program locations. If a printed value is truncated because of these limits, the value will be followed by ellipses.

Note that the limitation on array elements applies to each dimension of a multi-dimensional array. If you print a `50 x 20` two-dimensional array, and you have the array limit set to `5`, then you will see the first 5 elements of the each of the first 5 rows (or columns, for Fortran).

The default limits are 100 array elements, 100 characters, and 10 addresses. To find out what the current limits are, use the **`info limits`** command (See "info limits" on page 7-124).

set-prompt

Set the string used to prompt for command input.

`set-prompt` *string*

string

Specify the string the debugger uses to prompt for command input. The string must be enclosed in double quotes. If you include any of the following sub-strings in the prompt, they will be expanded by the debugger immediately prior to printing the prompt.

%q

Expands to the current default qualifier. This prints out the same way the qualifier was defined. If you used a family name, it shows the family name (not the individual PIDs), etc. If the default qualifier is **`auto`**, it prints the current automatically selected PID.

%p

Expands to the complete list of PIDs implied by the current default qualifier.

%d

Expands to the complete list of dialogues implied by the current default qualifier.

%a

Expands to the complete list of dialogues, if the current default qualifier is `all`. Otherwise, this expands to the current default qualifier.

`%%`

Expands to the single character %.

The *string* argument may also include the escape sequences recognized in C language strings, such as '\n' to indicate a newline.

The string ``(%a)" is the default prompt.

The qualifier on the **`set-prompt`** command is ignored.

Examples:

```
(afamily) set-prompt "%p> "
local:2047,2048>
```

The above example shows what happens when the default qualifier is a process family named `afamily` assumed to contain two PIDs (2047 and 2048), both in dialogue `local`. The initial prompt is "`(%q)`" and the **`set-prompt`** command changes it to expand to a list of PIDs.

```
(afamily) set-prompt "Dialogues: %d\nProcesses: %p>"
Dialogues: mach1,mach2
Processes: mach1:15 mach2:15,549,2047,2048>
```

The above example prints two lines as a prompt, the first containing a list of dialogues and the second containing a list of processes.

set-terminator

Set the string used to recognize end of dialogue input mode.

`set-terminator` *string*

string

Define the *string* used to terminate dialogue input mode (see "!" on page 7-27).

When the **`!`** command is used to switch all input to a dialogue, the terminator string is recognized to switch input back to the debugger. The terminator string must appear on a line by itself to be recognized. The default string is "`-.`" (different from **`rlogin`** and **`cu`**).

Unlike normal debugger commands, this string must be typed exactly as specified in the **`set-terminator`** command. The case of the letters must match, and the full string must be typed.

Only one terminator string is defined. The qualifier on this command is ignored.

Leading and trailing whitespace in the specified terminator string is ignored. Macros are *not* expanded when reading the new terminator string.

If no terminator string is given, then the current terminator string is printed, otherwise the new terminator string is printed.

set-safety

Control debugger response to dangerous commands.

`set-safety` `[forbid | verify | unsafe]`

`forbid`

In `forbid` mode, the debugger simply refuses to execute a dangerous command and explains why it will not execute. (You may have tried to **`quit`** while processes were still running, etc.).

`verify`

In `verify` mode, the debugger tells you what dangerous thing you are about to do and asks if you really meant that (see "Replying to Debugger Questions" on page 7-16). If you answer `yes`, it goes ahead and does it. This is the default safety level of the debugger.

`unsafe`

In `unsafe` mode, the debugger simply tells you what it did. It assumes you meant what you said and does not try to stop you.

If no mode is specified then the **`set-safety`** command prints the current safety level.

The qualifier on the **`set-safety`** command is ignored.

set-restart

Control whether restart information is applied.

`set-restart` `[always | never | verify]`

`always`

Restart information is unconditionally applied when a program starts. This is the default mode.

`never`

Restart information is never applied when a program starts.

`verify`

When a program starts, you are asked whether to apply restart information to it.

If no keyword is specified then the **`set-restart`** command prints the current restart mode.

The restart mode is a global mode, not a per-process or per-dialogue mode. The qualifier on the **`set-restart`** command is ignored.

See "Restarting a Program" on page 3-14.

set-local

Define process local convenience variables.

`set-local` *identifier ...*

identifier

The name of a convenience variable (the leading '$' on each identifier, normally used to reference convenience variables, is optional).

Each named identifier is defined to be a process local convenience variable.

A process local variable always has a unique value in each process. If the variable was already defined as a global at the time it appears in a **`set-local`** command, then each process gets a separate copy of the current global value, but future changes will be unique for each process.

The command qualifier does not have any effect on this command. It is not possible to define a variable to be local for only one process, but globally shared among other processes.

set-patch-area-size

Control the size of patch areas created in your process.

`set-patch-area-size` {`data=`*data-size* | `eventpoint=`*eventpoint-size* | `monitor=`*monitor-size* | `text=`*text-size*} `...`

`data=`*data-size*

The `data` keyword parameter specifies the size of the data area in kilobytes.

`monitor=`*monitor-size*

The `monitor` keyword parameter specifies the size of the shared memory region used by all monitorpoints in this dialogue, in kilobytes.

`text=`*text-size*

The `text` keyword parameter specifies the size of the text area in kilobytes.

`eventpoint=`*eventpoint-size*

The `eventpoint` keyword parameter specifies the size of the eventpoint areas in kilobytes.

The `data`, `monitor`, `text`, and `eventpoint` keywords may be abbreviated and may be specified in any order.

NightView creates some regions in your process, and uses these regions to store text and data. There is usually one data region, one text region, one or more eventpoint regions, and, if there are any monitorpoints in the process, one shared memory region for the monitorpoints. These regions are called *patch areas.* See Appendix E [Implementation Overview] on page E-1.

You can adjust the sizes of the patch areas with this command. For example, if you have a lot of conditional eventpoints, then you may need to make the size of the eventpoint and text regions larger so that NightView has room to allocate all the code necessary for those eventpoints. Similarly, if you have a lot of monitorpoints, then you may need to make the size of the monitorpoint shared memory region larger. On the other hand, if system memory resources are scarce, then you may need to make some of these regions smaller.

The patch area size values are associated with each dialogue and apply to all processes within the dialogue. This command sets the values for each dialogue specified in the qualifier.

Note that these values only apply to patch areas created in the future. Existing regions are not changed. Therefore, if you want to debug a program and use a large text or data area, you need to specify that before you run your program (i.e., before the process calls `exec`). (For `fork`, the child process inherits its regions from the parent, so the regions are the same size in the child and the parent.)

Each process has its own data, eventpoint and text areas, but the monitorpoint shared memory region is shared by all the processes that have monitorpoints in the dialogue, and by the dialogue itself. Therefore, if you want to change the size of the monitorpoint shared memory region, you need to do so before creating any monitorpoints in the dialogue. See “Monitorpoints” on page 3-10.

The initial values of the patch area sizes are 512 kilobytes each for the data and text patch areas, 256 kilobytes for the eventpoint areas, and 32 kilobytes for the monitorpoint shared memory region. This is adequate for most applications.

Use **`info dialogue`** to see the current patch area size values. (see “info dialogue” on page 7-126).

You can see information about the patch areas in an existing process with the **`info memory`** command (see “info memory” on page 7-126).

interest

Control which subprograms are interesting.

`interest` [*level*] [[at] [*location-spec*]]

Set or query the interest level for a subprogram.

`interest` `inline`[=*level*]

`interest` `justlines`[=*level*]

`interest` `nodebug`[=*level*]

`interest` `threshold`[=*level*]

Set or query the interest keyword values.

level

Specify a level for the subprogram defined by *location-spec*, or a value for the specified keyword. *level* is a signed integer or the keywords `minimum` or

`maximum`. If this argument is not present, then this command queries the level of the subprogram or the specified keyword.

`[at]` *location-spec*

Set or query the interest level for the subprogram specified by *location-spec*. See "Location Specifiers" on page 7-9. If no *location-spec* is present, it defaults to `*$cpc`. If the `at` keyword is present, it must be followed by a *location-spec*. If no *level* is specified, then the `at` keyword is required to distinguish some forms of location specifiers from a *level.*

`inline`

Set or query the inline interest level. If this level is less than the interest level threshold, then all inline subprograms have the `minimum` interest level unless their interest level has been explicitly set with **`interest`** *level location-spec*. The initial value of this level is `0`.

`justlines`

Set or query the interest level for subprograms with line number information but no other debug information. The initial value is `-2`.

`nodebug`

Set or query the interest level for subprograms with no debug information (e.g., system library routines). Without debug information, the interest level cannot be specified for individual subprograms, so NightView uses the value specified by this form. The initial value is `-4`.

`threshold`

Set or query the interest level threshold NightView uses to decide whether a subprogram is interesting. The initial value is `0`.

The **`interest`** command sets or queries the information NightView uses to decide which subprograms are interesting for each process in the qualifier. See "Interesting Subprograms" on page 3-27.

The `minimum` keyword specifies the lowest possible interest level. The `maximum` keyword specifies the highest possible interest level.

A query prints the interest information requested. If an interest level is being set, the command prints the new interest level.

Some compilers provide a means to specify the interest level of a subprogram through the debug information. If the subprogram has debug information, but it does not specify an interest level, the default level is `0`. The **`interest`** command overrides an interest level set at compile time.

The interest levels and the interest level threshold are remembered as part of the restart information. See "Restart Information" on page 3-15. For a way to see all the interest levels that have been explicitly set, see "info on restart" on page 7-128.

If an interest level or the interest level threshold is changed, then NightView checks the current frame to see if it has become uninteresting. See "Current Frame" on page 3-25. If

it has, then the current frame is reset to frame 0 of the current context and frame information is printed. See "select-context" on page 7-110. Even if the current frame does not have to be reset, it gets a different frame number if frames below it have become hidden or unhidden.

Examples:

```
(local) run fact 7
...process startup information...
(local) interest
local:6729: Interest level is -4 (uninteresting) for
0x100024d0 (nodebug)
```

You query the interest level, using the default location specifier of `*$cpc`. The program begins in the C runtime startup routine, which has no debug information, so it is uninteresting.

```
(local) breakpoint 26
local:6729 Breakpoint 1 set at fact.c:26
(local) continue
local:6729: at Breakpoint 1, 0x10002780 in main(int argc
= 2, unsigned char ** argv = 0x2ff7eae4) at fact.c line
26
26 B=|       answer = factorial(x);
(local) step
#0  0x100026f4  in factorial(int x = 7) at fact.c line 6
6  = |    if (x <= 1) {
(local) interest -1
local:6729: Interest level set to -1 (uninteresting) for
factorial
#0  0x10002780  in main(int argc = 2, unsigned char **
argv = 0x2ff7eae4 at fact.c line 26S
26 B<>|       answer = factorial(x);
```

You step into the `factorial` function, then decide that it is not interesting. You mark `factorial` uninteresting, using the default location specifier. Your current frame becomes uninteresting, so it is reset to frame 0. Frame 0 is now the frame for `main`, because `factorial` is not interesting. The source decorations for line 26 show that `$pc` and `$cpc` are within that line. See "Source Line Decorations" on page 7-63.

```
(local) interest threshold=-1
local:6729: threshold interest level set to -1
(local) frame
Output for process local:6729
#1  0x10002780  in main(int argc = 2, unsigned char **
argv = 0x2ff7eae4) at fact.c line 26
26 B<>|       answer = factorial(x);
```

You change the interest level threshold, which makes `factorial` interesting again. Your current frame is still interesting, so it is not reset to frame 0. The **`frame`** command shows that your current frame is still the frame for `main`, but now that frame is frame number 1.

set-auto-frame

Control the positioning of the stack when a process stops.

set-auto-frame *args* ...

The functionality of this command has been subsumed by the **interest** command. See "interest" on page 7-51. This command has been retained for compatibility, but it might be removed in some future release.

set-overload

Control how NightView treats overloaded operators and routines in expressions.

set-overload [operator={on | off}] [routine={on | off}]

operator={on | off}

Turn operator overloading on or off.

routine={on | off}

Turn routine overloading on or off.

The **set-overload** command determines how NightView treats overloaded operators, functions, and procedures in expressions. See "Expression Evaluation" on page 3-20. This behavior can be controlled for operators separately from functions and procedures using the keywords on the command. The specified settings apply to all expressions evaluated by NightView. The qualifier is ignored by the **set-overload** command. The routine mode also controls overloading of function names which appear in location specifiers.

After setting the specified overloading modes, the **set-overload** command prints the new settings. If no arguments are specified, the command simply prints the existing overloading modes.

For a discussion of how overloading works in NightView see "Overloading" on page 3-23. For the details of the syntax used to specify overloading in expressions and location specifiers see "Selecting Overloaded Entities" on page 7-2.

set-search

Control case sensitivity of regular expressions in NightView.

set-search [sensitive | insensitive]

sensitive

Make regular expressions case sensitive (this is the default setting).

insensitive

Make regular expressions case insensitive.

The **`set-search`** command controls case sensitivity for the regular expressions (see “Regular Expressions” on page 7-12) used by several commands as well as some dialog boxes in the graphical interface.

When the **`set-search`** command is run with no argument, it reports (but does not change) the current mode setting.

When the `sensitive` argument is specified, regular expressions become case sensitive. The case of alphabetic characters must match exactly as written in the regular expression. This is the default **`set-search`** mode.

When the `insensitive` argument is specified, regular expressions become case insensitive. Either the upper case or the lower case form of an alphabetic character will match both the upper and lower case form of that same character.

set-editor

Set the mode for editing commands in the simple full-screen interface.

`set-editor` *mode*

mode

One of `emacs`, `gmacs` or `vi`.

Determine which kind of keystroke commands are available to edit commands in the simple full-screen interface.

See “Editing Commands in the Simple Full-Screen Interface” on page 8-2.

Debugger Environment Control

cd

Set the debugger's default working directory.

cd *dirname*

dirname

The name of the directory.

The **cd** command changes the working directory of NightView to the specified directory. You usually use this command to control the search for source files, core files, and program files. It affects the behavior of the following commands:

- **shell** (see "shell" on page 7-112)
- **list** (see "list" on page 7-58)
- **directory** (see "directory" on page 7-60)
- **symbol-file** (see "symbol-file" on page 7-33)
- **core-file** (see "core-file" on page 7-34)
- **exec-file** (see "exec-file" on page 7-35)

The **cd** command does not affect commands executed in dialogue shells (see "login" on page 7-18). Also, the qualifier does not have any effect on this command.

You can use the **pwd** command to find out what NightView's current working directory is. See "pwd" on page 7-56.

pwd

Print NightView's current working directory.

pwd

This command prints the current working directory of the debugger. Note that this directory may not be the same as the current working directory of your dialogue shells, nor need it be the same as the current working directory of any program you are debugging.

You can use the **cd** command to set the current working directory. (see "cd" on page 7-56).

The qualifier does not have any effect on this command.

Source Files

This section describes commands to view source files and to search for text in source files.

Viewing Source Files

list

List a source file. This command has many forms, which are summarized below.

`list` *where-spec*

List ten lines centered on the line specified by *where-spec*.

`list` *where-spec1*`,` *where-spec2*

List the lines beginning with *where-spec1* up to and including the *where-spec2* line.

`list` `,`*where-spec*

List ten lines ending at the line specified by *where-spec*.

`list` *where-spec*`,`

List the ten lines starting at *where-spec*. Note the comma.

`list +`

List the ten lines just after the lines last listed.

`list -`

List the ten lines immediately preceding the lines last listed.

`list =`

List the last set of lines listed. If the previous command was a search command, list the ten lines around the line found by the search.

`list`

If a list command has not been given since the current source file was last established (see below), this form lists the ten lines centered around the line where execution is stopped in the current source file. Otherwise, this form lists the ten lines just after the last lines listed.

Abbreviation: **`l`**

Each *where-spec* argument can be any one of the following forms.

`[at]` *location-spec*

Specifies a location in the program or a source file (See "Location Specifiers" on page 7-9). No matter which form of *location-spec* you use, it is always translated into a source line specification for this command. If you give two arguments on the **`list`** command, they cannot specify different source files.

`[at]` *file_name*

Specifies the first line of the file. The *file_name* may be a quoted or unquoted string, but be aware that an unquoted string may be ambiguous. A string without quotes will be interpreted first as a function name or an Ada unit name; if no such function or Ada unit exists, the string will then be interpreted as a file name.

`+`*n*

Specifies the line that is *n* lines *after* the last line in the last group listed (see below). If this is the second *where-spec*, it specifies the line *n* lines after the first argument.

`-`*n*

Like `+`*n*, except it specifies the line *n* lines *before* the last line in the last group listed (see below). If this is the second *where-spec*, it specifies the line *n* lines before the first argument.

The **`list`** command is applied to each process in the qualifier. If the qualifier specifies more than one process, you get one listing for each process; each listing is preceded by a notation indicating which process the listing is for. The specified source file is found using the directory search path you established using the **`directory`** command (see "directory" on page 7-60). Note that each program has its own directory search path.

NightView maintains, for each process, a current source file. The current source file is usually the most recent file listed or searched. However, when the process stops execution, the current source file is automatically set to the file where execution stopped. The context selection commands (see "Selecting Context" on page 7-108) also set the current source file to the one associated with the selected stack frame. When a process first starts execution, the current source file is the one containing the main program. If the first argument to the **`list`** command does not explicitly specify a source file, then the current source file is used.

When you list one or more lines in a source file, NightView remembers the first and last line of that group. If you subsequently give a **`list`** command that uses a relative *where-spec* or contains just a `+` or `-` argument, those arguments are interpreted relative to the lines in the last group listed. Arguments containing a `+` are relative to the last line in the group, and arguments containing a `-` are relative to the first line in the group. This also affects the **`forward-search`** and **`reverse-search`** commands. See "forward-search" on page 7-61 and "reverse-search" on page 7-61.

Repeating the **`list`** command by entering a blank line behaves differently depending on the form of **`list`** you used last. In most cases, repeating the command lists the next ten lines following the last line in the last group. However, if you used the **`list -`** form last, then repetition lists the ten lines preceding the first line in the last group.

The listed source lines are preceded by *source decorations*. (see "Source Line Decorations" on page 7-63).

You can use the **`info line`** command to determine the location in your program of the code for a particular source line. (see "info line" on page 7-133).

directory

Set the directory search path.

directory [*dirname* ...]

dirname

The name of a directory to include in the search path. If this is not an absolute pathname, it is interpreted relative to NightView's current working directory and transformed into an absolute pathname. Thus, if you later change NightView's working directory, the search path will not be affected. See "cd" on page 7-56 and "pwd" on page 7-56.

The **directory** command sets the directory search path for the program in each process in the qualifier. The arguments are used in order as the elements of the directory search path. Subsequent **directory** commands contribute directories to the head of the current search path.

The directory search path is used for displaying source files. When you list a source file (see "list" on page 7-58), NightView looks for the source file in each of the directories in the search path, starting at the beginning of the search path each time.

If no **directory** command has been specified for the program, the search path implicitly contains the path to the executable file and NightView's current working directory. Once a **directory** command is specified for the program, these directories are no longer implicit in the search path.

If you enter a **directory** command with no arguments, the search path is reset to its initial state.

The directory search path is associated with a program, not with a process. If you debug multiple instances of a program, the directory search path is the same for each instance. If your process calls **exec(2)**, the directory search path is implicitly set for the new program.

Use the **info directories** command to display the directory search path for a program. See "info directories" on page 7-123.

For ELF programs, the debugging information contains absolute pathnames to source files, so the directory search path may not be needed. It is still sometimes useful to indicate that a source tree is not where the debugging information indicates.

Example:

Suppose your ELF program was compiled from two source files: **/usr/bob/src/main/main.c** and **/usr/bob/src/doit/doit.c**. You want to debug your program, but you have moved the source files to **/usr/joe/main/main.c** and **/usr/joe/doit/doit.c**. Enter a **directory** command to indicate the new root of the source tree:

```
(local) directory /usr/joe
```

Searching

forward-search

Search forward through the current source file for a specified regular expression.

`forward-search` [*regexp*]

Abbreviation: **`fo`**

regexp

The regular expression to search for. *No* anchored match is implied. (see "Regular Expressions" on page 7-12). If *regexp* is omitted, the previous *regexp* is used.

The search command is applied to the current source file of each process specified by the qualifier.

The search starts at the first line displayed by the last **`list`** command, the last place the process stopped, or the last place a search was satisfied, whichever was most recent, and proceeds forward through the file to the end. In the graphical user interface, the search position is not affected by scrolling the source window. If the regular expression is found, the containing source line is listed. This will affect subsequent **`list`** commands that specify relative arguments.

If the end of the file is encountered without finding the regular expression, a message is printed indicating the search was unsuccessful. For a definition of current source file, see "list" on page 7-58.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

reverse-search

Search backwards through the current source file for a specified regular expression.

`reverse-search` [*regexp*]

regexp

The regular expression to search for. *No* anchored match is implied. (see "Regular Expressions" on page 7-12). If *regexp* is omitted, the previous *regexp* is used.

The search command is applied to the current source file of each process specified by the qualifier. The search starts at the last line displayed by the last **`list`** command, the last place the process stopped, or the last place a search was satisfied, whichever was most recent, and proceeds backward through the file to the beginning. In the graphical user interface, the search position is not affected by scrolling the source window. If the regular expression is found, the containing source line is listed. This will affect subsequent **`list`** commands that specify relative arguments.

If the beginning of the file is encountered without finding the regular expression, a message is printed indicating the search was unsuccessful. For a definition of current source file, see "list" on page 7-58.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

Source Line Decorations

When NightView lists source lines in your program or displays the assembly instructions of your program, it precedes each line with decorations providing information about that line. Every source line gets a *line number*, which is relative to the beginning of that file. Each instruction displayed is preceded by the line number of the source line that generated it (see "x" on page 7-68). Following the line number may be one or more of the decorations shown in the following table.

Table 7-6. Source Line Decorations

'A'

Indicates that one or more agentpoints, possibly disabled, are set somewhere within this source line. When displaying instructions, this indicates that one or more agentpoints are set on this instruction. (see "agentpoint" on page 7-87).

'B'

Indicates that one or more breakpoints, possibly disabled, are set somewhere within this source line. When displaying instructions, this indicates that one or more breakpoints are set on this instruction. (see "breakpoint" on page 7-79).

'M'

Indicates that one or more monitorpoints, possibly disabled, are set somewhere within this source line. When displaying instructions, this indicates that one or more monitorpoints are set on this instruction. (see "monitorpoint" on page 7-84).

'P'

Indicates that one or more patchpoints, possibly disabled, have been inserted somewhere within this source line. (see "patchpoint" on page 7-80). When displaying instructions, this indicates the instruction where the patchpoint was inserted, and the patched expressions are displayed elsewhere.

'T'

Indicates that one or more tracepoints, possibly disabled, are set within this source line. When displaying instructions, this indicates a tracepoint immediately preceding this instruction. (see "tracepoint" on page 7-83).

'='

Indicates that execution is stopped somewhere within or at the beginning of this line. When displaying instructions, this indicates the instruction at which execution is stopped (the one that will next be executed).

'>'

Indicates the line (or instruction) in the current frame (see "frame" on page 7-108), where execution will resume when the called routine returns.

This decoration is not displayed if the current frame is frame #0 (with no hidden frames below frame 0); in this case the '=' decoration will appear in its place.

'<'

Indicates the line (or instruction) in the current frame (see "frame" on page 7-108), which was executing when the called frame was created, i.e., `$cpc`. See "Program Counter" on page 3-24.

This decoration is not displayed if the current frame is frame #0 (with no hidden frames below frame 0); in this case the '= 'decoration will appear in its place.

'*'

Indicates that this source line corresponds to executable code. A line that appears executable may still not have executable code associated with it because of optimization or conditional compilation. Not used when displaying instructions.

This decoration is not displayed if there are any other indicators also on that line, since the other indicators imply there is executable code for the line.

'@'

Used only when displaying instructions, this character indicates that the associated instruction is the first for the corresponding source line.

NightView reserves enough columns for displaying a 3-digit line number, 2 decoration characters, and a 2-character separator. If the line number and decorations fit within this space, the source text displayed lines up in columns just as it does in the source file. If more space is needed for line number or decorations, the line is shifted over accordingly.

In the source listing, the 2-character separator is a vertical bar followed by a space. This helps distinguish decorations from source characters. In the disassembly listing, the 2-character separator consists of 2 spaces.

Example source listing:

```
20   | void
21 * | main(argc, argv)
22   |    int argc;
23   |    char ** argv;
24   | {
25   |    int i, errors;
26 * |    errors = 0;
27 * |    for (i = 1; i < argc; ++i) {
28   |       long xl;
29   |       int x;
30   |       int answer;
31 * |       char * ends = NULL;
32 T |       xl = strtol(argv[i], &ends, 10);
33 B=|       x = (int)xl;
34 B |       answer = factorial(x);
35 P |       printf("factorial(%d) == %d\n", x, answer);
36   |    }
37 * |    exit(errors);
38   | }
```

In this example, line 32 has a tracepoint set on it; line 33 has a breakpoint set somewhere within the line, and execution is stopped on the line (but not necessarily at the breakpoint). Line 34 has a breakpoint set somewhere within the line (perhaps on the

return from `factorial`). Line 35 has a patchpoint inserted somewhere within it. Apart from these lines, the other lines with asterisks on them have executable code associated with them.

Example instruction listing:

```
31 @   0x10002788 <main+52>:     li r6,0
31     0x1000278c <main+56>:     stw r6,0x40(r1)
32 @T  0x10002790 <main+60>:     slwi r5,r16,2
32     0x10002794 <main+64>:     lwzx r3,r17,r5
32     0x10002798 <main+68>:     addi r4,r1,64
32     0x1000279c <main+72>:     li r5,10
32     0x100027a0 <main+76>:     bl 0x100010e0 <strtol>
33 @B=  0x100027a4 <main+80>:    mr r20,r3
34 @   0x100027a8 <main+84>:     bl 0x10002700 <factorial>
34 B   0x100027ac <main+88>:     mr r5,r3
35 @P  0x100027b0 <main+92>:     lis r3,12288
35     0x100027b4 <main+96>:     addi r3,r3,12528
35     0x100027b8 <main+100>:    mr r4,r20
35     0x100027bc <main+104>:    bl 0x10001100 <printf>
```

This is a partial assembly listing for the preceding example source listing.

Examining and Modifying

backtrace

Print an ordered list of the currently active stack frames.

`backtrace` [*number-of-frames*]

Abbreviation: **`bt`**

number-of-frames

Number of stack frames to print, starting with the currently executing frame.

The **`backtrace`** command prints, for each process specified in the qualifier, a summary of the active stack frames, starting with the currently executing frame. Each subsequent entry corresponds to the caller of the frame which precedes it in the listing. All active frames are indicated, unless a value for *number-of-frames* is given, in which case, the given number of frames is printed.

Each entry in the **`backtrace`** listing includes the frame number (the first frame is numbered `0`), the program counter, the subprogram name (if known), the arguments of the subprogram (if known), the source file name (if known), and the line number (if known).

For information on changing the current stack frame, see “frame” on page 7-108, “up” on page 7-109, or “down” on page 7-109.

Frames corresponding to uninteresting subprograms are not shown in the listing. See “Interesting Subprograms” on page 3-27.

print

Print the value of a language expression.

print [/*print-format-letter*] *expression*

Abbreviation: **p**

print-format-letter

One of the following letters specifying the format in which to print each component value of the expression:

a

P rint the value of the expression in hexadecimal and as an address relative to a program symbol.

c

Treat the rightmost (least significant) eight bits of the value as a character constant and print the constant.

d

Print the bit representation of the value in signed decimal.

f

Print the bit representation of the value as a single precision floating-point number and print using floating-point syntax. If the data type of the language expression is double precision, however, then the bit representation is printed as a double precision floating-point number.

o

Print the bit representation of the value in octal.

s

Print the data as a character string. Arrays of characters will print as one character string (terminated with a zero byte if the language is C or C++); scalar types will print using their default format plus the bytes of the value will be printed as a string. (You might want to use this in Fortran if you put Hollerith data in INTEGER variables.)

See note below about limits on the length of printed strings.

u

Print the bit representation of the value in unsigned decimal.

x

Print the bit representation of the value in hexadecimal.

expression

A language expression (see "Expression Evaluation" on page 3-20).

print displays the value of a language expression in each process specified by the qualifier. When the expression is an aggregate item, such as an array, record, or union, each component value of the expression is printed, along with the appropriate subscript, record field name, etc.

The space between **print** and **/** may be omitted. If no *print-format-letter* is given, *expression* is printed in a format corresponding to the data type of the expression in the currently defined language.

The printed value is given a value history number (see "Value History" on page 3-32), indicated in the output by $ followed by the history number.

If the value printed contains an array or a character string, the number of array elements and characters will be limited to the values set by the **set-limits** command (see "set-limits" on page 7-46).

NOTE

For ease in debugging C and C++ programs, the **print** command treats expressions of type 'char *' specially. Whenever **print** prints the value of a 'char *' pointer, it also prints the string it points to, inside double-quote marks; **print** assumes the string is terminated by a null byte.

Most other commands that print expressions or variables also treat 'char *' pointers in this manner.

Examples:

```
(local) (12) p/x var_name*4
(local) (12) p array_name
```

The first example prints, in hexadecimal, a number equal to four times the value of `var_name`, for process 12. The second example prints the value of each member of the array `array_name` in a format based on the data type of `array_name`, for process 12.

set

Evaluate a language expression without printing its value.

set *expression*

expression

A language expression (see "Expression Evaluation" on page 3-20).

This command is similar to the **print** command (see "print" on page 7-66), in that it

evaluates a language expression for each process specified in the qualifier. However, `set` does not accept a format specifier, print the value of the expression, or place the value of the expression in the value history. It is useful for doing assignments to language objects (e.g., memory addresses preceded by the C language cast syntax, variables, and array elements) and convenience variables, as well as for performing calls to subprograms whose return value is unimportant.

Examples:

```
(local) set $i = 98
(local) (27) set vector[5] = x * 2.5
(local) set *(int *)0x1234 = 0xabcd0123
(local) set routine(3,4)
```

The first example assigns the value `98` to the convenience variable `$i`. The second example assigns the value of `x * 2.5` to element five of array `vector`, in process 27. The third example assigns the hexadecimal value `abcd0123` to the hexadecimal absolute memory location `1234`. The final example performs a call to the subprogram `routine`.

x

Print the contents of memory beginning at a given address.

x [/[*repeat-count*][*size-letter*][*x-format-letter*]] [*addr-expression*]

repeat-count

Decimal number of consecutive memory units to print, where a unit is defined by the *size-letter* and the *x-format-letter.*

size-letter

One of the following letters specifying the size of each memory unit:

`b`

Each memory unit is one byte (8 bits) long.

`h`

Each memory unit is one halfword (two bytes) long.

`w`

Each memory unit is one word (four bytes) long.

`g`

Each memory unit is one giant word (eight bytes) long.

The *size-letter* may appear either before or after the *x-format-letter.*

x-format-letter

One of the following letters specifying the format in which to print the contents of memory:

`a`

Print as an integer in hexadecimal and as an address relative to a program symbol. This format ignores *size-letter* and always uses `w`.

`c`

Print as character constants. This format ignores *size-letter* and always uses `b`.

`d`

Print as signed integers in decimal format.

`f`

Print as floating-point values.

`i`

Print as machine instructions in assembler syntax, using the length of each instruction as the unit size. A *repeat-count* given with this format indicates how many instructions to print.

`o`

Print as unsigned integers in octal format.

`s`

Print as a null-terminated string, using the length of the string (including the null byte) as the specified unit size; the *size-letter*, if any, is ignored. A *repeat-count* given with this format indicates how many strings to print.

If the string to be printed is longer than the string limit set by the **`set-limits`** command, the initial characters of the string are printed, with an ellipsis following the closing quote. (see “set-limits” on page 7-46).

`u`

Print as unsigned integers in decimal format.

`x`

Print as unsigned integers in hexadecimal format.

`z`

Print as unsigned integers in hexadecimal format with a display of the corresponding ASCII characters.

addr-expression

An expression yielding a memory address (see "Expression Evaluation" on page 3-20).

The **x** command prints the contents of memory beginning at the address specified by *addr-expression* in each process specified by the qualifier. If an *addr-expression* is not given, the address corresponds to the byte following the end of the memory contents printed in the last **x** command.

The space between **x** and `/` may be omitted. If *repeat-count* is omitted, one memory unit is printed. If either *size-letter* or *x-format-letter* is omitted, the default is the last value used in an **x** command (beginning defaults are `w` and `d`, respectively).

If the **x** command is repeated, memory contents are printed using the same *repeat-count*, *size-letter*, and *x-format-letter* as in the previous **x** command, and the beginning address corresponds to the byte following the end of the memory contents printed in the previous command.

A `0` precedes octal numbers. A `0x` precedes hexadecimal numbers. Thus decimal 64 would appear in hexadecimal as `0x40` and in octal as `0100`.

The *x-format-letter* `z` produces a hexadecimal display *without* the leading `0x` prefix. The character display shows non-printable characters replaced by `.` (period). Here, *printable* is determined by the current locale. The display of characters is framed in `|` and `|`.

After an **x** command, the convenience variables `$_` and `$__` are set and ready to use in expressions (see "Predefined Convenience Variables" on page 7-6). The convenience variable `$_` is set to address of the last memory unit examined. The convenience variable `$__` is set to the contents and type of the last memory unit examined.

Examples:

```
(local) (14544) x/4i $pc
7  @B=  0x1000271c <factorial+28>: li r3,1
7       0x10002720 <factorial+32>:  lwz r16,0x40(r1)
7       0x10002724 <factorial+36>:  lwz r13,0x58(r1)
7       0x10002728 <factorial+40>:  mtlr r13
```

For the process with process id 14544, print memory as four machine instructions starting with the address of the current program counter. See "Source Line Decorations" on page 7-63 for a description of the characters at the beginning of each line of this format.

```
(local) x /4wx 0x40a188
0x0040a188: 0x77767574 0x73727170 0x6f6e6d6c 0x6b6a6968
(local) x /8bz 4235656
0x0040a188: 77 76 75 74 73 72 71 70 |wvutsrqp|
(local)
0x0040a190: 6f 6e 6d 6c 6b 6a 69 68 |onmlkjih|
(local) p $_ - 4235656
17: $_ - 4235656 = 0xf
(local) p $__
$18: $__ = 104 'h'
```

Print memory as four words (four-byte memory units) starting at hexadecimal address `0x0040a188` as unsigned integers in hexadecimal format with `0x` prefixes.

Print memory as eight bytes (one-byte memory units) starting at the same address expressed in decimal (`4235656`) as unsigned integers in hexadecimal format with a display of the printable characters.

Print in the same format and repeat count starting at the next address (`0x0040a190`).

Print an expression `$_ - 4235656` to show the relative difference between the address of the last memory unit printed `$_ - 4235656` and address of the first memory unit two commands ago `4235656`.

Print expression `$__` to show the value of the last memory unit printed.

output

Print the value of a language expression with minimal output.

`output` [`/`*print-format-letter*`]` *expression*

print-format-letter

A letter specifying the format in which to print the expression, as described in the **`print`** command (see “print” on page 7-66).

expression

A language expression (see “Expression Evaluation” on page 3-20).

`output` prints the value of a language expression for each process specified by the qualifier in the same manner as the **`print`** command, except that a newline is not printed, the value is not entered in the value history, and the `"$`*history-number* `= "` string does not prefix the output.

The space between **`output`** and `/` may be omitted. If no *print-format-letter* is given, *expression* is printed in a format corresponding to the data type of the expression.

echo

Print arbitrary text.

`echo` *text*

text

Arbitrary text to be printed, up to the end of the line. Non-printing characters may be represented with C language escape sequences, such as ’\n’ for new-line.

This command prints the given text. It is intended as an adjunct to the other commands which print information about the program, so that the output can be customized to whatever is desired.

A backslash (’\’) may be used to correctly print leading and trailing spaces. In other

words, a backslash may be used at the beginning of *text* to print leading spaces appearing after the backslash, and one may be used at the end of *text* to print the spaces appearing before the backslash. The backslash characters themselves are not printed.

Note that a newline is not printed unless the newline sequence ('\n') is included.

Examples:

```
(local) echo \  Text with two leading spaces and a newline\n
(local) echo      A backslash (\\) and the number three (\063)
```

The first example prints " Text with two leading spaces and a newline", followed by a newline. The second example prints "A backslash (\) and the number three (3)", but does not print a newline.

display

Add to the list of expressions to be printed each time the process stops.

display [[/*print-format-letter*] *expression*]
display /[*repeat-count*][*size-letter*][*x-format-letter*] *addr-expression*

print-format-letter

A letter specifying the format in which to print the expression, as in the **print** command (see "print" on page 7-66).

expression

A language expression (see "Expression Evaluation" on page 3-20).

repeat-count

Decimal number of consecutive memory units to print, where a unit is defined by the *size-letter* and the *x-format-letter.*

size-letter

A letter specifying the size of each memory unit, as described in the **x** command (see "x" on page 7-68). The *size-letter* may appear either before or after the *x-format-letter.*

x-format-letter

A letter specifying the format in which to print the contents of memory, as described in the **x** command (see "x" on page 7-68).

addr-expression

An expression yielding a memory address (see "Expression Evaluation" on page 3-20).

The display item list contains language and memory address expressions which will be used to print expression values or contents of memory, respectively, each time one of the specified processes in the qualifier stops (hits a breakpoint, receives a signal, etc.).

display adds a language or memory address expression to the list.

In order to determine whether the given expression is a language or address expression, the parameters before the expression are first examined. If a *repeat-count* or *size-letter* is given, or if either of the *x-format-letter*s 's' or 'i' is given, then the expression is treated as an *addr-expression*. Otherwise, the expression is treated as a language *expression*.

When one of the processes specified by the qualifier stops, each enabled item in the display item list is evaluated. The indicated expression value or memory location is displayed, each item beginning on a new line. Each display item has an item number, followed by the text of the expression and then the expression's value or the contents of memory. If a language expression for an item cannot be evaluated in the currently defined language, output will not appear for that item; however, a summary of the unevaluated items will appear at the end of the **display** output.

The space between **display** and `/` may be omitted. If no *print-format-letter* is given for a language expression, *expression* is printed in a format corresponding to the data type of the expression at the time the process stops. If *repeat-count* is omitted, one memory unit will be printed. If *size-letter* or *x-format-letter* is omitted, the defaults are `w` and `d`, respectively.

If **display** is entered on a line by itself, the current values of the expressions or contents of memory for each item on the display list are printed. To simply see the expressions themselves, use the **info display** command (see "info display" on page 7-123).

Examples:

```
(local) (12) display/x var_name
(local) (12) display/4d 0x1234
```

If these commands are entered, then each time process 12 stops, the value of `var_name` will be printed in hexadecimal on one line, and four words of memory starting at hexadecimal address `1234` will be printed on the next line.

undisplay

Disable an item from the display expression list.

undisplay *item_number* `...`

item_number

An item number of an item to be disabled in the list of expressions to be printed each time the program stops, as specified in previous **display** commands (see "display" on page 7-72).

The **undisplay** command disables the given items in each of the processes specified by the qualifier. The associated expressions or memory locations cease to be displayed when the corresponding process stops, until you enable them again using the **redisplay** command (see "redisplay" on page 7-74). The effect of the qualifier on this command is to limit the items to be disabled to only those that occur in the specified processes.

Item numbers prefix each displayed language expression and memory section. The item numbers also may be viewed by entering the **info display** command (see "info display" on page 7-123).

redisplay

Enable a display item.

redisplay *item_number* ...

item_number

An item number of an item to be enabled in the list of expressions to be printed each time the program stops, as specified in previous **display** commands (see "display" on page 7-72).

The **redisplay** command enables the specified display items so that they once again print data when the corresponding process stops. The **redisplay** command reverses the effect of the **undisplay** command. The effect of the qualifier on this command is to limit the items to be enabled to only those that occur in the specified processes.

Item numbers prefix each displayed language expression and memory section. The item numbers also may be viewed by entering the **info display** command (see "info display" on page 7-123).

printf

Print the values of language expressions using a format string.

printf *format-string*[, *expression* ...]

format_string

A string within quotes containing text to be printed and print formats for expressions to be printed.

expression

A language expression (see "Expression Evaluation" on page 3-20).

printf prints user-specified text plus, optionally, values of language expressions evaluated in the currently defined language, for each process specified in the qualifier. This command acts the same as the C language library routine **printf (3C)**, with the exception of the '%n' format descriptor. As in that routine, each print format (i.e., substring beginning with '%' and or width specifier '*') in the *format-string* corresponds to one language expression in the specified list. The number of language expressions entered must match the number of print formats.

If a '%n' format descriptor is present in the format string, it is considered a syntax error and the **printf** command is aborted.

Example:

```
(local) (27) printf "The value of var_name = %d.\n", var_name
```

This example prints "The value of var_name = " followed by the decimal value of `var_name` and a newline, for the process with PID 27.

load

Dynamically load an object file, possibly replacing existing routines.

`load` *object*

object

The name of an object file to be loaded into the program.

object is subject to object filename translations (see “translate-object-file” on page 7-21).

This command dynamically loads the designated object file into the address space of the running program. If the loaded file contains any routines which are already defined in the program, the entry points of the existing routines are patched to jump directly to the new routines just loaded. If there are any active stack frames for old routines, the return addresses in the stack still point to the old code. New calls made following the **`load`** will call the new routines.

If you had any breakpoints or other eventpoints set in the old routine, you may need to set equivalent ones again in the new routine (the old ones are still there, but since the old routine will never be called again, you will probably never hit any of them).

The primary purpose of this command is to allow you to replace an existing routine with a new version, avoiding the overhead of forcing you to stop debugging the program, relink it, and rerun to get back to the point of interest.

This command must be used with care. If the new object file contains any global data definitions, you are very likely to wind up with an erroneous program in which old routines refer to the original data locations and new routines refer to the newly loaded data definitions. Patching the old routine entry points to jump to the new routine definitions is simple, but it is not possible to locate all the places that might refer to data items defined in the object file, so loading object files that define static data items is likely to generate unexpected results.

If the object file refers to other routines or external data items that are not already defined in the program file, you are told about the undefined symbols, and the object file is not loaded. If you load an object file that defines new symbols, they are added to the symbol table for the program, so subsequent loads may refer to the new names.

This command checks for obvious problems with the new object file and warns you of anything that is likely to be a mistake, but it loads the new object anyway.

vector-set

Set the value of a vector.

`vector-set` *l-value* `=` *component*`,` *component* `...`

`vector-set` *l-value* `=` *repeat-count*`,`*component*

The arguments to **`vector-set`** are all expressions separated by commas (`','`). The expressions may not contain commas.

l-value

A vector variable, array element, register, etc., which can be assigned. The *l-value* may be followed by either a comma or an equals sign (`'='`). This expression may not contain an equals sign (`'='`) or a comma (`','`).

component

One component of the vector.

repeat-count

The number of times to repeat the following component.

This command creates a vector value by concatenating the component values and assigns the value to the specified *l-value*, which must be a vector, for each process in the command qualifier. **`vector-set`** is meaningful only on the Power Hawk 700 Series.

The command operates based on the number of arguments *following* the *l-value*. If there are 2 arguments, then the first argument is taken as a *repeat-count.* The value of the component is replicated to make *repeat-count* components. The value of the *repeat-count* must be one of `4`, `8` or `16`. If there are more than 2 arguments, then each argument is a component.

The format of each component is determined by the number of components.

Number of Components	Component Format
16	1 byte integer
8	2 byte integer
4	4 byte integer or float

For the 4-byte case, the component type, integer or floating-point, is determined by the type of the component value. NightView warns you if the components are not all of the same type. For the integer formats, each component is implicitly cast to an unsigned integer.

Examples:

```
(local) vector-set my_vec = 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1
(local) vector-set my_vec = 16, 1
```

Both of these commands set `my_vec` to 16 bytes, with each byte set to `1`.

```
(local) vector-set $v0 = 1.0, 2.0, 0.0, 4.0
```

Set `$v0` to these floating-point values.

```
(local) vector-set $v0 = f(0), f(1), f(2), f(3)
```

Set `$v0` to the results of calling function `f` with various values. The type of the 4 components is detemined by the return type of `f`.

Manipulating Eventpoints

This subsection describes the various commands that are used to set and modify eventpoints.

Some of the commands which operate on breakpoints also operate on patchpoints, tracepoints, monitorpoints, agentpoints, and watchpoints as well. The following table indicates which types of eventpoints may be affected by which commands:

Table 7-7. Eventpoint Commands

Command Name	What the Command Applies to					
	Breakpoints	**Patchpoints**	**Tracepoints**	**Agentpoints**	**Monitorpoints**	**Watchpoints**
name	X	X	X	X	X	X
clear	X	X	X	X	X	
commands	X				X	X
condition	X	X	X	X	X	X
delete	X	X	X	X	X	X
disable	X	X	X	X	X	X
enable	X	X	X	X	X	X
ignore	X	X	X	X	X	X
tbreak	X					
tpatch		X				

Eventpoint Modifiers

An *eventpoint modifier* modifies the setting of eventpoints in a program.

The modifiers come after the eventpoint commands as follows:

command [*modifier* ...]

The eventpoint modifiers are:

`/delete`

Causes the eventpoint to be deleted after the first hit. This eventpoint modifier is valid only with the **`breakpoint`**, **`enable`** and **`watchpoint`** commands (see "breakpoint" on page 7-79, "enable" on page 7-92, and "watchpoint" on page 7-95).

`/disabled`

Causes the eventpoint to be created in a disabled state. You must use the **`enable`** command to activate the eventpoint (see "enable" on page 7-92).

name

Give a name to a group of eventpoints.

`name` `[/add]` *name* `[[-]` *eventpoint-spec*`] ...`

`/add`

Add the eventpoints to the named set, rather than redefining the set.

name

The name of the set of eventpoints to be defined. This must not be the same as the name of any dialogue you currently have, or of any process family that is currently defined. The name must consist only of alphanumeric characters and underscores and must begin with an alphabetic character. The name may be of arbitrary length.

eventpoint-spec

An eventpoint specifier. See "Eventpoint Specifiers" on page 7-12.

The total set of eventpoints is accumulated by scanning the *eventpoint-spec* arguments left to right. An argument is added to the set unless it is preceded by a '-', in which case it is subtracted from the set accumulated so far.

If no *eventpoint-spec* is given, then this command removes any previous definition of *name*.

Any qualifier applied to this command has the effect of restricting the set of eventpoints named to those which exist in the processes specified by the qualifier.

Examples:

```
(local) name evpt1 12 25 18
(local) name evpt2 evpt1 99
(local) name evpt1 evpt1 16
```

The first command gives the name *evpt1* to three eventpoints identified by eventpoints 12, 18, and 25. The second command gives the name *evpt2* to the three eventpoints in *evpt1* plus eventpoint 99. The third command extends the definition of *evpt1* to include eventpoint 16; thus *evpt1* is a synonym for four eventpoints: 12, 16, 18, and 25. Note that extending *evpt1* has no effect on *evpt2*, which still consists of eventpoints 12, 18, 25, and 99.

Using the names defined in the previous examples, the use of a minus sign on arguments can be illustrated by the following examples:

```
(local) name evpt3 evpt1 evpt2 -12
(local) name evpt3 evpt1 -12 evpt2
```

The first command defines `evpt3` to be the eventpoints 16, 18, 25, and 99. In contrast, the second command defines `evpt3` to be the eventpoints 12, 16, 18, 25, and 99. In this case, the argument `-12` removed eventpoint 12 from the set accumulated from `evpt1`, but the `evpt2` argument adds that eventpoint back in.

breakpoint

Set a breakpoint.

`breakpoint` [*eventpoint-modifier*] [`name=`*breakpoint-name*]
[[`at`] *location-spec*] [`if` *conditional-expression*]

Abbreviation: **`b`**

eventpoint-modifier

Specifies the breakpoint modifier. See “Eventpoint Modifiers” on page 7-78.

`name=`*breakpoint-name*

Gives a name to the breakpoint for later reference. (see “name” on page 7-78). If *breakpoint-name* is already defined, then this command adds the newly created breakpoints to the list of eventpoints associated with the name.

location-spec

Specifies the breakpoint location. (see “Location Specifiers” on page 7-9).

If *location-spec* is omitted, then the location used is the next instruction to be executed in the current stack frame. See “Current Frame” on page 3-25.

`if` *conditional-expression*

Specifies a breakpoint condition. The language and scope of the expression is determined by the location at which the breakpoint is set (see “Scope” on page 3-24 and “Context” on page 3-24). See also “Expression Evaluation” on page 3-20.

NOTE

The `at`, `if`, and `name` keywords may not be abbreviated in this command.

`breakpoint` sets a breakpoint in each of the processes specified by the qualifier. This causes the program to suspend execution at the breakpoint location. An optional condition may be applied to the breakpoint which causes execution to be suspended only if the condition evaluates to TRUE. The conditional expression is evaluated in the user program when the breakpoint location is reached (unless the breakpoint is currently being ignored, see "ignore" on page 7-93).

If more than one breakpoint is set (through the use of more than one process in the qualifier) then each breakpoint in each process is assigned a unique breakpoint number.

You can specify debugger commands to be executed when a breakpoint is hit. See "commands" on page 7-89.

It is possible (and sometimes useful) to set more than one breakpoint at the same location in a process. Perhaps you have two breakpoints set at the same place and each has its own set of commands. By enabling only one of the two breakpoints at a time, you can effectively toggle the set of commands that gets executed when the process reaches that location.

If more than one breakpoint is set at the same location in a given process, then the oldest breakpoint with an ignore count of zero and a condition that evaluates to TRUE will be the first breakpoint responsible for stopping the process. After this breakpoint has stopped the process, before continuing on to the next instruction, NightView will check for any remaining breakpoints at that location which may stop the process. If there are any, then the process will stop at least once more (at the same location) before continuing on to the next instruction.

Example:

```
(local) (441 115) break name=loop sort.c:42
```

This example sets two breakpoints at line 42 of the file named **`sort.c`** and associates both breakpoints with the name 'loop'. One of the breakpoints is set in process 441 and the other breakpoint is set in process 115. Each of the two breakpoints is assigned a unique breakpoint number.

patchpoint

Install a small patch to a routine.

`patchpoint` [*eventpoint-modifier*] [name=*patchpoint-name*]
[[at] *location-spec*] `eval` *expression*

Insert an expression in the program.

`patchpoint` [*eventpoint-modifier*] [name=*patchpoint-name*]
[[at] *location-spec*] `goto` *location-spec*

Insert a branch in the program.

eventpoint-modifier

Specifies the patchpoint modifier. See “Eventpoint Modifiers” on page 7-78.

`name=`*patchpoint-name*

Patchpoints are assigned event numbers, and the `name=` syntax as well as the **`name`** command (see “name” on page 7-78) may be used to give them names. See “Manipulating Eventpoints” on page 7-77.

`at` *location-spec*

Specify the exact point in the program to execute the patchpoint (see “Location Specifiers” on page 7-9). The patchpoint is executed immediately prior to any existing code at this location.

If *location-spec* is omitted, then the location used is the next instruction to be executed in the current stack frame. See “Current Frame” on page 3-25.

`eval` *expression*

This variant of the **`patchpoint`** command specifies an expression to insert in the program at the designated *location-spec*. Ada, C, and C++ programmers should note that this is an expression and not a statement; therefore, it does *not* end with a semicolon. (The concept of *expression* is extended to include assignments and procedure calls in Ada and Fortran.) See “Expression Evaluation” on page 3-20.

`goto` *location-spec*

This variant of the **`patchpoint`** command specifies a location to branch to when the program reaches the point of the patchpoint. The instruction originally at the patchpoint location will not be executed.

Note that if an expression is used as a *location-spec*, the expression is evaluated only once for each process in the qualifier. For example, if the *location-spec* is `*$lr`, the value of register `lr` in the *current context* is used as the location to branch to.

NOTE

The keywords `name`, `at`, `eval`, and `goto` may not be abbreviated in this command.

Once an `eval` patchpoint is installed, the language expression will be executed each time control reaches *location-spec* in the program. After the patchpoint is executed, the original instruction will also execute.

Once a `goto` patchpoint is installed, the branch will be executed before the patched instruction each time execution reaches *location-spec* in the program. It is important to note that the original instruction is not executed if the patchpoint is *hit* (that is, depending

on the enabled status, the ignore count and any eventpoint condition on the patchpoint). If the patchpoint is not hit, the original instruction is executed normally.

When patching in a `goto`, you should be aware that the compiler has probably generated code which expects certain register contents and altering the flow of control in your program can very easily send it to a new location with unexpected values in registers, so the `goto` patchpoint should be used only when you are sure you know all the consequences.

You may attach a condition or ignore count to both kinds of patchpoints, using the **`condition`** (see "condition" on page 7-89) or **`ignore`** (see "ignore" on page 7-93) commands. This suppresses execution of the patched expression unless the ignore count is zero and the conditional expression evaluates to TRUE.

Patchpoints are implemented by modifying the executable code for the program, so they will remain in effect until the program exits, even if you **`detach`** the debugger from the program, unless the patchpoint was disabled when you detached (see "detach" on page 7-32 and "disable" on page 7-91). Note that the disk copy of the program is not modified; you must edit your source, recompile and relink to make a permanent modification to the program.

If multiple patchpoints are made at the same point in the program, they will all be executed in the order they were applied. This is especially important to note for **`goto`** patchpoints, because once a **`goto`** is executed, any subsequent patchpoints (or other kinds of eventpoints, such as breakpoints and tracepoints) at that same location will not be executed. If a **`goto`** patchpoint is not hit (because it was disabled, or the ignore count or condition caused it to be skipped), then the branch will not be taken and subsequent patchpoints will be executed, as well as the original patched instruction.

Example:

```
(local) patchpoint file.c:12 eval i=0
```

This C example patches the code to initialize the variable `i` to zero immediately prior to executing line 12 in the file **`file.c`**. Note that no semicolon appears in this example.

set-trace

Establish tracing parameters.

`set-trace` [`eventmap`=*event-map-file*]

`eventmap`=*event-map-file*

Names the file that contains the mapping between symbolic trace-event tags and numeric trace-event IDs. This should be the same as the event-map file passed to **`ntrace(1)`**.

The **`set-trace`** command is used to specify information that may be required before any tracepoints may be set in a process (see "tracepoint" on page 7-83).

If you want to use symbolic trace-event tags rather than numeric trace-event IDs as the *event-id* parameter of the **`tracepoint`** command, then you must specify an event-map file. You may specify multiple event-map files by repeating the `eventmap` parameter.

As long as the files do not contain conflicting definitions for tags, all the tags will be defined for use as trace-event identifiers.

tracepoint

Set a tracepoint.

tracepoint [*eventpoint-modifier*] *event-id* [name=*tracepoint-name*]
[[at] *location-spec*] [value=*logged-expression*]
[**if** *conditional-expression*]

eventpoint-modifier

Specifies the tracepoint modifier. See “Eventpoint Modifiers” on page 7-78.

event-id

An identifier for the trace event to be traced by **NightTrace**. This is either a numeric trace-event ID or a symbolic trace-event tag obtained from the event-map file specified by the *eventmap* parameter of the **set-trace** command (see “set-trace” on page 7-82).

name=*tracepoint-name*

Gives a name to the tracepoint for later reference. See “name” on page 7-78. If *tracepoint-name* is already defined, then this command adds the newly created tracepoints to the list of eventpoints associated with the name.

location-spec

Specifies the tracepoint location. See “Location Specifiers” on page 7-9.

If *location-spec* is omitted, then the location used is the next instruction to be executed in the current stack frame.

value=*logged-expression*

Specifies that the value of *logged-expression* should be recorded with the trace event. The expression is evaluated in the user program, so it obeys the same rules that conditional and patchpoint expressions do. See “Expression Evaluation” on page 3-20.

if *conditional-expression*

Specifies a tracepoint condition. The language and scope of the expression is determined by the location at which the tracepoint is set (see “Scope” on page 3-24 and “Context” on page 3-24). See also “Expression Evaluation” on page 3-20.

NOTE

The name, value, and if keywords may not be abbreviated in this command.

The **tracepoint** command sets a tracepoint in each of the processes specified by the qualifier. This causes the program to emit special tracing output at the tracepoint location. An optional condition may be applied to the tracepoint which causes tracing to be performed only if the condition evaluates to TRUE. The conditional expression *conditional-expression* is evaluated in the user program when the tracepoint location is reached (unless the tracepoint is currently being ignored, see "ignore" on page 7-93).

Tracepoints set in a process remain set even if you **detach** the debugger from the program, unless the tracepoint was disabled at the time you detached (See "detach" on page 7-32 and "disable" on page 7-91).

NOTE

The **ntrace(3X)** routines must have been linked into the program when it was built. If the program does not initialize tracing, then you must initialize tracing manually by evaluating expressions that contain calls to the appropriate trace routines (`trace_start` followed by `trace_open_thread`).

The debugger does *not* start the **ntraceud(1)** monitor process. You must do that manually (see "NightTrace Monitor" on page 3-36).

If more than one tracepoint is set (through the use of more than one process in the qualifier) then each tracepoint in each process is assigned a unique tracepoint number.

It is possible (and sometimes useful) to set more than one tracepoint at the same location in a process. Perhaps there is more than one noteworthy event that takes place at the same location in your program. If more than one tracepoint is set at the same location in a given process, then the tracepoints at that location are recorded in the order they were defined.

Example:

```
(local) (441 115) tracepoint 27 name=loop_trace sort.c:42
```

This example sets two tracepoints at line 42 of the file named **sort.c** and associates both tracepoints with the name 'loop_trace'. One of the tracepoints is set in process 441 and the other tracepoint is set in process 115. Each of the two tracepoints is assigned a unique tracepoint number. The ID of the trace event to trace is given by the number 27.

monitorpoint

Monitor the values of one or more expressions at a given location.

monitorpoint [*eventpoint-modifier*] [name=*monitorpoint-name*]
[[at] *location-spec*]

eventpoint-modifier

Specifies the monitorpoint modifier. See “Eventpoint Modifiers” on page 7-78.

`name=`*monitorpoint-name*

Gives a name to the monitorpoint for later reference. See “name” on page 7-78. If *monitorpoint-name* is already defined then this command adds the newly created monitorpoints to the list of eventpoints associated with the name.

location-spec

Specifies the monitorpoint location. See “Location Specifiers” on page 7-9.

If *location-spec* is omitted, then the location used is the next instruction to be executed in the current stack frame.

The **`monitorpoint`** command sets a monitorpoint in each of the processes specified by the qualifier. Each line following the **`monitorpoint`** command must be a special form (described later) of **`print`** command; each **`print`** command specifies an expression to be evaluated and monitored at the location of the monitorpoint. To end the list of **`print`** commands, type **`end monitor`** on a line by itself.

In the command-line and simple full-screen interfaces, the prompt changes to `>` while you are entering the attached **`print`** commands. See “Command Syntax” on page 7-1.

When the monitorpoint is executed, the expressions specified in the attached commands will be evaluated and their values saved in a location reserved by NightView. The monitored values are displayed periodically in a monitor display area; see “Monitor Window” on page 3-27. For a more detailed description of monitorpoints, see “Monitorpoints” on page 3-10.

The syntax of the commands attached to a monitorpoint is:

`print [``/`*print-format-letter*`]`** **`[`**`id="`*string*`"]`** *expression*

This syntax is identical to the **`print`** NightView command (see “print” on page 7-66), with the addition of the optional `id="`*string*`"` argument. The *string*, if specified, is used to identify the monitored expression in the monitor display area. If you do not specify the `id=` parameter, the text of the expression itself is used as the identifying string. Note that you may not abbreviate the `id=` keyword to anything shorter (like "i").

Once you have created a monitorpoint, you can change the set of commands attached to it (and thus the expressions being monitored) using the **`commands`** command. See “commands” on page 7-89.

Example:

```
(local) monitorpoint file.c:12
> print variable1
> print id="Velocity (ft/sec)" variable2
> end monitor
```

In this example, two variables will be monitored at line 12 of **`file.c`**. The first

variable, `variable1`, will be displayed using its name as the identifying string. The second variable, `variable2`, will be displayed with the string `Velocity (ft/sec)`.

mcontrol

Control the monitor display window.

`mcontrol` `{display | nodisplay} [`*monitorpoint-spec* `...]`

Turn on or off the display of individual monitorpoints in the monitor window.

`mcontrol` `delay` *milliseconds*

Set the milliseconds to delay between monitor window updates.

`mcontrol` `{off | on | stale | nostale | hold | release}`

Toggle a monitoring parameter.

Abbreviation: **`hold`**

This is an abbreviation for **`mcontrol`** `hold`.

Abbreviation: **`release`**

This is an abbreviation for **`mcontrol`** `release`.

`display nodisplay`

These keywords are used to enable or disable the display of specific monitorpoints in the monitor window. The monitorpoints appearing in the argument and in the processes specified by the qualifier are either added to or removed from the monitor window display area. This does not affect the monitorpoint itself, it simply determines which monitorpoints are shown in the window. See "monitorpoint" on page 7-84.

`on off`

These keywords turn the monitor window on or off. You may wish to turn off the monitor window to reclaim screen space, then turn it back on later. Turning `off` the window also does a `hold`, but turning the window `on` does not implicitly do a `release`.

`stale nostale`

The monitor window normally displays a stale data indication next to each value. The `nostale` keyword causes the monitor window to display blank space rather than one of the stale data indicators. The indicators may be turned back on with the `stale` keyword.

`hold release`

The `hold` and `release` keywords are used to hold or release updates of the monitor window. When the window is held, the values displayed in the moni-

tor window will no longer change (the processes containing the values are not affected, they continue to run). The `release` keyword allows the monitor window to start updating the values again.

Interrupting the debugger implicitly causes the Monitor Window to stop updating. See "Interrupting the Debugger" on page 3-30.

`delay`

The monitor window normally waits one second (1000 milliseconds) between updates. A different number of milliseconds may be specified following the `delay` keyword. If you tell it to wait zero milliseconds, it updates the monitor window as fast as it possibly can.

All of the **`mcontrol`** parameters allow you to control various aspects of the monitor display window (see "Monitor Window" on page 3-27).

You may not combine parameters on the **`mcontrol`** command. Only one keyword may be used in one invocation of the command. The command qualifier is only used when the `display` or `nodisplay` keywords are used to specify a list of monitorpoints.

agentpoint

Insert a call to a debug agent at a given location.

`agentpoint` [*eventpoint-modifier*] [`name=`*agentpoint-name*]
[[`at`] *location-spec*]

eventpoint-modifier

Specifies the agentpoint modifier. See "Eventpoint Modifiers" on page 7-78.

`name=`*agentpoint-name*

Gives a name to the agentpoint for later reference. See "name" on page 7-78. If *agentpoint-name* is already defined then this command adds the newly created agentpoints to the list of eventpoints associated with the name.

location-spec

Specifies the agentpoint location. See "Location Specifiers" on page 7-9.

If *location-spec* is omitted, then the location used is the next instruction to be executed in the current stack frame.

Once an agentpoint is installed, a call to a special debug agent (see "Debug Agent" on page 3-17) will be executed each time control reaches *location-spec* in the program. After the debug agent is executed, the original instruction will also execute. The debug agent allows you to debug your process while it is running.

You may attach a condition or ignore count to an agentpoint, using the **`condition`** (see "condition" on page 7-89) or **`ignore`** (see "ignore" on page 7-93) commands. This suppresses execution of the debug agent unless the ignore count is zero and the conditional expression evaluates to TRUE.

Agentpoints are implemented by modifying the executable code for the program, so they remain in effect until the program exits, even if you **detach** the debugger from the program, unless the agentpoint was disabled when you detached (see "detach" on page 7-32 and "disable" on page 7-91).

For best results, the debug agent should be executed frequently. If you cannot find just one place in your program that is executed frequently enough, you may create multiple agentpoints, each at a different location. You can enable and disable each agentpoint independently.

clear

Clear all eventpoints at a given location.

clear [[**at**] *location-spec*]

location-spec

Specifies the location from which all eventpoints are to be removed. See "Location Specifiers" on page 7-9.

If *location-spec* is omitted, then the location used is the next instruction to be executed in the current stack frame. See "Current Frame" on page 3-25.

clear removes all eventpoints at the specified location in each process. Once an eventpoint has served its purpose, the eventpoint may be removed by using the **clear** or **delete** commands (see "delete" on page 7-90). Both commands remove an eventpoint. **clear** removes eventpoints based on where they are in the process. **delete** removes eventpoints specified by name or by eventpoint-number.

NOTE

A location specifier may sometimes designate multiple locations (see "Location Specifiers" on page 7-9). Hence, it is possible for a single eventpoint to be set at multiple locations. If any of the locations at which an eventpoint is set match any of the locations implied by the location specifier for the **clear** command, then that eventpoint will be removed (from *all* of its corresponding locations).

It is unnecessary to clear a breakpoint in order to continue execution after the breakpoint has stopped the program.

Example:

```
(local) clear sort.c:42
```

This example removes all eventpoints set at line 42 of the file named **sort.c** in each of the processes specified by the default qualifier.

commands

Attach commands to a breakpoint, monitorpoint, or watchpoint.

`commands` *eventpoint-spec*

eventpoint-spec

The breakpoints, monitorpoints, or watchpoints to which the given commands are attached. See “Eventpoint Specifiers” on page 7-12.

The **`commands`** command attaches the given list of commands to the given breakpoints, monitorpoints, or watchpoints in processes specified by the qualifier. Each line following the **`commands`** command-line should be a command to associate with the eventpoints. To end the list of commands, type ’end’ on a line by itself.

Each of the commands given is implicitly qualified with the PID of the process associated with the eventpoint.

In the command-line and simple full-screen interfaces, the prompt changes to **>** while you are entering this command. See “Command Syntax” on page 7-1.

If the first line given is ’silent’, then the usual message that is printed when a breakpoint or watchpoint stops the process will be suppressed. Furthermore, the ’silent’ command will also prevent the current source line from being listed, and will prevent any displays from being updated. The ’silent’ command is valid only when attached to a breakpoint or watchpoint and is useful for breakpoints or watchpoints that are intended only to print a specific message and then resume execution.

Certain commands (such as **`continue`**, **`resume`**, and **`signal`**), once executed, will automatically terminate the command stream associated with a set of commands that were attached to a breakpoint or watchpoint using the **`commands`** command. See “continue” on page 7-97, “resume” on page 7-98, and “signal” on page 7-104.

Although you can use the **`commands`** command to attach commands to breakpoints, monitorpoints, or watchpoints, the eventpoints specified on the command line must be all of the same type. Also note that the commands allowed for monitorpoints are restricted to **`print`** commands. See “monitorpoint” on page 7-84.

condition

Attach a condition to an eventpoint.

`condition` *eventpoint-spec* [*conditional-expression*]

eventpoint-spec

The eventpoints associated with the condition. See “Eventpoint Specifiers” on page 7-12.

conditional-expression

The condition to be associated with the eventpoints. See "Expression Evaluation" on page 3-20.

The simplest type of breakpoint is one which stops the program each time it is encountered (an *unconditional breakpoint*). Often however, you may wish to stop the program at a given location only after a certain event has occurred or when a specified condition has been met (a *conditional breakpoint*). The **condition** command may be used to attach a condition to a breakpoint.

In a similar manner, conditions may also be attached to tracepoints, monitorpoints, agentpoints, patchpoints, and watchpoints, causing the associated action to take effect only when the attached condition evalutes to TRUE.

The **condition** command attaches the condition *conditional-expression* to one or more eventpoints in the processes specified by the qualifier. If *conditional-expression* is omitted, then any condition attached to the specified eventpoint is removed in each of the processes specified by the qualifier, and the eventpoint becomes an unconditional one. If the specified eventpoint already has a condition attached to it, the existing condition is replaced with *conditional-expression*.

Examples:

```
(local) breakpoint name=loop at foo.c:12
(local) condition loop (index == 0)
(local) condition loop
```

The first **condition** command attaches a condition to the breakpoint named 'loop' so that it only stops the program when the variable 'index' is zero. The second **condition** command removes any condition associated with the breakpoint named 'loop' (thus making it an unconditional breakpoint).

```
(local) trace MyEvent name=trace1 at foo.c:12
(local) condition trace1 (x>12)
```

In this example, a tracepoint named 'trace1' is set, and the condition 'x>12' is attached to the tracepoint. Therefore, the event will be traced only when 'x' is greater than 12.

delete

Delete an eventpoint.

delete [*eventpoint-spec* ...]

Abbreviation: **d**

eventpoint-spec

The eventpoints to be deleted. See "Eventpoint Specifiers" on page 7-12.

delete removes the specified eventpoints in each of the processes specified by the qualifier. Both **delete** and **clear** may be used to delete eventpoints (see "clear" on page 7-88). The difference is that **delete** removes eventpoints specified by name or by

eventpoint-number and **`clear`** removes eventpoints specified by location.

If *eventpoint-spec* is omitted and your safety-level is `unsafe` then *all* eventpoints in the processes specified by the qualifier are removed (see "set-safety" on page 7-49). If *eventpoint-spec* is omitted and your safety-level is `verify`, then you are prompted for confirmation before the eventpoints are removed (see "Replying to Debugger Questions" on page 7-16). If *eventpoint-spec* is omitted and your safety-level is `forbid` then no eventpoints are removed.

The effect of the qualifier on this command is to limit the eventpoints deleted to be only those that occur in the processes specified by the qualifier.

Examples:

```
(local) d loop
(local) d 2 5
```

The first example removes all eventpoints associated with the name 'loop'. The second example removes eventpoints 2 and 5.

disable

Disable an eventpoint.

`disable` [*eventpoint-spec* `...`]

eventpoint-spec

The eventpoints to be disabled. See "Eventpoint Specifiers" on page 7-12.

The **`disable`** command disables the given eventpoints in each of the processes specified by the qualifier. Disabling an eventpoint is not quite the same as removing an eventpoint. When an eventpoint is removed, it is made inoperative and all the information associated with the eventpoint is removed. When an eventpoint is disabled, it is simply made inoperative. It may still be seen, however, if you use the **`info eventpoint`** command (see "info eventpoint" on page 7-115). All information associated with the eventpoint is still retained so that the eventpoint may later be reactivated using the **`enable`** command (see "enable" on page 7-92).

If *eventpoint-spec* is omitted and your safety-level is `unsafe` then *all* eventpoints in the processes specified by the qualifier are disabled (see "set-safety" on page 7-49). If *eventpoint-spec* is omitted and your safety-level is `verify`, then you are prompted for confirmation before the eventpoints are disabled (see "Replying to Debugger Questions" on page 7-16). If *eventpoint-spec* is omitted and your safety-level is `forbid` then no eventpoints are disabled.

The effect of the qualifier on this command is to limit the eventpoints disabled to be only those that occur in the processes specified by the qualifier.

Example:

```
(local) disable  4
(local) (115 441) disable  calvin
(local) (549) disable  8 hobbes 12 14
```

The first example disables eventpoint number 4 in the processes specified by the default qualifier. The second example disables the eventpoints associated with the name 'calvin' in process 115 and in process 441. The third example disables the eventpoints associated with the name 'hobbes' and disables eventpoints numbered 8, 12, and 14 in process 549.

enable

Enable an eventpoint for a specified duration.

enable [/once|/delete] *eventpoint-spec* ...

/once

Specify whether the given eventpoints are to be enabled once only and then immediately disabled after the next time they are hit. There need not be a space between the command name and the '/'.

/delete

Valid only for breakpoints and watchpoints. Specify whether the given breakpoints and watchpoints are to be enabled once only and then immediately deleted after the next time they are executed. There need not be a space between the command name and the '/'.

eventpoint-spec

The eventpoints to be enabled. See "Eventpoint Specifiers" on page 7-12.

The **enable** command enables for the specified duration each of the eventpoints in the processes specified by the qualifier. If neither /once nor /delete is specified, then the given eventpoints are simply enabled. If /once is specified, then the given eventpoints are temporarily enabled. The eventpoints will be disabled again after the next time they are hit. If /delete is specified, then for each process in the qualifier, the given breakpoints and watchpoints are enabled and also marked for deletion. The breakpoints and watchpoints will be deleted after the next time they are hit.

The effect of the qualifier on this command is to limit the eventpoints enabled to be only those that occur in the processes specified by the qualifier.

Examples:

```
(local) enable calvin
(local) enable /once 4 6 23
(local) enable /delete 8 hobbes
```

The first example enables all eventpoints associated with the name 'calvin' in the default qualifier. The second example enables eventpoints number 4, 6, and 23 for once-only execution (the eventpoints will be disabled after the next time they are hit). The third example enables breakpoint number 8, and the breakpoints and watchpoints associated with the name 'hobbes' for deletion (these breakpoints and watchpoints will be deleted after the next time they are hit).

ignore

Attach an ignore-count to an eventpoint.

`ignore` *eventpoint-spec count*

eventpoint-spec

The eventpoints to be ignored. See “Eventpoints” on page 3-8.

count

The number of times to ignore the eventpoint. Specifying an ignore-count of zero has the effect of causing the eventpoints to no longer be ignored. The ignore-count is evaluated in the user's process.

The **`ignore`** command causes the specified eventpoints to be skipped the next *count* times execution reaches them (even if the eventpoint is a conditional eventpoint). This is accomplished by attaching an ignore-count to the given eventpoints. In the case of a breakpoint, any NightView commands associated with the breakpoint will not be executed until the breakpoint is hit.

Example:

```
(local) ignore calvin 4
```

This example causes the eventpoints associated with the name ’calvin’ to be ignored 4 times before they may be hit again.

tbreak

Set a temporary breakpoint.

`tbreak` `[name=`*breakpoint-name*`] [[at]` *location-spec*`]`
`[if` *conditional-expression*`]`

`name=`*breakpoint-name*

Gives a name to the breakpoint for later reference. See “name” on page 7-78. If *breakpoint-name* is already defined then this command adds the newly created breakpoints to the list of eventpoints associated with the name.

location-spec

Specifies the breakpoint location. See “Location Specifiers” on page 7-9.

`if` *conditional-expression*

Specifies an eventpoint condition. The language and scope of the expression is determined by the location at which the breakpoint is set (see “Scope” on page 3-24 and “Context” on page 3-24). See “Expression Evaluation” on page 3-20.

Note: The `at`, `if`, and `name` keywords may not be abbreviated in this command.

Like the **`breakpoint`** command (see "breakpoint" on page 7-79), the **`tbreak`** command sets a breakpoint. The difference between the two is that **`tbreak`** sets a one-time-only breakpoint in each of the processes specified by the qualifier. The breakpoint will be disabled after being hit once.

Example:

```
(local) (115) tbreak sort.c:48
```

This example sets a temporary breakpoint in process 115 at line 48 of the source file **`sort.c`**.

tpatch

Set a patchpoint that will execute only once.

`tpatch` [`name=`*patchpoint-name*`]` [[`at`] *location-spec*`]` `eval` *expression*

Insert an expression in the program that will be executed the next time the patchpoint is hit, then never executed again unless explicitly enabled. See "enable" on page 7-92.

`tpatch` [`name=`*patchpoint-name*] [[`at`] *location-spec*] `goto` *location-spec*

Overwrite an instruction in the program with a branch that will only be taken once. Subsequent execution will ignore the patchpoint and execute the original instruction.

`name=` *patchpoint-name*

Patchpoints are assigned event numbers, and the `name=` syntax as well as the **`name`** command (see "name" on page 7-78) may be used to give them names. See "Manipulating Eventpoints" on page 7-77.

`at` *location-spec*

Specify the exact point in the program to execute the patchpoint. See "Location Specifiers" on page 7-9. The patchpoint is executed immediately prior to any existing code at this location.

If *location-spec* is omitted, then the location used is the next instruction to be executed in the current stack frame. See "Current Frame" on page 3-25.

`eval` *expression*

This variant of the **`patchpoint`** command specifies an expression to insert in the program at the designated *location-spec*. Ada, C and C++ programmers should note that this is an expression and not a statement; therefore, it does *not* end with a semicolon. (The concept of *expression* is extended to include assignments and procedure calls in Ada and Fortran.) See "Expression Evaluation" on page 3-20.

`goto` *location-spec*

This variant of the **`patchpoint`** command specifies a location to branch to when the program reaches the point of the patchpoint. The instruction originally at the patchpoint location will not be executed.

NOTE

The keywords `name`, `at`, `eval`, and `goto` may not be abbreviated in this command.

The **`tpatch`** command is a variant of the **`patchpoint`** command. See "patchpoint" on page 7-80. It works exactly like the patchpoint command, but a temporary patchpoint will automatically disable itself after executing one time. A temporary patchpoint may be enabled later, in which case it will act exactly like a normal patchpoint. See "enable" on page 7-92.

A temporary patchpoint may be useful for patching in initialization code which should only execute once.

watchpoint

Set a watchpoint.

`watchpoint` [*eventpoint-modifier*] [`/once`] [`/read`] [`/write`] [`name=`*watchpoint-name*] [`at`] *lvalue* [`if` *conditional-expression*]

`watchpoint` [*eventpoint-modifier*] [`/once`] [`/read`] [`/write`] `/address` [`name=`*watchpoint-name*] [`at`] *address-expression* {`size` *size-expression* | `type` *expression*} [`if` *conditional-expression*]

eventpoint-modifier

Specifies the watchpoint modifier. See "Eventpoint Modifiers" on page 7-78.

`/once`

The watchpoint is enabled only until the first time it is hit.

`/read`

Watchpoint processing occurs for a read (i.e., a "load") of the specified address. Either or both of `/read` and `/write` may be specified.

`/write`

Watchpoint processing occurs for a write (i.e., a "store") of the specified address. Either or both of `/read` and `/write` may be specified. If neither is specified, the default is `/write`.

`/address`

Indicates this is the *address-expression* form of the command.

`name=`*watchpoint-name*

Gives a name to the watchpoint for later reference. (see "name" on page 7-78). If *watchpoint-name* is already defined, then this command adds the newly created watchpoints to the list of eventpoints associated with the name.

lvalue

An expression that yields an addressable item to watch. For example, *lvalue* may be a variable name or an array element.

address-expression

An expression that yields an address to watch.

`size` *size-expression*

The size of the item to watch.

`type` *expression*

An expression whose type indicates the size of the item to watch. `type` is used only in restart information.

`if` *conditional-expression*

Sets a condition on the watchpoint. The watchpoint is considered to be hit only if *conditional-expression* evaluates to TRUE. The *conditional-expression* is always evaluated in the global scope. *conditional-expression* is evaluated *after* the process has executed the instruction causing the trap.

conditional-expression may refer to the process-local convenience variables `$is` and `$was`. `$was` is the value of the watched item before the process has executed the instruction causing the trap. `$is` is the value of the watched item after the process has executed the instruction causing the trap.

NOTE

The `at`, `if`, `name`, `size` and `type` keywords may not be abbreviated in this command.

`watchpoint` sets a watchpoint in each of the processes specified by the qualifier. This causes the process to stop when it accesses the *lvalue* or *address-expression*. See "Watchpoints" on page 3-11.

You can specify commands to be executed when the watchpoint is hit. See "commands" on page 7-89.

Controlling Execution

This section describes commands used to control the execution of a process.

Most of the commands described in this section cause the processes specified in the qualifier to resume execution and then wait for something to happen. (This is what you usually want when you are debugging a single process.) Only **resume** resumes execution and then returns immediately for another command.

Some of the commands continue until something special happens. For example, **step** continues until control crosses a source line boundary. However, you should be aware that another event, such as a signal or hitting a breakpoint, may cause the process to stop sooner.

If the process stopped because of a signal, then it will receive that signal when the process resumes, subject to the setting of the **handle** command, see "handle" on page 7-105. If you want the process to receive a different signal, or no signal at all, then use the **signal** command. See "signal" on page 7-104.

If you ask to continue execution of a process with any of the commands here, and that process is already executing, then you get a warning message. Any other processes specified by the qualifier are continued.

If a process is stopped at a breakpoint or watchpoint, it is *not* necessary to remove the breakpoint or watchpoint before continuing.

continue

Continue execution and wait for something to happen.

continue [*count*]

Abbreviation: **c**

count

If the *count* argument is specified, the processes will not stop at the current breakpoint or watchpoint again until they have hit it *count* times. This argument is ignored for any processes that are not stopped at breakpoints or watchpoints.

continue causes the processes specified by the qualifier to resume execution at the point where they last stopped. Processes run concurrently. Each process will execute until some event, such as hitting a breakpoint, causes it to stop.

If this command is entered interactively, the debugger does not prompt for any more commands until one of the processes specified by the qualifier stops executing for some reason. Note that only one of the specified processes has to stop for the **continue** command to complete; it does not wait for *all* of the processes to stop. Note also that a process is considered to be stopped the moment it hits a breakpoint or watchpoint; if the breakpoint or watchpoint has commands attached to it, they probably will not execute before you receive a prompt for another command.

If a **continue** command in a breakpoint (or watchpoint) command stream continues execution of the process stopped at that breakpoint or watchpoint, the command stream is terminated; no further commands are executed from that stream. If a **continue** command continues execution of a process that is currently executing another breakpoint (or watchpoint) command stream, the **continue** command does not take effect until that command stream has completed execution. See "Command Streams" on page 3-29.

If a **continue** command continues execution of a process that is currently executing an **on program** or **on restart** command stream, the **continue** command does not take effect until the affected process has been completely initialized by NightView and is ready to be debugged.

continue is similar to **resume**. See "resume" on page 7-98.

Example:

```
(local) c 5
```

The processes specified by the default qualifier are resumed and will not stop again at the current breakpoint or watchpoint until it has been hit 5 times.

resume

Continue execution.

resume [*sigid*]

sigid

The processes receive the specified signal when they resume execution. *sigid* is a signal name or number. You may specify a signal name with or without the SIG prefix; the name is case-insensitive. If *sigid* is `0`, then the processes receive no signal when they resume execution. See "signal" on page 7-104.

If this argument is not present, then the processes are resumed with the signal that caused them to stop, similar to **continue**.

resume causes the processes specified by the qualifier to resume execution at the point where they last stopped. The processes run concurrently. Each process will execute until some event, such as hitting a breakpoint or watchpoint, causes it to stop.

If a **resume** command in a breakpoint (or watchpoint) command stream continues execution of the process stopped at that breakpoint or watchpoint, the command stream is terminated; no further commands are executed from that stream. If a **resume** command continues execution of a process that is currently executing another breakpoint (or watchpoint) command stream, the **resume** command does not take effect until that command stream has completed execution. See "Command Streams" on page 3-29.

If a **resume** command continues execution of a process that is currently executing an **on program** or **on restart** command stream, the **resume** command does not take effect until the affected process has been completely initialized by NightView and is ready to be debugged.

The difference between **resume** and **continue** is that **resume** does not wait for the

processes to stop. The debugger continues to read and process commands. See "continue" on page 7-97.

Example:

```
(local) resume 0
```

The processes specified by the default qualifier are resumed with no signal.

Example:

```
(local) resume 2
```

The processes specified by the default qualifier are resumed with signal number 2.

step

Execute one line, stepping into procedures.

step [*repeat*]

Abbreviation: **s**

repeat

The *repeat* argument specifies the number of lines to single step. The default is one line.

step causes the processes specified by the qualifier to continue execution until they have crossed a source line boundary. With a repeat count, this happens *repeat* times.

step follows execution into called procedures. That is, if the current line is a procedure call, and you **step**, then the process will execute until it is in that new procedure and then stop. If you want to step over the procedure, use **next**. See "next" on page 7-100.

If a **step** command causes execution to enter or leave a called procedure, then the output includes the equivalent of a **frame 0** command to show this. See "frame" on page 7-108.

This command completes only when all of the processes specified by the qualifier have completed the single step or stopped for some other reason (like receiving a signal). See "Signals" on page 3-12 for a discussion of the interactions between single-stepping and signals.

step is interpreted relative to the current frame. See "Current Frame" on page 3-25. That is, any lower frames are automatically finished before stepping.

There are also commands to single step individual instructions. See "stepi" on page 7-101 and "nexti" on page 7-102.

When the program has just started, **step** steps to the beginning of the procedure that calls static initializers or library-level elaboration procedures, if any. If there are none, **step** steps to the beginning of the main procedure.

Because of optimization and other considerations, a process may appear to stop multiple times in the same line or not at all in some lines. The decorations that appear when you list the source can help you decide which lines are executable (see "Source Line Decorations" on page 7-63). Also, disassembly can help you determine the flow of control through your program (see "x" on page 7-68).

If the **step** command causes execution to enter a procedure which is uninteresting, the **step** acts like **next**. See "Interesting Subprograms" on page 3-27. See "next" on page 7-100.

If an exception propagates to the current frame or a calling frame, then the **step** completes and execution is stopped at the beginning of the exception handler.

NOTE

If you step to a source line, and the instructions corresponding to that line begin with an inline call, NightView positions you at the beginning of the inline subprogram, rather than on the line with the call.

next

Execute one line, stepping over procedures.

next [*repeat*]

Abbreviation: **n**

repeat

The *repeat* argument specifies the number of lines to single step. The default is one line.

next causes the processes specified by the qualifier to continue execution until they have crossed a source line boundary. With a repeat count, this happens *repeat* times.

next steps over called procedures. That is, if the current line is a procedure call, and you single step with **next**, then the process will execute until that new procedure has returned. If you want to follow execution into the procedure, use **step**. See "step" on page 7-99.

If a **next** command causes execution to leave a called procedure, then the output includes the equivalent of a **frame 0** command to show this. See "frame" on page 7-108.

This command completes only when all of the processes specified by the qualifier have completed the single step or stopped for some other reason (like receiving a signal). See "Signals" on page 3-12 for a discussion of the interactions between single-stepping and signals.

next is interpreted relative to the current frame. See "Current Frame" on page 3-25. That is, any lower frames are automatically finished before stepping.

There are also commands to single step individual instructions. See “nexti” on page 7-102 and “stepi” on page 7-101.

When the program has just started, **next** steps to the beginning of the main procedure.

Because of optimization and other considerations, each process may appear to stop multiple times in the same line or not at all in some lines. The decorations that appear when you list the source can help you decide which lines are executable (see “Source Line Decorations” on page 7-63). Also, disassembly can help you determine the flow of control through your program (see “x” on page 7-68).

If an exception propagates to the current frame or a calling frame, then the **next** completes and execution is stopped at the beginning of the exception handler.

NOTE

If you step to a source line, and the instructions corresponding to that line begin with an inline call, NightView positions you at the beginning of the inline subprogram, rather than on the line with the call.

stepi

Execute one instruction, stepping into procedures.

stepi [*repeat*]

Abbreviation: **si**

repeat

The *repeat* argument specifies the number of instructions to single step. The default is one instruction.

stepi executes a single machine instruction in each of the processes specified by the qualifier.

This is very similar to **step**, except that **step** executes lines and **stepi** executes individual instructions. See “step” on page 7-99.

This command completes only when all of the processes specified by the qualifier have completed the single step or stopped for some other reason (like receiving a signal). See “Signals” on page 3-12 for a discussion of the interactions between single-stepping and signals.

stepi is interpreted relative to the current frame. See “Current Frame” on page 3-25. That is, any lower frames are automatically finished before stepping.

Sometimes, when stepping by instructions, it is useful to set up a **display** command to show the instruction that is just about to be executed each time the process stops. To do that, say

```
(local) display/i $pc
```

See "display" on page 7-72.

If the **stepi** command causes execution to enter a procedure which is uninteresting, the **stepi** acts like **nexti**. See "Interesting Subprograms" on page 3-27. See "nexti" on page 7-102.

If an exception propagates to the current frame or a calling frame, then the **stepi** completes and execution is stopped at the beginning of the exception handler.

nexti

Execute one instruction, stepping over procedures.

nexti [*repeat*]

Abbreviation: **ni**

repeat

The *repeat* argument specifies the number of instructions to single step. The default is one instruction.

nexti executes a single machine instruction in each of the processes specified by the qualifier.

This is very similar to **next**, except that **next** executes lines and **nexti** executes individual instructions. See "next" on page 7-100.

This command completes only when all of the processes specified by the qualifier have completed the single step or stopped for some other reason (like receiving a signal). See "Signals" on page 3-12 for a discussion of the interactions between single-stepping and signals.

nexti is interpreted relative to the current frame. See "Current Frame" on page 3-25. That is, any lower frames are automatically finished before stepping.

If an exception propagates to the current frame or a calling frame, then the **nexti** completes and execution is stopped at the beginning of the exception handler.

finish

Continue execution until the current function finishes.

finish

finish causes a process to continue execution until the current frame returns. This happens in each process specified by the qualifier.

Note that this may cause the process to finish multiple procedures, depending on which frame is the current frame. See "frame" on page 7-108. If the current frame is in the context of a task, thread, or LWP chosen by the **select-context** command, execution

continues until that task, thread, or LWP completes execution of that procedure, or until the process stops for some other reason.

In general, the exact action of this command is dependent on the language being debugged.

The **finish** command causes execution to leave a called procedure, so the output includes the equivalent of a **frame 0** command to show this.

This command completes only when all of the processes specified by the qualifier have completed the function execution or stopped for some other reason (like receiving a signal). The discussion in "Signals" on page 3-12 concerning interactions between single-stepping and signals also applies to the **finish** command.

If an exception propagates past the current frame, then the **finish** completes and execution is stopped at the beginning of the exception handler.

stop

Stop a process.

stop

The **stop** command stops each of the processes specified by the qualifier. In many cases (such as setting breakpoints), NightView requires a process to be stopped before a command may be applied to the process.

The **stop** command does not complete until all of the specified processes have been stopped. If a specified process is already stopped, this command silently ignores that process.

WARNING

It is possible, though unlikely, that the process will stop of its own accord (say by hitting a breakpoint) while NightView is trying to stop it. If that happens, your process may receive a spurious SIGTRAP signal the next time you resume its execution. This signal should be harmless; resuming your process after this signal occurs should get everything back to normal.

Example:

```
(local) (addams) stop
```

This example stops each of the processes in the process family named 'addams'.

jump

Continue execution at a specific location.

jump [at] *location-spec*

location-spec

The *location-spec* specifies where to continue execution. See "Location Specifiers" on page 7-9.

jump causes execution to continue at the specified location. This happens for each process specified in the qualifier.

jump does not modify the stack frames or registers, it just modifies the program counter and continues execution. Unless you are sure the registers have the right contents for the new location, you are cautioned to avoid using this command.

You must be in frame 0, with no hidden frames below frame zero, to use **jump**. See "Interesting Subprograms" on page 3-27.

signal

Continue execution with a signal.

signal *sigid*

sigid

Specifies the name or number of the signal with which to continue. If *sigid* is `0`, then the processes are continued without a signal. You may specify a signal name with or without the SIG prefix; the name is case-insensitive.

signal resumes execution of the processes specified in the qualifier, passing them a signal.

signal is useful if a process has received a signal (causing it to stop and be recognized by the debugger), but you don't want it to see the signal. Then, rather than using **continue** to continue the process, use **signal 0**.

Or, perhaps you want the process to receive a different signal. **signal** can resume your process with any signal.

If a **signal** command in a breakpoint (or watchpoint) command stream continues execution of the process stopped at that breakpoint or watchpoint, the command stream is terminated; no further commands are executed from that stream. If a **signal** command continues execution of a process that is currently executing another breakpoint (or watchpoint) command stream, the **signal** command does not take effect until that command stream has completed execution. See "Command Streams" on page 3-29.

If a **signal** command continues execution of a process that is currently executing an **on program** or **on restart** command stream, the **signal** command does not take effect until the affected process has been completely initialized by NightView and is ready to be debugged.

For a way to have the debugger deal with signals automatically, see "handle" on page 7-105. **signal** overrides the `pass` setting of **handle**.

Type **`info signal`** to get a list of all of the signals on your system. See "info signal" on page 7-125.

Example:

```
(local) signal 2
```

The processes resume with signal number 2.

handle

Specify how to handle signals and Ada exceptions in the user process.

`handle` `[/signal]` *sigid keyword* `...`

`handle` `/exception` *exception-name keyword* `...`

`handle` `/exception` *unit-name keyword* `...`

`handle` `/exception all` *keyword* `...`

`handle` `/unhandled_exception` *keyword* `...`

`/signal`

Specifies handling of a signal. This is the default.

sigid

Specifies the name or number of a signal to handle. Does not apply to **`handle /exception`** commands. You may specify a signal name with or without the SIG prefix; the name is case-insensitive.

`/exception`

Specifies handling of an Ada exception.

exception-name

Specifies the name of a particular Ada exception to be handled. This form of **`handle/exception`** takes precedence over any previous **`handle/exception`** command that specified `all`.

unit-name

Specifies that all Ada exceptions defined in the specified unit will be handled according to the keyword specifications. The effect is identical to the effect obtained by mentioning each of those exceptions in a **`handle/exception`** command.

`all`

Specifies that all Ada exceptions will be handled as specified by the keywords. This overrides any previous **`handle/exception`** com-

mand that specifies either an *exception-name* or a *unit-name*. Doesn't apply to signal handling specifications, nor to the handling of exceptions for which the user program does not have a handler (use **handle/unhandled_exception** for that).

`/unhandled_exception`

Specifies the handling (by NightView) of exceptions raised by the program when the program has no handler of its own for that exception.

keyword

keyword is one of `stop`, `nostop`, `print`, `noprint`, `pass` or `nopass`. Multiple keywords may be specified.

`handle` tells the debugger how to deal with signals sent to, or exceptions generated by, the user program.

Here are the meanings of the keywords:

`stop`

The process stops when it gets this signal or exception. `print` is implied with this keyword.

`nostop`

The process continues executing automatically after the signal or exception. You may still use `print` to tell you when the signal or exception has occurred.

`print`

NightView notifies you that the signal or exception has occurred. In the command-line interface, a message is printed to your terminal. In the graphical user interface, a message is printed in the Debug Message Area. See Chapter 9 [Graphical User Interface] on page 9-1. See "Debug Message Area" on page 9-28.

`noprint`

You do not receive notification when the signal or exception occurs. `nostop` is implied with this keyword.

`pass`

The signal will be passed to your process the next time it executes. This keyword is not applicable to Ada exceptions.

`nopass`

The signal is discarded, after stopping and printing if that's appropriate. This keyword is not applicable to Ada exceptions.

In most cases, a signal sent to a debugged program will cause that program to be stopped and NightView to be notified of the signal. NightView's normal action for most signals is to notify you of the signal and save it to be passed to the process the next time it is

continued. For example, the default setting for SIGQUIT would be described as:

```
(local) handle sigquit stop print pass
```

This default behavior can be altered by the **`handle`** command. Some settings allow the system to avoid stopping your process and notifying NightView of the signal. See "Signals" on page 3-12 for more information about this.

The default action for a few signals is different than the behavior described above. Consider SIGALRM, which is not usually an error; it is used in the normal functioning of the program. You usually don't want to know when your program gets a SIGALRM (but your program does) so the default setting for SIGALRM is:

```
(local) handle sigalrm nostop noprint pass
```

This says that if NightView discovers that your process has been sent a SIGALRM, it will automatically resume execution and pass the signal to the process without notifying you. (NightView may not even be aware of the signal with these settings of the **`handle`** command. See "Signals" on page 3-12.)

SIGINT is handled a little differently; when the process receives a SIGINT, the process stops and NightView notifies you, but the signal is discarded, so that the process never sees it. The normal setting for SIGINT is:

```
(local) handle sigint stop print nopass
```

For Ada programs the signal SIGADA is set as follows.

```
(local) handle sigada nostop noprint pass
```

For a way to deal with signals one at a time, see "signal" on page 7-104.

To find out the current settings for all the signals, see "info signal" on page 7-125.

If two conflicting keywords are specified, they are both applied, in the order they appear. For example, if the initial setting for signal number 1 is `stop`, `print`, `pass`, and you say:

```
(local) handle 1 noprint print
```

then the new setting is `nostop`, `print`, `pass`, because `noprint` implies `nostop`.

`handle` applies to all the processes specified in the qualifier.

The default settings for all Ada exceptions are `nostop`, `noprint`. If the settings are changed to `stop` and `print`, then execution is stopped in the Ada runtime routine that routes exceptions to the proper handler. This routine is usually uninteresting, so the current frame is set to the code that caused the exception. See "Interesting Subprograms" on page 3-27. The user is informed of the name of the exception and the Ada Reference Manual references.

To find out how one or more exceptions will be handled, you may use the **`info exception`** command. See "info exception" on page 7-129.

Selecting Context

frame

Select a new stack frame or print a description of the current stack frame.

`frame` [*frame-number*]

`frame` `*`*expression* [`at` *location-spec*]

Abbreviation: **`f`**

frame-number

Frame number selected as the new current stack frame. Frame number zero corresponds to the currently executing frame. Frame numbers for all the currently available stack frames may be obtained with the **`backtrace`** command (see "backtrace" on page 7-65).

`*`*expression*

Expression which yields an address at which the stack frame should start. This is the value that `$fp` would have, not the value of `$sp`.

location-spec

Specifies a location in the program to use to interpret the stack frame at the address given by `*`*expression*. See "Location Specifiers" on page 7-9. If you do not supply this argument, the default is the current value of `$cpc`.

NOTE

The `at` keyword may not be abbreviated in this command.

If no argument is given, a brief description of the current stack frame is printed. If multiple processes are specified in the command qualifier, each of them is described separately. For a more complete description of a frame, see "info frame" on page 7-122.

If a *frame-number* is given, the chosen stack frame is selected as the current frame (see "Current Frame" on page 3-25).

The `*`*expression* form of this command is provided for those occasions in which the stack is in an inconsistent state, or you wish to examine some memory whose contents look like stack frames. You should be very careful when using this form, observing the following cautions.

- A stack frame cannot be interpreted except in the context of some program-counter value. Therefore, you must be sure that the *location-spec* you give (or the value of `$cpc`) is consistent with the stack frame you are examining.

- The values of the machine registers are not altered by this form of the **frame** command. This means that variables that reside in registers cannot be reliably examined.
- The **up**, **down**, and **backtrace** commands are executed relative to the given frame address and program-counter value. However, the register contents for calling frames may still be incorrect, since only the registers saved in the stack can be restored by NightView.
- Modifying a register (or a variable stored in a register) may alter the current value of a machine register, or it may alter the value of that register stored on the stack. You must be very careful when doing this.
- Unless you have modified `$pc` or other machine registers, resuming execution of the process will resume with the state the process was in before the **frame** command was issued.

Once you have issued a **frame** command with a *****_expression_ argument, you can restore the previous view of the stack by issuing a **frame** command with a *frame-number* argument. This restores NightView's view of the stack to what it was before you issued the **frame** *****_expression_ command.

We recommend that, while you have the frame set using the *****_expression_ form, you should restrict yourself to just using the **up**, **down**, **backtrace**, and **print** commands, and that you print only global variables or variables stored on the stack.

up

Move one or more stack frames toward the caller of the current stack frame.

up [*number-of-frames*]

number-of-frames

Number of stack frames to advance toward the oldest calling frame. The number zero may be used to restore the current source position in the current frame (see "Current Frame" on page 3-25). If a negative number is specified, then frames are advanced toward the newest stack frame (see "down" on page 7-109).

If *number-of-frames* is not given, the number defaults to one, corresponding to the caller of the current frame.

This command is applied to each process in the qualifier.

down

Move one or more stack frames toward frames called by the current stack frame.

down [*number-of-frames*]

number-of-frames

Number of stack frames to advance toward the currently executing (newest) stack frame. The number zero may be used to restore the current source position in the current frame (see "Current Frame" on page 3-25). If a negative number is specified, then frames are advanced toward the oldest stack frame (see "up" on page 7-109).

If *number-of-frames* is not given, the number defaults to one, corresponding to the frame called by the current frame.

This command is applied to each process in the qualifier.

select-context

Select the context of an Ada task, a thread, or of a Lightweight Process (LWP).

`select-context` `default`

`select-context` `task=`*expression*

`select-context` `thread=`*expression*

`select-context` `lwp=`*lwpid*

`default`

This keyword selects the stack frame for the context where the process has stopped. If the process contains multiple Lightweight Processes, the operating system chooses one of them as the default context. See "Multithreaded Programs" on page 3-34.

`task=`*expression*

The `task=` keyword selects the context of an Ada task. The *expression* must either denote a task object or it must be an integer or pointer whose value is the address of a Task Control Block (TCB).

`thread=`*expression*

The `thread=` keyword selects the context of a thread created by `thr_create(3thread)`. The *expression* must be the `thread_t` value returned by `thr_create` for a currently active thread.

`lwp=`*lwpid*

The `lwp=` keyword selects the context of a specific Lightweight Process (LWP). The *lwpid* is the ID of the Lightweight Process whose context is selected.

The **`select-context`** command allows you to examine the context (see "Examining Your Program" on page 3-20) of an Ada task, a thread, or an LWP. Using **`select-context`**, you can get a backtrace (see "backtrace" on page 7-65) and examine registers and variables in the context of the selected task, thread, or LWP.

When a process that contains multiple LWPs, tasks, or threads stops, the current context becomes that of one specific task, thread, or LWP. (For a discussion of how this choice is made, see "Multithreaded Programs" on page 3-34.) You can use the **`select-context`** command to temporarily change the context to that of some other task, thread, or LWP.

Once a context has been selected, all **`frame`**, **`up`**, **`down`**, and **`backtrace`** commands apply to that context. All expressions and references to registers also refer to that context, with one exception. When an Ada task is not assigned to an LWP, the state of the task is saved in memory, but only certain registers are saved. If you reference other registers, their contents reflect the `default` context.

Note that execution control is on a process basis: if you resume execution, all LWPs are allowed to execute. If you enter a **`finish`**, **`step`**, **`next`**, **`stepi`**, or **`nexti`** command, the process executes until the selected task, thread or LWP completes the stepping operation, but other tasks, threads or LWPs may execute as well.

If you request evaluation of an expression containing a function call, the process is allowed to execute and all LWPs are allowed to run. If another LWP hits a breakpoint, or stops for some other reason, the function call is terminated prematurely and an error message is issued.

Miscellaneous Commands

help

Access the online help system.

`help` [*section*]

section

The name of a section in this manual (anything in the table of contents).

You can read any section in this document by giving the section name (or a unique prefix of the section name) as an argument to the **`help`** command.

If you type **`help`** without arguments, the help system displays the document section most relevant to the last error you received. Type **`help`** again to see help on the previous error you received, and so on.

Error message identifiers are section names, so you can get help for a specific error by giving the **`help`** command with the error message identifier. An error message identifier, beginning with `E-`, is printed with each error message. See "Errors" on page 3-29.

In the non-graphical user interfaces, **`help`** prints to the terminal. In the graphical user interface, **`help`** uses another program to display the documentation in a separate window. See "GUI Online Help" on page 9-2.

NOTE

In the non-graphical user interfaces, help is available only for error messages.

The **help** command ignores the command qualifier.

Examples:

```
(local) help Summary of Commands
```

The above example displays the section of the document that contains a brief description of each command.

```
(local) help backtrace
```

Display the description of the **backtrace** command.

```
(local) help E-command_proc003
```

Display help for the error with error message identifier `E-command_proc003`.

refresh

Refresh the terminal screen.

refresh

The **refresh** command clears the terminal screen and redraws it. This is helpful when the screen becomes garbled, such as with a modem and noisy phone lines.

refresh is only useful in the simple full-screen interface. This command does nothing in the command-line interface. See Chapter 8 [Simple Full-Screen Interface] on page 8-1.

shell

Run an arbitrary shell command.

shell [*shell-command*]

The **shell** command is used to execute a single line in a subshell. This command has nothing to do with debugging and the qualifier is ignored. It is simply provided because it is sometimes convenient to have a way to execute a shell command without having to suspend or exit the debugger.

If you just type **shell** without arguments, the debugger puts you in a shell where you can execute arbitrary commands until you exit the shell, at which time the debugger will get control again. You cannot use this form of the **shell** command inside a macro (See "Defining and Using Macros" on page 7-134).

The programs run by this command run on the local system only (the same one you are running NightView on) and inherit the current working directory of the debugger (see "cd" on page 7-56).

If you start background processes via **shell**, they will continue to run normally even if you quit out of the debugger.

The shell used is determined by looking for the SHELL environment variable, and if that is not found, by using your login shell.

In the simple full-screen interface, NightView does not have control over the terminal while you are executing a **shell** command, so after the command has completed you are asked to press return. This gives you a chance to view the command output before NightView redraws the screen. See Chapter 8 [Simple Full-Screen Interface] on page 8-1.

source

Input commands from a source file.

source *command-file*

command-file

The file to read.

This command reads the designated file and treats each line in the file as though it were a command you typed in. After reading all the commands in the file, the debugger returns to reading commands from the keyboard again. (If **source** commands are nested, ending one file returns to reading from the previous file.)

If NightView encounters any serious error, it stops reading from a **source** file. See "Command Streams" on page 3-29.

The qualifier on the **source** command has no effect. The default qualifier is applied to any commands in the source file which do not have explicit qualifiers.

delay

Delay NightView command execution for a specified time.

delay [*milliseconds*]

milliseconds

The number of milliseconds to delay command execution. If not specified, the default is 1.

This command delays the execution of NightView commands for at least the specified time period, expressed in milliseconds. The actual delay may be longer than the specified period. The command following a **delay** command in the same command stream will

not execute until at least the specified time has elapsed.

The primary use of the **delay** command is in command scripts, when you may want to prevent a command from executing immediately after the preceding one. For instance, you may wish to allow time for your program to execute for some length of time between the execution of two NightView commands.

The qualifier on the **delay** command has no effect.

Info Commands

The info commands all start with the word **info**, which may always be abbreviated to the single character **i**. The keyword following **info** identifies one of the many topics for which info is available. Each info command may also have additional arguments specific to the individual command.

The info commands can be broadly divided into two basic categories:

- Status queries, returning information about the current state of the debugger and the processes being debugged.
- Symbol table queries, returning information about program variables and type definitions.

Status Information

The status info commands allow you to query various information about the current state of the debugger (e.g., what breakpoints are set, how many dialogues are active, etc.).

info log

Describe any open log files.

`info log`

Describes any open log files currently in use by the debugger. The log files may be created by **`set-log`** (see "set-log" on page 7-44) or by **`set-show`** (see "set-show" on page 7-28).

info eventpoint

Describe current state of breakpoints, tracepoints, patchpoints, monitorpoints, agentpoints, and watchpoints.

`info eventpoint` [`/verbose`] [*name* | *number*] `...`

`/verbose`

Specify that the locations of all eventpoints displayed will be in verbose format. Verbose location format includes the program counter address (or addresses) of the eventpoint and, where possible, the corresponding function name, file name, and line number. The number of PC addresses printed is subject to the limit on printing addresses (see "set-limits" on page 7-46). The `verbose` keyword may be abbreviated.

name

An eventpoint name.

number

An eventpoint number.

This command describes eventpoints associated with the processes in the command qualifier. An eventpoint is any of a breakpoint, tracepoint, patchpoint, monitorpoint, agentpoint, or watchpoint. See "breakpoint" on page 7-79, "tracepoint" on page 7-83, "patchpoint" on page 7-80, "agentpoint" on page 7-87, "monitorpoint" on page 7-84, and "watchpoint" on page 7-95.

The information printed includes:

- The eventpoint ID.
- The eventpoint type.
- Current state of eventpoint (enabled, disabled, temporary).

- The eventpoint location. If `/verbose` was specified, then the location will be printed in verbose format. Otherwise it will be printed in the format in which it was specified when the eventpoint was created. For watchpoints, information is printed about the address being watched.
- The number of times program execution has crossed the eventpoint since the program started execution (even if the ignore count or condition was not satisfied, this count is incremented).
- The number of times the eventpoint has been *hit* since the program started execution (this count is incremented only if the condition and ignore count are satisfied).
- Any conditions attached to the eventpoint.
- The current ignore count.
- Any commands attached to the eventpoint (if it is a breakpoint, monitorpoint, or watchpoint).

This command sets the default **`x`** command dump address as well as the `$_` predefined convenience variable to the address of the last eventpoint listed. See "x" on page 7-68 and "Predefined Convenience Variables" on page 7-6.

info breakpoint

Describe current state of breakpoints.

`info breakpoint` [`/verbose`] [*name* | *number*] `...`

Abbreviation: **`i b`**

`/verbose`

Specify that the locations of all breakpoints displayed will be in verbose format. Verbose location format includes the program counter address (or addresses) of the breakpoint and, where possible, the corresponding function name, file name, and line number. The number of PC addresses printed is subject to the limit on printing addresses (see "set-limits" on page 7-46). The `verbose` keyword may be abbreviated.

name

An eventpoint name.

number

An eventpoint number.

This command normally describes all breakpoints associated with the processes indicated by the command qualifier. If you specify a list of eventpoint names or numbers, only those events are described. If any of the specified eventpoints are not breakpoints, they are ignored. Breakpoints are created with the **`breakpoint`** command. See "breakpoint" on page 7-79.

The information printed includes:

- The breakpoint ID.
- Current state of breakpoint (enabled, disabled, temporary).
- The breakpoint location. If `/verbose` was specified, then the location will be printed in verbose format. Otherwise it will be printed in the format in which it was specified when the breakpoint was created.
- The number of times program execution has crossed the breakpoint since the program started execution (even if the ignore count or condition was not satisfied, this count is incremented).
- The number of times the breakpoint has been *hit* since the program started execution (this count is incremented only if the condition and ignore count are satisfied).
- Any conditions attached to the breakpoint.
- The current ignore count.
- Any commands attached to the breakpoint.

This command sets the default **x** command dump address as well as the $_ predefined convenience variable to the address of the last breakpoint listed. See "x" on page 7-68 and "Predefined Convenience Variables" on page 7-6.

info tracepoint

Describe current state of tracepoints.

info tracepoint [/verbose] [*name* | *number*] ...

`/verbose`

Specify that the locations of all tracepoints displayed will be in verbose format. Verbose location format includes the program counter address (or addresses) of the tracepoint and, where possible, the corresponding function name, file name, and line number. The number of PC addresses printed is subject to the limit on printing addresses (see "set-limits" on page 7-46). The `verbose` keyword may be abbreviated.

name

An eventpoint name.

number

An eventpoint number.

This command describes tracepoints in the processes indicated by the qualifier. Normally all tracepoints are described, but if an argument is given, only those named are described. Any eventpoints specified in the argument list which are not tracepoints are ignored. Tracepoints are created with the **tracepoint** command. See "tracepoint" on page 7-83.

The information printed includes:

- The tracepoint ID.
- Current state of tracepoint (enabled, disabled, temporary).
- The tracepoint location. If `/verbose` was specified, then the location will be printed in verbose format. Otherwise it will be printed in the format in which it was specified when the tracepoint was created.
- The tracepoint event ID.
- The number of times program execution has crossed the tracepoint since the program started execution (even if the ignore count or condition was not satisfied, this count is incremented).
- The number of times the tracepoint has been *hit* since the program started execution (this count is incremented only if the condition and ignore count are satisfied).
- Any conditions attached to tracepoint.
- The current ignore count.
- The expression being recorded at the tracepoint.

This command sets the default **`x`** command dump address as well as the `$_` predefined convenience variable to the address of the last tracepoint listed. See "x" on page 7-68 and "Predefined Convenience Variables" on page 7-6.

info patchpoint

Describe current state of patchpoints.

`info patchpoint` [`/verbose`] [*name* | *number*] `...`

`/verbose`

Specify that the locations of all patchpoints displayed will be in verbose format. Verbose location format includes the program counter address (or addresses) of the patchpoint and, where possible, the corresponding function name, file name, and line number. The number of PC addresses printed is subject to the limit on printing addresses (see "set-limits" on page 7-46). The `verbose` keyword may be abbreviated.

name

An eventpoint name.

number

An eventpoint number.

This command describes patchpoints in the processes indicated by the qualifier. Normally all patchpoints are described, but if an argument is given, only those named are described. Any eventpoints specified in the argument list which are not patchpoints are ignored. Patchpoints are created using the **`patchpoint`** command. See "patchpoint" on page 7-80.

The information printed includes:

- The patchpoint ID.
- Current state of patchpoint (enabled, disabled, temporary).
- The patchpoint location. If `/verbose` was specified, then the location will be printed in verbose format. Otherwise it will be printed in the format in which it was specified when the patchpoint was created.
- The number of times program execution has crossed the patchpoint since the program started execution (even if the ignore count or condition was not satisfied, this count is incremented).
- The number of times the patchpoint has been *hit* since the program started execution (this count is incremented only if the condition and ignore count are satisfied).
- Any conditions attached to patchpoint.
- The current ignore count.
- The expression patched in at that point, or a description of where the program will branch.

This command sets the default **`x`** command dump address as well as the `$_` predefined convenience variable to the address of the last patchpoint listed. See "x" on page 7-68 and "Predefined Convenience Variables" on page 7-6.

info monitorpoint

Describe current state of monitorpoints.

`info monitorpoint` [`/verbose`] [*name* | *number*] ...

`/verbose`

Specify that the locations of all monitorpoints displayed will be in verbose format. Verbose location format includes the program counter address (or addresses) of the monitorpoint and, where possible, the corresponding function name, file name, and line number. The number of PC addresses printed is subject to the limit on printing addresses (see "set-limits" on page 7-46). The `verbose` keyword may be abbreviated.

name

An eventpoint name.

number

An eventpoint number.

This command describes monitorpoints in the processes indicated by the qualifier. Normally all monitorpoints are described, but if an argument is given, only those named are described. Any eventpoints specified in the argument list which are not monitorpoints are ignored. Monitorpoints are created with the **`monitorpoint`** command. See "monitorpoint" on page 7-84.

The information printed includes:

- The monitorpoint ID.
- Current state of monitorpoint (enabled, disabled, temporary).
- The monitorpoint location. If `/verbose` was specified, then the location will be printed in verbose format. Otherwise it will be printed in the format in which it was specified when the monitorpoint was created.
- The number of times program execution has crossed the monitorpoint since the program started execution (even if the ignore count or condition was not satisfied, this count is incremented).
- The number of times the monitorpoint has been *hit* since the program started execution (this count is incremented only if the condition and ignore count are satisfied).
- Any conditions attached to monitorpoint.
- The current ignore count.
- The commands attached to the monitorpoint.

This command sets the default **`x`** command dump address as well as the `$_` predefined convenience variable to the address of the last monitorpoint listed. See "x" on page 7-68 and "Predefined Convenience Variables" on page 7-6.

info agentpoint

Describe current state of agentpoints.

`info agentpoint` [`/verbose`] [*name* | *number*] `...`

`/verbose`

Specify that the locations of all agentpoints displayed will be in verbose format. Verbose location format includes the program counter address (or addresses) of the agentpoint and, where possible, the corresponding function name, file name, and line number. The number of PC addresses printed is subject to the limit on printing addresses (see "set-limits" on page 7-46). The `verbose` keyword may be abbreviated.

name

An eventpoint name.

number

An eventpoint number.

This command describes agentpoints in the processes indicated by the qualifier. Normally all agentpoints are described, but if an argument is given, only those named are described. Any eventpoints specified in the argument list which are not agentpoints are ignored. Agentpoints are created with the **`agentpoint`** command. See "agentpoint" on page 7-87.

The information printed includes:

- The agentpoint ID.
- Current state of agentpoint (enabled, disabled, temporary).
- The agentpoint location. If `/verbose` was specified, then the location will be printed in verbose format. Otherwise it will be printed in the format in which it was specified when the agentpoint was created.
- The number of times program execution has crossed the agentpoint since the program started execution (even if the ignore count or condition was not satisfied, this count is incremented).
- The number of times the agentpoint has been *hit* since the program started execution (this count is incremented only if the condition and ignore count are satisfied).
- Any conditions attached to agentpoint.
- The current ignore count.

This command sets the default **`x`** command dump address as well as the `$_` predefined convenience variable to the address of the last agentpoint listed. See "x" on page 7-68 and "Predefined Convenience Variables" on page 7-6.

info watchpoint

Describe current state of watchpoints.

`info watchpoint` [`/verbose`] [*name* | *number*] `...`

`/verbose`

The `verbose` keyword is accepted for compatibility with other watchpoints, but is ignored. The `verbose` keyword may be abbreviated.

name

An eventpoint name.

number

An eventpoint number.

This command describes watchpoints in the processes indicated by the qualifier. Normally all watchpoints are described, but if an argument is given, only those named are described. Any eventpoints specified in the argument list which are not watchpoints are ignored. Watchpoints are created with the **`watchpoint`** command. See "watchpoint" on page 7-95.

The information printed includes:

- The watchpoint ID.
- Current state of the watchpoint (enabled, disabled, temporary).

- The address being watched.
- The number of times the process accessed the address being watched since the program started execution. This count is incremented even if the ignore count or condition was not satisfied. This number is displayed as `#crossings` (for consistency with other eventpoint types).
- The number of times the watchpoint has been *hit* since the program started execution (this count is incremented only if the condition and ignore count are satisfied).
- Any conditions attached to the watchpoint.
- The current ignore count.
- Any commands attached to the watchpoint.

info frame

Describe a stack frame.

`info frame` `[/v] [*`*expression* `[at` *location-spec*`]]`

`/v`

If this option is supplied, NightView prints detailed, machine-specific, information about the requested stack frame. You are seldom likely to be interested in this information; it is provided primarily for detecting problems with the generated debugging information.

`*`*expression*

The address of a stack frame. This is the value that `$fp` would have, not `$sp`.

location-spec

Specifies a location in the program to use to interpret the stack frame at the address given by `*`*expression*. See "Location Specifiers" on page 7-9. If you do not supply this argument, the default is the current value of `$cpc`.

NOTE

The `at` keyword may not be abbreviated in this command.

This command describes all available information about the current stack frame for a process (see "Current Frame" on page 3-25). See also "frame" on page 7-108.

If multiple processes are specified in the command qualifier, each of them is described separately. An error message is printed if any of the processes are running.

If the optional `*`*expression* is given, then the frame at that address is described (but the current frame is not changed). If you supply the *location-spec* argument, the frame is interpreted as a frame for the routine at the resulting address. If you omit this argument, the current value of `$cpc` is used in decoding the frame.

If ******expression* does not evaluate to a valid frame address, or the frame at that address does not correspond to the given program location, the information printed will probably be nonsense.

The information printed about a frame includes:

- The address of the frame.
- The addresses of the adjacent frames (if any).
- The current frame size.
- The saved return address and its location on the stack (or in a register).
- Any saved registers and their locations on the stack.
- Which registers are currently in use as stack and/or frame pointers and their relation to the current frame.
- The name of the subroutine associated with the frame along with the source line and file name (if known).

info directories

Print the search path used to locate source files.

`info directories`

Print the search path used to locate source files. If multiple processes are given in the qualifier, print the list of directories for each process. See “directory” on page 7-60, for the command used to set the search path.

info convenience

Describe convenience variables.

`info convenience`

This command describes all the convenience variables that have been defined. Convenience variables may be global or process local (see “set-local” on page 7-50). This command first describes the global variables, then (for each process specified by the command qualifier) describes the process local variables. The name, data type, and value of each variable is listed.

The convenience variables that correspond to the process registers are not described by this command (see “info registers” on page 7-124).

info display

Describe expressions that are automatically displayed.

`info display`

This command describes the set of expressions that are automatically displayed each time

a program stops (see "display" on page 7-72).

info history

Print value history information.

info history [*number*]

number

Specifies an item in the value history list (each value has a unique sequence number). The default value is the most recent history list entry.

This command prints ten history-list values centered around the specified entry. It also prints information about how many history items currently exist. See "set-history" on page 7-46.

info limits

Print information about limits on expression and location output.

info limits

The command prints the limits on array elements and character-string elements printed by expression output commands, and the limits on program locations printed by other **info** commands. See "set-limits" on page 7-46.

The qualifier is ignored by this command.

info registers

Print information about registers.

info registers [*regexp*]

regexp

A regular expression matching register names. An anchored match *is* implied. See "Regular Expressions" on page 7-12.

If the *regexp* argument is not given, this command prints all the normally accessible registers that are of general interest to most programmers (such as accumulators, program counter, stack pointer, etc.). If you give a regular expression argument, any register with a name matching that regular expression is printed. To print *all* the registers, you must specify the regular expression `.*` as an argument (this includes all the obscure control registers and any other registers not normally of interest to a programmer). See "Predefined Convenience Variables" on page 7-6.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

Registers are printed relative to the current frame (see "Current Frame" on page 3-25).

This means that any register saving is logically unwound as you change frames (the register contents are not actually modified). You see the value the register would have if you returned to the current frame. (If the current frame is also the most recent frame at the end of the stack, the current machine register contents are the correct contents relative to frame zero.)

If the current frame is not frame zero, but you want to see the current active contents of the machine registers, you have to move to frame zero before running the **`info registers`** command (see "frame" on page 7-108).

If the command qualifier names multiple processes, the registers from each process are printed separately. If any of the processes are running, an error is printed.

Since this command operates only on register names, the dollar sign (`$`) normally used to refer to registers is optional for this command.

Some registers are defined by the architecture to be composed of various fields. **`info registers`** expands those fields symbolically. If a field is a single bit, NightView prints an abbreviation for that field only if the value of the field is `1`. See the architecture manual for descriptions of the fields and a list of the abbreviations for each register.

info signal

Print information about signals.

`info signal` [*signal* `...`]

signal

A signal number or signal name.

This command describes how signals will be handled by the process receiving them. If the command qualifier specifies multiple processes, then the signal information is listed separately for each process. The information printed includes:

- The signal name.
- The signal number.
- The way the debugger will handle this signal. (see "handle" on page 7-105).

If no *signals* are specified, then information for all signals is printed.

info process

Describe processes being debugged.

`info process`

This command lists information about all the processes specified in the command qualifier (qualify with `(all)` to list all of them). The information includes:

- The process ID (PID).

- The controlling dialogue for the process.
- The arguments passed to the program on startup (`argv` array).
- The current process state (running, stopped).
- When the process state is stopped, list where and why it stopped.
- The current language setting. See "set-language" on page 7-44.
- The disposition of child processes; that is, under what circumstances a child process will be debugged. See "set-children" on page 7-41.

info memory

Print information about the virtual address space.

`info memory` `[/verbose]`

`/verbose`

Indicates that extra information should be printed.

This command prints information about the virtual address space for each process specified in the command qualifier. For each region of memory, this command prints the following information:

- The beginning address and ending address of the region.
- The size, in bytes, of the region.
- If the region is the first region associated with a shared library, the name of the library is printed.
- Whether the region is readable, writable, executable, shared, or locked in physical memory.
- Whether the region is being used as the process' stack or memory heap.
- If the region was attached by NightView, what the region is for and how much space is left in the region. See Appendix E [Implementation Overview] on page E-1. If the `/verbose` option is specified, NightView prints information about the individual blocks allocated in the region.

The list also includes any regions reserved by the user with the **`mreserve`** command. See "mreserve" on page 7-43.

info dialogue

Print information about active dialogues.

`info dialogue`

This command lists information about all the dialogues specified in the command qualifier (qualify with `(all)` to list all of them). The information includes:

- The machine running the dialogue.
- The sizes that will be used for patch areas created in the future. See "set-patch-area-size" on page 7-50.
- The list of **debug** and **nodebug** patterns for this dialogue. See "debug" on page 7-20.
- The processes being debugged under control of the dialogue.
- The user running the dialogue.
- The status of any dialogue output (see "set-show" on page 7-28).
- The list of object filename translations for this dialogue. See "translate-object-file" on page 7-21.

info family

Print information about an existing process family.

`info family` [*regexp*]

regexp

A regular expression matching family names. An anchored match *is* implied. See "Regular Expressions" on page 7-12.

For each family name that matches *regexp* this command lists each process that is a member of that family (see "family" on page 7-40). If *regexp* is omitted, then the contents of all process families are printed.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

info name

Print information about an existing eventpoint-name.

`info name` [*regexp*]

regexp

A regular expression matching eventpoint-names. An anchored match *is* implied. See "Regular Expressions" on page 7-12.

For each eventpoint-name that matches *regexp*, this command lists each eventpoint that is a member of that eventpoint-name (see "name" on page 7-78). If *regexp* is omitted, then the contents of all eventpoint-names are printed.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

Each eventpoint is identified by a dialogue-name, a process-id (PID), and an eventpoint-id that is unique for that process.

info on dialogue

Print **on dialogue** commands.

info on dialogue [*name*]

name

The name of a prospective dialogue.

If no arguments are given, then all existing **on dialogue** commands are printed. If a dialogue name is given, then only the **on dialogue** commands that would be executed if a dialogue named *name* were to be created are printed. See "on dialogue" on page 7-24

info on program

Print **on program** commands.

info on program [*program*]

program

The path name of a prospective executable file.

If no arguments are given, then **info on program** prints all existing **on program** commands for each dialogue specified by the qualifier. If a program path is given, then **info on program** prints the **on program** commands that would be executed if *program* were run in each dialogue specified by the qualifier. See "on program" on page 7-36.

info on restart

Print **on restart** commands.

info on restart [output=*outname* | append=*outname*] [*program*]

output=*outname*

Write the information to *outname*.

append=*outname*

Append the information to *outname*.

program

The path name of a prospective executable file.

If no *program* is given, then **info on restart** prints all existing **on restart** commands for each dialogue specified by the qualifier. If a *program* path is given, then **info on restart** prints the **on restart** commands that would be executed if *program* were run in each dialogue specified by the qualifier. See "on restart" on page 7-38.

If no *outname* is specified, then the output is to the terminal or to the GUI message area.

`info on restart` may be used to preserve restart information in a file for use in a later debug session. See "source" on page 7-113. See "Restarting a Program" on page 3-14. For an example, see "checkpoint" on page 7-39.

info exception

Print information about Ada exception handling.

`info exception` *exception-name...*

`info exception` *unit-name*

`info exception`

Abbreviation: **`exception`**

exception-name

Specifies the name of a particular Ada exception.

unit-name

Specifies all Ada exceptions defined in the specified unit.

This command describes the current exception handling settings for the processes specified by the qualifier. See "handle" on page 7-105. With no arguments, the current default handling of exceptions is displayed along with the handling of any specific exceptions to which the default is not applicable. If an argument is given, the handling of those specific exceptions is displayed. The **`info exception`** command will list:

- The exception name, or the keyword `all` denoting the default.
- The exception handling settings.

Symbol Table Information

The info commands in this section are used to lookup and report on information recorded in the debug tables of program files. This includes the names and declarations of variables, the address of generated code for source lines, etc.

info args

Print description of current routine arguments.

`info args`

This command prints a description of each argument of the subroutine associated with the current frame (see "Current Frame" on page 3-25).

info locals

Print information about local variables.

`info locals` [*regexp*]

regexp

A regular expression matched against local variable names. An anchored match *is* implied. See "Regular Expressions" on page 7-12.

Print a description of every local variable visible in the current context. If the *regexp* argument is given, print only the variables with names matching the regular expression.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

The term *local variables* is defined to include all variables with any sort of restricted scope. External variables visible throughout the program are never listed by this command.

The information listed for each variable includes:

- The name of the variable.
- The type of the variable.
- The current value of the variable.
- The location of the variable.
- The scope of the variable (directly visible, inherited from an outer block, etc.).

info variables

Print global variable information.

info variables [*regexp*]

regexp

A regular expression matched against global variable names. An anchored match *is* implied. See "Regular Expressions" on page 7-12.

This command prints information about global variables. When the *regexp* argument is given, it prints only variable names matching the regular expression.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

info address

Determine the location of a variable.

info address *identifier*

identifier

The name of the variable to be described.

Print out information about where the given variable (visible in the current context) is located. If the variable is in a register, it prints the register name. If it is on the stack, it prints the stack frame offset. If it is in static memory, it prints the absolute location.

To determine the absolute address of a particular instance of a stack variable you must use the **print** command to evaluate an expression which returns the address (for the C language, this would be something like **print &name**, see "print" on page 7-66).

info sources

List names of source files.

info sources [*pattern*]

pattern

Wildcard pattern to match against source file names. See "Wildcard Patterns" on page 7-14.

This command lists the names of the source files recorded in the debug tables. If a wildcard pattern is given, it lists only file names matching the wildcard pattern.

If multiple processes are specified in the command qualifier, the source files for each process are listed separately.

info functions

List names of functions, subroutines, or Ada unit names.

info functions [*regexp*]

regexp

A regular expression to match against function names. An anchored match *is* implied. See "Regular Expressions" on page 7-12.

This command lists the names of functions, subroutines, or Ada unit names recorded in the debug tables. If a regular expression is given, it lists only names matching the regular expression.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

info types

Print type definition information.

`info types` [*regexp*]

regexp

A regular expression to match against type names. An anchored match *is* implied. See "Regular Expressions" on page 7-12.

This command prints information about type definitions. When the *regexp* argument is given, it prints only type names matching the regular expression; otherwise, it prints all the types defined in the program.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

info whatis

Describe the result type of an expression visible in the current context.

`info whatis` *expression*

Abbreviation: **`whatis`**

expression

An expression for which the data type is to be determined. See "Expression Evaluation" on page 3-20.

Describe the result type of the expression. The expression is not normally evaluated, but operations which require run time type determination may require portions of the expression to be evaluated. If the expression includes the Ada `'self` attribute or the C++ `dynamic_cast<>` function, their operands must be evaluated in order to determine the actual type of the result.

info representation

Describe the storage representation of an expression.

`info representation` *expression*

Abbreviation: **`representation`**

expression

An expression for which the data type is to be determined. See "Expression Evaluation" on page 3-20.

Describe the storage representation of the result type of the expression. The expression is not evaluated.

info declaration

Print the declaration of variables or types.

`info declaration` *regexp*

Abbreviation: **`ptype`**

regexp

A regular expression to match against type names and variable names. An anchored match *is* implied. See "Regular Expressions" on page 7-12.

The *regexp* parameter may specify type or variable names visible in the current context. This command prints the complete declaration of all matching names.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

info files

Print the names of the executable, symbol table and core files.

`info files`

For each process specified in the command qualifier, print the names of the executable file, symbol table file, and core file associated with the process (the executable and symbol table files are usually the same).

info line

Describe location of a source line.

`info line` `[at]` *location-spec*

location-spec

Query the source line number associated with this location.

Describe the location of the source line implied by the *location-spec* argument (see "Location Specifiers" on page 7-9). The information printed includes:

- The address of the *location-spec*.
- The ranges of addresses occupied by the generated code for the line. The number of address ranges printed is subject to the current limit on addresses (see "set-limits" on page 7-46).
- The source file and line number.
- The function containing the line.

This command sets the default **x** command dump address as well as the $_ predefined convenience variable to the address of the first instruction in the line. See "x" on page 7-68 and "Predefined Convenience Variables" on page 7-6.

Defining and Using Macros

NightView provides a macro facility so you can augment the NightView commands with your own features. Macros can either be used as part of another command, or as a new command.

A *macro* is a named set of text, possibly with arguments, that can be substituted later in any NightView command. The arguments allow macros to expand to different text in different circumstances. Macros are useful in extending the command set available in NightView; they can also serve as shortcuts for frequently used constructs in commands or expressions.

define

Define a NightView macro.

define *macro-name*[(*arg-name* [, *arg-name*] ...)] [*text*]

define *macro-name* [(*arg-name* [, *arg-name*] ...)] as

macro-name

This is the name of the macro. Macro names follow the usual rules for identifiers in most languages: they must begin with an alphabetic character, followed by zero or more alphanumeric characters or underscore. There is no limit to the length of a macro name.

A macro name can be the same as a NightView command name, but this may render the command unusable. See "Referencing Macros" on page 7-137 for more information.

arg-name

A *formal argument* name. These names follow the same rules as *macro-name*.

text

The text to be substituted when the macro is invoked. In this form, the substituted text will not contain any newline characters, so the *text* becomes part of whatever command the macro invocation appears in.

NOTE

There must not be any blanks separating the *macro-name* from the left parenthesis that introduces the formal arguments.

In the second form of the **define** command, the text of the macro begins on the line following the **define** command and extends until a line containing only the words **end define** is encountered. Except for the newline character immediately following the as keyword and the newline immediately preceding the **end define** command, the newline characters within the body of the macro will be retained in the substituted text. Thus, each line of text in the macro body must normally be a complete NightView command.

Comments appearing in the body of the macro become part of the body. Thus, they appear in the text that is substituted for a reference to the macro. You should avoid having a comment as the last line of a macro, because it may cause any text following the macro invocation to be ignored.

In the command-line and simple full-screen interfaces, the prompt changes to > while you are entering the second form of the **define** command. (See “Command Syntax” on page 7-1.)

The **define** command associates a body of text with the given *macro-name*. When the macro is invoked (see “Referencing Macros” on page 7-137), the macro name and its actual arguments are replaced by the associated text. The text of the macro, called the *macro body*, may contain references to other macros (in particular, they will want to reference their formal arguments). A macro may not reference itself, either directly or indirectly; that is, macros cannot be recursive.

Within the body of a macro, each *arg-name* becomes a macro without arguments that expands to its corresponding actual argument. “Referencing Macros” on page 7-137 describes the syntax of macro invocations and actual arguments.

A macro body should not contain another **define** command.

The **define** command ignores any qualifier supplied for it.

If the given *macro-name* was previously defined as a macro, the new definition replaces the old one. If you omit the *text* in a one-line definition, or the **end define** command appears on the line immediately following the **define**...as command, any prior definition of *macro-name* is removed.

Examples:

```
(local) define printhex(str,x) printf "The value of %s is 0x%x\n",
@str, @x
```

The above example defines a macro that prints a descriptive string and the value of an

arbitrary variable, using the **printf** command.

```
(local) define advance(p) as
>          set @p = @p->next
>          print *@p
>       end define
```

The preceding example defines a macro that advances a pointer to the next item in a linked list, then prints the item. Note that this macro requires the language context to be C or C++, but the type of the argument pointer can be a pointer to any structure that contains an appropriately-typed field named "next".

```
(local) define short (VERY_LONG_NAME(INDEX*2,INDEX-1)*SOME_CONSTANT)
```

This example simply defines a shorthand for a long Fortran expression. Note that it does not have any arguments; the parentheses surround the substituted text to make sure that precedence of operators is preserved when the macro is invoked.

Referencing Macros

Macros are usually referenced by preceding the macro name with the @ character, and following the macro name with a parenthesized list of arguments, if the macro was defined with arguments. If you wish, you may enclose the macro name inside of '{' and '}' (but any argument list must appear *outside* of the braces). The number of arguments you supply must be the same as the number of formal arguments (i.e., the *arg-names*) specified in the **`define`** command; otherwise, NightView issues an error. Arguments are matched with each formal argument name by position.

A reference to a macro without any arguments consists solely of the @ character followed (without intervening blanks) by the macro name. A reference to a macro with one or more arguments consists of the @ character, the macro name, and a list of actual arguments. The actual arguments begin with a left parenthesis and end with a matching right parenthesis. If more than one argument is given, a comma must separate them. If an actual argument contains a left parenthesis, then the argument extends until a matching right parenthesis is encountered, irrespective of any other characters, including commas, in the intervening text. Note that an unmatched right parenthesis appearing in an actual argument prematurely ends the list of actual arguments; this may cause an error, or it may produce unexpected results.

An actual argument may contain an invocation of another macro; that invocation is expanded immediately when the actual argument is read during the processing of the enclosing macro invocation. This can lead to some surprising results, because NightView expands these actual arguments without regard to the context in which they will ultimately appear.

For example:

```
(local) define abc xyz
(local) define printit(x) print "The value is %s\n", @x
(local) print "The value is %s\n", "@abc"
(local) @printit("@abc")
```

The **`print`** command will print "The value is @abc", because macros are not normally expanded within string literals. However, the `@printit` command will print "The value is xyz", because NightView expands the macro `@abc` when it is processing the invocation of macro `@printit`. At that time, it does not know that the double quotes imply a string literal.

String literals as actual arguments can cause other problems as well. For example:

```
(local) # Illegal reference:
(local) @mymac("This has a left-parenthesis(")
(local) # Okay:
(local) @mymac("This has two parentheses()")
```

The first invocation of **`mymac`** is invalid because the actual argument contains an unmatched left parenthesis. Since NightView attempts to balance parentheses without regard to any other text (including quotes), the right parenthesis matches the left parenthesis in the argument, leaving the argument list without a closing right parenthesis.

If a macro invocation appears where a command keyword is expected, then you can leave off the @ prefix character (but the macro name may *not* be enclosed between '{' and '}').

This allows macros to be used conveniently as command shortcuts. However, if the macro requires arguments, these must still be placed within parentheses after the macro name.

Macros take precedence over commands when the macro name appears in place of a command keyword. This means that if you name a macro the same as a built-in NightView command, you may not be able to reference the built-in command anymore. However, you cannot abbreviate the macro name in an invocation, so you may be able to use an abbreviation for the built-in command. If you name a macro the same as a built-in command abbreviation, you won't be able to use that particular abbreviation for the built-in command later, but you can still use the full form, or a different abbreviation. If you accidentally name a macro the same as a built-in command, you can remove the definition by entering

```
(local) # Note, no text given in definition.
(local) define macro-name
```

You may want to refer to the Summary of Commands (see Appendix B [Summary of Commands] on page B-1) for a complete list of the NightView commands, so you can avoid these kinds of conflicts.

Macro references can generally appear anywhere within a NightView command, but you should be aware of the following rules:

- NightView never expands macros that appear within command comments.
- NightView usually does not expand macros that appear within string literals. However, if the literal appears as an actual argument in another macro invocation, macros within the string literal may be expanded.
- Macros are not expanded in the *format-string* argument to the **printf** command. See "printf" on page 7-74.
- Macros appearing in an **echo** command are expanded. See "echo" on page 7-71.
- Macros appearing in a **!** (see "!" on page 7-27), **run** (see "run" on page 7-30), or **shell** (see "shell" on page 7-112) command are not expanded.
- A macro referenced within a language expression must expand to text that makes sense as part of that expression.
- A macro can be used to form part of a syntactic item, or token, in a NightView command. For example, you could form a variable name in an expression from the results of two macro invocations. However, you cannot use this technique to construct the name of a macro to be invoked.

Examples:

```
(local) define short (VERY_LONG_NAME(INDEX*2,INDEX-1)*SOME_CONSTANT)
(local) set $x=i + @{short*10
```

The above example uses a macro in an expression.

```
(local) define printhex(str,x) printf "The value of %s is 0x%x\n",
@str, @x
(local) printhex("ptr1", ptr1)
(local) printhex("ptr1->next", (ptr1=ptr1->next, ptr1))
```

This example invokes the macro 'printhex' twice. The second invocation demonstrates how an expression containing a comma can be included as a formal argument.

The following C fragment defines some data types for use in the next example:

```
struct list_element {
   struct list_element  * next ;
   struct data          * the_data ;
};
extern struct list_element   * hd ;
```

Example NightView commands:

```
(local) define printdata(p) as
>          printf "The data is:\n"
>          print  *(@p)->the_data
>       end define
(local) define next(p) as
>          set @p = (@p)->next
>       end define
```

info macros

Print a description of one or more NightView macros.

`info macros` [*regexp*]

regexp

A regular expression matching macro names. An anchored match *is* implied. See "Regular Expressions" on page 7-12.

If the *regexp* argument is not given, the **`info macros`** command prints a description of every macro you have defined. If you give a *regexp* argument, a description of every macro whose name matches the regular expression is printed.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

The description of each macro includes:

- The name of the macro.
- The formal argument names, if any, of the macro.
- The macro body text, exactly as it will appear when substituted, except that the last line of the macro will be followed by a newline.

8
Simple Full-Screen Interface

NightView is designed to be able to debug multiple processes asynchronously. That means your processes may be running and producing output or hitting breakpoints, all at the same time. You might be entering NightView commands at the same time as well.

This can be a little confusing. It would be especially confusing if NightView were to write to your terminal at the same time you are trying to enter a command For this reason, NightView doesn't usually show you output or event notifications while it is reading your commands (It will do that if you want it to, though. See “set-show” on page 7-28.)

This means that NightView may have output or event notifications to show you, but it will not show them to you because it is waiting for you to type a command. You can press carriage return a few times to see output you are expecting, but that can be annoying

A full-screen interface gives NightView a way to show you output and event notifications as soon as they are available without interfering with your typing

The simple full-screen interface has the same basic functionality as the command-line interface. All the commands are the same. In fact, the simple full-screen interface looks a lot like the command-line interface. The main difference is that NightView has control over the entire screen, so it can print output to you while you are "at a prompt".

Using the Simple Full-Screen Interface

To use the simple full-screen interface, you should have your TERM environment variable set to the type of your terminal. If you are using a full-screen editor, such as **`vi(1)`**, you probably have already taken care of this.

Invoke NightView with the **`-simplescreen`** option:

```
nview -simplescreen
```

NightView clears the screen before it writes its welcome message. Then the prompt is written to the bottom line and you can type a command.

NightView does not have control over the terminal while you are executing a **`shell`** command, so after the command has completed you are asked to press return This gives you a chance to view the command output before NightView redraws the screen. See “shell” on page 7-112.

The simple full-screen interface creates a special window when you use monitorpoints. See “Monitor Window - Simple Full-Screen” on page 8-2 for more information about this window.

Editing Commands in the Simple Full-Screen Interface

You can use special key sequences to edit your commands. The key sequences are based on the line editing modes of **`ksh(1)`**. NightView implements the `emacs`, `gmacs` and `vi` modes of **`ksh`**. In particular, you can use the various key sequences to retrieve previously entered commands.

The initial editor mode is set from your `VISUAL` or `EDITOR` environment variables. If NightView cannot determine the mode from those variables, then the default mode is `emacs`. You can explicitly set the editor mode with the **`set-editor`** command. See "set-editor" on page 7-55.

Monitor Window - Simple Full-Screen

The Monitor Window is created when you use monitorpoints while running NightView with the simple full-screen interface. See "Monitor Window" on page 3-27.

In the simple full-screen interface, the Monitor Window appears at the top of the screen and takes up as many lines as it needs for the number of items displayed, plus one status line, while leaving at least ten lines for other debugger operations at the bottom of the screen.

Only the items that fit in the space available at the top of the screen are displayed. Any further items are left in the same state they would be in following an **`mcontrol nodisplay`** command (See "mcontrol" on page 7-86)

The stale data indicators used in the simple full-screen Monitor Window are simple characters used to indicate each state. A space ( ) is used to indicate updated values. A period (`.`) is used for monitorpoints that have not been executed. An exclamation point (`!`) is used for monitorpoints which have executed but not taken a sample. For more information about stale data indicators, see "Monitor Window" on page 3-27.

A status line at the bottom of the simple full-screen Monitor Window divides it from the remainder of the screen. The status line indicates the state of the Monitor Window (`held` or `running`) and shows the current delay time in milliseconds between updates of the window.

9
Graphical User Interface

This chapter describes the graphical user interface (GUI) for NightView. The GUI provides more flexibility and functionality than either the command-line interface or the simple full-screen interface.

The graphical user interface for NightView is based on OSF/Motif™. NightView runs in the environment of the X Window System™ Version 11, Release 6 (or later).

This chapter assumes that you have a basic understanding of window system concepts such as selecting objects by clicking with the mouse. For more information, see the *OSF/Motif User's Guide*.

It is assumed that your X server has a three-button mouse. By default, mouse button 1 is the leftmost button, button 2 the middle button, and button 3 the rightmost button. You can reassign the functions associated with mouse buttons by using **`xmodmap(1)`**. If you do not have a three-button mouse, see your system administrator or read sections on input and navigation in the *OSF/Motif Style Guide*. Use mouse button 1 when you are told to click, drag, press, and select.

This chapter refers to using a mouse, and refers to *clicking* on objects to select them or to activate them, but you may also use keyboard selection and activation. See “Keys” on page 9-10.

You can customize the NightView GUI. See Appendix D [GUI Customization] on page D-1.

Sample debug sessions showing how to use the NightView graphical user interface are available. See Chapter 2 [A Quick Start - GUI] on page 2-1. See Chapter 5 [Tutorial - GUI] on page 5-1.

NightView GUI Concepts

This section explains concepts that you need to understand so that you can use the NightView graphical user interface to its fullest advantage.

GUI Overview

The Graphical User Interface contains these major types of windows.

- Dialogue Window
- Debug Window

- Monitor Window - GUI
- Global Window
- Help Window

Each of these major windows has supporting dialog boxes which are described with the corresponding major window. See "Dialogues and Dialog Boxes" on page 9-10.

A Dialogue Window is used to control a NightView dialogue and for input and output with the dialogue shell. See "Dialogue Window" on page 9-16. A Debug Window is used to debug and manipulate one or more processes. See "Debug Window" on page 9-20. The GUI Monitor Window displays monitorpoints. See "Monitorpoints" on page 3-10. The Global Window is used to control the debugger in general. See "Global Window" on page 9-48. The Help Window is used only when you ask for help. See "Help Window" on page 9-50.

Typically, while debugging a process, you have the Debug and Dialogue Windows available, but most of the actual debugging is done with the Debug Window. You may iconify any windows you don't need at the moment.

Each of the NightView windows has a unique icon image that relates to the window's function. The Debug Window icon displays the identifying NightView image, and each of the other windows includes this image as part of its icon. If you are displaying the icon images (it is possible to display only the icon labels), you can quickly see which iconified windows belong to the NightView application.

GUI Online Help

The graphical user interface provides several ways of providing help on particular topics.

- Context-sensitive help is available in all major NightView windows. See "Context-Sensitive Help" on page 9-3.
- Each of the major windows has a Help Menu. See "Help Menu" on page 9-3.
- Pressing the F1 function key displays help for the part of the window that has the current focus.
- Some of the windows have help buttons that pop up help for the particular window.
- You can use the **help** command from the command-line interface. See "help" on page 7-111.

Help information is displayed in a Help Window. NightView uses a separate program to display the Help Window. Once a Help Window is displayed, you can move around in the help system in a variety of ways. You can keep the Help Window on your screen, or dismiss it. You can also iconify it, and it redisplays itself the next time you ask for help. See "Help Window" on page 9-50.

Context-Sensitive Help

Context-sensitive help is available through the Help menu found in each major NightView window. See "Help Menu" on page 9-3. In addition, the <Help> key (usually this is the F1 function key) displays help information for the currently selected window component.

Generally, help is not provided on individual graphical items, such as individual buttons. Instead, you are given help for the region you have selected. For example, if you select help on the Detach button in the Dialogue Window, the Help Window displays information about the process summary area. See "Process Summary" on page 9-18.

To get context-sensitive help using the Help menu, select the On Context menu item. The pointer changes to a question mark with an arrow. Place the point of the arrow over the graphical region for which you want help and click mouse button 1. The Help Window is displayed with information about that region. The pointer changes back to its original shape.

To get context-sensitive help using the F1 (Help) key, select a window component that you have a question about. Press the F1 (Help) key. A Help Window is displayed with information about that region.

Help Menu

Mnemonic: H

Each major window in NightView has a Help menu. The Help menu in each window has the same menu items.

On Context

Mnemonic: C

This item provides help about a particular graphical region of a window. See "Context-Sensitive Help" on page 9-3.

On Last Error

Mnemonic: E

If NightView just displayed an error message, you can get help on that error by selecting this menu item.

Selecting this item is similar to using the **help** command with no argument. See "help" on page 7-111.

On Help

Mnemonic: H

This item gives help about how to use NightView's help system. See "GUI Online Help" on page 9-2.

On Window

Mnemonic: W

This item gives help about the window where you selected the Help menu.

On Commands

Mnemonic: M

This item gives a summary of NightView commands.

On Keys

Mnemonic: K

This item gives help about using special keys in NightView. See "Keys" on page 9-10.

Index

Mnemonic: I

This item shows an index that lists all the help topics available for NightView.

Table of Contents

Mnemonic: N

This item shows a table of contents that lists all the help topics available for NightView.

A Quick Start

Mnemonic: Q

This item takes you to the beginning of the GUI quick start chapter. See Chapter 2 [A Quick Start - GUI] on page 2-1.

Tutorial

Mnemonic: T

This item takes you to the beginning of the GUI tutorial chapter. See Chapter 5 [Tutorial - GUI] on page 5-1.

On Version

Mnemonic: V

This item pops up an information dialog box that describes which version of NightView you are running.

Help Buttons

Dialog boxes include a Help button in the lower right corner. You can click on this

button to receive help on the dialog box. See “Dialogues and Dialog Boxes” on page 9-10.

Help Command

You can type the **`help`** command, followed by the topic you want help on, into the command entry area of a major NightView window to obtain online help. See “help” on page 7-111. A Help Window is displayed that contains information about the requested topic. See “Help Window” on page 9-50.

If a Help Window does not exist, NightView displays one for you. Otherwise, the text of the existing Help Window changes to show you the information that you requested.

If NightView cannot find the information you requested, a warning dialog box and a Help Window are displayed. See “Warning Dialog Box” on page 9-15. You must acknowledge the warning before you can interact with any of the other NightView windows. Click on the Dismiss button.

GUI Components

This section describes GUI components that you need to understand to use the graphical user interface.

Text Input Areas

Text input areas receive text from the keyboard. The most important examples of text input areas are the single line input areas in the major windows, which are used to enter NightView commands. See “Debug Command Area” on page 9-35.

The command areas use a *combo box* to provide access to the command history. See “Combo Boxes” on page 9-6.

Text input areas have many special keys that can be used to position the text cursor and to edit the text. A description of all the special keys is beyond the scope of this chapter. However, this section describes some of the most important keys as they apply to NightView's command areas. For more information on keys, see the *OSF/Motif User's Guide*.

Text input areas can take input only when they have the keyboard focus. See “Keyboard Focus” on page 9-10.

For color devices, NightView uses a different color for areas of a window that you can type into. You can configure this color. Refer to the NightView color application defaults file. See Appendix D [GUI Customization] on page D-1.

left and right arrows

> move the cursor by one character to the left or right, respectively

Home

moves the cursor to the beginning of the line of text

End

moves the cursor to the end of the line of text

Return

completes the text entry

Backspace

deletes the character before the text cursor

Delete

deletes the character following the text cursor

Combo Boxes

Combo boxes combine a text input area and a drop-down list (see "Text Input Areas" on page 9-5). You can see the list by clicking on the downward-pointing arrow next to the text input area. You may then select any item in the list. The selected item replaces the text in the text input area. You may then edit that text and enter it if desired.

You can also use the keyboard to manipulate the combo box. Use the up arrow or down arrow to replace the text with the next item in the list without displaying the list. Display the list by holding down CTRL and pressing the up arrow or the down arrow. When the list is displayed you can move within the list with up arrow and down arrow. Hide the list by pressing Esc.

NightView uses combo boxes to provide access to the command history. See "GUI Command History" on page 9-12.

Message Areas

Each major NightView window (Global, Dialogue, and Debug windows) has an output area that displays messages pertaining to that element along with the output of commands and actions performed in that window. These message areas are scrolling text areas, and each is headed by the word "Messages:" above the scrolling text. See "Global Output Area" on page 9-49. See "Dialogue Message Area" on page 9-17. See "Debug Message Area" on page 9-28.

Above each scrolling text area, to the right of the "Messages:" label, is an area used to provide feedback to the user when NightView is busy performing a task that might prevent or delay other user interaction. This area will display one of two forms of feedback:

- An output-only text field displaying a message. The message indicates the task that NightView is performing. The background color of this field indicates that it is an output-only field. See "NightStar Color Resources" on page D-4.

- A progress bar displaying both a message and a visual indication of progress. The message again indicates the task that NightView is performing, and the progress bar gives an approximate indication of how much of the task is done and how much is left to do. The colors used for progress indication are user customizable; see "NightStar Color Resources" on page D-4.

  Some tasks involve an amount of work that is difficult to quantify from the beginning. In those cases, a number may be included in the feedback message that indicates the current estimate of the amount of work to be done. This number may change as the task progresses, and consequently the progress bar may "back up" rather than progress smoothly. This is normal behavior.

Some examples of tasks that you may see feedback for are:

Initializing process *name*

This message appears when NightView is preparing a new process, executing file *name*, for debugging.

Initial scan of object file *name*

This message appears when NightView is scanning the debug information of file *name* prior to debugging a program for the first time. The named file may be the name of either an executable program or a shared library.

Translating *n* type definitions in *name*

If you are debugging a program that was built from many different source files compiled separately, your program may contain debug information for the same user-defined type many times. NightView must resolve these many definitions before it can manipulate items of that type. Because NightView attempts to minimize overhead by reading and interpreting debug information only when required, this process may be incurred at any time during your debug session.

This type resolution process is typically only incurred in C and C++ programs, although it may also be required for some Ada programs that use shared libraries. You may be able to eliminate or considerably reduce the amount of time NightView takes to resolve these type definitions by running the **`cprs(1)`** program on your executable file.

Note that, once NightView has resolved the definition of a particular data type, it does not need to resolve *that* type again for that executable, regardless of how many times you debug that program during your NightView session. As long as you do not exit NightView and do not modify the executable file, NightView will be able to retain the information it has acquired from debug information and thus reduce your debugging time. See "Restarting a Program" on page 3-14.

File Selection Dialog Box

A file selection dialog box allows you to browse through directories and choose a file from a list. Or, you can type in a file name. You can change directories and view subdirectories and their files. Typically, the file selection dialog box lists files in the current directory. In some cases, NightView may instruct the file selection dialog box to

list certain files in a specific directory.

There are two possible versions of the file selection dialog box; this section describes the default version used by NightView. You can use the other version if you comment out the following resources in `/usr/lib/X11/app-defaults/Nview`:

```
*XmFileSelectionBox.pathMode
*XmFileSelectionBox.fileFilterStyle
```

Also, see the *OSF/Motif Style Guide*.

The file selection dialog box consists of a Directory text input area for the directory name, a Filter mask, a list of subdirectories, a list of files, a Selection text input area for the filename, and buttons that allow you to take actions related to the file selection dialog box. See "Text Input Areas" on page 9-5.

See "List Selection Policies" on page 9-9.

Directory.

This area shows the name of the directory whose files and subdirectories appear in the lists.

This is a text input area. You can change the directory name; click on the Filter button and the file selection dialog box updates the Directories and Files lists.

File Filter.

By editing the Filter string and clicking on the Filter button, you can change the files that are displayed in the Files lists.

This is a text input area. See "Text Input Areas" on page 9-5.

Directories List.

This list shows the subdirectory names that are located in the directory indicated by the Directory string.

To choose a directory from the list, click on its name in the list and click on the Filter button. Double-clicking on a directory entry changes the Filter directory to that directory. The list of subdirectories and the list of files are also changed.

Files List.

This list shows the file names that are located in the directory indicated by the Directory string.

To select a file from the list, you can click on its name in the list and click on the OK button. You can also double-click on a file in the list to select that file.

File Selection.

This area shows the currently selected file name in the Files list, or you can type in a file name.

This is a text input area. See "Text Input Areas" on page 9-5.

Action Area Buttons.

If you are satisfied with the file name selection, click on the OK button. NightView uses the file you chose; how the file is used depends on the application context.

Clicking on Filter changes the Directories and Files list contents to reflect the contents of the Directory and Filter fields.

Clicking on Cancel cancels the current action and closes this dialog box.

You can get help for this dialog box by clicking on Help.

List Selection Policies

A list allows you to select one or more items. The selected items are highlighted. Once selected, you can cause some action to be taken on the items; usually, this action is invoked by pressing a button near the list.

With some lists in NightView, you can change the default selection policy. The resource that controls the list selection policy is `selectionPolicy`. Refer to the NightView application defaults file to determine which windows have list selection policies that are configurable. See Appendix D [GUI Customization] on page D-1.

Lists may have different selection policies, depending on what type of selection is most appropriate in a given application context. For example, a list may allow only one item at a time to be selected, or it may allow you to select multiple discontiguous items. Unless otherwise indicated, *browse* is the default list selection policy.

In the case where it is appropriate to select only one item at a time, there are two possible selection policies: *browse* and *single*.

The *browse* selection policy allows you to select, at most, one item. One item is always selected, although the list may initially display with no selected item. You can click on an item to select it, or you can hold down mouse button 1 and drag the pointer through the list of items, scrolling the list. As you browse through the list with the mouse pointer, the selected item changes.

The *single* selection policy allows you to click on an item to select it. Click on a selected item to deselect it. At most, one item is selected. There may be no item selected.

In some cases, a list allows multiple list items to be selected. For these lists, there are two possible selection policies: *multiple* and *extended.*

The *multiple* selection policy allows you to click on one or more items to select items. Clicking on a selected item deselects it. You can select all items by using Ctrl+ /. You can deselect all items by using Ctrl+ \.

The *extended* selection policy allows you to select multiple discontiguous ranges of items. Use mouse button 1 to drag the pointer and select a range of items. Once you have selected one or more items, press the Ctrl key while using the mouse pointer to add more items to the set of currently selected items. You can click on any item to deselect all other items in the selection set; that item will be selected. To deselect items, use Ctrl while clicking on a selected item or while dragging the pointer through a range of selected items.

You can also use keyboard methods to select and deselect all items in a list with an extended selection policy. You select all items by using Ctrl+ /. While in *normal mode* (notice that the location cursor is a solid box), you can deselect all list items, except the item indicated with the location cursor, by using Ctrl+ \. To deselect all items in the list, you must change to *add mode* (notice that the location cursor is a dashed box), and use Ctrl+ \. The standard key binding for toggling between normal mode and add mode is Shift+F8.

For more information on list selection policies, virtual keys, and common key bindings see the *OSF/Motif Style Guide.* For information on using lists, see the *OSF/Motif User's Guide*.

Dialogues and Dialog Boxes

NightView has a concept called a *dialogue*, which is a way of communicating with an ordinary command shell. See "Dialogues" on page 3-4. Note that this kind of dialogue is spelled with a "ue" at the end.

The graphical user interface uses another term: *dialog box*. This is not related to the NightView concept of a *dialogue*. *Dialog box* refers to a particular type of window that may appear during your session. A dialog box usually appears only briefly and typically allows you to specify a particular item, such as a file name.

These two concepts are distinct and unrelated, even though they sound alike.

Keyboard Focus

The GUI uses the concept of *keyboard focus*. Keyboard input is accepted only in a field when that field has the keyboard focus. When a field of a window has the keyboard focus, the window is also considered to have the keyboard focus, for the purposes of using mnemonics and accelerators. See "Keys" on page 9-10. The field that has the keyboard focus is highlighted.

How you set the keyboard focus depends on the focus policy. If the focus policy is *pointer*, then the keyboard focus is in whatever field the pointer is in. If the focus policy is *explicit*, then you must take some action to move the keyboard focus to a field. You can do this by clicking on the field or by using certain keys. See "Keys" on page 9-10.

The default keyboard focus policy for NightView is *explicit*. The resource used to control this is `keyboardFocusPolicy`. Information about how to change this resource can be found in the NightView application defaults file. See Appendix D [GUI Customization] on page D-1.

For more information on how to manipulate the keyboard focus, see the *OSF/Motif User's Guide*.

Keys

NightView uses certain key combinations as shortcuts for displaying menus and selecting menu items. These key combinations are called *accelerators* and *mnemonics*. Each window has its own set of accelerators and mnemonics that are active only while the

keyboard focus is in that window. However, the keyboard focus does not have to be in any particular field of the window to use accelerators and mnemonics. See "Keyboard Focus" on page 9-10.

Menus can be displayed with mnemonics.

> Menus can be displayed from the keyboard by typing Alt+*mnemonic*. Each of the main windows has a menu bar near the top of the window. The different menus are labeled. For example, the Debug Window has a Process menu. If you look at the Process menu, you can see that the P is underlined. P is the mnemonic for the Process menu. That means that, in addition to displaying the Process menu by clicking on it with mouse button 1, you can also display it with Alt+ p (hold down Alt and press p).
>
> If you decide you don't want to select any of the menu items, you can make the menu go away by typing Esc or by clicking somewhere else.

Menu items can be selected with mnemonics.

> Once a menu is displayed, you can select a menu item by typing only the mnemonic for that item. The mnemonics for the menu items are underlined, just as the mnemonics for the menus are underlined. To select a menu item by using its mnemonic, just press the key.

Menu functions can be invoked with accelerators.

> Some commonly used menu items have accelerator keys. The functions associated with these menu items can be invoked directly, without displaying the menu, by pressing the accelerator keys. The accelerator keys for a particular menu item are listed next to the item in the menu.
>
> The accelerator keys are often a combination of a control key plus a letter, such as Ctrl+ O. To type Ctrl+ O, hold down the control key and press 0.

In addition to mnemonics and accelerators, there are also special keys used for navigation within and among windows and fields. These keys include Tab, Shift+Tab, Home, End, Page Up, Page Down and the arrow keys. The documentation of these keys is beyond the scope of this chapter. For more information about keys, see the *OSF/Motif User's Guide*.

There are many special keys used to edit text input areas. See "Text Input Areas" on page 9-5.

Sashes

Some of the windows are divided into panes and have sashes. A sash is a little box near the right end of the line that separates the panes.

A sash may be used to change the sizes of two adjoining panes, relative to each other. You can do this by dragging the sash with mouse button 1. As you increase the size of one pane, the adjoining pane's size is decreased proportionally. The size of the window does not change, only the sizes of the adjoining panes within the window are affected.

For more information about Paned Windows, see the *OSF/Motif User's Guide*.

Toggle Buttons

A toggle button is a graphical element that can be toggled on or off. There are two types of toggle buttons: check buttons and radio buttons. More than one check button can be selected in a group of check buttons, whereas only one radio button can be selected in a group of radio buttons.

The graphical item used for a check button to indicate the on state is either a check mark graphic in a square box (the default), or a filled square check box. The graphical item used for a radio button to indicate the on state is either a filled circle (the default), or a filled diamond. The off state is indicated with an empty box, circle or diamond.

You can configure the selection color of the toggle button by defining the `selectColor` resource. Refer to the NightView color application defaults file. See Appendix D [GUI Customization] on page D-1.

GUI Command History

NightView keeps a history of the commands you enter. See "Command History" on page 3-32. In the graphical user interface you can access the command history through the combo box in the command area of each major window.

The combo box in each window shows the entire history from all the windows. See "Combo Boxes" on page 9-6.

Understanding the Debug Window

This section explains the concepts you need to understand so that you can debug and manipulate processes in a NightView Debug Window.

Debug Window Behavior

NightView automatically creates one Principal Debug Window. This Debug Window contains all processes that appear in a NightView session. You can debug processes using only this window, or you can create additional Debug Windows and define which processes appear in them.

Any single process may be represented in one or more Debug Windows at a time. While all Debug Windows share common behavior traits, the behavior of the Principal Debug Window varies slightly from Debug Windows that you create.

Common Debug Window Behavior.

> NightView allows you to control one or more processes in one or more Debug Windows. You can choose to manipulate one process at a time, or to manipulate all the processes in the window as a group. To accomplish this, the Debug Window allows you to switch between *single* and *group* process modes. See "Single Process Mode" on page 9-13. See "Group Process Mode" on page 9-14.

If the window is in single process mode, commands and actions apply to the currently displayed process. If the window is in group process mode, commands and actions apply to each of the processes in the group area list. See “Debug Group Area” on page 9-35.

New processes always appear in the same windows as their parent.

If a process exits, it is removed from the group list of all Debug Windows where it appeared.

You can choose to close a Debug Window at any time during the NightView session. Closing a Debug Window has no effect on the processes that are represented in that window.

Principal Debug Window.

The Principal Debug Window can be empty.

This window remains available throughout the NightView session; it is not automatically closed. If you choose to close it, the Principal Debug Window can be reopened by using the `NightView` menu found in the Debug, Dialogue, and Global Windows.

User-Created Debug Windows.

You can create other Debug Windows and define which processes initially appear in each window. See “Debug Group Selection Dialog Box” on page 9-36.

In contrast to the Principal Debug Window, a Debug Window that you create is never empty; NightView automatically closes the window when the last process in the window exits.

You can tell NightView to automatically display each process in its own Debug Window by setting the `oneWindowPerProcess` resource to `True` (the default is `False`). See Appendix D [GUI Customization] on page D-1. When this resource is `True`:

- NightView displays a separate Debug Window for each process. Any Debug Windows created this way are considered to be user-created Debug Windows.
- NightView sets the window's title to the process's qualifier.
- The Principal Debug Window is not automatically displayed.
- You might also want to consider setting the `displayGroupToggleButton.set` resource to `False`. See “Debug View Menu” on page 9-26.

Single Process Mode

By default, the Debug Window is in single process mode. This means that any commands that you issue apply only to the currently displayed process. This includes commands that are typed into the command area or commands that are issued using graphical methods. If there is more than one process in the window, you can change the currently displayed process by selecting a process from the debug group area list and

clicking on the Switch To button. See "Debug Group Area" on page 9-35. Initially, the process that occurs first in the group area list is the currently displayed process in the source display area. See "Debug Source Display" on page 9-31.

When the Debug Window is in single process mode, some of the command buttons may be disabled to indicate that their use is not appropriate at this time. For example, when the selected process is stopped, the Stop button is disabled. Any messages generated by commands are displayed in the debug message area. See "Debug Message Area" on page 9-28.

You can determine when the Debug Window is in single process mode by looking at the debug qualifier area. See "Debug Qualifier Area" on page 9-34. When the window is in single process mode, you see the Qualifier: label and the process's qualifier displayed here. Otherwise, you see the phrase [Group Mode]. See "Group Process Mode" on page 9-14.

The View menu contains radio buttons that also indicate which of the two modes is currently set, and allows you to change your view of the window between single and group process mode. See "Debug View Menu" on page 9-26. See "Toggle Buttons" on page 9-12.

There are keyboard accelerators associated with these menu items which allow you to switch between modes without displaying the menu. See "Keys" on page 9-10.

Group Process Mode

If you want to issue commands that apply to more than one process, you can do this by changing to group process mode. This means that any commands that you issue apply to each of the processes listed in the group area. This includes commands that are typed into the debug command area or commands issued using graphical methods.

When the Debug Window is in group mode, all of the command buttons are enabled and any messages generated by any of the processes in the group are displayed in the debug message area.

You can determine when the Debug Window is in group process mode by looking at the debug qualifier area. See "Debug Qualifier Area" on page 9-34. When the window is in group process mode, you do not see the Qualifier: label, and instead of a specific qualifier you see the phrase [Group Mode]. To see the value of the qualifier, use the View menu item Show Qualifier.... See "Debug View Menu" on page 9-26.

The View menu contains radio buttons that also indicate which of the two modes is currently set, and allows you to change your view of the window between single and group process mode. See "Debug View Menu" on page 9-26. See "Toggle Buttons" on page 9-12.

There are keyboard accelerators associated with these menu items which allow you to switch between modes without displaying the menu. See "Keys" on page 9-10.

Confirm Exit Dialog Box

If you try to close a window and NightView determines that this is the last visible

window on your screen, NightView assumes you want to exit the debugger. NightView displays a dialog box allowing you to confirm this assumption.

Message.

The dialog box that pops up contains text that indicates that this is the last open window and asks you if you want to exit the debugger.

Action Area Buttons.

Selecting the OK button tells the debugger to go ahead and exit the debugger.

Selecting the Cancel button tells the debugger to cancel the request to exit the debugger.

If you wish to get help, select the Help button.

You must select either the OK button or the Cancel button before you can continue.

Warning and Error Dialog Boxes

If an error occurs, or if you have instructed NightView to take an action that may result in the loss of data, NightView displays warning or error windows to alert you to the error or the unsafe action. Often, you need to acknowledge the warning or error before you can continue by clicking on one of the buttons. A default choice is indicated by a highlighted box around one of the buttons.

Warning Dialog Box

Certain actions performed by the debugger are considered unsafe. They cause a warning dialog box to pop up and ask you for verification to perform the unsafe action.

Warning Message.

The warning dialog box that pops up contains text that specifies the unsafe action that is to be performed.

Action Area Buttons.

Selecting the OK button tells the debugger to go ahead and perform the unsafe action.

Selecting the Cancel button tells the debugger to cancel the request to perform the unsafe action.

If you wish to get help, press the F1 (Help) key. Or, you can select the Cancel button and then either get help on the last diagnostic or error message that was displayed or on the section that was referenced by the last diagnostic message or error message. See “Help Menu” on page 9-3.

You must select either the OK button or the Cancel button before you can use any other NightView windows.

Error Dialog Box

If you make an error while using NightView, an error dialog box may pop up to inform you of the mistake.

Error Message.

The error dialog box that pops up contains a message about the error condition.

Action Area Buttons.

Click on OK to acknowledge the error and dismiss the error dialog box.

If you wish to get help, press the F1 (Help) key.

You must acknowledge the error by selecting the OK button before you can use any other NightView windows.

Dialogue Window

The Dialogue Window lets you communicate with and control a NightView dialogue. See "Dialogues" on page 3-4.

Any programs that you run in the dialogue I/O area can be debugged and manipulated by NightView. See "Dialogue I/O Area" on page 9-17.

Dialogue Menu Bar

The dialogue menu bar lets you perform global NightView actions, control the dialogue and access online help.

Dialogue NightView Menu

Mnemonic: N

The NightView menu is used to control NightView windows and perform global NightView actions. The NightView menu appears in the Debug, Dialogue and Global windows and has the same menu items in each window.

See "Debug NightView Menu" on page 9-20, for a description of the individual NightView menu items.

Dialogue Menu

Mnemonic: D

The Dialogue menu lets you terminate the dialogue.

Logout

Mnemonic: L

Selecting this item terminates the dialogue and closes the Dialogue Window. This is similar to using the **logout** command. See "logout" on page 7-23.

Depending on the safety level (see "set-safety" on page 7-49) and whether there are any active processes, NightView may display a warning dialog box when you use the Logout menu item. See "Warning Dialog Box" on page 9-15.

Dialogue Help Menu

Mnemonic: H

This menu provides ways of getting context-sensitive help, help on the current window, help on the last error NightView encountered, as well as several other categories of help. NightView help information is displayed in a Help Window. See "Help Window" on page 9-50.

The Help menu is described in another section. See "Help Menu" on page 9-3.

A general discussion of NightView's online help is also available. See "GUI Online Help" on page 9-2.

Dialogue Identification Area

This area shows the name of the particular dialogue that this window is associated with.

There is also a label showing the name of the system the dialogue is running on.

Dialogue Message Area

This area displays messages related to this dialogue. The displayed information includes process exit messages, error messages and output from commands that are processed by this Dialogue Window.

This is a scrolling area. You can use the scroll bar to look at older or newer messages.

You can change the height of this area by moving the sash up or down. See "Sashes" on page 9-11.

Dialogue I/O Area

This area allows you to interact with the dialogue shell and with your programs. See "Dialogues" on page 3-4. You can run your program here, just as you would normally run it, providing any arguments that it needs. Shell and program output is displayed here.

You can also enter input to the shell and to your programs. This window acts something like a little terminal. If your shell lets you do command-line editing, then you can do that in this window, too.

This is a scrolling area. You can use the scroll bar to look at older or newer output.

You can change the height of this area by moving the sash up or down. See "Sashes" on page 9-11.

Dialogue Interrupt Button

Clicking on this button interrupts whatever the debugger is doing. This is similar to using the shell interrupt character in the command-line interface. See "Interrupting the Debugger" on page 3-30.

Dialogue Qualifier Area

The dialogue qualifier area is a label to remind you that commands entered in the dialogue command area are implicitly qualified by the dialogue associated with this Dialogue Window. The label shows the name of the dialogue.

Unlike the default qualifier in the global qualifier area in the Global Window, you cannot change this qualifier.

Dialogue Command Area

The dialogue command area in the Dialogue Window is used to enter NightView commands. Like the debug command area in the Debug Window and the global command area in the Global Window, all the command-line interface commands, except for **`shell`**, can be entered in the dialogue command area.

Input to this area is similar to using the command-line interface. For example, you can enter an explicit qualifier followed by a command. If you do not specify a qualifier, the command is implicitly qualified by the dialogue associated with this Dialogue Window.

The dialogue command area is a combo box. See "Combo Boxes" on page 9-6.

Process Summary

The process summary provides a list of all the processes that exist in the dialogue. The list is followed by buttons that provide related process actions. Select one or more processes, then press one of the buttons. The button action that you choose applies to all selected processes.

The buttons allow you to detach and terminate processes.

To detach from a process, first select one or more processes in the summary window. The selected processes are highlighted. Then click on Detach. The selected processes are detached from the dialogue. This is similar to using the **`detach`** command. See "detach" on page 7-32.

The Kill button may be used to terminate one or more processes. This is similar to using the **`kill`** command. See "kill" on page 7-33.

The default list selection policy is extended, which means you can select discontiguous ranges of items. This list selection policy is configurable. (The only other selection policy that is appropriate is multiple.) See "List Selection Policies" on page 9-9. See Appendix D [GUI Customization] on page D-1.

You can change the height of this area by moving the sash up or down. See "Sashes" on page 9-11.

Dialogue Window Dialog Boxes

This section describes dialog boxes that may appear while you are using the Dialogue Window.

Program Arguments Dialog Box

This dialog box pops up if you invoke NightView with a program name as a command-line argument (see Chapter 6 [Invoking NightView] on page 6-1). It allows you to specify the arguments that your program expects. The message in the dialog box tells you the name of the program and what to do.

You cannot interact with other NightView windows, except the Help Window, until you select either OK or Cancel in this dialog box.

Enter Program Arguments.

> Enter the arguments, if any, in the text input area. Pressing Return activates the OK button.
>
> See "Text Input Areas" on page 9-5.

Choose an Action Button.

> If you are satisfied with the arguments you entered, click on OK.
>
> If you decide you do not want to debug this program, click on Cancel. You can still debug the program later, by entering the appropriate shell command in the dialogue I/O area. See "Dialogue I/O Area" on page 9-17).
>
> You can get help for this dialog box by clicking on Help.

The dialog box will disappear, and you should see a shell command for your program, with the arguments you specified in this dialog box, appear in the dialogue I/O area. The program is started, causing a Debug Window to appear for it; at that point, you can debug the program. See "Debug Window" on page 9-20.

Debug Window

The Debug Window provides the primary means of debugging and manipulating one or more processes.

By default, the window is in single process mode, which means you can debug and manipulate the currently displayed process or switch to any other process represented in this Debug Window. See "Single Process Mode" on page 9-13. See "Debug Group Area" on page 9-35. If you want to debug and manipulate all the processes represented in this window at the same time, you can change to group process mode. See "Group Process Mode" on page 9-14. See "Debug View Menu" on page 9-26.

You can create Debug Windows and define the group of processes that appear in them. See "Debug NightView Menu" on page 9-20.

The behavior of a Debug Window differs slightly depending on whether it is the Principal Debug Window (created automatically by NightView) or a Debug Window created by you. See "Debug Window Behavior" on page 9-12.

Debug Menu Bar

From the debug menu bar you can perform global NightView actions, perform actions on one or more processes, choose source to display or edit, manipulate eventpoints, change the way you view the window, and obtain online help.

Debug NightView Menu

Mnemonic: N

The NightView menu is used to control NightView windows and perform global NightView actions. The NightView menu appears in the Debug, Dialogue and Global windows and has the same menu items in each window.

Create Debug Window...

Mnemonic: D

Selecting this menu item allows you to create a new Debug Window. See "Debug Window" on page 9-20.

A dialog box is displayed that allows you to select one or more qualifier specifiers to define the new window. You can also provide a name for the new Debug Window. See "Qualifier Specifiers" on page 7-10.

See "Debug Group Selection Dialog Box" on page 9-36.

Open Principal Debug Window

Mnemonic: P

Selecting this menu item opens the Principal Debug Window. See “Debug Window Behavior” on page 9-12.

This menu item is disabled (dimmed) if the Principal Debug Window is already open.

Open Global Window

Mnemonic: G

Selecting this menu item opens the Global Window.

This menu item is disabled (dimmed) if the Global Window is already open. See “Global Window” on page 9-48.

Start Remote Dialogue...

Mnemonic: R

Selecting this menu item allows you to create a remote dialogue on a target system of your choice. A dialog box is displayed that allows you to choose parameters for the remote dialogue. See “Remote Login Dialog Box” on page 9-45.

Close Window

Mnemonic: C

Selecting this menu item closes this window and any related dialog box windows.

If this is a Debug Window, closing the window has no effect on the processes in the window. If this is a Dialogue Window, closing the window has the same effect as logging out of the dialogue. See “Dialogue Menu” on page 9-16.

Exit (Quit NightView)

Mnemonic: X

Accelerator: Ctrl+ Q

Selecting this menu item causes NightView to exit. This has the same effect as the **`quit`** command. See “quit” on page 7-17.

Depending on the safety level (see “set-safety” on page 7-49) and whether there are any active processes, NightView may display a warning dialog box when you use the Exit menu item. See “Warning Dialog Box” on page 9-15.

Debug Process Menu

Mnemonic: P

This menu is used to perform actions on processes.

If the window is in single process mode, the menu item you select will affect only the currently displayed process. See “Single Process Mode” on page 9-13. If the window is in group process mode, then the menu item you select will act on each of the processes in the group area list. See “Group Process Mode” on page 9-14.

Detach

Mnemonic: D

Selecting this item causes NightView to detach from the currently displayed process (if in single process mode) or from each process listed in the group area list (if in group process mode). See "Single Process Mode" on page 9-13. See "Group Process Mode" on page 9-14. See "Debug Group Area" on page 9-35.

This is similar to using the **detach** command. See "detach" on page 7-32.

Depending on the safety level (see "set-safety" on page 7-49), NightView may display a warning dialog box when you use the Detach menu item. See "Warning Dialog Box" on page 9-15.

Kill

Mnemonic: K

Selecting this item causes NightView to terminate the currently displayed process (if in single process mode) or each process listed in the group area list (if in group process mode). See "Single Process Mode" on page 9-13. See "Group Process Mode" on page 9-14. See "Debug Group Area" on page 9-35.

This is similar to using the **kill** command. See "kill" on page 7-33.

Depending on the safety level (see "set-safety" on page 7-49), NightView may display a warning dialog box when you use the Kill menu item. See "Warning Dialog Box" on page 9-15.

Debug Source Menu

Mnemonic: S

This menu provides ways of changing the program code displayed in this window's source display area and editing source files that are listed. See "Debug Source Display" on page 9-31.

Because the source display area shows only one process's program code at a time, the items in this menu act independently of whether the window is in single or group process mode. See "Single Process Mode" on page 9-13. See "Group Process Mode" on page 9-14.

List Function/Unit...

Mnemonic: F

Selecting this menu item pops up a dialog box that allows you to list the program code of a function or Ada unit in the debug source display. See "Debug Source Display" on page 9-31.

This dialog box is titled Select a Function/Unit, and displays the process's qualifier specifier. See "Qualifier Specifiers" on page 7-10. It allows you to optionally enter a regular expression that is used to search for function names that NightView knows about. (An anchored match is *not* implied.) See "Regular Expressions" on

page 7-12. For example, enter `set$` to search for function names ending with 'set'. A list of functions is displayed, and one function can be selected for display in the debug source display. For Ada and C++, the regular expression is only applied to the final component of a name.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 7-54).

The Select a Function/Unit dialog box is one variation of the debug source selection dialog box, which is also used by the List Source File... menu item. See "Debug Source Selection Dialog Box" on page 9-37.

List Source File...

Mnemonic: S

Selecting this menu item pops up a dialog box that allows you to list a source file in the debug source display. See "Debug Source Display" on page 9-31.

This dialog box is titled Select a Source File, and displays the process's qualifier specifier. See "Qualifier Specifiers" on page 7-10. It allows you to optionally enter a wildcard pattern which is used to search for source file names that NightView knows about. See "Wildcard Patterns" on page 7-14. For example, enter `mod*.c` to search for source file names that start with 'mod' followed by any number of characters and ending with '.c'. A list of source files is displayed, and one source file can be selected for display in the debug source display.

The Select a Source File dialog box is one variation of the debug source selection dialog box, which is also used by the List Function/Unit... menu item. See "Debug Source Selection Dialog Box" on page 9-37.

List Any File...

Mnemonic: A

Selecting this menu item pops up a file selection dialog box that allows you to choose any file you wish and list it in the debug source display. See "Debug Source Display" on page 9-31.

This dialog box is titled Select a File. See "Debug File Selection Dialog Box" on page 9-38.

Edit

Mnemonic: E

Selecting this item lets you edit the source file that is currently displayed in the debug source display. See "Debug Source Display" on page 9-31.

There are some rules for determining how the editor is invoked. The resource `editor` is expected to be a string, *editorstring*, that describes how to invoke the editor. The string may contain variable specifiers, which are composed of a % followed by another character. The variable specifiers are replaced by an appropriate value to create the editor string. The variable specifier characters are:

%

Replaced by %. That is, to get a %, use %%.

`s`

Replaced by the name of the source file.

`l`

Replaced by the line number of the current position.

`p`

Replaced by the offset, in characters, of the current position from the beginning of the file.

`c`

Replaced by the column of the current position.

A % followed by any other character is ignored.

An example `editor` resource is:

```
nview*editor:                emacsclient +%l %s
```

If the `editor` resource is not defined, then the name of the editor is taken from the EDITOR environment variable. If there is no EDITOR variable, then **`vi`** is used. In these cases the editor is invoked with the name of the current source file as the sole argument.

If your editor can communicate with the X Window System display directly, then you should set the resource `editorTalksX` to `true`. Then the editor is invoked as *editorstring*. Otherwise, the editor is run via **`/usr/bin/X11/xterm -e`** *editorstring*.

Note that once you have edited the source file, NightView displays the *new* contents, but the debugging information still refers to the *old* contents. For this reason, the source decorations may no longer match. Also, you might get confusing results from using the special keys in the debug source display or from entering commands based on the new contents.

Debug Eventpoint Menu

Mnemonic: `E`

This menu provides ways to set and change eventpoints, and see a summary of eventpoints. See "Eventpoints" on page 3-8.

Before selecting one of the menu items, position the text insertion cursor on the line of interest in the debug source display. See "Debug Source Display" on page 9-31. NightView uses this line to determine the location specifier for you. See "Location Specifiers" on page 7-9.

Once you select a menu item, NightView displays the eventpoint dialog box for the selected item.

Set Breakpoint...

Mnemonic: B

Accelerator: Ctrl+ B

Selecting this menu item pops up a breakpoint dialog box that allows you to set a new breakpoint at a given location and apply eventpoint attributes to it. See "Manipulating Eventpoints" on page 7-77. See "breakpoint" on page 7-79.

For information on using the breakpoint dialog box, see "Debug Eventpoint Dialog Boxes" on page 9-38.

Like the Breakpoint button, this menu item allows you to set a breakpoint. But using the breakpoint dialog box provides you with more control and flexibility. Using the Breakpoint button, you can only set a simple breakpoint. See "Debug Command Buttons" on page 9-32.

Set Monitorpoint...

Mnemonic: M

Accelerator: Ctrl+ M

Selecting this menu item pops up a monitorpoint dialog box that allows you to set a new monitorpoint at a given location and apply eventpoint attributes to it. See "Manipulating Eventpoints" on page 7-77. See "monitorpoint" on page 7-84.

For information on using the monitorpoint dialog box, see "Debug Eventpoint Dialog Boxes" on page 9-38.

Set Patchpoint...

Mnemonic: P

Accelerator: Ctrl+ P

Selecting this menu item pops up a patchpoint dialog box that allows you to set a new patchpoint at a given location and apply eventpoint attributes to it. See "Manipulating Eventpoints" on page 7-77. See "patchpoint" on page 7-80.

For information on using the patchpoint dialog box, see "Debug Eventpoint Dialog Boxes" on page 9-38.

Set Tracepoint...

Mnemonic: T

Accelerator: Ctrl+ T

Selecting this menu item pops up a tracepoint dialog box that allows you to set a new tracepoint at a given location and apply eventpoint attributes to it. See "Manipulating Eventpoints" on page 7-77. See "tracepoint" on page 7-83.

For information on using the tracepoint dialog box, see "Debug Eventpoint Dialog Boxes" on page 9-38.

Set Agentpoint...

Mnemonic: A

Accelerator: Ctrl+ A

Selecting this menu item pops up an agentpoint dialog box that allows you to set a new agentpoint at a given location and apply eventpoint attributes to it. See "Manipulating Eventpoints" on page 7-77. See "agentpoint" on page 7-87.

For information on using the agentpoint dialog box, see "Debug Eventpoint Dialog Boxes" on page 9-38.

Set Watchpoint...

Mnemonic: W

Accelerator: Ctrl+W

Selecting this menu item pops up a watchpoint dialog box that allows you to set a new watchpoint at a given location and apply eventpoint attributes to it. See "Manipulating Eventpoints" on page 7-77. See "watchpoint" on page 7-95.

For more information on using the watchpoint dialog box, see "Debug Eventpoint Dialog Boxes" on page 9-38.

Summarize/Change...

Mnemonic: U

Accelerator: Ctrl+ U

Selecting this menu item pops up an eventpoint dialog box that allows you to see a summary of eventpoints and make changes to existing eventpoints. See "Manipulating Eventpoints" on page 7-77.

For information on using the eventpoint summary dialog box, see "Debug Eventpoint Summarize/Change Dialog Box" on page 9-42.

Debug View Menu

Mnemonic: V

This menu allows you to change the way you view the contents of the Debug Window.

Display Group Area

Mnemonic: D

Accelerator: Ctrl+ D

This menu item displays a check button which is either *set* or *unset*, depending on whether the debug group area is displayed or hidden from view. See "Debug Group Area" on page 9-35. See "Toggle Buttons" on page 9-12.

The default initial setting is *set*: the debug group area is displayed. You can change this setting at any time by selecting this menu item.

You can change the initial setting by setting the resource `displayGroupToggleButton.set`. The default value of this resource is `True`. See Appendix D [GUI Customization] on page D-1.

Single Process Mode

Mnemonic: S

Accelerator: Ctrl+ S

Selecting this menu item causes the appearance of the window to change and causes NightView commands to operate on a single process, rather than on all the processes which may be represented in this window's group area. See “Single Process Mode” on page 9-13. See “Debug Group Area” on page 9-35. Single process mode is the default setting.

When the Debug Window is in single process mode, a filled radio button is shown next to the menu item to indicate that this option is selected. See “Toggle Buttons” on page 9-12.

The other member of this set of options is the Group Process Mode menu item. See “Group Process Mode” on page 9-14.

The behavior of the Debug Window when in single or group process mode is described in another section. See “Debug Window Behavior” on page 9-12.

Group Process Mode

Mnemonic: G

Accelerator: Ctrl+ G

Selecting this menu item causes the appearance of the window to change and causes NightView commands to operate on all the processes represented in the group area, rather than on a single process. See “Group Process Mode” on page 9-14. See “Debug Group Area” on page 9-35.

When the Debug Window is in group process mode, a filled radio button is shown next to the menu item to indicate that this option is selected. See “Toggle Buttons” on page 9-12.

The other member of this set of options is the Single Process Mode menu item. See “Single Process Mode” on page 9-13.

The behavior of the Debug Window when in single or group process mode is described in another section. See “Debug Window Behavior” on page 9-12.

Show Qualifier...

Mnemonic: L

Accelerator: Ctrl+ L

Use this menu item to see the value of the qualifier for this window. See "Qualifier Specifiers" on page 7-10. The qualifier is displayed in an information dialog box entitled Window Qualifier. Select OK to dismiss the dialog box.

The Window Qualifier dialog box is not dynamically updated when the qualifier changes. You must redisplay this dialog box each time you want to see the current value of the qualifier.

Debug Help Menu

Mnemonic: H

This menu provides ways of getting context-sensitive help, help on the current window, help on the last error NightView encountered, as well as several other categories of help. NightView help information is displayed in a Help Window. See "Help Window" on page 9-50.

The Help menu is described in another section. See "Help Menu" on page 9-3.

A general discussion of NightView's online help is also available. See "GUI Online Help" on page 9-2.

Debug Message Area

This area displays messages related to the processes represented by this window. The displayed information includes process status messages, error messages and output from commands that are processed by this Debug Window.

If the window is in single process mode, NightView displays output from only the currently selected process plus messages from any commands that are executed in this window while that process is selected. See "Single Process Mode" on page 9-13.If the window is in group process mode, then NightView displays output from all the processes in the group, plus messages from any commands that are executed in this window. See "Group Process Mode" on page 9-14.

This is a scrolling area. You can use the scroll bar to look at older or newer messages.

You can change the height of this area by moving the sash up or down. See "Sashes" on page 9-11.

Debug Identification Area

This area shows the name of the executable program that the currently displayed process is running.

There is also a label showing the qualifier specifier for this process. See "Qualifier Specifiers" on page 7-10.

Debug Source Lock Button

The Debug Window contains a source lock button. The source lock button looks like a little padlock.

You can lock the source display by clicking on the source lock button. The padlock changes from being unlocked to locked, and the button is highlighted with the selection color. This indicates that the source display is locked. Click on the button again to unlock.

A locked source display does not change which file is displayed unless you explicitly change it. These actions explicitly change the source display:

- Issuing a **`list`** command in the command area of the window.
- Using the Source menu.
- Issuing an **`up`**, **`down`** or **`frame`** command in the command area of the window.

These events can cause an *unlocked* source display to change:

- The currently displayed process stops.
- The source file for the currently displayed process changes because of an action in another window.

Debug Source File Name

This area shows the name of the source file displayed in the debug source display. See "Debug Source Display" on page 9-31. If there is no source file displayed, then this field shows No Source File. If NightView cannot find the source file, this field shows Cannot find: *filename*.

Debug Status Area

This area shows the status of the currently displayed process. Here are the values that this field may have:

About to exit

The process hit the exit breakpoint. See "Exited and Terminated Processes" on page 3-16.

Calling function

The process is executing to evaluate a function call.

Exited

The process has exited. See "Exited and Terminated Processes" on page 3-16. This

status does not normally appear, because the process is removed from the Debug Window when the process exits.

Finish frame

The process is executing until a designated instance of a subprogram returns to its caller. See "finish" on page 7-102.

New process

This process has just been created by a `fork()` call in the parent process. The process is stopped. See "Multiple Processes" on page 3-2.

Running

The process is currently executing.

Stepping

The process is executing because of a stepping command. See "step" on page 7-99.

Stopped after finish

The process has completed a **`finish`** command. See "finish" on page 7-102.

Stopped after step

The process has finished a stepping command. See "step" on page 7-99.

Stopped at breakpoint *number*

The process hit breakpoint number *number*. See "Breakpoints" on page 3-10.

Stopped at watchpoint *number*

The process stopped because of watchpoint *number.* See "Watchpoints" on page 3-11.

Stopped for watchpoint error

The process stopped because of an error during watchpoint processing. An error message in the debug message area in the Debug Window should explain the problem. See "Watchpoints" on page 3-11. See "Debug Message Area" on page 9-28.

Stopped by attach

The process has just been attached by the debugger. See "Attaching" on page 3-3.

Stopped by user

The process stopped because of a **`stop`** command. See "stop" on page 7-103.

Stopped for *exception-name*

The process stopped because of the Ada exception named *exception-name*. See "Exception Handling" on page 3-34.

Stopped for exec

The process has just `exec()`'ed a new program image. See “Programs and Processes” on page 3-2.

Stopped with *signal*

The process stopped with signal *signal*. See “Signals” on page 3-12.

Terminated with *signal*

The process terminated with signal *signal*. See “Exited and Terminated Processes” on page 3-16. This status appears only for core files. See “Core Files” on page 3-4.

Debug Source Display

The debug source display area lists the program code corresponding to the currently-selected frame in the currently-selected process. See “Current Frame” on page 3-25. See “Debug Group Area” on page 9-35. See “list” on page 7-58, for information on how the current source file is determined.

The text in this area includes the program listing along with line numbers and source decorations. See “Source Line Decorations” on page 7-63.

The text in this area changes if you use the debug source menu to list other functions or files.

You can change the height of this area by moving the sash up or down. See “Sashes” on page 9-11.

There are several special keys that may be used within this area. The function of most keys is independent of the position of the text cursor in this area. Some keys, like `b` and `h`, do depend on the position of the text cursor so that NightView can determine the source line of interest.

The text cursor (an "I-beam" cursor) can be moved to different locations within this area by using the arrow keys or by pointing to a source line and clicking mouse button 1.

`s`

This key is similar to using the **`step`** command with no argument. See “step” on page 7-99.

`S`

This key is similar to using the **`stepi`** command with no argument. See “stepi” on page 7-101.

`n`

This key is similar to using the **`next`** command with no argument. See “next” on page 7-100.

`N`

This key is similar to using the **nexti** command with no argument. See "nexti" on page 7-102.

r

This key is similar to using the **resume** command with no argument. See "resume" on page 7-98.

h

Run the process until it reaches the line the source window cursor is on. This key is identical to the Run to Here button. See "Debug Command Buttons" on page 9-32. It combines the actions of **breakpoint**, **enable/delete**, and **resume**.

f

This key is similar to using the **finish** command. See "finish" on page 7-102.

u

This key is similar to using the **up** command with no argument. See "up" on page 7-109.

d

This key is similar to using the **down** command with no argument. See "down" on page 7-109.

=

This key is similar to using the **frame 0** command. See "frame" on page 7-108.

>

This key is similar to using the **frame** command with no arguments. See "frame" on page 7-108.

e

This key is similar to selecting the Edit item in the Source menu. See "Debug Source Menu" on page 9-22.

p

This key performs the same action as the Print button in the debug command buttons area. See "Debug Command Buttons" on page 9-32).

b

This key performs the same action as the Breakpoint button (see "Debug Command Buttons" on page 9-32).

Debug Command Buttons

The debug command buttons let you control one or more processes by clicking with

mouse button 1. See “Understanding the Debug Window” on page 9-12. Some buttons may be disabled (dimmed) under certain circumstances.

If the Debug Window is in single process mode, button-activated commands apply only to the currently displayed process. See “Single Process Mode” on page 9-13. If the Debug Window is in group process mode, button-activated commands apply to each of the processes represented in the debug group area list. See “Group Process Mode” on page 9-14.

Resume

Clicking on this button is similar to using the **`resume`** command with no argument. See “resume” on page 7-98.

Step

Clicking on this button is similar to using the **`step`** command with no argument. See “step” on page 7-99.

Stepi

Clicking on this button is similar to using the **`stepi`** command with no argument. See “stepi” on page 7-101.

Next

Clicking on this button is similar to using the **`next`** command with no argument. See “next” on page 7-100.

Nexti

Clicking on this button is similar to using the **`nexti`** command with no argument. See “nexti” on page 7-102.

Finish

Clicking on this button is similar to using the **`finish`** command. See “finish” on page 7-102.

Stop

Clicking on this button is similar to using the **`stop`** command. See “stop” on page 7-103.

Print

Clicking on this button is similar to using the **`print`** command. See “print” on page 7-66. You must have selected an expression in the debug source display area before pressing this button. See “Debug Source Display” on page 9-31. When you press the button, the value of the selected expression is printed using the default format for the type of the expression.

Breakpoint

Clicking on this button is similar to using the **`breakpoint`** command with a line number for the location specifier. See “breakpoint” on page 7-79. You must have moved the text cursor in the debug source display area to the source line where you

want to set the breakpoint. See "Debug Source Display" on page 9-31. NightView uses this source line as the location specifier for the breakpoint. See "Location Specifiers" on page 7-9. When you press this button, a breakpoint is set. You see the source line decoration change and a message is displayed in the debug message area. See "Debug Message Area" on page 9-28. You can also set a breakpoint using the breakpoint dialog box, which provides you with more control and flexibility than the Breakpoint button. See "Debug Eventpoint Menu" on page 9-24.

Run to Here

Run the process until it reaches the line the source window cursor is on. This allows you to use the Run to Here button to quickly skip past chunks of code without single stepping through each line.

Clicking on this button combines the actions of three commands: First, it sets a **`breakpoint`** at the source window line where the text cursor is located. See "Debug Source Display" on page 9-31. Next, it runs **`enable/delete`** on that breakpoint (which will cause it to be deleted when it is hit). Finally, it **`resume`**s the process. See "breakpoint" on page 7-79. See "enable" on page 7-92. See "resume" on page 7-98.

When you press the button, you will see the source line decoration for the breakpoint appear and the message area will print a message about the new breakpoint. When the process finally stops at that breakpoint, the breakpoint will be deleted, and the decoration will disappear. See "Debug Message Area" on page 9-28.

Clear

Clicking on this button is similar to using the **`clear`** command with a line number for the location specifier. See "clear" on page 7-88. You must have moved the text cursor in the debug source display area to the source line where you want to clear eventpoints. See "Debug Source Display" on page 9-31. NightView uses this source line as the location specifier. See "Location Specifiers" on page 7-9. When you press this button, any eventpoints that are set at the first instruction of this line are removed. (If you have eventpoints set at instructions within the line, they will not be cleared.) You see the source line decoration change and a message is displayed in the Debug message area. See "Debug Message Area" on page 9-28.

Debug Interrupt Button

Clicking on this button interrupts whatever the debugger is doing. This is similar to using the shell interrupt character in the command-line interface. See "Interrupting the Debugger" on page 3-30.

Debug Qualifier Area

In single process mode, the debug qualifier area is a label that reminds you that commands entered in the debug command area are implicitly qualified by the currently displayed process in this Debug Window. See"Debug Command Area" on page 9-35. See "Single Process Mode" on page 9-13. The label shows the process's qualifier specifier.

See “Qualifier Specifiers” on page 7-10.

In group process mode, any commands that you enter are implicitly qualified by the qualifier associated with this Debug Window. See “Group Process Mode” on page 9-14. The qualifier label is replaced by an indicator that you are in Group Mode. To see the value of the qualifier, use the View menu item Show Qualifier.... See “Debug View Menu” on page 9-26.

Debug Command Area

The debug command area in the Debug Window is used to enter NightView commands. Like the dialogue command area in the Dialogue Window and the global command area in the Global Window, all the command-line interface commands, except for **shell**, can be entered in the debug command area.

Input to this area is similar to using the command-line interface. For example, you can enter an explicit qualifier followed by a command. If you do not specify a qualifier, the command is implicitly qualified by the currently displayed process (if you are in single process mode), or by the group of processes represented in this Debug Window (if you are in group process mode). See “Single Process Mode” on page 9-13. See “Group Process Mode” on page 9-14.

The debug command area is a combo box. See “Combo Boxes” on page 9-6.

Debug Group Area

The debug group area provides a list of all the processes that are represented in this Debug Window. Scroll bars appear if the list requires more space than the group area currently provides. The list is followed by the Switch To button, which allows you to switch the currently displayed process to a process that you have selected in the list. The list selection policy is *browse*, which means you can select only one list item at a time. See “List Selection Policies” on page 9-9.

Each item, or row, in the list contains the following information about one process: the qualifier specifier of each process, the executable file name, and an abbreviated status indicator. See “Qualifier Specifiers” on page 7-10. If the status information for a process changes, it is updated in the list.

To change the currently displayed process, select a list item, then press the Switch To button. Or, you can double-click on a list item to both select the item and switch to it. The program code for the currently selected process is represented in the source display area, and the identification area, status area and source file name area contain information about the currently selected process. See “Debug Source Display” on page 9-31. See “Debug Identification Area” on page 9-28. See “Debug Source File Name” on page 9-29.

The highlighted item in the group area list normally represents the currently displayed process in the source display area. This is true unless you select a different list item (process) and fail to switch to it.

You can cycle through the stopped processes in the group area list by using the button panel, labeled Switch To Stopped Process, located to the right of the list. If you click on Auto, NightView determines which process in the list is currently stopped and has been stopped the longest, highlights it in the list, and automatically switches the currently displayed process (in the source display area) to this process. (This is similar to the auto qualifier specifier. See "Qualifier Specifiers" on page 7-10.) The "up arrow" and "down arrow" buttons cause NightView to automatically select, and switch to, the next stopped process that is located up in the list, or down in the list, relative to the currently selected list item. Continuing to click on a directional arrow in this button panel after the top or bottom list item has been reached causes NightView to "wrap around" its search in the list for the next stopped process.

You can use the View menu item Display Group Area to display this area or to hide it from view. See "Debug View Menu" on page 9-26.

The size of the Debug Window does not change to accommodate the presence of the group area; rather, the source display area expands or shrinks depending on whether it needs to make room for the group area.

The group area can be resized within the Debug Window by adjusting the sash that separates it from the pane containing the source display. See "Sashes" on page 9-11.

Debug Dialog Boxes

This section describes dialog boxes that may appear while you are using the Debug Window. See "Dialogues and Dialog Boxes" on page 9-10.

Debug Group Selection Dialog Box

This dialog box pops up when you use the NightView menu to create a new Debug Window. See "Debug NightView Menu" on page 9-20. You can select qualifier specifiers and provide a name for the new window. See "Qualifier Specifiers" on page 7-10.

Select qualifier specifiers.

> Select one or more items to define the new Debug Window. If you choose a qualifier-specifier such as a dialogue name, the existing processes in that dialogue appear in the new Debug Window; future processes that start up in that dialogue do not necessarily appear in the new Debug Window unless their parent process is already there. See "Debug Window Behavior" on page 9-12.
>
> The default list selection policy is *extended*, which means you can select discontiguous ranges of items. See "List Selection Policies" on page 9-9. This list selection policy is configurable. See Appendix D [GUI Customization] on page D-1.

Debug Window name.

> By default, NightView uses the first selected item in the list for the window's name. Or, you can type in a name for the new Debug Window. Pressing Return activates the OK button.

This name appears in the window manager's title bar and as the window's icon label.

This is a text input area. See “Text Input Areas” on page 9-5.

Choose an action button.

When you are satisfied with your choices, click on OK. The dialog box is dismissed and a new Debug Window is created that contains the items you selected.

Clicking on Cancel cancels the action and closes this dialog box.

You can get help for this dialog box by clicking on Help.

Debug Source Selection Dialog Box

This dialog box pops up when you ask to list a function or Ada unit, or ask to list a source file from the Debug Window's Source menu. See “Debug Source Menu” on page 9-22. It allows you to change the program code that is listed in the Debug Window by selecting a function, Ada unit name or source file name from a list. You can interact with other NightView windows while this dialog box is displayed.

This dialog box is titled Select a Function/Unit or Select a Source File, depending on which menu item you selected, and displays the qualifier of the currently displayed process.

Enter search criteria.

Enter the regular expression (if you are searching for functions) or wildcard pattern (if you are searching for source files) you want to search for, then either press Return or click on Search. (For a regular expression, an anchored match is *not* implied.) See “Regular Expressions” on page 7-12. See “Wildcard Patterns” on page 7-14.

If you do not want to enter a regular expression or wildcard pattern, you can simply press Return or click on Search and all functions or files are displayed.

For Ada and C++, the regular expression is only applied to the final component of a name.

The next time you use this dialog box, this text is redisplayed.

Select a list item.

If NightView finds any functions or source files, their names are displayed in the list area. Scroll bars appear if the list requires more space than the dialog box currently provides. If no functions or files are found, a message is displayed in the debug message area. See “Debug Message Area” on page 9-28. The list uses the *browse* selection policy, which means that only one item can be selected at a time. See “List Selection Policies” on page 9-9.

Select an item in the list. If you double-click on an item in the list, the OK button is activated.

Choose an action button.

Click on OK to list that function, Ada unit name or source file in the source display area. See "Debug Source Display" on page 9-31. This button is disabled (dimmed) if the list is empty.

You can cancel the listing of the selected function or source file by clicking on Cancel.

You can get help for this dialog box by clicking on Help.

Debug File Selection Dialog Box

This dialog box pops up when you select List Any File... from the Debug Window's Source menu. It allows you to list a file of your choice in the Debug Window.

This is a file selection dialog box. See "File Selection Dialog Box" on page 9-7.

Select a file name.

Select the file you want to list. If you double-click on a file name in the Files list, the OK button is activated.

Choose an action button.

If you are satisfied with the file you selected, click on OK.

Clicking on Cancel cancels the action and closes this dialog box.

You can get help for this dialog box by clicking on Help.

Debug Eventpoint Dialog Boxes

NightView provides a dialog box for each type of eventpoint. See "Eventpoints" on page 3-8. These dialog boxes pop up when you use the Debug Window's Eventpoint menu to set or change an eventpoint. See "Debug Eventpoint Menu" on page 9-24. You decide how you want the eventpoint set or changed, then select the OK button and NightView will set or modify it for you.

All types of eventpoints share common traits; some eventpoints have additional optional or required information.

The eventpoint dialog boxes generally present the common eventpoint information first, followed by any data that is specific to a given eventpoint. The watchpoint dialog box first presents information specific to watchpoints, followed by the common eventpoint information.

For *inserted eventpoints*, NightView provides default settings for new eventpoints, including a default location specifier. See "Location Specifiers" on page 7-9. In addition, you can enter other information to define the eventpoint. Required data that must be provided by you before NightView can set the eventpoint is visually emphasized.

Depending on whether you are setting a new eventpoint, or changing an existing eventpoint, NightView allows or disallows access to certain fields in the eventpoint

dialog boxes.

Define the eventpoint.

Description (display only)

The title bar of each eventpoint dialog box indicates which kind of eventpoint the dialog box deals with and whether the dialog box allows you to set a new eventpont or to change an existing eventpoint.

Location

This field is displayed only for inserted eventpoint dialog boxes, not for watchpoint dialog boxes.

When the dialog box appears, the Location field contains a location specifier.

When setting a new eventpoint, NightView determines this value from the location of the text insertion cursor in the debug source display area. See "Debug Source Display" on page 9-31. You can edit this text input area. See "Text Input Areas" on page 9-5.

When changing an existing eventpoint, NightView displays the location specifier associated with this eventpoint. You cannot change this location.

Watchpoint options (watchpoint dialog box only)

These controls let you indicate whether you want to specify an L-value (e.g., a variable name) or an explicit program address and size. You can also control whether you want the watchpoint to be for memory reads, memory writes, or both.

When changing an existing watchpoint, these controls cannot be changed.

Watchpoint target (watchpoint dialog box only)

This text input area lets you enter an L-value or an explicit program address, depending on the setting of the controls in the watchpoint options area. See "Text Input Areas" on page 9-5.

When changing an existing watchpoint, this field cannot be changed.

Watchpoint size (watchpoint dialog box only)

This text input area lets you enter the size of the watchpoint target if you have selected Watch address and size in the watchpoint options area. See "Text Input Areas" on page 9-5. If you have not selected Watch address and size, then this area is not enabled.

When changing an existing watchpoint, this field cannot be changed.

Eventpoint Number (display only)

This labeled field is dimmed if NightView has not yet assigned a unique number to the eventpoint. See "Eventpoints" on page 3-8.

When changing an existing eventpoint, NightView displays the eventpoint number.

Enable Options

When setting a new eventpoint, you can choose from several enable options. By default, the eventpoint is created enabled. This is similar to using the **`enable`** or **`disable`** commands. See "enable" on page 7-92. See "disable" on page 7-91.

When changing an existing eventpoint, NightView displays the eventpoint's enabled state. You can select a different enable option by clicking on one of the choices. These options are dimmed if NightView cannot determine this state.

Enable

This is the default choice when setting a new eventpoint. The eventpoint is enabled.

Enable, disable after next hit

You can have the eventpoint be disabled automatically after the next hit.

For breakpoints, this is similar to using the **`tbreak`** command, or the **`enable/once`** command. See "tbreak" on page 7-93.

For patchpoints, this is similar to using the **`tpatch`** command, or the **`enable/once`** command. See "tpatch" on page 7-94.

For other eventpoint types, this is similar to using the **`enable/once`** command.

Enable, delete after next hit

Valid for breakpoints and watchpoints only. You can have the eventpoint be deleted automatically after the next hit. This is similar to using the **`enable/delete`** command.

Disable

You can disable the eventpoint.

Condition

You can attach a condition to this eventpoint, or change an existing condition, by editing this text input field. This is similar to using the **`condition`** command. See "condition" on page 7-89.

If you delete an existing condition, the eventpoint becomes unconditional.

Ignore Count

You can attach an ignore count to this eventpoint, or change an existing ignore count, by entering a number in this text input area. This is similar to using the **`ignore`** command. See "ignore" on page 7-93.

The default ignore count is zero and is represented by a blank field.

Name

When setting a new eventpoint, you can assign a name to it by entering text in this text input area. The name must consist only of alphanumeric characters and underscores and must begin with an alphabetic character. The name may be of arbitrary length. This is similar to using the **name** command. See "name" on page 7-78.

You cannot change an existing eventpoint's name using the dialog box. Use the **name** command to change eventpoint names.

Commands

Valid for breakpoints, monitorpoints, and watchpoints only; *required* to set monitorpoints. You can attach commands to this eventpoint, or change existing commands, by entering one command per line in this multi-line text input area. This is similar to using the **commands** command. See "commands" on page 7-89.

Evaluate Expression - Go To Location

Valid for patchpoints only; you are *required* to enter either an expression or a location specifier to set a patchpoint. Select one of the two choices by clicking on it. The radio button appears filled for your selection, and the label for the text input area changes to either Evaluate or Go to. See "Toggle Buttons" on page 9-12. Enter the expression or location specifier in the text input area.

Insert an expression at this location

This field represents the *eval* argument of one variant of the **patchpoint** command. See "patchpoint" on page 7-80. This is the default choice.

Branch to a different location

This field represents the *goto* argument of one variant of the **patchpoint** command.

Once set, this field cannot be changed.

Event ID

Valid for tracepoints only; *required* to set a tracepoint. This field represents the *event-id* argument of the **tracepoint** command. You must enter a trace-event number or symbolic name. See "tracepoint" on page 7-83.

Once set, this field cannot be changed.

Value

Valid for tracepoints only. This field represents the value= argument of the **tracepoint** command. You can enter an expression whose value should be logged with the trace event.

Once set, this field cannot be changed.

Choose an action button.

Click on OK to set or change the eventpoint. The dialog box is dismissed.

Click on Delete to delete an existing eventpoint. The dialog box is dismissed. This button is disabled (dimmed) if this is a new eventpoint.

Clicking on Cancel cancels the action and closes this dialog box.

You can get help for this dialog box by clicking on Help. The dialog box is not dismissed.

If you are setting a new eventpoint or deleting an existing one, you see the source line decoration change. NightView displays a message in the Debug message area to tell you if the eventpoint was set.

If you make an error while entering data, NightView may display an error dialog box and allow you to re-enter the data. See "Error Dialog Box" on page 9-16. Other warnings or errors associated with setting or changing this eventpoint are displayed in the debug message error. See "Debug Message Area" on page 9-28.

You can use the **`info eventpoint`** command to check the eventpoint settings. See "info eventpoint" on page 7-115.

Debug Eventpoint Summarize/Change Dialog Box

This dialog box pops up when you use the Debug Window's Eventpoint menu to select the Summarize/Change… item. See "Debug Eventpoint Menu" on page 9-24. If the Debug Window is in single process mode, it shows you a summary of existing eventpoints for the process. If the Debug Window is in group process mode, the Qualifier changes to [Group Mode] and the list of eventpoints includes all the processes in the qualifier. This dialog box also provides several ways for you to change eventpoints. See "Single Process Mode" on page 9-13. See "Group Process Mode" on page 9-14.

See "Eventpoints" on page 3-8.

Specify eventpoints to appear in the list.

NightView displays a list of eventpoints according to the selections you make in this section of the dialog box.

By default, NightView displays all eventpoints that occur for the qualifier. If there is only one eventpoint, NightView selects it for you in the list.

Pressing Return while the focus is in one of the text input areas causes the default Update List button to be activated. See "Text Input Areas" on page 9-5.

Choose eventpoints.

You can choose any combination of eventpoint types to display in the list by clicking on the check button (or its label) for each eventpoint type you are interested in. See "Toggle Buttons" on page 9-12. Two buttons are also avail-

able to help you check all of the eventpoint types (Check All) or clear all of the checked eventpoint types (Clear All).

Choose location.

By default, the location field is blank. NightView shows you all eventpoints for the qualifier regardless of their locations (taking into consideration your other list specifications).

If you want to see a list of eventpoints found at a given location, type a location specifier into this field. See "Location Specifiers" on page 7-9. Watchpoints are not associated with a location, so no watchpoints will match if you enter a location specifier.

If you want NightView to fill in the location field with a location specifier that corresponds to the location of the text insertion cursor in the source display area, press the Update button next to the location field. See "Debug Source Display" on page 9-31.

The location field is a text input area.

Choose eventpoint name.

If you want to see a list of eventpoints that have a certain name, enter the name in this field.

If the name field is blank, NightView shows you all eventpoints in the process regardless of any name (taking into consideration your other list specifications).

The name field is a text input area.

Qualifier.

The qualifier is displayed to remind you that this list of eventpoints applies to the process or processes represented by this qualifier. See "Qualifier Specifiers" on page 7-10.

Update the list.

The Update List button updates the list of eventpoints and the qualifier that represents them. Press this button whenever you want to see the current list of eventpoints and their status for the Debug Window's current qualifier.

The list of eventpoints is automatically updated when you change an eventpoint by using this dialog box. The list is not updated if you create new eventpoints or type in commands to change eventpoint characteristics; use the Update List button if you are unsure of the current state of eventpoints.

Select eventpoints from the list to change.

The eventpoint list displays eventpoint ID numbers, tells you what type of eventpoint it is, its enabled state, process and address. Scroll bars appear for the list, if necessary. Messages related to the list are displayed below the list.

If NightView cannot determine a piece of eventpoint information, that part of the list will be empty. For example, this situation may occur if the process is running when NightView tries to determine the enabled state of an eventpoint.

If there is only one item in the list, NightView selects it for you. Otherwise, you must select the items in the list you are interested in and then choose an action area button to perform the requested action on each selected eventpoint.

The default list selection policy is Extended, which means you can select discontiguous ranges of items. This list selection policy is configurable. See Appendix D [GUI Customization] on page D-1. See "List Selection Policies" on page 9-9.

Choose an action button.

The buttons in the action area allow you to make changes to *selected* eventpoints, dismiss the dialog box, and request help.

Some buttons may be disabled (dimmed) under certain circumstances. For example, if the list is empty, it does not make sense to use some of the buttons.

The eventpoint summary dialog box is dismissed only if you choose the Close button.

Change...

Click on Change... to see additional information about an eventpoint and to change eventpoint characteristics. An eventpoint dialog box is displayed for each eventpoint you selected in the list. See "Debug Eventpoint Dialog Boxes" on page 9-38.

Enable

Click on Enable to enable the selected eventpoints. This is similar to using the **`enable`** command. See "enable" on page 7-92.

Disable

Click on Disable to disable the selected eventpoints. This is similar to using the **`disable`** command. See "disable" on page 7-91.

Delete

Click on Delete to delete selected eventpoints.

Depending on your safety level, NightView may display a warning dialog box to make sure you want to delete the eventpoints. See "set-safety" on page 7-49. Once deleted, you cannot refer to these eventpoints again. This is similar to using the **`delete`** command. See "delete" on page 7-90.

If you think you may want to "turn off" an eventpoint temporarily, then use it again later, you should disable the eventpoint and enable it when you are ready to use it.

Close

Clicking on Close cancels any action and closes this dialog box. This button is never disabled (dimmed).

Help

You can get help for this dialog box by clicking on Help. This button is never disabled (dimmed).

Warnings or errors associated with using this dialog box or changing eventpoints are displayed in dialog boxes or in the debug message area. See “Warning and Error Dialog Boxes” on page 9-15.. See “Debug Message Area” on page 9-28.

You can use the **`info eventpoint`** command to check eventpoint settings. See “info eventpoint” on page 7-115.

Remote Login Dialog Box

This dialog box pops up when you use the NightView menu's Start Remote Dialogue... item. See “Debug NightView Menu” on page 9-20. This dialog box allows you to specify the parameters for creating a remote NightView session. See “Remote Dialogues” on page 3-6. Some of these parameters are required, but most are optional.

The parameters specified in this dialog apply to the NightView processes that execute on the remote system. These processes include a NightView target program, a dialogue shell, and (unless you specify otherwise using the **`run(1)`** shell command) all the processes started by that dialogue shell.

Remote host information

Remote host

This is the name or address of the remote system on which you want a remote dialogue. This field is required information.

Login name

This specifies the user name to use to log into the remote system. This field is required, but it defaults to the user running NightView.

Password

This specifies the password for the user name specified in the Login name field. For security, the password you type is not echoed in the window; instead, an asterisk (`*`) replaces each character. You may leave this field empty if the specified user name does not have a password on the designated remote system.

Name for new Dialogue

This field specifies the name to give to the dialogue. See “Qualifier Specifiers” on page 7-10. If you leave this field empty, the name of the dialogue will default to be the same as the Remote host field. If the remote system name is not a valid dialogue name, an error dialog will appear. See “Warning and Error Dialog Boxes” on page 9-15. A common reason for the remote system to be an invalid dialogue name is that the remote system name contains period (`.`) characters (e.g., it includes domain names), or it is an IP address instead of a name.

Scheduling information

Priority

This field specifies the priority you want applied to the NightView processes running on the remote system. You will usually want to leave this empty, to select the default value. However, if your application contains continuously-running processes that run at real-time priorities, you may need to set the priority of NightView or it will not get sufficient CPU time to perform its debugging chores. We suggest that you set this only if the target system has little or no spare CPU resources *and* you notice a lack of responsiveness in NightView.

Valid values for the priority depend on the scheduling class you select. See the **`run(1)`** command for valid values.

Note that you may need special privileges on the remote system to be able to specify a priority explicitly.

Nice Value

This is an alternative way to adjust the priority of the remote NightView processes. If you specify both priority and nice value, the priority takes precedence. Nice values only apply to the Time Sharing scheduling class.

Scheduling Class

This option menu selects the scheduling policy for the remote NightView processes. You will usually want to leave this at its default selection. However, if you need greater control over how much CPU resources the remote NightView processes get, you may need to select a different scheduling class and priority.

Time Quantum

This field is enabled only if you select the Round Robin scheduling class. See the **`run(1)`** command man page for more information about time quantum. You may use the units option menu to the right of this field to specify the time units to apply to the quantum value.

CPU and memory binding information

Binding Type

This option menu selects the kind of CPU binding to apply to the remote NightView processes. You may wish to use this if you want to isolate the NightView processes to a particular CPU or set of CPUs.

If you select the Bias option, the CPU toggle buttons will be enabled and allow you to select any set of CPUs. If you select Exclusive, the CPU toggle buttons are enabled but you are restricted to selecting exactly one CPU. If you select any other choice in the Binding Type menu, the CPU toggle buttons are disabled.

CPU

These toggle buttons allow you to select the CPUs on which the remote NightView processes can execute. They are enabled only for the Bias or Exclusive binding type options. When these toggle buttons are enabled, the OK button is disabled until you select at least one CPU.

NUMA

These option menus allow you to select the memory binding parameters for the remote NightView processes. You may need to use these to keep NightView from interfering with your application's use of certain memory pools. See the **run(1)** command and the **memory(7)** man pages for more information about NUMA policies.

The Default option menu selects the overall memory binding policy. This policy applies to all pages unless overridden by one of the other more-specific NUMA option menus. The Text option menu selects the NUMA policy to apply to text (code) pages, the Private option menu selects the NUMA policy to apply to private data pages, and so forth for the Shared and UBlock option menus.

All of these menus contain the Global, Soft Local, and Hard Local options. The Global option specifies that the designated pages should be placed in global memory. The Soft Local option specifies that the designated pages be placed in local memory if space is available, otherwise they should be placed in global memory. The Hard Local option specifies that the designated pages *must* be placed in local memory.

For the Default option menu, selecting System Default specifies that the NUMA policy will be inherited by the parent process that starts the remote dialogue processes. For the Text, Private, Shared, and UBlock menus, choosing Default specifies that whatever policy was selected by the Default option menu applies to that class of memory pages.

For example, if you select Default/Soft Local, Text/Default, Private/Hard Local, Shared/Global, and UBlock/Default, then text and ublock pages will be placed in local memory if possible (soft local policy, specified by the Default/Soft Local selection), while private data will be forced to local memory and shared data will be forced to global memory.

Action Buttons

OK

The OK button is enabled if you have specified all the required information. Required information is the remote host name and login name and, if you selected a binding type of Bias or Exclusive, at least one CPU must be selected.

When you press the OK button, the remote dialogue is created and the remote login dialog is dismissed. If the remote dialogue cannot be created, either an error dialog box will appear or the remote login dialog disappears and a message is displayed in the message area. See “Warning and Error Dialog Boxes” on page 9-15. See “Debug Message Area” on page 9-28.

Cancel

Pressing the Cancel button dismisses the dialog box without creating a remote dialogue.

Help

Pressing the Help button brings up the online help with information about the remote login dialog.

Monitor Window - GUI

The Monitor Window is created when you use monitorpoints while running NightView with the graphical user interface. See "Monitor Window" on page 3-27.

In the GUI, the Monitor Window uses a scrolling area to display monitored values, so there is essentially no limit to the number of items you can have in the active display. To remain compatible with the simple full-screen interface, it uses the same item layout algorithm and assumes a column width for the window to determine how many items to put on one line. See "Monitor Window - Simple Full-Screen" on page 8-2. The default value for this column width is 80, but you can set the `monitorWindowColumns` resource to any other appropriate value (a common alternative might be 132). See Appendix D [GUI Customization] on page D-1. Dynamically resizing the window to be wider does not cause NightView to put more items on one line.

The stale data indicators in the graphical display take the form of icons. A blank icon indicates updated values, a triangular warning symbol indicates not executed values, and a triangular warning symbol containing an exclamation point indicates executed but not sampled values. For more information about stale data indicators, see "Monitor Window" on page 3-27.

A label at the top of the window indicates the current held or running status and shows the current delay time in milliseconds between samples. A legend shows a brief description of the stale data icons.

Global Window

The Global Window provides global interaction and gives you control over dialogues. There is only one instance of a Global Window for an invocation of NightView.

The Global Window is normally hidden and appears only when you ask to see it or when no Dialogue Windows exist. You can display the Global Window by choosing the Open Global Window menu item found in the NightView menu of both the Debug and Dialogue Windows. See "Debug NightView Menu" on page 9-20. See "Dialogue NightView Menu" on page 9-16.

The following sections describe the parts of the Global Window.

Global Menu Bar

The menu bar in the Global Window allows you to perform global NightView actions and access the online help system.

Global NightView Menu

Mnemonic: N

The NightView menu is used to control NightView windows and perform global NightView actions. The NightView menu appears in the Debug, Dialogue and Global windows and has the same menu items in each window.

See "Debug NightView Menu" on page 9-20, for a description of the individual NightView menu items.

Global Help Menu

Mnemonic: H

This menu provides ways of getting context-sensitive help, help on the current window, help on the last error NightView encountered, as well as several other categories of help. NightView help information is displayed in a Help Window. See "Help Window" on page 9-50.

The Help menu is described in another section. See "Help Menu" on page 9-3.

A general discussion of NightView's online help is also available. See "GUI Online Help" on page 9-2.

Global Output Area

The output area in the Global Window is similar to the output from the command-line interface. It shows a combination of the output and messages displayed in the Debug Window and the Dialogue Window as well as the output and error messages from commands that are processed by this Global Window.

In contrast, the message area in the Dialogue Window shows only messages and program output associated with that dialogue, and the message area in the Debug Window shows only messages associated with processes represented in that window. See "Dialogue Message Area" on page 9-17, and "Debug Message Area" on page 9-28.

This is a scrolling area. You can use the scroll bar to look at older or newer messages.

Global Interrupt Button

Clicking on this button interrupts whatever the debugger is doing. This is similar to using

the shell interrupt character in the command-line interface. See "Interrupting the Debugger" on page 3-30.

Global Qualifier Area

The qualifier area in the Global Window shows the current default qualifier for the global interactive command stream, which you can access through the global command area (see "Global Command Area" on page 9-50). You can set the default qualifier using the **`set-qualifier`** command. See "set-qualifier" on page 7-46.

Global Command Area

The global command area in the Global Window is used to enter NightView commands. Like the debug command area in the Debug Window and the dialogue command area in the Dialogue Window, all the command-line interface commands, except for **`shell`**, can be entered in the global command area.

Input to this area is similar to using the command-line interface. For example, you can enter an explicit qualifier followed by a command.

Commands entered in this area are implicitly qualified by the default qualifier. You can change the default qualifier by using the **`set-qualifier`** command. See "Global Qualifier Area" on page 9-50.

The global command area is a combo box. See "Combo Boxes" on page 9-6.

Help Window

NightView displays online help in the Help Window. The Help Window allows you to display any section of the *NightView User's Guide* and provides different methods to allow you to navigate from one section to another.

NightView uses HyperHelp™ to display help. To learn about HyperHelp, click on the Help menu of the Help Window and select Help On HyperHelp.

For a general discussion of NightView's online help, see "GUI Online Help" on page 9-2.

A
System Resource Requirements

This appendix describes system resources used by NightView. System administrators may want to modify the "System Tuning Parameters" so that their users can use NightView effectively. See *System Administration Volume 1*.

This discussion refers to the *local* system and the *remote* system. The local system is the system where NightView is invoked. The remote system is the system where the application program is running. In many cases, these are the same system, but they are distinguished here so that special purpose applications can be dealt with appropriately. Many system administrators will simply want to make all their systems be able to be both local and remote systems.

System components

> If you are using the remote dialogue feature (see “Remote Dialogues” on page 3-6), you must have networking installed on both the local and remote systems. You must also have **`telnetd`** running on the remote system, or you must arrange for **`inetd`** to run it. See the man pages for these facilities for more information.

shared memory regions

> NightView uses a variety of shared memory regions on both the local and the remote system. Each shared memory region contributes to the total number of regions and the total number of shared memory clicks on the system. Most of the shared memory regions also contribute to the number of shared memory identifiers on the system as long as the debugger is running.
>
> Therefore, in order to use NightView, both the local and remote systems must be configured with shared memory enabled. The maximum number of shared memory identifiers and the maximum number of shared memory clicks system wide may need to be increased.
>
> IPC
>
> > Make sure the ipc module is configured (`/etc/conf/sdevice.d/ipc`).
>
> `SHMMNI`
>
> > Check the "maximum number of shared memory identifiers" system tunable using the **`idtune(1M)`** utility.
>
> The following information about the particular memory regions used by NightView is supplied only to aid in fine-tuning of the memory parameters.
>
> Regions on the local system:
>
> > Communications among processes which make up the debugger.
> >
> > > One shared memory region per invocation of NightView.

Regions on the remote system:

Debug agent

One shared memory region for each process using a debug agent. See "Debug Agent" on page 3-17. The shared memory identifier for this region exists as long as the process is running.

Monitorpoints

One shared memory region per invocation of NightView on each remote system that is using monitorpoints. See "Monitorpoints" on page 3-10.

processes

Each invocation of NightView uses at least one process on the local system. The remote system uses two processes per dialogue, not including the processes being debugged.

The maximum number of processes on the system (`NPROC` tunable) and the maximum number of processes per user (`MAXUP` tunable) may need to be increased for the local and remote systems.

ptys

NightView uses one pty per dialogue on the remote system.

For the graphical user interface, X server memory may also be a concern. See Appendix D [GUI Customization] on page D-1.

B
Summary of Commands

This section gives a summary of all the commands in NightView. The table is organized alphabetically by command. The abbreviations for the commands are included with the corresponding commands, rather than alphabetically.

Also, remember that you can abbreviate commands by using a unique prefix.

!

Pass input to a dialogue. See "!" on page 7-27 for more information.

agentpoint

Insert a call to a debug agent at a given location. See "agentpoint" on page 7-87 for more information.

apply on dialogue

Execute **on dialogue** commands for existing dialogues. See "apply on dialogue" on page 7-25 for more information.

apply on program

Execute **on program** commands for existing processes. See "apply on program" on page 7-38 for more information.

attach

Attach the debugger to a process that is already running. See "attach" on page 7-32 for more information.

backtrace
bt

Print an ordered list of the currently active stack frames. See "backtrace" on page 7-65 for more information.

breakpoint
b

Set a breakpoint. See "breakpoint" on page 7-79 for more information.

cd

Set the debugger's default working directory. See "cd" on page 7-56 for more information.

checkpoint

Take a restart checkpoint now. See "checkpoint" on page 7-39 for more information.

clear

Clear all eventpoints at a given location. See "clear" on page 7-88 for more information.

commands

Attach commands to a breakpoint or monitorpoint. See "commands" on page 7-89 for more information.

condition

Attach a condition to an eventpoint. See "condition" on page 7-89 for more information.

continue
c

Continue execution and wait for something to happen. See "continue" on page 7-97 for more information.

core-file

Create a pseudo-process for debugging an aborted program's core image file. See "core-file" on page 7-34 for more information.

debug

Specify names for programs you wish to debug. See "debug" on page 7-20 for more information.

define

Define a NightView macro. See "define" on page 7-134 for more information.

delay

Delay NightView command execution for a specified time. See "delay" on page 7-113 for more information.

delete
d

Delete an eventpoint. See "delete" on page 7-90 for more information.

detach

Stop debugging a list of processes. See "detach" on page 7-32 for more information.

directory

Set the directory search path. See "directory" on page 7-60 for more information.

disable

Disable an eventpoint. See "disable" on page 7-91 for more information.

`display`

Add to the list of expressions to be printed each time the process stops. See “display” on page 7-72 for more information.

`down`

Move one or more stack frames toward frames called by the current stack frame. See “down” on page 7-109 for more information.

`echo`

Print arbitrary text. See “echo” on page 7-71 for more information.

`enable`

Enable an eventpoint for a specified duration. See “enable” on page 7-92 for more information.

`exec-file`

Specify the location of the executable file corresponding to a process. See “exec-file” on page 7-35 for more information.

`family`

Give a name to a family of one or more processes. See “family” on page 7-40 for more information.

`finish`

Continue execution until the current function finishes. See “finish” on page 7-102 for more information.

`forward-search`
`fo`

Search forward through the current source file for a specified regular expression. See “forward-search” on page 7-61 for more information.

`frame`
`f`

Select a new stack frame or print a description of the current stack frame. See “frame” on page 7-108 for more information.

`handle`

Specify how to handle signals and Ada exceptions in the user process. See “handle” on page 7-105 for more information.

`help`

Access the online help system. See “help” on page 7-111 for more information.

ignore

Attach an ignore-count to an eventpoint. See "ignore" on page 7-93 for more information.

info address

Determine the location of a variable. See "info address" on page 7-131 for more information.

info agentpoint

Describe current state of agentpoints. See "info agentpoint" on page 7-120 for more information.

info args

Print description of current routine arguments. See "info args" on page 7-130 for more information.

info breakpoint
i b

Describe current state of breakpoints. See "info breakpoint" on page 7-116 for more information.

info convenience

Describe convenience variables. See "info convenience" on page 7-123 for more information.

info declaration
ptype

Print the declaration of variables or types. See "info declaration" on page 7-133 for more information.

info dialogue

Print information about active dialogues. See "info dialogue" on page 7-126 for more information.

info directories

Print the search path used to locate source files. See "info directories" on page 7-123 for more information.

info display

Describe expressions that are automatically displayed. See "info display" on page 7-123 for more information.

info eventpoint

Describe current state of breakpoints, tracepoints, patchpoints, monitorpoints, agentpoints, and watchpoints. See "info eventpoint" on page 7-115 for more information.

info exception
exception

Print information about Ada exception handling. See “info exception” on page 7-129 for more information.

info family

Print information about an existing process family. See “info family” on page 7-127 for more information.

info files

Print the names of the executable, symbol table and core files. See “info files” on page 7-133 for more information.

info frame

Describe a stack frame. See “info frame” on page 7-122 for more information.

info functions

List names of functions, subroutines, or Ada unit names. See “info functions” on page 7-131 for more information.

info history

Print value history information. See “info history” on page 7-124 for more information.

info limits

Print information about limits on expression and location output. See “info limits” on page 7-124 for more information.

info line

Describe location of a source line. See “info line” on page 7-133 for more information.

info locals

Print information about local variables. See “info locals” on page 7-130 for more information.

info log

Describe any open log files. See “info log” on page 7-115 for more information.

info macros

Print a description of one or more NightView macros. See “info macros” on page 7-139 for more information.

info memory

Print information about the virtual address space. See “info memory” on page 7-126 for more information.

`info monitorpoint`

Describe current state of monitorpoints. See "info monitorpoint" on page 7-119 for more information.

`info name`

Print information about an existing eventpoint-name. See "info name" on page 7-127 for more information.

`info on dialogue`

Print **`on dialogue`** commands. See "info on dialogue" on page 7-128 for more information.

`info on program`

Print **`on program`** commands. See "info on program" on page 7-128 for more information.

`info on restart`

Print **`on restart`** commands. See "info on restart" on page 7-128 for more information.

`info patchpoint`

Describe current state of patchpoints. See "info patchpoint" on page 7-118 for more information.

`info process`

Describe processes being debugged. See "info process" on page 7-125 for more information.

`info registers`

Print information about registers. See "info registers" on page 7-124 for more information.

`info representation`
`representation`

Describe the storage representation of an expression. See "info representation" on page 7-132 for more information.

`info signal`

Print information about signals. See "info signal" on page 7-125 for more information.

`info sources`

List names of source files. See "info sources" on page 7-131 for more information.

`info tracepoint`

Describe current state of tracepoints. See “info tracepoint” on page 7-117 for more information.

info types

Print type definition information. See “info types” on page 7-132 for more information.

info variables

Print global variable information. See “info variables” on page 7-130 for more information.

info watchpoint

Describe current state of watchpoints. See “info watchpoint” on page 7-121 for more information.

info whatis
whatis

Describe the result type of an expression visible in the current context. See “info whatis” on page 7-132 for more information.

interest

Control which subprograms are interesting. See “interest” on page 7-51 for more information.

jump

Continue execution at a specific location. See “jump” on page 7-103 for more information.

kill

Terminate a list of processes. See “kill” on page 7-33 for more information.

list
l

List a source file. See “list” on page 7-58 for more information.

load

Dynamically load an object file, possibly replacing existing routines. See “load” on page 7-75 for more information.

login

Login to a new dialogue shell. See “login” on page 7-18 for more information.

logout

Terminate a dialogue. See “logout” on page 7-23 for more information.

mcontrol
hold

release

Control the monitor display window. See "mcontrol" on page 7-86 for more information.

monitorpoint

Monitor the values of one or more expressions at a given location. See "monitorpoint" on page 7-84 for more information.

mreserve

Reserve a region of memory in a process. See "mreserve" on page 7-43 for more information.

name

Give a name to a group of eventpoints. See "name" on page 7-78 for more information.

next
n

Execute one line, stepping over procedures. See "next" on page 7-100 for more information.

nexti
ni

Execute one instruction, stepping over procedures. See "nexti" on page 7-102 for more information.

nodebug

Specify names for programs you do not wish to debug. See "nodebug" on page 7-20 for more information.

notify

Ask about pending event notifications. See "notify" on page 7-31 for more information.

on dialogue

Specify debugger commands to be executed when a dialogue is created. See "on dialogue" on page 7-24 for more information.

on program

Specify debugger commands to be executed when a program is `exec`ed. See "on program" on page 7-36 for more information.

on restart

Specify debugger commands to be executed when a program is restarted. See "on restart" on page 7-38 for more information.

output

Print the value of a language expression with minimal output. See “output” on page 7-71 for more information.

patchpoint

Install a small patch to a routine. See “patchpoint” on page 7-80 for more information.

print
p

Print the value of a language expression. See “print” on page 7-66 for more information.

printf

Print the values of language expressions using a format string. See “printf” on page 7-74 for more information.

pwd

Print NightView’s current working directory. See “pwd” on page 7-56 for more information.

quit
q

Stop everything. Exit the debugger. See “quit” on page 7-17 for more information.

redisplay

Enable a display item. See “redisplay” on page 7-74 for more information.

refresh

Refresh the terminal screen. See “refresh” on page 7-112 for more information.

resume

Continue execution. See “resume” on page 7-98 for more information.

reverse-search

Search backwards through the current source file for a specified regular expression. See “reverse-search” on page 7-61 for more information.

run

Run a program in a dialogue and wait for NightView to start debugging it. See “run” on page 7-30 for more information.

select-context

Select the context of an Ada task, a thread, or of a Lightweight Process (LWP). See “select-context” on page 7-110 for more information.

`set`

Evaluate a language expression without printing its value. See "set" on page 7-67 for more information.

`set-auto-frame`

Control the positioning of the stack when a process stops. See "set-auto-frame" on page 7-54 for more information.

`set-children`

Control whether children should be debugged. See "set-children" on page 7-41 for more information.

`set-editor`

Set the mode for editing commands in the simple full-screen interface. See "set-editor" on page 7-55 for more information.

`set-exit`

Control whether a process stops before exiting. See "set-exit" on page 7-42 for more information.

`set-history`

Specify the number of items to be kept in the value history list. See "set-history" on page 7-46 for more information.

`set-language`

Establish a default language context for variables and expressions. See "set-language" on page 7-44 for more information.

`set-limits`

Specify limits on the number of array elements, string characters, or program addresses printed when examining program data. See "set-limits" on page 7-46 for more information.

`set-local`

Define process local convenience variables. See "set-local" on page 7-50 for more information.

`set-log`

Log session to file. See "set-log" on page 7-44 for more information.

`set-notify`

Control how you are notified of events. See "set-notify" on page 7-30 for more information.

`set-overload`

Control how NightView treats overloaded operators and routines in expressions. See “set-overload” on page 7-54 for more information.

`set-patch-area-size`

Control the size of patch areas created in your process. See “set-patch-area-size” on page 7-50 for more information.

`set-prompt`

Set the string used to prompt for command input. See “set-prompt” on page 7-47 for more information.

`set-qualifier`

Specify the default list of processes or dialogues that will be affected by subsequent commands which accept qualifiers. See “set-qualifier” on page 7-46 for more information.

`set-restart`

Control whether restart information is applied. See “set-restart” on page 7-49 for more information.

`set-safety`

Control debugger response to dangerous commands. See “set-safety” on page 7-49 for more information.

`set-search`

Control case sensitivity of regular expressions in NightView. See “set-search” on page 7-54 for more information.

`set-show`

Control where dialogue output goes. See “set-show” on page 7-28 for more information.

`set-terminator`

Set the string used to recognize end of dialogue input mode. See “set-terminator” on page 7-48 for more information.

`set-trace`

Establish tracing parameters. See “set-trace” on page 7-82 for more information.

`shell`

Run an arbitrary shell command. See “shell” on page 7-112 for more information.

`show`

Control dialogue output. See “show” on page 7-29 for more information.

`signal`

Continue execution with a signal. See "signal" on page 7-104 for more information.

source

Input commands from a source file. See "source" on page 7-113 for more information.

step
s

Execute one line, stepping into procedures. See "step" on page 7-99 for more information.

stepi
si

Execute one instruction, stepping into procedures. See "stepi" on page 7-101 for more information.

stop

Stop a process. See "stop" on page 7-103 for more information.

symbol-file

Establish the file containing symbolic information for a program. See "symbol-file" on page 7-33 for more information.

tbreak

Set a temporary breakpoint. See "tbreak" on page 7-93 for more information.

tpatch

Set a patchpoint that will execute only once. See "tpatch" on page 7-94 for more information.

tracepoint

Set a tracepoint. See "tracepoint" on page 7-83 for more information.

translate-object-file
xl

Translate object filenames for a remote dialogue. See "translate-object-file" on page 7-21 for more information.

undisplay

Disable an item from the display expression list. See "undisplay" on page 7-73 for more information.

up

Move one or more stack frames toward the caller of the current stack frame. See "up" on page 7-109 for more information.

vector-set

Set the value of a vector. See “vector-set” on page 7-76 for more information.

`watchpoint`

Set a watchpoint. See “watchpoint” on page 7-95 for more information.

`x`

Print the contents of memory beginning at a given address. See “x” on page 7-68 for more information.

C
Quick Reference Guide

Invoking NightView

nview [-editor *program***] [-help] [-ktalk] [-nogui]**
[-noktalk] [-nolocal] [-nx] [-prompt *string***]**
[-safety *safe-mode***] [-simplescreen] [-version]**
[-_Xoption_ **...] [-x** *command-file***] [-xeditor]**
[_program-name_ **[**_corefile-name_**]]**

Controlling the Debugger

Quitting NightView

quit

Abbreviation: **q**

Managing Dialogues

login [/conditional] [/popup] [name=*dialogue name*] [user=*login name*] [*others* ...] *machine*

debug *pattern* ...

nodebug *pattern* ...

translate-object-file [*from* [*to*]]

Abbreviation: **xl**

logout

on dialogue [*regexp*]

on dialogue *regexp command*

on dialogue *regexp* **do**

apply on dialogue

Dialogue Input and Output

! [*input line*]

set-show [silent | notify=*mode* | continuous=*mode*] [log[=*filename*]] [buffer=*number*]

show [*number* | all | none] [| *shell-command*]

Managing Processes

run *input line*

set-notify [silent | continuous=*mode*]

notify

attach *pid*

detach

kill

symbol-file *program-name*

core-file *corefile-name* [exec-file=*program-name*]

exec-file *program-name*

on program [*pattern*]

on program *pattern command*

on program *pattern* **do**

apply on program

on restart [*pattern*]

on restart *pattern command*

on restart *pattern* **do**

checkpoint

family *family-name* [[-] *qualifier-spec*] ...

set-children { all [resume] | exec | none }

set-exit [stop | nostop]

mreserve start=*address* {length=*bytes* | end=*address*}

Setting Modes

set-log *keyword filename*

set-language {ada | auto | c | c++ | fortran}

set-qualifier [*qualifier-spec* ...]

set-history *count*

set-limits {array=*number* | string=*number* | addresses=*number*} ...

set-prompt *string*

set-terminator *string*

set-safety [forbid | verify | unsafe]

set-restart [always | never | verify]

set-local *identifier* ...

set-patch-area-size {data=*data-size* | eventpoint=*eventpoint-size* |
monitor=*monitor-size* | text=*text-size*} ...

interest [*level*] [[at] [*location-spec*]]

interest inline[=*level*]

interest justlines[=*level*]

interest nodebug[=*level*]

interest threshold[=*level*]

set-auto-frame *args*...

set-overload [operator={on | off}] [routine={on | off}]

set-search [sensitive | insensitive]

set-editor *mode*

Debugger Environment Control

cd *dirname*

pwd

Source Files

Viewing Source Files

list *where-spec*

list *where-spec1*, *where-spec2*

list ,*where-spec*

list *where-spec*,

list +

list -

list =

list

Abbreviation: **l**

directory [*dirname* ...]

Searching

forward-search [*regexp*]

Abbreviation: **fo**

reverse-search [*regexp*]

Examining and Modifying

backtrace [*number-of-frames*]

Abbreviation: **bt**

print [/*print-format-letter*] *expression*

Abbreviation: **p**

set *expression*

x [/[*repeat-count*][*size-letter*][*x-format-letter*]] [*addr-expression*]

output [/*print-format-letter*] *expression*

echo *text*

display [[/*print-format-letter*] *expression*]

display /[*repeat-count*][*size-letter*][*x-format-letter*] *addr-expression*

undisplay *item-number* ...

redisplay *item-number* ...

printf *format-string*[, *expression* ...]

load *object*

vector-set *l-value* = *component*, *component...*

vector-set *l-value* = *repeat-count*,*component*

Manipulating Eventpoints

name [/add] *name* [[-] *eventpoint-spec*] ...

breakpoint [*eventpoint-modifier*] [name=*breakpoint-name*] [[at] *location-spec*] [if *conditional-expression*]

Abbreviation: **b**

patchpoint [*eventpoint-modifier*] [name=*patchpoint-name*] [[at] *location-spec*] eval *expression*

patchpoint [*eventpoint-modifier*] [name=*patchpoint-name*] [[at] *location-spec*] goto *location-spec*

set-trace [eventmap=*event-map-file*]

tracepoint [*eventpoint-modifier*] *event-id* [name=*tracepoint-name*] [[at] *location-spec*] [value=*logged-expression*] [if *conditional-expression*]

monitorpoint [*eventpoint-modifier*] [name=*monitorpoint-name*] [[at] *location-spec*]

mcontrol {display | nodisplay} [*monitorpoint-spec* ...]

mcontrol delay *milliseconds*

mcontrol {off | on | stale | nostale | hold | release}

Abbreviation: **hold**

Abbreviation: **release**

agentpoint [*eventpoint-modifier*] [name=*agentpoint-name*] [[at] *location-spec*]

clear [[at] *location-spec*]

commands *eventpoint-spec*

condition *eventpoint-spec* [*conditional-expression*]

delete [*eventpoint-spec* ...]

Abbreviation: **d**

disable [*eventpoint-spec* ...]

enable [/once|/delete] *eventpoint-spec* ...

ignore *eventpoint-spec count*

tbreak [name=*breakpoint-name*] [[at] *location-spec*] [if *conditional-expression*]

tpatch [name=*patchpoint-name*] [[at] *location-spec*] eval *expression*

tpatch [name=*patchpoint-name*] [[at] *location-spec*] goto *location-spec*

watchpoint [*eventpoint-modifier*] [/once] [/read] [/write] [name=*watchpoint-name*] [at] *lvalue* [if *conditional-expression*]

watchpoint [*eventpoint-modifier*] [/once] [/read] [/write] /address [name=*watchpoint-name*] [at] *address-expression* {size *size-expression* | type *expression*} [if *conditional-expression*]

Controlling Execution

`continue` [*count*]

Abbreviation: **`c`**

`resume` [*sigid*]

`step` [*repeat*]

Abbreviation: **`s`**

`next` [*repeat*]

Abbreviation: **`n`**

`stepi` [*repeat*]

Abbreviation: **`si`**

`nexti` [*repeat*]

Abbreviation: **`ni`**

`finish`

`stop`

`jump` [at] *location-spec*

`signal` *sigid*

`handle` [/signal] *sigid keyword* ...

`handle` /exception *exception-name keyword* ...

`handle` /exception *unit-name keyword* ...

`handle` /exception all *keyword* ...

`handle` /unhandled_exception *keyword* ...

Selecting Context

`frame` [*frame-number*]

frame **expression* [at *location-spec*]

Abbreviation: **f**

up [*number-of-frames*]

down [*number-of-frames*]

select-context default

select-context task=*expression*

select-context thread=*expression*

select-context lwp=*lwpid*

Miscellaneous Commands

help [*section*]

refresh

shell [*shell-command*]

source *command-file*

delay [*milliseconds*]

Info Commands

Status Information

info log

info eventpoint [/verbose] [*name* | *number*] ...

info breakpoint [/verbose] [*name* | *number*] ...

Abbreviation: **i b**

info tracepoint [/verbose] [*name* | *number*] ...

info patchpoint [/verbose] [*name* | *number*] ...

info monitorpoint [/verbose] [*name* | *number*] ...

info agentpoint [/verbose] [*name* | *number*] ...

info frame [/v] [**expression* [at *location-spec*]]

info directories

info convenience

info display

info history [*number*]

info limits

info registers [*regexp*]

info signal [*signal* ...]

info process

info memory [/verbose]

info dialogue

info family [*regexp*]

info name [*regexp*]

info on dialogue [*name*]

info on program [*program*]

info on restart [output=*outname*|append=*outname*] [*program*]

info exception *exception-name*...

info exception *unit-name*

info exception

Symbol Table Information

info args

info locals [*regexp*]

info variables [*regexp*]

info address *identifier*

info sources [*pattern*]

info functions [*regexp*]

info types [*regexp*]

info whatis *expression*

Abbreviation: **whatis**

info representation *expression*

Abbreviation: **representation**

info declaration *regexp*

Abbreviation: **ptype**

info files

info line [at] *location-spec*

Defining and Using Macros

define *macro-name*[(*arg-name* [, *arg-name*] ...)] [*text*]

define *macro-name* [(*arg-name* [, *arg-name*] ...)] as

info macros [*regexp*]

D
GUI Customization

This appendix contains information that you need if you want to customize the graphical user interface.

NightView's behavior may be modified by specifying resources. Resources can be specified in many ways. A complete discussion of this topic is beyond the scope of this text. For more information on setting X11 client resources, refer to the *X Window System User's Guide* or to the X man page **X(1)**.

NightView's default resources are specified in the file **/usr/lib/X11/app-defaults/Nview**. Default color resources are specified in the file **/usr/lib/X11/app-defaults/Nview-color**; default monochrome resources are specified in the file **/usr/lib/X11/app-defaults/Nview-mono**. See “Color Selection” on page D-6. You can look in these files for examples of ways to customize NightView's appearance and behavior.

One way to specify resources is to copy the default resource files to your home directory and change your versions of NightView's resource files. That is the method used in this appendix.

Application Resources

In addition to the standard resources associated with an X11 or Motif program, NightView defines special *application resources* you can use to customize NightView's appearance and behavior. See Appendix D [GUI Customization] on page D-1. These resources affect the entire NightView graphical user interface; they are "global" to the application.

There are two categories of application resources used by NightView. One set of application resources applies to all products that are part of the NightStar™ tool set. In addition to these, NightView has its own application resources.

NightStar Resources

NightView is part of the NightStar tool set. To provide a consistent appearance among these tools and to provide an easy way for you to change the default appearance, special application resources exist that define fonts and colors. They allow you to change one resource (instead of many) to affect the font or color for a set of window components that have similar characteristics. These resources are applied only to certain window components; many of NightView's window components are unaffected by the NightStar resources.

For example, some textual display areas show only program output and some areas accept input only from you. Different colors are used for these areas to distinguish them. If you want to change the color for input fields, for example, you need to change only one resource in NightView's color resource file. See "NightStar Color Resources" on page D-4. The next time you run NightView, the color of all the input fields has the new setting.

Changing the `inputBackground` line in your **`Nview-color`** file to:

```
*inputBackground:          Yellow
```

causes the background color for all input areas to be yellow. (This assumes that you are using a color display and that the `useNightStarColors` resource is `True`. See "Using NightStar Resources" on page D-2.)

Resource values are specified in the application resource files. See Appendix D [GUI Customization] on page D-1.

Using NightStar Resources

The following resources are provided so you can control NightView's appearance as it applies to the NightStar resources. In most cases, however, the default values for the following resources should be used.

`useNightStarFonts`

> By default, this resource is `True`. It controls whether the NightStar fonts are used by NightView.

`useNightStarColors`

> By default, this resource is `True`. It controls whether the NightStar colors are used by NightView.

These resources are specified in the **`/usr/lib/X11/app-defaults/Nview`** resource file.

If you set one of these resources to `False`, NightView does not use the corresponding NightStar resource. Instead, only standard X11 resources are used (such as `*background`, `*foreground`, `*fontList`, as defined in the resource files), and you are responsible for explicitly specifying fonts and colors for NightView's window components.

For example, if you set `useNightStarFonts` to `False`, all of NightView's textual display would use the font defined for the standard `fontList` resource. See "Font Selection" on page D-6. The NightStar default font is a proportional-width font. Some of NightView's textual displays require a fixed-width font for proper text alignment, so this default proportional font is inappropriate for these areas. You would need to specify a fixed-width font, individually, for some of NightView's display areas. Adding the following lines to your resource file would tell NightView to use the `6x13` fixed-width font for the text in the Dialogue Window's process summary list, and the Debug

Window's source display area. See “Widget Hierarchy” on page D-7.

```
*processSummaryHeadingsLabel*fontList:        6x13
*processSummaryList*fontList:                 6x13
*sourceText*fontList:                         6x13
```

If `useNightStarFonts` is set to `True`, NightView takes care of setting the fonts for you based on font resource values in the resource file.

NightStar Font Resources

This section describes the special font resources available for NightStar tools. In addition to these resources, NightStar tools specify an overall *default font* that is used for most of the textual display. See “Font Selection” on page D-6. NightStar tools use proportional-width fonts except in areas that depend on text alignment; in these instances a fixed-width font is important for readability. If you decide to change fonts, make sure that you choose another fixed-width font for the font resources that have *fixed* in their names.

NightStar font resources include:

`boldFontList`

Used for text that is emphasized to attract your attention.

`smallFontList`

Used for areas that require a smaller font.

NightView does not currently use this font.

`infoFontList`

Used for areas that display informational messages, warnings, errors.

NightView does not currently use this font. The default font is used for these areas.

`fixedFontList`

Used for areas that depend on text alignment.

NightView areas that use this font include headings for lists, lists and the display area in the Monitor Window.

`smallFixedFontList`

Used for areas that depend on text alignment but require a smaller font.

NightView areas that use this font include message areas in the Dialogue, Debug and Global Windows; dialogue I/O area; and Debug source display area.

The **`/usr/lib/X11/app-defaults/Nview`** resource file specifies the font values for NightView.

NightStar Color Resources

This section describes the special color resources available for NightStar tools. In addition to these resources, NightStar tools specify an overall *default color* that is used for most of the window areas. See "Color Selection" on page D-6. NightStar tools use the same color scheme to indicate that they are part of the same tool set and to provide cues about the usage of different areas in the windows. Each NightStar tool uses a unique color for its menu bars.

The following NightStar color application resources are defined:

`outputBackground`
`outputForeground`

Used for the background and foreground colors in output-only areas.

NightView areas that use these color resources include message areas in the Dialogue, Debug and Global Windows; lists; Debug source display area; display area in the Monitor Window; and text input areas that are used for displaying information, such as the eventpoint dialog boxes when used for changing an existing eventpoint's attributes.

`inputBackground`
`inputForeground`

Used for the background and foreground colors in areas that accept user input.

NightView areas that use these color resources include text entry areas such as the command areas in the Dialogue, Debug and Global Windows; the dialogue I/O area; and other text input areas in dialog boxes.

`distinctBackground`
`distinctForeground`

Used for the background and foreground colors in areas that *require* user input.

NightView areas that use these color resources include fields in the eventpoint dialog boxes that require you to enter data before the eventpoint can be successfully set.

`feedbackBackground`
`feedbackForeground`

Used for the background and foreground colors of the user feedback area (see "Message Areas" on page 9-6) for feedback that does not provide progress information. These colors default to the same values as `outputBackground` and `outputForeground`, respectively.

`feedbackNotDoneBackground`
`feedbackNotDoneForeground`

Used for the background and foreground colors of the user feedback area for that portion representing work to be done, in those cases where progress information is provided.

`feedbackDoneBackground`

`feedbackDoneForeground`

Used for the background and foreground colors of the user feedback area for that portion representing the amount of work completed, in those cases where progress information is provided.

We recommend that the `feedbackDoneForeground` and `feedbackNotDoneForeground` colors always be the same.

The **`/usr/lib/X11/app-defaults/Nview-color`** resource file specifies the color values for NightView.

NightView Resources

In addition to NightStar resources, NightView has application resources that you can set. See "NightStar Resources" on page D-1. These resources are not shared by other NightStar tools.

The following NightView resources are available.

`editor`

This resource allows you to define the editor that is invoked by the `Source->Edit` menu item. If the `editor` resource is not defined (default), then the name of the editor is taken from the environment variable EDITOR. If there is no EDITOR variable, then **`vi`** is used. The editor is invoked with the name of the current source file as the sole argument. See "Debug Source Menu" on page 9-22.

`editorTalksX`

Setting this resource to `True` indicates that your editor can communicate with the X Window System directly. The default value for this resource is `False`. See the description of the `Source->Edit` menu item for further information. See "Debug Source Menu" on page 9-22.

`monitorWindowColumns`

This resource controls the column width in the Monitor Window. The default value for this resource is 80. See "Monitor Window - GUI" on page 9-48.

`lockButtonSelectColor`

This resource controls the selected color of the debug source lock button. See "Debug Source Lock Button" on page 9-29. The default value for this resource is `#ff0000` (red) and it is set in the **`/usr/lib/X11/app-defaults/Nview-color`** file.

`useKoalaTalk`

Set this resource to `False` if you do not want NightView to communicate with other tools. See "Using NightView with Other Tools" on page 3-35. The default value of this resource is `True`.

Resource values are specified in the application resource files. See Appendix D [GUI

Customization] on page D-1.

Font Selection

NightView defines a *default font* to use for most of the textual display in the windows. This proportional-width font is specified in the **/usr/lib/X11/app-defaults/Nview** resource file as the value of the standard Motif `fontList` resource. This font is used by window components that do not have a font specified for them.

A few of the window components use fonts specified by NightStar font resources. These fonts are specified in the same resource file as the default font. See "NightStar Font Resources" on page D-3.

You can change the fonts used by NightView, and you can control whether or not you use the NightStar fonts. See "Using NightStar Resources" on page D-2. You can, for example, change the default font by setting the resource `fontList`. Changing the `fontList` line in your **Nview** file to:

```
*fontList:          9x15
```

causes NightView to use the `9x15` font for most of the textual display.

Fonts can take up a lot of memory in your X server. If you are running low on server memory, you might want to set up your resources so that you use fewer fonts.

Color Selection

NightView defines a *default color* to use for most of the window areas. This color is specified in the **/usr/lib/X11/app-defaults/Nview-color** resource file as the value of the standard X11 `background` resource. This color is used by window components that do not have a color specified for them.

A few of the window components use colors specified by NightStar color resources. These colors are specified in the same resource file as the default color. See "NightStar Color Resources" on page D-4.

You can change the colors used by NightView, and you can control whether or not you use the NightStar colors. See "Using NightStar Resources" on page D-2.

NightView determines whether you are using a monochrome or color display and automatically loads the appropriate NightView monochrome or color application defaults file. This means that you do not have to specify an X11 `customization` resource explicitly. If you do specify this resource (using either `-color` or `-mono` for the value), NightView still loads the appropriate application defaults file and uses its resource values.

Monochrome Display

The file **/usr/lib/X11/app-defaults/Nview-mono** has examples of monochrome resource specifications that were chosen to help distinguish certain fields using standard X Window System bitmaps.

If you want NightView to have white text on a black background, you can add these resources to your **Nview-mono** or **Nview** file.

```
*background: black *foreground: white
```

Color Display

The file **/usr/lib/X11/app-defaults/Nview-color** has examples of color resource specifications. These resources include the default color and NightStar colors. See "NightStar Color Resources" on page D-4. The colors in this file were chosen to help distinguish certain fields and to emphasize areas that accept user input.

If you want to make changes to the colors, change your copy of the **Nview-color** file.

Window Geometry

If you want to specify window geometries for the individual NightView windows, then you need to refer to the TopLevelShell widget for each window. See "Widget Hierarchy" on page D-7. For example,

```
*globalWindowShell*geometry:    +0+0
*dialogueWindowShell*geometry:  -0+0
*DebugWindowShell*geometry:     +0-0
```

would put the Global Window in the upper left corner, the Dialogue Window in the upper right corner, and the Debug Window in the lower left corner.

Widget Hierarchy

Information about the widget hierarchy for the graphical user interface is useful for modifying the behavior of NightView through the use of standard X11 or Motif resources. You can get this information by using **editres(1)**. See the man page for information about **editres**.

The widget hierarchy specific to the Monitor Window requires additional explanation not covered by the functionality of **editres(1)**. The items displayed in the Monitor Window (in the graphical user interface) are composed of three label gadgets, one each for the identifier, stale data indicator and value. These labels inherit attributes from their

parent (the monitorBulletinBoard). The names of the gadgets are "label", "status" and "value".

The icons for the various stale data indicators may be changed by changing the resources `updatedStatusPixmap`, `notExecutedStatusPixmap` and `notSampledStatusPixmap`.

E
Implementation Overview

This section gives a very high-level description of how the debugger is implemented.

The user invokes **`nview`**. **`nview`** is a script that runs either **`snview`** or **`xnview`**. **`snview`** implements the command-line and simple full-screen interfaces. **`xnview`** implements the graphical user interface. (Users are discouraged from invoking **`snview`** or **`xnview`** directly.) The user interface programs deal with all aspects of the user interface and with managing the symbolic debugging information from executable files. See Chapter 6 [Invoking NightView] on page 6-1.

NightView runs `NightView.p` for each dialogue. If the dialogue is on the local machine, then NightView communicates with `NightView.p` via a shared memory region. There is one such shared memory region per invocation of NightView. See “Dialogues” on page 3-4. For remote dialogues, NightView establishes a socket connection with `NightView.p`.

`NightView.p` is responsible for controlling the user processes by some combination of the `/proc` file system and the debug agent. See “Debugger Mechanisms” on page 3-17.

If the debug agent is used, it communicates with `NightView.p` via a shared memory region. There is one of these shared memory regions for each process using a debug agent. See “Debug Agent” on page 3-17.

Monitorpoints communicate with `NightView.p` via a shared memory region created in your process. There is one shared memory region for each dialogue using monitorpoints. See “Monitorpoints” on page 3-10. The shared memory region is placed in your process somewhere in the range `0xa0000000` to `0xb0000000`, if there is space available in that range. Otherwise, it is placed anywhere NightView can find space.

Each dialogue runs a shell and controls it using `/proc`. This is not to get control of the shell, but so that the debugger is notified of the shell's children, which are the processes to be debugged. The shell runs at a pseudo-terminal controlled by the debugger, so that the debugger can capture the program I/O.

Watchpoints are implemented by setting the hardware `dabr`. Other eventpoints are implemented by replacing the instruction at the target address by a branch to a patch area. The patch area contains instructions to implement the particular eventpoint, emulate the replaced instruction, and return to the target address.

Space for a patch area is acquired by using `mmap` or by creating a shared memory region in the process's address space. The debugger usually creates one data patch area, one text patch area, and one or two eventpoint patch areas. The user can adjust the sizes of the patch areas. See “set-patch-area-size” on page 7-50. Each region is only created in the process if necessary.

Eventpoint patch areas must be within 2**25 bytes of the eventpoint target address. Due to the small amount of memory covered, a complex method is used to determine where the eventpoint patch areas are placed, in order to cover as much of the code space as possible without overlapping the user code. The statically-linked portion of the program

usually begins at 0x10000000, so an eventpoint patch area is placed at (0x12000000-(eventpoint patch area size)), if possible. If that address space is already occupied, NightView attempts to place the region somewhere within the range 0x10000000 - 0x12000000.

For the dynamically-linked portion of the program, eventpoint patch areas may be placed at (0xb0000000-(eventpoint patch area size)) and at (0xb4000000-(2*(eventpoint patch area size))). Other eventpoint patch areas may be created for some programs. The data patch area is placed somewhere in the range 0xa0000000 to 0xb0000000, if space is available; otherwise it is placed anywhere NightView can find room. The text patch area is placed in a manner similar to the data patch area.

You can see where NightView has placed patch areas with the **`info memory`** command (see "info memory" on page 7-126).

The user process is sometimes forced to execute code on behalf of the debugger. This is how function calls work in evaluated expressions, and it is also used to do some of the housekeeping chores, e.g., creating memory regions.

F
Performance Notes

Debug Agent Performance

The performance of the debug agent (see "Debug Agent" on page 3-17) is affected by the operations it is asked to perform, and by whether NightView is able to tell whether the memory locations accessed by such operations are valid or not. Reading from or writing to memory locations that NightView already knows are valid addresses takes much less time than if the locations are not known to NightView. (Some examples of locations not known to NightView are addresses in the heap and stack and addresses in shared-memory regions your program attaches to.)

Writing to memory that contains executable instructions is more expensive than other forms of reads or writes. You should be aware that NightView must modify the executable instructions when it creates or deletes an eventpoint. See "Eventpoints" on page 3-8.

The effect of debug-agent calls on the performance of the debugged program has been measured on a NightHawk® 4800 system running CX/UX 7.1. All pages of the debugged program were locked in memory, and the program was isolated to a CPU from which all interrupts were excluded; the program was also running at the highest possible priority. Various statistics from this measurement, separated according to the types of operation performed, are set forth below. Times are approximate and may vary under different circumstances.

Calls to the agent when there is nothing to do:

Maximum

278 microseconds per call

Minimum

35 microseconds per call

Average

38 microseconds per call

Calls to agent when it performs only reads and writes of data (maximum of 160 bytes per operation):

Maximum

1194 microseconds per call

Calls to agent when it performs a mix of operations, including writes to executable

instructions:

Maximum

1816 microseconds per call

G
Tutorial Files

The following sections show source listings for the files used in the tutorials. These files all reside under the **`/usr/lib/NightView/Tutorial`** directory.

C Files

msg.h

```
 1   #include <stdio.h>
 2   #include <sys/types.h>
 3   #include <signal.h>
```

main.c

```
 1   #include "msg.h"
 2
 3   /* This program spawns a child process and sends
 4    * signals from the parent to the child.
 5    *
 6    */
 7
 8   main()
 9   {
10      int total_sig;
11      pid_t pid;
12      char *tracefile = "msg_file";
13      extern void parent_routine();
14      extern void child_routine();
15      extern void signal_handler();
16
17      signal( SIGUSR1, signal_handler );
18      printf( "How many signals should the parent send
the child?\n" );
19      scanf( "%d", &total_sig );
20      pid = fork();
21
22      if( pid == 0 )
23      {
24         /* It's the child */
25         child_routine( total_sig );
```

```
26      }
27
28      else
29      {
30         /* It's the parent */
31         parent_routine( pid, total_sig );
32      }
33
34      exit( 0 );
35   }
```

parent.c

```
 1   #include "msg.h"
 2
 3   /* Every time the parent sends the child a signal,
 4    * the parent writes a message.
 5    */
 6
 7   void parent_routine( child_pid, total_sig )
 8   pid_t child_pid;
 9   int   total_sig;
10   {
11      int isec = 2;
12      int sig_ct;
13
14      for( sig_ct = 1; sig_ct <= total_sig; ++sig_ct )
15      {
16         printf( "%d.  Parent sleeping for %d
seconds\n", sig_ct, isec );
17         sleep( isec );
18         kill( child_pid, SIGUSR1 );
19      }
20   }
```

child.c

```
 1   #include "msg.h"
 2
 3   /* Every time the child receives a signal from
 4    * the parent, the child writes a message.
 5    */
 6
 7   int sig_ct_child = 0;
 8
 9   void child_routine( total_sig )
10   int total_sig;
```

```
11   {
12      extern void signal_handler();
13
14      signal( SIGUSR1, signal_handler );
15
16      while( sig_ct_child < total_sig )
17      {
18         pause();
19         printf( "Child got ordinal signal #%d\n",
sig_ct_child );
20      }
21   }
22
23
24
25   /* Count how many signals have been received */
26
27   void signal_handler(sig_num)
28   int sig_num;
29   {
30      signal( SIGUSR1, signal_handler );
31      ++sig_ct_child;
32   }
```

Fortran Files

msg.i

```
 1   C     Constants for main.f, parent.f, and child.f
 2
 3         integer    SIGUSR1
 4         parameter  (SIGUSR1=30)
```

main.f

```
 1   C     This program spawns a child process and sends
 2   C     signals from the parent to the child.
 3   C
 4
 5         program  main
 6         common   /msg_comm/ total_sig
 7         integer  total_sig
 8         integer  pid
 9         integer  fork
10         character *8 tracefile
11         external parent_routine
12         external child_routine
```

```
   13
   14         tracefile = "msg_file"
   15         write( 6,* ) "How many signals should the
parent send the child?"
   16         read( 5,* ) total_sig
   17         pid = fork()
   18
   19         if( pid .eq. 0 ) then
   20   C        It's the child
   21            call child_routine()
   22         else
   23   C        It's the parent
   24            call parent_routine( pid )
   25         end if
   26
   27         call exit
   28         end
```

parent.f

```
    1   C     Every time the parent sends the child a signal,
    2   C     the parent writes a message.
    3
    4         subroutine parent_routine( child_pid )
    5         common     /msg_comm/ total_sig
    6         integer    child_pid
    7         integer    total_sig
    8         integer    isec
    9         integer    ireturn
   10         integer    sig_ct
   11         integer    kill
   12         include    "msg.i"
   13         data       isec/2/
   14
   15         do 10 sig_ct = 1, total_sig
   16            write( 6,* ) sig_ct, ".  Parent sleeping
for", isec, "seconds"
   17            call sleep( isec )
   18            ireturn = kill( child_pid, SIGUSR1 )
   19   10    continue
   20
   21         return
   22         end
```

child.f

```
    1   C     Every time the child receives a signal from
```

```
 2   C     the parent, the child writes a message
 3
 4         subroutine child_routine( )
 5         common     /msg_comm/ total_sig
 6         common     /sig_comm/ sig_ct_child
 7         integer    total_sig
 8         integer    sig_ct_child
 9         integer    ireturn
10         integer    csignal
11         cexternal  pause
12         integer    pause
13         external   signal_handler
14         integer    signal_handler
15         include    "msg.i"
16
17         ireturn = csignal( SIGUSR1, signal_handler, -1
)
18
19         while( sig_ct_child .lt. total_sig )
20            ireturn = pause()
21            ireturn = csignal( SIGUSR1, signal_handler,
-1 )
22            write( 6,* ) "Child got ordinal signal #",
sig_ct_child
23         end while
24
25         return
26         end
27
28
29
30   C     Count how many signals have been received
31
32         integer function signal_handler()
33         common  /sig_comm/ sig_ct_child
34         integer sig_ct_child
35         data    sig_ct_child  /0/
36
37         sig_ct_child = sig_ct_child + 1
38         return
39         end
```

Ada Files

main.a

```
 1   -- This program spawns a child process and sends
 2   -- signals from the parent to the child
 3
 4   with child_routine;
```

```
 5   with parent_routine;
 6   with text_io;
 7   with posix_1003_1;
 8   procedure main is
 9
10      pid       : posix_1003_1.pid_t;
11      total_sig : integer;
12      tracefile : constant string := "msg_file";
13      buffer    : string (1..80) ;
14      last      : natural ;
15
16   begin
17
18     text_io.put_line("How many signals should the
parent send the child?" );
19     text_io.get_line (buffer, last) ;
20     total_sig := integer'value(buffer(1..last)) ;
21     pid := posix_1003_1.fork;
22
23     if (pid = 0) then
24         -- It's the child
25         child_routine( total_sig );
26     else
27         -- It's the parent
28         parent_routine( pid, total_sig );
29     end if;
30
31   end main;
```

parent.a

```
 1   -- Every time the parent sends the child a signal,
 2   -- the parent writes a message.
 3
 4   with posix_1003_1;
 5   with text_io;
 6   procedure  parent_routine(child_pid :
posix_1003_1.pid_t;total_sig : integer ) is
 7
 8      isec      : integer := 2;
 9      sig_ct    : integer := 1;
10      stat      : integer;
11
12   begin
13
14      while sig_ct <= total_sig loop
15         text_io.put_line( integer'image(sig_ct) & ".
Parent sleeping for "
16            & integer'image(isec) & " seconds" );
17         delay duration( isec );
```

```
   18         stat := posix_1003_1.kill( child_pid,
posix_1003_1.SIGUSR1 );
   19         sig_ct := sig_ct + 1;
   20      end loop;
   21
   22   end parent_routine;
```

child.a

```
    1   -- Every time the child receives a signal from
    2   -- the parent, the child writes a message.
    3
    4   package child_signal_handler is
    5
    6      sig_ct_child     : integer;
    7
    8      procedure signal_handler;
    9
   10   end child_signal_handler;
   11
   12   package body child_signal_handler is
   13
   14      procedure signal_handler is
   15      begin
   16         sig_ct_child := sig_ct_child + 1;
   17      end signal_handler;
   18
   19   end child_signal_handler;
   20
   21   with system;
   22   with posix_1003_1;
   23   with text_io;
   24   with child_signal_handler;
   25
   26   procedure child_routine( total_sig : integer ) is
   27   --
   28      act : posix_1003_1.sigaction_t;
   29      stuff : integer;
   30   --
   31   begin
   32   --
   33      act.sa_handler :=
child_signal_handler.signal_handler'address;
   34      stuff :=
posix_1003_1.sigemptyset(act.sa_mask'address);
   35      act.sa_flags := 0;
   36      child_signal_handler.sig_ct_child := 0;
   37      stuff :=
posix_1003_1.sigaction(posix_1003_1.SIGUSR1, act'address);
   38      while child_signal_handler.sig_ct_child <
total_sig loop
```

```
   39         stuff :=
posix_1003_1.sigsuspend(act.sa_mask'address);
   40         text_io.put_line( "Child got ordinal signal #"
&
   41
integer'image(child_signal_handler.sig_ct_child) );
   42      end loop;
   43   --
   44   end child_routine;
   45
```

H
Reporting Bugs

This section describes how to report bugs in NightView. It is important to report problems, otherwise we might never know about them. You can report a problem by calling the Concurrent Software Support Center. For more information, see the section *Direct Software Support* in the release notes for the current release.

It is also important to report a problem in a way that helps us understand and reproduce the problem.

Here is a list of things you should tell us in a problem report.

- What version of NightView are you using? You can get this by running `nview -v` or by activating On Version in the Help menu in the GUI.
- What type of machine are you running on, and what is the version of the operating system? Use the command `uname -a` to get this information.
- Are you using the GUI, the simple-screen interface, or the command line interface? Sometimes this is not obvious from the description of the problem.
- Try to be very explicit about what you see happen when the problem occurs. Do you get any error messages? Exactly what incorrect behavior do you see? How will we know when we have reproduced the problem?
- Try to isolate the problem to a small test program and a small series of actions in the debugger. This is not always possible, but to the extent that you can isolate the problem, that will help us reproduce and fix it.
- Be explicit about exactly how to reproduce the problem. Try not to leave out any facts, even if you think you know the cause of the problem.

Glossary

This glossary defines terms used in NightView. Terms in *italics* are defined here.

accelerator

A special key used to select a menu item quickly in the graphical user interface. See also *mnemonic*. See “Keys” on page 9-10.

Ada task

Ada tasks are entities whose executions proceed in parallel. Different tasks proceed independently, except at points where they synchronize.

agentpoint

A call to the *debug agent* (see “Debug Agent” on page 3-17) inserted by NightView at your direction. You can set an agentpoint with the **`agentpoint`** command. See “agentpoint” on page 7-87. Agentpoints may be conditional.

anchored match

The entire string must match the regular expression. Put another way, a `^` is implied at the beginning of the regular expression, and a `$` is implied at the end of the regular expression. See “Regular Expressions” on page 7-12.

application

A group of related processes. The processes may be running the same program or different programs.

application resource

Application resources are application-specific resources defined for an X11 or Motif application. They allow you to customize the appearance or behavior of the application. Application resources affect the entire application. See “Application Resources” on page D-1.

attaching

Attaching to a process means that the debugger will have control over it. This is how you debug processes that already exist. See “attach” on page 7-32.

breakpoint

A breakpoint is a place in your program where execution will stop. You can set a breakpoint with the **`breakpoint`** command. See “Breakpoints” on page 3-10. Breakpoints

may be conditional, see *conditional breakpoint*. Breakpoints may have debugger commands associated with them, see *breakpoint commands.*

breakpoint commands

A set of debugger commands to be executed when a breakpoint is hit. See *breakpoint.*

checkpoint

A checkpoint saves information about the eventpoints, signal disposition, and other information, for a program. This information is used when a program is *restarted*. See "Restarting a Program" on page 3-14.

child process

When a process *forks*, a new process is created that looks just like the old process. The new process is called a child process and the old process is called the parent process. A process may have many child processes, but only one parent process. You can control whether the child process is debugged with the **`set-children`** command. See "set-children" on page 7-41.

command history

NightView keeps a history of all the commands you enter. You can retreive commands, edit them, and re-enter them. See "Command History" on page 3-32.

command-line interface

A command-line interface deals with only one line at a time. This kind of interface can be used from a terminal or from other programs that expect simple behavior, such as a shell running in emacs. Contrast this with a *full-screen interface* and a *graphical user interface*. See Chapter 7 [Command-Line Interface] on page 7-1.

command stream

A command stream is a set of commands executed sequentially by NightView. The commands attached to a breakpoint form a command stream, as do the commands you type as input to NightView. Execution of commands in one command stream may be interleaved with the execution of commands from another command stream. See "Command Streams" on page 3-29.

conditional breakpoint

A breakpoint may have a language expression associated with it. The breakpoint is ``hit'' only if the expression evaluates to TRUE when the breakpoint is encountered. See *breakpoint*.

context

Context refers to the information the debugger uses to determine how to evaluate an expression. The main components of the context are the *program counter*, which determines the *scope*, and the *stack*. Context determines the language (i.e., Ada, C, C++ or Fortran) as well as the type and location of variables in the program. NightView allows

you to specify the context to be used in interpreting an expression. See "Context" on page 3-24.

convenience variables

A convenience variable is a variable maintained by the debugger to hold the value of an expression. The type of a convenience variable is determined by the type of the expression assigned to it. See "Convenience Variables" on page 3-31.

core file

A core file is a snapshot of a process's memory created by the operating system when the process is aborted. You can examine this process state using NightView. See "Core Files" on page 3-4.

crossing count

A crossing count is the number of times program execution has crossed an eventpoint since the program has started execution. This count is updated even if the ignore count or condition was not satisfied. The crossing count is not updated if the eventpoint is disabled.

current frame

The current frame is one of the frames on the stack of a stopped process. It is often the same as the *currently executing frame*, but other frames can be selected using the **`up`**, **`down`**, and **`frame`** commands. The current frame is used to determine the *context* for evaluating an expression. See "Current Frame" on page 3-25.

currently executing frame

The currently executing frame is the stack frame associated with the most recently called routine in a stopped process. Contrast this with *current frame*.

debugger

A debugger is a tool to help you debug programs. A debugger lets you control the execution of your program and look at your program's memory.

debug agent

A debug agent is a module supplied with NightView that enables debugging while your process is running. The debug agent communicates with NightView through shared memory. See "Debug Agent" on page 3-17.

debug session

A debug session is one invocation of NightView; it lasts until you exit from the debugger. See Chapter 6 [Invoking NightView] on page 6-1. See "Quitting NightView" on page 7-17.

Debug Window

In the graphical user interface, a Debug Window allows you to manipulate and debug one or more processes. See also *process*. See "Debug Window" on page 9-20.

default color

The default color is specified by the X11 `background` resource and applies only to the graphical user interface. See "Color Selection" on page D-6.

default font

The default font is specified by the Motif `fontList` resource and applies only to the graphical user interface. See "Font Selection" on page D-6.

detaching

Detaching from a process means that the debugger no longer has control over that process and any future children that are created by that process. The debugger still has control over previously created children. See "detach" on page 7-32.

dialogue

NightView provides dialogues as a means of starting processes, via a shell, and communicating with those processes. See "Dialogues" on page 3-4. See also *remote dialogue*.

Dialogue Window

In the graphical user interface, a Dialogue Window provides you with a way to interact with a NightView dialogue. See also *dialogue*. See "Dialogue Window" on page 9-16.

disassembly

A symbolic representation of the raw machine language that makes up your program. To disassemble part of your program, use the **`x`** command with the `i` format. See "x" on page 7-68.

display item

A display item is an expression or memory location whose value or contents are to be printed out whenever the associated process stops. NightView assigns a unique number to each display item in each process. See "display" on page 7-72 and "info display" on page 7-123.

DWARF

DWARF is the standard format for symbolic debugging information used with ELF files. See *ELF*.

ELF

Executable and Linking Format. This is a standard for the format and contents of an executable file. It also determines the form and content of information about your program available to the debugger.

event-map file

An event-map file lets you associate or map symbolic *trace-event tags* and numeric *trace-event IDs*. This file appears on the **`ntrace`** invocation line when performing *NightTrace* tracing. See *trace*.

eventpoint

An eventpoint is a generic name given to the various kinds of modifications NightView can insert at a particular location of a process. The different kinds of eventpoints are: *breakpoint*, *monitorpoint*, *tracepoint*, *patchpoint*, *agentpoint*, and *watchpoint*. See "Eventpoints" on page 3-8.

eventpoint modifier

An eventpoint modifier modifies the meaning of an eventpoint command. The only eventpoint modifiers are `/delete` and `/disabled.` The modifier `/delete` is valid only for breakpoints and watchpoints, and tells NightView to delete the eventpoint after the next time it is hit. The modifier `/disabled` tells NightView to create the eventpoint, but leave it disabled initially. See "Eventpoint Modifiers" on page 7-78.

exception

An Ada exception is an error or other exceptional situation that arises during program execution. Normal program execution is abandoned, and special actions are executed. Executing these actions is called handling the exception. An exception can also be caused by a raise statement. When an exception arises, control can be transferred to a user written handler at the end of a block statement, body of a subprogram, package or task unit. If a handler is not present in the frame of context in which the exception arises, execution of this sequence of statements is abandoned. The exception will be propagated to the innermost enclosing frame of context if possible. See "handle" on page 7-105. See "info exception" on page 7-129.

family

A group of related processes. See "family" on page 7-40.

focus

See *keyboard focus.*

fork

Create a new process. The debugger informs you when your process forks. See *child process*.

frame

See *stack frame.*

full-screen interface

A full-screen interface uses the capabilities of a terminal to control the display of information on the entire screen, rather than just writing to the terminal one line at a time. Contrast this with a *command-line interface* and a *graphical user interface*. See Chapter 8 [Simple Full-Screen Interface] on page 8-1.

Global Window

In the graphical user interface, the Global Window shows all of NightView's output messages and allows you to control the debugger if all other windows are closed. See "Global Window" on page 9-48.

graphical user interface

A graphical user interface may be used on a graphics display. This kind of display allows much more flexibility and functionality than a text display. Contrast this with a *command-line interface* and a *full-screen interface*. See Chapter 9 [Graphical User Interface] on page 9-1.

group process mode

In the graphical user interface, a Debug Window can operate in group process mode. This means that you can issue commands that apply to all the processes in the Debug Window. See also *Debug Window.* See "Group Process Mode" on page 9-14.

GUI

A *graphical user interface*.

Help Window

In the graphical user interface, the Help Window displays NightView's online help information. You can choose to look at any part of the *NightView User's Guide.* See also *online help system*. See "Help Window" on page 9-50.

hit a breakpoint

A breakpoint is hit when execution reaches the breakpoint location and the ignore count and conditions, if any, are satisfied. Thus, hitting a breakpoint stops the process. See "Breakpoints" on page 3-10.

hit an eventpoint

An *inserted eventpoint* is hit when execution reaches the eventpoint location and the ignore count and conditions, if any, are satisfied. A watchpoint is hit when the specified addresses are referenced, and the ignore count and conditions are satisified. Thus, hitting an eventpoint causes that eventpoint to perform its specified action; e.g., a breakpoint stops the process, a monitorpoint evaluates its expressions and saves their values, a trace-

point logs a trace event, and so on. See *eventpoint*, *breakpoint*, *monitorpoint*, *tracepoint*, *agentpoint*, and *watchpoint*.

ignore count

An ignore count causes NightView to skip an eventpoint the next *count* times that execution reaches the eventpoint. You use the **`ignore`** command to attach an ignore count to an eventpoint. See "ignore" on page 7-93.

initialization file

An initialization file is a file containing NightView commands that are executed before NightView reads commands from standard input. NightView has a default initialization file, and you can specify others on the NightView invocation line. See "Initialization Files" on page 3-32.

inline subprogram

A subprogram that is expanded directly into the calling program. See "Inline Subprograms" on page 3-26.

inline interest level

The level that determines if any inline subprograms are interesting. You may set an *interest level* for individual inline subprograms to override this level. See "Inline Subprograms" on page 3-26. You can change or query this level with the **`interest`** command. See "interest" on page 7-51.

inserted eventpoint

An *eventpoint* that is associated with a location in your program. Inserted eventpoints are implemented by inserting code into your process. See "Eventpoints" on page 3-8.

interest level

Each subprogram has an associated interest level. NightView compares the interest level to the *interest level threshold* to determine if the subprogram is interesting. NightView generally avoids showing you uninteresting subprograms. See "Interesting Subprograms" on page 3-27. You can change or query the interest level with the **`interest`** command. See "interest" on page 7-51.

interest level threshold

Each process has an interest level threshold. If the *interest level* of a subprogram is less than the interest level threshold, the subprogram is considered to be uninteresting. See "Interesting Subprograms" on page 3-27.

keyboard focus

The keyboard focus determines which field receives keyboard input in the graphical user interface. See "Keyboard Focus" on page 9-10.

macro

A macro is a named set of text, possibly with arguments, that can be substituted in a NightView command by referencing the name. This is a means of extending the facilities provided by NightView. See "Defining and Using Macros" on page 7-134.

mnemonic

A mnemonic is a way of selecting a menu or a menu item quickly in the graphical user interface. See also *accelerator*. See "Keys" on page 9-10.

monitorpoint

A monitorpoint is a location in a debugged process where one or more expressions are evaluated and the values saved. The saved values are displayed periodically by NightView. Monitorpoints thus provide a means of viewing program data while the program is executing. See "Monitorpoints" on page 3-10 and "monitorpoint" on page 7-84.

NightTrace

An interactive debugging and performance analysis tool that lets you examine trace events logged by user applications and the kernel. See *trace*. See the *NightTrace Manual* for details.

NightView

A pretty good debugger.

online help system

All of the *NightView User's Guide* is available to you, online, through NightView's online help system. In the graphical user interface, help information is displayed in the Help Window. See also *Help Window*. See "help" on page 7-111. See "GUI Online Help" on page 9-2.

overloading

Overloading means that more than one entity with the same name is visible at some point in the program. See "Overloading" on page 3-23.

patch

A patch is an expression (or a branch) inserted into a debugged process to alter its behavior (usually to fix a bug). See *patchpoint*. See "Patchpoints" on page 3-10.

patch area

NightView creates regions, known as patch areas, in your process. This is where NightView puts code and data that is inserted into your process. See Appendix E [Implementation Overview] on page E-1. See "set-patch-area-size" on page 7-50.

patchpoint

A patchpoint is a location in a debugged process where a patch is inserted. See *patch*. See "Patchpoints" on page 3-10.

pattern

A pattern is used in the **debug** and **nodebug** commands to control which programs will be debugged in a particular *dialogue*. Patterns are similar to shell wildcard patterns. See "debug" on page 7-20.

Principal Debug Window

In the graphical user interface, this is the only Debug Window you see unless you decide to create additional Debug Windows. It contains all processes that appear in a NightView session. The Principal Debug Window remains available throughout the NightView session. See also *user-created Debug Window*. See "Debug Window Behavior" on page 9-12.

PID

A process identifier. This is an integer from 1 to 30000 which uniquely identifies a process on a particular system. In some situations, NightView may create false PIDs, outside the normal range, to identify false processes, e.g., core files.

procedure

See *routine*.

process

The execution of a program. Many processes may be executing the same program. See "Programs and Processes" on page 3-2.

process state

A process state describes whether the process is actively executing and what you can do with the process using NightView. The two most common process states are *running* and *stopped*. See "Process States" on page 3-16.

program

A file containing instructions and data. A program is usually created with the **ld(1)** program. An executing program represents a *process*. See "Programs and Processes" on page 3-2.

program counter

The program counter is a register that locates the instruction that is to be executed next. See "Program Counter" on page 3-24.

qualifier

A qualifier specifies the set of processes or dialogues that a command affects. See "Qualifiers" on page 3-4.

registers

Registers are special storage locations in the CPU for holding frequently accessed data. In NightView, you can access most of these registers using specially-named convenience variables. See "Predefined Convenience Variables" on page 7-6.

remote dialogue

A remote dialogue is a *dialogue* started on a system other than the one on which NightView was invoked. See "Remote Dialogues" on page 3-6.

restarted

When a program is run again in the same debug session, it is considered to be *restarted*. Information from the most recent *checkpoint* is applied to the process. See "Restarting a Program" on page 3-14.

routine

Routine is a generic term denoting a function or subroutine in a program. Different languages use different terms for this concept; other similar terms are subprogram and procedure.

scope

A scope is a section of your program where a particular set of variables can be referenced. Scope forms a part of the *context*. See "Scope" on page 3-24.

shell

The shell is the program the system normally executes when you log in. There are several varieties of shell: Bourne shell, C shell, and Korn shell are some examples. In NightView, each *dialogue* you create executes an instance of your login shell.

signal

A signal is a notification of some event to your process. This event may be external to your process, or it may be the result of an erroneous action by the process itself. NightView allows you to control how signals are delivered to your process. See "Signals" on page 3-12.

single process mode

In the graphical user interface, the Debug Window can operate in single process mode. This means that you can issue commands that apply only to the currently displayed process in the Debug Window. See also *Debug Window*. See "Single Process Mode" on page 9-13.

stack

An area of memory used to hold local variables and return information for each active routine. The stack consists of a sequence of *stack frames*. Calling a routine pushes a new frame onto the stack; returning from the routine removes that frame from the stack. See "Stack" on page 3-25.

stack frame

A stack frame is a contiguous set of locations in the process' stack that corresponds to the execution of an active routine. The stack frame holds the local automatic variables of the routine, and it also holds information needed to return to the calling routine. See "Stack" on page 3-25.

stale data indicator

A stale data indicator is a character or icon displayed with a monitored value to indicate the validity and reliability of that value. See *monitorpoint.*

symbol file

An executable file containing symbolic debug information. Normally, the symbol file is the same as the program's executable file, but it may be different if, for example, you are debugging a stripped program. See "symbol-file" on page 7-33.

thread

Each instance of execution of a program contains one or more threads of execution. Some programs have a single thread. Ada programs, through the use of tasking, have multiple parallel threads. See "Multithreaded Programs" on page 3-34.

trace

The collection of data produced by executing *tracepoints* in a process is called a trace. See *NightTrace*.

trace-event ID

An integer that identifies a *NightTrace* trace event. User trace event IDs are in the range 0 through 4095, inclusive. See *event-map file* and *trace-event tag*.

trace-event tag

A symbolic name that identifies a *NightTrace* trace event. It is mapped to a numeric *trace-event ID* in an *event-map file*.

tracepoint

A tracepoint is a call to one of the **`ntrace(3X)`** library routines for recording the time when execution reached the tracepoint. You can insert a tracepoint in your source, or you can use NightView to insert them after starting your process. See "Tracepoints" on page 3-11.

user-created Debug Window

In the graphical user interface, a user-created Debug Window initially contains processes that are defined by you. This type of Debug Window can not be empty. When the last process in the window exits, this type of Debug Window is automatically closed by NightView. See also *Principal Debug Window*. See "Debug Window Behavior" on page 9-12.

value history

The value history is a list of values you have printed in your NightView session. You can view this list, and you can reference the values in other expressions. See "Value History" on page 3-32.

watchpoint

A watchpoint stops the process when the process reads or writes a variable in memory. See "Watchpoints" on page 3-11.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

N

O

P

Q

R

S

T

X

Spine for 1.5” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up

NightView

User’s Guide

0890395