NightView User's Guide

Version 7.3

(RedHawk™ Linux®)

Preface

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.

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 tutorial, see Chapter 4 [Tutorials] on page 4-1.

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 6 [Command-Line Interface] on page 6-1.

The next chapter describes a simple full-screen interface to NightView. See Chapter 7 [Simple Full-Screen Interface] on page 7-1.

The next chapter describes the graphical user interface for NightView. See Chapter 8 [Graphical User Interface] on page 8-1.

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.

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. You do not type the brackets if you choose to specify such option or arguments. Mutually exclusive choices are separated by the pipe (|) character.

{ }

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

The following publications are referenced in this document:

0890514	*NightBench™ User's Guide*
0898004	*RedHawk Linux User's Guide*
0898008	*NightStar RT Installation Guide*

0898008	*NightStar RT Tutorial*
0898398	*NightTrace™ User's Guide*
0898465	*NightProbe™ User's Guide*
0898480	*NightSim™ User's Guide*
0898515	*NightTune™ User's Guide*
0898537	*MAXAda™ for Linux Reference Manual*

Reporting Bugs

Please report any bugs you encounter. Software support is available from the Concurrent Software Support Center at our toll free number 1-800-245-6453. For calls outside the United States, the number is 1-954-283-1822. The Software Support Center operates Monday through Friday from 8 a.m. to 5 p.m. Eastern Standard Time. You may also submit a bug report or a request for assistance at any time by using the Concurrent Computer Corporation web site at http://www.ccur.com/isd_support_contact.asp or by sending an email to support@ccur.com.

Contents

Chapter 7 Simple Full-Screen Interface

Chapter 8   Graphical User Interface

Appendix A NightStar Licensing

Appendix B Kernel Dependencies

Appendix C   Summary of Commands

Appendix D   Quick Reference Guide

Appendix E   Implementation Overview

Appendix F   Tutorial Files

Glossary

Index

Tables

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 [Tutorials] 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 6-140).

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 runs 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 6-37) in a dialogue shell.

Below is an example of starting up the debugger with a program name and a program argument. 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 7
NightView debugger - Version 7.1, linked Fri Jun  8 10:24:51 EDT 2007
Copyright (C) 2007, Concurrent Computer Corporation

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

Note that you invoked NightView with a program name argument **./fact** and program argument **7**. NightView responded with information about the debugger.

```
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 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 6-33.

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: Unrecognized 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:
Unrecognized 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.

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 6-76 and "breakpoint" on page 6-99.

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 6-85).

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.

NOTE

If your system has debug information installed for system librar-
ies, the process may appear to be stopped in the _exit() library
routine after the command below. If so, enter the command **up**
until the debugger reports that the process is in main.

```
(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 6-122, "backtrace" on page 6-85, "print" on page 6-85, and "quit" on page 6-23.

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 4 [Tutorials] on page 4-1.

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 8-18. Click on the NightView User's Guide menu item or use the U mnemonic. NightView puts up a Help Window that displays the table of contents for the manual. See "Help Window" on page 8-88. 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 and arguments. If you start it with a program name, NightView begins debugging the program immediately. If you start NightView without a program name, or you want to debug another program, you may run the program with the Run menu item in the Process menu, or by typing in the shell in a shell panel. See "Shell Panel" on page 8-60. In either case, the program is run in a dialogue shell (see "Dialogues" on page 3-4).

Below is an example of starting up the debugger with a program name and a program argument. 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 7
```

NightView displays the main window. See "Main Window" on page 8-4.

Starting the debugger with the program name **./fact** and argument **7** 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 a shell panel. See "Shell Panel" on page 8-60.

The message panel (see "Message Panel" on page 8-61) 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
```

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 status bar at the bottom of the window displays the program name, fact, the dialogue name and PID, local:2347, and the state, Stopped for exec. See "Status Bar" on page 8-24. The source panel title bar displays the program name, the dialogue name and PID, and the name of the source file, fact.c. The source code from file fact.c appears in the source panel, centered around main. See "Source Panel" on page 8-53.

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 command toolbar is labeled Command:. Click in the combo box of the command toolbox (see "Command Toolbar" on page 8-20) 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:  Unrecognized 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 8-18. 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: Unrecognized 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 8-88.

Next you will read about the **list** command. Click on the Help menu or use the H mnemonic. See "Help Menu" on page 8-18. 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.

apply on dialogue

　Execute on dialogue commands for existing dialogues.

(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.

Right-click on the line with the return statement (line 7) in the source panel. The line becomes highlighted and a context menu appears. See "Source Panel Keystrokes" on page 8-59. Select the first item, Set Simple Breakpoint.

The source line decoration beside line 7 is now a stop sign ● to indicate a breakpoint.

See "breakpoint" on page 6-99 and "Source Line Decorations" on page 6-82. The eventpoint panel now has an entry for the breakpoint.

The message panel shows:

```
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 in the process toolbar. ▶ See "Process Toolbar" on page 8-20.

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 stop sign overlaid with a triangle pointing to the right ▶ to indicate where execution will resume.

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. The status bar indicates the process is Stopped at breakpoint 1. Now look at the call stack. The context panel and the locals panel are in the same area with tabs below them. Click on the Context tab. The context panel has an entry for each frame on the stack, displayed in tree form. See "Context Panel" on page 8-64. You see all the expected recursive calls. Scroll to the bottom of the panel. One of the icons is an arrowhead pointing down. ❤ Click that icon to show more stack frames, until you see the call to main. Then scroll to the top again and click on the first frame.

Now look at the local variables. Click on the Locals tab. You see the local variables displayed in tree form. In this case, there is only one local variable, x. The locals panel tracks the current context, which you set when you clicked in the context panel. The value of x in this frame is 1. See "Locals Panel" on page 8-64.

Now finish running the program. Click on the Resume button. See "Process Toolbar" on page 8-20.

This allowed the process to run to completion. The program printed a message, which appeared in the message panel:

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

NOTE

If your system has debug information installed for system libraries, the process may appear to be stopped in the `_exit()` library routine. If so, click the Up button ⬆ until the debugger reports that the process is in main.

NightView showed the call to `exit(0)` in the source panel and displayed the following message in the message panel.

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

Exit from NightView by selecting the File menu. See "File Menu" on page 8-4. Click on File or use the F mnemonic. Click on the Exit 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^{TM} 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 3-9. 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 6-28. In addition, remote files can be specified by using the form *user*@*host***:/***path*. See "Remote File Access" on page 3-7.

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 6-40. 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 6-51.

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 6-27.

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 6-50.

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 6-39) or with the Attach Dialog Box (see "Attach Dialog Box" on page 8-27.)

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 clicking on the Attach menu item in the Process menu (see "Process Menu" on page 8-8) to bring up the Attach Dialog Box.

As an alternative, you can run the **ps(1)** program. You can use the **shell** command (see "shell" on page 6-141) 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/bin/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.

For the security restrictions on **attach**, see "Attach Permissions" on page 3-42.

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 3-18. 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 6-66.

NOTE

Attaching to a process from which you have detached is not supported on Linux. Avoid detaching from processes unless you are sure you will not want to debug them further.

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 6-41). 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 6-1 and "Qualifier Specifiers" on page 6-16.

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 **rtcp** 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.

NightView sets the TERM environment variable to dumb in the dialogue shell, to avoid problems with some shell programs.

Once the shell is started, you can change directory, set environment variables, or set **ulimit(1)** 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 6-26 and "nodebug" on page 6-27.

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 6-24). 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 6-33) or the **run** command (see "run" on page 6-37). 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 a shell panel (see "Shell Panel" on page 8-60) 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 6-34), and you can review the log with the **show** command (see "show" on page 6-35). 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 3-11.

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 3-12.

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 3-9.

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 6-24. See "Remote Login Dialog Box" on page 8-36.

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 execs a new program, NightView attempts to determine the location of the executable program file. See "Finding Your Program" on page 3-9. 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 6-41. Also, see "translate-object-file" on page 6-28.

Creating a new dialogue involves logging into a system (see "login" on page 6-24) via **ssh(1)**. 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). For convenience with logging in, you might want to investigate **ssh-agent(1)**.

Logging in runs your .profile or other initialization file appropriate to your normal login shell. Your .profile should avoid reading from the standard input if NIGHTVIEW_ENV has a non-empty value.

Remote File Access

Referencing remote files can be useful either during remote debugging or when the files of interest reside on another host.

For remote debugging, in most situations NightView can find files automatically and you don't need to worry about it. However, sometimes you need to provide more information. In those situations, you need to know the rules used by NightView to find files.

If NightView cannot find an object file on the local host, including using object transla-
tions, and the download mode is not off , it attempts to download it from the target system.
See "translate-object-file" on page 6-28 and "set-download" on page 6-73.

If NightView cannot find a source file on the local host and the download mode is not off,
it tries to download it from the target system. See "directory" on page 6-78.

NightView always interprets **exec-file**, **core-file**, and **load** filenames relative to
the target system. See "exec-file" on page 6-44, "core-file" on page 6-41, and "load" on
page 6-95.

Filenames on other commands are interpreted relative to the local host by default; how-
ever, you may explicitly refer to files on other hosts using the form *user*@*host***:/***path*. If
you omit the *user*@ portion, your current user is used. When you use this form, NightView
transfers the file from the *host* system onto the local host system. (The file is downloaded
into a file cache. You can control the behavior of the file cache with the **set-download**
command (see "set-download" on page 6-73) or with the Preferences Advanced Page in
the graphical user interface [see "Preferences Advanced Page" on page 8-48].)

As a special case, if **exec-file** sees the *user*@*host***:/***path* form, the **exec-file** com-
mand is treated as a **symbol-file** command (see "symbol-file" on page 6-41).

If you need to refer to remote files but do not want NightView to transfer them, don't use
the *user*@*host***:/***path* form. Instead, set up another way for NightView to see the files, such
as an NFS mount. NightView automatically sets up object file translations for NFS
mounts. See "translate-object-file" on page 6-28.

The following commands interpret filenames relative to the host by default, but can take
the *user*@*host***:/***path* form:

- **symbol-file** (see "symbol-file" on page 6-41)

- **translate-object-file** (see "translate-object-file" on page 6-28)

- **directory** (see "directory" on page 6-78)

- **source** (see "source" on page 6-141)

- **list** (see "list" on page 6-76)

Example:

Assuming you have a remote dialogue to system fred (see "Remote Dialogues" on page
3-6), and the program and its source are on that system, all you need to do is ensure that
the download mode is set (see "set-download" on page 6-73) and then run the program.

> **set-download temporary**

Example:

Suppose that the program is in **/usr/biff** on system fred, but it has been stripped of
debugging information. The version with the debugging information is in **/usr/joe** on
system barney. The source is in **/usr/bob** on system betty. Use the following com-
mands:

```
set-download temporary
translate-object-file /usr/biff/ barney:/usr/joe/
directory betty:/usr/bob
```

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 6-44.

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, tracepoints, watchpoints, heappoints, and syscallpoints. 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.

Breakpoints, monitorpoints, patchpoints, tracepoints, and heappoints are *inserted eventpoints*. They are implemented by inserting code into your process. Syscallpoints and watchpoints are notn inserted eventpoints. 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 6-111. However, a condition on a syscallpoint or a watchpoint is evaluated in the debugger. For syscallpoints and watchpoints, the language of the expression is determined by your language setting. See "set-language" on page 6-60. Because syscallpoint and 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 6-114. 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 6-96.

Table 3-1. Eventpoint Summary

	Action	Code is inserted	May have commands
breakpoint	stop the process when the breakpoint is reached	X	X
heappoint	check the heap or configure the heap debugger	X	
monitorpoint	display the value of expressions in the monitorpoint window	X	X

Table 3-1. Eventpoint Summary

	Action	Code is inserted	May have commands
patchpoint	execute an expression or modify the flow of the program	X	
tracepoint	record an event when execution reaches the tracepoint	X	
syscallpoint	stop the process when a system call is entered or existed (optionally restarting it)		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 6-99.

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 6-110.

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 3-28). You can set a monitorpoint using the **monitorpoint** command. See "monitorpoint" on page 6-106.

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 6-100) 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 6-95) 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 6-104) 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.

Heappoints

Heappoints can be used to narrow the search for a memory bug. A heappoint either checks the process's heap or changes the configuration of heap debugging, depending on which options you specify. Heappoints may be set only after configuring the heap debugger with

the **heapdebug** command (see "heapdebug" on page 6-54) or from the Process menu (see "Process Menu" on page 8-8). Heappoints are set with the **heappoint** command (see "heappoint" on page 6-107).

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 6-117.

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. Night-View provides a process-local convenience variable, $is, that is useful in watchpoint conditional expressions. See "Convenience Variables" on page 3-37. $is contains the value of the variable (or other program element) after the instruction that triggers the watchpoint has completed.

NOTE

In some cases it would be nice to have a $was (the value of the variable before the instruction began). NightView has no way to know what the value was immediately before the instruction executed. You can approximate $was by adding some code to your conditional expression. However, note that this picks up the old value only when the watchpoint is triggered and that there are circumstances, described below, that can change the value without triggering the watchpoint.

Example:

This example code assumes the current language is C. Suppose you want the process to stop if the old value was 1. Set the condition on the watchpoint to this expression:

```
$was=$prev, $prev=$is, $was==1
```

Set these convenience variables before resuming the process:

```
set $was=0
set $prev=0
set-local $was
set-local $prev
```

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 watch-point is not an *inserted eventpoint*. See "Eventpoints" on page 3-9.

You can have many watchpoints per process, but there is a limit on the number of watch-points that can be *enabled* at the same time. On an IA-32 or AMD64 system, at most 4 watchpoints can be enabled at one time.

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 watch-point 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 calcu-lates 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 pro-gram 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 3-16.

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 con-trol.

NOTE

If the target system is an IA-32 system, then the mechanism NightView uses for watchpoints watches 1, 2 or 4 bytes. If the variable you are watching is 3 bytes long, then you may get some extraneous triggers on the next byte. If the variable you are watch-ing is longer than 4 bytes, only the smallest address bytes of the variable are watched. If you need to watch more bytes of the vari-able, you can use multiple watchpoints, specifying addresses and lengths.

If the target system is an AMD64 system, then the mechanism NightView uses for watchpoints watches 1, 2, 4 or 8 bytes. If the variable you are watching is 3, 5, 6 or 7 bytes long, then you may get some extraneous triggers on the trailing bytes. If the variable you are watching is longer than 8 bytes, only the smallest address bytes of the variable are watched. If you need to watch more bytes of the variable, you can use multiple watchpoints, specifying addresses and lengths.

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 3-9.

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.

Syscallpoints

A syscallpoint stops your program when a system call is entered or exited. It prints the name of the system call and whether the program is about to enter it, or is exiting from it. Options are available to resume the process automatically after printing such information and to control whether entry, exit, or both are trapped.

You can specify a specifc set of system calls to trace or trace all of them.

Syscallpoints are set with the **syscallpoint** command (see "syscallpoint" on page 6-119 or the Syscallpoint Dialog (see "Eventpoint Dialog Boxes" on page 8-29).

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 6-133).

In addition, you may use the debugger to explicitly send a signal to a process (see "signal" on page 6-132). 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 6-121). 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 6-133). 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 6-37.

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 execs 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 execs, put a **resume** command (see "resume" on page 6-123) inside an **on program** command (see "on program" on page 6-46), 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 execs, 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 6-52. 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 6-48. 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.

Unlike other command streams, execution of an **on restart** command stream is not terminated by an error. See "Command Streams" on page 3-36.

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 6-49.

It is not possible to turn off checkpoints. However, you can control whether restart information is applied. See "set-restart" on page 6-65.

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 6-163. 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 6-141.

Restart Information

This section describes the restart information saved during a checkpoint.

- Any memory reservations made with the **mreserve** command. See "mreserve" on page 6-53.

- Eventpoints, including any names, conditions, ignore counts and commands associated with each eventpoint. See "Eventpoints" on page 3-9.

• Directory search path. See "directory" on page 6-78.

• Child disposition. See "set-children" on page 6-51.

• Signal and exception disposition. See "handle" on page 6-133.

• Display list. See "display" on page 6-92.

• Symbol file. See "symbol-file" on page 6-41.

• Default language. See "set-language" on page 6-60.

• Whether or not the process will stop before exiting. See "set-exit" on page 6-52.

• The interest level threshold, the interest level for `inline`, `justlines`, and `nodebug`, and any explicit interest levels for subprograms. See "interest" on page 6-67.

• Information to reproduce the items in the data panel. See "Data Panel" on page 8-64. See "data-display" on page 6-92.

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 6-170. For example, you could define `restart_begin_hook` to be **echo** to disable the **on program** processing. See "on program" on page 6-46.

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 6-52.) 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 3-11. See "Tracepoints" on page 3-12. The first *inserted eventpoint* in a process must be set while the process is stopped, unless eventpoint memory preallocation is on. See "Eventpoints" on page 3-9. See "set-preallocate" on page 6-71. A watchpoint may be enabled or disabled only when the process is stopped. See "Watchpoints" on page 3-13.

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.

Operations While the Process Is Executing

This section lists what you can do while the process is executing.

- 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 3-21. 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 3-25). You can use the special forms Night-View provides to change this context, if you want to access variables local to a procedure, for instance. See "Special Expression Syntax" on page 6-4. 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* (breakpoints, monitorpoints, patchpoints, tracepoints, and heappoints). See "Eventpoints" on page 3-9. 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 6-96.

  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 3-13.

  If preallocation is on (default), the debugger performs special processing to support eventpoints and monitorpoints as soon as the program starts. See "set-preallocate" on page 6-71. If preallocation is off, the following restrictions apply to setting the first eventpoint and the first monitorpoint:

  - 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 3-11.

  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.

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 6-6.

Note that monitorpoints and tracepoints also provide ways of monitoring your program without stopping it. See "Real-Time Debugging" on page 3-6.

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 6-4.

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 3-25). 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 6-60).

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. For Ada and Fortran, the concept of an expression is extended to assignment. (Assignment is an expression in C and C++.)

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 3-24.

- 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 3-23.

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 3-24.

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 means that more than one entity with the same name is visible at the same point in the program. Overloading is allowed for location specifiers and for expressions. In C++ language mode, overloading of functions and operators is allowed. In Ada language mode, overloading of enumeration constants, functions, operators and procedures is allowed. See "set-language" on page 6-60. NightView refers to the appropriate entity if it has enough context to determine that there is only one choice. Otherwise, you 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 6-2. Overloaded names are supported in language expressions (see "Expression Evaluation" on page 3-21) and location specifiers (see "Location Specifiers" on page 6-14), and the same syntax is used for both.

The **set-overload** command (see "set-overload" on page 6-70) 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 3-25) 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 3-26). 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 6-4) 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 6-4) 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 6-85).

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 6-138.

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 6-137) or the **frame** command ("frame" on page 6-136) 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 6-4.

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 3-39 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 3-26.

You can use single step commands to step into inline subprograms, to step over them, or to finish them. See "step" on page 6-124, "next" on page 6-126, and "finish" on page 6-129.

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 6-14.

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 3-21.

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 6-124. 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 6-67.

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 3-11). When you set a monitorpoint (see "monitorpoint" on page 6-106), 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 6-108) 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 3-36.

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 7 [Simple Full-Screen Interface] on page 7-1) or the graphical user interface (see Chapter 8 [Graphical User Interface] on page 8-1) 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 7-2. See "Monitorpoint Values Data Item" on page 8-74.

Debugging the Heap

NightView has features to help debug problems with a program's heap (i.e. memory obtained via `malloc`, `calloc`, `realloc`, etc.). Common problems with a program's heap include buffer overruns, reads or writes of memory through "dangling" pointers which reference freed memory, and memory leaks. The debugger can also provide information about memory usage, such as the number of blocks that are allocated.

The heap debugger can be enabled and configured with the **heapdebug** command (see "heapdebug" on page 6-54), or with the Debug Heap… item in the Process menu. See "Process Menu" on page 8-8.

Once enabled, the heap debugger intercepts calls to the following heap routines:

- `calloc`
- `free`
- `malloc`
- `memalign`
- `posix_memalign`
- `pvalloc`
- `realloc`
- `valloc`

Before allowing the allocator to perform the requested operation, the heap debugger performs some checks for each such call. In addition, it is usually configured to perform a "heap check" with a specified frequency. Also, depending on the configuration, it may allocate extra memory for each block or may fill certain regions with fill bytes. See "heapdebug" on page 6-54 for details on configuration.

In addition, heappoints may be inserted at user-specified locations in the program. They can perform heap checks or change the configuration of heap debugging dynamically. See "Heappoints" on page 3-12.

The heap debugger can remember a walkback list for each allocator operation. The walkback list has the program counter for the caller of the heap routine, and the caller of that routine, and so on. If you encounter a heap error or a memory leak, this tells you which part of your program allocated the block.

The heap debugger can be used to provoke bugs by filling allocated blocks with trash to reveal problems with uninitialized fields, or by filling freed blocks with trash to reveal problems with dangling pointer references. You can also discover how your program behaves when it runs out of memory by restricting the amount of memory the heap debugger will allocate.

You can also hide the effects of some bugs. This is not intended as a remedy, but rather as information about what might be wrong with your program. For example, you can allocate extra memory for each block, which helps determine if your program is not allocating blocks of the right size, or have `malloc` zero-fill each block, which helps determine if your program is not initializing fields before using them.

Levels and Common Errors

If you have no interest in the details of heap debugging, you may want to use heap debugging levels. The levels are just a convenient way to configure heap debugging.

Level 0 (Disable) sets the controls so that the heap debugger does as little as possible, but can still issue errors for invalid heap operations.

Level 1 (Low) sets the heap debugger to do some heap checking with relatively small overhead, and is the default setting. Level 1 also enables all the features that do not change the behavior of the system allocator. That is, if you run your program without the heap debugger and then you run your program with the heap debugger at level 1, you will get the same pattern of block addresses and block sizes (however, note that some versions of the operating system vary the address space layout randomly for each process). Enabling more features may change the pattern.

Level 2 (Medium) sets the heap debugger to do more checking with greater overhead in memory used and execution time. It may also cause subtle changes in the behavior of the system allocator which can hide or expose different bugs.

Level 3 (High) does a very high level of checking at the cost of extreme overhead in memory used and execution time. In particular, the heap is checked before every heap operation and all freed blocks are retained. It may also cause subtle changes in the behavior of the system allocator.

Similarly, using the `common_errors` keyword in the **heapdebug** command (see "heapdebug" on page 6-54), or clicking one of the Common Errors buttons in the graphical user interface, provides a convenient way to configure heap debugging for particular kinds of errors.

Entering

    heapdebug common_errors=block_overrun

or clicking the Block Overrun button in the graphical user interface, configures the heap debugger to detect if the program references past the end of an allocated block.

Entering

    heapdebug common_errors=dangling_pointer

or clicking the Dangling Pointer button in the graphical user interface, configures the heap debugger to detect if the program references a block after it has been freed.

Entering

    heapdebug common_errors=uninitialized_field

or clicking the Uninitialized Field button in the graphical user interface, configures the heap debugger to detect if the program reads from a block that it has failed to initialize.

Once you have selected a level or a common error, you may then make more detailed customizations if you wish. The heap debugging level does not affect error control, fill byte values, heap size, internal checks or slop.

Fences

The heap debugger can set "fences" on either end of each block. The fence before the beginning of the block is called the pre-fence. The fence after the end of the block is called the post-fence. The fence bytes are filled with a specified fill pattern. During a heap check, or when the block is freed, the heap debugger checks that the fence bytes have not been altered. If the fence bytes have been altered, it is an indication that your program is writing outside the block, and the heap debugger stops your process with an error status.

As a special case, blocks allocated with a size of zero have no pre-fence, no post-fence, and no slop.
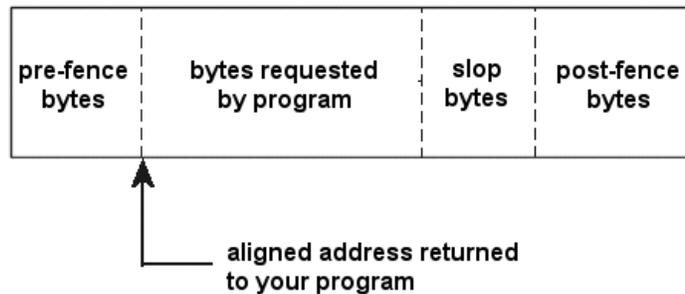
This figure shows the layout of heap debugger overhead in a block. The pre-fence, slop, and post-fence are optional. See "`slop`=n" on page 6-59.

Hardware Overrun Protection

The heap debugger can also set up hardware overrun protection. When enabled, each block is placed as close as possible to the end of a page, and the following page is protected from reads and writes. Then if your program tries to read or write past the end of the block, it gets a signal (`SIGSEGV`). Note that this is not reported as a heap error in the same way that errors appear in the **info memory** report (see "info memory" on page 6-158) or in a data panel.

The advantage of hardware overrun protection is that it catches stray references immediately, which makes it easier to find bugs, and it catches both reads and writes. The main disadvantage of hardware overrun protection is that it uses a great deal of overhead memory in order to position each block. (The protected page takes up virtual memory in your address space, but does not use any physical memory or system swap space. However, the overhead to position the block does take system memory.) It is possible for your program to exhaust the system's memory. You may need to adjust your "`ulimit -v`" setting or talk to your system administrator about increasing the system configuration variables `vm.max_map_count` and `vm.overcommit_ratio`.

This option is useful for small programs, and for large programs that overrun blocks before they have allocated a lot of memory. This option is also useful if you enable it dur-

ing only part of the process's execution so that only particular blocks are protected. See "Heappoints" on page 3-12.

Note also that the heap debugger cannot place every block such that it ends right at the end of a page. The beginning of each block is aligned to an 8-byte boundary on IA-32 and a 16-byte boundary on AMD64. (For some allocation calls, your program may specify a larger alignment.) If the block's size is not a multiple of this alignment, then there will be a gap of a few bytes before the protected page. In this case, the program will not get the SIGSEGV unless it strays outside the block further than the gap.

When hardware overrun protection is turned on, the heap debugger automatically fills the gap with post-fence fill bytes, to help catch stray references into the gap. The gap is never made wider than it has to be to accommodate the alignment and size restrictions. For hardware overrun protection, the number of post-fence bytes is 8 on IA-32 (16 on AMD64), or the number you specify, whichever is larger, but no more than will fit in the gap.
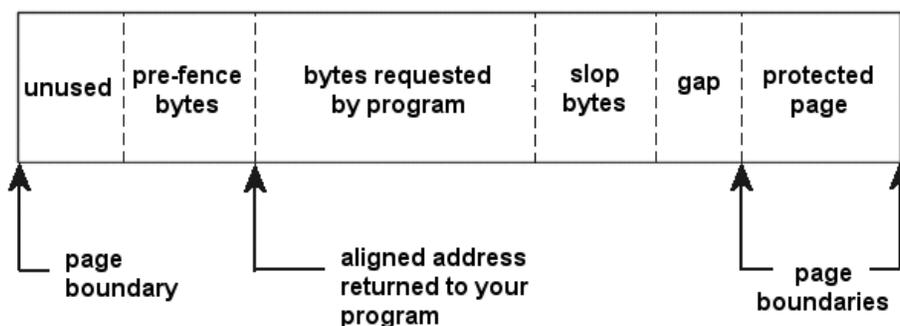
This figure shows the layout of heap debugger overhead in a block with hardware overrun protection. The pre-fence and slop are optional. See "slop=n" on page 6-59. The gap may be zero sized; otherwise it typically is filled with the post-fence fill byte.

Retained Free Blocks

You can configure the heap debugger to retain some number of free blocks, or even all free blocks. A retained free block is not available for reuse. Once the desired number of retained free blocks is reached, blocks are made available for reuse in the same order they were freed by your program.

Retaining free blocks can help you find dangling pointer bugs. This is particularly effective when hardware overrun protection also is used, because when your program frees a block it is protected from reads and writes. However, note that retaining a large number of free blocks can use a lot more memory than your program would use normally, especially when you also use hardware overrun protection.

Heap Check

Heap checks are scans of allocated blocks, and possibly some free blocks (see "Retained Free Blocks" on page 3-33), looking for errors that could not be detected immediately as they occurred. The possible errors are:

- free-fill modified

- post-fence modified

- pre-fence modified

A heap check will be performed in the following circumstances:

- repeated automatically after a specified number of heap operations

- when the **heapcheck** command with no expression is issued (see "heapcheck" on page 6-165)

- when selecting the Heap Errors… item from the Debug Display Menu, and selecting either:

    - Check Heap For New Errors First

    - Check Heap For All Errors First

- when selecting one of the following menu items from the Data Item Popup Menu:

    - Check Heap and Report New Errors

    - Check Heap and Report All Errors

- at a user-specified location with a **heappoint check** (see "heappoint" on page 6-107)

In addition, a limited heap check, pertaining only to a single block, is performed in the following circumstances:

- automatically when free or realloc is called on a block

- when the **heapcheck** command with an expression is issued

- when selecting the Update Block Errors menu item from the Data Item Popup Menu

Leak Detection

Programs sometimes "leak" blocks, which wastes memory. If NightView's heap debugging functionality is turned on with the **heapdebug** command (see "heapdebug" on page 6-54) or from the Process menu (see "Process Menu" on page 8-8), then it is possible to obtain a report of leaked heap blocks. See "info memory" on page 6-158 or "Leak Sets / Still Allocated Sets Data Items" on page 8-72.

An allocated block is considered leaked if no pointer in your program references it. The means of detecting leaks generally is conservative. It does not distinguish pointers from

other data types. So if, for example, an integer is encountered which happens to have a value equivalent to that of a pointer to a leaked block, then leak detection will believe that it is a pointer to the leaked block, and therefore will not consider the block leaked.

Despite this conservatism, there are ways to disguise pointers which will cause leak detection to believe that a heap block is leaked when it really is not. Typically this arises when a pointer to a heap block is stored in some non-standard format. This could happen if a pointer is stored in big endian format on a little endian machine, or vice versa, or if a pointer is marshalled or pickled, and the original unmarshalled or unpickled pointer is destroyed. In practice, these situations occur rarely.

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 3-36.

`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 6-65).

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.

on restart command streams are an exception to this rule. They continue to execute even if the commands get errors. See "on restart" on page 6-48.

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 the process toolbar. See Chapter 8 [Graphical User Interface] on page 8-1. See "Process Toolbar" on page 8-20.

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 3-28.

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 6-170 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 6-66) 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 watch-point, 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 6-34).

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 6-60).

Value History

NightView keeps the results of the **print** command (see "print" on page 6-85) 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 6-155), or use previous history values in new expressions (see "Special Expression Syntax" on page 6-4).

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 7-2. See "Command Toolbar" on page 8-20.

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 3-36.

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 6-141).

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 5 [Invoking NightView] on page 5-1.

Optimization

There are some problems associated with debugging optimized code. 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.

  Concurrent compilers generate additional debug information that indicates where variables are (i.e., register or stack) at different locations in your program. NightView uses this information to access the variables when their location(s) contain accurate values, and to prevent you from accessing them when no location contains an accurate value.

In general, you must be alert to the possibility that the compiler has changed things in your program. It may be easier to debug if you temporarily compile your program without optimization, provided your bug is still reproducible in that case.

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 6-76.

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 6-14. 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 6-124.

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 6-133.

Multithreaded Programs

NightView gives you facilities for debugging threads and Ada tasks. On Linux, threads are implemented with separate "processes" that share resources, including memory, file descriptors, etc. In this manual, those processes are referred to as thread processes, or more simply, threads. When this manual refers to a process, that means all the threads together. See **pthread_create(3)**.

When a thread or Ada tasks stops, NightView stops the entire process (i.e. all other threads that are still executing are stopped). NightvView shows you the thread that caused the process to stop -- known as the current thread. To see other tasks or threads, use the **select-context** command (see "select-context" on page 6-138). When using the graphical interface, all threads are shown in a tree display in a data panel. The "current" thread is shown in green.

NightView provides you some control over the execution of a task or thread independently of the others in that process. When you resume execution (see "resume" on page 6-123), you can choose whether all threads are allowed to execute, or just a single thread. The behavior is dependent on the run mode (see "set-run-mode" on page 6-121) and the options provided to the resume command. The run mode controls whether a single threads resume execution or all threads resume execution. The mode is consulted when you use the graphical interface to cause execution to resume, or when you use similar commands without explicit options that override the mode.

Similarly, if you issue a single-step command (see "step" on page 6-124), the selected task or thread will be stepped according to the command, but the other threads may also

execute during that time — depending on the run mode or the options you use with the single-step command. The next command works similarly (see "next" on page 6-126).

All the commands that cause execution to resume have the following options: **/all** and **/one**. The former indicates that all threads should resume, regardless of the run mode. The **/one** option indicates that only the current thread will resume execution, regardless of the run mode.

It is worth noting that the command **step /all** does **not** single step all threads. It single steps the current thread and all other threads are allowed to execute during the single step and they are stopped when the current thread's single step completes. The command **next /all** operates in the same manner -- it does **not** imply that all threads step over one line of code.

IMPORTANT

> The **/one** option should be used with care. It is not uncommon for a single thread to block waiting on operations to be completed by other threads (e.g. by **pthread_cond_wait(3)** and many other services that hold resources). If those other threads are not permitted to resume, the one thread that was resumed may block indefinitely. In such circumstances, stop the thread and resume all threads using **/all**.

Thread Tags

NightView provides the ability to associate "tag variables" with individual threads in your program. An artificial record with the reserved convenience variable name **$thr** is created for any threads in which you set thread specific tag values. The tag values are fields in the **$thr** struct and each thread gets a unique **$thr** struct.

When new tags are created, the **$thr** structs are automatically reallocated to hold the new fields, as required.

The tag values are typically created with the special **tags** syntax of the **patchpoint** command, but tags may also created by simply using the **$thr** record in an expression and refering to some record field. Any thread which executes through the patch code gets the new tag values set in that thread's **$thr** struct. Eventpoints can also test values in the **$thr** struct in the eventpoint condition, so thread tags provide a way to make eventpoints conditional on which threads execute them.

Any tags created without an explicit declaration are always one bit boolean flags with a default value of *false*. The **declare-thread-tag** command may be used to declare tags of other types (but not all types are allowed: Tag types have to be fixed size types with no dynamic components). The default value of tags is always all zero bits.

Tag values are shown in the data panel thread descriptions as well as the **info threads** command output, so setting unique tag values where unique kinds of threads are created in your program serves as a way to mark threads to easily locate different threads in your program. If a program uses general purpose "worker bee" threads that are dispatched on

different tasks as there are things to do, you can set a **tags** patchpoint in the thread dis-patching code to mark the thread with a tag that says what it is doing while it is at work, then clear that tag when it goes back into the pool of worker bees. To keep the output uncluttered, only tag values with non-zero values are printed, and if you declare tags with complex aggregate types, those tags are not printed in the thread descriptions. You can always say **print $thr** to see the entire set of tags for the current thread.

This is all implemented behind the scenes by calling various library routines such as `pthread_getspecific` and `pthread_setspecific` as well as `malloc` and `realloc`, so tags will only function in programs linked with the normal pthread and libc libraries and using the pthread threading model. This also means that tags do cause a small amount of time and memory overhead to be added to your program (but not much more than any other kind of eventpoint adds). Of course, if you declare lots of tags, or use large types for tag values, the memory usage can be arbitrarily large. Also note that you should never count on the address of **$thr** to remain fixed. If you define new tags, NightView may need to realloc the space for the **$thr** struct and copy the values to a new location.

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.

Note that programs run using the **shell** command (see "shell" on page 6-141) 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 and group as the dialogue in which the **attach** command was issued, or, if a qualifier was specified, as the dialogue in the qualifier. 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. However, the root user can 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) or by clicking Resume in NightSim.

NightTrace Daemon

The **tracepoint** command (see "tracepoint" on page 6-104) can be used to trace variables in a process. Tracing only works if a NightTrace daemon has been started prior to adding tracepoints to the process.

It is the responsibility of the user to start a NightTrace daemon, using either the command line tool **ntraceud(1)** or the Daemons dialog in the tool **ntrace(1)**. See the *NightTrace User's Guide*.

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). The debugger may access memory using 8-bit, 16-bit, or 32-bit references. 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 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 blocked interrupts).

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(3)** 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 6-44. 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
Tutorials

This chapter is divided into two sections.

The first is a general graphical tutorial which introduces the very basic functions of Night-View and symbolic program debugging in general.

The second portion is devoted to short tutorials that concentrate on a single feature or concept. These currently include the following.

- The "Thread Tags Tutorial" on page 4-28 discusses the advantages of using Thread Tags when debugging multi-threaded processes.

- The "Tracing Tutorial" on page 4-34 discusses inserting NightTrace trace events into user code for timing analysis.

General Graphical Tutorial

This is the general 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 8 [Graphical User Interface] on page 8-1. There is a much shorter general tutorial in Chapter 2 [A Quick Start - GUI] on page 2-1. Additionally, there are short topical tutorials in the second half of this chapter, beginning on page 28.

About the Tutorial

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.

The source files used in this tutorial are found under the `/usr/lib/NightView/Tutorial` directory in tar files, and include the following:

C	Fortran	Ada	
msg.h	–	–	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
`-`	`ftint.c`	`-`	Provides Fortran interfaces to system services

This tutorial takes significant time to complete and since each section must be performed in order, it's not easy (or wise) to skip to later sections. If you want a short general tutorial, you may want to see Chapter 2 [A Quick Start - GUI] on page 2-1.

RECOMMENDATION

Perform each step as indicated, or your results may differ from those provided in later steps of the tutorial.

Since this is a live debugging session and NightView is supported on many Linux distributions, the process IDs and hexadecimal program addresses will likely differ from those shown in the tutorial. Additionally, the line breaks in your output may differ from those shown because the lengths of displayed data items may vary.

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

Some of the commands that appear in this tutorial are an abbreviated form of the corresponding canonical NightView command. You may abbreviate NightView commands and some keywords to the shortest unambiguous prefix. For more information, see "Command Syntax" on page 6-1. You cannot abbreviate file names, symbolic names, or NightView construct names.

You could run this entire tutorial with commands and operations from the keyboard. However, use the mouse whenever possible during this tutorial.

Creating the Program

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

```
mkdir nview
cd nview
```

Decide what language program you want to debug.

Make the `msg` program contain debug information. For the Fortran program, you should also build the `ftint.c` interface, but, for this tutorial, do *not* build it with debug information.

-

- For C, you should enter the following:

```
tar xvf /usr/lib/NightView/Tutorial/C.tar
cc -g -o msg *.c
```

- For Fortran, you should enter:

```
tar xvf /usr/lib/NightView/Tutorial/Fortran.tar
cc -c ftint.c
cf77 -g -o msg  ftint.o *.f
```

- For MAXAda, you should enter:

```
PATH=$PATH:/usr/ada/bin
tar xvf /usr/lib/NightView/Tutorial/Ada.tar
a.mkenv  -g -i
a.intro *.a
a.partition  -create active  -main main msg
a.build  -v msg
```

You should now have a `msg` program with debug information in your `nview` directory.

Starting NightView

- Enter the following command.

```
nview
```

Note that in this tutorial `msg` does not appear on the **nview** invocation line, although NightView does accept program invocations and program arguments on the command line.

Shells in NightView

NightView communicates with a system through a dialogue which contains a shell where you run shell commands and debug running programs. For information about dialogues, see "Dialogues" on page 3-4.

The shell is always present but isn't shown by default. If you're just debugging a single program and don't need to provide terminal-oriented input to it, then you don't really need to have a shell panel visible -- since all output the program generates will also be shown in the NightView Message panel.

In most debug sessions, you simply put the program to debug and any arguments it needs on the **nview** command line, or, run the process using the Process menu's Run… option. A shell panel is useful when you need to debug multiple programs or perhaps need to redirect program output or input.

In this tutorial, for demonstration purposes, we will use a shell panel. We will create a shell panel on a separate page.

- Click on the View menu and select Add Page.

A new tabbed page appears with the name Page 2. The old page now shows a tab with the name Main.

- Click on the View menu and select New Shell Panel.

We'll talk about what is displayed in the shell panel in a minute, but it is important to understand the distinction between a shell panel and a shell. The shell always exists,

whereas a shell panel is simply a view of the existing shell that allows you to interact with it, just as if you were interacting with a shell in a terminal session.

In a single NightView session, you can create additional shells; normally you do this when debugging a process on a remote system. When multiple shells exist, NightView always has you decide which one you mean when you choose to create a shell panel for it, or attach to a process from it.

Each shell has a name; the default shell is `local`. The page now has a panel with the title `local shell`. For more information about shell panels, see "Shell Panel" on page 8-60.

The shell panel we created in the step above displays any output generated from your shell's initialization routine (e.g. .bash_init or .kshrc) as well as the following text:

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

NightView runs the **ReadyToDebug** program automatically as part of its initialization. It does this so it knows when your shell's initialization is complete and when it should start considering processes spawned by the shell as processes it should debug.

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 3-9. Note that in this tutorial the dialogue shell prompt is "$ ". Yours may differ.

Getting Help

NightView has an integrated help system. You can get help on a widget, a command, or get help on the last error that occurred. Use the Help menu to access help, or type the following command:

help

Starting Your Program

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.

- If the current tabbed page shown is not Page 2, switch to it by clicking on its tab.

- Click in the shell panel to give it the keyboard focus.

- In the shell panel, you should enter the following command and press Return:

  ./msg

- Switch back to the other page by clicking on the tab labeled Main.

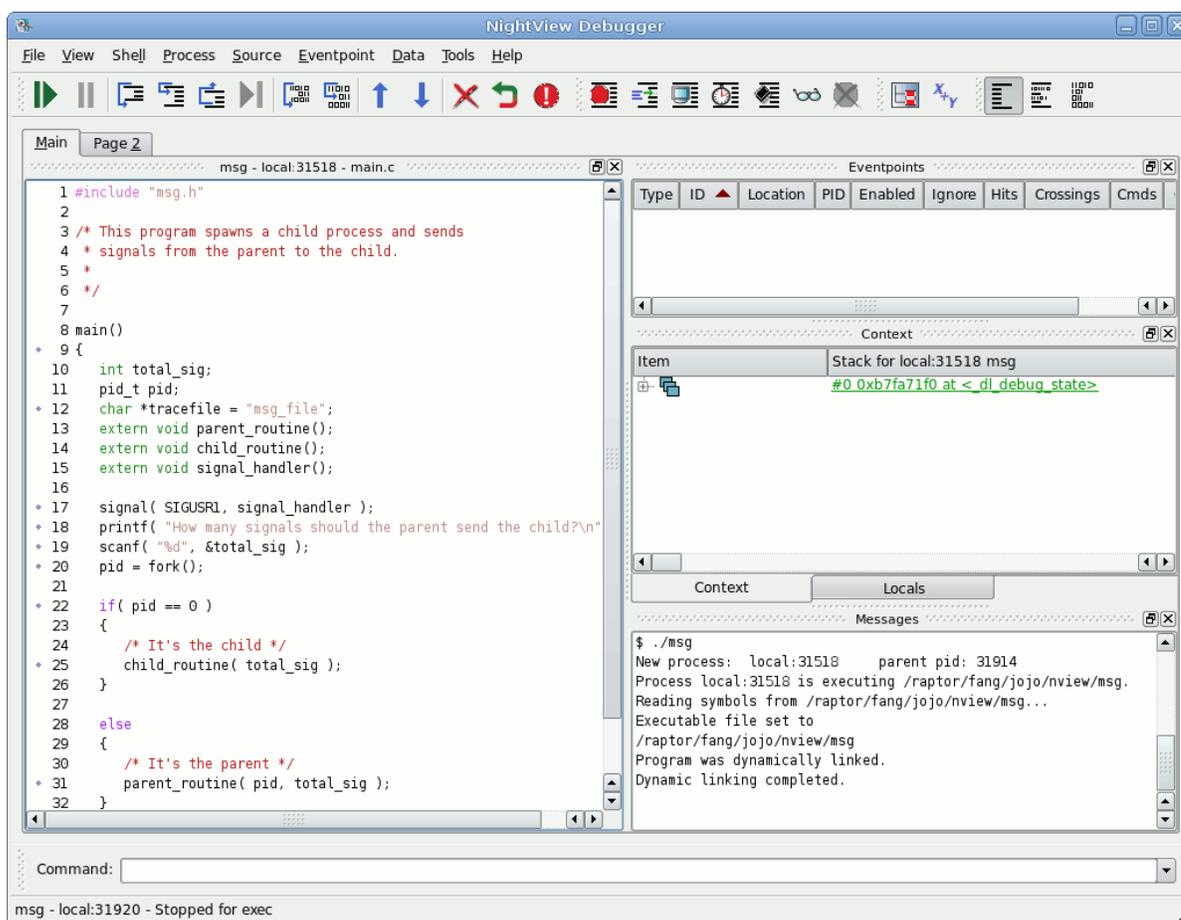NightView should look similar to the following figure:

Figure 4-1. General Tutorial - NightView after program launch

The status bar at the bottom of the window shows that msg is the executable program the process is running.

The message panel shows:

```
New Process:  local:  31518  parent pid:  17882
Process local:31518 is executing /raptor/fang/jojo/nview/
msg.
Reading symbols from /raptor/fang/jojo/nview/msg...done
Executable file set to
/raptor/fang/jojo/nview/msg
```

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

```
Program was dynamically linked.
Dynamic linking completed.
```

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 (in fact, if they don't, you should go by a lottery ticket immediately). For information about processes, see "Programs and Processes" on page 3-2.

The source panel title bar shows the program being debugged, `msg`, the qualifier, `local:31518`, and the name of the source file that is being displayed in the source panel, `main.c`, `main.f`, or `main.a`.

In the source panel, NightView displays numbered source lines. Executable lines have a small blue diamond ◆ source line decoration beside the line numbers.

For more information about source line decorations, see "Source Line Decorations" on page 6-82. The vertical and horizontal scroll bars in the source panel let you examine the rest of the source file.

The status bar shows the status Stopped for exec. This means that the process has just **exec(3)**'ed a new program image.

The context panel has an entry for this process. The header shows the qualifier, `local:31518`, and the name of the program this process is running, `msg`. The context panel entry is for the current stack frame, which is in a start-up routine that gets control before `main`. Later, we will see process entries here, but for now there is only one process, and the context panel shows process entries only if there is more than once process. See "Context Panel" on page 8-64.

Note that by appending an ampersand (`&`) to the `./msg`, you could have started your program in the background of the 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 in the shell panel, or by clicking on the Rerun button in the process toolbar.

Debugging All Child Processes

By default, NightView debugs child processes only when they have called **exec(3)**. In the `msg` program, the child process never calls `exec`. To be able to debug this child process, you must tell NightView to debug children *before* `msg` forks the child process. Also, you have tell NightView to debug children *after* invoking `./msg` so this setting can be applied to existing processes. See "Multiple Processes" on page 3-2.

- Click on the Process menu and select the Process Settings… entry.

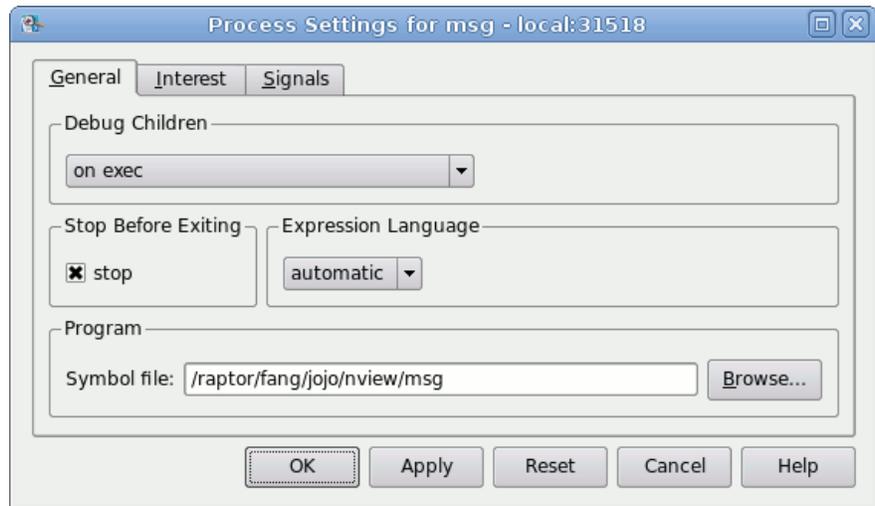**Figure 4-2. General Tutorial - Process Settings General Page**

- In the General page, in the Debug Children area, click on the combo box and select always.

- Click on the OK button to apply the change and dismiss the dialog box.

NightView echoes a **set-children** command in the message panel.

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.

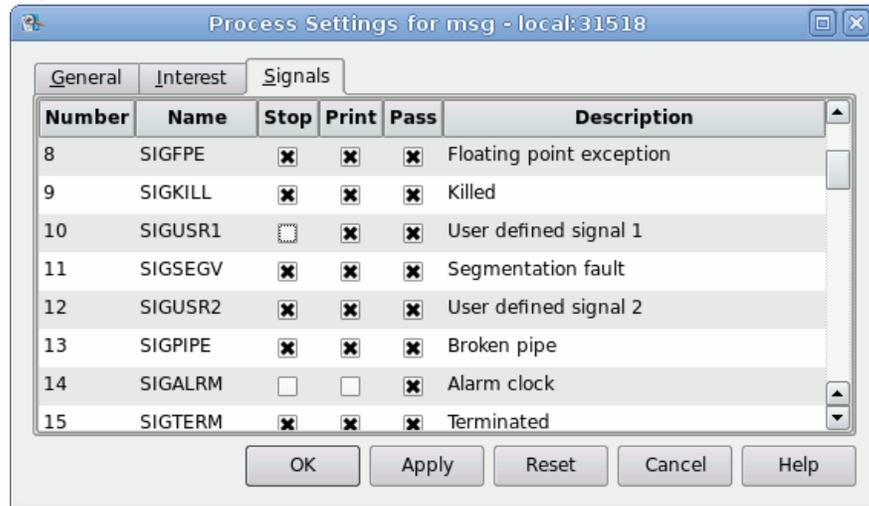- Click on the Process menu and select the Process Settings… entry, and then click on the Signals tab.

Figure 4-3. General Tutorial - Process Settings Signals Page

- Scroll down to the entry for SIGUSR1. Turn off the checkbox for Stop, but leave the checkboxes set for Print and Pass.

- Click on the OK button to apply the change and dismiss the dialog box.

NightView echoes a **handle** command in the message panel.

Note: you had to change the signal settings *after* invoking ./msg so they could be applied to existing processes.

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.

- Right click on the line 18 line in the source panel.

Right clicking brings up the context menu for the source panel. Clicking on the particular line identifies it as the target for the subsequent action. This is indicated by changing the background color of the line.

- Select Set simple breakpoint in the context menu. Repeat this for the other breakpoints as indicated below.

For the C program, the lines are 18, 25, and 30. NightView displays the following information in the message panel.

```
local:31518 Breakpoint 1 set at main.c:18
local:31518 Breakpoint 2 set at main.c:25
local:31518 Breakpoint 3 set at main.c:30
```

For the Fortran program, the lines are 18, 26, and 28. NightView displays the following information in the message panel.

```
local:31518 Breakpoint 1 set at main.f:18
local:31518 Breakpoint 2 set at main.f:26
local:31518 Breakpoint 3 set at main.f:28
```

For the Ada program, the lines are 18, 25, and 27. NightView displays the following information in the message panel.

```
local:31518 Breakpoint 1 set at main.a:18
local:31518 Breakpoint 2 set at main.a:25
local:31518 Breakpoint 3 set at main.a:27
```

An *eventpoint* is a generic term which includes breakpoints, patchpoints, monitorpoints, tracepoints, and heappoints. 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 message panel has the number of the line you clicked on.

The Eventpoints panel will look similar to the following figure:

Type	ID ▲	Location	PID	Enabled	Ignore	Hits	Crossings	Cmds	Condition
🔴 Break	1	main.c line 18 in main()	31518	☒	0	0	0	No	
🔴 Break	2	main.c line 25 in main()	31518	☒	0	0	0	No	
🔴 Break	3	main.c line 31 in main()	31518	☒	0	0	0	No	

Figure 4-4. General Tutorial - Eventpoint Panel

NightView changes the source panel when you set a breakpoint. Note that each line with a breakpoint on it now has a stop sign 🔴 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.

- Click on the Resume button. ▮▶

The status bar shows the status Stopped at breakpoint 1. This means that the process hit breakpoint number 1.

NightView changes the source line decoration on the statement with the breakpoint to a stop sign overlaid with a green triangle pointing to the right 🔴 . The stop sign still indicates a breakpoint, and the triangle indicates that execution is stopped there.

For the C program, NightView displays the following in the message panel:

```
local:31518: at Breakpoint 1, 0x10002818 in main() at
main.c line 18
```

For the Fortran program, NightView displays the following in the message panel:

```
local:31518: at Breakpoint 1, 0x10003878 in main() at
main.f line 18
```

For the Ada program, NightView displays the following in the message panel:

```
local:31518: at Breakpoint 1, 0x10010b18 in main() at
main.a line 18
```

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.

- Click on the Next button.

In the source panel, NightView changes the source line decoration of the next line to a green triangle pointing to the right, which shows that execution is stopped there.

The status bar shows the status Stopped after step. This means that the process has finished a stepping command.

- Switch to Page 2.

The msg program writes the prompt "How many signals should the parent send the child?" in the shell panel.

Entering Input

You must respond to the msg program prompt "How many signals should the parent send the child?".

Remember that you may need to click in the shell panel to put the keyboard focus there.

- In the shell panel, you should enter:

  10

and press Return.

- Switch back to the first page by clicking on the tab labeled Main.

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.

- Click on the Resume button.

The status bar shows 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 message panel:

```
local:31518: at Breakpoint 3, 0x1000284c in main() at
main.c line 31
```

For the Fortran program, NightView displays the following in the message panel:

```
local:31518: at Breakpoint 3, 0x10003904 in main() at
main.f line 29
```

For the Ada program, NightView displays the following in the message panel:

```
local:31518: at Breakpoint 3, 0x10010bdc  in main() at
main.a line 28
```

The source line decoration is now a stop sign overlaid with a triangle pointing to the right. The stop sign still indicates a breakpoint, and the triangle indicates that execution is stopped there.

The context panel has a new entry for the child process, as shown in the following figure.
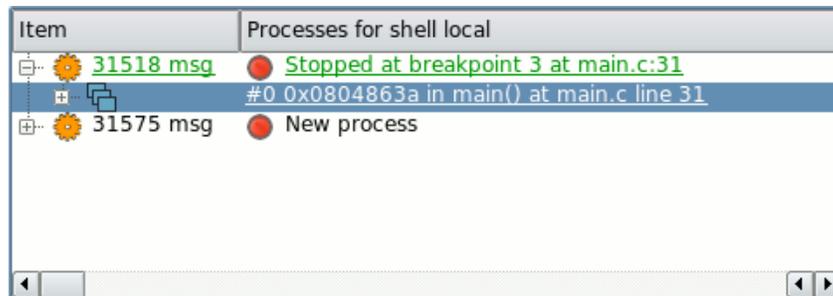

Figure 4-5. General Tutorial - Context Panel

The child process is the one with the status New Process. The parent process, 31518, is the *current process*, shown with green text. Buttons, menus and commands generally apply to the current process.

You would like to view the child process as the current process.

-   In the context panel, click on the entry for the child process.

Now the source panel is displaying the child process.

The status bar still shows that msg is the executable program the current process is running. (The child is executing the same program as the parent.) The qualifier in the status bar now shows the qualifier of the child process.

NOTE

If your system has debug information installed for system librar-
ies, the process may appear to be stopped in the `fork()` library
routine. If so, click the Up button ⬆ until the debugger reports
that the process is in `main`.

For the C program, the message panel shows:

```
New process:  local:31575    parent pid: 31518
#0  0x10002838  in main() at main.c line 20
```

For the Fortran program, the message panel shows:

```
New process:  local:31575    parent pid: 31518
#0  0x100038e4  in main() at main.f line 22
```

For the Ada program, the message panel shows:

```
New process:  local:31575    parent pid: 31518
#0  0x10010bc8  in main() at main.a line 21
```

In this example, the child process has process ID 31575, and the parent process has process
ID 31518. 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 panel 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, a gray (rather than green) triangle pointing to the right, indicates that this line
made a subprogram call which has not yet returned. ▷

The status bar 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 status bar shows the qualifier, `local:31575`.

The context panel lists entries for processes 31518 and 31575.

Catching up the Child Process

We want to get the **child** process to continue execution up to the breakpoint on the call to
`child_routine` (line 25 in `main.c`, line 26 in `main.f`, and line 25 in `main.a`).

- With the *child* as the current process, click on the Resume button.

⏩

For the C program, NightView displays in the message panel:

```
local:31575: at Breakpoint 5, 0x10002840 in main() at mai
n.c line 25
```

For the Fortran program, NightView displays in the message panel:

```
local:31575: at Breakpoint 4, 0x100038fc in main() at mai
n.f line 26
```

For the Ada program, NightView displays in the message panel:

```
local:31575: at Breakpoint 4, 0x10010bd0 in main() at mai
n.a line 25
```

The debug source file name is `main.c` or `main.f` or `main.a`.

NightView puts a source line decoration of a stop sign overlaid by a green triangle pointing to the right ![stop sign icon] in the source panel on line 25 for the C and Ada programs and line 26 for the Fortran program.

The status bar shows 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

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

- In the source panel, 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. ![print icon]

NightView displays in the message panel:

```
$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 3-38.

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 3-39. In the Fortran program, `total_sig` was put in COMMON so you could consistently see its value in the tutorial.

Listing the Source

We want to look at the source code for `child_routine`.

- Switch to the **parent** process by clicking on the parent process's entry in the context panel. (The parent has the status Stopped at breakpoint 3.)

- Click on the Source menu, and select List Function/Unit….

After clicking on the parent process, the status bar shows `Stopped at breakpoint 3`. The source panel 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.

- In the Select a Function/Unit dialog box, you should enter `child_routine` as the regular expression, and click on the Filter button. (For more information about regular expressions, see "Regular Expressions" on page 6-18.)

NightView finds the `child_routine` function and puts it in the list.

- In the Select a Function/Unit dialog box, you should click on the Select button.

NightView closes the Select a Function/Unit dialog box.

The title bar of the source panel changes the file name to `child.c`, or `child.f`, or `child.a`, and the source panel shows the source code.

Entering Functions

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

- In the command toolbar, enter the following command:

    (all) step

(Remember, you may need to click in the command toolbar to get the keyboard focus to be there.)

Note that if you had wanted to enter a routine in only one process, you could have qualified the **step** command with the process ID, or you could have made the process the current process before entering the command.

Because you used the `(all)` qualifier, the **step** command causes both processes to step.

For the C program, NightView displays in the message panel:

```
#0  0x10002884  in child_routine(int total_sig = 10) at c
hild.c line 14
#0  0x10002944  in parent_routine(pid_t child_pid = 31575
, int total_sig = 10)
                at parent.c line 11
```

For the Fortran program, NightView displays in the message panel:

```
#0  0x1000393c  in child_routine() at child.f line 17
#0  0x10003a48  in parent_routine(INTEGER child_pid /
 31575 / )
                at parent.f line 16
```

For the Ada program, NightView displays in the message panel:

```
#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 =
31575,
                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 process entries in the context panel show the status Stopped after step. This means that the processes have finished a stepping command. The status bar shows the same status for the parent process.

The source file name in the title bar of the source panel changes to parent.c, or par-ent.f, or parent.a, and the source panel shows the source code. Alternatively, the current context could be that of the child, in which case child.c, or child.f, or child.a would be shown.

Line 11 of parent.c, or line 16 of parent.f, or line 6 of parent.a in the source panel has the source line decoration of a green triangle pointing to the right, which indicates that execution is stopped there.

Examining the Stack Frames

Observe the context panel. You may need to scroll and to expand the frames in the child process by clicking on the box with the **+** sign: There are entries for the two processes. Under each process are entries for the stack frames.

For the C program, NightView displays in the context panel:

```
31518 msg  Stopped after single step
  #0  0x10002944  in parent_routine(pid_t child_pid =
    31575, int total_sig = 10) at parent.c line 11
  #1  0x10002854  in main() at main.c line 31
  #2  0xb7dd5879 in __libc_start_main(...
31575 msg  Stopped after single step
  #0  0x10002884  in child_routine(int total_sig =
 10) at
    child.c line 14
  #1  0x10002848  in main() at main.c line 25
  #2  0xb7dd5879 in __libc_start_main(...
```

For the Fortran program, NightView displays in the context panel:

```
31518 msg  Stopped after single step
  #0  0x10003a48  in parent_routine(INTEGER child_pid /
    31575 / ) at parent.f line 16
  #1  0x10003910  in main() at main.f line 29
31575 msg  Stopped after single step
  #0  0x1000393c  in child_routine() at child.f line 17
  #1  0x10003900  in main() at main.f line 26
```

For the Ada program, NightView displays in the message panel:

```
31518 msg  Stopped after single step
  #0  0x10010578  in parent_routine(child_pid : IN pid_t
    = 31575, total_sig : IN integer = 10) at parent.a
    line 6
  #1  0x10010be4  in main() at main.a line 28
  #2  0x10022750  in <environment>() at <environment>
(try -elab_src link option) line 78
31575 msg  Stopped after single step
  #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 <environment>() at <environment>
(try -elab_src link option) line 78
```

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 the name <environment>. If you use the -elab_src link option when building the Ada program, the compiler constructs a pseudo-source file for the elaboration routine so you can debug through each of its statements.

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`.

- Click on the entry for `parent_routine` in the parent process in the context panel.

- In the process toolbar, you should click on the Up button. ⬆

The file name in the source panel title bar changes to `main.c`, `main.f`, or `main.a`, and the source panel shows the source code.

For the C program, NightView displays in the message panel:

```
Output for process local:31518
#1  0x10002854  in main() at main.c line 31
```

For the Fortran program, NightView displays in the message panel:

```
Output for process local:31518
#1  0x10003910  in main() at main.f line 29
```

For the Ada program, NightView displays in the message panel:

```
Output for process local:31518
#1  0x10010be4  in main() at main.a line 28
```

The source panel for the C program is shown in the following figure:

```
·········· msg - local:31518 - 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
```

Figure 4-6. General Tutorial - Source Panel for C Program

The source line decoration in the source panel is a gray triangle pointing to the right, which indicates that execution will resume there when the called routine returns. This source line decoration appears on line 34 of main.c and line 29 of main.f, and line 31 of main.a. The source line decoration may appear on different lines depending on which compiler you used.

Verifying Data Values in Other Stack Frames

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

The locals panel may be sharing screen space with the context panel. In that case there will be tabs at the bottom of those panels. Click the one labeled `Locals`.

The Locals panel for the C program is shown in the following figure:

Figure 4-7. General Tutorial - Locals Panel for C Program

For the C program, NightView displays in the locals panel:

```
pid 31575
total_sig 10
tracefile 0x80487a0 "msg_file"
```

For the Fortran program, NightView displays in the locals panel:

```
pid 31575
sigusr1 10
total_sig 10
tracefile "msg_file"
```

For the Ada program, NightView displays in the locals panel:

```
pid 31575 (or cannot retrieve value...)
total_sig 10
tracefile array(1..8) of character
```

For Ada, click the [+] box to expand the array and see the contents.

Returning to a Stack Frame

We want to return to `parent_routine`.

- Click on the entry for `parent_routine` under the parent process in the `Context` panel. This frame becomes the current stack frame.

For the C program, NightView displays in the message panel:

```
Output for process local:31518
#0  0x10002944  in parent_routine(pid_t child_pid =
31575, int total_sig = 10)
                  at parent.c line 11
```

For the Fortran program, NightView displays in the message panel:

```
Output for process local:31518
#0  0x10003a48  in parent_routine(INTEGER child_pid /
31575 / )
                  at parent.f line 16
```

For the Ada program, NightView displays in the message panel:

```
Output for process local:31518
#0  0x10010578  in parent_routine(child_pid : IN pid_t =
31575,
                  total_sig : IN integer = 10) at
parent.a line 6
```

The file name in the title bar of the source panel changes to `parent.c` or `parent.f`, or `parent.a`, and the source panel shows the source code.

The green triangle ▶ source line decoration in the source panel indicates that execution stopped there. This source line decoration appears on line 11 of `parent.c`, and line 15 of `parent.f`, and line 6 of `parent.a`.

Resuming Execution

We 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.

- Switch to the **child** process by clicking on the child process's entry. Then you should click on the Resume button. ▶

After clicking the child process's entry, the file name in the source panel title bar is `child.c`, or `child.f` or `child.a`.

After pressing Resume, NightView disables (dims) most of the buttons in the process toolbar.

Figure 4-8. General Tutorial - Process Toolbar

The status bar status bar shows the status Running. This means that the process is currently executing.

Removing a Breakpoint

Breakpoint 1 (set in "Setting the First Breakpoints" on page 4-8) is no longer needed.

- Right-click on the entry for breakpoint 1 in the eventpoint panel. The context menu appears. Select Delete.

NightView deletes the breakpoint from the eventpoint panel.

Setting Conditional Breakpoints

It is often useful to suspend execution conditionally.

We will 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.

- In the context panel, click on the parent process.

- In the source panel, click on the line containing the print code. For parent.c, it is line 16. For parent.f, it is line 17. For parent.a, it is line 15. You should click on the Eventpoint menu. Select Set Breakpoint….

NightView displays the breakpoint dialog box.
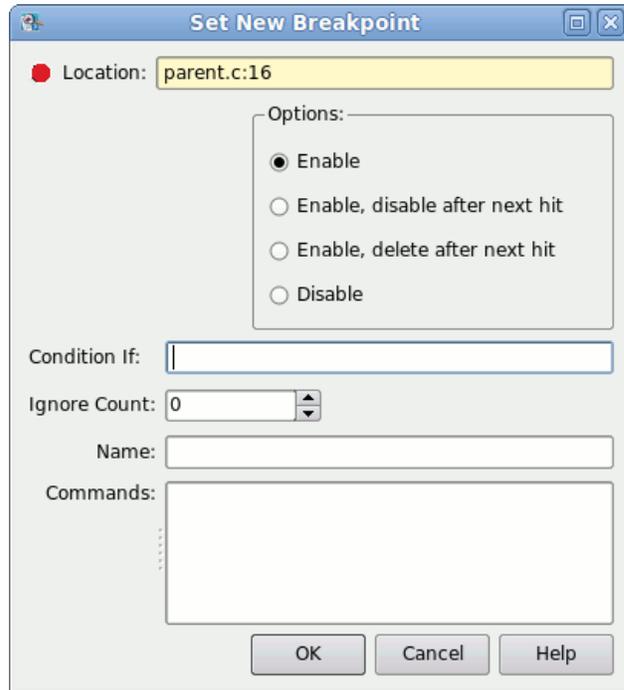
Figure 4-9. General Tutorial - Set New Breakpoint Dialog

Do *not* press Return after you enter the following text.

For the C program, you should enter in the Condition If: text input area:

    isec == total_sig

For the Fortran program, you should enter in the Condition If: 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.

- Click on the OK button.

NightView closes the breakpoint dialog box.

For the C program, NightView displays in the message panel:

```
local:31518 Breakpoint 7 set at parent.c:16
```

For the Fortran program, NightView displays in the message panel:

```
local:31518 Breakpoint 7 set at parent.f:17
```

For the Ada program, NightView displays in the message panel:

```
local:31518 Breakpoint 7 set at parent.a:15
```

The indicated line gets a stop sign source line decoration in the source panel.

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.

We will set a breakpoint on the line that displays how long the parent is sleeping in parent_routine, ignoring the next five iterations.

- In the source panel, click on the line containing the print code. For parent.c, it is line 16. For parent.f, it is line 17. 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: spin box. Do *not* press Return.

You are ready to finish attaching an ignore count to a breakpoint.

- Click on the OK button.

NightView closes the breakpoint dialog box.

For the C program, NightView displays in the message panel:

```
local:31518 Breakpoint 8 set at parent.c:16
```

For the Fortran program, NightView displays in the message panel:

```
local:31518 Breakpoint 8 set at parent.f:17
```

For the Ada program, NightView displays in the message panel:

```
local:31518 Breakpoint 8 set at parent.a:15
```

Attaching Commands to a Breakpoint

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

We will add 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 Break-
point" on page 4-21).

- In the eventpoint panel, you should right-click on the entry for breakpoint
  8. The context menu appears. Select Edit....

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" on page 4-21.

Do *not* press Return after you enter the following text.

- In the Commands: text input area, enter the following command:

  print total_sig

- Click on the OK button.

NightView closes the breakpoint dialog box.

Automatically Printing Variables

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

We will tell NightView to display the value of the sig_ct variable.

- In the source panel, start at one side of any instance of the sig_ct vari-
  able, 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 Data Display button.

A data panel appears, with a line for sig_ct. The value displayed is meaningless,
because sig_ct has not yet been initialized by the program.

Note that the data panel is updated every time execution stops, and the **print** command
from "Attaching Commands to a Breakpoint" on page 4-22 runs only when execution
stops at a specific breakpoint.

Watching Inter-Process Communication

You already resumed the execution of the child process, so NightView did not wait for the child process.

- Click on the Resume button to continue execute of the **parent** process.

In the shell I/O area, NightView responds with something like the following:

```
 1.   Parent sleeping for 2 seconds
 2.   Parent sleeping for 2 seconds
Process local:31575 received SIGUSR1
Child got ordinal signal #1
 3.   Parent sleeping for 2 seconds
Process local:31575 received SIGUSR1
Child got ordinal signal #2
 4.   Parent sleeping for 2 seconds
Process local:31575 received SIGUSR1
Child got ordinal signal #3
 5.   Parent sleeping for 2 seconds
Process local:31575 received SIGUSR1
Child got ordinal signal #4
Process local:31575 received SIGUSR1
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 lines that mention signal SIGUSR1 appear because the signal settings are implicitly set to print and explicitly set to nostop.

The status bar status bar shows 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 message panel:

```
local:31518: at Breakpoint 8, 0x10002950 in
parent_routine(
                    pid_t child_pid = 31575, int total_sig
= 10)
                    at parent.c line 16
$3: total_sig = 10
```

For the Fortran program, NightView displays something like the following in the message panel:

```
local:31518: at Breakpoint 8, 0x105d0 in parent_routine(
                    INTEGER child_pid / 31575 / ) at
parent.f line 17
$3: total_sig = 10
```

For the Ada program, NightView displays something like the following in the message panel:

```
local:31518: at Breakpoint 8, 0x30324 in parent_routine(
                    child_pid : IN integer = 31575,
                    total_sig : IN integer = 10) at
parent.a line 15
$3: total_sig = 10
```

Initial lines show where execution stopped. One 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.

The data panel shows the value of `sig_ct` as 6.

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.

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`.

- Click on the line that displays how long the parent is sleeping, then click on the Eventpoint menu. Select Set Patchpoint….

NightView displays the patchpoint dialog box.
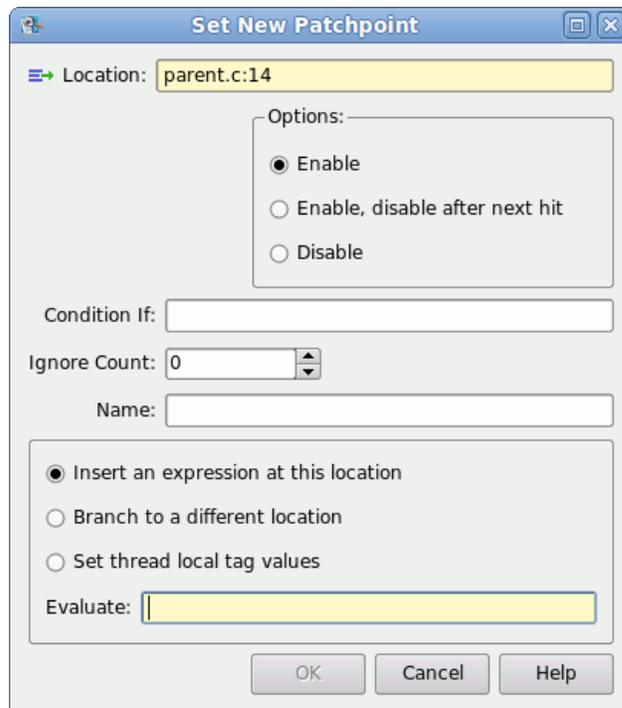
Figure 4-10. General Tutorial - Set New Patchpoint Dialog

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.

  - Click on the OK button.

NightView closes the patchpoint dialog box.

Note that the line in the source panel with a patchpoint on it now has the multiple event-point ⬤ source line decoration, because it now has multiple *kinds* of eventpoints, breakpoint and patchpoint. This is overlaid with the program counter decoration. ⬤

For the C program, NightView displays in the message panel:

    local:31518 Patchpoint 9 set at parent.c:16

For the Fortran program, NightView displays in the message panel:

    local:31518 Patchpoint 9 set at parent.f:17

For the Ada program, NightView displays in the message panel:

    local:31518 Patchpoint 9 set at parent.a:15

Disabling a Breakpoint

We want to run msg to completion without stopping at breakpoint 8.

  - In the eventpoint panel, click the checkbox in the Enabled field of the entry for breakpoint 8 to clear the checkbox.

Examining Eventpoints

We want to examine the types, locations, and statuses of the eventpoints we have set in msg.

Observe the eventpoint panel.

	Type	ID ▲	Location	PID	Enabled	Ignore	Hits	Crossings	Cmds	Condition
🔴	Break	2	main.c line 25 in main()	31518	☒	0	0	0	No	
🔴	Break	3	main.c line 31 in main()	31518	☒	0	1	1	No	
🔴	Break	4	main.c line 18 in main()	31575	☒	0	1	1	No	
🔴	Break	5	main.c line 25 in main()	31575	☒	0	1	1	No	
🔴	Break	6	main.c line 31 in main()	31575	☒	0	0	0	No	
🔴	Break	7	parent.c line 16 in par...	31518	☒	0	1	6	No	isec == total_sig
▷	Break	8	parent.c line 16 in par...	31518	☐	0	1	6	Yes	
⇥	Patch	9	parent.c line 16 in par...	31518	☒	0	0	0	No	

Figure 4-11. General Tutorial - Reviewing Evenpoint Panel for C Program

NightView displays all eventpoints for process `local:31518` followed by the event-points for process `local:31575`.

Breakpoints 1, 2, and 3 were set in "Setting the First Breakpoints" on page 4-8. Breakpoint 1 has no entry because it was deleted in "Removing a Breakpoint" on page 4-19. 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" on page 4-20 and is still enabled.

Breakpoint 8 was set in "Attaching an Ignore Count to a Breakpoint" on page 4-21 and was disabled in "Disabling a Breakpoint" on page 4-25.

Patchpoint 9 was set in "Patching Your Program" on page 4-24 and is still enabled.

Continuing to Completion

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

- If the parent process is not the currently displayed process, switch to it.

- Click on the `Resume` button.

NightView displays in the message panel:

```
6.   Parent sleeping for 1 seconds
7.   Parent sleeping for 2 seconds
Process local:31575 received SIGUSR1
Child got ordinal signal #6
8.   Parent sleeping for 3 seconds
Process local:31575 received SIGUSR1
Child got ordinal signal #7
9.   Parent sleeping for 1 seconds
```

```
Process local:31575 received SIGUSR1
Child got ordinal signal #8
10.  Parent sleeping for 2 seconds
Process local:31575 received SIGUSR1
Child got ordinal signal #9
Process local:31575 received SIGUSR1
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" on page 4-24. The lines that mention signal SIGUSR1 appear because the signal settings are implicitly set to `print` and explicitly set to `nostop`.

Note the order of the displayed lines may vary.

NOTE

> If your system has debug information installed for system librar-ies, the process may appear to be stopped in the `_exit()` library routine. If so, click the Up button ⬆ until the debugger reports that the process is in `main`.

The source panel shows the main program, at the call to `exit`.

The status bar status bar shows the status About to exit. This means that the process has called the exit system service. See "Exited and Terminated Processes" on page 3-18.

NightView displays in the message panel:

```
Process local:31518 is about to exit normally
```

The data panel shows that the `sig_ct` variable is not visible at this point in the parent process.

```
Identifier "sig_ct" is not visible in the given context.
```

Depending on which compiler you used, the value may still be visible.

Leaving the Debugger

- Click on the File menu. Select Exit NightView.

Neither process has completely exited, so NightView puts up a warning dialog box, asking the following question:

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

- In the warning dialog box, click on the OK button.

The main window is removed.

This concludes the general tutorial.

Topical Tutorials

This section contains short tutorials which concentrate on a single topic or feature. A general working knowledge of debugging with NightView is assumed.

Thread Tags Tutorial

This tutorial demonstrates how to use thread tags to enhance the experience of debugging a multi-threaded application.

- Compile and link the `thread.c` file which is located in /usr/lib/Night-View/Tutorial, with commands similar to the following:

```
cat /usr/lib/NightView/Tutorial/threads.c > threads.c
cc -g threads.c -lpthread -lm -lrt
```

NOTE

On some RedHawk Linux systems, if the above command fails to link the program, you may need to specify `-lccur_rt` instead of `-lrt`.

The example program is written in C and creates a `director` thread which doles out work to a set of three `worker_bee` threads.

- Start the program under NightView, using the following command:

```
nview ./a.out
```

- Once NightView starts, add a `Data` panel with a `Threads` display, by using the `Data` menu and selecting the `Threads` option.

- Now let the process run for a bit by issuing the following two debugger commands, separated by a few seconds:

```
resume
stop
```

The program has executed for a few seconds and then stopped. The Threads display in the Data panel will look similar to the following figure:

Item	Value
⊟ ⚙ Threads	local:11566 a.out
⊞ ⚙ 11566	C thread 0x401df6c0
⊞ ⚙ 11570	C thread 0x403e0b90 (director)
⊞ ⚙ 11571	C thread 0x405e1b90 (worker_bee)
⊞ ⚙ 11572	C thread 0x407e2b90 (worker_bee)
⊞ ⚙ 11573	C thread 0x409e3b90 (worker_bee)

Figure 4-12. Thread Tags Tutorial - Threads Automatically Named

For each thread (other than the main program thread), NightView has determined the name of the start routine for the thread and displays that in parentheses.

The name is derived from the start routine address passed to **pthread_create(3)**, which is the first user function that executes when a new thread is created.

In our example program, there is a single director thread and three instances of a worker_bee thread. Using worker threads is a fairly common programming paradigm; the activities of those threads are dynamically assigned by the application as it executes.

In our application, the director thread creates a workload message and queues the message waiting for a worker_bee thread to service the message.

Here's an excerpt from the source of the worker_bee routine (some lines were deleted for brevity).

```
62: for (;;) {
64:    mq_receive(q,(char*)&msg,sizeof(msg_t),&priority);
71:    switch (msg.job) {
72:    case _EXIT:
73:       break;
74:    default:
75:       calc = (*msg.work)(msg.arg);
76:    }
81: }
```

The worker_bee thread receives a message (when one is available) and then starts executing the body of work that the message indicates. In this case, it calls a work function supplied by the message.

It would be convenient to know what job each thread is running whenever we look at the thread display in NightView.

To achieve that, we'll create a thread tag which represents the type of job being executed. Then we'll insert a patchpoint into the `worker_bee` routine to assign the current job to the current thread whenever it is dispatched.

- Issue the following command in NightView:

  declare-thread-tag job jobs

Here we have declared that a thread tag called `job` to be of type `jobs`, which in this case is defined as an enumerated type in the program on line 14.

```
13
14 typedef enum { _Nuthin, _Sine, _Cosine, _Pi, _EXIT } jobs;
15
```

- Click on line 71 in the source panel and then right-click and select the Set eventpoint menu option and the Set Patchpoint option from that sub-menu.

- The Patchpoint dialog appears.
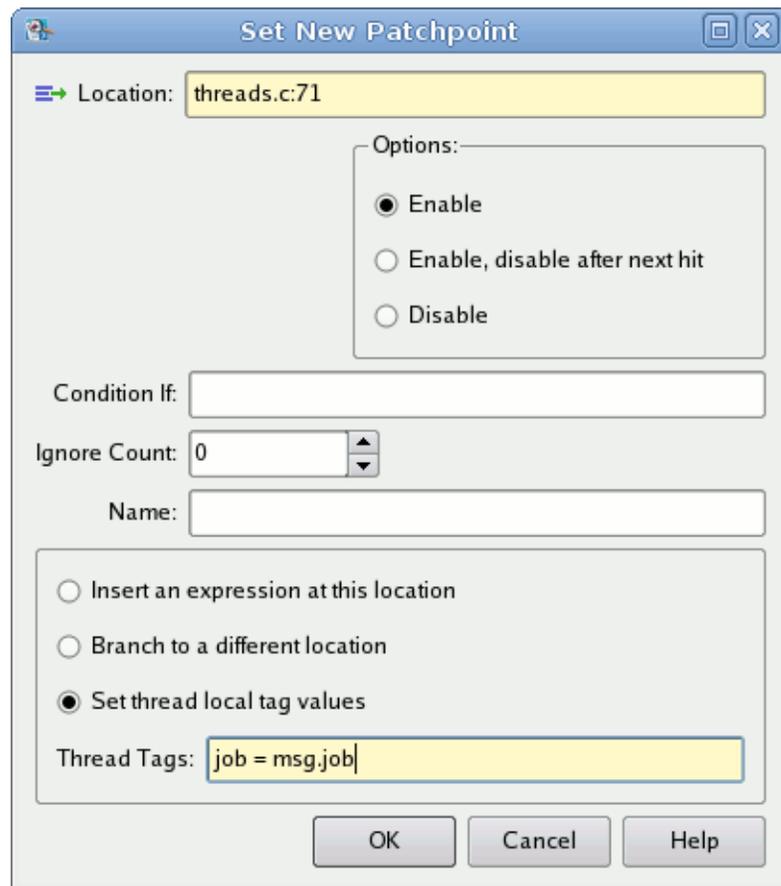
Figure 4-13. Thread Tags Tutorial - Patchpoint Dialog

- Click the Set thread local tags values radio button

- Type the following in the Thread Tags field

```
job = msg.job
```

- Click the OK button to close the dialog

- Resume the process for about five seconds, then stop it again.

When the process stops, the Threads display in the Data panel now includes the value of the thread tag job for all threads where its value is non-zero.

Item	Value
⊟ ⚙ Threads	local:11566 a.out
⊞ ⚙ 11566	C thread 0x401df6c0
⊞ ⚙ 11570	C thread 0x403e0b90 (director)
⊞ ⚙ 11571	C thread 0x405e1b90 (worker_bee), job=_Pi (3)
⊞ ⚙ 11572	C thread 0x407e2b90 (worker_bee), job=_Sine (1)
⊞ ⚙ 11573	C thread 0x409e3b90 (worker_bee), job=_Pi (3)

Figure 4-14. Thread Tags Tutorial - Threads display with job tag

NOTE

Since the directory threads assigns jobs randomly and we are randomly stopping the process, the jobs assigned to each thread will likely differ from the figure above, as will the current thread, which is shown in green.

NOTE

If there is no value shown next to the word job in the Threads display, then perhaps you forgot the **declare-thread-tag** step described above (if you forgot, then the default type of job is boolean; so it only shows up because its value is non-zero -- i.e. true). If so, execute the **declare-thread-tag** command now, remove the patchpoint and then set it again and proceed.

The director thread assigns a priority to each message that it sends; it might be nice to see which threads are executing the most urgent messages. Here we can set a boolean thread tag to indicate this.

Since thread tags are of type boolean by default, we don't need to declare the type of this new tag we're going to create, we can just start using it.

- Issue the following command in NightView:

```
patch threads.c:65 tags urgent = priority > 2
```

- Resume the process and let it execute for a few seconds, then stop it again.

Item	Value
⊟ ⚙ Threads	local:11566 a.out
⊞ ⚙ 11566	C thread 0x401df6c0
⊞ ⚙ 11570	C thread 0x403e0b90 (director)
⊞ ⚙ 11571	C thread 0x405e1b90 (worker_bee), job=_Pi (3)
⊞ ⚙ 11572	C thread 0x407e2b90 (worker_bee), job=_Sine (1)
⊞ ⚙ 11573	C thread 0x409e3b90 (worker_bee), job=_Sine (1) urgent

Figure 4-15. Thread Tags Tutorial - Urgent Processing Shown

The Threads display in the Data panel will now include the word urgent for any thread that is executing an urgent message. Tags whose values are all zero are simply omitted from the list of tags in the Threads display.

NOTE

It is possible you stopped the process when no urgent messages were being serviced. If you don't see the urgent tag in the Threads display, repeatedly resume and stop the process until you do.

- Click on a thread in the Threads display in the Data panel which does not have the urgent tag displayed and then enter the following command:

p $thr

This command prints the values of all thread tags for the current thread, even if their value is zero:

```
$1: $thr = {
    job = _Pi (3);
    urgent = FALSE
}
```

You can also declare thread tags to have descriptive text. In our example, the message received by the worker_bee threads includes a string.

- Issue the following command in NightView:

declare-thread-tag handle char[20]

We've declared that the thread tag handle is an array of 20 characters. However, assigning to strings in C requires us to use a syntax other than a simple assignment (=) operator. As such, an alternative form of patchpoint can be used.

- Issue the following command in NightView:

  `patch threads.c:65 eval strcpy($thr.handle,msg.handle)`

In this case, we used the standard `eval` form of a patchpoint, and directly referenced the handle tag from the convenience variable **`$thr`**, which contains all tags.

- Resume the process in NightView, wait a few seconds, and then stop it.

Item	Value
⊟ ⚙ Threads	local:11571 a.out
⊞ ⚙ 11566	C thread 0x401df6c0
⊞ ⚙ 11570	C thread 0x403e0b90 (director)
⊞ ⚙ 11571	C thread 0x405e1b90 (worker_bee), job= Cosine (2) handle="cosine"
⊞ ⚙ 11572	C thread 0x407e2b90 (worker_bee), job=_Pi (3) urgent handle="apple"
⊞ ⚙ 11573	C thread 0x409e3b90 (worker_bee), job=_Pi (3) handle="apple"

Figure 4-16. Thread Tags Tutorial - String Tag Shown

The list of non-zero thread tags now includes the `handle` for the `worker_bee` threads.

Thread tags are not only useful for providing feedback to you on thread activities and state, you can use thread tags in eventpoint conditions.

- Issue the following command in NightView:

  `break do_math if $thr.job==_Cosine`

Here we've referenced a thread tag in a breakpoint, which is a convenient way to set thread-specific conditions.

- Resume the process and let it hit the breakpoint we just defined.

Within a few seconds, the process should stop and NightView will show that we've reached the breakpoint on line 115 and that the current thread is one executing the `_Cosine` job, as indicated in the following screen shot.


```
   113 static double do_math (double goal)
 ◆ 114 {
 ● 115     double angle = 0;
 ◆ 116     double delta = 2.0E-05;
```

Type	ID	Enabled	Ignore	Hits	Crossings	Cmds	Condition
● Break	5	Yes	0	1	4	No	$thr.job==_Cosine

Item	Value
🌼 Threads	local:11572 a.out
⚙ 11566	C thread 0x401df6c0
⚙ 11570	C thread 0x403e0b90 (director)
⚙ 11571	C thread 0x405e1b90 (worker_bee), job=_Sine (1) handle="sine"
⚙ 11572	C thread 0x407e2b90 (worker_bee), job= Cosine (2) handle="cosine"
⚙ 11573	C thread 0x409e3b90 (worker_bee), job=_Sine (1) urgent handle="sine"

Figure 4-17. Thread Tags Tutorial - Breakpoint Reached

In the figure above, we hovered the mouse cursor over the breakpoint decoration on line 115; this caused NightView to pop-up a tool-tip which summarized the breakpoint, its condition, and the hit and crossing counts. We can see that the breakpoint was crossed 4 times; 3 times the condition was false, so processing for a message other than `_Cosine` must have occurred during that time.

- Terminate the NightView session.

This concludes the Thread Tags tutorial.

Tracing Tutorial

This tutorial describes how to use NightTrace tracepoints while debugging an application with NightView.

NightTrace is part of the NightStar tool suite and allows you to instrument your application with function calls which log integer trace identifier values, along with any type of argument. Each logged trace event is timestamped. You can analyze the data graphically using the **ntrace(1)** command, or obtain ASCII listings of the data in chronological order, or write programs to consume the logged data using the NightTrace Analysis API.

This tutorial does not teach you how to use NightTrace, but will show you the minimal commands you need to use to collect and report on NightTrace data. The NightStar Tutorial is highly recommended as an introduction to using NightTrace, but you don't need to stop and read it to follow along in this tutorial.

In our example, we will insert trace events into an application that does not already use the NightTrace API.

- Compile the example program using commands similar to the following:

```
cat /usr/lib/NightView/Tutorial/tracing.c > tracing.c
cc -g tracing.c -lpthread -lrt -lm
```

NOTE

On some RedHawk Linux systems you may need to replace `-lrt` with `-lccur_rt` if the **cc** command above does not successfully link the program.

The example program is written in C and creates a `director` thread which doles out work to a set of three `worker_bee` threads.

- Start the process under NightView with the following command:

```
nview ./a.out &
```

NOTE

Be sure to background the NightView invocation, as we will be using this terminal session while NightView is still executing.

- Once NightView starts, add a `Data` panel with a `Threads` display, by using the `Data` menu and selecting the `Threads` option.

- Now let the process run for a bit by issuing the following two debugger commands, separated by a few seconds:

```
resume
stop
```

The program has executed for a few seconds and then stopped. The `Threads` display in the `Data` panel will look similar to the following figure:

Figure 4-18. Trace Tutorial - director and worker_bee threads

In our example program, there is a single `director` thread and three instances of a `worker_bee` thread. The `director` thread creates a workload message with a priority between 1 and 3, and queues the message waiting for a `worker_bee` thread to service the message.

Each `worker_bee` thread waits for a message, and then executes the work function specified in the message.

We might be interested in how long it takes to send a message, have it received, and then serviced.

We can do this by inserting NightTrace tracepoints into the application and use NightTrace to report on their instances and the times associated with them.

The `director` thread sends messages on line 48.

```
    48        if (mq_send(q,(const char*)&msg,sizeof(msg),priority)) {
 •  49            fprintf (stderr, "warning: mq_send() fails: %s\n", strerror(errno));
 •  50            sleep(10);
    51        }
```

Figure 4-19. Trace Tutorial - snippet of director source code

- Click on line 48 in the source panel and then right-click and select the Set eventpoint menu option and the Set Tracepoint option from that submenu.

- The Tracepoint dialog appears.
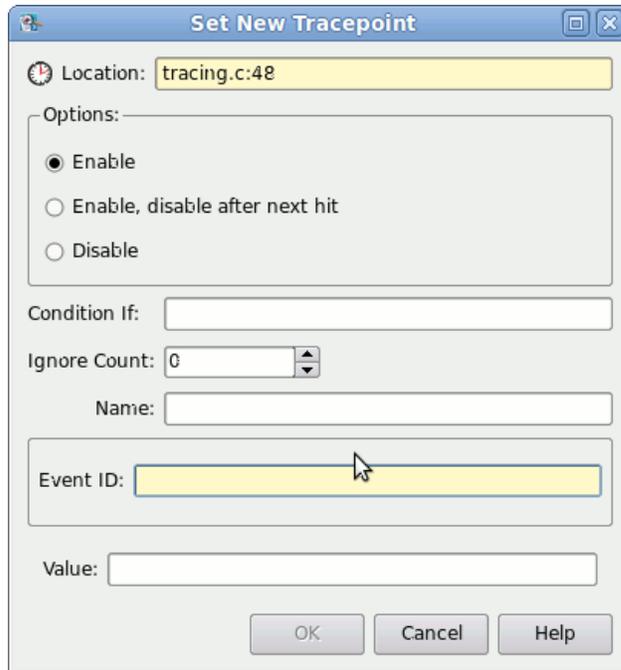
Figure 4-20. Thread Tags Tutorial - Tracepoint Dialog

- Ensure the Location field says `tracing.c:48`; if not, make it so.

- Enter the value `10` in the Event ID field.

- Click the OK button to close the dialog.

A tracepoint decoration now appears near the line number in the source display.

```
  48        if (mq_send(q,(const char*)&msg,sizeof(msg),priority)) {
  49            fprintf (stderr, "warning: mq_send() fails: %s\n", strerror(errno));
  50            sleep(10);
  51        }
```

Figure 4-21. Trace Tutorial - Source Panel w/ Tracepoint Decoration

Now we'll add another tracepoint at the point where messages are finished being serviced.

- Enter the following NightView command:

    trace 11 at tracing.c:78

Now we're almost ready to start collecting trace data. However, in order to use tracing, the application must make a call to the NightTrace Logging API to initialize tracing. Our application doesn't use the NightTrace Logging API, so we can have NightView take care of this for us.

- Enter the following NightView command:

    set-trace tracefile=tracing.data

The NightView message panel will respond with text similar to the following:

```
        set-trace tracefile=tracing.data
        tracing initialized, but no trace daemon running
```

NightView has linked in the required modules from `/usr/lib/libntrace_thr.so` and inserted a call to **trace_begin(3)** on the applications's behalf, passing `"tracing.data"` as the name of the trace file. NightView also detected that it could not locate a running NightTrace daemon associated with the `tracing.data` file.

NightTrace requires that a daemon is run to collect the trace data and either send it to a file, or send it to a live **ntrace(1)** analysis session. You can run your application without a daemon being active, but no trace records will be collected, until a daemon is subsequently launched.

- Resume process execution in NightView using the **resume** command.

The process is now executing and it is attempting to log tracepoints at lines 48 and 78, but since no daemon is active, they are harmlessly discarded.

- In the terminal session used to launch NightView, issue the following three commands:

    ntraceud --join tracing.data
    sleep 7
    ntraceud --quit-now tracing.data

These commands initiated a NightTrace daemon to collect the trace events, waited for seven seconds, and then terminated the daemon.

We can now use NightTrace to summarize the collected events.

IMPORTANT

Remember that this tutorial does not teach general use of Night-Trace and as such will be only using the most primitive of Night-Trace features so as not to confuse new users.

- Invoke the following command in the terminal session used in the previous step:

```
ntrace --summary=st:10-11 tracing.data
```

We have told NightTrace to summarize the data in file `tracing.data` and to report on a state which represents sending a message and subsequently servicing it.

The start of the state is defined by event 10, which we set at line 48 in the `director` thread at the call to `mq_send`. The end of the state is defined by event 11, which we set at line 78 in the `worker_bee` thread after processing the workload defined by the message.

NightTrace will generate output similar to the following:

```
==========================================================
Summary: States starting with event 10, ending with event
11.

State Summary Results
=====================

Number of states found:        3028

Maximum state duration:        0.003980141 at offset:
10201
Minimum state duration:        0.000006065 at offset: 933
Average state duration:        0.001202657
Total of state durations:      3.641646082

Number of state gaps found:    3028

Maximum state gap:             0.006318099 at offset:
10222
Minimum state gap:             0.000004960 at offset: 5043
Average state gap:             0.000668079
Total of state gaps:           2.022944709
```

Of most significance to us is the average state duration. We can see that on average, it took 1.2 milliseconds to send and subsequently receive and process a message.

We will now refine our tracepoints to get some more detailed data.

- In NightTrace, double-click on the tracepoint event in the Eventpoint panel to launch the Edit Tracepoint dialog.

- Set the Condition text field to the following:

  priority==1

- Press the OK button to close the dialog.

- Do the same for the second tracepoint shown in the Eventpoint panel.

- Now issue the following NightView commands; you can copy and paste these commands from this text into the command field of NightView if you wish:

  ```
  trace 20 at tracing.c:48 if priority==2
  trace 21 at tracing.c:78 if priority==2
  trace 30 at tracing.c:48 if priority==3
  trace 31 at tracing.c:78 if priority==3
  ```

The Eventpoint panel should now look like the following figure:

Type	ID ▲	Location	PID	Enabled	Ignore	Hits	Crossings	Cmds	Condition
🕐 Trace	1	tracing.c line 48 in director()	27761	✖	0	0	0	No	priority==1
🕐 Trace	2	tracing.c line 78 in worker_...	27761	✖	0	0	0	No	priority==1
🕐 Trace	3	tracing.c line 48 in director()	27761	✖	0	0	0	No	priority==2
🕐 Trace	4	tracing.c line 78 in worker_...	27761	✖	0	0	0	No	priority==2
🕐 Trace	5	tracing.c line 48 in director()	27761	✖	0	0	0	No	priority==3
🕐 Trace	6	tracing.c line 78 in worker_...	27761	✖	0	0	0	No	priority==3

Figure 4-22. Tracing Tutorial - 6 Tracepoints

Additionally, the source decorations for lines 48 and 78 should look like the following figure:

```
🕐  48        if (mq_send(q,(const char*)&msg,sizeof(msg),priority)) {
 ✦  49            fprintf (stderr, "warning: mq_send() fails: %s\n", strerror(errno));
 ✦  50            sleep(10);
    51        }
 ✦  77            calc = (*msg.work)(msg.arg);
🕐  78            if (min == 0.0 || calc < min) min = calc;
 ✦  79            if (max == 0.0 || calc > max) max = calc;
```

Figure 4-23. Tracing Tutorial - Conditional Tracepoint Decorations

The salmon-colored background shown on the tracing decoration indicates that a condition has been applied to the tracepoint at that line. The background color is obscured on line 78 because of the blue background used to indicate line selection.

- Issue the following commands in the terminal session we've been using in this tutorial:

  ntraceud --join tracing.data

```
sleep 7
ntraceud --quit-now tracing.data
```

The commands above collected trace data for seven seconds.

- Issue the following command to summarize the trace event data and print the average state durations.

```
ntrace --summary=st:10-11,st:20-21,st:30-31 \
fgrep "Average state duration"
```

That instructed NightTrace to summarize the events collected, in terms of three states; each state corresponds to messages of priority 1, 2, and 3; since we applied those conditions on the tracepoints in NightView.

The output of the last command should be similar to the following:

```
Average state duration:        0.001176966
Average state duration:        0.000314065
Average state duration:        0.000309296
```

Here we see that the average time required to send and subsequently receive and service messages seems to be affected by the message's priority.

To give you a glimpse into additional NightTrace capabilities, execute the following command:

```
ntrace /usr/lib/NightView/Tutorial/tracing.session \
   tracing.data
```

Here we have passed a NightTrace session file to **ntrace(1)** which includes some customization of graphs and event and state names. A window containing figures similar to the following should appear:
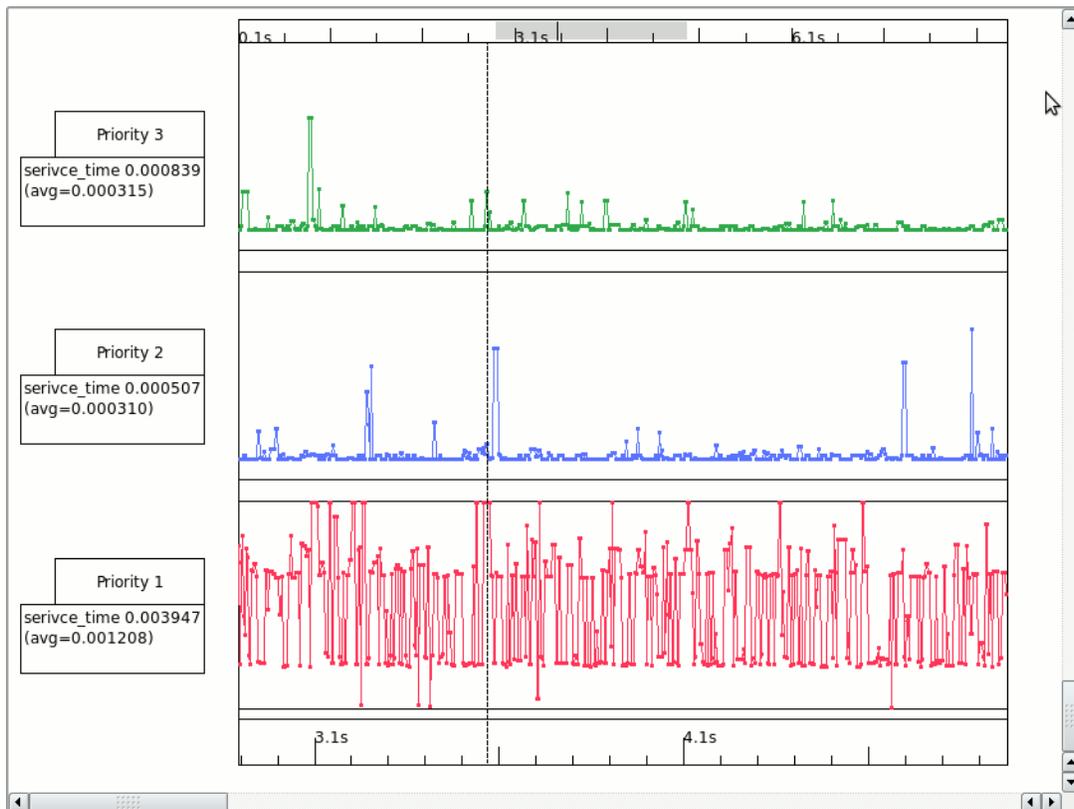
Figure 4-24. Tracing Tutorial - NightTrace Graph of Message Servicing

Figure 4-25. Tracing Tutorial - NightTrace Event List

The panel with the three graphs is showing the individual durations of each instance of a state; the red graph corresponds to priority 1 messages, blue to priority 2, and green to priority 3.

The data boxes to the left of the graphs show the individual service times of the message most recently serviced to the left of the "current timeline". The current timeline is defined

by wherever you click the mouse in the graph. The graphs can be zoomed in and out and panned left and right.

The textual events panel shows a simple text listing of every event, including the time at which it was logged. The current timeline in the data graphs is linked to the current row in the events panel; they always refer to the same event, regardless of which panel you interact with.

- Exit NightTrace.

- Exit NightView.

You can learn more about using NightTrace and its powerful features in the NightStar Tutorial. You can view the tutorial from the Help menu of any NightStar tool.

This concludes the Tracing tutorial.

5
Invoking NightView

This chapter describes how to start a NightView session.

nview

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 [-attach pid] [-config config-file] [-core core-file] [-help]
      [-nogui] [-nolocal] [-nx] [-prompt string]
      [-safety safe-mode] [-simplescreen] [-version]
      [-Xoption ...] [-x command-file]
      [program-name [program-argument ...]]
```

-attach *pid*

Attach to the process specified by *pid* in the local dialogue. This is similar to using the **attach** command. See "attach" on page 6-39. This option is not meaningful with **-nolocal**.

-config *config-file*

Load the configuration contained in config-file. This is similar to using the Load Config… item in the File menu. See "File Menu" on page 8-4. This option is valid only in the graphical user interface.

-core *corefile-name*

When you supply a **-core** option, NightView starts out by creating a pseudo-process for the given core file. See "Core Files" on page 3-4 and "core-file" on page 6-41.

-help

Causes NightView to print its command line syntax followed by a brief description of each option and then exit with code 0.

-nogui

Prevents NightView from automatically invoking the graphical user interface. See Chapter 8 [Graphical User Interface] on page 8-1.

-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 8-36).

-nx

> Prevents NightView from reading commands from the default initialization file. See "Initialization Files" on page 3-38.

-prompt *string*

> Sets NightView's initial prompt string to *string*. See "set-prompt" on page 6-63.

-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 6-65.

-simplescreen

> Directs NightView to use a simple full-screen interface. This option implies **-nogui**. See Chapter 7 [Simple Full-Screen Interface] on page 7-1.

-version

> Causes NightView to display its current version and then exit with code 0.

-*Xoption*

> NightView accepts a subset of the standard X Toolkit command line options (see **X(7x)**). These options are allowed only when using the graphical user interface. See below for a list of the options accepted.

-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.

program-name [*program-argument* **...**]

> If *program-name* is specified, NightView begins debugging that program.

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, a subset of other standard X Toolkit command line options are also allowed, e.g., **-geometry** *geometry-string*. See Chapter 8 [Graphical User Interface] on

page 8-1.

When using the graphical user interface, the X Toolkit options accepted include:

> **-display** *display*
>
> **-geometry** *geometry*
>
> **-fn** *font* or **-font** *font*
>
> **-bg** *color* or **-background** *color*
>
> **-fg** *color* or **-foreground** *color*
>
> **-btn** *color* or **-button** *color*
>
> **-name** *name*
>
> **-title** *title*

NightView uses the NightStar License Manager (NSLM) to control access to the NightStar tools. See "NightStar Licensing" on page A-1 for more information.

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 3-38), 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 6-23).

nview-save-core-file

When transporting a core file to another system for analysis, it is important to take all the shared libraries related to its execution. Locating them can be a cumbersome task that NightView can do for you more easily.

This command is provided as a convenience. It invokes NightView and saves all files associated with a core file in a compressed tar file for subsequent analysis and then exits.

The options are similar to the **save-core-file** command (see "save-core-file" on page 6-43).

nview-save-core-file [Options] *core-file* [*core-args*]

> **--nozip**
>
>> Do not compress the tar archive used to save the core file (the default is to generate a compressed file).

`--nodebuginfo`

Do not include the debuginfo files (if there are any). The default is to include them. Debuginfo files are special files that are optionally installed on systems which provide debug information for standard libraries. NightView knows how to locate these files, and by default, automatically consults them when they are present.

`--replace`

If the output file already exists, do not terminate the save operation, but instead to try to overwrite it.

`--keep`

If there are errors during the writing of the output file, keep the possibly broken file rather than removing it.

`--include`=*file*

Add *file* to the directory of information being packaged with the core file. The specified file can be of any type -- perhaps some data files used as input to your program or maybe just some notes you make so you can remember the circumstances relating to the core file. You can use multiple include options to include multiple files.

`--note`=*string*

Add the *string* as a note in the generated script that will be used to debug the core file subsequently. This note will be echoed at the end of the execution of the script that this script creates and places in the tar file.

`--output`=*file*

Defines the name of the output file; the final file will be named *file*.tar.gz or *file*.tar, depending on whether **`--nozip`** is specified. If this option is omitted, *file* defaults to *corefile*.

`--quiet`

Hide all output from the NightView session that is invoked as part of this script; that session is used to actually implement this script.

core-file [core-args]

Specifies the name of the core-file to be saved and passes *core-args* as arguments to the NightView **`core-file`** command which is invoked as part of the operation of this script.

This command will take a single core file and will generate the (optionally) compressed tar file. This can be useful if there is limited access time available on the current system or you wish to send the core file off to another person.

The tar file can then be transferred to another system for subsequent analysis. The system must be of the same architecture as the original system (i.e. 32-bit or 64-bit).

The tar file will contain all required files as well as a scripted called **debug-core-file**. Simply executing that script while positioned in the extracted directory will allow you to debug the core file.

6
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 8 [Graphical User Interface] on page 8-1), 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. An optional 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 6-16.

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 6-20.

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 6-170 and "source" on page 6-141).

NightView prompts you for input. The format of the prompt may be controlled by the **set-prompt** command (see "set-prompt" on page 6-63). 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 6-23).

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 3-24.

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 3-21. 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 6-1. Special '$' Constructs

$

A simple '$' by itself is a special convenience variable which always refers to the last value history entry (see "print" on page 6-85). See "Value History" on page 3-38.

$$

The name '$$' refers to the value history entry immediately prior to '$'. See "Value History" on page 3-38.

$*number*

A '$' followed by a number refers to that number entry in the value history. See "Value History" on page 3-38.

$\{-number\}$

> 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 3-38.

$identifier$

> 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 3-37, and "Predefined Convenience Variables" on page 6-6.

$\{file:line\ expression\}$

> 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 6-137) 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 6-85).

$\{*frame-addr\ expression\}$

> 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, regardless

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 6-92) 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 6-154) 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, monitorpoint, patchpoint, tracepoint, or heappoint), 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 6-4). The following special variables are all automatically defined on a per process basis.

Table 6-2. Predefined Convenience Variables

$_

This variable holds the address of the last item dumped with the **x** command (see "x" on page 6-88). 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 6-88, "info eventpoint" on page 6-146, "info breakpoint" on page 6-147, "info tracepoint" on page 6-148, "info patchpoint" on page 6-149, "info monitorpoint" on page 6-150, "info heappoint" on page 6-151, and "info line" on page 6-170.

$__

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 3-25.

`$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 6-167 and "info frame" on page 6-154).

`$cfa`

> `$cfa` is the *canonical frame address*. This is how the debug information describes the locations of the return address and the low-level registers saved in a frame. This may or may not be the same as `$fp`. See "info frame" on page 6-154.

`$is`

> `$is` is defined when a watchpoint is triggered. See "Watchpoints" on page 3-13. `$is` is the value of the variable being watched after the instruction that causes the trigger has completed.

IA-32 Registers

IA-32 machines are based on the Intel IA-32 architecture (see *IA-32 Intel Architecture Software Developer's Manual* for architectural details). See "info registers" on page 6-156.

In addition to the common register definitions for stack pointer (`$sp`), frame pointer (`$fp`), and program counter (`$pc`), the IA-32 machines support the registers shown in the following table.

Table 6-3. IA-32 Registers (iHawk Series 860)

$eax, $ebx, $ecx, $edx, $esi, $edi, $ebp, $esp

> These names map onto the 8 general purpose registers. Note that $sp is the same as $esp, and $fp may be the same as $ebp, depending on how the compiler generates code.

$ax, $bx, $cx, $dx, $si, $di, $bp, $sp16

> These names map onto the lower 16 bits of each of the 8 general purpose registers mentioned above, respectively. Note that the lower 16 bits of the ESP register is more commonly known as simply SP. But the name $sp is reserved for an architecture-independent stack pointer in NightView. So the name $sp16 is used for the lower 16 bits of the ESP register, instead.

$al, $bl, $cl, $dl

> These names map onto the low order 8 bits of each of the AX, BX, CX, and DX registers, respectively. In other words, they map onto bits 0-7 of each of the EAX, EBX, ECX, and EDX registers, respectively.

$ah, $bh, $ch, $dh

> These names map onto the high order 8 bits of each of the AX, BX, CX, and DX registers, respectively. In other words, they map onto bits 8-15 of each of the EAX, EBX, ECX, and EDX registers, respectively.

$eflags

> The program status and control register. NightView and the kernel use the TF flag of this register to implement stepping. See "step" on page 6-124, "stepi" on page 6-127, "next" on page 6-126, and "nexti" on page 6-128. Users should not modify the TF field of the $eflags register. Other flags in this register are used by the kernel. Care should be taken if modifying this register.

$eip

> The instruction pointer register. This is the same as the $pc register.

$cs

> The code segment register. The IA-32 architecture uses this register to determine the location of the executable code in memory. Users should not modify this register. Modification of this register in patchpoints and eventpoint conditions is prohibited.

$ss

> The stack segment register. The IA-32 architecture uses this register to determine the location of the process stack. Users should not modify this register.

$ds, $es, $fs, $gs

> The data segment registers. The IA-32 architecture uses the $ds register to determine the location of the process data. Users should not modify that register. Care should be taken if modifying $es, $fs, or $gs.

$csbase, $dsbase, $esbase, $fsbase, $gsbase, $ssbase

> These names map onto internal processor and kernel LDT data structures which hold the base addresses associated with the $cs, $ds, $es, $fs, $gs, and $ss registers, respectively. They are useful particularly for determining the location of thread-specific and task-specific data. For instance, if a disassembly address mode references memory with %fs:8 or %gs:(%eax), then the location can be determined in NightView with $fsbase+8 or $gsbase+$eax, respectively.

$st0 through $st7

> These names map onto the 8 floating point registers. The floating point registers on the IA-32 always hold 80 bit double extended precision (i.e. long double) values. Note that the architecture defines these registers as a stack. Also note that these registers are aliases of the registers $mm0 through $mm7.

$r0 through $r7

> These names map onto $st0 through $st7, but always referenced as though the floating point stack pointer were zero.

$cwd and $fctrl

> These names map onto the floating point control register. They are synonyms.

$swd or $fstat

> These names map onto the floating point status register. They are synonyms.

$twd or $ftag

> These names map onto the floating point tag word register. They are synonyms. These names may be used in the **info registers** command or in expressions in the **set** and **print** commands, but not in patchpoints or eventpoint conditions. This register may be read, but not modified. See $fxtag.

$fxtag

> This name maps to the floating point tag word register, but in a different form from $ftag. The form of this register is one byte with each bit corresponding to a floating point register. This name does not have the restrictions of $ftag.

$fip or $fioff

> These names map onto the lower 32 bits of the floating point last instruction pointer register. They are synonyms.

$fcs or $fiseg

> These names map onto the upper 16 bits of the floating point last instruction pointer register. They are synonyms.

$fop

> This name maps onto the floating point opcode register.

`$foo` or `$fooff`

> These names map onto the lower 32 bits of the floating point last data (operand) pointer register. They are synonyms.

`$fos` or `$foseg`

> These names map onto the upper 16 bits of the floating point last data (operand) pointer register. They are synonyms.

`$dr0` through `$dr3`

> These names map onto the debug address registers. NightView uses these registers to implement watchpoints. See "Watchpoints" on page 3-13. Users should not modify these registers.

`$dr6`

> This name maps onto the debug status register. NightView uses this register to implement watchpoints and single step. See "Watchpoints" on page 3-13, "step" on page 6-124, "stepi" on page 6-127, "next" on page 6-126, and "nexti" on page 6-128. Users should not modify this register.

`$dr7`

> This name maps onto the debug control register. NightView uses this register to implement watchpoints. See "Watchpoints" on page 3-13. Users should not modify this register.

`$mm0` through `$mm7`

> These names map onto the 64 bit vector registers available with Intel MMX Technology. Note that these registers are aliases of the registers `$st0` through `$st7`. However, the `$mm0` through `$mm7` registers are not defined as a stack.

`$xmm0` through `$xmm7`

> These names map onto the 128 bit vector registers available with the Streaming SIMD Extensions (SSE).

`$mxcsr`

> This name maps onto the SSE MXCSR control and status register.

Note that the floating point, debug, MMX, and SSE registers are not normally displayed by the **info registers** command (see "info registers" on page 6-156). If you want to display those registers, you can do so with the following commands:

info registers st.*	*displays floating point registers*
info registers dr.*	*displays debug registers*
info registers mm.*	*displays MMX registers*
info registers xmm.*	*displays SSE registers*
info registers .*	*displays all registers*

AMD64 Registers

AMD64 machines are based on the AMD AMD64 architecture (see *AMD64 Architecture Programmer's Manual* for architectural details). See "info registers" on page 6-156.

In addition to the common register definitions for stack pointer (`$sp`), frame pointer (`$fp`), and program counter (`$pc`), the AMD64 machines support the registers shown in the following table.

Table 6-4. AMD64 Registers (iHawk Series 870)

`$rax, $rbx, $rcx, $rdx, $rsi, $rdi, $rbp, $rsp, $r8` through `$r15`

> These names map onto the 16 general purpose registers. Note that `$sp` is the same as `$rsp`, and `$fp` may be the same as `$rbp`, depending on how the compiler generates code.

`$eax, $ebx, $ecx, $edx, $esi, $edi, $ebp, $esp, $r8d` through `$r15d`

> These names map onto the lower 32 bits of each of the 16 general purpose registers mentioned above, respectively.

`$ax, $bx, $cx, $dx, $si, $di, $bp, $sp16, $r8w` through `$r15w`

> These names map onto the lower 16 bits of each of the 16 general purpose registers mentioned above, respectively. Note that the lower 16 bits of the RSP register is more commonly known as simply SP. But the name `$sp` is reserved for an architecture-independent stack pointer in NightView. So the name `$sp16` is used for the lower 16 bits of the RSP register, instead.

`$al, $bl, $cl, $dl, $sil, $dil, $bpl, $spl, $r8b` through `$r15b`

> These names map onto the low order 8 bits of each of the AX, BX, CX, DX, SI, DI, R8W, R9W, R10W, R11W, R12W, R13W, R14W, and R15W registers, respectively. In other words, they map onto bits 0-7 of each of the RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14 and R15 registers, respectively.

`$ah, $bh, $ch, $dh`

> These names map onto the high order 8 bits of each of the AX, BX, CX and DX registers, respectively. In other words, they map onto bits 8-15 of each of the RAX, RBX, RCX, and RDX registers, respectively.

`$eflags`

> The program status and control register. NightView and the kernel use the TF flag of this register to implement stepping. See "step" on page 6-124, "stepi" on page 6-127, "next" on page 6-126, and "nexti" on page 6-128. Users should not modify the TF field of the `$eflags` register. Other flags in this register are used by the kernel. Care should be taken if modifying this register.

`$rip, $eip`

> `$rip` is the instruction pointer register. This is the same as the `$pc` register. `$eip` is the lower 32 bits of the `$rip`.

`$fs, $gs`

The data segment registers. Care should be taken if modifying `$fs` or `$gs`.

`$fsbase, $gsbase`

These names map onto the `FS.base` and `GS.base` model-specific registers or internal processor and kernel LDT data structures which hold the base addresses associated with the `$fs` and `$gs` registers, respectively. They are useful particularly for determining the location of thread-specific and task-specific data. For instance, if a disassembly address mode references memory with `%fs:8` or `%gs:(%eax)`, then the location can be determined in NightView with `$fsbase+8` or `$gsbase+$eax`, respectively.

`$st0` through `$st7`

These names map onto the 8 floating point registers. The floating point registers on the AMD64 always hold 80 bit double extended precision (i.e. long double) values. Note that the architecture defines these registers as a stack. Also note that these registers are aliases of the registers `$mm0` through `$mm7`.

`$fpr0` through `$fpr7`

These names map onto `$st0` through `$st7`, but always referenced as though the floating point stack pointer were zero.

`$cwd` and `$fctrl`

These names map onto the floating point control register. They are synonyms.

`$swd` or `$fstat`

These names map onto the floating point status register. They are synonyms.

`$twd` or `$ftag`

These names map onto the floating point tag word register. They are synonyms. These names may be used in the **info registers** command or in expressions in the **set** and **print** commands, but not in patchpoints or eventpoint conditions. This register may be read, but not modified. See `$fxtag`.

`$fxtag`

This name maps to the floating point tag word register, but in a different form from `$ftag`. The form of this register is one byte with each bit corresponding to a floating point register. This name does not have the restrictions of `$ftag`.

`$frip`

This name maps onto the 64-bit floating-point last instruction pointer register.

`$fioff`

This name maps onto the lower 32 bits of the floating-point last instruction pointer register.

`$frdp`

> This name maps onto the 64-bit floating-point last data (operand) pointer register.

`$fcs` or `$fiseg`

> These names map onto the upper 16 bits of the floating point last instruction pointer register. They are synonyms.

`$fop`

> This name maps onto the floating point opcode register.

`$foo` or `$fooff`

> These names map onto the lower 32 bits of the floating point last data (operand) pointer register. They are synonyms.

`$fos` or `$foseg`

> These names map onto the upper 16 bits of the floating point last data (operand) pointer register. They are synonyms.

`$dr0` through `$dr3`

> These names map onto the debug address registers. NightView uses these registers to implement watchpoints. See "Watchpoints" on page 3-13. Users should not modify these registers.

`$dr6`

> This name maps onto the debug status register. NightView uses this register to implement watchpoints and single step. See "Watchpoints" on page 3-13, "step" on page 6-124, "stepi" on page 6-127, "next" on page 6-126, and "nexti" on page 6-128. Users should not modify this register.

`$dr7`

> This name maps onto the debug control register. NightView uses this register to implement watchpoints. See "Watchpoints" on page 3-13. Users should not modify this register.

`$mm0` through `$mm7`

> These names map onto the 64 bit vector registers available with Intel MMX Technology. Note that these registers are aliases of the registers `$st0` through `$st7`. However, the `$mm0` through `$mm7` registers are not defined as a stack.

`$xmm0` through `$xmm15`

> These names map onto the 128 bit vector registers available with the Streaming SIMD Extensions (SSE).

`$xmm0gf` through `$xmm15gf`

> These names map onto the single low-order 8-byte floating-point value for each of the `$xmm0` through `$xmm15` registers, respectively.

`$xmm0wf` through `$xmm15wf`

> These names map onto the single low-order 4-byte floating-point value for each of the `$xmm0` through `$xmm15` registers, respectively.

`$mxcsr`

> This name maps onto the SSE MXCSR control and status register.

Note that the floating point, debug, MMX, and SSE registers are not normally displayed by the **info registers** command (see "info registers" on page 6-156). If you want to display those registers, you can do so with the following commands:

info registers st.*	*displays floating point registers*
info registers dr.*	*displays debug registers*
info registers mm.*	*displays MMX registers*
info registers xmm.*	*displays SSE registers*
info registers .*	*displays all registers*

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

line_number:*function_name* or *line_number*:*unit_name*

> specifies the first instruction for the given line in the file containing the given function or unit

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 3-26.

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 a 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 6-96), 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 (see "info line" on page 6-170) 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 6-2). 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 6-70).

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 6-62). 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 6-50.

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 6-24). 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.

> If your process is threaded, the threads are implemented as multiple processes sharing resources (thread processes). See "Multithreaded Programs" on page 3-40. You may use the PID of any of those thread processes in a qualifier. NightView considers them all to refer to the same process.

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 quali-

fier if you want to work on only one process at a time (see "set-qualifier" on page 6-62). 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 definition 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 6-98). In addition, the name '.' is a reserved name that always refers to the set of eventpoints most recently created by a single command. (If an eventpoint creation command fails, the definition of '.' is cleared.) Eventpoint numbers and eventpoint names are two types of *eventpoint specifiers*.

Another kind of eventpoint specifier is a location-spec. The location-spec must begin with the keyword **at**. See "Location Specifiers" on page 6-14. A location-spec eventpoint specifier with a line number refers only to eventpoints set at the beginning of that source line, not to any eventpoints that may be set on addresses within the line. Note also that a location-spec eventpoint specifier may refer to multiple locations, such as when a breakpoint is set in an inline function that is expanded multiple times.

Some commands expect more arguments after the eventpoint specifier. These commands do not accept a location-spec as an eventpoint specifier, because a location-spec eventpoint specifier must be the last argument.

Eventpoint specifiers that refer only to breakpoints may also be called *breakpoint specifiers* (*tracepoint specifiers*, *patchpoint specifiers*, *monitorpoint specifiers*, *heappoint specifiers*, *watchpoint specifiers* and *syscallpoint 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 6-71).

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 6-19). 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 6-5. 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 6-26, "nodebug" on page 6-27, and "on program" on page 6-46.

Wildcard patterns are similar to regular expressions, but are usually more convenient for

representing file names. See "Regular Expressions" on page 6-18.

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 6-6. 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  20:func
(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 @ln:func
>       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 6-65) 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.

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 6-65) 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 6-21).

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 6-27.

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 6-170) and for other programs that communicate with NightView.

/popup

Pop up the Remote Login Dialog Box (see "Remote Login Dialog Box" on page 8-36) 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 6-50). 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. *cpu list* is a comma-separated list of CPU IDs or CPU ID ranges. For example: "**0,2-4,6**". *cpu list* may also be **active** or **boot** to specify all active processors or the boot processor, respectively.

priority=*value*

> Specify the priority of the remote dialogue processes. The scheduling policy determines what values may be specified for the priority. *value* must be an integer value that is valid for the current scheduling policy. Higher numerical values represent more favorable scheduling priorities.

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. A quantum value is meaningful only under the **sched_rr** and **sched_other** scheduling policies. *time* is specified either as a nice value or a millisecond value corresponding to a nice value. Nice values must be between -20 and 19 inclusive. By default, a quantum value of -20 results in a ~300ms slice, and a quantum value of 19 results in a ~10ms slice. Millisecond values are in the form *number***ms** and must reflect the times defined in the system for the nice values. If a non-defined millisecond value is supplied, an "unsupported quantum" error message is returned.

The cpu, 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 6-19.

This command and its inverse (see "nodebug" on page 6-27) 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 /bin/* /sbin/* /usr/X11R6/bin/* /usr/ada/*/bin/*
        /usr/ada/bin/* /usr/bin/* /usr/bin/X11/* /usr/ccs/*/*
        /usr/ccs/*/*/*/* /usr/kerberos/bin/* /usr/kerberos/sbin/*
        /usr/lib/* /usr/lib/*/* /usr/lib/*/*/* /usr/lib/*/*/*/*
        /usr/lib/*/*/*/*/* /usr/lib/*/*/*/*/*/*
        /usr/lib/*/*/*/*/*/*/* /usr/local/bin/* /usr/local/sbin/*
        /usr/sbin/* /usr/ucb/* /usr/ucblib/*
debug *
```

To print the list of **debug** and **nodebug** patterns, see "info dialogue" on page 6-161.

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 6-26).

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 6-40).

To print the list of **debug** and **nodebug** patterns, see "info dialogue" on page 6-161.

set-debug-file-directory

Tell NightView where to look for .debug files.

set-debug-file-directory [*path*]

> *path*
>
> > The name of the directory in which to find .debug files.

The **set-debug-file-directory** sets the directory to use when searching for .debug files associated with shared libraries, for each dialogue in the qualifier. With no argument, NightView prints the current setting.

The default path is **/usr/lib/debug** which is where .debuginfo rpms usually install their files.

If you don't care about debugging libraries, you can improve performance by setting this to a directory that does not exist.

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.
>>
>> A file residing on a system other than the target system can be specified using the form *user*@*host***:/***path*. NightView will download this file from the specified system to the host. See "Remote File Access" on page 3-7.

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 6-19. See "Regular Expressions" on page 6-18.

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 6-44. See "load" on page 6-95. Dynamic library

names are also subject to object filename translation. See "Debugging with Shared Libraries" on page 3-43. Object filenames from **symbol-file** commands are *not* subject to object filename translation. See "symbol-file" on page 6-41.

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.

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.

Translate-object-file commands take effect in existing processes as well as future ones.

Examples:

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).

        (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 6-75.

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 6-30. Also, see "Initialization Files" on page 3-38.

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 6-65). 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 6-21). 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 6-40). 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 6-27.

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 6-18.

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 6-65.

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 6-32. See "Initialization Files" on page 3-38.

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 6-30. 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 3-38. 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 6-64, 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 6-170), 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 6-20.

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 6-37.

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 6-35).
>
> 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.

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.

> 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 6-34).

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 6-33): 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 or use the **rerun** command. See "Restarting a Program" on page 3-16. 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.)

rerun

Run a program again.

rerun

The qualifier must evaluate to exactly one process or exactly one dialogue and no process. This command takes no arguments.

Whenever a process starts up, NightView remembers the most recent dialogue input line and associates it with the new process.

If the qualifier contains a process, NightView kills the process and sends the associated dialogue input line again.

If there is no process, NightView sends the dialogue input line associated with the process that terminated most recently in the specified dialogue.

The method of remembering recent dialogue input lines works for nearly all situations, but there may be situations of complex process start-up where NightView cannot send an appropriate dialogue input line and this command should not be used.

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 6-133), 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 6-35 and "set-show" on page 6-34).

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 6-39) 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 6-38), 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 [{/resume | /stop}] PID

> /resume
>
> > Resume the process when the attach is complete.
>
> /stop
>
> > Keep the process stopped when the attach is complete.
>
> 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 6-33) 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 6-51. Children created prior to the attach must be explicitly attached if you want to debug them.

See "Attach Permissions" on page 3-42 for a description of what processes you are allowed to attach.

Once the **attach** is complete, the process will stay stopped or will be resumed depending on the setting from the **set-resume** command (see "set-resume" on page 6-72). You can override that setting by explicitly giving a /resume or /stop option.

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, heappoints, watchpoints, or syscallpoints set in those processes are removed, but patchpoints and tracepoints remain if they are enabled when you execute the **detach** command. See "breakpoint" on page 6-99, "patchpoint" on page 6-100, "monitorpoint" on page 6-106, "heappoint" on page 6-107, "tracepoint" on page 6-104, "watchpoint" on page 6-117, and "syscallpoint" on page 6-119.

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 6-65), 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 6-27. Also, see "set-children" on page 6-51.

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 6-65) 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. 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 informa-
>>   tion for the program.
>>
>>   If *program-name* is a relative pathname, it is interpreted relative to Night-
>>   View's current working directory.
>>
>>   If the program is on a remote system other than the specified target system,
>>   use the form *user*@*host*:**/**path*. NightView will download this file from the
>>   remote system to read the debug information. See "Remote File Access" on
>>   page 3-7.

program-name is *not* subject to object filename translations. See "translate-object-file" on
page 6-28.

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 6-44).

In some situations, such as when debug information is needed from shared libraries, an
object filename translation is more appropriate than a **symbol-file** command. See
"translate-object-file" on page 6-28.

core-file

Create a pseudo-process for debugging an aborted program's core image file.

core-file *corefile-name* [exec-file=*program-name*][with-translations]
                        [interpreter-base=*address*]

corefile-name

> The name of a core file. When used in a remote dialogue, this file must reside on the target system.
>
> 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. When used in a remote dialogue, this file must reside on the target system.
>
> If *program-name* is a relative pathname, it is interpreted relative to Night-View's current working directory.

with-translations

> Indicates that the lines following the **core-file** command are library translations.

interpreter-base=*address*

> Specify the address of the dynamic loader. This may be useful for a core file generated on a system different from the target system.

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 6-50.

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. The core file must have been created on the system the dialogue is running on.) 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 6-44) 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 3-9). 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 6-44 and "symbol-file" on page 6-41).

If you specify `with-translations`, then the lines following the **core-file** command are library translations of the form:

> *from-string   to-string*

End the translations with a line that contains only:

> `end translations`

This allows debugging core files from dynamically-linked programs on systems where the installed libraries do not match the libraries that were being used when the core file was generated. This is not necessary for most users. The translations are similar to the object file translations in the **translate-object-file** command, but they refer to dynamic libraries and are applied only to this process (see "translate-object-file" on page 6-28). For remote debugging, the translations are applied on the target system, not the host system.

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.

save-core-file

Packages up all files required for subsequent core file analysis into a compressed file.

When transporting a core file to another system for analysis, it is important to take all the shared libraries related to its execution. Locating them can be a cumbersome task that NightView can do for you more easily.

save-core-file [/nozip] [/nodebuginfo] [/replace] [/keep] [include=*file*] [note=*string*] *savename*

> /nozip
>
>> Do not compress the tar archive used to save the core file (the default is to generate a compressed file).
>
> /nodebuginfo
>
>> Do not include the debuginfo files (if there are any). The default is to include them. Debuginfo files are special files that are optionally installed on systems which provide debug information for standard libraries. NightView knows how to locate these files, and by default, automatically consults them when they are present.

`/replace`

> If the output file already exists, do not terminate the save operation, but instead to try to overwrite it.

`/keep`

> If there are errors during the writing of the output file, keep the possibly broken file rather than removing it.

`include=`*file*

> Add *file* to the directory of information being packaged with the core file. The specified file can be of any type -- perhaps some data files used as input to your program or maybe just some notes you make so you can remember the circumstances relating to the core file. You can use multiple `include` options to include multiple files.

`note=`*string*

> Add the *string* as a note in the generated script that will be used to debug the core file subsequently. This note will be echoed at the end of the script.

savename

> The name of the file where the (optionally) compressed tar file will be written. The actual file name will be *savename*.tar.gz or s*avename*.tar.

This command will take a single core file process and will generate the *savename* tar file. This can be useful if there is limited access time available on the current system or if you wish to send the core file off to another person.

The tar file can then be transferred to another system for subsequent analysis. The system must be of the same architecture as the original system (i.e. 32-bit or 64-bit).

If you are not already debugging the core file of interest, you can use the command **nview-save-core-file** to save the core file. See "nview-save-core-file" on page 5-3 for information on that script.

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.

> If *program-name* is on a remote system other than the specified target system, use the form *user*@*host*:*/path*. NightView will download this file from the

specified system to read the debug information. In this case, **exec-file** is treated as though you had used **symbol-file**.

program-name is subject to object filename translations. See "translate-object-file" on page 6-28.

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 6-41). 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 3-9.

If you do not explicitly specify a symbol file for a process (see "symbol-file" on page 6-41), 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 6-46.

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.

When dealing with shared libraries, an object filename translation is more appropriate than a **exec-file** command. See "translate-object-file" on page 6-28.

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 6-19.
>
> *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 6-48. 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 3-16.

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 6-65.

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 3-36.

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 6-46). 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 execed programs. See "Wildcard Patterns" on page 6-19.
>
> *command*
>
>> A debugger command to be executed when a new program whose executable file name matches *pattern* is execed.

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 3-16. 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 6-46 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 6-163.

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 3-16. **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 6-16.

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 6-65.

If one or more *qualifier-spec* arguments are supplied, they are immediately evaluated (see "Qualifier Specifiers" on page 6-16) 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 resumed 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(3)** (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 6-26. This is the default setting for direct children of the dialogue shell and processes debugged with the **attach** command. See "attach" on page 6-39.

    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 6-40.

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 3-18.

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 initial **set-exit** mode for each process comes from the global **set-resume** mode. See "set-resume" on page 6-72. 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 6-46 and "on restart" on page 6-48. See also "Restarting a Program" on page 3-16.

If you also want a process to automatically resume execution after an exec, use the **set-resume** command, or put a **resume** command in an **on program** specification. See "set-resume" on page 6-72, "resume" on page 6-123 and "on program" on page 6-46.

set-shared-lib-update

Control whether a process stops before exiting.

set-shared-lib-update [on | off]

The default is off, but if you turn it on, then a hidden breakpoint in the dynamic linker will have its crossing count checked each time the process stops, and if it changes, it will do the equivalent of an **exec-file** command to re-read the object including the new shared libraries.

If this overhead is worrisome, set it to off and use the Refresh Shared Libs option from the Process menu manually when you want NightView to re-load shared library

information.

wait

Wait for processes to stop.

wait [{all | any} [new]]

> all
>
>> Wait for all processes in the qualifier to stop.
>
> any
>
>> Wait for any process in the qualifier to stop.
>
> new
>
>> Implicitly add any new processes that show up to the qualifier.

The **wait** command waits for processes in the qualifier to stop. See "Process States" on page 19. That is, no more commands are read from this command stream until the specified processes stop. See "Command Streams" on page 36. See "Interrupting the Debugger" on page 36.

If no arguments are specified, the default behavior is **wait** any new.

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 3-9. See "load" on page 6-95. See "Expression Evaluation" on page 3-21.

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 6-158.

Memory reservations are remembered as part of the restart information. See "Restart Information" on page 3-17. During restart, memory reservations are applied before any commands that would allocate space in the process.

Heap Debugging

heapdebug

Specify parameters for heap debugging.

heapdebug [check_free_fill={0|1}]
        [common_errors={block_overrun |
                        dangling_pointer |
                        uninitialized_field}]
        [do_free_fill={0|1}]
        [do_malloc_fill={0|1}]
        [*error-name* [{noprint | nostop | print | stop} ...]]
        [free_fill_byte={*n* | trash}]
        [frequency=*n*[{k|m}]]
        [heap_size={*n*[{k|m}] | unlimited}]
        [internal_checks={0|1}]
        [level={0|1|2|3}]
        [malloc_fill_byte={*n* | trash}]
        [off]
        [on]
        [post_fence_size=*n*]
        [post_fill_byte={*n* | trash}]
        [pre_fence_size=*n*]
        [pre_fill_byte={*n* | trash}]
        [protected={0|1}]
        [retain_free_blocks={*n*[{k|m}] | unlimited}]
        [slop=*n*]
        [walkback=*n*]

Abbreviation: **hd**

`check_free_fill={0|1}`

> During heap checks, check that the free fill has not been disturbed in retained free blocks. Setting this to 0 (turning it off) improves performance, but does not detect as many errors. The default value is 1 (check free fill).

`common_errors={block_overrun | dangling_pointer |`
`                uninitialized_field}`

> This is a convenient way to set parameters to detect common program errors.

> `block_overrun`

>> detect program writing past the end of a block

> `dangling_pointer`

>> detect program referencing a freed block

> `uninitialized_field`

>> detect program failing to initialize fields in a block

`do_free_fill={0|1}`

> When a block is freed, fill it with the `free_fill_byte`. Free fill applies to free blocks that are retained and also to free blocks that are immediately available for reuse. Setting this to 0 (turning it off) will disable free filling. The default value is 1 (fill free blocks).

`do_malloc_fill={0|1}`

> When a block is allocated, fill it with `malloc_fill_byte`. Setting this to 0 (turning it off) disables malloc filling. The default value is 1 (fill allocated blocks).

error-name `[{noprint | nostop | print | stop} ...]`

> Specify how the debugger responds when an error condition is detected.

> `stop`

>> stop the process when the error occurs; implies `print`

> `nostop`

>> let the process continue when the error occurs

> `print`

>> print a message when the error occurs

> `noprint`

>> do not print a message when the error occurs; implies `nostop`

error-name can be any of the following:

```
free_fill_modified
free_not_at_beginning
free_unallocated
internal_error
malloc_zero
memalign_not_power_2
out_of_memory
post_fence_modified
pre_fence_modified
realloc_not_at_beginning
realloc_unallocated
```

The default for all the errors is `stop print`, except for `malloc_zero`, `memalign_not_power_2`, and `out_of_memory`, which are not normally considered to be heap errors; the defaults for those errors is `nostop noprint`.

Blocks are checked for errors during a heap check (see "Heap Check" on page 3-34) and when they are freed or `realloc`'ed. Other errors are detected during heap operations.

`free_fill_modified`

> The free fill pattern in a retained free block has been modified.

`post_fence_modified`

> The post-fence fill pattern in a block has been modified.

`pre_fence_modified`

> The pre-fence fill pattern in a block has been modified.

`free_not_at_beginning`

> `free` was called with an address that is within an allocated block, not at the beginning of a block.

`free_unallocated`

> `free` was called with an address that does not correspond to any currently allocated block.

`internal_error`

> An inconsistency was found in the internal data structures. There is a bug in the heap debugger, or the process has modified the heap debugger's internal data structures.

`malloc_zero`

> The program asked for a block of size 0 bytes. This is not normally considered to be an error. The default disposition for `malloc_zero` is `nostop, noprint`.

`memalign_not_power_2`

> The program called `memalign` with an alignment that is not a power of 2. This is not normally considered to be an error. The default disposition for `memalign_not_power_2` is `nostop, noprint`.

`out_of_memory`

> The process ran out of memory, either because the system could not satisfy the request or because of the setting of the heapsize parameter. This is not normally considered to be an error. The default disposition for `out_of_memory` is `nostop, noprint`.

`realloc_not_at_beginning`

> `realloc` was called with an address that is within an allocated block, not at the beginning of a block.

`realloc_unallocated`

> `realloc` was called with an address that does not correspond to any currently allocated block.

`free_fill_byte={`*n*` | trash}`

> The value to put in each byte of each block when it is freed if `do_free_fill` is 1. The default is `trash`, which, for `free_fill_byte`, is 0xc3.

`frequency=`*n*`[{k|m}]`

> The heap is checked every *n* heap operations (`mallocs`, `frees`, etc.). You may append `k` to multiply *n* by 1024 or `m` to multiply by 1048576. If *n* is zero, the heap is checked only by a **heappoint** (see "heappoint" on page 6-107) or a **heapcheck** command (see "heapcheck" on page 6-165). The default value is 10000.

`heap_size={`*n*`[{k|m}] | unlimited}`

> The program is not allowed to allocate more than *n* total bytes. The default value is `unlimited`. You may append `k` to multiply *n* by 1024 or `m` to multiply by 1048576.

`internal_checks={0|1}`

> If set to 1 (turned on), then during a heap check, check internal data structures for integrity. This adds a large overhead to each heap check. The default value is 0 (do not check internal data structures).

`level={0|1|2|3}`

> This is a convenient way to set many of the other parameters.

> 0

>> disable checking

1

> minimal checking

2

> a medium level of checking

3

> extreme checking

See "Levels and Common Errors" on page 3-31 for a discussion of heap debugging levels.

`malloc_fill_byte={`*n*` | trash}`

> The value to put in each byte of a block when it is allocated, if `do_malloc_fill` is 1. The default is `trash`, which, for `malloc_fill_byte`, is 0xc5.

`off`

> Turn heap debugging off.

> If heap debugging is off when the process makes its first allocation, the heap debugger adds little or no overhead. If heap debugging is turned off after the first allocation, the heap debugger still adds overhead, but it no longer checks for errors.

`on`

> Turn heap debugging on.

> Heap debugging may be turned on before the program makes its first allocation. After the program makes its first allocation, heap debugging may be turned on only if it was on when the program made its first allocation.

`post_fence_size=`*n*

> Add *n* bytes after the end of a block when it is allocated, fill them with `post_fill_byte`, and check them during a heap check. The default is zero (no fence).

> In hardware overrun protection mode, there may be a gap between the end of the block and the protected page, due to aligment requirements and the size of the block. At most *n* bytes of the gap are filled and checked. See "Hardware Overrun Protection" on page 3-32.

`post_fill_byte={`*n*` | trash}`

> The value to put in each post-fence byte of a block when it is allocated. The default is `trash`, which, for `post_fill_byte`, is 0xaf.

`pre_fence_size=`*n*

> Add *n* bytes before the beginning of a block when it is allocated, fill them with `pre_fill_byte`, and check them during a heap check. The default is zero (no fence).

`pre_fill_byte={`*n* `| trash}`

> The value to put in each pre-fence byte of a block when it is allocated. The default is `trash`, which, for `pre_fill_byte`, is 0xbf.

`protected={0|1}`

> If set to 1, turn on hardware overrun protection. Each block is allocated such that the end of the block is as near as possible to the end of a page. The following page is protected from reads and writes. See "Hardware Overrun Protection" on page 3-32.

> The default value is 0 (no hardware overrun protection).

`retain_free_blocks={`*n*`[{k|m}] | unlimited}`

> The number of recently-freed blocks to retain. You may append `k` to multiply *n* by 1024 or `m` to multiply by 1048576. Retained free blocks are not immediately available for reuse. See "Retained Free Blocks" on page 3-33.

> The default value is 0 (no retained blocks).

`slop=`*n*

> Add *n* bytes to the size of each allocation. For example, if *n* is 4 and the program calls `malloc(8)`, the allocation proceeds as though the program had called `malloc(12)`. The default value is 0 (no slop).

`walkback=`*n*

> The maximum number of walkback entries to keep for each heap operation (`malloc`, `free`, etc.). More walkback entries may help you identify which routines are causing heap problems. The default value is 8 entries. This count refers to physical walkback entries. The number of walkback frames may differ from this number when displayed in NightView. The number of frames displayed may include extra inline frames, as they are not physical frames. The number of frames displayed may be fewer if certain frames are deemed uninteresting (see "interest" on page 6-67). See "Debugging the Heap" on page 3-30.

The **heapdebug** command configures the heap debugger in each of the processes in the qualifier. See "Debugging the Heap" on page 3-30. Another way to configure the heap in the graphical user interface is with the Debug Heap… item in the Process menu (see "Process Menu" on page 8-8).

All arguments may be abbreviated to the shortest unambiguous prefix.

The heap debugger remembers its settings when turned off. This way, it can be turned back on at a later time and will retain all of its former settings.

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 6-146).

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 6-34).

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 6-85), 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 6-16). 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*| source=*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 6-14). If you want unlimited output, specify zero as the limit.

source=*number*

> The `source` keyword parameter specifies the maximum number of bytes a source file can have to be displayed in a source panel. This is useful for extremely large source files which overwhelm NightView due to the overhead involved in building individual widgets associated with each line. The number is in units of 1000 bytes. The default value is 4000, which indicates ~4MB. The source for files that exceed the limit are not displayed, but the assembly associated with the function associated with the current stack frame within the file is displayed.

The `array`, `string`, `addresses`, and `source` 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 6-156).

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 substrings 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 6-33).

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 6-21). 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 3-16.

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 3-11.

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 6-161).

You can see information about the patch areas in an existing process with the **info memory** command (see "info memory" on page 6-158).

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 6-14. 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 3-17. For a way to see all the interest levels that have been explicitly set, see "info on restart" on page 6-163.

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 3-26. 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 6-138. 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 6-82.

```
(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 6-67. 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 3-21. 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 3-24. For the details of the syntax used to specify overloading in expressions and location specifiers see "Selecting Overloaded Entities" on page 6-2.

When NightView starts, the overloading modes are initially:

>   **set-overload operator=off routine=on**

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 6-18) 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 7-2.

set-preallocate

Control how NightView preallocates memory for eventpoints and monitorpoint buffers.

set-preallocate [/eventpoint] [/monitorpoint] [{off | on}]

> /eventpoint
>
>> Indicates the eventpoint mode should be set or queried.

```
/monitorpoint
```

Indicates the monitorpoint buffer mode should be set or queried.

```
{off | on}
```

Turn preallocation off or on for eventpoints or monitorpoint buffers as specified.

The default is for NightView to preallocate space in the user process for eventpoints and monitorpoint buffers. If these modes are off, then NightView allocates space only when needed. However, space can be allocated only when the process is stopped. With preallocation, you do not need to worry about whether the process is stopped when you set an eventpoint. See "Process States" on page 3-19. See "Operations While the Process Is Executing" on page 3-19.

Monitorpoints cannot be used in the command-line interface, so in that interface monitorpoint buffers are never preallocated.

This is a global mode. The qualifier is ignored.

With no arguments, **set-preallocate** prints the current settings.

set-resume

Control NightView's behavior on events that normally stop a process.

set-resume [/attach] [/exec] [/exit] [/fork] [{off | on}]

```
/attach
```

Indicates the attach mode should be set or queried.

```
/exec
```

Indicates the exec mode should be set or queried.

```
/exit
```

Indicates the exit mode should be set or queried.

```
/fork
```

Indicates the fork mode should be set or queried. NightView pays attention to this for a child only if the parent process has **set-children all**. See "set-children" on page 6-51.

```
{off | on}
```

Turn automatic resume off or on for the modes specified.

A process is normally stopped when NightView attaches to it, when it execs, when it is about to exit, and when it is created (i.e., its parent forks). This command allows you to control that for each case.

This is a global mode. The qualifier is ignored.

If off or on is not specified, **set-resume** prints the current settings.

set-download

Control how NightView downloads files from remote targets.

set-download [{off | permanent | temporary}]
                [directory=*path-to-cache*]

> off
>
>> This disables the download feature.
>
> permanent
>
>> This enables downloading files to the cache. The files are left in the cache for future use.
>
> temporary
>
>> This enables downloading files to the cache. When NightView exits, it removes all the files in the cache that were downloaded under temporary mode. This is the default setting.
>
> directory=*path-to-cache*
>
>> This specifies the directory to use to build the cache. The default directory name is **~/.NightViewCache**. The downloads will be faster if this directory is local to the system where you are running NightView.

If NightView cannot find an object file on the local system, and this process is in a remote dialogue, then NightView searches on the target system for the file and copies it to the cache directory. See "Remote File Access" on page 3-7.

This is a global mode. Any qualifier is ignored.

To manipulate this mode in the graphical user interface, see "Preferences Advanced Page" on page 8-48.

With no arguments, **set-download** prints the current download information.

set-disassembly

Control how NightView displays disassembled instructions.

set-disassembly [flavor={att | intel}] [symbols={off | on}]
                [comment_level=*number*]

> flavor={att | intel}
>
>> Set the flavor of disassembly to the att style or the intel style.
>>
>> The default flavor is att. The att flavor is the one used by the standard compilation tools. The intel flavor is the one described in the *IA-32 Intel*

Architecture Software Developer's Manual or the *AMD64 Architecture Programmer's Manual*. For a discussion of the differences in the two flavors, enter this command in a shell outside of NightView:

info 'gas' 'Machine Dependencies' i386-Dependent

symbols={off | on}

Indicate whether each line of the disassembly should include the name of the routine being disassembled. If this is set to on, <*routine+offset*> is appended to each address, where *routine* is the name of the routine being disassembled and *offset* is the offset (in bytes) from the beginning of *routine*. If this is set to off, then the routine name appears once at the beginning of the disassembly, and <+*offset*> is appended to each address. References to addresses outside the routine being disassembled include the name in either mode. The default is off, which makes the disassembly listing more compact.

comment_level=*number*

Indicate the kind of comments the disassembler should provide. Comments provide more information about the instructions.

number is one of:

0

   never print comments

1

   print operation information for certain instructions

2

   print "aka" ("also known as") opcode aliases in addition to the level 1 comments

This command sets the disassembly modes for the debug session. Any qualifier is ignored. See "x" on page 6-88. See "Source Menu" on page 8-10.

If no arguments are specified, **set-disassembly** prints the current disassembly information.

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 6-141)

- **list** (see "list" on page 6-76)

- **directory** (see "directory" on page 6-78)

- **symbol-file** (see "symbol-file" on page 6-41)

- **core-file** (see "core-file" on page 6-41)

- **exec-file** (see "exec-file" on page 6-44)

The **cd** command does not affect commands executed in dialogue shells (see "login" on page 6-24). 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 6-75.

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 6-75).

The qualifier does not have any effect on this command.

Source Files

This section describes commands to view and edit source files and to search for text in source files.

Viewing and Editing 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 6-14). 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.

> A file name on a remote system can be specified using the form *user*@*host*:*/path*. See "Remote File Access" on page 3-7.

+*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 6-78). 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 6-136) 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 6-80 and "reverse-search" on page 6-80.

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 6-82).

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 6-170).

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 Night-View's working directory, the search path will not be affected. See "cd" on page 6-75 and "pwd" on page 6-75.
>>
>> The search can be performed on remote systems by specifying *dirname* in the form *user*@*host*:**/***dirname*. See "Remote File Access" on page 3-7.

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 6-76), 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(3)**, 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 6-155.

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.

Examples:

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**

Similarly, if the source files are now on system "oursys", use this command:

> (local) **directory oursys:/usr/joe**

edit

Edit the current source file.

edit

This command invokes a text editor on the source file currently displayed in the source panel.

This command can be used only from the graphical user interface. See "Source Panel" on page 8-53.

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 6-18). 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 panel. 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 6-76.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 6-71).

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 6-18). 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 panel. 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 6-76.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 6-71).

Source Line Decorations

When NightView lists source lines in your program or displays the disassembled 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 6-88).

In the serial interface, line numbers precede the decorations. If multiple decorations are needed, they are concatenated, such as BP=.

In the graphical user interface, line numbers follow the decorations. If multiple eventpoints are present, that is represented by a separate icon. If the program is stopped on the line, the program counter icon is overlaid on any other icon. See "Graphical User Interface" on page 8-1. If you hover the mouse pointer over the icon, a tooltip shows the eventpoint information for the line.

Also, in the graphical user interface, the icon may indicate additional information. Here are the rules NightView uses for choosing which icon is displayed:

- If there are different kinds of eventpoints on the line, a generic eventpoint icon is shown. No other information is represented with this icon.

- If all the eventpoints on the line are disabled, the icon has reduced color.

- If the eventpoint has a non-zero ignore count, or the eventpoint has a condition, or there are multiple eventpoints (of the same kind) on the line, the icon has a pink background. This indicates that more information is available.

- Otherwise, this is a single simple eventpoint and the regular icon is used.

The following table lists the source line decorations. The decoration for the serial interface is listed first, followed by the icon for the graphical user interface.

Table 6-7. Source Line Decorations

'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 6-99).

'H'

Indicates that one or more heappoints, possibly disabled, are set somewhere within this source line. When displaying instructions, this indicates that one or more heappoints are set on this instruction. (See "heappoint" on page 6-107).

'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 6-106).

'P'

Indicates that one or more patchpoints, possibly disabled, have been inserted somewhere within this source line. (See "patchpoint" on page 6-100). 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 6-104).

Indicates that multiple kinds of eventpoints, possibly disabled, are set within this source line. When displaying instructions, this indicates multiple kinds of eventpoints immediately preceding this instruction. In the serial interface, eventpoint characters are concatenated.

'='

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). In the graphical user interface, this icon is overlaid over any other icon.

'>'

Indicates the line (or instruction) in the current frame (see "frame" on page 6-136), where execution will resume when the called routine returns. This is not represented in the graphical user interface.

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 6-136), which was executing when the called frame was created, i.e., $cpc. See "Program Counter" on page 3-25.

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 decoration indicates that the associated instruction is the first for the corresponding source line.

In the serial interface, 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 graphical user interface, the line numbers are expanded as necessary and the source decorations always take the same amount of space.

In the serial interface source listing, the 2-character separator is a vertical bar followed by a space. This helps distinguish decorations from source characters. In the serial interface disassembly listing, the 2-character separator consists of 2 spaces.

> Example source listing, in the serial interface:

```
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 disassembly 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 6-136, "up" on page 6-137, or "down" on page 6-138.

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`

Print 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 3-21).

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 3-38), 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 6-62).

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 3-21).

This command is similar to the **print** command (see "print" on page 6-85), 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.
>>
>> p
>>
>>> Each memory unit is the size of a pointer on the target system. On an IA-32 system, this is 4 bytes. On an AMD64 system, this is 8 bytes.
>>
>> t
>>
>>> Each memory unit is the size of a C long double on the target system. On an IA-32 system, this is 12 bytes. On an AMD64 system, this is 16 bytes. A t memory unit cannot be printed as decimal d or u.
>>
>> 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 `p`.

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.

See "set-disassembly" on page 6-73 to control the form of the diassembly.

You can also view disassembly in a source panel (see "Source Menu" on page 8-10).

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 6-62).

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 3-21).

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 p 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 6-6). The convenience variable $_ is set to the 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 6-82 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 6-85).
>
> *expression*
>
>> A language expression (see "Expression Evaluation" on page 3-21).

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.

data-display

Control items in a data panel.

data-display [/window="*window name*"] {/kind=*value* | *expression*}

/window="*window name*"

Determines which data panel is affected by this command.

The default is Data.

/kind=*value*

value indicates which kind of item to placed in the data panel. *value* is one of locals, registers, callstack or threads.

expression

An expression to place in a data window. There should not be a /kind keyword in this form of the command.

The **data-display** command is not intended to be used directly by users. Its main use is in restart information. See "Restart Information" on page 3-17. A description of all the forms of this command is beyond the scope of this document. However, users may sometimes have a use for the simplest forms of the **data-display** command described here.

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 6-85).

expression

> A language expression (see "Expression Evaluation" on page 3-21).

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 6-88). 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 6-88).

addr-expression

> An expression yielding a memory address (see "Expression Evaluation" on page 3-21).

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 6-155).

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 6-92).

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 6-94). 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 6-155).

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 6-92).

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 6-155).

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 3-21).

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(3)**, 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 6-28).

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.

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, heappoints, watchpoints and syscallpoints as well. The following table indicates which types of eventpoints may be affected by which commands:

Table 6-8. Eventpoint Commands

Command Name	What the Command May Apply To						
	Breakpoints	**Patchpoints**	**Tracepoints**	**Monitorpoints**	**Watchpoints**	**Heappoints**	**Syscallpoints**
`name`	X	X	X	X	X	X	X
`clear`	X	X	X	X		X	
`commands`	X			X	X		X
`condition`	X	X	X	X	X	X	X
`delete`	X	X	X	X	X	X	X
`disable`	X	X	X	X	X	X	X
`enable`	X	X	X	X	X	X	X
`ignore`	X	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 **enable** command (see "enable" on page 6-113).

`/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 6-113).

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 6-17.
>>
>> 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 6-98.

name=*breakpoint-name*

> Gives a name to the breakpoint for later reference. (see "name" on page 6-98). 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 6-14).

> 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 3-26.

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 3-25 and "Context" on page 3-25). See also "Expression Evaluation" on page 3-21.

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 6-114).

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 6-110.

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

Patchpoints can be used to:

- Install a small patch to a routine.

- Insert an expression in the program which modifies the value of a variable or register or invokes a function.

- Insert a branch in the program.

- Set a thread-specific tag value

patchpoint [*eventpoint-modifier*] [name=*patchpoint-name*]
   [[at] *location-spec*] eval *expression*

patchpoint [*eventpoint-modifier*] [name=*patchpoint-name*]
   [[at] *location-spec*] goto *location-spec*

patchpoint [*eventpoint-modifier*] [name=*patchpoint-name*]
   [[at] *location-spec*] tag *tag-assignments*

eventpoint-modifier

> Specifies the patchpoint modifier. See "Eventpoint Modifiers" on page 6-98.

name**=***patchpoint-name*

> Patchpoints are assigned event numbers, and the name= syntax as well as the **name** command (see "name" on page 6-98) may be used to give them names. See "Manipulating Eventpoints" on page 6-96.

at *location-spec*

> Specify the exact point in the program to execute the patchpoint (see "Location Specifiers" on page 6-14). 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 3-26.

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 3-21.

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.

`tags` *tag-assignment [, tag-assignment...]*

This variant of the **patchpoint** command creates a patchpoint that sets one or more tag values in **$thr** for any thread that executes through the patchpoint (see "Thread Tags" on page 3-41 for more information on **$thr**).

Following the **tags** keyword, one or more comma separated *tag-assignments* should appear. A *tag-assignment* has the following syntax:

tagname assign-op expression

- *where, tagname* is a simple identifier that names the tag value to be set;

- *where assign-op* may be one of =, :=, -=, +=, *=, /=. The = and := operators are identical and represent simple assignment. The -=, +=, *=, and /= operators are like the C operators (but in a **tags** patch will work even for Ada and Fortran expression).

- *where expression* is the expression to evaluate and use in the assignment to the tag. Unless the *tagname* has been previously declared using the **declare-thread-tag** command, the expression must be a 1-bit boolean value (true, false, 1, or 0).

  To set a C string value, you need to copy string characters, so you should use the alternative `eval` form of **patchpoint**, e.g.:

  patchpoint file.c:23 eval strcpy($thr.strtag,"hamster")

  assuming *strtag* was declared with **declare-thread-tag** as a C char array.

NOTE

The keywords `name`, `at`, `eval`, and `goto` and `tags` 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 6-111) or **ignore** (see "ignore" on page 6-114) 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 6-40 and "disable" on page 6-113). 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.

Example:

> (local) **patchpoint file.c:12 tags working=true**

This C example patches the code to define the **$tags** field *working* the value *true* for each thread that executes line 12 of file.c.

set-trace

The **set-trace** command is used to specify information that may be useful before any tracepoints are set in a process (see "tracepoint" on page 6-104).

set-trace [eventmap=*event-map-file*][tracefile=*key-filename*]

> 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)**.
>>
>> 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
>
> tracefile=*key-filename*

This argument causes NightView to initiate application tracing by issuing the NightTrace API `trace_begin` call. **tracepoints** aren't useful if the application has not initalized the NightTrace API subsystem. If your application already calls `trace_begin` you should not use this argument.

The *key-filename* is passed as the first parameter to `trace_begin` (in the form of a string constant). The *key-filename* identifies the daemon session which collects the NightTrace events that are logged by the application and by NightView **tracepoint** eventpoints. See **ntraceud(1)** and **trace_begin(3)** for more information on NightTrace.

To specify a file on a remote system, use the form *user***@***host***:***key-filename*. See "Remote File Access" on page 3-7.

NightView will automatically linking in the required NightTrace API library modules if they don't already exist in the program.

tracepoint

Set a tracepoint.

tracepoint [*eventpoint-modifier*] *event-id* [name**=***tracepoint-name*]
                [[at] *location-spec*]
                [value=*logged-expression*[**,** *logged-expression*...]]
                [if *conditional-expression*]

eventpoint-modifier

Specifies the tracepoint modifier. See "Eventpoint Modifiers" on page 6-98.

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 6-103).

name**=***tracepoint-name*

Gives a name to the tracepoint for later reference. See "name" on page 6-98. 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 6-14.

If *location-spec* is omitted, then the location used is the next instruction to be executed in the current stack frame.

value=*logged-expression*[**,** *logged-expression*...]

> Specifies that the value of each *logged-expression* should be recorded with the trace event. The expressions are separated by commas. (To include a comma in an expression, surround the expression by parentheses.)
>
> The number of expressions and the type of the expressions must match a `trace_event` routine defined by the **ntrace** library. See the section "trace_event() and Its Variants" in the "Using the NightTrace Logging API" chapter of the *NightTrace User's Guide*.
>
> The expressions are evaluated in the user program, so they obey the same rules that conditional and patchpoint expressions do. See "Expression Evaluation" on page 3-21.

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 3-25 and "Context" on page 3-25). See also "Expression Evaluation" on page 3-21.

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 6-114).

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 6-40 and "disable" on page 6-113).

NOTE

> The debugger does *not* start a NightTrace daemon, which is required in order to actually collect the events logged with trace-points. You must do that using **ntraceud(1)** or the Daemons panel in **ntrace(1)** (see "NightTrace Daemon" on page 3-43).

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 eventpoint number, but all instances will share the same *event-id* you specified on the command.

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 6-98.
>
> name**=***monitorpoint-name*
>
> > Gives a name to the monitorpoint for later reference. See "name" on page 6-98. 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 6-14.
> >
> > 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 6-1.

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 3-28. For a more detailed description of monitorpoints, see "Monitorpoints" on page 3-11.

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 6-85), 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 6-110.

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).

heappoint

Check the heap for errors, or change the heap debugger settings, at a given location.

heappoint [*eventpoint-modifier*][name=*heappoint-name*] [[at] *location-spec*]
        [{check | debug *parameters*}}] [if *conditional-expression*]

eventpoint-modifier

> Specifies the heappoint modifier. See "Eventpoint Modifiers" on page 6-98.

name**=***heappoint-name*

> Gives a name to the heappoint for later reference. See "name" on page 6-98. If *heappoint-name* is already defined then this command adds the newly created heappoints to the list of eventpoints associated with the name.

location-spec

> Specifies the heappoint location. See "Location Specifiers" on page 6-14.

> If *location-spec* is omitted, then the location used is the next instruction to be executed in the current stack frame.

check

> Specifies that the heappoint does a heap check. This is the default if neither check nor debug is specified.

debug *parameters*

> Specifies that the heappoint changes the heap debugger settings. *parameters* are the same as the arguments to the **heapdebug** command (see "heapde-bug" on page 6-54).

if *conditional-expression*

> Specifies a heappoint condition. The language and scope of the expression is determined by the location at which the heappoint is set (see "Scope" on page 3-25 and "Context" on page 3-25). See also "Expression Evaluation" on page 3-21.

The **heappoint** command sets a heappoint in each of the processes specified by the qualifier. See "Heappoints" on page 3-12.

When the heappoint is executed, the process does a heap check if check was specified, or changes the heap debugger settings if debug was specified. The check and debug parameters are mutually exclusive.

Putting check heappoints at various places in your program can help you narrow down where heap problems are occurring.

Changing the heap debugger settings dynamically within your program can help you get reasonable performance while still getting strong heap checking. For example, if you have a suspicious section of code, you could set a heappoint at the beginning of the section to set automatic heap checks to occur before every heap operation, and set another heappoint at the end of the section to set automatic heap checks to occur only every 10,000 heap operations.

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 monitor-points 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 6-106.

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 monitor 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 3-36.

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 3-28).

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.

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 6-14.

> 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 3-26.

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 6-112). 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 6-14). 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 6-17.

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 6-1.

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 6-122, "resume" on page 6-123, and "signal" on page 6-132.

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 6-106.

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 6-17.
>
> *conditional-expression*
>
>> The condition to be associated with the eventpoints. See "Expression Evaluation" on page 3-21.

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, heappoints, patchpoints, watchpoints, and syscallpoints, causing the associated action to take effect only when the attached condition evaluates 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 6-17.

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 6-109). 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 6-65). 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 6-21). 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 6-17.

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 6-146). 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 6-113).

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 6-65). 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 6-21). 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 break-points 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 6-17.

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.

If *eventpoint-spec* is omitted and your safety level is unsafe then *all* eventpoints in the processes specified by the qualifier are enabled (see "set-safety" on page 6-65). If *eventpoint-spec* is omitted and your safety level is verify, then you are prompted for confirmation before the eventpoints are enabled (see "Replying to Debugger Questions" on page 6-21). If *eventpoint-spec* is omitted and your safety level is forbid then no eventpoints are enabled.

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 3-9.

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 6-98. 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 6-14.

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 3-25 and "Context" on page 3-25). See "Expression Evaluation" on page 3-21.

Note: The at, if, and name keywords may not be abbreviated in this command.

Like the **breakpoint** command (see "breakpoint" on page 6-99), 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 6-113.

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 6-98) may be used to give them names. See "Manipulating Eventpoints" on page 6-96.

at *location-spec*

Specify the exact point in the program to execute the patchpoint. See "Location Specifiers" on page 6-14. 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 3-26.

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 3-21.

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 6-100. 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 6-113.

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 6-98.

    /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.

        Watchpoint processing always occurs for a write, even if /write is omitted, because it is not possible to create a read-only watchpoint on an IA-32 or AMD64.

    /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 6-98). 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, in bytes.

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 variable $is. $is is the value of the watched item after the process has executed the instruction causing the trap. See "Watchpoints" on page 3-13.

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 3-13.

You can specify commands to be executed when the watchpoint is hit. See "commands" on page 6-110.

syscallpoint

Print system call information on entry and exit from system calls.

syscallpoint [*eventpoint-modifiers*] [name=*eventpoint-name*] [*syscall-list*] [if *conditional-expression*]

> *eventpoint-modifiers:*

> > /disabled
> > /delete
> > /once

> > > These modifiers have their standard meaning, as described in "Event-point Modifiers" on page 6-98.

> > /before
> > /after

> > > The /before and /after modifiers control when the syscallpoint applies (before the syscall is executed, after, or both). If both options are omitted, then the syscallpoint applies to both entry and exit of the service call.

> > /nostop

> > > The /nostop modifier causes the debugger to automatically resume execution of the process after it prints the system call information; thus manual intervention is not required. Regardless, the process *is* stopped while the debugger gets control and prints a message (and resumes the process unless /nostop is specified). Note that all threads are stopped during this time.

> > /except

> > > The /except modifier changes the meaning of the *syscall-lis*t. Instead of matching those system calls, it matches all except those system calls. The *syscall-list* must be non-empty if you use /except (because an empty *syscall-list* implies all system calls -- and using /except in such a circumstance would be useless).

> name**=***eventpoint-name*

> > Gives a name to the syscallpoint for later reference. See "name" on page 6-98. If *evenpoint-name* is already defined then this command adds the newly created syscallpoints to the list of eventpoints associated with the name.

syscall-list

> Specifies the list of system calls that you are interested in tracing. If *syscall-list* is omitted, it implies all system calls. Otherwise, *syscall-list* should be one or more valid system call names, separated by commas or spaces.
>
> The list of valid system call names can be found using the graphical interface in the Syscallpoint Dialog by pressing the Select… button (see "System Call Selection Dialog" on page 8-34).
>
> It is possible while debugging multiple processes from different target systems at the same time in the same NightView session that the list of valid system calls can differ between processes; although, the majority of system call names are common between Linux kernel versions.

`if` *conditional-expression*

> Specifies a syscallpoint condition. If specified, the expression is evaluated to see if the syscallpoint applies or not. This can test more complex conditions that merely the service call number. Since NightView cannot predict ahead of time where a syscallpoint might stop, the condition is always evaluated in the global scope.

The **commands** command (see "commands" on page 6-110) can be used to add commands to be executed when the process stops on an eventpoint. No commands are allowed if the /nostop modifier is specified.

Special Restrictions

The combination of /before and /after modifiers as well as the list of system calls must be unique across the entire process. You are not allowed to define multiple syscallpoints which have overlapping system calls. The /except option may be useful to define a catch-all syscallpoint that does not apply to any of the other syscallpoints already defined.

Due to the way the Linux kernel implements system call tracing, NightView is limited when stopped at a syscallpoint. Anything that requires NightView to execute the process will not work at a syscallpoint. This means that you cannot call functions in expressions (which may mean you can't do some operations internally implemented as functions that aren't obvious).

These restrictions also apply to the syscallpoint condition and any commands you may want to execute when the syscallpoint stops. For the same reason, you can't do things like modify the program counter in an expression and prevent the service call from completing.

Due to Linux internals, the system call number on exit from a system call is sometimes reported as -1 rather than the actual service number. This behavior is typically limited to special service calls used to do things like return from signal handlers.

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 6-133. If you want the process to receive a different signal, or no signal at all, then use the **signal** command. See "signal" on page 6-132.

NOTE

On Linux, there is no way to pass SIGSTOP to a process being debugged.

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.

set-run-mode

Controls whether all threads in a multi-threaded process resume execution when a single thread is told to resume execution.

set-run-mode *mode*

one

In this mode, commands or actions that cause a single thread to resume execution have no effect on other threads in the process. Just the current thread executes.

all

In this mode, commands or actions that cause a thread to resume execution cause all other threads in the process to resume execution as well.

The run mode is consulted when a thread's execution is resumed by an action in the graphical interface (e.g. pressing the resume icon, using a keyboard shortcut in a source panel, etc.) or when a command is typed that does not supply overriding run mode options (e.g. **/one** or **/all**).

IMPORTANT

The *one* mode should be used with care. It is not uncommon for a single thread to block waiting on operations to be completed by other threads (e.g. by **pthread_cond_wait(3)** and many other services that hold resources). If those other threads are not permitted to resume, the one thread that was resumed may block indefinitely. In such circumstances, stop the blocked thread and resume all threads using **/all** on your command or change the run mode to *all*.

continue

Continue execution and wait for something to happen.

continue [/one | /all] [*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.
>
> /one
>
>> This option causes only the current thread to resume execution. All other threads in the process remain stopped. If both this option and /all are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.
>
> /all
>
>> This option causes all threads in the process to resume execution. If both this option and /one are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

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 3-36.

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 6-123.

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 [/one | /all] [*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 6-132.
>>
>> If this argument is not present, then the processes are resumed with the signal that caused them to stop, similar to **continue**.

> **NOTE**
>
> On Linux, there is no way to pass SIGSTOP to a process being debugged.

> /one

This option causes only the current thread to resume execution. All other threads in the process remain stopped. If both this option and /all are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

/all

This option causes all threads in the process to resume execution. If both this option and /one are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

resume causes the processes or threads specified by the qualifier to resume execution at the point where they last stopped. The processes or threads 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 3-36.

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 6-122.

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 [/one | /all] [*repeat*]

Abbreviation: **s**

repeat

> The *repeat* argument specifies the number of lines to single step. The default is one line.

/one

> This option causes only the current thread to resume execution. All other threads in the process remain stopped. If both this option and /all are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

/all

> This option causes all threads in the process to resume execution. If both this option and /one are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

> It is worth noting that the /all option does **not** imply that all threads will single step. All threads will execute while the current thread steps and then all threads will be stopped when the step operation completes. The other threads may execute many source lines or only a few instructions.

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 6-126.

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 6-136.

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 3-15 for a discussion of the interactions between single-stepping and signals.

step is interpreted relative to the current frame. See "Current Frame" on page 3-26. That is, any lower frames are automatically finished before stepping.

There are also commands to single step individual instructions. See "stepi" on page 6-127 and "nexti" on page 6-128.

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 6-82). Also, disassembly can help you determine the flow of control through your program (see "x" on page 6-88 and "Source Menu" on page 8-10).

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 6-126.

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 [/one | /all] [*repeat*]

Abbreviation: **n**

>   *repeat*
>
>   > The *repeat* argument specifies the number of lines to single step. The default is one line.
>
>   /one
>
>   > This option causes only the current thread to resume execution. All other threads in the process remain stopped. If both this option and /all are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.
>
>   /all
>
>   > This option causes all threads in the process to resume execution. If both this option and /one are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.
>
>   > It is worth noting that the /all option does **not** imply that all threads will step over a single line. All threads will execute while the current thread steps over its current line and then all threads will be stopped when the step operation completes. The other threads may execute many source lines or only a few instructions.

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, including "inline" procedures. See "Inline Subprograms" on page 3-27. 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 6-124.

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 6-136.

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 3-15 for a discussion of the interactions between single-stepping and signals.

next is interpreted relative to the current frame. See "Current Frame" on page 3-26. That is, any lower frames are automatically finished before stepping.

There are also commands to single step individual instructions. See "nexti" on page 6-128 and "stepi" on page 6-127.

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 6-82). Also, disassembly can help you determine the flow of control through your program (see "x" on page 6-88 and "Source Menu" on page 8-10).

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 [/one | /all] [*repeat*]

Abbreviation: **si**

> *repeat*
>
>> The *repeat* argument specifies the number of instructions to single step. The default is one instruction.
>
> /one
>
>> This option causes only the current thread to resume execution. All other threads in the process remain stopped. If both this option and /all are omit-

ted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

`/all`

This option causes all threads in the process to resume execution. If both this option and `/one` are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

It is worth noting that the `/all` option does **not** imply that all threads will single step an instruction. All threads will execute while the current thread steps and then all threads will be stopped when the step operation completes. The other threads may execute many source lines or only a few instructions.

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 6-124.

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 3-15 for a discussion of the interactions between single-stepping and signals.

stepi is interpreted relative to the current frame. See "Current Frame" on page 3-26. 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 6-92.

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 6-128.

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 [/one | /all] [*repeat*]

Abbreviation: **ni**

repeat

> The *repeat* argument specifies the number of instructions to single step. The default is one instruction.

/one

> This option causes only the current thread to resume execution. All other threads in the process remain stopped. If both this option and /all are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

/all

> This option causes all threads in the process to resume execution. If both this option and /one are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

> It is worth noting that the /all option does **not** imply that all threads will step over a single instruction. All threads will execute while the current thread steps and then all threads will be stopped when the step operation completes. The other threads may execute many source lines or only a few instructions.

nexti executes a single machine instruction in each of the processes specified by the qualifier, except that **nexti** steps over procedure calls and inlined procedures. See "Inline Subprograms" on page 3-27.

This is very similar to **next**, except that **next** executes lines and **nexti** executes individual instructions. See "next" on page 6-126.

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 3-15 for a discussion of the interactions between single-stepping and signals.

nexti is interpreted relative to the current frame. See "Current Frame" on page 3-26. 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 [/one | /all]

/one

> This option causes only the current thread to resume execution. All other threads in the process remain stopped. If both this option and /all are omit-

ted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

`/all`

This option causes all threads in the process to resume execution. If both this option and `/one` are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

It is worth noting that the `/all` option does **not** imply that all threads will finish out of their current routines. All threads will execute while the current thread finshes executing the current routine and then all threads will be stopped. The other threads may execute only a few instructions, many source lines, or many functions.

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 6-136. If the current frame is in the context of a task, thread, or thread process chosen by the **select-context** command, execution continues until that task, thread, or thread process completes execution of that procedure, or until the process stops for some other reason. See "Multithreaded Programs" on page 3-40.

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 3-15 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 [/one | /all] [at] *location-spec*

> *location-spec*
>
> > The *location-spec* specifies where to continue execution. See "Location Specifiers" on page 6-14.
>
> /one
>
> > This option causes only the current thread to resume execution. All other threads in the process remain stopped. If both this option and /all are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.
>
> /all
>
> > This option causes all threads in the process to resume execution. If both this option and /one are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.
> >
> > It is worth noting that the /all option does **not** imply that all threads will begin execution at the specified location. Only the current thread will resume execution at the specified location; all other threads will resume execution from their current PC.

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 [/one | /all] *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.
>
> /one
>
>> This option causes only the current thread to resume execution. All other threads in the process remain stopped. If both this option and /all are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.
>
> /all
>
>> This option causes all threads in the process to resume execution. If both this option and /one are omitted, the current run mode controls which threads execute (see "set-run-mode" on page 6-121). This option is redundant for single-threaded processes.

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 3-36.

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 6-133. **signal** overrides the pass setting of **handle**.

NOTE

On Linux, there is no way to pass SIGSTOP to a process being debugged.

Type **info signal** to get a list of all of the signals on your system. See "info signal" on page 6-157.

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** command 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 8 [Graphical User Interface] on page 8-1. See "Message Panel" on page 8-61.

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.

NOTE

There is no way to pass SIGSTOP to a process being debugged.

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 3-15 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 3-15.)

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**

Ada programs use some signals in the run-time library, so, by default, NightView sets these to nostop, noprint, pass. These are SIGABRT, SIGFPE, SIGSEGV and signal 47.

For a way to deal with signals one at a time, see "signal" on page 6-132.

To find out the current settings for all the signals, see "info signal" on page 6-157.

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 6-164.

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 6-85).
>
> *expression*
>
>> Expression which yields an address at which the stack frame should start. This is the value that `$cfa` would have, not the value of `$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 6-14. 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 6-154.

If a *frame-number* is given, the chosen stack frame is selected as the current frame (see "Current Frame" on page 3-26).

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 3-26). If a negative number is specified, then frames are advanced toward the newest stack frame (see "down" on page 6-138).

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 3-26). If a negative number is specified, then frames are advanced toward the oldest stack frame (see "up" on page 6-137).

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 a thread process.

select-context default

select-context task=*expression*

select-context thread=*expression*

select-context pid=*pid*

select-context name=*name*

select-context stackaddress=*expression*

 default

> This keyword selects the stack frame for the context where the process has stopped. If the process has threads, the default context is the thread process that stopped the process. See "Multithreaded Programs" on page 3-40.

 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.

pid=*pid*

> The `pid=` keyword selects the context of a specific thread process. The *pid* is the ID of the thread process whose context is selected.

name=*name*

> The `name=` keyword selects the context of a specific thread by matching the supplied name; *name* must identify a single thread in the current process. See "set-thread-name" on page 6-143 for more information on thread names.

stackaddress=address

> The `stackaddress=` keyword selects the context of a specific thread by matching the supplied address; *address* must be a process address value inside some thread's stack.

> This can be useful if you know the `pthread_self()` values reported by threads, but you are looking at a core file which has lost the thread associations. For all but the main thread, the `pthread_self()` value is normally contained within the stack segment for the associated thread.

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 a thread process. Using **select-context**, you can get a backtrace (see "backtrace" on page 6-85) and examine registers and variables in the context of the selected task, thread, or thread process.

When a process that contains multiple tasks, threads or thread processes stops, the current context becomes that of one specific task, thread, or thread process. (For a discussion of how this choice is made, see "Multithreaded Programs" on page 3-40.) You can use the **select-context** command to temporarily change the context to that of some other task, thread, or thread process.

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.

Note that execution control is on a process basis: if you resume execution, all threads are allowed to execute. If you enter a **finish**, **step**, **next**, **stepi**, or **nexti** command, the process executes until the selected task, thread, or thread process completes the stepping operation, but other tasks, threads, or thread processes may execute as well.

If you request evaluation of an expression containing a function call, the process is allowed to execute and all threads are allowed to run. If another thread hits a breakpoint, or stops for some other reason, the function call is terminated prematurely and an error message is issued.

To get a list of the tasks, threads, and thread processes in a process, see "info threads" on page 6-164.

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 3-35.

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 8-1.

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

Re-read source files and refresh the terminal screen.

refresh

NightView normally notices when source files have changed when it switches the current source display. If you want to update the display without switching files, or if you suspect that a system problem has fooled the automatic mechanism, you can force source files to be read again with the **refresh** command.

In the simple full-screen interface, the **refresh** command also clears the terminal screen and redraws it. This is helpful when the screen becomes garbled, such as with a modem and noisy phone lines. See Chapter 7 [Simple Full-Screen Interface] on page 7-1.

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 6-170).

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 6-75).

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 7 [Simple Full-Screen Interface] on page 7-1.

source

Input commands from a source file.

source *command-file*

command-file

The file to read.

To specify a file on a remote system, use the form *user*@*host***:/***path*. See "Remote File Access" on page 3-7.

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 3-36.

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.

declare-thread-tag

Declare the type of a thread tag.

declare-thread-tag *tagname typespec*

A thread tag that is used without being declared is always a 1 bit boolean flag value. Use **declare-thread-tag** to give the tag a different type before assigning to the tagname in a **patchpoint** command, or manually via

set $thr.tagname = ...

The *typespec* consists of one or two parts. The first part names the type, and the second (optional) part declares it to be a simple one dimensional array with a fixed size.

The name of the type can be specified two different ways:

It can be a type from the program, as long as it is a type that can be referenced with a name (where compound names are allowed). The compound name syntax will accept either **.** or **::** to separate the name components. The leading components should name scopes to look in for the next component. The last component should be the name of a type.

It can also be a simple C-like type composed of some (sensible) combination of the keywords: **bool**, **char**, **short**, **int**, **long**, **float**, **double**, **signed**, **unsigned**.

The optional array declaration consists of a C-like open bracket **[** followed by a positive integer constant, followed by a close bracket **]**.

For example,

```
declare-thread-tag foo int
declare-thread-tag bar char[20]
declare-thread-tag zort TypesClass::VectorType
declare-thread-tag color MyPackage.Types.Colors
```

The latter two invocations referring to types declared in a source code present in your program; for example:

```
// C++ Example
class TypesClass {
public:
   typedef float VectorType[10];
};

-- Ada Example
package MyPackage is
   type Colors is (blue, green, chartreuse, vermilion);
end MyPackage;
```

See "Thread Tags" on page 3-41 for more information.

set-thread-name

Set the name of a thread.

set-thread-name [modifier] *"name"*

Where modifier, if supplied, is one of the following:

name=*"current_name"*

> The name= keyword selects the thread that matches the supplied *current_name*. This command will fail if the specified *current_name* is not unique among the current list of threads. In this form of the command, two names are specified: the *current_name* which is used to identify the thread; and *name*, the thread's new name. Both names must be specifed as quoted strings.

task=*task_name_or_id*

> The task= keyword selects the thread by it's Ada task name or its Ada task ID, which is a pointer value assigned by the Ada runtime.

thread=*thread_id*

> The thread= keyword selects the thread by it's thread ID value, as returned by **pthread_self(3)** and **pthread_create(3)**.

pid=*pid*

> The pid= keyword selects the thread by its **gettid(2)** value; displayed as its pid in NightView. Under Linux, the main thread's **getpid(2)** and **gettid(2)** values are identifcal; all other threads in the process share the same **getpid(2)** value, but have unique **gettid(2)** values.

If modifier is omitted, then the command applies to the current thread.

The **set-thread-name** command allows you to change the thread name used by NightView to further identify the thread in **info-thread** commands and in the

Context Panel and in Context, Process, and Thread displays in Data Panels.

By default, NightView automatically assigns names to threads using the name of the start routine whose address is passed to **pthread_create(3)** when the thread is created.

This command overrides that default name.

User defined names are not preserved on restart, so any names assigned to threads using the **set-thread-name** command are not automatically reassigned when rerunning the process. For a way to identify threads across restart, see "Thread Tags" on page 3-41.

This command does not select a thread to be the "current thread" in NightView; it merely changes the specified thread's name.

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 6-60) or by **set-show** (see "set-show" on page 6-34).

info eventpoint

Describe current state of breakpoints, tracepoints, patchpoints, monitorpoints, heappoints, watchpoints, and syscallpoints.

info eventpoint [/verbose] [*eventpoint-spec*] **...**

> /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 6-62). The verbose keyword may be abbreviated.

> *eventpoint-spec*
>
> > An eventpoint specifier, which is an eventpoint name, an eventpoint number, or the keyword **at** followed by a location specifier. See "Eventpoint Specifiers" on page 6-17.

This command describes eventpoints associated with the processes in the command qualifier. An eventpoint is any of a breakpoint, tracepoint, patchpoint, monitorpoint, heappoint, watchpoint, or syscallpoint. See "breakpoint" on page 6-99, "tracepoint" on page 6-104, "patchpoint" on page 6-100, "monitorpoint" on page 6-106, "watchpoint" on page 6-117, "syscallpoint" on page 6-119, and "heappoint" on page 6-107.

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, monitor-point, watchpoint, or syscallpoint).

- For heappoints, the word `check` if the heappoint does a heap check, or the word `debug` followed by the new settings if the heappoint changes the heap debugger settings.

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 6-88 and "Predefined Convenience Variables" on page 6-6.

info breakpoint

Describe current state of breakpoints.

info breakpoint [/verbose] [*eventpoint-spec*] ...

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 6-62). The verbose keyword may be abbreviated.

> *eventpoint-spec*
>
>> An eventpoint specifier, which is an eventpoint name, an eventpoint number, or the keyword **at** followed by a location specifier. See "Eventpoint Specifiers" on page 6-17.

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 6-99.

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 6-88 and "Predefined Convenience Variables" on page 6-6.

info tracepoint

Describe current state of tracepoints.

info tracepoint [/verbose] [*eventpoint-spec*] ...

> /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 6-62). The verbose keyword may be abbreviated.

> *eventpoint-spec*
>
>> An eventpoint specifier, which is an eventpoint name, an eventpoint number, or the keyword **at** followed by a location specifier. See "Eventpoint Specifiers" on page 6-17.

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 6-104.

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 6-88 and "Predefined Convenience Variables" on page 6-6.

info patchpoint

Describe current state of patchpoints.

info patchpoint [/verbose] [*eventpoint-spec*] ...

/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 6-62). The verbose keyword may be abbreviated.

eventpoint-spec

An eventpoint specifier, which is an eventpoint name, an eventpoint number, or the keyword **at** followed by a location specifier. See "Eventpoint Specifiers" on page 6-17.

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 6-100.

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 6-88 and "Predefined Convenience Variables" on page 6-6.

info monitorpoint

Describe current state of monitorpoints.

info monitorpoint [/verbose] [*eventpoint-spec*] ...

/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 6-62). The verbose keyword may be abbreviated.

eventpoint-spec

An eventpoint specifier, which is an eventpoint name, an eventpoint number, or the keyword **at** followed by a location specifier. See "Eventpoint Specifiers" on page 6-17.

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 6-106.

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 6-88 and "Predefined Convenience Variables" on page 6-6.

info heappoint

Describe the current state of heappoints.

info heappoint [/verbose] [*eventpoint-spec*] ...

> /verbose
>
>> Specify that the locations of all heappoints displayed will be in verbose format. Verbose location format includes the program counter address (or addresses) of the heappoint 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 6-62). The verbose keyword may be abbreviated.

> *eventpoint-spec*
>
>> An eventpoint specifier, which is an eventpoint name, an eventpoint number, or the keyword **at** followed by a location specifier. See "Eventpoint Specifiers" on page 6-17.

This command describes heappoints in the processes indicated by the qualifier. Normally all heappoints are described, but if an argument is given, only those named are described. Any eventpoints specified in the argument list which are not heappoints are ignored. Heappoints are created with the **heappoint** command. See "heappoint" on page 6-107.

The information printed includes:

- The heappoint ID.

- Current state of heappoint (enabled, disabled, temporary).

- The heappoint 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 heappoint was created.

- The number of times program execution has crossed the heappoint since the program started execution (even if the ignore count or condition was not satisfied, this count is incremented).

- The number of times the heappoint 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 heappoint.

- The current ignore count.

- The word check if this heappoint does a heap check, or the word debug followed by the new settings if this heappoint changes the heap debugger settings.

This command sets the default **x** command dump address as well as the $_ predefined convenience variable to the address of the last heappoint listed. See "x" on page 6-88 and "Predefined Convenience Variables" on page 6-6.

info watchpoint

Describe current state of watchpoints.

info watchpoint [/verbose] [*eventpoint-spec*] ...

/verbose

The verbose keyword is accepted for compatibility with other watchpoints, but is ignored. The verbose keyword may be abbreviated.

eventpoint-spec

An eventpoint specifier, which is an eventpoint name, an eventpoint number, or the keyword **at** followed by a location specifier. See "Eventpoint Specifiers" on page 6-17.

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 6-117.

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 `#cross-ings` (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 syscallpoint

Describe current state of watchpoints.

info syscallpoint [*eventpoint-spec*] ...

    eventpoint-spec

        An eventpoint specifier, which is an eventpoint name or an eventpoint number. See "Eventpoint Specifiers" on page 6-17.

This command describes syscallpoints in the processes indicated by the qualifier. Normally all syscallpoints are described, but if an argument is given, only those named are described. The name or number identifies the syscallpoint and not the system calls themselves.

Any eventpoints specified in the argument list which are not syscallpoints are ignored. Syscallpoints are created with the **syscallpoint** command. See "syscallpoint" on page 6-119.

The information printed includes:

- The syscallpoint ID.

- Current state of the syscallpoint (enabled, disabled, temporary).

- The system call(s) being traced.

- The number of times the process encounterd a matching system call since the program started execution. This count is incremented even if the ignore count or condition was not satisfied. This number is displayed as `#cross-ings` (for consistency with other eventpoint types).

- The number of times the syscallpoint 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 syscallpoint.

- The current ignore count.

- Any commands attached to the syscallpoint.

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 $cfa 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 6-14. 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 3-26). See also "frame" on page 6-136.

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 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).

- How $fp is computed for the frame.

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 6-78, 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 6-66). 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 6-156).

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 6-92).

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 6-62.

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 6-62.

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 6-18.

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 6-6.

The regular expression case sensitivity depends on the current search mode (see "set-search" on page 6-71).

Registers are printed relative to the current frame (see "Current Frame" on page 3-26). 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 6-136).

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 6-133).

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, running one thread, stopped).

- When the process state is stopped, list where and why it stopped.

- The current language setting. See "set-language" on page 6-60.

- The disposition of child processes; that is, under what circumstances a child process will be debugged. See "set-children" on page 6-51.

- If the current source file resides on a remote system, its name is preceded by *user@host*. Local source files will not have that designation. If the current source file is different than that recorded in the program's debug information, both source paths are listed.

- The run-mode of the process (see "set-run-mode" on page 6-121).

info memory

Print information about memory, which may include information about the virtual address space, or the heap.

info memory [/ranges] [/heap] [/leaks] [/allocated]
        [/all] [/append=*filename*] [/output=*filename*]
        [/verbose] [*expression*]

/ranges

If this option is specified, the command prints information about the virtual address space. If an expression is specified, then it should evaluate to an address, and only information about the region that contains that address is displayed. If no expression is specified, then all regions in each process are displayed. For each region of memory displayed, this command displays 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 created by NightView, what the region is for and how much space is left in the region. See "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 6-53.

/heap

If this option is specified, the command prints information about the heap. This option is supported only for processes which have turned on heap debugging.

If an expression is specified, then it should evaluate to an address, and the command displays the following heap information about the memory block that contains the address, if any:

- its state, which will be one of:

    - allocated

    - freed, but retained

    - freed or never allocated, but owned by heap

    - not owned by heap

    In the case of the latter two, no further information is displayed.

- the address range of its memory block

- its size in bytes

- descriptions of any errors detected pertaining to it

- information pertaining to each heap operation (allocation, most recent realloc, and free) that has happened for the block, including this information for each:

    - number and address range of post-fence bytes and the post-fence fill byte

    - number and address range of pre-fence bytes and the pre-fence fill byte

    - number of slop bytes

    - whether free filling was enabled and the free fill byte

    - whether malloc filling was enabled and the malloc fill byte

    - whether hardware overrun protection was enabled

    - walkback of stack frames at the time of that operation, as restricted by the walkback setting at that time

If no expression is specified, the command displays the following global heap information:

- totals, including:

    - number of blocks ever allocated

    - number of bytes ever allocated

    - number of additional bytes of debugger overhead ever allocated

    - number of blocks ever freed

    - number of bytes ever freed

    - number of additional bytes of debugger overhead ever freed

    - number of blocks currently allocated

- number of bytes currently allocated

- number of additional bytes of debugger overhead currently allocated

- number of blocks currently freed but still retained

- number of bytes currently freed but still retained

- number of additional bytes of debugger overhead currently freed but still retained

- whether heap debugging is on or off

- number of post-fence bytes and the post-fence fill byte

- number of pre-fence bytes and the pre-fence fill byte

- number of slop bytes

- whether free filling is enabled and the free fill byte

- whether malloc filling is enabled and the malloc fill byte

- whether hardware overrun protection is enabled

- frequency of automatic heap checks (i.e. the number of heap operations between automatic heap checks)

- maximum heap size, which may be "unlimited"

- maximum number of retained free blocks, which may be "unlimited"

- maximum number of walkback frames per heap operation

- whether or not to check fill bytes of free blocks

`/leaks`

> If this option is specified, the command prints information about heap blocks which very likely have leaked. See "Leak Detection" on page 3-34 for accuracy limitations on leak detection. An expression may not be specified with this option. This option is supported only for processes which have turned on heap debugging.
>
> If the `/all` option is specified, then all leaks will be displayed. Otherwise, only new leaks since the last leak report will displayed.
>
> If the `/verbose` option is omitted, then heap blocks are reported as sets. A set contains all heap blocks with identical sizes and walkbacks at the time of their allocations (or most recent `realloc`s), regardless of other characteristics. For each set, the following information is reported:

- number of blocks in the set

- block size of the blocks in the set

- walkback of stack frames, as restricted by the walkback setting at that time of the allocation (or most recent `realloc`) operation, of the blocks in the set

If the `/verbose` option is specified, then each heap block is reported individually. For each block, the following information is reported:

- its beginning address

- its size in bytes

- walkback of stack frames at the time of the allocation (or most recent `realloc`) operation, as restricted by the walkback setting at that time

`/allocated`

If this option is specified, the command prints information about heap blocks which are still allocated. An expression may not be specified with this option. This option is supported only for processes which have turned on heap debugging.

If the `/all` option is specified, then all blocks still allocated will be displayed. Otherwise, only blocks still allocated and allocated since the last still allocated blocks report will displayed.

The format of the output is identical to that for the `/leaks` option. The only difference is that all allocated blocks are reported, instead of only those which are determined to be leaks.

`/all`

This option is meaningful only with the `/leaks` or `/allocated` options. See the descriptions of those options for its effect.

`/append=`*filename*

Write the output of the command to the specified *filename*, appending to any existing contents.

`/output=`*filename*

Write the output of the command to the specified filename, replacing any existing contents.

`/verbose`

Indicates that extra information should be printed.

This command displays information about the virtual address space, or about the heap, for each process specified in the command qualifier. If no options are specified, the default behavior is like that for the `/ranges` and `/heap` options.

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 6-66.

- The list of **debug** and **nodebug** patterns for this dialogue. See "debug" on page 6-26.

- 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 6-34).

- The list of object filename translations for this dialogue. See "translate-object-file" on page 6-28.

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 6-18.

For each family name that matches *regexp* this command lists each process that is a member of that family (see "family" on page 6-50). 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 6-71).

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 6-18.

For each eventpoint-name that matches *regexp*, this command lists each eventpoint that is a member of that eventpoint-name (see "name" on page 6-98). 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 6-71).

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 6-30

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 6-46.

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 6-48.

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 6-141. See "Restarting a Program" on page 3-16. For an example, see "checkpoint" on page 6-49.

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 6-133. 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.

info threads

Describe Ada tasks, C threads, and thread processes.

info threads [/verbose]

> /verbose
>
>> Show all thread tags for each thread, using a multi-line display. By default, only values for non-zero simple thread tag types are shown.

This command describes the Ada tasks, C threads, and thread processes for the processes specified by the qualifier. The identifiers listed for each thread type may be used with the

`select-context` command to switch to that thread. See "select-context" on page 6-138.

The current thread is marked with the leading characters =>.

If only one thread is running (see "set-run-mode" on page 6-121) then the leading characters => are followed by the word Running for the running thread.

For each thread, the following information is printed:

- The process ID (PID) for the thread. This is the value that would be returned by the gettid() system call (which is not available as a library call). It is also the value you would see for the PID in a **ps(1)** listing if you asked to see thread information. For Linux threads, all threads share the same value as returned by getpid(2), but each thread has its own gettid() value.

- The pthread_t value for the thread, which is assigned by **pthread_create(3)** and returned by **pthread_self(3)**.

- The name of the start routine of the thread, if NightView is able to locate it. This is the name of the user's function whose address is passed to **pthread_create(3)**. Note that for Ada tasks, the start routine is omitted, because a more accurate name of the corresponding Ada task is shown before the PID.

- A list of all simple **Thread Tags** which have non-zero values. See "Thread Tags" on page 3-41 for information on setting user-defined, thread-specific values. Only tags with simple types and those which are similar to a C char[] are shown by default; to see all thread tag values, using the /verbose option.

In the graphical user interface, you can also see this thread information from the Display menu. See "Data Menu" on page 8-14.

heapcheck

Check the heap for errors.

heapcheck [/all] [/append=*filename*] [/output=*filename*] [*expression*]

> /all
>
>> Report all existing heap errors. Without this option, the only errors shown are the ones that have occurred since the most recent **heapcheck** command, automatic heap check performed by the process during heap operations (see "Heap Check" on page 3-34), or heappoint check (see "heappoint" on page 6-107).
>
> /append=*filename*
>
>> Write the output of the command to the specified filename, appending to any existing contents.

/output=*filename*

> Write the output of the command to the specified filename, replacing any existing contents.

The **heapcheck** command checks the heap for errors, such as overwritten fences, for each process in the qualifier. See "Fences" on page 3-32. If an expression is specified, then it should evaluate to a heap address, and only the block that contains that address is checked. If no expression is specified, then all heap blocks are checked. Heap debugging must have been turned on already, via the **heapdebug** command (see "heapdebug" on page 6-54) or the Debug Heap… item in the Process menu (see "Process Menu" on page 8-8).

The output is identical to error reporting when heap errors are discovered by automatic heap checks, or by a heappoint check. Possible errors are:

- post-fence modified in allocated block (value=address)

- pre-fence modified in allocated block (value=address)

- free-fill modified in free block (value=address)

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 3-26).

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 6-18.

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 6-71).

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 6-18.

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 6-71).

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 6-85).

info sources

List names of source files.

info sources [/v] [*pattern*]

>   *pattern*
>
>>   Wildcard pattern to match against source file names. See "Wildcard Patterns" on page 6-19.

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 the **/v** option is supplied, it lists the full pathnames of the files as recorded in the debug tables.

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 6-18.

This command lists the names of functions, or 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 6-71).

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 6-18.

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 6-71).

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 3-21.

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 3-21.

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 6-18.

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 6-71).

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). If the file resides on a remote system, its name is preceded by *user@host*. Local files will not have that designation.

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 6-14). 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 6-62). If this is the first range for an instance of the line, the address range is preceded by @.

- 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 6-88 and "Predefined Convenience Variables" on page 6-6.

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 6-173 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 6-1.)

The **define** command associates a body of text with the given *macro-name*. When the macro is invoked (see "Referencing Macros" on page 6-173), 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 6-173 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) printf "The value is %s\n", @x
(local) printf "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) define mymac body_does_not_matter
(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 C [Summary of Commands] on page C-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 6-95.

- Macros appearing in an **echo** command are expanded. See "echo" on page 6-91.

- Macros appearing in a **!** (see "!" on page 6-33), **run** (see "run" on page 6-37), or **shell** (see "shell" on page 6-141) 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.

- If the @ character is preceded by a non-whitespace character, the macro is expanded only if the macro name is enclosed between '{' and '}'.

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.

```
(local) directory user@{host}:/mydir  # a macro invocation
(local) directory user@host:/mydir  # not a macro invocation
```

In this example, the @ character is preceded by "r", so you must use '{' and '}' to have the macro expanded.

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 6-18.

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 "setsearch" on page 6-71).

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.

7
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 6-34.)

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 6-141.

The simple full-screen interface creates a special window when you use monitorpoints. See "Monitor Window - Simple Full-Screen" on page 7-2 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 6-71.

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 3-28.

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 6-108)

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 3-28.

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.

8
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 the Qt toolkit. 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 and working with dialog boxes and standard controls. Use mouse button 1 when you are told to click, drag, press, and select.

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 4 [Tutorials] on page 4-1.

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 Online Help

The graphical user interface provides several ways of providing help on particular topics.

- Context-sensitive help is available in the main window and all the dialogs. See "Context-Sensitive Help" on page 8-2.

- The main window has a Help menu. See "Help Menu" on page 8-18.

- Pressing the F1 function key displays help for the part of the window that has the current focus.

- The dialog boxes have help buttons that pop up help for the particular dialog box.

- You can use the **help** command from the command-line interface. See "help" on page 6-140.

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 8-88.

Context-Sensitive Help

Context-sensitive help is available through the Help menu. See "Help Menu" on page 8-18. In addition, 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 Kill button, the help window displays information about the process toolbar. See "Process Toolbar" on page 8-20.

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 key, select a window component that you have a question about. Press the F1 key. A help window is displayed with information about that region.

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 8-2.

Help Command

You can type the **help** command, followed by the topic you want help on, into the command toolbar to obtain online help. See "help" on page 6-140. A help window is displayed that contains information about the requested topic. See "Help Window" on page 8-88. See "Command Toolbar" on page 8-20.

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.

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.

Context Menu

Each panel has a context menu with entries appropriate for that panel type. You show the context menu by right-clicking in the panel.

In the source panel, where you click may affect the operation of the menu items by setting the source panel target line. See "Source Panel Context Menu" on page 8-54 and "Source Panel Target Line" on page 8-54.

In the data panel and related panels, different context menu entries appear depending on where you click. In some cases the menu entry's operation is directed to the data item on which you click. See "Data Panel Context Menu" on page 8-74.

In the eventpoint panel, you may select one or more eventpoints (rows) and then right-click. If you right-click on a row that is not selected, the selection is cleared and the row you clicked on becomes selected. If you right-click on a row that is selected, the selection does not change. The context menu's entries are enabled or disabled based on which rows are selected.

Current Process

In the graphical user interface, NightView has the concept of a current process. When you click on toolbar buttons, or enter commands, the operation is performed on the current process. (If you are debugging only one process, that process is the current process.)

The status bar shows the status of the current process. See "Status Bar" on page 8-24. A locals panel shows variables in the current frame of the current process. See "Locals Panel" on page 8-64. If you have more than one process, a context panel shows the current process with green underlined text. See "Context Panel" on page 8-64.

You can switch to a different process by clicking on the other process in a context panel or by clicking on a source panel displaying source for the other process.

GUI Configuration

NightView can save your current GUI configuration. The configuration includes the geometry of the main window, the positions of the toolbars, which pages are present, which panels are present on each page, and the geometry and other information about each panel.

When NightView starts up, it looks for a configuration in the following places, in order.

- filename supplied with the **-config** option
- **.NightView_config** in the current directory
- $HOME**/.NightView_config**
- **/usr/lib/NightView/lib/config**

You can explicitly load or save a configuration from the File menu. See "File Menu" on page 8-4.

Main Window

The main window has a menu bar, toolbars and a status bar. The remaining space is for docking various panels. See "Panels" on page 8-52. NightView can remember the arrangement of the panels. See "GUI Configuration" on page 8-3.

Menu bar

From the menu bar you can perform global NightView actions, perform actions on a shell or a process, choose source to display or edit, manipulate eventpoints, change the way you view the window, select items to display, invoke other NightStar tools, and obtain online help.

File Menu

Mnemonic: F

The File menu has the following items.

Load Config File...

Mnemonic: O

This brings up a file selection dialog to load a configuration. See "GUI Configuration" on page 8-3.

Load System Default Config

Mnemonic: D

This loads the default configuration in **/usr/lib/NightView/lib/config**. This is useful if you have modified your configuration and want to return to the default state. See "GUI Configuration" on page 8-3.

Save Config File

Mnemonic: S

Accelerator: Ctrl+S

If the configuration was loaded from the default, **/usr/lib/NightView/lib/config**, then this saves the configuration to $HOME**/.NightView_config**. Otherwise, this saves the configuration to the configuration file previously loaded or saved. See "GUI Configuration" on page 8-3.

Save Config File As...

Mnemonic: A

This brings up a file selection dialog to save the configuration. See "GUI Configuration" on page 8-3.

Preferences...

Mnemonic: F

This brings up a dialog that lets you set preferences, such as fonts. See "Preferences Dialog Box" on page 8-39.

Save Preferences

Mnemonic: V

This saves the current preference settings to disk.

Print Window...

Mnemonic: P

This brings up a dialog that lets you print an image of the main window to a printer.

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 6-23.

Depending on the safety level (see "set-safety" on page 6-65), NightView displays a warning dialog box when you click the Exit menu item if there are any active processes.

View Menu

Mnemonic: V

The View menu lets you create new panels, modify pages, change the size of text in panels, or modify which toolbars are shown.

New Context Panel

Mnemonic: C

Selecting this menu item creates a new context panel. See "Context Panel" on page 8-64.

New Locals Panel

Mnemonic: L

Selecting this menu item creates a new locals panel. See "Locals Panel" on page 8-64.

New Source Panel

Mnemonic: S

Selecting this menu item creates a new source panel. See "Source Panel" on page 8-53.

New Data Panel

Mnemonic: D

Selecting this menu item creates a new data panel. See "Data Panel" on page 8-64.

New Monitor Panel

Mnemonic: M

Selecting this menu item creates a new monitor panel. See "Monitor Panel" on page 8-64.

New Shell Panel

Mnemonic: H

Selecting this item creates a new shell panel. See "Source Panel" on page 8-53.

If there is more than one shell, then this item opens a sub-menu from which you can choose which shell the new panel represents.

New Message Panel

Mnemonic: G

Selecting this item creates a new message panel. See "Message Panel" on page 8-61.

New Eventpoint Panel

Mnemonic: E

Selecting this item creates a new eventpoint panel for the current shell. See "Eventpoint Panel" on page 8-61.

Add Page

Mnemonic: A

Accelerator: Ctrl+A

Selecting this item creates a new tabbed page.

Rename Current Page...

Mnemonic: R

Selecting this item brings up a dialog box that lets you give the current page a new name. See "Rename Page Dialog Box" on page 8-51. You can create a mnemonic that switches to the page: put an ampersand (&) in front of one of the characters of the name. The page's tab will have an underscore under that character. To get a real &, use &&.

Delete Current Page

Mnemonic: T

Selecting this item deletes the current page.

Text Size

Mnemonic: Z

Selecting this item opens a sub-menu that lets you select the size of text in the panels.

Increase

Mnemonic: I

Accelerator: Ctrl++

Selecting this item increases the size of the text in the panels. Panels that use a fixed-width font and panels that use a variable-width font are adjusted separately.

Decrease

Mnemonic: D

Accelerator: Ctrl+-

Selecting this item decreases the size of the text in the panels. Panels that use a fixed-width font and panels that use a variable-width font are adjusted separately.

Normal

Mnemonic: N

Accelerator: Ctrl+0 (zero)

Selecting this item resets the size of the text in the panels to normal.

Toolbars

Mnemonic: B

Selecting this item opens a sub-menu that lets you choose which toolbars are shown. See "Toolbars" on page 8-19.

Shell Menu

Mnemonic: L

The Shell menu lets you start a remote shell, terminate a shell, or create a new shell panel.

Start Remote Shell...

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 8-36.

Terminate shell

Mnemonic: T

Selecting this item terminates the dialogue. This is similar to using the **logout** command. See "logout" on page 6-30.

If there is more than one shell, then this item opens a sub-menu from which you can choose the shell to terminate.

Depending on the safety level (see "set-safety" on page 6-65) and whether there are any active processes, NightView may display a warning dialog box when you use the Terminate shell menu item.

New Shell Panel

Mnemonic: H

Selecting this item creates a new shell panel for the current shell. See "Source Panel" on page 8-53.

If there is more than one shell, then this item opens a sub-menu from which you can choose which shell the new panel represents.

Process Menu

Mnemonic: P

This menu is used to perform actions on processes.

Run...

Mnemonic: R

Use this item to run a program. Selecting this item pops up a dialog box for you to enter a shell command line. See "Run Program in Shell Dialog Box" on page 8-27. If your program takes no input and you want to debug more than one program at a time, end the command with &.

If there is more than one shell, then this item opens a sub-menu from which you can choose in which shell to run the program.

Another way to run a program is to type a command into a shell panel. See "Shell Panel" on page 8-60.

Attach...

Mnemonic: A

Selecting this item pops up a dialog box you can use to view the processes on the system and attach to one of them. See "Attach Dialog Box" on page 8-27.

If there is more than one shell, then this item opens a sub-menu from which you can choose in which shell to attach.

Detach

Mnemonic: D

Selecting this item causes NightView to detach from the current process. See "Current Process" on page 8-3.

This is similar to using the **detach** command. See "detach" on page 6-40.

Depending on the safety level (see "set-safety" on page 6-65), NightView may display a warning dialog box when you use the Detach menu item.

Kill

Mnemonic: K

Selecting this item causes NightView to terminate the current process. See "Current Process" on page 8-3.

This is similar to using the **kill** command. See "kill" on page 6-40.

Depending on the safety level (see "set-safety" on page 6-65), NightView may display a warning dialog box when you use the Kill menu item.

Debug Heap...

Mnemonic: H

Selecting this item pops up the Debug Heap dialog box, which allows you to turn on and adjust heap debugging for the current process. See "Debug Heap Dialog Box" on page 8-34. See "Debugging the Heap" on page 3-30. See "Current Process" on page 8-3.

Process Settings...

Mnemonic: S

Selecting this item pops up the Process Settings dialog box, which allows you to change how NightView treats the current process. See "Process Settings Dialog Box" on page 8-50. See "Current Process" on page 8-3.

Refresh Shared Libs

Mnemonic: L

Selecting this item causes NightView to re-read the list of shared libaries in use by the process and to refresh any debug information that may have changed or is new. This is not normally necessary, however, if your process dynamically loads libraries via **dlopen(2)**, NightView needs to be informed.

You can also set a mode to have NightView automatically detect this; there is some overhead involved however when that mode is set to automatic. See ... and ... for more information.

Source Menu

Mnemonic: S

This menu provides ways of changing the program code displayed in source panels and editing source files that are listed. See "Source Panel" on page 8-53.

The items that change which source file is displayed act on all source panels displaying the current process. The items that select source or disassembly are enabled only if there is exactly one source panel, or if there is a source panel that has a target line. See "Source Panel Target Line" on page 8-54.

List Function/Unit…

Mnemonic: U

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 "Source Panel" on page 8-53.

This dialog box is titled Select a Function/Unit. The title bar also displays the process's qualifier specifier. See "Qualifier Specifiers" on page 6-16. 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 6-18. 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 6-71).

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 "Source Selection Dialog Box" on page 8-27.

List Source File…

Mnemonic: S

Selecting this menu item pops up a dialog box that allows you to list a source file in the source panels. See "Source Panel" on page 8-53.

This dialog box is titled Select a Source File. The title bar also displays the process's qualifier specifier. See "Qualifier Specifiers" on page 6-16. 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 6-19. 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 "Source Selection Dialog Box" on page 8-27.

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 source panels displaying the current process. See "Source Panel" on page 8-53.

This dialog box is titled Select a File.

List Location…

Selecting this menu item pops up a dialog box in which you can enter arguments for a **list** command, to apply to any source panels displaying the current process. See "List Location Dialog Box" on page 8-52 and "list" on page 6-76.

Edit

Mnemonic: I

Selecting this item lets you edit the current process's current source. See "Source Panel" on page 8-53.

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.

Show Source

Mnemonic: O

Accelerator: Ctrl+O

Selecting this menu item causes the source panel to list source, if possible. If the debugger tries to show a position, such as a library routine, that does not have a corresponding source file, then the source panel shows disassembly instead.

When switching between display modes, NightView uses the position of the text cursor to determine the line or address to show in the new mode.

This item is enabled only if there is exactly one source panel, or if a source panel has a target line. See "Source Panel Target Line" on page 8-54. Each source panel has its

own display mode, so, for example, you can show source in one source panel and disassembly in another source panel.

Show Mixed Source and Disassembly

Mnemonic: M

Accelerator: Ctrl+M

In this mode the debugger shows a line of source followed by the instructions that correspond to that line. Source lines that do not produce code are not shown. Only one source line is shown for each group of instructions, so statements that span lines are only partially shown. Note that because of inlining and optimization, not all the instructions that follow a line are generated by that line. Also note that lines from multiple files may be shown in this mode.

See the description of the Show Source menu item for more information.

Show Disassembly

Mnemonic: D

Accelerator: Ctrl+D

Selecting this menu item causes the source panels to list assembly instructions.

The range of instructions displayed usually corresponds to a single subprogram.

See the description of the Show Source menu item for more information.

Eventpoint Menu

Mnemonic: E

This menu provides ways to set eventpoints and to see a summary of eventpoints. See "Eventpoints" on page 3-9.

Before selecting one of the eventpoint menu items, select the line of interest in a source panel. See "Source Panel" on page 8-53. NightView uses this line to initialize the location specifier for you. See "Location Specifiers" on page 6-14.

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 6-96. See "breakpoint" on page 6-99.

For information on using the breakpoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Set Monitorpoint...

Mnemonic: 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 6-96. See "monitorpoint" on page 6-106.

For information on using the monitorpoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

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 6-96. See "patchpoint" on page 6-100.

For information on using the patchpoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Set Tracepoint...

Mnemonic: 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 6-96. See "tracepoint" on page 6-104.

For information on using the tracepoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Set Heappoint...

Mnemonic: H

Selecting this menu item pops up a heappoint dialog box that allows you to set a new heappoint at a given location and apply eventpoint attributes to it. See "Manipulating Eventpoints" on page 6-96. See "heappoint" on page 6-107.

For information on using the heappoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Set Watchpoint...

Mnemonic: 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 6-96. See "watchpoint" on page 6-117.

For more information on using the watchpoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Set Syscallpoint...

Mnemonic: S

Selecting this menu item pops up a syscallpoint dialog box that allows you to set a new syscallpoint and apply eventpoint attributes to it. See "Manipulating Event-points" on page 6-96. See "syscallpoint" on page 6-119.

For more information on using the syscallpoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

New Eventpoint Panel

Mnemonic: E

Selecting this menu item creates a new eventpoint panel. See "Eventpoint Panel" on page 8-61.

Eventpoint Panels Update Interval…

Mnemonic: U

Clicking this menu item brings up a dialog box that lets you set the interval between eventpoint panel automatic updates. See "Eventpoint Panel Update Interval Dialog Box" on page 8-52.

Data Menu

Mnemonic: D

Use this menu to select a data item to place in a data panel or to load or save a data panel layout.

When you add a data item, if there is no data panel, then one is created. If there is one data panel, then the data item goes to that panel. If there is more than one data panel, then NightView pops up a dialog box to ask you which data panel to add the new item to. For Expression…, the dialog box is always popped up and also has a field for entering the expression. See "Data Panel" on page 8-64. The Save Snapshot button lets you save the current contents of the panel. The Save Layout… and Load Layout… buttons let you save the current data panel layout or restore an old layout.

Most data items are added for the current process. See "Current Process" on page 8-3. Shells items, processes items and monitorpoint items are global, not just for one process.

The items in this menu are similar to some of the items in the data panel context menu. See "Data Panel Context Menu" on page 8-74.

The menu items are:

Expression…

Mnemonic: E

Accelerator: Ctrl+E

The dialog box allows you to enter an expression. A data item for that expression is placed in the data panel. See "Data Panel Add Expression" on page 8-81. See "Expression Data Item" on page 8-67.

Local Variables...

Mnemonic: L

A local-variables data item is placed in the data panel. See "Local Variables Data Item" on page 8-68.

Registers...

Mnemonic: R

A registers data item is placed in the data panel. See "Registers Data Item" on page 8-68.

Stack...

Mnemonic: S

A stack data item is placed in the data panel. See "Stack Data Item" on page 8-69.

Threads...

Mnemonic: T

A threads data item is placed in the data panel. See "Threads Data Item" on page 8-70.

Processes...

Mnemonic: C

A processes data item is placed in the data panel. See "Processes Data Item" on page 8-71.

Shells...

Mnemonic: H

A shells data item is placed in the data panel. See "Shells Data Item" on page 8-71.

Heap Information...

Mnemonic: H

A heap information data item is placed in the data pane. See "Heap Information Data Item" on page 8-71.

Heap Errors...

Mnemonic: P

A heap errors data item is placed in the data panel. See "Data Panel Add Heap Errors" on page 8-82. See "Heap Errors Data Item" on page 8-72.

Heap Leaks...

Mnemonic: **K**

A heap leaks data item is placed in the data panel. See "Data Panel Add Heap Leaks" on page 8-82. See "Leak Sets / Still Allocated Sets Data Items" on page 8-72.

Still Allocated Blocks

Mnemonic: **A**

A still allocated blocks data item is placed in the data panel. See "Data Panel Add Still Allocated Blocks" on page 8-82. See "Leak Sets / Still Allocated Sets Data Items" on page 8-72.

Monitorpoint Values

A monitorpoint values data item is placed in the data panel. See "Monitorpoint Values Data Item" on page 8-74.

Save Snapshot...

Mnemonic: **V**

This menu item lets you save the current contents of the data panel to a text file. Clicking on this button brings up a dialog box that lets you specify the name of a file in which to save the data. You can also record a comment in the file to describe the data that is being saved. See "Data Panel Save Snapshot" on page 8-84.

Save Layout...

Mnemonic: **A**

Selecting this menu item pops up a dialog box that lets you save the layout of all the data items for a particular process in all the data panels. The information saved includes the type and format of each data item, and to which data panel the item belongs. See "Data Panel Save Layout" on page 8-84.

Load Layout...

Mnemonic: **O**

Selecting this menu item pops up a dialog box that lets you load a saved layout for one or more processes. Any data panels mentioned in the layout are created if they do not exist. See "Data Panel Load Layout" on page 8-83.

Set Stack Frames...

Clicking on this button pops up a dialog box that lets you set the number of stack frames displayed in a context panel or data panel. See "Data Panel Call Stack Frames" on page 8-82.

Set Pointer as Array Indices...

Clicking on this button pops up a dialog box that lets you set the number of elements to show when displaying a pointer as an array. See "Data Panel Pointer Array Dimension" on page 8-84.

Tools Menu

Mnemonic: T

The Tools menu may be used to invoke other NightStar tools. The tools are invoked on the same display as NightView.

NightView makes a menu entry only for those tools that are installed on your system.

NightProbe Monitor

Mnemonic: P

Opens the NightProbe Data Monitoring application.

NightProbe™ is a real-time graphical tool for monitoring, recording, and altering program data within one or more executing programs without significant intrusion. NightProbe can be used in a development environment as a tool for debugging, or in a production environment to create a "control panel" for program input and output.

See the *NightProbe User's Guide* for more information.

NightSim Scheduler

Mnemonic: S

Opens the NightSim Application Scheduler.

NightSim™ is a tool for scheduling and monitoring real-time applications which require predictable, repetitive process execution. With NightSim, application builders can control and dynamically adjust the periodic execution of multiple coordinated processes, their priorities, and their CPU assignments.

See the *NightSim User's Guide* for more information.

NightTrace Analyzer

Mnemonic: T

Opens the NightTrace Analyzer.

NightTrace™ is a graphical tool for analyzing the dynamic behavior of single and multiprocessor applications. NightTrace can log application data events from simultaneous processes executing on multiple CPUs or even multiple systems and can combine these application events with kernel events to present a synchronized view of the entire system. NightTrace allows users to zoom, search, filter, summarize, and analyze events in a wide variety of ways. In addition, NightTrace allows users to manage user and kernel NightTrace daemons, providing the user with the ability to

start, stop, pause, and resume execution of any of the daemons under its management.

See the *NightTrace User's Guide* for more information.

NightTune Tuner

Mnemonic: U

Opens the NightTune Tuner.

NightTune™ is a graphical tool that can be used to tune an application or to monitor various aspects of the system. NightTune monitors CPU utilization, context switching levels, virtual memory paging activity, disk I/O activity, interrupt activity, and network traffic levels. In addition, NightTune can monitor processes and threads, and can change their scheduling parameters and CPU binding. NightTune can also set CPU shielding and change interrupt CPU affinities.

See the *NightTune User's Guide* for more information.

Help Menu

Mnemonic: H

The Help menu has the following items.

On Context

Mnemonic: C

This item provides help about a particular graphical region of a window. See "Context-Sensitive Help" on page 8-2.

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 6-140.

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 "List of Shortcuts" on page 8-26.

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.

NightView Tutorial

Mnemonic: **R**

This item takes you to the beginning of the GUI tutorial chapter. See Chapter 4 [Tutorials] on page 4-1.

NightView User's Guide

Mnemonic: **U**

The item opens the online version of the *NightView User's Guide* in the help window.

NightStar Tutorial

Mnemonic: **T**

This item opens the online version of the *NightStar Tutorial* in the help window. This tutorial incorporates the use of NightSim, NightProbe, NightView, NightTrace, and NightTune in one complete example.

License Report...

Mnemonic: **L**

Opens a dialog box with information about how many licenses are in use and how to get licenses.

On Version

Mnemonic: **V**

This item pops up an information dialog box that describes which version of Night-View you are running.

Toolbars

Most of the controls in the toolbars apply to the current process. See "Current Process" on page 8-3. Some buttons may be disabled (dimmed) under certain circumstances. For example, when the process is running, the Resume button is disabled.

A toolbar may be moved to different areas around the main window by dragging on the drag handle.

Toolbars may be hidden or shown by clicking on an area of a toolbar that has no controls. This brings up a menu of the toolbars. Choose which toolbars you want shown. Another way to choose which toolbars are shown is to select the Toolbars… entry in the View menu.

Command Toolbar

The command toolbar has a label Command: and a combo box, and is initially near the bottom of the main window. The combo box is used to enter NightView commands. All the command-line interface commands, except for **shell**, can be entered in the command toolbar.

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 current process. See "Current Process" on page 8-3.

The combo box has entries for older commands. To retrieve older commands, press the down arrow key. To see the whole list, click on the downward-pointing triangle.

Process Toolbar

Resume

Clicking on this button is similar to using the **resume** command with no argument. See "resume" on page 6-123.

Stop

Clicking on this button is similar to using the **stop** command, except that the button also interrupts any pending commands for the current process. See "stop" on page 6-130.

Next

Clicking on this button is similar to using the **next** command with no argument. See "next" on page 6-126.

Step

Clicking on this button is similar to using the **step** command with no argument. See "step" on page 6-124.

Finish

Clicking on this button is similar to using the **finish** command. See "finish" on page 6-129.

Run to Here

Run the process until it reaches the target line in the source panel. See "Source Panel Target Line" on page 8-54. See "Source Panel" on page 8-53. 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 target line. 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 6-99. See "enable" on page 6-113. See "resume" on page 6-123.

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 "Message Panel" on page 8-61.

Nexti

Clicking on this button is similar to using the **nexti** command with no argument. See "nexti" on page 6-128.

Stepi

Clicking on this button is similar to using the **stepi** command with no argument. See "stepi" on page 6-127.

Up

Clicking on this button advances one stack frame toward the oldest calling frame. This action is similar to using the **up** command with no argument. See "up" on page 6-137.

Down

Clicking on this button advances one stack frame toward the currently executing (newest) stack frame. This action is similar to using the **down** command with no argument. See "down" on page 6-138.

Kill

Clicking this button causes NightView to terminate the current process. See "Current Process" on page 8-3.

This is similar to using the **kill** command. See "kill" on page 6-40.

Depending on the safety level (see "set-safety" on page 6-65), NightView may display a warning dialog box when you use the Kill menu item.

Rerun

Clicking this button causes NightView to terminate the current process, if there is one, and issue the shell command associated with this process. If there is no current process, then NightView issues the shell command associated with the process that terminated most recently. This action is similar to using the **rerun** command with no argument. See "down" on page 6-138.

Interrupt

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 3-36.

Run Mode Toolbar
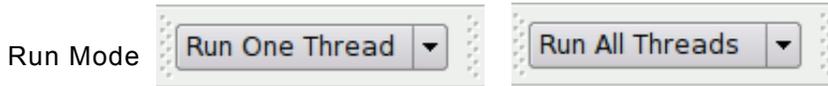
Run Mode

This option list reflects the current run mode and allows you to change it. See "set-run-mode" on page 6-121 for a description of run mode.

Eventpoint Toolbar

Before clicking on a button in the eventpoint toolbar, you may want to click on a target line in a source panel. The Location: field of the dialog box is initialized from the target line, if there is one. See "Source Panel Target Line" on page 8-54.

Set Breakpoint

Clicking on this button 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 6-96. See "breakpoint" on page 6-99.

For information on using the breakpoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Set Patchpoint

Clicking on this button 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 6-96. See "patchpoint" on page 6-100.

For information on using the patchpoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Set Monitorpoint

Clicking on this button 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 6-96. See "monitorpoint" on page 6-106.

For information on using the monitorpoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Set Tracepoint

Clicking on this button 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 6-96. See "tracepoint" on page 6-104.

For information on using the tracepoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Set Heappoint

Clicking on this button pops up a heappoint dialog box that allows you to set a new heappoint at a given location and apply eventpoint attributes to it. See "Manipulating Eventpoints" on page 6-96. See "heappoint" on page 6-107.

For information on using the heappoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Set Watchpoint

Clicking on this button 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 6-96. See "watchpoint" on page 6-117.

For more information on using the watchpoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Set Syscallpoint

Clicking on this button pops up a syscallpoint dialog box that allows you to set a new syscallpoint and apply eventpoint attributes to it. See "Manipulating Eventpoints" on page 6-96. See "syscallpoint" on page 6-119.

For more information on using the syscallpoint dialog box, see "Eventpoint Dialog Boxes" on page 8-29.

Clear

Clicking on this button is similar to using the **clear** command for the target line. See "clear" on page 6-109. You must have set the target line to the source line where you want to clear eventpoints. See "Source Panel Target Line" on page 8-54. 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 message panel. See "Message Panel" on page 8-61.

Value Toolbar

Data Display

Clicking on this button is similar to using the Expression… button in the Data menu. See "Data Menu" on page 8-14. You must have selected an expression in a source panel (or another panel) before pressing this button. See "Source Panel" on page 8-53. When you press the button, the selected expression is added to the default data panel. See "Data Panel" on page 8-64.

Print

> Clicking on this button is similar to using the **print** command. See "print" on page 6-85. You must have selected an expression in a source panel (or another panel) before pressing this button. See "Source Panel" on page 8-53. When you press the button, the value of the selected expression is printed using the default format for the type of the expression.

Source Display Toolbar

> The buttons in this toolbar let you change the source display between source, mixed source and disassembly and just disassembly. These act the same as the corresponding menu items in the Source menu. See "Source Menu" on page 8-10.

Status Bar

> This area shows the name of the executable program that the current process is running, the qualifier specifier for the current process, and the current process's current status. See "Current Process" on page 8-3. See "Qualifier Specifiers" on page 6-16. For threaded programs, the process ID in the qualifier indicates which of the thread processes last stopped. See "Multithreaded Programs" on page 3-40.
>
> A progress bar is shown here when the debugger is doing some lengthy operations, such as processing debugging information, or during commands that wait, such as the run command.
>
> Here are the values that may appear as the process status:

About to exit

> The process called the `_exit(2)` system service. See "Exited and Terminated Processes" on page 3-18.

Calling function

> The process is executing to evaluate a function call.

Exited

> The process has exited. See "Exited and Terminated Processes" on page 3-18. This status does not normally appear, because the process is removed from the 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 6-129.

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 6-124.

Stopped after finish

The process has completed a **finish** command. See "finish" on page 6-129.

Stopped after step

The process has finished a stepping command. See "step" on page 6-124.

Stopped at breakpoint *number*

The process hit breakpoint number *number*. See "Breakpoints" on page 3-11.

Stopped at watchpoint *number*

The process stopped because of watchpoint *number*. See "Watchpoints" on page 3-13.

Stopped for watchpoint error

The process stopped because of an error during watchpoint processing. An error message in the message panel should explain the problem. See "Watchpoints" on page 3-13. See "Message Panel" on page 8-61.

Stopped after unexpected trap

The process stopped due to an int3 instruction at a location that NightView was not expecting. Either the user program has an int3 instruction in it, which would be unusual, or else there is an internal error in NightView.

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 6-130.

Stopped for *exception-name*

The process stopped because of the Ada exception named *exception-name*. See "Exception Handling" on page 3-40.

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 3-15.

Terminated with *signal*

The process terminated with signal *signal*. See "Exited and Terminated Processes" on page 3-18. This status appears only for core files. See "Core Files" on page 3-4.

List of Shortcuts

These shortcut keys work when the focus is in any panel except a shell panel. In a shell panel, control keys are passed to the shell. See "Shell Panel" on page 8-60. There are extra keys you can type into a source panel. See "Source Panel Keystrokes" on page 8-59.

Ctrl+A	New page. See "View Menu" on page 8-5.
Ctrl+B	Breakpoint dialog. See "Eventpoint Menu" on page 8-12.
Ctrl+C	Copy in a text field.
Ctrl+D	Show disassembly. See "Source Menu" on page 8-10.
Ctrl+E	Add expression to data panel. See "Data Menu" on page 8-14.
Ctrl+F	Find. See "Find Bar" on page 8-53.
Ctrl+G	Find again. See "Find Bar" on page 8-53.
Ctrl+M	Show mixed source and disassembly. See "Source Menu" on page 8-10.
Ctrl+O	Show source. See "Source Menu" on page 8-10.
Ctrl+P	Patchpoint dialog. See "Eventpoint Menu" on page 8-12.
Ctrl+Q	Exit the debugger. See "File Menu" on page 8-4.
Ctrl+S	Save config. See "File Menu" on page 8-4.
Ctrl+V	Paste in a text field.
Ctrl+X	Cut in a text field.
Ctrl+Z	Undo in a text field.
Ctrl++	Increase font size. See "View Menu" on page 8-5.
Ctrl+-	Decrease font size. See "View Menu" on page 8-5.
Ctrl+0 (zero)	Restore default font size. See "View Menu" on page 8-5.

Main Window Dialog Boxes

This section describes the dialog boxes you might use while debugging, except for the dialog boxes related to the data panel. See "Data Panel Dialog Boxes" on page 8-81.

Run Program in Shell Dialog Box

This dialog box pops up when you click Run… in the Process menu. See "Process Menu" on page 8-8. This dialog box lets you enter a program to debug. You may enter the program name yourself or click Browse… to find the program in a file browser. Then enter any arguments to the program and click OK. The command is sent to the dialogue shell. See "Dialogue I/O" on page 3-5.

Attach Dialog Box

This dialog box pops up when you select the Attach… item in the Process menu. See "Process Menu" on page 8-8. This dialog box lets you view the processes on the system and select one or more to attach to.

Check the box to indicate whether you want the processes to continue to run after the attach. The initial value for this is from the global mode, set with the **set-resume** command or the preferences dialog box. See "set-resume" on page 6-72 and "Preferences Dialog Box" on page 8-39.

Select which processes you want to see with the filter fields, then click on Refresh. Select the processes you want to attach to by clicking on them, then click on Attach.

The filter has three text input areas. The text input areas each take a regular expression. When you click on the Refresh button, the Processes list is filled in with the processes that match all three regular expressions: one for the process identifier (PID), one for the User and one for the Program name. See "Regular Expressions" on page 6-18. Each regular expression must match the entire corresponding string (that is, each one uses an anchored match). The initial value of the PID and Program regular expressions is ".*", which match all processes. The initial value of the User regular expression is the name of the user logged in to the dialogue.

The Processes list indents the program names to show the parent/child relationship. Each process appears below its parent process and indented relative to the parent process.

The Attach button closes the dialog box and attaches to the selected processes.

Click on Cancel to dismiss the dialog box without attaching to any processes. Click on Help to get help for this dialog box.

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 Source menu. See "Source Menu" on page 8-10. It allows you to change the program code that is listed in the source panel 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 current 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 6-18. See "Wildcard Patterns" on page 6-19.

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. If no functions or files are found, a message is displayed in the message panel. See "Message Panel" on page 8-61.

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 "Source Panel" on page 8-53. 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.

File Selection Dialog Box

This dialog box pops up when you select List Any File… from the Source menu. It allows you to list a file of your choice in a source panel.

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.

Eventpoint Dialog Boxes

NightView provides a dialog box for each type of eventpoint. See "Eventpoints" on page 3-9. These dialog boxes pop up when you use the Eventpoint menu, the source panel context menu, or the eventpoint panel to set or change an eventpoint. See "Eventpoint Menu" on page 8-12.

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. Similarly, the syscallpoint dialog first presents widgets allowing you to specify the system calls of interest, 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 6-14. 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 eventpoint 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 target line in the source panel. See "Source Panel Target Line" on page 8-54. You can edit this text input area.

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. If the target is an IA-32 or AMD64, watchpoints always trap on memory writes, but you can control whether they also trap on memory reads.

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.

When changing an existing watchpoint, this field cannot be changed.

Watchpoint size (watchpoint dialog box only)

This combo box lets you select the size, in bytes, of the watchpoint target if you have selected Watch address and size in the watchpoint options area. 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.

Syscallpoint selection (syscallpoint dialog box only)

This text input area lets you enter one or more system call names, separated by commas or spaces. Leaving the list empty, or the word All, refers to all system calls.

To the right of the text input area is a Select… button; pressing it launches a dialog which aids you in selecting system call names. See "System Call Selection Dialog" on page 8-34 for information on this dialog.

Below the text input area is a checkbox, which, when checked, negates the specified list of system calls. Thus you can easily use the selection area to match all system calls but a select few. This can be especially useful in situations where you already have a syscallpoint that matches one system call, but want to take different actions for all other system calls with a new syscallpoint event.

When changing an existing syscallpoint, these fields cannot be changed.

Syscallpoint actions and modifiers (syscallpoint dialog box only)

The action to be taken when a syscallpoint is hit is specified by selection of one of the two radio buttons:

- Print only; do not stop

- Stop and print

Further, the modifiers to the right allow you to match a system call on entry, exit, or both.

When changing an existing syscallpoint, these fields cannot be changed.

Eventpoint Number (display only)

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 6-113. See "disable" on page 6-113.

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.

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 6-115.

For patchpoints, this is similar to using the `tpatch` command, or the `enable/once` command. See "tbreak" on page 6-115.

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 6-111.

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 6-114.

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 6-98.

You cannot change an existing eventpoint's name using the dialog box. Use the `name` command to change eventpoint names.

Commands

Valid for breakpoints, syscallpoints, and watchpoints only. 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 6-110.

There are additional restrictions for commands when used with watchpoints and syscallpoints; see "watchpoint" on page 6-117 and "syscallpoint" on page 6-119 for more information.

Monitorpoint Expressions

This area is shown for monitorpoints only. One or more expressions are *required* to set a monitorpoint. Enter an expression in the first column.

The default format to print the is determined by the type of the expression. If you want a different format, click on the format field. The field changes to a combo box. Select the desired format.

Enter a label if desired. If no label is entered, then the expression string is used as the label.

Click on New to make another entry in the expression table. You can also use the Tab key in the label field to make a new entry.

Patchpoint Action

Valid for patchpoints only; you are *required* to enter either an expression, a location specifier, or a _Thread Tags_ assignment to set a patchpoint. Select the appropriate choice by clicking on it. The radio button appears filled for your selection, and the label for the text input area changes to either Evaluate, Go to, or Tag. 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 **patch-point** command. See "patchpoint" on page 6-100. This is the default choice.

Branch to a different location

This field represents the *goto* argument of one variant of the **patch-point** command.

Set thread local tag values

This field must contain the tag assignment(s) to be executed. See the *tag assignment* description of the **patchpoint** command for more information.

NightTrace Event (for tracepoints only)

ID

You are *required* to supply an ID number 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 6-104.

Value

This optional field represents the value= argument of the **trace-point** command. You can enter an expression whose value should be logged with the trace event.

Once set, these fields cannot be changed.

Heappoint Action -- Check - Debug Settings

This area is shown for heappoints only. Select whether you want to perform a heap check or change the heap debugger settings by clicking on one of the radio buttons. If you want to change the heap debugger settings, enter the arguments in the Settings: text input area. Valid arguments are the same as for the **heapdebug** command. See "heapdebug" on page 6-54.

Dialog Buttons

Click on OK to set or change the eventpoint. The dialog box is dismissed.

Click on Delete to delete this eventpoint. The dialog box is dismissed. This button is present only for an existing 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 and the eventpoint panel change. NightView displays a message in the message panel 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. Other warnings or errors associated with setting or changing this eventpoint are displayed in the message panel. See "Message Panel" on page 8-61.

You can use the `info eventpoint` command or the eventpoint panel to check the eventpoint settings. See "info eventpoint" on page 6-146. See "Eventpoint Panel" on page 8-61.

System Call Selection Dialog

This dialog aids in selecting one or more system calls for the Syscallpoint Dialog. You can use the search facility to quickly locate a specific call, or you can scroll and select individual items. This dialog supports multiple selection; use Ctrl+<click> to select multiple non-contiguous system calls, or Shift+Ctrl+<click> to select a range of system calls.

When focus is in the list of displayed system calls, keystrokes entered take you to the next system call that starts with the letter associated with your keystroke.

Debug Heap Dialog Box

This dialog box pops up when you use the Debug Heap... item in the Process menu (see "Process Menu" on page 8-8). This dialog box allows you to turn on heap debugging and control the heap debugger. The effect of using this dialog box is similar to using the `heapdebug` command (see "heapdebug" on page 6-54).

Enable Heap Debugging

The most important control is the checkbox to turn heap debugging on and off. If you are a casual user of heap debugging, you may want to restrict your attention to this control and the level buttons and common error buttons. See "heapdebug" on page 6-54 for information about turning heap debugging on and off.

Debugging Level buttons

The level buttons provide a convenient way of setting some of the other controls. See "Levels and Common Errors" on page 3-31.

Common Errors Detection buttons

The common error buttons provide a convenient way of setting some of the other controls to configure the heap debugger to detect a particular kind of program error. See "Levels and Common Errors" on page 3-31.

General Settings

If Hardware Overrun Protection is checked, each block is placed at the end of a page and the following page is protected from reads and writes. See "Hardware Overrun Protection" on page 3-32.

Check Specify check heap freq to enable automatic heap checking. In the text field, set the number of heap operations between heap checks. Uncheck the box to turn off automatic heap checks.

Check Specify retained free blocks to give the number of free blocks that should be retained. Uncheck the box retain all free blocks. See "Retained Free Blocks" on page 3-33.

Check Specify heap size to limit the size of the heap to the specified number of bytes. Uncheck the box to indicate that the total size of the heap is limited only by system resources.

Enter the number of extra bytes to add to each allocation size in Slop size.

Enter the number of walkback entries to keep for each heap operation in Walkback Entries Per Block. This number refers to physical walkback entries. The number of walkback frames may differ from this number when displayed in NightView. The number of frames displayed may include extra inline frames, as they are not physical frames. The number of frames displayed may be fewer if certain frames are deemed uninteresting (see "interest" on page 6-67).

Pre-fence size is the number of bytes to fill and check before each block. Post-fence size is the number of bytes to fill and check after each block. See "Fences" on page 3-32.

NOTE

When specifying a number of heap operations, blocks, or bytes, you may append the letter k to multiply the number by 1024, or the letter m to multiply by 1048576.

Fill Settings

If Fill malloc space is checked, blocks are filled with the Malloc fill byte when they are allocated.

If Fill free space is checked, blocks are filled with the Free fill byte when they are freed.

If Check free fill is checked, retained free blocks are checked for the Free fill byte during a heap check.

Pre-fence fill byte is the value used to fill the pre-fence. Post-fence fill byte is the value used to fill the post-fence.

Error Control

These controls specify how the debugger responds when an error condition is detected. Each error has a Stop checkbox and a Print checkbox. If Stop is checked, the process will stop when it gets that error. If Print is checked, the debugger prints a message when the process gets that error.

Action buttons

Click the OK button to configure heap debugging with these settings and dismiss the dialog box. Click the Reset button to restore the settings to be the same as when the dialog box popped up. Click the Cancel button to dismiss the dialog box

without making any changes to the heap debugging configuration. Click the Help button to get help about the dialog box.

Remote Login Dialog Box

This dialog box pops up when you use the Shell menu's Start Remote Shell… item. See "Shell Menu" on page 8-8. 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 Login General Page

Target

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.

After you click on Login, you will be prompted for the passphrase for this user in another dialog box. If you leave the passphrase empty, then you will be prompted for the password for this user. For security, the passphrase or password you type is not echoed in the window; instead, an asterisk (*) replaces each character.

Shell Name

This field specifies the name to give to the dialogue. See "Qualifier Specifiers" on page 6-16. If you leave this field empty, the name of the dialogue will default to be the same as the Target field. If the remote system name is not a valid dialogue name, an error dialog will appear. 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.

Indicate with the checkbox whether you want a new shell panel to be created for this shell. See "Shell Panel" on page 8-60.

Remote Login Advanced Page

This area allows you to set scheduling attributes which will be applied to the NightView processes which will run on the specified target system.

Scheduling Class

This combo box allows you to select the POSIX scheduling class for the NightView processes:

- Other

   This class corresponds to the SCHED_OTHER scheduling policy which provides for general process scheduling with urgency less favorable than the other two classes. Processes in this class have their priority adjusted by the operating system based on CPU usage.

- Round Robin

   This class corresponds to the SCHED_RR scheduling policy which provides real-time process scheduling using a time-slicing algorithm to share CPU resources with other SCHED_RR processes of the same priority.

- First In First Out

   This class corresponds to the SCHED_FIFO scheduling policy which provides the strictest real-time process scheduling. Processes are not time-sliced with other SCHED_RR or SCHED_FIFO processes of the same priority.

See **sched_setscheduler(2)** for more information on these scheduling classes.

Class Attributes

Depending on the scheduling class chosen, the following attributes can be selected:

- Nice Value

   You can use this spin-box to set the initial nice value to be associated with the NightView processes. A nice value provides a bias to the default priority of a process, thereby affecting the effective priority. Positive values correspond to less favorable scheduling urgency. This attribute is only available when the Other class is selected. See **nice(1)** for more information on the effect of nice values.

- Real-time Priority

   You can use this spin-box to set the real-time priority of the NightView processes. Priority values are constrained to be between 1..99. Selecting a priority value exceeding 90 is not recommended as it may interfere with kernel daemon processing. A priority of one is sufficient to give the process more urgency than any process in the SCHED_OTHER class. Higher priority numbers correspond to more favorable scheduling urgency.

- Time Quantum

You can use this spin-box to set the duration of the time-slice for processes using the SCHED_RR class. This attribute is not applicable to any other scheduling class.

CPU List

NightView server process execution will be constrained to the CPUs listed here. Additionally, the remote shell being created and all programs you run in it will be similarly constricted, unless you specifically change their interrupt affinity programmatically or with the **run(1)** command.

By default, the list is `all`, which means the server process can run on any CPU on the target system which isn't shielded from process execution (consult the **shield(1)** man page for more information on shielding).

The list can either be the word `all`, or a comma-separated list of CPU numbers or ranges of CPU numbers; for example: `0,2-3.`

To the right of the text field a description of the resultant CPU mask is shown. Some system interfaces require CPU affinity to be specified as a mask, with each bit in the mask representing a CPU. The mask is shown to remind you that the numbers you enter into the text field here are logical CPU <u>numbers</u>, not hexadecimal characters in a CPU mask.

If you enter something invalid into the text field, the description to the right changes to the word `invalid`, shown in red. Ultimately, syntactically-invalid CPU lists are automatically replaced with a list indicating `all`.

NOTE

Selection of the Round Robin or First In First Out scheduling class or a non-default CPU bias requires privileged access.

Remote Login Action Buttons

Login

When you press the Login button, the remote dialogue is created and the remote login dialog box 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 panel. See "Message Panel" on page 8-61.

Reset

Set all the fields back to their default values.

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 box.

Preferences Dialog Box

This dialog box pops up when you click on Preferences… in the File menu. See "File Menu" on page 8-4. With it you can adjust things such as fonts and how you interact with the debugger. Preferences you set here apply only to this debug session unless you save them to disk. You can save your preferences to disk by clicking the Save button in this dialog box or with the Save Preferences item in the File menu.

Many of these preferences may be set by commands. This dialog box and the commands refer to the same preferences, so, for example, if you change a preference with a command, you can see the same value when you bring up this dialog box.

For a way to change settings that relate to a process, see "Process Settings Dialog Box" on page 8-50.

Click OK to apply the preferences and dismiss the dialog.

Click Apply to apply the preferences and leave the dialog up.

Click Reset to set the preferences back the way they were when the dialog popped up (or when Apply the button was clicked).

Click Save to apply the preferences, save them to disk and dismiss the dialog.

Click Cancel to dismiss the dialog without applying any changes.

Click Help to get help about the dialog.

Preferences General Page
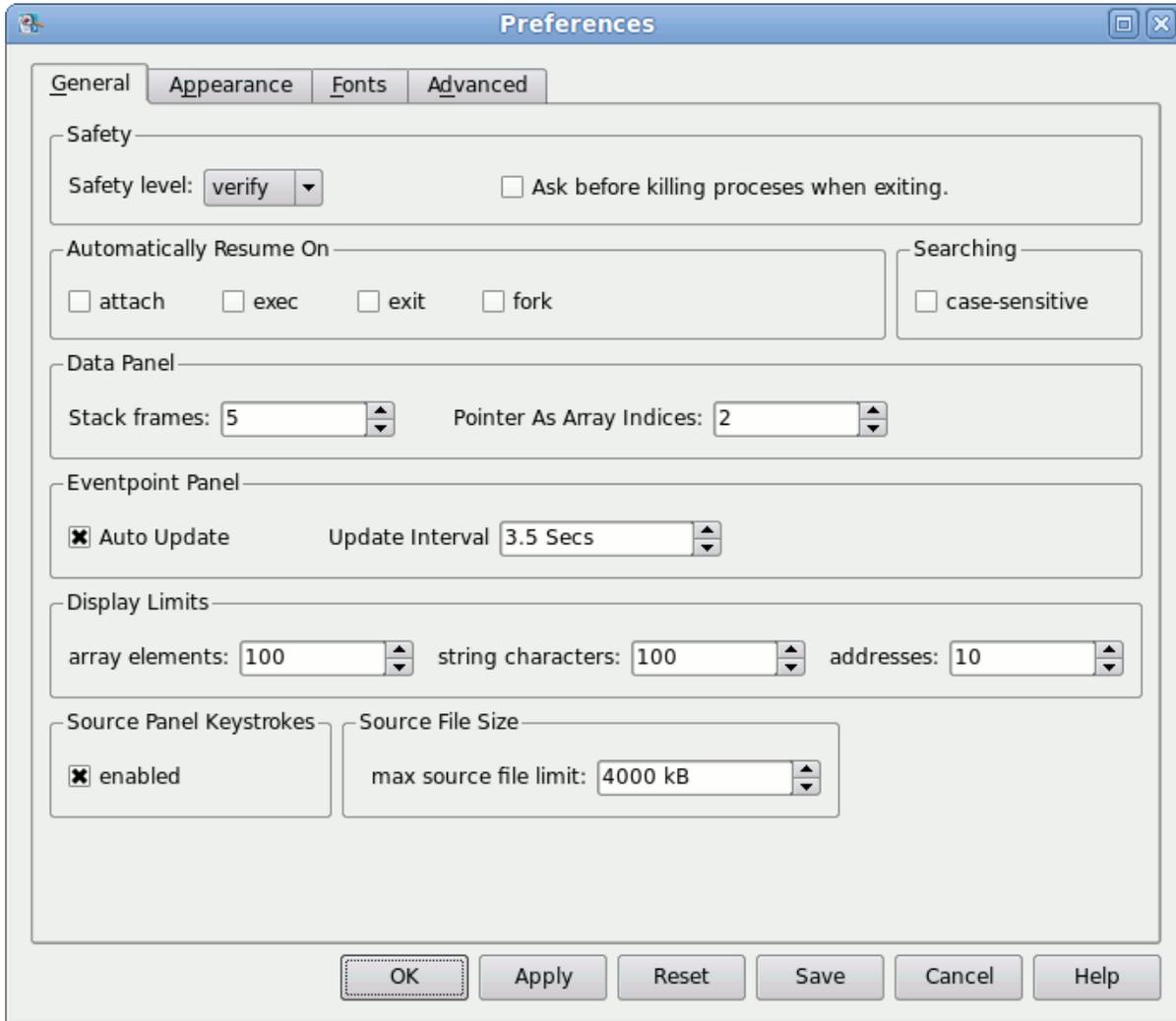
Safety

> Select a safety level. This is similar to using **set-safety**. See "set-safety" on page 6-65. If the level is verify, you can say whether you want to be warned if you ask to exit when you are still debugging processes.

Automatically Resume On

> The debugger normally stops a process when it is first attached, when it execs, when it is about to exit, or when it forks. Check the conditions for which you want the process to continue running. See "set-resume" on page 6-72.

Searching

> Indicate whether you want searching to be case-sensitive. This affects the forward-search and reverse-search commands, and is the initial setting for find bars. See "set-search" on page 6-71.

Data Panel

> Select the default number of stack frames to display in a data panel (assuming at least that many stack frames exist). Select the number of indices to show when treating a pointer as an array. See "Data Panel Call Stack Frames" on page 8-82 and "Data Panel Pointer Array Dimension" on page 8-84.

Display Limits

> Select the display limits for array elements, string characters and addresses in location specifier listings. These limits are to prevent large amounts of output from overwhelming the display. See "set-limits" on page 6-62.

Source Panel Keystrokes

> You can perform many actions by pressing keys when the keyboard focus is in the source panel. Use this control to turn source panel keystrokes on or off. See "Source Panel Keystrokes" on page 8-59.

Source File Size

> Select the maximum number of bytes a source file can have to be displayed in a source panel. This is useful for extremely large source files which overwhelm NightView due to the overhead involved in building individual widgets associated with each line. The number is in units of 1000 bytes. The default value is 4000, which indicates ~4MB. The source for files that exceed the limit are not displayed, but the assembly associated with the function associated with the current stack frame within the file is displayed. See "set-limits" on page 6-62.

Preferences Appearance Page
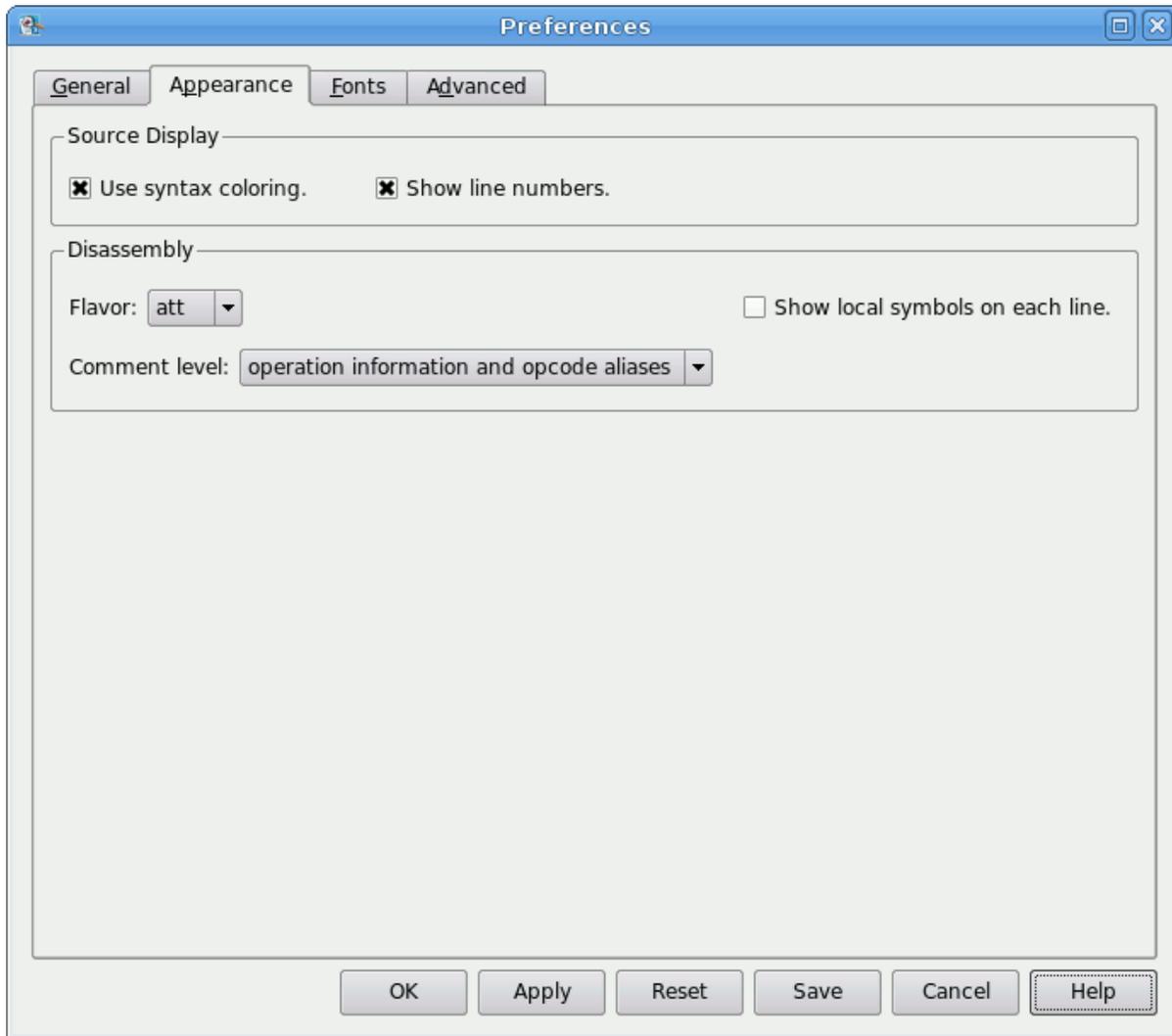
Source Display

Choose whether you want the source to be colored based on syntax and whether you want to see line numbers in the source panel. See "Source Panel" on page 8-53.

Disassembly

Choose how you want to see disassembled code. See "set-disassembly" on page 6-73.

Preferences Fonts Page

NightView uses multiple fonts to present text in the most effective manner throughout the various display areas of the tool.

Variable-width fonts are most commonly used; these fonts most closely resemble how people write or print words.

Fixed-width fonts require that all characters and numbers have the same width (visual footprint). Fixed-width fonts are of benefit when source code is being displayed or manipulated or when columns of numbers are viewed.

NightView further divides the use of fonts into the following categories; default and panel.

Default fonts are used for text associated with operational description and control, including: menus, buttons, selection devices, labels, tool tips, status bar messages, and generally descriptive verbiage.

Panel fonts are used in NightView panels, which display the data of highest importance.

Fonts are selected by querying font preferences from the following sources until a preference is found:

- Your NightView preference

- Your NightStar-wide preference

- The system's NightView preference

- The system's NightStar-wide preference

- NightView's ultimate default
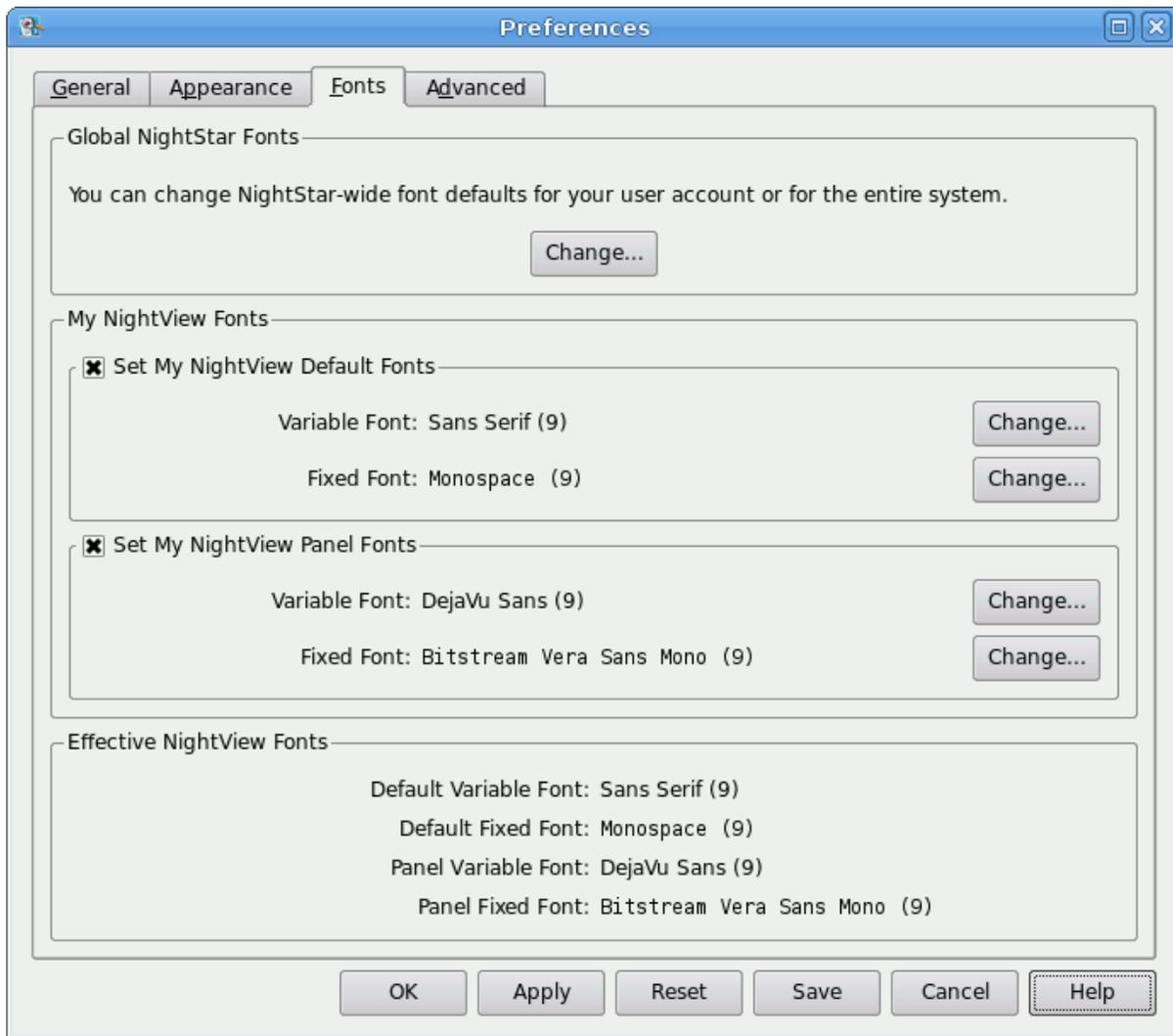
Figure 8-1. Font Preferences Page

This page is divided into three sections.

Global NightStar Fonts

The Change… button in this area launches the NightStar Global Fonts dialog which allows you to set your Nightstar-wide preferences, your preferences for another specific NightStar tool, or the system's tool or NightStar-wide preferences.

Note:

Setting a NightStar preference for the system typically requires `root` access.

Changes saved in the NightStar Global Fonts dialog are always saved to disk and apply to the current and subsequent NightView invocations.

See "NightStar Global Fonts Dialog" on page 8-46 for more information.

My NightView Fonts

This area allows you to set or clear your user's preferences for NightView.

Selection of the checkboxes for the individual font categories control whether or not your preferences are to be consulted. Clearing a checkbox effectively removes your user preference for that category. Setting a checkbox allows you to select specific fonts within the category.

Changes to any of the settings in this area, including individual fonts or category checkboxes, are immediately reflected in the Effective NightView Fonts area at the bottom of the page so you can see the ultimate effect a change will have.

To change a specific font, ensure that the corresponding category's checkbox is checked and then press the Change… button. This will launch a standard font selection dialog. When you select a font from the dialog and press OK, the name of the font family is displayed to the left of the Change… button and is displayed in the selected font as well.

Effective NightView Fonts

This area shows you the effective fonts that will be used based on your user settings and consultation of global settings which aren't shown in the page.

The values in this area immediately change to reflect the effective font whenever any change is made within the page.

Your changes in the My NightView Fonts area are applied to the current invocation of NightView when you press the OK button. However, your changes are not saved to disk and will not affect subsequent invocations of NightView unless you press the Save button.

Separation of apply and Save operations make it easy to experiment with fonts in the current invocation without affecting long-term usage.

Note:

Changes to font preferences in the NightStar Global Fonts dialog are always saved to disk and apply to the current and subsequent NightView invocations; i.e. there is no way to experiment with a global font preference without affecting subsequent NightView invocations.

NightStar Global Fonts Dialog

The NightStar Global Fonts dialog allows you to set your Nightstar-wide preferences, your preferences for another specific NightStar tool, or the system's tool or NightStar-wide preferences.
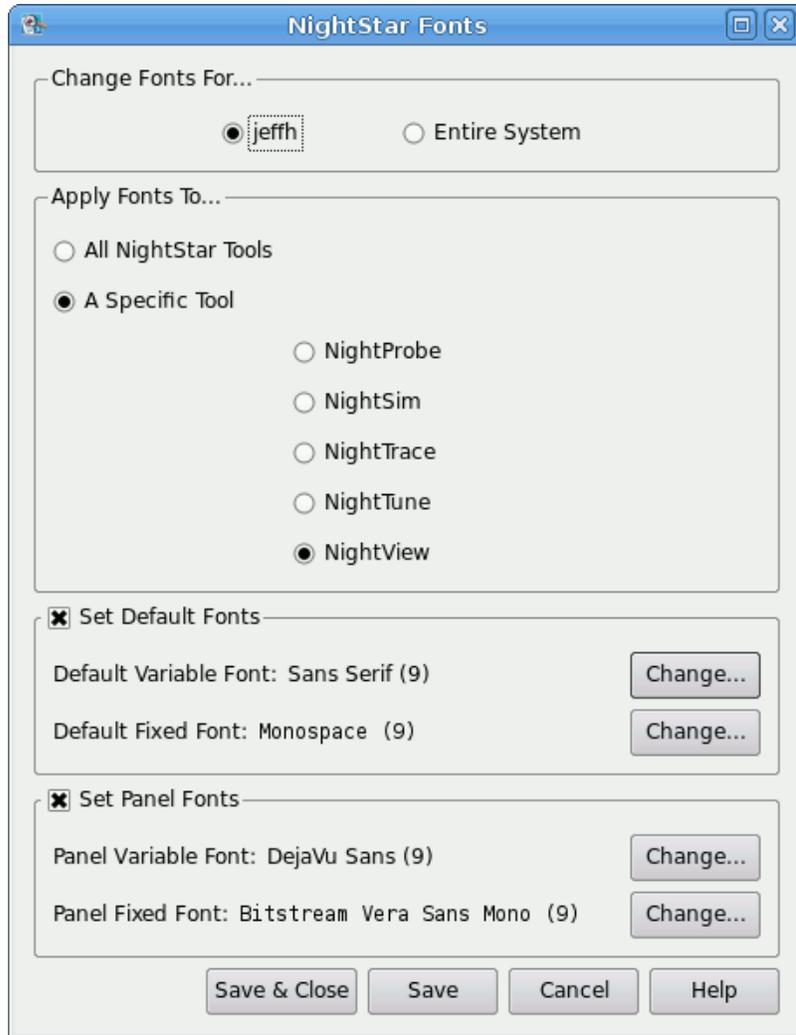
Figure 8-2. NightStar Global Fonts Dialog

Keep in mind that fonts are selected by querying font preferences from the following sources until a preference is found:

- Your NightView preference

- Your NightStar-wide preference

- The system's NightView preference

- The system's NightStar-wide preference

- NightView's ultimate default

This dialog has two control areas which define the scope of font preference application.

Changes Fonts For...

By default, the dialog is set up to apply font preferences to your user account. Select the `Entire System` button if you wish to set the system's preferences.

Note:

Changing font preference for the system typically requires `root` access.

Apply Fonts To...

This area additionally controls the scope of font preference application. You can change a preference for a specific NightStar tool or change the NightStar-wide preference.

If you wish to change the font for more than one tool from this dialog, but not change the NightStar-wide preference, select the first tool of interest, make your preference change in the areas below, and then press the `Save` button. Then select the second tool of interest and repeat.

Set Default Fonts
Set Panel Fonts

These areas contain the variable and fixed-width font preferences for each of the font categories, identified by the label next to each checkbox.

To remove the preferences in a category, clear its checkbox.

To change a specific font, ensure that the category's checkbox is checked and then press the `Change…` button. This will launch a standard font selection dialog. When you select a font from the dialog and press `OK`, the name of the font family is displayed to the left of the `Change…` button and is displayed in the selected font as well.

The buttons at the bottom of the page control the application of your changes.

Save & Close

Saves any changes made in this dialog to disk, thus affecting subsequent tool invocations, and closes the dialog.

These changes may affect the effective font preferences for the current invocation of NightView. When the dialog is closed, the fonts shown in the `Effective NightView Fonts` section of the `Preferences` dialog are updated. If you apply the changes in that dialog, they will take effect in the current invocation of NightView.

Save

Applies the preferences from the dialog to the current invocation of NightView, saves the preferences to disk thereby affecting subsequent NightView invocations.

These changes may affect the effective font preferences for the current invocation of NightView. When this dialog is subsequently closed, the fonts shown in the `Effective`

NightView Fonts section of the Preferences dialog are updated. If you apply the changes in that dialog, they will take effect in the current invocation of NightView.

Cancel

Cancels any unsaved changes and closes the dialog.

Help

Opens the help system to display this section.

Preferences Advanced Page
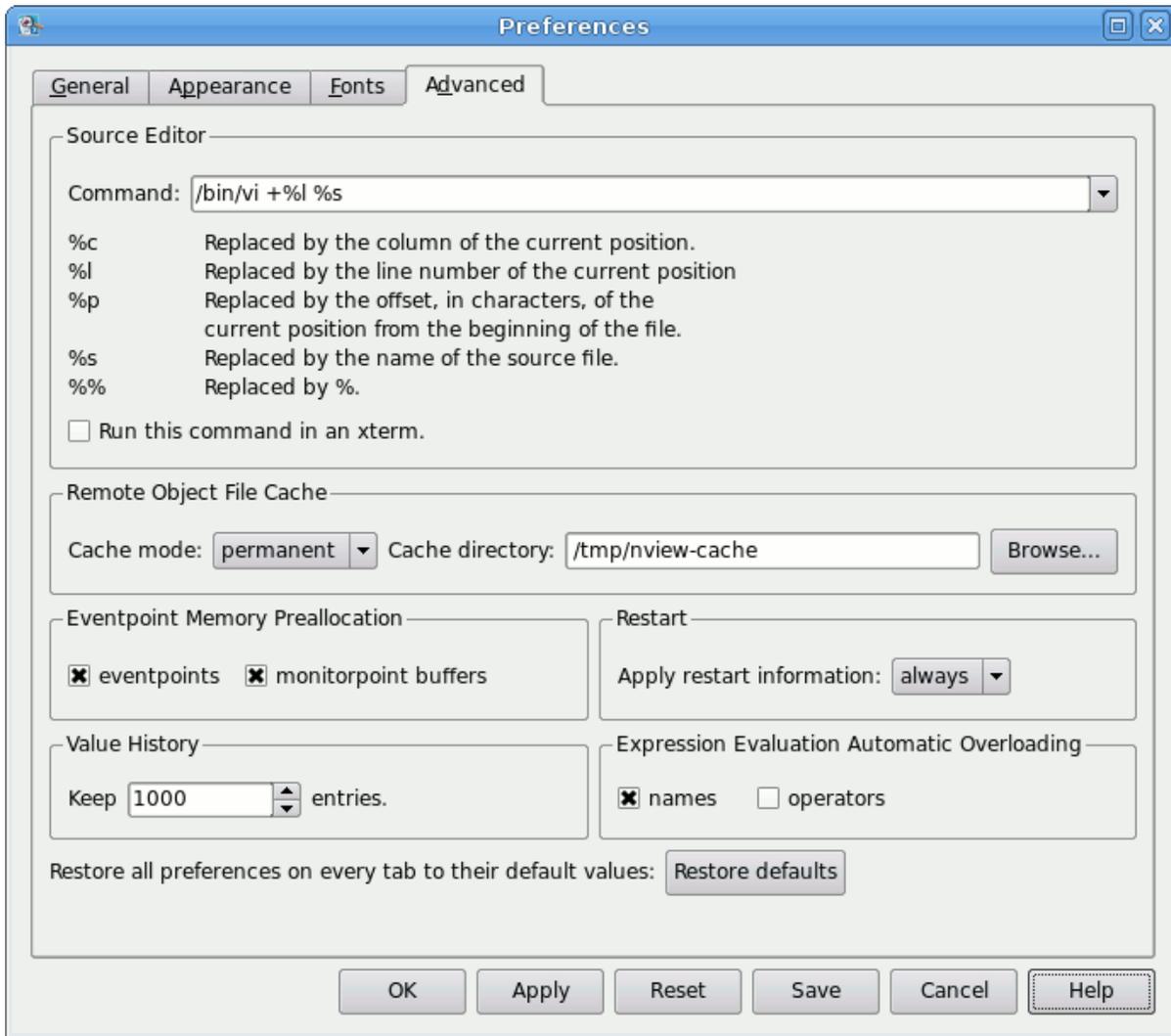
Source Editor

This section determines how to run the editor when you select the Edit… item in the Source menu or use the e key in the source panel. Enter the command you want to use to edit files. The combo box is already loaded with two popular editor commands, and one is

selected based on your `EDITOR` environment variable. You can pass information about the file and the current position with `%` specifiers.

`%`

> 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.

If your editor can communicate with the X Window System display directly, then you should clear the check box for `Run this command in an xterm`. The editor runs on the same display as NightView.

Remote Object File Cache

> Control how NightView downloads files from remote targets. See "set-download" on page 6-73.

Eventpoint Memory Preallocation

> Control how NightView preallocates memory for eventpoints and monitorpoint buffers. See "set-preallocate" on page 6-71.

Restart

> Control whether restart information is applied. See "set-restart" on page 6-65.

Value History

> Specify the number of items to be kept in the value history list. See "set-history" on page 6-62.

Expression Evaluation Automatic Overloading

> Control how NightView treats overloaded operators and routines in expressions. See "set-overload" on page 6-70.

Restore Defaults

> Click this button to give all the preferences their default values. The preferences on all the tabs of the preferences dialog are affected.

Process Settings Dialog Box

> This dialog box pops up when you click on Process Settings… in the Process menu. See "Process Menu" on page 8-8. With it you can adjust things such as whether children are debugged and how signals are handled.
>
> Many of these settings may be set by commands. This dialog box and the commands refer to the same settings, so, for example, if you change a setting with a command, you can see the same value when you bring up this dialog box.
>
> For a way to change user preferences that do not relate to individual processes, see "Preferences Dialog Box" on page 8-39.
>
> Click OK to apply the settings and dismiss the dialog.
>
> Click Apply to apply the settings and leave the dialog up.
>
> Click Reset to put the settings back the way they were when the dialog popped up (or when Apply the button was clicked).
>
> Click Cancel to dismiss the dialog without applying any changes.
>
> Click Help to get help about the dialog.

Process Settings General Page

Debug Children

> Control whether children of this process should be debugged. See "set-children" on page 6-51.

Set Run Mode

> Controls the execution of threads in a multi-threaded process when resuming a thread -- does one thread run or do all threads run. See "set-run-mode" on page 6-121 for a full description.

Stop Before Exiting

> Control whether this process stops before exiting. See "set-exit" on page 6-52.

Expression Language

> Establish a default language context for variables and expressions. See "set-language" on page 6-60.

Refresh debug info when shared libs change

> When this option is checked, NightView automatically detects when shared libraries are loaded (e.g. use of the `dlopen(2)` service) after the process starts. There is some overhead in using this option. Alternatively, you can use the `Refresh Shared Libs` option from the `Process` menu manually when you wish NightView to re-read the shared library list.

Program

> Type the name of the executable program or use the `Browse…` button to find the executable program with a file browser. See "exec-file" on page 6-44.

Process Settings Interest Page

> Control which subprograms are interesting. See "interest" on page 6-67.

Process Settings Signals Page

> Specify how to handle signals in the user process. For each signal, specify whether you want the process to stop when it receives the signal, whether you want NightView to print a message when the process receives the signal, and whether you want the signal to be passed to the process when it resumes. Note that `stop` implies `print`. See "handle" on page 6-133.

Process Settings Ada Exceptions Page

> This tab appears only for an Ada program.

> Select whether you want the process to `stop` and to `print` for exceptions in general. If you want to specify different actions for specific exceptions, use the `Add…` button to add exceptions to the table. The `Add…` button pops up a browser for exceptions. See "Browse Ada Exceptions Dialog Box" on page 8-51. Select whether you want the process to `stop` and to `print` for the individual exceptions in the table.

> You can also select whether you want the process to `stop` and to `print` for exceptions that are not handled by your program.

> In all cases `stop` implies `print`.

> See "handle" on page 6-133.

Browse Ada Exceptions Dialog Box

> Select the exceptions you want to add to the table in the process settings Ada exceptions page and click `OK`.

Rename Page Dialog Box

> This dialog box pops up when you click `Rename Current Page…` in the `View` menu. See "View Menu" on page 8-5.

Type the new name for the page. You can create a mnemonic for the page by preceding one of the characters with an ampersand (`&`). The page's tab will have an underscore under that character. To get a real `&`, use `&&`.

Print Dialog Box

This dialog box pops up when you click on `Print Window…` in the `File` menu. Click `Print` to print the main window.

List Location Dialog Box

This dialog box pops up when you click on `List Location…` in the `Source` menu or in the source panel context menu.

Enter a where-spec as you would for the list command. See "list" on page 6-76. If you clicked on `List Location…` in the `Source` menu, any source panels displaying the current process are affected (see "Source Menu" on page 8-10). If you clicked on `List Location…` in the source panel context menu, the source panel in which you clicked is the only one affected (see "Source Panel Context Menu" on page 8-54).

If desired, you can specify a file on another host with the form *user*`@`*host*`:/`*path*. See "Remote File Access" on page 3-7.

Eventpoint Panel Update Interval Dialog Box

This dialog box pops up when you click on `Eventpoint Panels Refresh Rate…` in the `Eventpoint` menu.

NightView can discover some of the eventpoint information only by querying the process. To reduce overhead, this is done only every few seconds.

Select the number of seconds between updates and whether you want automatic updates. You can use the eventpoint context panel to do updates manually. The eventpoint panel is updated when you make any eventpoint change regardless of whether you have automatic updates on.

The eventpoint panel update interval is not related to the monitorpoint update interval. See "Monitorpoint Update Interval Dialog Box" on page 8-87.

Panels

NightView shows different kinds of information in different panels. The panels are docked within the main window or may be undocked, separate windows. See "Main Window" on page 8-4.

Find Bar

In most panels you can use the find bar to search for text in that panel. The find bar shows up at the bottom of the panel when you use the Find… item in the panel's context menu. See "Context Menu" on page 8-3. You can also make the find bar appear by typing Ctrl+F when the focus is in the panel. (However, typing Ctrl+F in a shell panel does not show the find bar. Instead, the character is sent to the shell.)

The find bar has a button with an X you can use to close the find bar. Next is a text field where you can enter a search string. The search string is a regular expression. See "Regular Expressions" on page 6-18. As you type in more characters, the search progresses. If a matching line is found, the line is highlighted. If no match is found, the background color of the text entry field changes. A label at the end of the find bar shows the status of the search. You can press the Escape key here to close the find bar. You can press Enter to move to the next matching text.

The find bar has a button to find the next matching text and one to search backwards for the previous matching text. There is a check box to indicate whether you want the search to pay attention to case. The initial state of the check box comes from the Searching section of the preferences dialog box. See "Preferences Dialog Box" on page 8-39.

Once you have a search text in a panel, you can use the Find Again item in the panel's context menu, or you can type Ctrl+G when the focus is in the panel, to find the next matching text. (However, typing Ctrl+G in a shell panel does not search. Instead, the character is sent to the shell.)

Source Panel

The source panel lists the program source code or the disassembled instructions corresponding to the current frame in the current process. See "Current Frame" on page 3-26. See "Current Process" on page 8-3. You can select various display modes to display source or disassembly with the Source menu. See "Source Menu" on page 8-10. See "list" on page 6-76, for information on how the current source file is determined.

The text in this area includes the program source or disassembled instructions along with line numbers and source decorations. See "Source Line Decorations" on page 6-82. You can turn off the line numbers with the preferences dialog box. See "File Menu" on page 8-4. If you hover the mouse pointer over the source line decoration, a tooltip shows any eventpoint information for the line.

The text in this area changes if you use the Source menu to list other functions or files, if you use the `list` command (see "list" on page 6-76), and when the process stops or you change the current context.

The titlebar of the source panel shows the name of the program, the qualifier of the process associated with this source panel, and the name of the source file displayed in the panel.

If the source panel is in disassembly display mode, the title bar shows information about the region displayed, such as Function main.

Source Panel Target Line

When you click on a line in a source panel, that line becomes the *target line*. The source panel shows the target line with a different background. (This is independent of text selection, which has a different highlighting background.) At most one line in one source panel can be the target line.

Buttons and keystrokes that deal with eventpoints use the target line to identify the source file and line number to operate on. If the button brings up a dialog box, then the target line is used to initialize the location field. Otherwise, the target line is acted on immediately. See "Source Panel Keystrokes" on page 8-59. See "Eventpoint Menu" on page 8-12. See "Source Panel Context Menu" on page 8-54.

If there is more than one source panel, the target line also indicates which source panel to operate on in some items in the Source menu. See "Source Menu" on page 8-10.

In addition to setting the target line, clicking in a source panel also sets the current process to be the process currently associated with the source panel. See "Current Process" on page 8-3. If the current process changes, the target line is cleared.

Source Panel Expression Tooltip

If you select the text of an expression in the source panel, and then hover the mouse pointer over the selection, NightView displays a tooltip showing the value of the expression. (This is text selection, not the same as clicking to set the source panel target line. See "Source Panel Target Line" on page 8-54.) If there is some error, such as that the selected text is not a valid expression, the tooltip shows the error instead. If the process is stopped, then NightView evaluates the expression with respect to the current frame. See "Current Frame" on page 3-26. If the process is running, the evaluation is done in the global scope.

The surrounding text is ignored, so, for example, if you have a line that includes `a.b.c` and you select and hover over only the `c`, NightView tries to find a variable named `c`, not the member named `c` from `a.b`.

The expression is not evaluated if it would modify the process or cause the process to run, so you will get an error if you select and hover over "`i = 1`" or "`factorial(x)`".

To select text, either drag the mouse pointer over it, or move the text cursor to one end of the expression and then hold down Shift and press the Right or Left arrow key until the whole expression is selected.

Source Panel Context Menu

Right-click in the source panel to bring up the context menu. This menu provides ways of changing the program code displayed in the panel, manipulating eventpoints, and editing source files that are listed. See "Source Panel" on page 8-53.

Note that the right-click also sets the target line. See "Source Panel Target Line" on page 8-54.

The items that change which source file is displayed, and the items that select source or disassembly act on this panel only.

Set simple breakpoint

Mnemonic: B

Sets a simple breakpoint on the target line.

Clear eventpoint

Mnemonic: L

Clears all the eventpoints on the target line.

Run to Here

Mnemonic: H

Runs the process until it reaches the target line (by setting a breakpoint and deleting the breakpoint when it is hit).

Set eventpoint

Mnemonic: E

Brings up a sub-menu of eventpoint types. Click on an entry in the sub-menu to bring up a dialog to set an eventpoint on the target line.

Set Breakpoint...

Mnemonic: B

Accelerator: Ctrl+B

Set Monitorpoint...

Mnemonic: M

Set Patchpoint...

Mnemonic: P

Accelerator: Ctrl+P

Set Tracepoint...

Mnemonic: T

Set Heappoint...

Mnemonic: H

Set Watchpoint...

Mnemonic: W

Set Syscallpoint...

Mnemonic: S

Edit eventpoint

If there is a single eventpoint on the target line, this item appears as Edit eventpoint *n*, where *n* is the eventpoint number. If there are multiple eventpoints on the target line, this item brings up a sub-menu that lists the eventpoints. Selecting any of these items brings up a dialog box allowing you to edit the eventpoint attributes.

List Function/Unit...

Mnemonic: U

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 source panel. See "Source Panel" on page 8-53.

This dialog box is titled Select a Function/Unit. The title bar also displays the process's qualifier specifier. See "Qualifier Specifiers" on page 6-16. 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 6-18. 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 source panel. 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 6-71).

The Select a Function/Unit dialog box is one variation of the source selection dialog box, which is also used by the List Source File... menu item. See "Source Selection Dialog Box" on page 8-27.

List Source File...

Mnemonic: S

Selecting this menu item pops up a dialog box that allows you to list a source file in the source panels. See "Source Panel" on page 8-53.

This dialog box is titled Select a Source File. The title bar also displays the process's qualifier specifier. See "Qualifier Specifiers" on page 6-16. 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 6-19. 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 source panel.

The Select a Source File dialog box is one variation of the source selection dialog box, which is also used by the List Function/Unit... menu item. See "Source Selection Dialog Box" on page 8-27.

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 source panel. See "Source Panel" on page 8-53.

This dialog box is titled Select a File.

List Location...

Brings up a dialog box in which you can enter arguments for a **list** command, to apply only to this source panel. See "List Location Dialog Box" on page 8-52 and "list" on page 6-76.

List History

Mnemonic: R

Selecting this menu item brings up a sub-menu with entries for source files you have viewed recently. The most recently viewed files are listed first. (If you were viewing disassembly, then the entry describes the address region you were viewing.) Click on an entry in the sub-menu to view that file again.

If you view different parts of the same file, separate entries are kept in the list so you can easily switch back and forth.

A separate list is kept for each dialogue and program name. If you debug the same program again, the list is still available.

Find...

Mnemonic: F

Accelerator: Ctrl+F

Brings up the find bar and sets the keyboard focus there. See "Find Bar" on page 8-53.

Find again

Mnemonic: G

Accelerator: Ctrl+G

Finds the search string in the find bar again. The search begins from the text cursor position. At the end of each search the text cursor is left at the end of the found string.

Edit

Mnemonic: I

Selecting this item lets you edit the source file that is currently displayed in the source panel. See "Source Panel" on page 8-53. This item is disabled (dimmed) if the source panel is displaying disassembly.

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 source panel or from entering commands based on the new contents.

Show Source

Mnemonic: O

Accelerator: Ctrl+O

Selecting this menu item causes the source panel to list source, if possible. If the debugger tries to show a position, such as a library routine, that does not have a corresponding source file, then the source panel shows disassembly instead.

When switching between display modes, NightView uses the position of the text cursor to determine the line or address to show in the new mode.

Each source panel has its own display mode, so, for example, you can show source in one source panel and disassembly in another source panel.

Show Mixed Source and Disassembly

Mnemonic: M

Accelerator: Ctrl+M

In this mode the debugger shows a line of source followed by the instructions that correspond to that line. Source lines that do not produce code are not shown. Only one source line is shown for each group of instructions, so statements that span lines are only partially shown. Note that because of inlining and optimization, not all the instructions that follow a line are generated by that line. Also note that lines from multiple files may be shown in this mode.

Show Disassembly

Mnemonic: D

Accelerator: Ctrl+D

Selecting this menu item causes the source panel to list assembly instructions.

The range of instructions displayed usually corresponds to a single subprogram.

See the description of the Show Source menu item for more information.

Track any process

Mnemonic: Y

The panel changes whenever another process becomes the current process. See "Source Panel Tracking" on page 8-59.

Track one process

Mnemonic: N

The panel ignores changes for other processes. See "Source Panel Tracking" on page 8-59.

Panel locked

Mnemonic: K

The panel ignores changes for all processes. See "Source Panel Tracking" on page 8-59.

Source Panel Tracking

A source panel usually shows source for the current process and changes when the process stops or you select a different frame. When another process becomes the current process, the source panel also changes to show the new process. We say the source panel tracks any process. See "Current Process" on page 8-3.

You can restrict a source panel to track a particular process. To do this, select the process as the current process in the context panel, right-click in the source panel to get the source panel's context menu, then select Track One Process. See "Context Panel" on page 8-64. See "Source Panel Context Menu" on page 8-54. In this mode the source panel responds only to events for that process, such as clicking on frames for that process in the context panel, or the process stopping.

You can also lock a source panel. Right-click in the source panel to get the context menu, then select Panel Locked. In this mode the panel does not respond to the process stopping or changing frames.

In any mode, the source panel still responds to things you explicitly tell it to do, such as displaying a different file by using the source panel's context menu.

If a source panel is tracking one process or locked, and the associated process terminates, the source panel goes back to tracking any process.

Source Panel Keystrokes

There are several special keys that may be used when the keyboard focus is in a source panel. The function of most keys is independent of the target line. Some keys, like b and h, do depend on the target line so that NightView can determine the source line of interest. See "Source Panel Target Line" on page 8-54. Note that the meaning of these keys does <u>not</u> change between source display mode and disassembly display mode. For example, s means step one line in any display mode.

In addition to these keys that work only in a source panel, there are shortcuts that work in any panel (except for a shell panel). See "List of Shortcuts" on page 8-26.

b	This key sets a breakpoint. It performs the same action as the Breakpoint button (see "Process Toolbar" on page 8-20).
d	This key changes the current frame to the callee. This is similar to using the **down** command with no argument. See "down" on page 6-138.
Enter	Sets the line with the text cursor to be the target line. See "Source Panel Target Line" on page 8-54.
e	This key is similar to selecting the Edit item in the Source menu. See "Source Menu" on page 8-10. This key is disabled in disassembly display mode.

f	This key runs the process until it returns from the current frame. It is similar to using the **finish** command. See "finish" on page 6-129.
h	Run the process until it reaches the target line. This key is identical to the Run to Here button. See "Process Toolbar" on page 8-20. It combines the actions of **breakpoint**, **enable/delete**, and **resume**.
N	Step one instruction without entering called routines. This key is similar to using the **nexti** command with no argument. See "nexti" on page 6-128.
n	Step one line without entering called routines. This key is similar to using the **next** command with no argument. See "next" on page 6-126.
p	Print the result of evaluating the current selection. This key performs the same action as the Print button in the process toolbar. See "Process Toolbar" on page 8-20).
r	Resume the process. This key is similar to using the **resume** command with no argument. See "resume" on page 6-123.
S	Step one instruction, entering a called routine. This key is similar to using the **stepi** command with no argument. See "stepi" on page 6-127.
s	Step one line, entering any called routine. This key is similar to using the **step** command with no argument. See "step" on page 6-124.
u	This key changes the current frame to the caller. This is similar to using the **up** command with no argument. See "up" on page 6-137.
=	Move to the newest stack frame. This key is similar to using the **frame 0** command. See "frame" on page 6-136.
>	Print information about the current frame. This key is similar to using the **frame** command with no arguments. See "frame" on page 6-136.

Shell Panel

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 panel acts something like a little terminal. If your shell lets you do command-line editing, then you can do that in this panel, too. In addition to passing escape sequences and control keys that you type, NightView also passes standard escape sequences for the arrow keys (Right, Left, Up and Down) and the Home and End keys, for shells that recognize those sequences.

NightView shortcuts cannot be used here because the keystrokes are passed to the shell.

Any programs that you run in the shell can be debugged and manipulated by NightView.

Alternatives to using the shell panel are the Run… item in the Process menu, the **run** command, and giving the program and program arguments as arguments when invoking NightView. See "Process Menu" on page 8-8. See "run" on page 6-37.

Right-click in the panel to bring up the context menu. The shell panel's context menu contains these entries:

Find…

> Mnemonic: F

> Brings up the find bar and sets the keyboard focus there. See "Find Bar" on page 8-53.

Find again

> Mnemonic: G

> Finds the search string in the find bar again. The search begins from the text cursor position. At the end of each search the text cursor is left at the end of the found string.

Message Panel

This panel displays messages including process status messages, error messages, output from commands, and output from processes and the shell. All message panels have the same contents.

Right-click in the panel to bring up the context menu. The message panel's context menu contains these entries:

Find…

> Mnemonic: F

> Accelerator: Ctrl+F

> Brings up the find bar and sets the keyboard focus there. See "Find Bar" on page 8-53.

Find again

> Mnemonic: G

> Accelerator: Ctrl+G

> Finds the search string in the find bar again. The search begins from the text cursor position. At the end of each search the text cursor is left at the end of the found string.

Eventpoint Panel

The eventpoint panel shows you a table of existing eventpoints for all processes. It also provides ways for you to change eventpoints. See "Eventpoints" on page 3-9.

The table has a row for each eventpoint. The columns show:

- the type of the eventpoint (breakpoint, heappoint, monitorpoint, patch-point, tracepoint, syscallpoint, or watchpoint), with an icon for the type.

- the eventpoint ID number. Each eventpoint has a unique ID.

- the location of the eventpoint in the program. Watchpoints and syscall-points do not have a location in the program: for watchpoints, this area describes the location being watched; for syscallpoints, this area describes the system calls being traced.

- the ID of the process this eventpoint is in.

- whether the eventpoint is enabled.

- the ignore count.

- the hit count.

- the crossing count.

- whether the eventpoint has commands (only breakpoints, monitorpoints, syscallpoints, and watchpoints can have commands).

- any condition on the eventpoint.

If the program is stopped at a breakpoint, syscallpoint, or watchpoint, the eventpoint icon is overlaid with a green triangle pointing to the right, ▶ as it is in the source panel.

You can sort on the various columns in the table by clicking on the headers. You can rear-range the columns by dragging the headers.

One way to change eventpoints in this panel is to edit the fields in the table directly. The Enabled, Ignore and Condition fields can be edited by clicking on them. The field changes to a control appropriate for changing that field.

NOTE

It is important to click somewhere else after changing the fields in the table directly. The eventpoint will not be changed until you click somewhere else.

Right-click in the panel to bring up the context menu. You may select one or more event-points (rows) and then right-click. If you right-click on a row that is not selected, the selection is cleared and the row you clicked on becomes selected. If you right-click on a row that is selected, the selection does not change. The context menu's entries are enabled or disabled based on which rows are selected.

The eventpoint panel's context menu contains these entries:

Edit…

Mnemonic: I

Brings up a dialog box to change the attributes of the selected eventpoint. See "Eventpoint Dialog Boxes" on page 8-29.

Enable

Mnemonic: E

Enable the selected eventpoints. This is similar to using the **enable** command. See "enable" on page 6-113.

Disable

Mnemonic: D

Disable the selected eventpoints. This is similar to using the **disable** command. See "disable" on page 6-113.

Delete

Mnemonic: L

Delete the selected eventpoints. This is similar to using the **delete** command. See "delete" on page 6-112.

Once deleted, you cannot refer to these eventpoints again. 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.

Clear Ignore Count

Mnemonic: G

Set the ignore count to zero for the selected eventpoints.

Clear Commands

Mnemonic: M

Remove any commands on the selected eventpoints.

Clear Condition

Mnemonic: N

Remove any condition on the selected eventpoints.

List Source

Mnemonic: S

Show the source corresponding to the location of the selected eventpoint.

Resize Columns to Data

Mnemonic: Z

Adjust the width of the columns of the table to fit the data.

Update Now

Refresh the data in the table.

Warnings or errors associated with changing eventpoints are displayed in the message panel. See "Message Panel" on page 8-61.

You can also use the `info eventpoint` command to check eventpoint settings. See "info eventpoint" on page 6-146.

Context Panel

The context panel is a special data panel that lets you browse within the processes you are debugging and their stack frames. See "Data Panel" on page 8-64.

A context panel is essentially a specialized data panel that shows you shells, processes, threads, stack frames, and local variables within stack frames. However, it hides some of these panel components if they are uninteresting; shells are shown if you only multiple shells, process entries appear only if you have multiple processes, etc.

The current stack frame is shown in green underlined text. If processes are shown, the current process is shown in green underlined text. If shells are shown, the shell that contains the current process is shown in green underlined text.

To change the current context, click on a stack frame, process or shell. The source panels and the status bar are updated for that context and the items related to that context become green underlined text.

Locals Panel

A locals panel is a special data panel that has a single local variables data item. (The root of the data item is hidden.) This panel always shows the local variables for the current frame for the current process. See "Data Panel" on page 8-64. See "Local Variables Data Item" on page 8-68. See "Current Frame" on page 3-26. See "Current Process" on page 8-3.

Monitor Panel

A monitor panel is a special data panel that has a single monitorpoint values data item. (The root of the data item is hidden.) This panel always shows monitorpoint values. See "Data Panel" on page 8-64. See "Monitorpoint Values Data Item" on page 8-74. See "Current Frame" on page 3-26. See "Current Process" on page 8-3.

Data Panel

A data panel displays various information about your process. Each data panel has a name. If there is more than one data panel, then you are prompted for the name of a data panel when you place an item in the panel. If no panel of that name exists, then one is created with that name. There is no limit on the number of data panels.

Items are placed into a data panel by using the Data menu, or by using the Data Display button, or by using the data panel's context menu, or by invoking the `data-dis-`

play command. See "Data Menu" on page 8-14. See "Value Toolbar" on page 8-23. See "Data Panel Context Menu" on page 8-74. See "data-display" on page 6-92.

Monitor Bar

The monitor bar appears when you select Show Monitor Bar in the context menu for a monitorpoint values item or in a monitor panel. See "Data Panel Context Menu" on page 8-74.

This area contains a button to hide the monitor bar, the Hold/Release button, and a spinbox with the update interval in milliseconds.

Use the Hold/Release button to hold or release monitorpoint updates. The button is labeled Hold when updates are running, and Release when updates are held. You can also hold and release updates with the **mcontrol** command or its aliases **hold** and **release**. See "mcontrol" on page 6-108.

If you modify the update interval spinbox, you need to click somewhere else to have the value take effect.

Data Items

Each data item shows one piece of data from your process. The data items are arranged in a tree. A data item has a label and a value field. The format of the value field depends on the kind of data item.

If the data item has sub-items, then it appears with a small button to the left. If the button is shown with a +, then any sub-items the data item has are currently collapsed (not currently displayed). You can expand the sub-items by clicking on the + button.

If the button is shown with a -, then any sub-items the data item has are currently expanded (displayed). You can collapse the sub-items by clicking on the - button.

If there is no button with + or -, then the data item has no sub-items.

You can right-click on the data item to pop up a context menu for the data item. See "Data Panel Context Menu" on page 8-74.

You can change the size of scroll regions in this panel with the Data menu or the data panel context menu. See "Data Menu" on page 8-14.

For expression data items showing arrays, and for stack data items, you can extend the number of items shown by clicking on the arrowhead items ▲. ▼ See "Expression Data Item" on page 8-67 and "Stack Data Item" on page 8-69.

NightView may present the value field with the term "(invalid)" appended and using a red foreground. This happens with expression data items for pointer expressions, and for block data items and for some of the sub-items therein. For a pointer data item, this indicates that the pointer references nonexistent, freed, or never allocated memory (although NULL always is considered valid). For block data items, which are available if heap debugging is turned on, this indicates a heap block for which an error has been detected. For sub-items therein, it indicates the nature of the error.

Top-level data items may be moved by dragging them with the mouse. They may be moved to a different top-level position in the same data panel or to another data panel. Items in a context, locals or monitor panel may not be moved.

Expression Data Item

An expression data item displays the value of an expression, such as a variable.

The expression is re-evaluated whenever the process stops.

The expression is re-evaluated in the context that was current at the time the data item was created, or in the context that is current at the time of the re-evaluation, depending on the setting when you created the data item. See "Data Panel Add Expression" on page 8-81.

Figure 8-3. Expression Data Items

In the figure above, several different kinds of expressions have been added to the data panel.

If the value is a C struct or Ada record, then the sub-items are the members of the struct or record.

If the value is an array, the sub-items are the elements of the array. A limited number of elements is shown.

If the value is a pointer, the sub-item is the result of indirecting through the pointer.

Pointers can be treated as arrays or linked lists. When treated as a linked list, sub-items represent nodes in the linked list. See "Data Panel Linked List Expression Dialog" on page 8-85 on how linked lists are interpreted.

If the value is an array, or is being treated as an array, there are special arrowhead items on either end. ⌃ ⌄ Click on the arrowhead items to reveal more array elements. (Night-View lets you reference elements beyond the ends of an array.) Linked lists can be manipulated the same way, although such lists terminate when the link is NULL.

If heap debugging is turned on, and the value is a pointer which references heap memory, then it will have an additional sub-item named "`(heap info)`", which is a block data item describing the heap block containing the memory referenced by the pointer. See "Block Data Item" on page 8-73.

Local Variables Data Item

A local variables data item has sub-items for all the local variables visible in the current scope, including subprogram arguments. For C++ member functions, `this` is also included in the local variables.

This data item is updated whenever the process stops or you change the current context, e.g., by clicking on Up or Down in the process toolbar. See "Process Toolbar" on page 8-20.

Registers Data Item

A registers data item has sub-items for all of the registers.

This data item is updated whenever the process stops or you change the current context, e.g., by clicking on Up or Down in the process toolbar. See "Process Toolbar" on page 8-20.

Stack Data Item

A stack data item has sub-items for each frame on the stack. Initially, the number of frames shown is limited to the default number of stack frames. See "Data Menu" on page 8-14. The highest numbered stack frame is followed by an item with an arrowhead ⌄ . Click on the arrowhead item to reveal more stack frames. Once you have reached the end of the stack, the arrowhead item goes away.

Expanding a frame shows the local variables in that frame.

The current frame is shown in green underlined text. See "Current Frame" on page 3-26.

Figure 8-4. Stack Data Item

Threads Data Item

A threads data item has sub-items to describe Ada tasks and C threads. Expanding a sub-item shows the stack for that thread. If the program is not threaded, then it is considered to have one thread.

Figure 8-5. Threads Data Item

When only a single thread is running (see "Multithreaded Programs" on page 3-40 for a description of thread control), the thread's icon will be green with a small arrow wrapped around it, as shown in the figure above.

When the process is stopped, the current thread is shown in green underlined text.

Figure 8-6. Threads Data Item -- Expanded

Expanding a thread will show its stack frames -- expanding a stack frame will show the local variables associated with the stack frame.

You can switch context between threads by simply clicking on them in the panel. Switching context means that the current thread changes. The source panel will react to the change. Subsequent actions that cause the thread to run will cause the current thread to run, by itself when the run mode is **one**, or in tandem with all threads if the run mode is **all**. See "Run Mode Toolbar" on page 8-22 and "resume" on page 6-123.

Processes Data Item

A processes data item has sub-items to describe the processes you are debugging. The sub-items are the program's threads. If the program is not threaded, then it is considered to have one thread. The current process is shown in green underlined text.

Shells Data Item

A shells data item has sub-items to describe the shells you have. The sub-items are the processes you are debugging in each shell. The shell that has the current process is shown in green underlined text.

Normally you only have one shell active, but at times multiple shells are useful. You can open up a remote shell and debug processes on the remote system at the same time you debug processes on the local system. This is useful when debugging client and server processes that communicate across a network.

Note that you always have at least one shell, but by default, the local shell panel isn't displayed. You can view that shell panel by selecting New Shell Panel from the View menu.

Shell panels are used to invoke programs, provide input to programs that read from stdin, and to see output generated by such programs.

Shell data items simply represent the active shells in your NightView session in a data panel, and show you the current processes in each shell in tree form.

Heap Information Data Item

A heap information data item has the following sub-items:

- Totals
- Configuration

The Totals data item has the following sub-items which show totals for the heap:

- number of blocks ever allocated
- number of bytes ever allocated
- number of additional bytes of debugger overhead ever allocated
- number of blocks ever freed
- number of bytes ever freed
- number of additional bytes of debugger overhead ever freed

- number of blocks currently allocated

- number of bytes currently allocated

- number of additional bytes of debugger overhead currently allocated

- number of blocks currently freed but still retained

- number of bytes currently freed but still retained

- number of additional bytes of debugger overhead currently freed but still retained

The `Configuration` data item has the following sub-items which show the current configuration of heap debugging:

- whether heap debugging is on or off

- number of post-fence bytes and the post-fence fill byte

- number of pre-fence bytes and the pre-fence fill byte

- number of slop bytes

- whether free filling is enabled and the free fill byte

- whether malloc filling is enabled and the malloc fill byte

- whether hardware overrun protection is enabled

- frequency of automatic heap checks (i.e. the number of heap operations between automatic heap checks)

- maximum heap size, if any

- maximum number of retained free blocks

- maximum number of walkback frames per heap operation

- whether or not to check fill bytes of free blocks

Heap Errors Data Item

A heap errors data item describes the set of heap errors most recently reported (see "Heap Check" on page 3-34). The set of errors is updated whenever new errors are discovered by a heap operation, by an automatic heap check, or by a heappoint check. Additionally, the set of errors is updated whenever a heap check is requested explicitly by the user, regardless of whether or not it detects any errors.

The heap errors data item has one sub-item for each block with any errors in the last report. Each block item has sub-items for each separate error in the block.

Leak Sets / Still Allocated Sets Data Items

A leak sets data item contains a snapshot of any allocated blocks which likely leaked at the time that the data item is created. See "Leak Detection" on page 3-34 for accuracy limitations on leak detection.

A still allocated sets data item contains a snapshot of all allocated blocks at the time that the data item is created.

Note that these data items <u>do not</u> update automatically. In most programs, the determination of these sets can take a considerable amount of time, and this would impose an unacceptable delay if the sets were recomputed automatically.

For both of these data items, there is a sub-item for every set of heap blocks. A set contains all heap blocks with identical sizes and walkbacks at the time of their allocations (or most recent `reallocs`), regardless of other characteristics.

For each set data item, the following sub-items exist:

- walkback

- blocks

The walkback data item has a sub-item for each stack frame in the walkback at the time of the allocation (or most recent `realloc`) operation.

The blocks data item has a block sub-item for each individual block in the set.

Block Data Item

A block data item has the following sub-items:

- its state, which will be one of:

    - allocated

    - freed, but retained

    - freed or never allocated, but owned by heap

    - not owned by heap

    In the case of the latter two, no further sub-items are available.

- its address range

- its size in bytes

- an errors item

- an item for each heap operation (allocation, most recent `realloc`, and `free`) that has happened for the block

An errors data item contains a sub-item for each distinct error detected in the block, if any.

A heap operation (allocation, most recent `realloc`, or `free`) data item contains the following sub-items:

- configuration

- walkback

A heap operation's configuration data item contains the following sub-items:

- number and address range of post-fence bytes and the post-fence fill byte

- number and address range of pre-fence bytes and the pre-fence fill byte

- number of slop bytes

- whether free filling was enabled and the free fill byte

- whether malloc filling was enabled and the malloc fill byte

- whether hardware overrun protection was enabled

A heap operation's walkback data item has a sub-item for each stack frame in the walk-back at the time of its allocation (or most recent `realloc`) operation.

Monitorpoint Values Data Item

A monitorpoint values data item has a sub-item for each monitorpoint value. See "Moni-torpoints" on page 3-11.

Data Panel Context Menu

This menu pops up when you right-click in a data panel (or a context, locals or monitor panel). The menu lets you operate on the data item or its sub-items. In a data panel, the lower portion of the menu is the same as the Data menu and lets you add new data items or do other operations on the panel. See "Data Menu" on page 8-14. In a context, locals or monitor panel, you cannot add new items, so those menu entries do not appear.

Where you click determines which menu items appear. The menu items below appear only if they are appropriate for the type of the data item.

Collapse

> Mnemonic: L

Expand

> Mnemonic: X

> Either Collapse or Expand is shown depending on whether the sub-items are cur-rently expanded. Clicking on this button is the same as clicking on the + or - button.

Treat as Pointer to Single Item

> Mnemonic: P

> Click this radio button to consider the pointer as pointing to a single program ele-ment. The sub-item is the result of indirecting through the pointer.

Treat as Pointer to Array

> Mnemonic: A

> Click this radio button to consider the pointer as pointing to an array of elements, so that the sub-items are the elements of the array. The value column changes to show {array}.

Treat as Pointer to Linked List...

Mnemonic: K

Click this radio button to consider the pointer as pointing to a linked list of elements, so that the sub-items are the elements of the linked list. Clicking this radio button brings up a dialog so you can enter an expression that the debugger can use to get to successive elements. See "Data Panel Linked List Expression Dialog" on page 8-85.

The initial number of elements shown is the same as the number of indices to show when treating a pointer as an array. See "Data Panel Context Menu" on page 8-74. See "Preferences General Page" on page 8-40. As with an array, you can click on the arrowhead item ⌄ to show the next element. You can also use the Show nth Element... item in the data panel context menu to get to a particular element. The value column changes to show {linked list: *next-expression*}.

The list is considered to terminate if a next pointer is NULL or points to an item already displayed, or causes some error.

Expand Tree...

Mnemonic: T

Expand all the sub-items of this data item and their sub-items, etc. Clicking on this button pops up a dialog box to ask you how many levels of sub-items to expand. See "Data Panel Expand Tree" on page 8-83.

Collapse Tree

Mnemonic: R

All the sub-items of this data item and their sub-items, etc., are collapsed.

Re-evaluate

Mnemonic: V

The data item is re-evaluated. The new value is displayed in the value field.

Describe...

Pop up a dialog box with additional information about this data item. See "Data Panel Describe" on page 8-83.

Show Subscript...
Show nth Element...

Mnemonic: B (Show Subscript...)
Mnemonic: H (Show nth Element...)

This button is meaningful for arrays and linked lists. If the data item is a pointer, it is treated as an array. This menu item becomes Show nth Element... when the selected item is a linked list or when an array or linked list is filtered. You are prompted for a subscript or an element number. When you click on OK, the range of sub-items displayed is increased to include the subscript and the display is scrolled to make that sub-item visible.   See "Data Panel Subscript Array" on page 8-85. See "Data Panel Subscript Enum Array" on page 8-85.

Copy line to clipboard

Copies the selected line to the clipboard. The line is then available to be pasted into a text entry field, such as the command toolbar, with Ctrl-V. You may also click on either the label field or the value field and then use Ctrl-C to copy just that field to the clipboard.

Filter Elements with a Condition...
Change Condition Filter...

Use this to search for elements in an array or a linked list. Selecting this menu item brings up a dialog allowing you to enter a condition expression. When you click on OK, only elements matching the condition are displayed. See "Data Panel Condition Filter Expression Dialog" on page 8-86. If you already have a condition filter, this menu entry changes to Change Condition Filter...

Select Frame

Mnemonic: S

This button is meaningful only for a local variables data item (including sub-items of stack items). The frame becomes the current frame. See "Current Frame" on page 3-26.

Show Source

Mnemonic: H

This button is meaningful only for a local variables data item (including sub-items of stack items), a heap operation data item, a walkback data item or one of its sub-items. The location indicated by the data item is listed. See "list" on page 6-76.

Check Heap and Report New Errors

Mnemonic: N

This button is meaningful only for heap errors data items. It causes a heap check to be performed, searching for new errors since the last heap check was performed. See "Heap Check" on page 3-34.

Check Heap and Report All Errors

Mnemonic: A

This button is meaningful only for heap errors data items. It causes a heap check to be performed, searching for all errors. See "Heap Check" on page 3-34.

Update Block Errors

Mnemonic: U

This button is meaningful only for a block data item or block errors data item. It causes a heap check to be performed searching for all errors in the corresponding single block. See "Heap Check" on page 3-34.

Edit...

Mnemonic: E

This button pops up a dialog box that lets you modify the expression in this data item. See "Data Panel Edit Expression" on page 8-83.

Delete

Mnemonic: D

The data item is removed from the data panel.

Overriding Format

Mnemonic: F

This button pops up a sub-menu to allow you to control how the data item is printed in the value field. The Overriding Format applies to the data item and to each sub-item which does not override its own format.

None
None (use *format* from parent)
None (use *format* from monitorpoint)

Mnemonic: N

The value is normally displayed according to its type. See "print" on page 6-85. However, the format may be overridden by the item, by a parent item, or for a monitorpoint item, by the monitorpoint.

The precedence for choosing the format to display an item is:

1. the overriding format on the item

2. the overriding format on a parent item

3. the overriding format on the monitorpoint expression, for monitorpoint items only

4. the natural format for the type of the item

The label for this menu item varies depending on whether the format is currently overridden by a parent (or by the monitorpoint for a monitorpoint item).

Address + Offset

Mnemonic: A

The value is displayed in hexadecimal and as an address relative to a program symbol. See "print" on page 6-85.

Character

Mnemonic: C

The value is displayed in character format. See "print" on page 6-85.

Decimal

Mnemonic: D

The value is displayed in decimal format. See "print" on page 6-85.

Float

Mnemonic: F

The value is displayed in float format. See "print" on page 6-85.

Hexadecimal

Mnemonic: H

The value is displayed in hexadecimal format. See "print" on page 6-85.

Octal

Mnemonic: O

The value is displayed in octal format. See "print" on page 6-85.

String

Mnemonic: S

The value is displayed in string format. See "print" on page 6-85.

Unsigned Decimal

Mnemonic: U

The value is displayed in unsigned decimal format. See "print" on page 6-85.

Find...

Mnemonic: F

Accelerator: Ctrl+F

Brings up the find bar and sets the keyboard focus there. See "Find Bar" on page 8-53.

Find again

Mnemonic: G

Accelerator: Ctrl+G

Finds the search string in the find bar again. The search begins from the selected data item. At the end of each search item found is selected.

The following menu items are similar to the items in the Data menu. See "Data Menu" on page 8-14.

New Expression...

The dialog box allows you to enter an expression. A data item for that expression is placed in the data panel. See "Data Panel Add Expression" on page 8-81. See "Expression Data Item" on page 8-67.

New Local Variables...

A local-variables data item is placed in the data panel. See "Local Variables Data Item" on page 8-68.

New Registers...

A registers data item is placed in the data panel. See "Registers Data Item" on page 8-68.

New Stack...

A stack data item is placed in the data panel. See "Stack Data Item" on page 8-69.

New Threads...

A threads data item is placed in the data panel. See "Threads Data Item" on page 8-70.

New Processes...

A processes data item is placed in the data panel. See "Processes Data Item" on page 8-71.

New Shells...

A shells data item is placed in the data panel. See "Shells Data Item" on page 8-71.

New Heap Information...

A heap information data item is placed in the data panel. See "Heap Information Data Item" on page 8-71.

New Heap Errors...

A heap errors data item is placed in the data panel. See "Data Panel Add Heap Errors" on page 8-82. See "Heap Errors Data Item" on page 8-72.

New Heap Leaks...

A heap leaks data item is placed in the data panel. See "Data Panel Add Heap Leaks" on page 8-82. See "Leak Sets / Still Allocated Sets Data Items" on page 8-72.

New Still Allocated Blocks

A still allocated blocks data item is placed in the data panel. See "Data Panel Add Still Allocated Blocks" on page 8-82. See "Leak Sets / Still Allocated Sets Data Items" on page 8-72.

New Monitorpoint Values

A monitorpoint values data item is placed in the data panel. See "Monitorpoint Values Data Item" on page 8-74.

Save Snapshot...

This menu item lets you save the current contents of the data panel to a text file. Clicking on this button brings up a dialog box that lets you specify the name of a file in which to save the data. You can also record a comment in the file to describe the data that is being saved. See "Data Panel Save Snapshot" on page 8-84.

Save Layout...

Selecting this menu item pops up a dialog box that lets you save the layout of all the data items for a particular process in all the data panels. The information saved includes the type and format of each data item, and to which data panel the item belongs. See "Data Panel Save Layout" on page 8-84.

Load Layout...

Selecting this menu item pops up a dialog box that lets you load a saved layout for one or more processes. Any data panels mentioned in the layout are created if they do not exist. See "Data Panel Load Layout" on page 8-83.

Set Stack Frames...

Clicking on this button pops up a dialog box that lets you set the number of stack frames displayed for items in this panel. See "Data Panel Call Stack Frames" on page 8-82.

Set Thread Name...

Selecting this menu item pops up a dialog box which allows you to set the thread's name. See "set-thread-name" on page 6-143 for more information. This option is only shown for context menus associated with thread items.

Set Pointer as Array Indices...

Clicking on this button pops up a dialog box that lets you set the number of elements to show when displaying a pointer as an array. See "Data Panel Pointer Array Dimension" on page 8-84.

The remaining menu items appear only in a monitor panel or in a data panel when clicking on a monitorpoint value item.

Hold Monitor Updates

Clicking on this button causes monitorpoint items to stop updating. See "mcontrol" on page 6-108. When monitorpoint updates are held, this button reads Release Monitor Updates and clicking it causes monitorpoint items to resume updating.

Change Update Interval...

Mnemonic: I

Selecting this menu item pops up a dialog box that lets you change the interval between monitorpoint updates. See "Monitorpoint Update Interval Dialog Box" on page 8-87.

Show Monitor Bar

Clicking on this button causes the monitor bar to be shown. See "Monitor Bar" on page 8-65.

Data Panel Dialog Boxes

This section describes the dialog boxes related to the data panel. For other dialog boxes, see "Main Window Dialog Boxes" on page 8-26.

Data Panel Item Dialog Box

This section describes common information for several of the data panel dialog boxes. These dialog boxes all contain controls to set the data panel name, plus OK, Cancel and Help buttons.

Data Panel Name

Enter the name of the data panel to receive the data item, or select a name from the list by clicking on the arrow.

If no data panel exists with this name, one is created. The default name is "Data".

OK

Click on this button to perform the operation and dismiss the window.

Cancel

Click on this button to dismiss the window without performing any operation.

Help

Click on this button to get help about the specific dialog box.

Data Panel Add Expression

This dialog box pops up when you click on Expression… in the Data menu. See "Data Menu" on page 8-14.

Enter an expression and click on OK to add the expression to the data panel.

Radio buttons let you select the context for later re-evaluation.

When the expression is re-evaluated, it can be evaluated in the current context at the time of the re-evaluation ("Always evaluate in context where process stops"), or it can be evaluated with the context saved when the expression data item is created ("Always evaluate in context saved with expression.").

This is a Data Panel Item Dialog Box. See "Data Panel Item Dialog Box" on page 8-81.

Data Panel Add Heap Errors

This dialog box pops up when you click on Heap Errors… in the Data menu. See "Data Menu" on page 8-14.

Click on OK to add a heap errors data item to the data panel.

This dialog box contains 3 mutually exclusive buttons which may be used to perform a heap check before displaying the heap errors data item. If Don't Check Heap First, the default, is selected, then no heap check is performed and the last reported heap errors are displayed. If Check Heap for New Errors First is selected, then a heap check is performed looking for new errors since the last heap check, and those errors are displayed. If Check Heap for All Errors First is selected, then a heap check is performed looking for all errors, and those errors are displayed. See "Heap Errors Data Item" on page 8-72.

This is a Data Panel Item Dialog Box. See "Data Panel Item Dialog Box" on page 8-81.

Data Panel Add Heap Leaks

This dialog box pops up when you click on Heap Leaks… in the Data menu or in the data panel context menu. See "Data Menu" on page 8-14.

Click on OK to add a leak sets data item to the data panel.

This dialog box contains 2 mutually exclusive buttons. If New is selected, then only new leaks since the last leak report will displayed. If All is selected, then all leaks will be displayed. See "Leak Sets / Still Allocated Sets Data Items" on page 8-72.

This is a Data Panel Item Dialog Box. See "Data Panel Item Dialog Box" on page 8-81.

Data Panel Add Still Allocated Blocks

This dialog box pops up when you click on Still Allocated Blocks… in the Data menu or in the data panel context menu. See "Data Menu" on page 8-14.

Click on OK to add a still allocated sets data item to the data panel.

This dialog box contains 2 mutually exclusive buttons. If New is selected, then only blocks still allocated and allocated since the last still allocated blocks report will displayed. If All is selected, then all blocks still allocated will be displayed. See "Leak Sets / Still Allocated Sets Data Items" on page 8-72.

This is a Data Panel Item Dialog Box. See "Data Panel Item Dialog Box" on page 8-81.

Data Panel Call Stack Frames

This dialog box pops up when you click on Call Stack Frames… in the Data menu or in the data panel context menu. See "Data Menu" on page 8-14.

Use the spin box to enter the number of frames you would like to see for items in this data panel.

Click on the check button if you want to change the number of frames displayed for all the existing items in this data panel. Otherwise, only future data items use the new number of frames.

Click on OK to complete the operation.

Data Panel Edit Expression

This dialog box pops up when you click on Edit… in the data panel context menu. See "Data Panel Context Menu" on page 8-74.

Change the expression as desired.

The controls are the same as the Data Panel Add Expression dialog box. See "Data Panel Add Expression" on page 8-81.

Click on OK to complete the operation.

Data Panel Expand Tree

This dialog box pops up when you click on Expand… in the data panel context menu. See "Data Panel Context Menu" on page 8-74.

Use the spin box to enter how many levels of sub-items you want expanded.

Click on OK to complete the operation.

Data Panel Describe

This dialog box pops up when you click on Describe… in the data panel context menu. See "Data Panel Context Menu" on page 8-74.

The dialog box contains additional information about the data item.

Click on OK to complete the operation.

Data Panel Load Layout

This dialog box pops up when you select Load Layout… in the Data menu or the data panel context menu. See "Data Menu" on page 8-14.

It allows you to select a file from which to load data item layout information for the current process.

This is a file selection dialog box.

Select a filename.

Use the file selection controls to select a file. If you double-click on a filename in the Files section, the OK button is activated.

Choose an action button.

Click on Open to load the data items and close the window.

Clicking on Cancel cancels the action and closes this dialog box.

Data Panel Pointer Array Dimension

This dialog box pops up when you click on Pointer Array Dimension… in the Data menu or the data panel context menu. See "Data Menu" on page 8-14.

Use the spin box to enter the number of array elements you would like to see for pointers being treated as arrays in this data panel.

Click on the check button if you want to change the number of elements displayed for all the existing items in this data panel. Otherwise, only future data items use the new number of elements.

Click on OK to complete the operation.

Data Panel Save Layout

This dialog box pops up when you select Save Layout… in the Data menu or the data panel context menu. See "Data Menu" on page 8-14.

It allows you to select a file in which to save the layout information for all data items for a particular process, for use in a future debug session.

This is a file selection dialog box with an additional list for selecting a process.

Select a filename.

Use the file selection controls to select a file. If you double-click on a filename in the Files section, the OK button is activated.

Select a process from the list.

Choose an action button.

Click on OK to save the layout and close the window.

Clicking on Cancel cancels the action and closes this dialog box.

Click on Filter to update the files displayed based on the current filter string.

You can get help for this dialog box by clicking on Help.

Data Panel Save Snapshot

This dialog box pops up when you select Save Snapshot… in the Data menu or the data panel context menu. See "Data Menu" on page 8-14. It allows you to select a file in which to save a snapshot of the data panel.

The items are saved at their current level of expansion. For example, if a struct is shown in the data panel, but it is not expanded (i.e., the members of the struct are not shown), then the struct is saved in the snapshot, but the members of the struct are not.

This is a file selection dialog box.

Select a filename.

Use the file selection controls to select a file. If you double-click on a filename in the Files section, the OK button is activated.

Enter comments.

Enter some comments to save with this file. For example, "Stopped in blarg just before error occurs." The comments are saved at the beginning of the snapshot, followed by a timestamp.

Select append or overwrite.

Use the radio buttons to indicate whether the snapshot should overwrite the file or append to it if it already exists.

Choose an action button.

Click on OK to save the snapshot and close the window.

Clicking on Cancel cancels the action and closes this dialog box.

Data Panel Subscript Array

This dialog box pops up when you click on Show Subscript… or Show nth Element… in the data panel context menu. See "Data Panel Context Menu" on page 8-74.

Use the spin box to enter the subscript of the array element, linked list element or filter element you want to see.

If the array or linked list is filtered, then the subscript you specify is applied to the filter, not to the underlying array or linked list. Enter **7** to see the 8th item that matches the filter.

Click on OK to show the element.

Data Panel Subscript Enum Array

This dialog box pops up when you click on Show Subscript… in the data panel context menu and the array is indexed by an Ada enumeration type. See "Data Panel Context Menu" on page 8-74.

Use the combo box to select the array subscript name.

Click on OK to show the array element.

Data Panel Linked List Expression Dialog

This dialog pops up when you click on the Treat as Pointer to Linked List… radio button in data panel context menu (see "Data Panel Context Menu" on page 8-74).

Enter an expression in the combo box. The debugger uses this expression to get to the next element in the linked list. Note that any side effects in the expression, such as assignment, will affect your process. When the expression is evaluated, a temporary convenience variable ("Convenience Variables" on page 3-37), $p, holds the address of the current element. For example, if you have a linked list made of structures like this:

```
struct foo {
  struct foo * next;
  int value;
};
```

you would enter this as the expression:

```
$p->next
```

NightView is often able to supply a suitable expression. If there are multiple pointers in the structure, there may be multiple entries in the combo box.

Click on OK to complete the operation.

Clicking on Cancel cancels the action and closes this dialog box.

Data Panel Condition Filter Expression Dialog

This dialog pops up when you click on Filter Elements with a Condition… or Change Condition Filter… in the data panel context menu.

Enter an expression in the combo box. The debugger evaluates this expression for each element of the array or linked list. The debugger displays only the elements for which the expression value is true.

When the expression is evaluated, some temporary convenience variables are available to be used in the expression. See "Convenience Variables" on page 3-37.

$i   is the index of the element
$p   is the address of the element. This is not available for some element types.
$v   is the element.

Setting a condition filter does not increase the number of elements displayed for the underlying array or linked list. If you want to see more filtered elements after you set the filter, you can click on the arrowhead items ▲ ▼ if they are available, or use Show nth Element… in the context menu (see "Data Panel Context Menu" on page 8-74). If no elements match, then none are displayed.

The combo box holds expressions you have entered previously.

Note that your process can be affected by side effects in the condition expression.

Example:

Stop in a main program that uses `argv`. In a local variables panel, right-click on argv to bring up the data panel context menu. Click on Treat as Pointer to Array. Then right-click on argv again and then click on Filter Elements with a Condition… Enter this condition:

$v != 0 && strstr($v, "HOME") != 0

You probably see only the up and down arrows. Click on the down arrow. You should see your HOME environment variable. (When your program starts, the environment variables are stored just beyond the program arguments.)

If you are having trouble getting your condition right, try an expression like this (note the comma after the `printf` call):

`printf("%p\n", $v), $v != 0 && strstr($v, "HOME") != 0`

This causes the program to print each pointer as the condition is evaluated.

Example:

If you have an array of structures like this:

```
struct foo {
  int value;
};
```

you might use this condition:

`$v.value > 5`

Example:

To see every fifth element, use this condition:

`($i % 5) == 0`

There is a limit on the number of array (or linked-list) elements NightView will search to find the next element to display. The limit is imposed to prevent NightView from using excessive resources and time in a fruitless search. You can adjust the limit here. You may also use the Interrupt button to stop a search. See "Process Toolbar" on page 8-20.

Click on OK to complete the operation. The value column for the array or linked list changes to show {filtered: *condition*}. Filtered elements are displayed in blue. The label field shows "…" at the end if the following element is not displayed due to not matching the condition.

Click on Cancel to cancel the action and close the dialog box.

Monitorpoint Update Interval Dialog Box

This dialog box pops up when you select Change Update Interval… in the context menu of the monitor panel or the data panel. See "Monitor Panel" on page 8-64. See "Data Panel" on page 8-64.

Enter the number of milliseconds to delay between updates.

Click on OK to change the interval and close the dialog box.

Click on Cancel to cancel the action and close the dialog box.

The update interval can also be changed with the **mcontrol** command. See "mcontrol" on page 6-108.

The monitorpoint update interval is not related to the eventpoint panel update interval. See "Eventpoint Panel Update Interval Dialog Box" on page 8-52.

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.

For a general discussion of NightView's online help, see "GUI Online Help" on page 8-1.

A
NightStar Licensing

NightStar RT uses the NightStar License Manager (NSLM) to control access to the Night-Star RT tools.

License installation requires a licence key provided by Concurrent (see "License Keys" on page A-1).The NightStar RT tools request a licence (see "License Requests" on page A-2) from a license server (see "License Server" on page A-2).

Two license modes are available, fixed and floating, depending on which product option you purchased. Fixed licenses can only be served to NightStar RT users from the local system. Floating licenses may be served to any NightStar RT user on any system on a network.

Tools are licensed per system, per concurrent user. A single license is shared among any or all of the NightStar RT tools for a particular user on a particular system. The intent is to allow *n* developers to fully utilize all the tools at the same time while only requiring *n* licenses. When operating the tools in remote mode, where a tool is launched on a local system but is interacting with a remote system, licenses are required only from the host system.

You can obtain a license report which lists all licenses installed on the local system, current usage, and expiration date for demo licenses (see "License Reports" on page A-3).

The default configuration includes a strict firewall which interferes with floating licenses. See "Firewall Configuration for Floating Licenses" on page A-3 for information on handling such configurations.

See "License Support" on page A-4 for information on contacting Concurrent for additional assistance with licensing issues.

License Keys

Licenses are granted to specific systems to be served to either local or remote clients, depending on the license model, fixed or floating.

License installation requires a license key provided by Concurrent. To obtain a license key, you must provide your system identification code. The system identification code is generated by the **nslm_admin** utility:

```
nslm_admin --code
```

System identification codes are dependent on system configurations. Reinstalling Linux on a system or replacing network devices may require you to obtain new license keys.

To obtain a license key, use the following URL:

<p style="text-align:center">**http://www.ccur.com/NightStarRTKeys**</p>

Provide the requested information, including the system identification code. Your license key will be immediately emailed to you.

Install the license key using the following command:

nslm_admin --install=*xxxx-xxxx-xxxx-xxxx-xxxx*

where *xxxx-xxxx-xxxx-xxxx-xxxx* is the key included in the license acknowledgment email.

License Requests

By default, the NightStar RT tools request a license from the local system. If no licenses are available, they broadcast a license request on the local subnet associated with the system's hostname.

You can control the license requests for an entire system using the **/etc/nslm.config** configuration file.

By default, the **/etc/nslm.config** file contains a line similar to the following:

:server @default

The argument @default may be changed to a colon-separated list of system names, system IP addresses, or broadcast IP addresses. Licenses will be requested from each of the entities found in the list, until a license is granted or all entries in the list are exhausted.

For example, the following setting prevents broadcast requests for licenses, by only specifying the local system:

:server localhost

The following setting requests a license from **server1**, then **server2**, and then a broadcast request if those fail to serve a license:

:server server1:server2:192.168.1.0

Similarly, you can control the license requests for individual invocations of the tools using the **NSLM_SERVER** environment variable. If set, it must contain a colon-separated list of system names, system IP addresses, or broadcast IP addresses as described above. Use of the **NSLM_SERVER** environment variable takes precedence over settings defined in **/etc/nslm.config**.

License Server

The NSLM license server is automatically installed and configured to run when you install NightStar RT.

The **nslm** service is automatically activated for run levels 2, 3, 4, and 5. You can check on these settings by issuing the following command:

/sbin/chkconfig --list nslm

In rare instances, you may need to restart the license server via the following command:

/sbin/service nslm restart

See **nslm(1)** for more information.

License Reports

A license report can be obtained using the **nslm_admin** utility.

nslm_admin --list

lists all licenses installed on the local system, current usage, and expiration date (for demo licenses). Use of the **--verbose** option also lists individual clients to which licenses are currently granted.

Adding the **--broadcast** option will list this information for all servers that respond to a broadcast request on the local subnet associated with the system's hostname.

See **nslm_admin(1)** for more options and information.

Firewall Configuration for Floating Licenses

RedHawk Linux does not support a firewall configuration by default, because iptables support is disabled. However, it is possible to build a custom kernel with iptables support enabled. If that is done, and floating licenses are used, the iptables firewall rules must be configured to allow the license requests and responses to pass.

If the system with iptables support and firewall rules is serving licenses, then the firewall rules must be arranged to allow license requests on UDP port 25517 and TCP port 25517 from any systems that will make license requests. For example, in a simple firewall, rules like the following, inserted before any DROP or REJECT rules, might work:

```
iptables -A INPUT -p udp -m udp -s subnet/mask --dport 25517 -j ACCEPT
iptables -A INPUT -p tcp -m tcp -s subnet/mask --dport 25517 -j ACCEPT
```

If the system with iptables support and firewall rules is running NightStar RT tools and receiving floating licenses, then the firewall rules must be arranged to allow license responses on UDP port 25517 from any system serving licenses. For example, in a simple firewall, rules like the following, inserted before any DROP or REJECT rules, might work:

```
iptables -A INPUT -p udp -m udp -s subnet/mask --sport 25517 -j ACCEPT
```

License Support

For additional aid with licensing issues, contact the Concurrent Software Support Center at our toll free number 1-800-245-6453. For calls outside the continental United States, the number is 1-954-283-1822. The Software Support Center operates Monday through Friday from 8 a.m. to 5 p.m., Eastern Standard Time.

You may also submit a request for assistance at any time by using the Concurrent Computer Corporation web site at **http://www.ccur.com/isd_support_contact.asp** or by sending an email to **support@ccur.com**.

B
Kernel Dependencies

Concurrent's RedHawk kernel provides features and performance gains that are critical for the optimal operation of the NightStar RT tools.

The NightStar RT tools can operate in a host-only mode on Red Hat systems without Concurrent's RedHawk kernel, cross-targeting to RedHawk systems.

Additionally, the NightStar RT tools can function on Red Hat systems without the RedHawk kernel, but will lack the numerous advantages afforded by running with it.

The following sections describe the additional functionality and capabilities of the NightStar RT tools when running Concurrent's RedHawk kernel

Advantages for NightView

The following advantages are afforded NightView when Concurrent's RedHawk kernel is running:

- Application speed conditions

    Provides "execution-speed" patches, conditions, and ignore counts.

- Signal handling

    Allows NightView to pass signals directly to a particular process, avoiding context switching.

Advantages for NightTrace

The following advantage is afforded NightTrace when Concurrent's RedHawk tracing kernel is running:

- Kernel tracing

    Users of NightTrace gain the ability to obtain kernel trace data and combine that with user trace data. Kernel tracing is an incredibly powerful feature that not only provides insight into the operating system kernel but also provides useful information relating to the execution of user applications.

    The RedHawk kernel is provided in three flavors:

    - Tracing

- Debug

- Plain

The Tracing and Debug flavors provide the features required for NightTrace kernel tracing. These kernels can be selected at boot-time from the boot-loader menu.

Advantages for NightProbe

The following advantages are afforded NightProbe when Concurrent's RedHawk a RedHawk or SLERT kernel is running:

- Minimal intrusion

  Allows NightProbe to read and write variables without stopping the process for each sample or write operation.

- Sampling performance

  Allows NightProbe to use direct memory fetches for data sampling (as opposed to programmed I/O) which is important for high-rate data acquisition.

- Concurrent debugging/probing

  Allows NightProbe to probe programs already under the control of a debugger or another NightProbe session.

- PCI Device probing

  Allows NightProbe to probe PCI device memory via the Base Address Register (BAR) file system.

  The PCI BAR File System is only available with the RedHawk kernel from Concurrent Computer Corporation and SLERT versions 1.0-1.6 kernel from Novell. On other systems, PCI Device probing will be disabled within NightProbe.

Advantages for NightTune

The following advantage is afforded NightTune when Concurrent's RedHawk a RedHawk or SLERT kernel is running:

- Context switch rate

  Allows NightTune user to display the context switch counts per CPU instead of for the overall system.

- CPU shielding

  Individual CPUs can be shielded from interrupts and processes allowing CPUs to be dedicated solely to specific interrupts and processes that are bound to the CPU.

- CPU sibling interference

  Individual CPUs can be marked down to avoid interfering with hyperthreaded sibling CPUs and dual-core sibling CPUs. Hyperthreaded CPUs share all the resources of their sibling CPU. Dual-core CPUs share the CPU cache and a path to memory with their sibling CPU.

- Detailed memory information

  Detailed process memory descriptions include the residency and lock state of any page in a process, and their association with physical memory pools for NUMA systems.

Frequency Based Scheduler

The Frequency Based Scheduler is only available on RedHawk systems from Concurrent Computer Corporation. It is required for all NightSim usage.

FBS Process Deadlines are only available for use on RedHawk 5.2.1 and later systems.

On systems without FBS Process Deadline support, the "Apply Deadline" group box will appear shaded and disabled.

NightSim is only included in NightStar distributions intended for use on RedHawk systems.

C
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 6-33 for more information.

apply on dialogue

>   Execute **on dialogue** commands for existing dialogues. See "apply on dialogue" on page 6-32 for more information.

apply on program

>   Execute **on program** commands for existing processes. See "apply on program" on page 6-48 for more information.

attach

>   Attach the debugger to a process that is already running. See "attach" on page 6-39 for more information.

backtrace
bt

>   Print an ordered list of the currently active stack frames. See "backtrace" on page 6-85 for more information.

breakpoint
b

>   Set a breakpoint. See "breakpoint" on page 6-99 for more information.

cd

>   Set the debugger's default working directory. See "cd" on page 6-75 for more information.

checkpoint

>   Take a restart checkpoint now. See "checkpoint" on page 6-49 for more information.

clear

>   Clear all eventpoints at a given location. See "clear" on page 6-109 for more information.

commands

> Attach commands to a breakpoint or monitorpoint. See "commands" on page 6-110 for more information.

condition

> Attach a condition to an eventpoint. See "condition" on page 6-111 for more information.

continue
c

> Continue execution and wait for something to happen. See "continue" on page 6-122 for more information.

core-file

> Create a pseudo-process for debugging an aborted program's core image file. See "core-file" on page 6-41 for more information.

data-display

> Control items in a data panel. See "data-display" on page 6-92 for more information.

debug

> Specify names for programs you wish to debug. See "debug" on page 6-26 for more information.

declare-thread-tag

> Declate the type of a thread tag. See "declare-thread-tag" on page 6-142 for more information.

define

> Define a NightView macro. See "define" on page 6-171 for more information.

delay

> Delay NightView command execution for a specified time. See "delay" on page 6-144 for more information.

delete
d

> Delete an eventpoint. See "delete" on page 6-112 for more information.

detach

> Stop debugging a list of processes. See "detach" on page 6-40 for more information.

directory

> Set the directory search path. See "directory" on page 6-78 for more information.

disable

> Disable an eventpoint. See "disable" on page 6-113 for more information.

display

> Add to the list of expressions to be printed each time the process stops. See "display" on page 6-92 for more information.

down

> Move one or more stack frames toward frames called by the current stack frame. See "down" on page 6-138 for more information.

echo

> Print arbitrary text. See "echo" on page 6-91 for more information.

edit

> Edit the current source file. See "edit" on page 6-79 for more information.

enable

> Enable an eventpoint for a specified duration. See "enable" on page 6-113 for more information.

exec-file

> Specify the location of the executable file corresponding to a process. See "exec-file" on page 6-44 for more information.

family

> Give a name to a family of one or more processes. See "family" on page 6-50 for more information.

finish

> Continue execution until the current function finishes. See "finish" on page 6-129 for more information.

forward-search
fo

> Search forward through the current source file for a specified regular expression. See "forward-search" on page 6-80 for more information.

frame
f

> Select a new stack frame or print a description of the current stack frame. See "frame" on page 6-136 for more information.

handle

> Specify how to handle signals and Ada exceptions in the user process. See "handle" on page 6-133 for more information.

heapcheck

Check the heap for errors. See "heapcheck" on page 6-165 for more information.

heapdebug

Specify parameters for heap debugging. See "heapdebug" on page 6-54 for more information.

heappoint

Check the heap for errors, or change the heap debugger settings, at a given location. See "heappoint" on page 6-107 for more information.

help

Access the online help system. See "help" on page 6-140 for more information.

ignore

Attach an ignore-count to an eventpoint. See "ignore" on page 6-114 for more information.

info address

Determine the location of a variable. See "info address" on page 6-167 for more information.

info args

Print description of current routine arguments. See "info args" on page 6-166 for more information.

info breakpoint
i b

Describe current state of breakpoints. See "info breakpoint" on page 6-147 for more information.

info convenience

Describe convenience variables. See "info convenience" on page 6-155 for more information.

info declaration
ptype

Print the declaration of variables or types. See "info declaration" on page 6-169 for more information.

info dialogue

Print information about active dialogues. See "info dialogue" on page 6-161 for more information.

`info directories`

Print the search path used to locate source files. See "info directories" on page 6-155 for more information.

`info display`

Describe expressions that are automatically displayed. See "info display" on page 6-155 for more information.

`info eventpoint`

Describe current state of breakpoints, tracepoints, patchpoints, monitorpoints, heap-points, and watchpoints. See "info eventpoint" on page 6-146 for more information.

`info exception`
`exception`

Print information about Ada exception handling. See "info exception" on page 6-164 for more information.

`info family`

Print information about an existing process family. See "info family" on page 6-162 for more information.

`info files`

Print the names of the executable, symbol table and core files. See "info files" on page 6-170 for more information.

`info frame`

Describe a stack frame. See "info frame" on page 6-154 for more information.

`info functions`

List names of functions, or subroutines, or Ada unit names. See "info functions" on page 6-168 for more information.

`info heappoint`

Describe the current state of heappoints. See "info heappoint" on page 6-151 for more information.

`info history`

Print value history information. See "info history" on page 6-155 for more information.

`info limits`

Print information about limits on expression and location output. See "info limits" on page 6-156 for more information.

info line

Describe location of a source line. See "info line" on page 6-170 for more information.

info locals

Print information about local variables. See "info locals" on page 6-166 for more information.

info log

Describe any open log files. See "info log" on page 6-146 for more information.

info macros

Print a description of one or more NightView macros. See "info macros" on page 6-175 for more information.

info memory

Print information about memory, which may include information about the virtual address space, or the heap. See "info memory" on page 6-158 for more information.

info monitorpoint

Describe current state of monitorpoints. See "info monitorpoint" on page 6-150 for more information.

info name

Print information about an existing eventpoint-name. See "info name" on page 6-162 for more information.

info on dialogue

Print **on dialogue** commands. See "info on dialogue" on page 6-163 for more information.

info on program

Print **on program** commands. See "info on program" on page 6-163 for more information.

info on restart

Print **on restart** commands. See "info on restart" on page 6-163 for more information.

info patchpoint

Describe current state of patchpoints. See "info patchpoint" on page 6-149 for more information.

info process

Describe processes being debugged. See "info process" on page 6-157 for more information.

info registers

>Print information about registers. See "info registers" on page 6-156 for more information.

info representation
representation

>Describe the storage representation of an expression. See "info representation" on page 6-169 for more information.

info signal

>Print information about signals. See "info signal" on page 6-157 for more information.

info sources

>List names of source files. See "info sources" on page 6-168 for more information.

info syscallpoint

>List information about syscallpoints. See "info syscallpoint" on page 6-153 for more information.

info threads

>Describe Ada tasks, C threads, and thread processes. See "info threads" on page 6-164 for more information.

info tracepoint

>Describe current state of tracepoints. See "info tracepoint" on page 6-148 for more information.

info types

>Print type definition information. See "info types" on page 6-168 for more information.

info variables

>Print global variable information. See "info variables" on page 6-167 for more information.

info watchpoint

>Describe current state of watchpoints. See "info watchpoint" on page 6-152 for more information.

info whatis
whatis

>Describe the result type of an expression visible in the current context. See "info whatis" on page 6-169 for more information.

interest

Control which subprograms are interesting. See "interest" on page 6-67 for more information.

jump

Continue execution at a specific location. See "jump" on page 6-131 for more information.

kill

Terminate a list of processes. See "kill" on page 6-40 for more information.

list
l

List a source file. See "list" on page 6-76 for more information.

load

Dynamically load an object file, possibly replacing existing routines. See "load" on page 6-95 for more information.

login

Login to a new dialogue shell. See "login" on page 6-24 for more information.

logout

Terminate a dialogue. See "logout" on page 6-30 for more information.

mcontrol
hold
release

Control the monitorpoint value display. See "mcontrol" on page 6-108 for more information.

monitorpoint

Monitor the values of one or more expressions at a given location. See "monitorpoint" on page 6-106 for more information.

mreserve

Reserve a region of memory in a process. See "mreserve" on page 6-53 for more information.

name

Give a name to a group of eventpoints. See "name" on page 6-98 for more information.

next

n

> Execute one line, stepping over procedures. See "next" on page 6-126 for more information.

nexti
ni

> Execute one instruction, stepping over procedures. See "nexti" on page 6-128 for more information.

nodebug

> Specify names for programs you do not wish to debug. See "nodebug" on page 6-27 for more information.

notify

> Ask about pending event notifications. See "notify" on page 6-39 for more information.

on dialogue

> Specify debugger commands to be executed when a dialogue is created. See "on dialogue" on page 6-30 for more information.

on program

> Specify debugger commands to be executed when a program is execed. See "on program" on page 6-46 for more information.

on restart

> Specify debugger commands to be executed when a program is restarted. See "on restart" on page 6-48 for more information.

output

> Print the value of a language expression with minimal output. See "output" on page 6-91 for more information.

patchpoint

> Install a small patch to a routine. See "patchpoint" on page 6-100 for more information.

print
p

> Print the value of a language expression. See "print" on page 6-85 for more information.

printf

> Print the values of language expressions using a format string. See "printf" on page 6-95 for more information.

pwd

Print NightView's current working directory. See "pwd" on page 6-75 for more information.

quit
q

Stop everything. Exit the debugger. See "quit" on page 6-23 for more information.

redisplay

Enable a display item. See "redisplay" on page 6-94 for more information.

refresh

Re-read source files and refresh the terminal screen. See "refresh" on page 6-141 for more information.

rerun

Run a program again. See "rerun" on page 6-37 for more information.

resume

Continue execution. See "resume" on page 6-123 for more information.

reverse-search

Search backwards through the current source file for a specified regular expression. See "reverse-search" on page 6-80 for more information.

run

Run a program in a dialogue and wait for NightView to start debugging it. See "run" on page 6-37 for more information.

select-context

Select the context of an Ada task, a thread, or a thread process. See "select-context" on page 6-138 for more information.

set

Evaluate a language expression without printing its value. See "set" on page 6-87 for more information.

set-auto-frame

Control the positioning of the stack when a process stops. See "set-auto-frame" on page 6-70 for more information.

set-children

Control whether children should be debugged. See "set-children" on page 6-51 for more information.

set-debug-file-directory

Tell NightView where to look for .debug files. See "set-debug-file-directory" on page 6-27 for more information.

set-disassembly

Control how NightView displays disassembled instructions. See "set-disassembly" on page 6-73 for more information.

set-download

Control how NightView downloads files from remote targets. See "set-download" on page 6-73 for more information.

set-editor

Set the mode for editing commands in the simple full-screen interface. See "set-editor" on page 6-71 for more information.

set-exit

Control whether a process stops before exiting. See "set-exit" on page 6-52 for more information.

set-history

Specify the number of items to be kept in the value history list. See "set-history" on page 6-62 for more information.

set-language

Establish a default language context for variables and expressions. See "set-language" on page 6-60 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 6-62 for more information.

set-local

Define process local convenience variables. See "set-local" on page 6-66 for more information.

set-log

Log session to file. See "set-log" on page 6-60 for more information.

set-notify

Control how you are notified of events. See "set-notify" on page 6-38 for more information.

set-overload

Control how NightView treats overloaded operators and routines in expressions. See "set-overload" on page 6-70 for more information.

set-patch-area-size

Control the size of patch areas created in your process. See "set-patch-area-size" on page 6-66 for more information.

set-preallocate

Control how NightView preallocates memory for eventpoints and monitorpoint buffers. See "set-preallocate" on page 6-71 for more information.

set-prompt

Set the string used to prompt for command input. See "set-prompt" on page 6-63 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 6-62 for more information.

set-restart

Control whether restart information is applied. See "set-restart" on page 6-65 for more information.

set-resume

Control NightView's behavior on events that normally stop a process. See "set-resume" on page 6-72 for more information.

set-safety

Control debugger response to dangerous commands. See "set-safety" on page 6-65 for more information.

set-search

Control case sensitivity of regular expressions in NightView. See "set-search" on page 6-71 for more information.

set-show

Control where dialogue output goes. See "set-show" on page 6-34 for more information.

set-terminator

Set the string used to recognize end of dialogue input mode. See "set-terminator" on page 6-64 for more information.

set-thread-name

> Set the name of a thread. See "set-thread-name" on page 6-143 for more information.

set-trace

> Establish tracing parameters. See "set-trace" on page 6-103 for more information.

shell

> Run an arbitrary shell command. See "shell" on page 6-141 for more information.

show

> Control dialogue output. See "show" on page 6-35 for more information.

signal

> Continue execution with a signal. See "signal" on page 6-132 for more information.

source

> Input commands from a source file. See "source" on page 6-141 for more information.

step
s

> Execute one line, stepping into procedures. See "step" on page 6-124 for more information.

stepi
si

> Execute one instruction, stepping into procedures. See "stepi" on page 6-127 for more information.

stop

> Stop a process. See "stop" on page 6-130 for more information.

symbol-file

> Establish the file containing symbolic information for a program. See "symbol-file" on page 6-41 for more information.

syscallpoint

> Set an eventpoint to trace one or more system calls. See "syscallpoint" on page 6-119 for more information.

tbreak

> Set a temporary breakpoint. See "tbreak" on page 6-115 for more information.

tpatch

Set a patchpoint that will execute only once. See "tpatch" on page 6-116 for more information.

tracepoint

Set a tracepoint. See "tracepoint" on page 6-104 for more information.

translate-object-file
xl

Translate object filenames for a remote dialogue. See "translate-object-file" on page 6-28 for more information.

undisplay

Disable an item from the display expression list. See "undisplay" on page 6-94 for more information.

up

Move one or more stack frames toward the caller of the current stack frame. See "up" on page 6-137 for more information.

wait

Wait for processes to stop. See "wait" on page 6-53 for more information.

watchpoint

Set a watchpoint. See "watchpoint" on page 6-117 for more information.

x

Print the contents of memory beginning at a given address. See "x" on page 6-88 for more information.

Invoking NightView

```
nview [-attach pid] [-config config-file] [-core core-file] [-help]
      [-nogui] [-nolocal] [-nx] [-prompt string]
      [-safety safe-mode] [-simplescreen] [-version]
      [-Xoption ...] [-x command-file]
      [program-name [program-argument ...]]
```

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* ...

set-debug-file-directory [*path*]

translate-object-file [*from* [*to*]]

    Abbreviation: **xl**

logout

on dialogue [*regexp*]

NightView RT User's Guide

> **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*
>
> **rerun**
>
> **set-notify** [silent | continuous=*mode*]
>
> **notify**
>
> **attach** [{/resume | /stop}] *pid*
>
> **detach**
>
> **kill**
>
> **symbol-file** *program-name*
>
> **core-file** *corefile-name* [exec-file=*program-name*]
>   [with-translations]
>
> **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]

wait [{all | any} [new]]

mreserve start=*address* {length=*bytes* | end=*address*}

Heap Debugging

```
heapdebug [check_free_fill={0|1}]
          [common_errors={block_overrun |
                          dangling_pointer |
                          uninitialized_field}]
          [do_free_fill={0|1}]
          [do_malloc_fill={0|1}]
          [error-name [{noprint |
                   nostop |
                   print |
                   stop } ...]]
          [free_fill_byte={n | trash}]
          [frequency=n[{k|m}]]
          [heap_size={n[{k|m}] | unlimited}]
          [internal_checks={0|1}]
          [level={0|1|2|3}]
          [malloc_fill_byte={n | trash}]
          [off]
          [on]
          [post_fence_size=n]
          [post_fill_byte={n | trash}]
          [pre_fence_size=n]
          [pre_fill_byte={n | trash}]
          [protected={0|1}]
          [retain_free_blocks={n[{k|m}] | unlimited}]
          [slop=n]
          [walkback=n]
```

error-name can be any of the following:

```
free_fill_modified
free_not_at_beginning
free_unallocated
internal_error
```

```
                             malloc_zero
                             memalign_not_power_2
                             out_of_memory
                             post_fence_modified
                             pre_fence_modified
                             realloc_not_at_beginning
                             realloc_unallocated
```

Abbreviation: **hd**

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*

set-preallocate [/eventpoint] [/monitorpoint] [{off | on}]

set-resume [/attach] [/exec] [/exit] [/fork] [{off | on}]

set-download [{off | permanent | temporary}]
               [directory=*path-to-cache*]

set-disassembly [flavor={att | intel}]
                [symbols={off | on}]
                [comment_level=*number*]

Debugger Environment Control

cd *dirname*

pwd

Source Files

Viewing and Editing Source Files

list *where-spec*

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

list ,*where-spec*

list *where-spec*,

list +

list –

list =

list

    Abbreviation: **l**

directory [*dirname ...*]

edit

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*

data-display [/window="*window name*"]
            {/kind=*value* | *expression*}

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*

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*[,*logged-expression*...]]
    [if *conditional-expression*]

monitorpoint [*eventpoint-modifier*] [name=*monitorpoint-name*]
    [[at] *location-spec*]

heappoint [*eventpoint-modifier*][name=*heappoint-name*]
    [[at] *location-spec*]
    [{check | debug *parameters*}}]
    [if *conditional-expression*]

mcontrol {display | nodisplay} [*monitorpoint-spec* ...]

mcontrol delay *milliseconds*

mcontrol {off | on | stale | nostale | hold | release}

  Abbreviation: **hold**

  Abbreviation: **release**

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*]

syscallpoint [*eventpoint-modifier*] [/nostop] [/before] [/after]
        [name=*patchpoint-name*] [*syscall-list*]
        [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 pid=*pid*

Miscellaneous Commands

help [*section*]

refresh

shell [*shell-command*]

source *command-file*

delay [*milliseconds*]

Info Commands

Status Information

info log

info eventpoint [/verbose] [*eventpoint-spec*] ...

info breakpoint [/verbose] [*eventpoint-spec*] ...

    Abbreviation: **i b**

info tracepoint [/verbose] [*eventpoint-spec*] ...

info patchpoint [/verbose] [*eventpoint-spec*] ...

info monitorpoint [/verbose] [*eventpoint-spec*] ...

info heappoint [/verbose] [*eventpoint-spec*] ...

info syscallpoint [*eventpoint-spec*] ...

info watchpoint [/verbose] [*eventpoint-spec*] ...

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 [/ranges] [/heap] [/leaks] [/allocated]
[/all] [/append=*filename*] [/output=*filename*]
[/verbose] [*expression*]

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

info threads [/verbose]

heapcheck [/all] [/append=*filename*] [/output=*filename*] [*expression*]

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*]

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 5 [Invoking NightView] on page 5-1.

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 `Night-View.p`.

`NightView.p` is responsible for controlling the user processes by a combination of the `/proc` file system and the **ptrace** system service.

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 3-11. The shared memory region is placed in your process at a preferred address if that address is available. Otherwise, it is placed anywhere Night-View can find space. On IA-32, the preferred address is `0xafe78000`. On AMD64, the preferred address is `0x15aaaa2000`.

Each dialogue runs a shell and controls it using `/proc` and **ptrace**. 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 DRn registers on IA-32 or AMD64 systems. Other eventpoints are implemented by replacing the instruction at the target address by a trap instruction. When your program hits the trap, the kernel translates this into 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 6-66. Each region is only created in the process if necessary.

On IA-32, NightView tries to put all the patch areas between 0xa0000000 and 0xb0000000.

On AMD64, an eventpoint patch area may need to be placed near the instruction being patched. NightView tries to put all the other patch areas between 0x15aaaab000 and 0x2000000000.

You can see where NightView has placed patch areas with the **info memory** command (see "info memory" on page 6-158).

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. On AMD64, NightView may need to create additional memory areas to hold this code.

To implement heap debugging, NightView patches a heap debugger module into your program and arranges for calls to malloc, free, and other heap functions, to be intercepted by this module. The heap debugger modifies each allocation request to allow for slop and fences and then passes the new request to the system allocator. The heap debugger remembers where each block is and what attributes it has.

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 <stdlib.h>
2   #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

main.f

```
 1  C      This program spawns a child process and sends
 2  C      signals from the parent to the child.
 3  C
 4         program  main
 5         common   /msg_comm/ total_sig
 6         common   /usr1_comm/ sigusr1
 7         integer  total_sig
 8         integer  sigusr1
 9       integer  pid
10         integer  ftfork
11         integer  ftsigusr1
12         character *8 tracefile
13         external parent_routine
14         external child_routine
15
16         sigusr1 = ftsigusr1()
17         tracefile = "msg_file"
18         write( 6, FMT='(A)' )
19       X   "How many signals should the parent send the child?"
20
21         read( 5,* ) total_sig
22         pid = ftfork()
23
24         if( pid .eq. 0 ) then
25  C        It's the child
26            call child_routine()
27         else
28  C        It's the parent
29            call parent_routine( pid )
30         end if
31
32         call exit
33         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         common    /usr1_comm/ sigusr1
 7         integer   child_pid
 8         integer   total_sig
 9         integer   sigusr1
10         integer   isec
11         integer   ireturn
12         integer   sig_ct
13         integer   kill
14         data      isec/2/
15
16         do 10 sig_ct = 1, total_sig
17             write( 6, FMT='(I3, A, I2, X, A)' ) sig_ct,
18       X     ".  Parent sleeping for", isec, "seconds"
19             call sleep( isec )
20             ireturn = kill( child_pid, sigusr1 )
21  10     continue
22
23         return
24         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         common    /usr1_comm/ sigusr1
 8         integer   total_sig
 9         integer   sig_ct_child
10         integer   sigusr1
11         integer   ireturn
12         integer   ftpause
13         integer   ftsignal
14         external  signal_handler
15         integer   signal_handler
16
17         ireturn = ftsignal( sigusr1, signal_handler )
18
19         do while ( sig_ct_child .lt. total_sig )
20             ireturn = ftpause()
21             ireturn = ftsignal( sigusr1, signal_handler )
22             write( 6, FMT='(A, I3)' ) "Child got ordinal signal #",
23       X         sig_ct_child
24         end do
25
26         return
27         end
```

```
28
29
30
31  C     Count how many signals have been received
32
33        integer function signal_handler()
34        common  /sig_comm/ sig_ct_child
35        integer sig_ct_child
36        data    sig_ct_child  /0/
37
38        sig_ct_child = sig_ct_child + 1
39        return
40        end
```

ftint.c

```
 1  /*
 2   * C routines to provide simple Fortran interfaces
 3   * to some system services, so that the tutorial
 4   * works the same way on different systems.
 5   */
 6
 7  #include <signal.h>
 8  #include <unistd.h>
 9
10  int
11  ftfork_()
12  {
13     int status = fork();
14     return status;
15  }
16
17  int
18  ftpause_()
19  {
20     int status = pause();
21     return status;
22  }
23
24  typedef void (*sig_handler)(int);
25
26  int
27  ftsignal_(int * signum, sig_handler handler)
28  {
29     int status = (int)signal(*signum, handler);
30     return status;
31  }
32
33  int
34  ftsigusr1_()
35  {
36     return SIGUSR1;
37  }
```

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 ada.text_io;
7    with ada.integer_text_io;
8    with posix_1003_1;
9
10   procedure main is
11
12      pid       : posix_1003_1.pid_t;
13      total_sig : integer;
14      tracefile : constant string := "msg_file";
15
16   begin
17
18      ada.text_io.put_line("How many signals should the parent send the
child?" );
19      ada.integer_text_io.get(total_sig);
20
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 ada.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        ada.text_io.put_line( integer'image(sig_ct) & ".  Parent sleep
ing 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

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 "List of Shortcuts" on page 8-26.

Ada task

Ada tasks are entities whose executions proceed in parallel. Different tasks proceed independently, except at points where they synchronize.

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 6-18.

application

A group of related processes. The processes may be running the same program or different programs.

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 6-39.

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 3-11. 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 3-16.

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 6-51.

command history

> NightView keeps a history of all the commands you enter. You can retrieve commands, edit them, and re-enter them. See "Command History" on page 3-38.

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 6 [Command-Line Interface] on page 6-1.

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 3-36.

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 3-25.

context menu

> In the graphical user interface, a context menu pops up when you right-click in a window. "Context" here refers to the position of the mouse in the window and has nothing to do with the debugging concept of *context*. See "Context Menu" on page 8-3.

context panel

> In the graphical user interface, a context panel is a convenient way to navigate in your process's *stack*, and also to navigate between processes and between shells. See "Context Panel" on page 8-64.

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 3-37.

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 3-26.

current process

In the graphical user interface, one process is the current process. Button clicks and commands apply to that process. If there is more than one process, the *context panel* indicates the current process with green text. See "Current Process" on page 8-3.

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*.

data item

Each data item shows one piece of data from your process in the *data panel*. A data item can show an expression, local variables, registers, the stack, or the process threads. See "Data Items" on page 8-65

data panel

In the graphical user interface, a data panel allows you to view data items in your process. See "Data Panel" on page 8-64.

data panel layout

The organization of *data items* in the *data panel* in your debug session. This information can be saved and reused in future debug sessions. See "Data Menu" on page 8-14.

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 session

A debug session is one invocation of NightView; it lasts until you exit from the debugger. See Chapter 5 [Invoking NightView] on page 5-1. See "Quitting NightView" on page 6-23.

default font

The default font is specified by the Motif `fontList` resource and applies only to the graphical user interface. See "Preferences Dialog Box" on page 8-39.

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 6-40.

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*.

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 6-88. In the graphical user interface, you can view disassembly in the source panels by using the Source menu. See "Source Menu" on page 8-10.

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 6-92 and "info display" on page 6-155.

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 ID*s. 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*, *heappoint*, *tracepoint*, *patchpoint*, and *watchpoint*. See "Eventpoints" on page 3-9.

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 6-98.

eventpoint panel

In the graphical user interface, an eventpoint panel is a convenient way to view and modify information about *eventpoints*. See "Eventpoint Panel" on page 8-61.

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 6-133. See "info exception" on page 6-164.

family

A group of related processes. See "family" on page 6-50.

fence

A region of memory preceding or following a block, when heap debugging is turned on. It can be used to detect block overrun bugs. See "Fences" on page 3-32.

find bar

In the graphical user interface, you can use the *context menu* to bring up a find bar in a panel to search for text in that panel. See "Find Bar" on page 8-53.

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 7 [Simple Full-Screen Interface] on page 7-1.

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 8 [Graphical User Interface] on page 8-1.

GUI

A *graphical user interface*.

heap

Memory allocated and deallocated dynamically via calls to `malloc` and `free`. NightView can help you debug problems with heap usage. See "Debugging the Heap" on page 3-30.

heap check

A complete check of all allocated and retained free blocks in the heap, when heap debugging is turned on. It can detect numerous heap-related bugs. See "Heap Check" on page 3-34.

heap debugger

A module that NightView loads into your process to monitor memory allocation and deallocation. See "Debugging the Heap" on page 3-30.

heappoint

An *eventpoint* that checks the *heap* or changes the heap debugging parameters. See "Heappoints" on page 3-12.

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 8-88.

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 3-11.

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 satisfied. 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 tracepoint logs a trace event, and so on. See *eventpoint*, *breakpoint*, *monitorpoint*, *tracepoint*, *heappoint*, 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 6-114.

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 3-38.

inline subprogram

> A subprogram that is expanded directly into the calling program. See "Inline Subprograms" on page 3-27.

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 3-27. You can change or query this level with the **interest** command. See "interest" on page 6-67.

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 3-9.

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 6-67.

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.

leaked block

A block which was allocated from the heap, which was never freed, and which the program no longer references. See "Leak Detection" on page 3-34.

locals panel

In the graphical user interface, a locals panel displays the local variables in the *current frame* of the *current process*. See "Locals Panel" on page 8-64.

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 6-170.

message panel

In the graphical user interface, a message panel displays process status, command output, error messages and process and shell output. See "Message Panel" on page 8-61.

mnemonic

A mnemonic is a way of selecting a menu or a menu item quickly in the graphical user interface. See also *accelerator*. See "List of Shortcuts" on page 8-26.

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 Night-View. Monitorpoints thus provide a means of viewing program data while the program is executing. See "Monitorpoints" on page 3-11 and "monitorpoint" on page 6-106.

monitor bar

> In the graphical user interface, in the *monitor panel*, you can use the *context menu* to bring up a monitor bar to control the display of *monitorpoint* values. See "Monitor Bar" on page 8-65.

monitor panel

> In the graphical user interface, a monitor panel displays the values of *monitorpoints*. See "Monitor Panel" on page 8-64.

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 6-140. See "GUI Online Help" on page 8-1.

overloading

> Overloading means that more than one entity with the same name is visible at some point in the program. See "Overloading" on page 3-24.

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 3-12.

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 6-66.

patchpoint

> A patchpoint is a location in a debugged process where a patch is inserted. See *patch*. See "Patchpoints" on page 3-12.

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 6-26.

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 3-19.

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 3-25.

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 6-6.

remote dialogue

> A remote dialogue is a *dialogue* started on a system other than the one on which Night-View 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 3-16.

retained free block

A block which was freed by the user, but which the heap debugger has not yet made available for reuse. It can be used to detect dangling pointer bugs. See "Retained Free Blocks" on page 3-33.

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 3-25.

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.

shell panel

In the graphical user interface, a shell panel provides a way to interact with your *dialogue* shell. See "Shell Panel" on page 8-60.

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 3-15.

source panel

In the graphical user interface, a source panel displays program source or *disassembly*. You can also interact with the process through the source panel. See "Source Panel" on page 8-53.

source panel target line

In the graphical user interface, When you click on a line in a source panel, that line becomes the source panel target line. Some buttons and keystrokes use the target line to identify the source file and line number to operate on. See "Source Panel Target Line" on page 8-54.

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 6-41.

syscallpoint

An *eventpoint* that informs you of system call entry and exit. See "Heappoints" on page 3-12.

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 3-40.

thread processes

Threaded programs are implemented with multiple processes that share resources, including memory. This manual refers to these processes as *thread processes*.

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 3-12.

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 3-38.

watchpoint

A watchpoint stops the process when the process reads or writes a variable in memory. See "Watchpoints" on page 3-13.

Index