



NightStar RT Tutorial

Version 5.3

(RedHawk™ Linux®)

Copyright 1990-2025 Concurrent Real-Time, Inc. All rights reserved. This publication or any part thereof is intended for use with Concurrent Real-Time products by Concurrent Real-Time personnel, customers, and end-users. It may not be reproduced in any form without the written permission of the publisher.

Concurrent Real-Time and its logo are registered trademarks of Concurrent Real-Time, Inc. All other Concurrent Real-Time product names are trademarks of Concurrent Real-Time while all other product names are trademarks or registered trademarks of their respective owners.

Linux[®] is used pursuant to a sublicense from the Linux Mark Institute.

NightStar's integrated help system is based on Assistant, a Qt[®] utility. Qt is a registered trademark of The Qt Company.

NVIDIA[®] CUDA[™] is a trademark of NVIDIA Corporation.

Preface

General Information

NightStar RT™ allows users running RedHawk Linux to schedule, monitor, debug and analyze the run-time behavior of their time-critical applications as well as the operating system kernel.

NightStar RT consists of the NightTrace™ event analyzer; the NightProbe™ data monitoring tool, the NightView™ symbolic debugger, the NightSim™ scheduler, the NightTune™ system and application tuner, the Data Monitoring API, and the Shmdefine shared memory utility.

Scope of Manual

This manual is a tutorial for NightStar RT.

Structure of Manual

This manual consists of seven chapters and an appendix which comprise the tutorial for NightStar RT.

Syntax Notation

The following notation is used throughout this guide:

italic

Books, reference cards, and items that the user must specify appear in *italic* type. Special terms and comments in code may also appear in *italic*.

list bold

User input appears in **list bold** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **list bold** type.

list

Operating system and program output such as prompts and messages and listings of files and programs appears in list type. Keywords also appear in list type.

emphasis

Words or phrases that require extra emphasis use emphasis type.

window

Keyboard sequences and window features such as push buttons, radio buttons, menu items, labels, and titles appear in **window** type.

[]

Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments.

{ }

Braces enclose mutually exclusive choices separated by the pipe (|) character, where one choice must be selected. You do not type the braces or the pipe character with the choice.

. . .

An ellipsis follows an item that can be repeated.

::=

This symbol means *is defined as* in Backus-Naur Form (BNF).

Referenced Publications

The following publications are referenced in this document:

0898395	<i>NightView™ User's Guide</i>
0898398	<i>NightTrace™ User's Guide</i>
0898465	<i>NightProbe™ User's Guide</i>
0898480	<i>NightSim™ User's Guide</i>
0898515	<i>NightTune™ User's Guide</i>

Contents

Chapter 1 Overview

Getting Started	1-2
Setting Up User Privileges	1-2
Creating a Tutorial Directory	1-4
Building the Program	1-4

Chapter 2 Panels

Moving Panels	2-2
Tabbed Panels	2-6
Context Menus	2-8
Tutorial Screen Shots	2-9

Chapter 3 Using NightView

Invoking NightView	3-2
Debugging Multiple Threads	3-5
Rerunning the Process	3-8
Traversing Linked Lists	3-10
Using Monitorpoints	3-16
Using Eventpoint Conditions and Ignore Counts	3-19
Using Patchpoints	3-20
Using Tracepoints	3-23
Using Lightweight Timing Points	3-25
Heap Debugging	3-28
Activating Heap Debugging	3-28
Setting up Heap Debugging Scenarios	3-31
Scenario 1: Use of a Freed Pointer	3-32
Scenario 2: Freeing an Invalid Pointer Value	3-35
Scenario 3: Writing Past the End of an Allocated Block	3-37
Scenario 4: Use of Uninitialized Heap Blocks	3-38
Scenario 5: Detection of Leaks	3-39
Scenario 6: Allocation Reports	3-41
Disabling Heap Debugging	3-43
Ready for NightTrace	3-43
Conclusion - NightView	3-43

Chapter 4 Using NightTrace

Invoking NightTrace	4-1
Configuring a User Daemon	4-3
Streaming Live Data to the NightTrace GUI	4-4
Halting the Daemon	4-4
Viewing Events	4-5

Using NightTrace Timelines	4-7
Zooming	4-8
Moving The Interval	4-9
Using the Events Panel for Textual Analysis	4-10
Customizing Event Descriptions	4-11
Searching the Events List.	4-12
Using States	4-15
Displaying State Duration.	4-19
Generating Summary Information	4-20
Defining a Data Graph	4-24
Kernel Tracing	4-28
Obtaining Kernel Trace Data	4-28
Using Prerecorded Kernel Data	4-30
Analyzing Kernel Data	4-31
Mixing Kernel and User Data.	4-35
Using the NightTrace Analysis API	4-39
Shutting Down	4-41
Conclusion - NightTrace	4-41

Chapter 5 Using NightProbe

Invoking NightProbe	5-1
Selecting Processes	5-2
Viewing Live Data	5-3
Modifying Variables	5-5
Selecting Variables for Recording and Alternative Viewing	5-7
Selection of Views	5-8
Table View	5-8
Graph View	5-11
Sending Probed Data to Other Programs	5-15
Using Datamon to Modify Program Variables.	5-19
Conclusion - NightProbe	5-21

Chapter 6 Using NightTune

Invoking NightTune	6-1
Monitoring a Process	6-2
Tracing System Calls	6-3
Process Details	6-4
Process Details - Memory Details	6-6
Process Details - Signals	6-8
Process Details - File Descriptors	6-8
Changing Process Scheduling Parameters	6-9
Setting Process CPU Affinity.	6-11
Setting Interrupt CPU Affinity.	6-14
Shielding CPUs for Maximum Determinism and Performance.	6-16
Conclusion - NightTune.	6-17

Chapter 7 Using NightSim

Creating FBS Applications.	7-1
Invoking NightSim.	7-2
Creating a Scheduler	7-3

Running the Scheduler	7-7
Using Datamon to Modify Program Variables	7-9
Overrun Detection and System Tuning	7-10
Shutting Down the Scheduler	7-15

A Tutorial Files

api.c	A-1
app.c	A-5
function.c	A-11
report.c	A-11
set_workload.c	A-11
set_rate.c	A-12
work.c	A-13
worker.c	A-13

Illustrations

Figure 2-1. Viewing Page with List & Graph Panels	2-2
Figure 2-2. Panel Detaches from Page	2-3
Figure 2-3. Panel Movement in Progress	2-4
Figure 2-4. Graph Panel on Top of List Panel	2-5
Figure 2-5. Table View added to Page	2-6
Figure 2-6. Panel in Motion Creating Tab	2-7
Figure 3-1. NightView Main Window	3-2
Figure 3-2. app Program Loaded	3-4
Figure 3-3. Context Panel With Stack Frames Expanded	3-6
Figure 3-4. Run Mode Selector	3-7
Figure 3-5. Set Patchpoint dialog with changes	3-9
Figure 3-6. Pointer to Linked List Expanded	3-11
Figure 3-7. Dialog Selecting Linked List Component	3-11
Figure 3-8. Pointer Variable Displayed As Linked List	3-12
Figure 3-9. Filter Dialog	3-13
Figure 3-10. Filtered Linked List	3-14
Figure 3-11. Filtered Linked List Expanded	3-14
Figure 3-12. Monitorpoint Dialog	3-17
Figure 3-13. NightView Monitor Panel	3-18
Figure 3-14. Patchpoint Dialog	3-22
Figure 3-15. Result of Patching in Call to Newly Loaded Function	3-25
Figure 3-16. Tracepoint Dialog	3-27
Figure 3-17. NightView Debug Heap Dialog	3-31
Figure 3-18. Heap Totals and Configuration	3-34
Figure 3-19. info memory Command Output	3-37
Figure 3-20. Heap Error Description	3-38
Figure 3-21. Heap Leaks Display	3-42
Figure 3-22. Still Allocated Blocks Display	3-44
Figure 4-1. NightTrace Main Window	4-2
Figure 4-2. Import Daemon Definitions Dialog	4-3
Figure 4-3. Logging Data	4-4
Figure 4-4. app_data Page	4-6
Figure 4-5. NightTrace Timeline	4-8
Figure 4-6. Timeline Interval Panel	4-10
Figure 4-7. Events Panel	4-11

Figure 4-8. Add Event Description dialog	4-12
Figure 4-9. Searching using the Profiles Dialog	4-13
Figure 4-10. Browse Events Dialog	4-14
Figure 4-11. Timeline Panel After Search	4-15
Figure 4-12. Events Panel After Search	4-16
Figure 4-13. Profiles Dialog With Obtuse Profile Selected	4-17
Figure 4-14. Timeline Editing	4-18
Figure 4-15. Edit State Graph Profile dialog	4-19
Figure 4-16. Sine State in Timeline	4-21
Figure 4-17. Summary Results Page	4-23
Figure 4-18. Summary Graph	4-24
Figure 4-19. State Durations Graph Modified	4-25
Figure 4-20. Timeline in Edit Mode	4-26
Figure 4-21. Adding a Data Graph	4-27
Figure 4-22. Edit Data Graph Profile Dialog	4-28
Figure 4-23. Display Page with Data Graph	4-29
Figure 4-24. Edit Daemon Definition Dialog	4-31
Figure 4-25. Kernel Display Page	4-33
Figure 4-26. System Call Resume for Nanosleep	4-36
Figure 4-27. Events Panel after Search	4-37
Figure 4-28. Longest Instance of State	4-39
Figure 4-29. Export Profiles to NightTrace API Source File dialog	4-41
Figure 4-30. nlight Wizard - Select Programs Step	4-45
Figure 4-31. nlight Wizard - Define Illuminators Step	4-47
Figure 4-32. nlight Wizard - Select Illuminators Step	4-48
Figure 4-33. nlight Wizard - Relink Programs Step	4-50
Figure 4-34. nlight Wizard - Activate Illuminators Step	4-52
Figure 4-35. NightTrace - Import File Name	4-54
Figure 4-36. NightTrace - Daemon Ready to Launch	4-55
Figure 4-37. NightTrace - Daemon Collection Events	4-56
Figure 4-38. NightTrace - /tmp/data_import Timeline	4-57
Figure 4-39. NightTrace - Events Panel w/ Tool Tip	4-58
Figure 4-40. NightTrace - Event Panel Search Dialog	4-59
Figure 4-41. NightTrace - Events Panel after Search	4-59
Figure 4-42. NightTrace - Events Panel Context Menu	4-60
Figure 4-43. NightTrace - Launches Editor with Source File at Line Number	4-61
Figure 4-44. NightTrace - Functions Summary Table	4-62
Figure 4-45. Function Details Table for the work function	4-63
Figure 5-1. NightProbe Main Window	5-2
Figure 5-2. Program Selection Dialog	5-3
Figure 5-3. Process Selection Dialog	5-3
Figure 5-4. NightProbe Browse Panel	5-4
Figure 5-5. Expanded Data Item	5-5
Figure 5-6. Variable Modification in Progress	5-6
Figure 5-7. Mark and Record Attributes Set	5-7
Figure 5-8. Table View	5-9
Figure 5-9. Item Selection Dialog	5-10
Figure 5-10. Table in Automatic Sampling Mode	5-11
Figure 5-11. Graph Panel	5-12
Figure 5-12. Graph Panel Actively Displaying Values	5-13
Figure 5-13. Edit Curve Attributes Dialog	5-14
Figure 5-14. Graph Panel with Modified Curves	5-15
Figure 5-15. Graph Zoomed In	5-16
Figure 5-16. Recording area of Configuration Page	5-18

Figure 5-17. Clock Selection Dialog	5-18
Figure 5-18. Record To Program Dialog	5-19
Figure 5-19. Recording Area of Configuration Page w/ Destination	5-20
Figure 5-20. Example Output of Graph Program	5-21
Figure 6-1. NightTune Initial Panels	6-2
Figure 6-2. Expanded Process List	6-3
Figure 6-3. Process List with Threads	6-3
Figure 6-4. Strace Output of Thread	6-4
Figure 6-5. Process Details Dialog	6-6
Figure 6-6. Process Memory Details Page	6-7
Figure 6-7. File Descriptors Page	6-9
Figure 6-8. Signals Page	6-10
Figure 6-9. Process Scheduler Dialog	6-11
Figure 6-10. NightTune Process List with modified thread	6-12
Figure 6-11. CPU Shielding and Binding Panel	6-13
Figure 6-12. CPU Shielding and Binding Panel with Bound Thread	6-14
Figure 6-13. NightTune with Interrupt Detail Activity Panel	6-16
Figure 6-14. Interrupt Affinity Dialog	6-17
Figure 7-1. NightSim Initial Window	7-3
Figure 7-2. NightSim Edit Process Dialog	7-5
Figure 7-3. Runtime Properties Tab	7-6
Figure 7-4. Scheduling Started	7-7
Figure 7-5. NightSim Monitor Page - Metrics Panel	7-7
Figure 7-6. NightSim Monitor Page - Percent of Period Used Panel	7-8
Figure 7-7. NightTune with Interrupt and CPU Shielding & Binding Panels ..	7-11
Figure 7-8. Process and Interrupt Bound to CPU 0	7-12
Figure 7-9. Change Shielding Dialog	7-13
Figure 7-10. Shielding Changes Pending	7-13
Figure 7-11. NightSim Percentage of Period Panel - Shielded CPU	7-15

1

Overview

NightStar RT™ is an integrated set of debugging tools for developing time-critical Linux® applications. NightStar RT are designed to be minimally intrusive, preserving the execution behavior and determinism of your applications. Users can quickly and easily debug, monitor, analyze, and tune their applications.

The NightStar RT tools consist of:

- NightView™ source-level debugger
- NightTrace™ event analyzer
- NightProbe™ data monitor
- NightTune™ system and application tuner
- NightSim™ scheduler

In this tutorial, we will integrate these tools into one cohesive example incorporating various scenarios which demonstrate their extensive functionality.

Getting Started

Certain activities in this tutorial require enhanced user privileges which are not granted to user accounts by default. You will need to obtain appropriate privileges as detailed in the “Setting Up User Privileges” on page 1-2 or run as the root user.

Setting Up User Privileges

Linux provides a means to grant otherwise unprivileged users the authority to perform certain privileged operations. **pam_capability(8)**, the Pluggable Authentication Module, is used to manage sets of capabilities, called roles, required for various activities.

Linux systems should be configured with a `nightstar` role which provides the capabilities required by NightStar RT. In order to take full advantages of NightStar RT features, each user must be configured to use (at a minimum) the capabilities specified below.

Edit `/etc/security/capability.conf` and define the `nightstar` role (if it is not already defined) in the “ROLES” section:

```
role nightstar cap_sys_nice cap_ipc_lock cap_sys_rawio
```

Additionally, for each NightStar RT user on the target system, add the following line at the end of the file:

```
user username nightstar
```

where *username* is the login name of the user.

If the user requires capabilities not defined in the `nightstar` role, add a new role which contains `nightstar` and the additional capabilities needed, and substitute the new role name for `nightstar` in the text above.

In addition to registering your login name in `/etc/security/capability.conf`, files under the `/etc/pam.d` directory must also be configured to allow capabilities to be activated.

IMPORTANT

Before modifying any file, create a terminal session on the system running as the root user. When editing the file, do not exit the editor until you have verified that you can still log into the system. Rather, write the file and attempt to log in again in another terminal session. If you make a typographical error or the stated file doesn't actually exist on your system you could be prevented from logging in again. Thus, you can undo your change, write the file and then exit the editor.

To activate capabilities, add the following line to the end of selected files in `/etc/pam.d` if it is not already present:

session required pam_capability.so

The table lists files that historically have been needed to be modified. In modern systems, it is likely that you only need to modify one or two files. For example, on Ubuntu 22.04, modifying **common-auth** is sufficient.

Be sure you test all the ways you access your system. For example, logging into the GUI console or using **ssh** to log into the system.

Table 1-1. Recommended /etc/pam.d Configuration

/etc/pam.d File	Affected Services	Comment
common-auth common-password passwd password-auth	Many services	On newer Linux systems, these files must be modified. If they exist, add the line to that file as well.
sshd	ssh	You should also edit /etc/ssh/sshd_config and ensure that the following line is present: UsePrivilegeSeparation no
gdm-password gdm	gnome sessions	If logging into the system via a graphical console, you may need to modify these files as well.
login	local login (e.g. console) telnet* rlogin* rsh* (when used <u>w/o</u> a command)	*On some versions of Linux, the presence of the remote file limits the scope of the login file to local logins. In such cases, the other services listed here with login are then affected solely by the remote configuration file.
rsh	rsh (when used <u>with</u> a command)	e.g. rsh system_name a.out
remote	telnet rlogin rsh (when used <u>w/o</u> a command)	Depending on your system, the remote file may not exist. Do not create the remote file, but edit it if it is present.
systemd-user	Some GUI sessions	
On legacy systems, the following files may also be useful:		
kde	kde sessions	

If you modify **/etc/pam.d/sshd** or **/etc/ssh/sshd_config**, you must restart the **sshd** service for the changes to take effect. The service is called **ssh** or **sshd**, depending on your system. Enter one of following:

```
systemctl restart sshd
systemctl restart ssh
```

In order for the above changes to take effect, the user must log off and log back onto the target system.

NOTE

To verify that you have been granted capabilities, issue the following command:

```
/sbin/getpcaps $$
```

The output from that command will list the roles currently assigned to you.

Creating a Tutorial Directory

We will start by creating a directory in which we will do all our work. Create a directory and position yourself in it:

- Use the **mkdir(1)** command to create a working directory.

We will name our directory **tutorial** using the following command:

```
mkdir tutorial
```

- Position yourself in the newly created directory using the **cd(1)** command:

```
cd tutorial
```

Source files, as well as configuration files for the various tools, are installed into **/usr/lib/NightStar/tutorial** during the installation of NightStar RT. We will copy these tutorial-related files to our **tutorial** directory.

- Copy all tutorial-related files to our local directory.

```
cp /usr/lib/NightStar/tutorial/* .
```

Building the Program

Our example uses a cyclic multi-threaded program which performs various tasks during each cycle. The cycle will be controlled by the main thread which uses a timeout with a configurable rate.

A portion of the main source file, **app.c**, is shown below:

```
int
main (int argc, char * argv[])
{
    pthread_t thread;
    pthread_attr_t attr;
    struct sembuf trigger = {0, 2, 0};
    const char * data_file = strdup("/tmp/data");
    nosighup();
```

```

if (argc > 1) {
    data_file = argv[1];
}
trace_begin (data_file, NULL);

sema = semget (IPC_PRIVATE, 1, IPC_CREAT+0666);

pthread_attr_init(&attr);
pthread_create (&thread, &attr, watchdog_thread, NULL);

pthread_attr_init(&attr);
pthread_create (&thread, &attr, sine_thread, &data[0]);

pthread_attr_init(&attr);
pthread_create (&thread, &attr, cosine_thread, &data[1]);

pthread_attr_init(&attr);
pthread_create (&thread, &attr, heap_thread, NULL);

for (;;) {
    struct timespec delay = { 0, rate } ;
    nanosleep(&delay, NULL);
    work(random() % 1000);
    if (state != hold) {
        semop(sema, &trigger, 1);
    }
}

trace_end ();
}

```

The program creates four threads and then enters a loop which cyclically wakes each of two threads based on a common timeout. The third and fourth threads, `heap_thread` and `watchdog_thread`, run asynchronously.

To build the executable

Type **make** to build all the programs needed in this tutorial.

2 Panels

NightStar provides flexibility in configuring the graphical user interface to suit your needs through the use of resizable and movable panels.

This chapter presents the concepts involved in moving and resizing panels. It is designed merely for reference, not as a step-by-step instructional guide.

If you are already familiar with NightStar GUI panel operations, then you can skip to “Using NightView” on page 3-1.

Moving Panels

Consider the following NightProbe page which contains a List view and a Graph view each in their own panel:

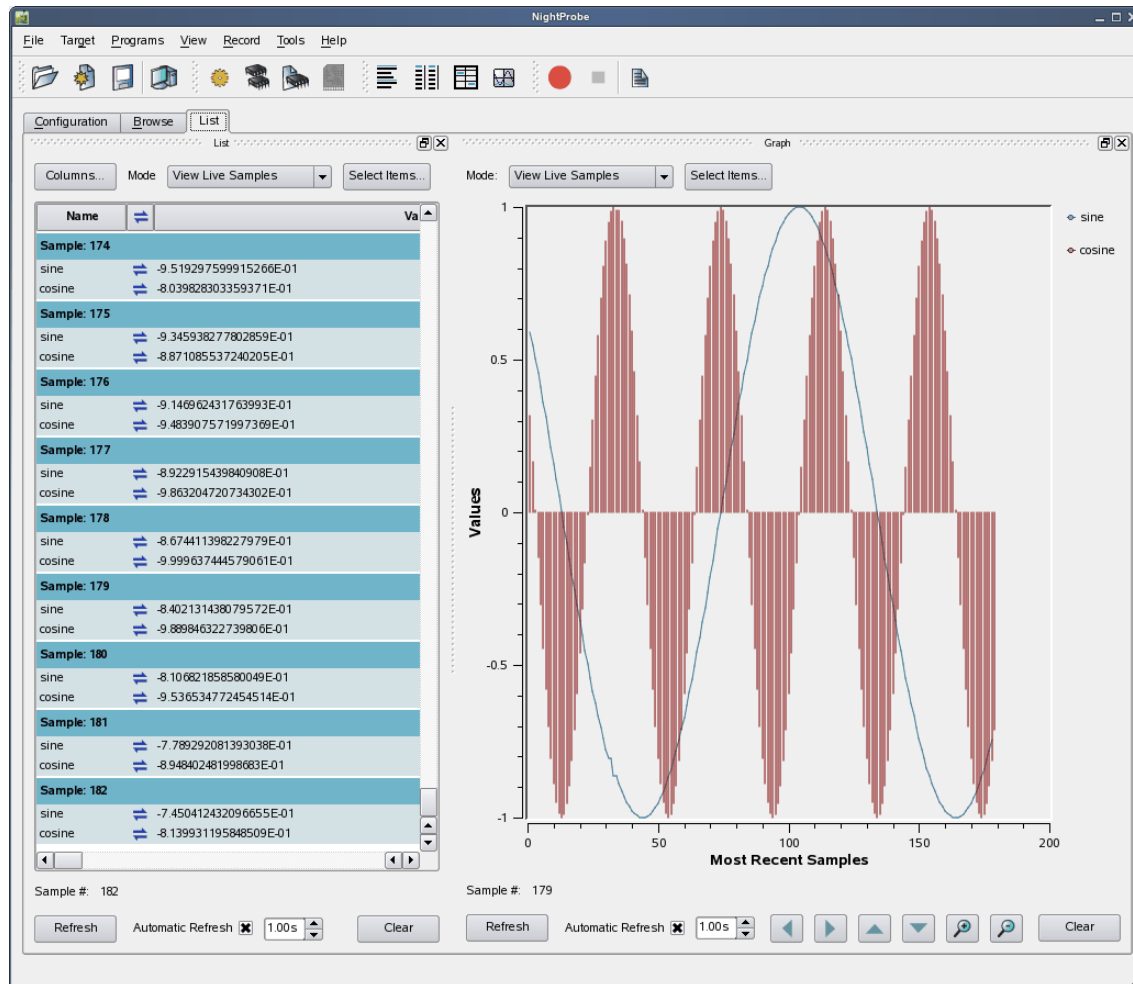


Figure 2-1. Viewing Page with List & Graph Panels

Panels are moved by left-clicking the title bar, dragging them to a new location, and then releasing the mouse button. Depending on the location of the panel when the mouse button is released, the panel will either remain detached or will be inserted into the page again.

To detach the panel from the page without inserting it, click the left-most control box in the upper right-hand corner of the panel.

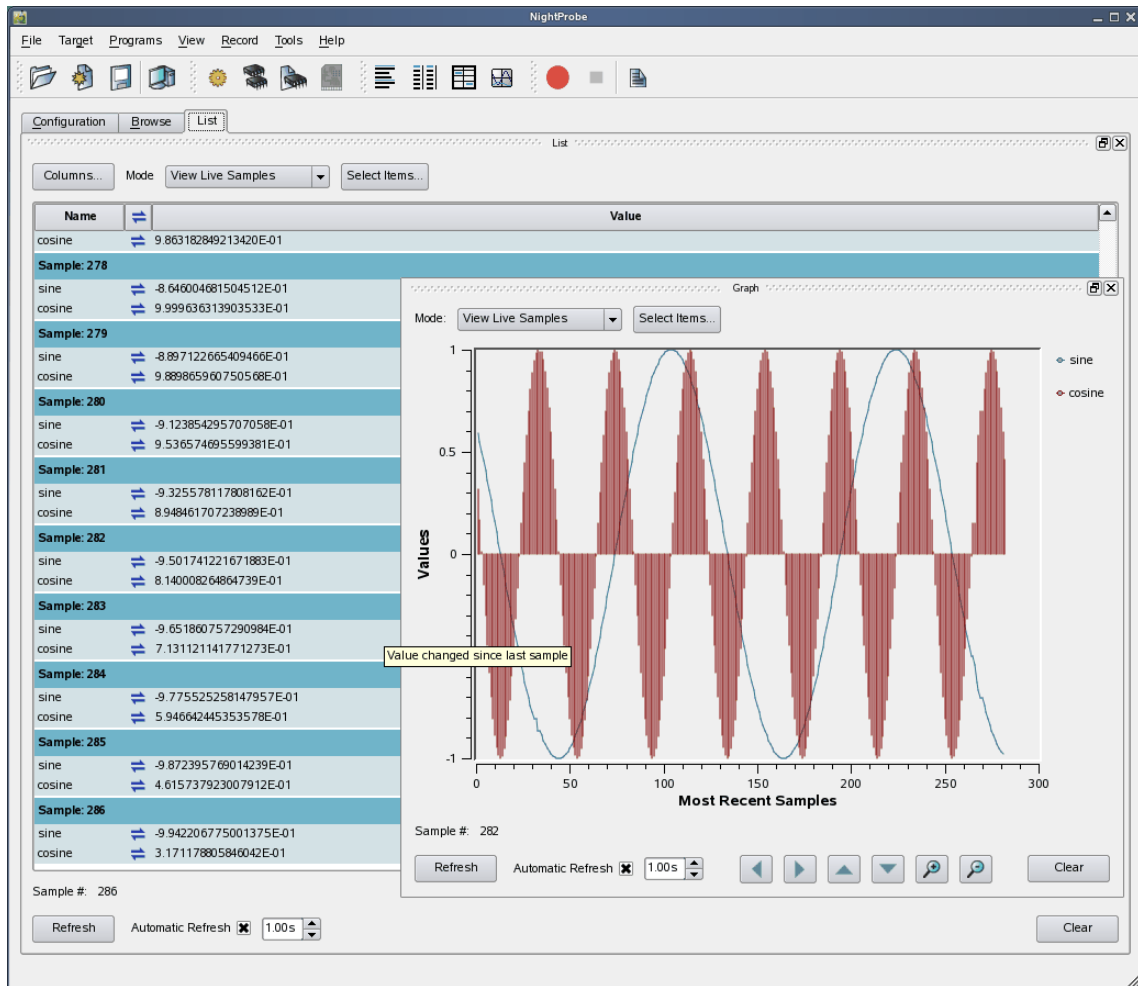


Figure 2-2. Panel Detaches from Page

The Graph panel detaches from the page and becomes free floating.

If moved outside the boundaries of the main window and released, the panel will remain detached from the main window. However, even in detached mode, if the main window is iconified, the detached panel will be iconified with it. For this reason, detached panels are not very useful in and of themselves. Detaching is most often useful as part of moving a panel and re-docking it.

To insert a panel into the page at a new location, drag the panel using the left mouse button on its title bar and move it until it approaches a boundary of the page. The window will respond by creating space indicating where the panel will be inserted.

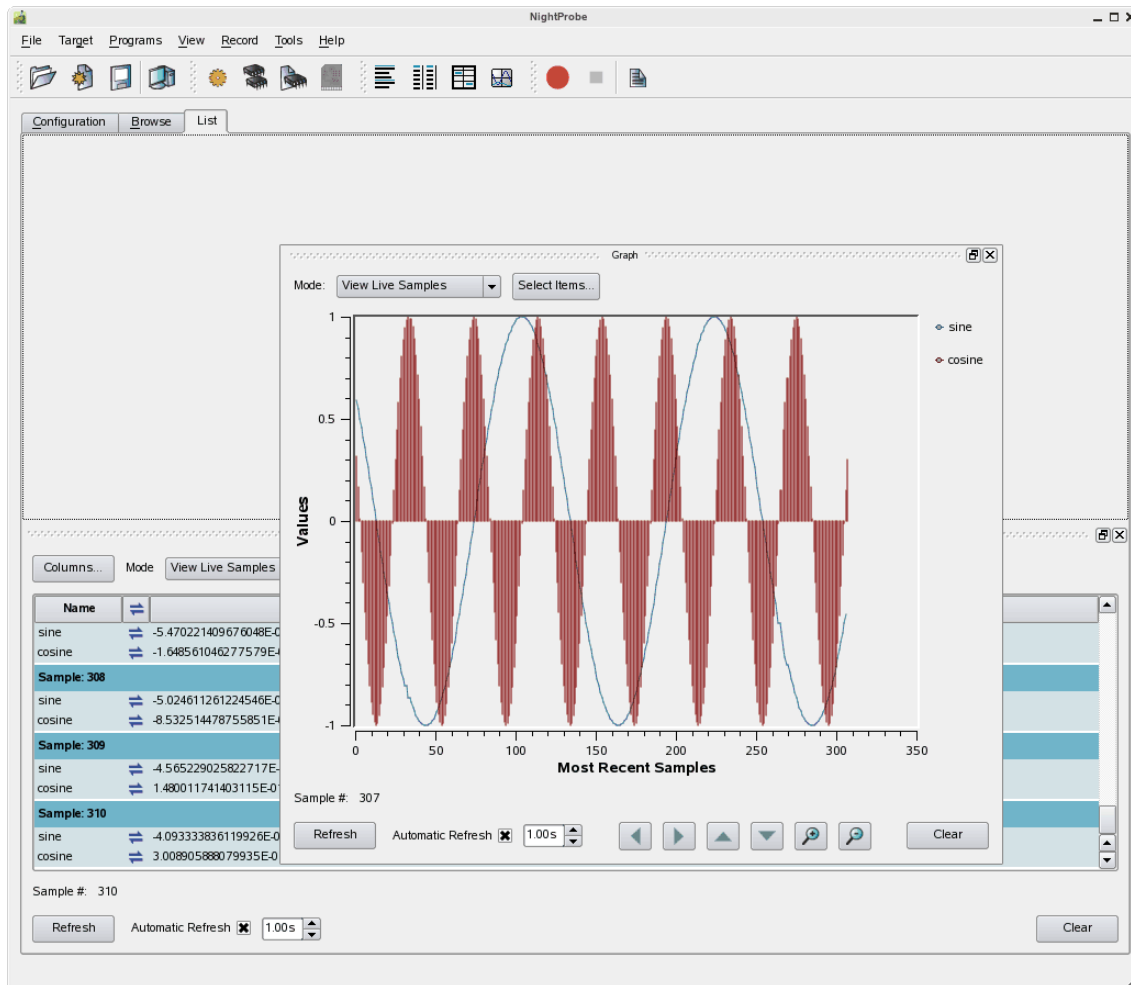


Figure 2-3. Panel Movement in Progress

The figure above shows space being created above the List panel as the Graph panel is dragged towards the upper horizontal boundary of the page.

At this point, releasing the mouse button will cause the Graph panel to be inserted into the page, consuming the recently created space.

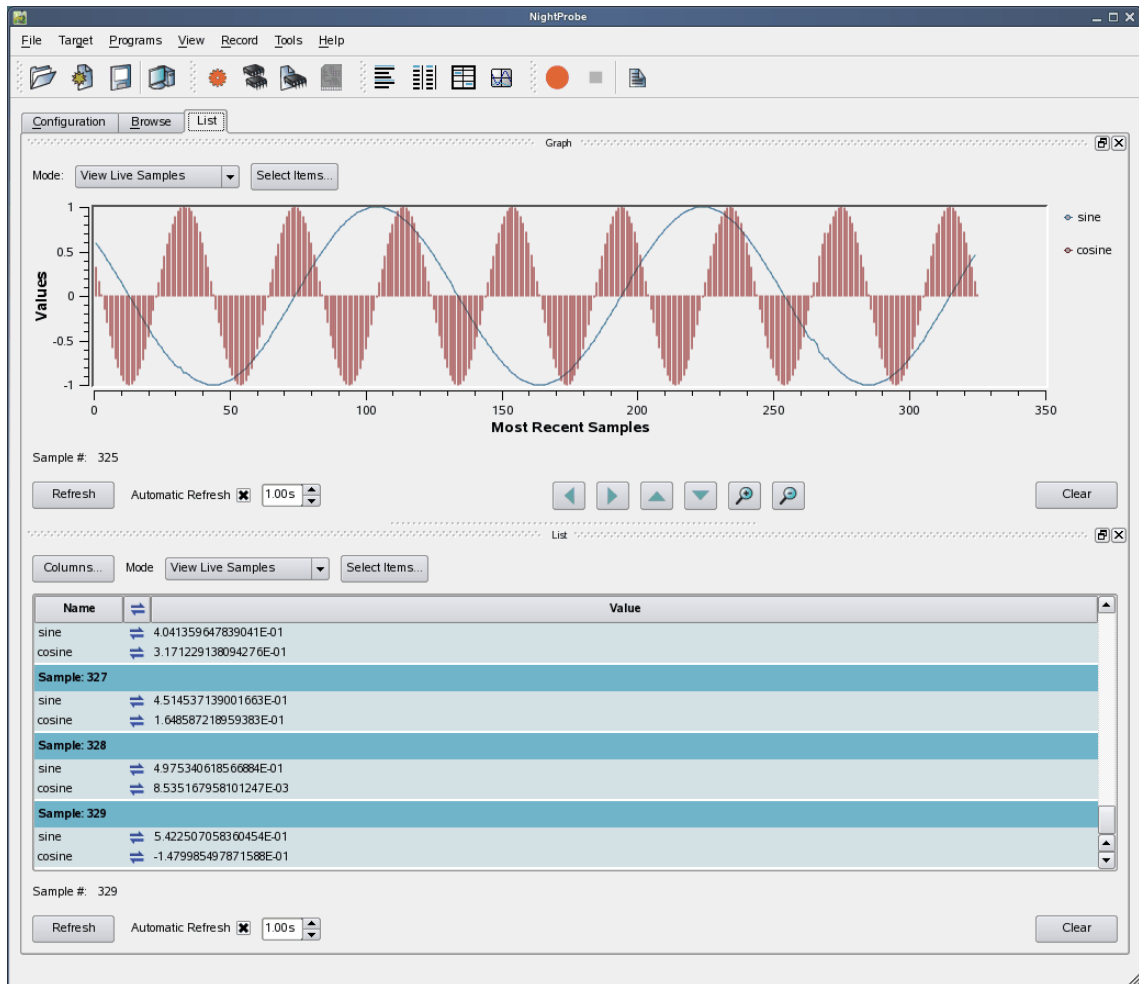


Figure 2-4. Graph Panel on Top of List Panel

IMPORTANT

When attempting to move panels inside of a page, if an empty space does not appear where you desire it, try increasing the size of the main window, decreasing the size of the undocked panel, and moving an alternative edge of the undocked panel near where you want to place it.

By default, the tools usually add panels to the right-hand side of the page when a new panel is created.

In the following figure, a Table panel has been added to the right-hand side of the Graph and List panels.

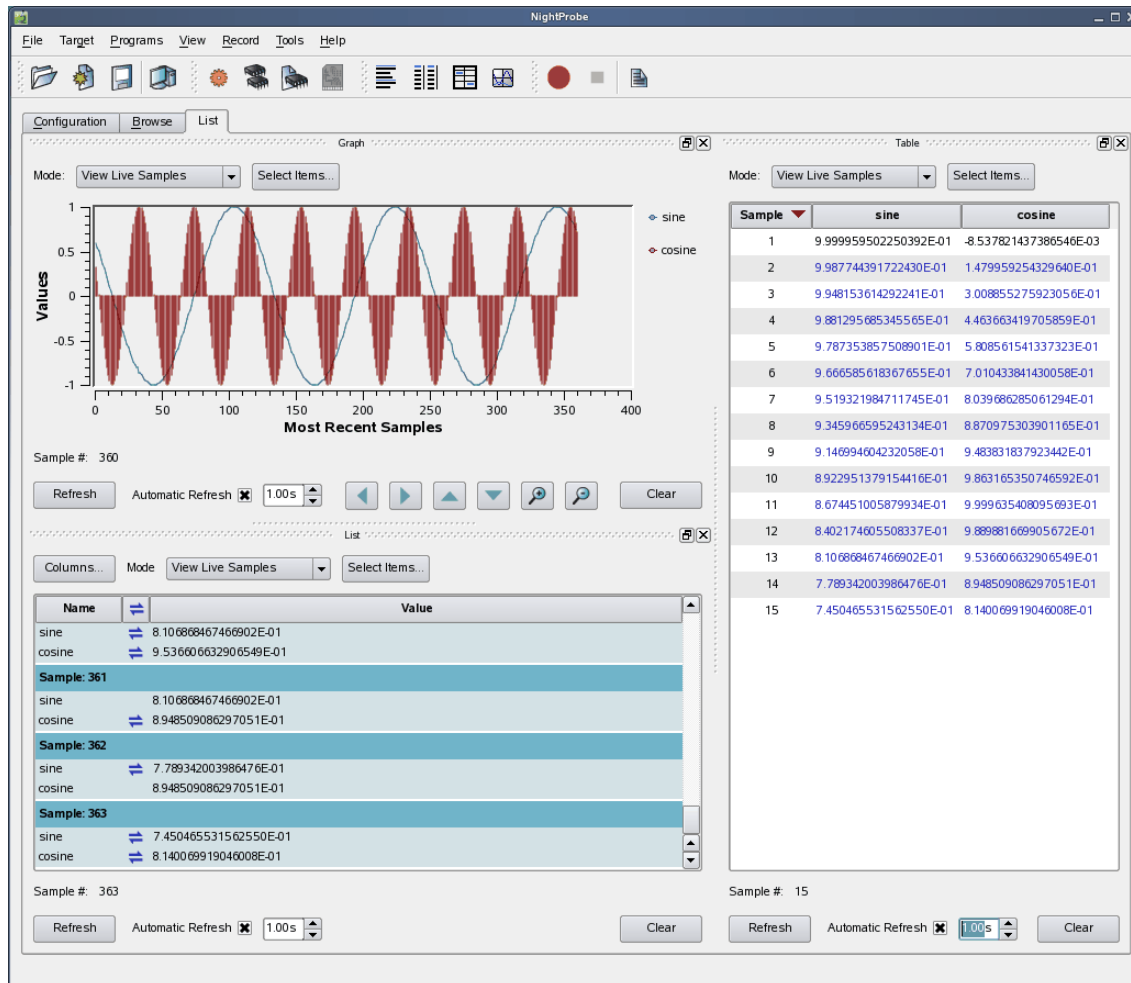


Figure 2-5. Table View added to Page

Panels can be resized by left-clicking on the separator between the panels and dragging it to the desired size.

Tabbed Panels

Another feature of the graphical user interface is the use of tabbed panels. Tabbed panels allow you to maximize your GUI real estate by placing two or more panels in the same location by stacking them on top of each other. You can then raise a panel to the top by clicking on its tab.

To create a tabbed panel, move a panel to the lower horizontal edge of another panel until a tab appears at the bottom of the panel still connected to the page.

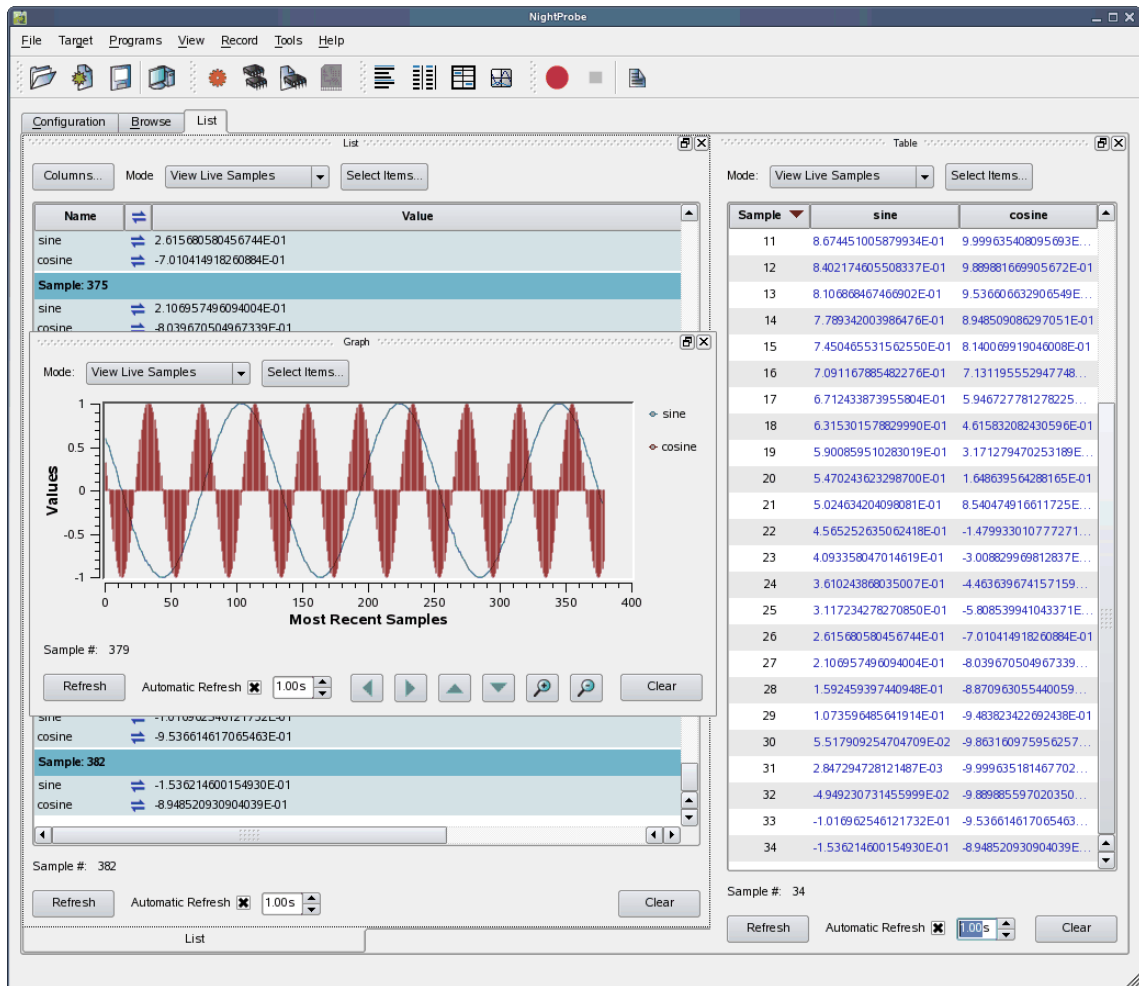


Figure 2-6. Panel in Motion Creating Tab

In the figure above, the Graph panel is being dragged from its original position on top of the List panel towards the bottom of the List panel. A tab appears on the List panel indicating that if the mouse button is released, the Graph and List panels will be tabbed and therefore consume the same area of the page.

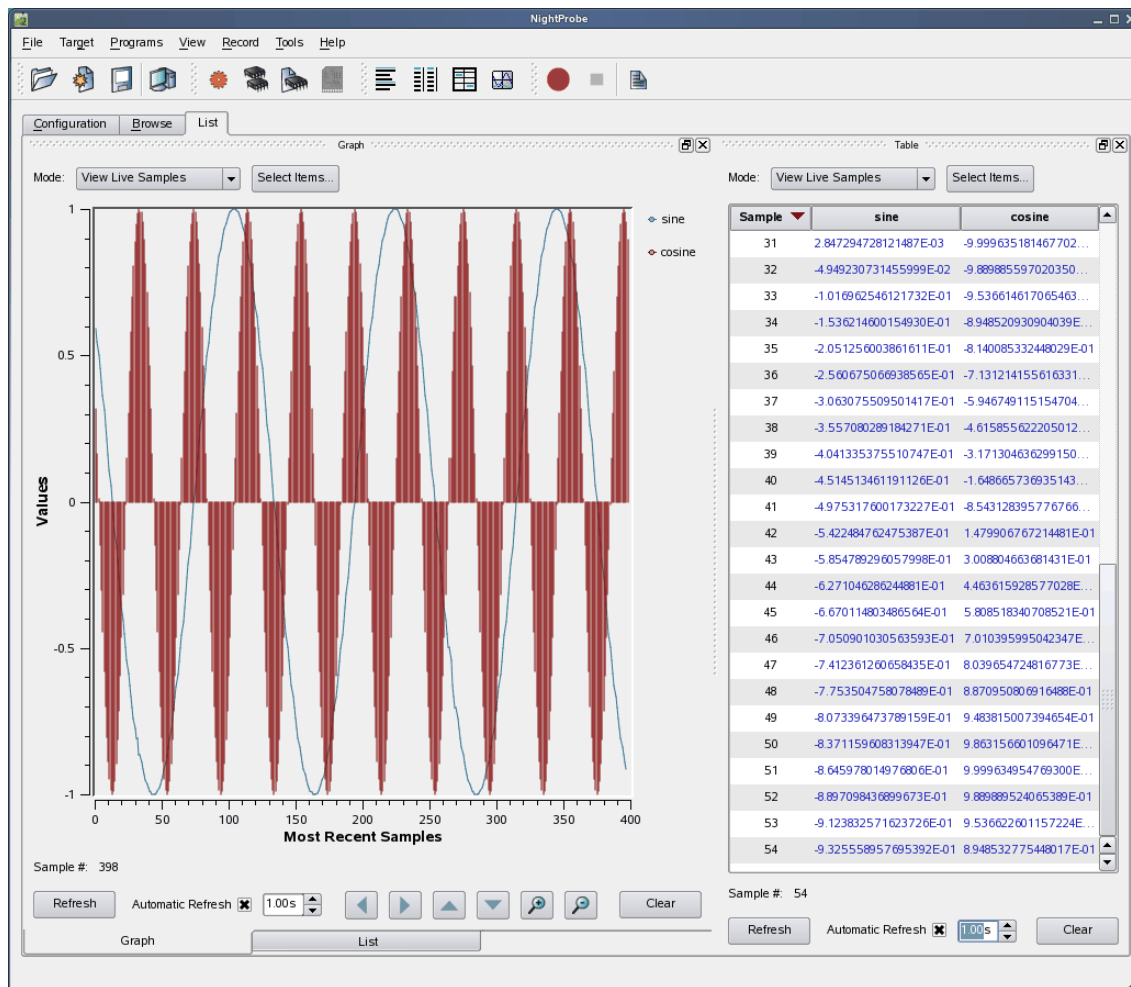


Figure 2-7. Graph Panel Created a New Tab

IMPORTANT

To move a panel above another panel, move the desired panel to the top boundary of the other panel. If you move a panel to the bottom boundary of another panel, it will become a tabbed panel instead.

Context Menus

The NightStar tools rely heavily on use of context menus.

Context menus are menus that appear when you use the mouse to right-click when the mouse cursor is positioned over an area or item of interest. They are called context menus because their content is often dependent on the context of the area in which you right-click, or the item which you right-click upon.

When in doubt, try a right-click operation and see if a menu becomes available.

Tutorial Screen Shots

In order to show full screen shots in this tutorial, the size of each main window has often been left to its default setting. Displaying larger windows would require compression in order to fit the image within the available space of a printed page; such compression obscures detail.

However, as a user of the tutorial, increasing the size of the main window is highly recommended so you can see more data without having to scroll the contents of individual panels.

In many cases within this tutorial, portions of expanded areas of the screen have been extracted from the main window and are included as stand-alone screen shots. These correspond to panels within the main window of each tool.

Using NightView

NightView is a graphical source-level debugging and monitoring tool specifically designed for time-critical applications. NightView can monitor, debug, and patch multiple processes running on multiple processors with minimal intrusion.

NightView supports all the features you find in standard debuggers, including:

- breakpoints
- single-stepping through statements
- single-stepping over function calls
- full symbolic expression analysis
- conditions and ignore counts for breakpoints
- assembly and symbolic debugging

In addition to standard debugging capabilities, NightView provides the following features:

- application-speed eventpoint conditions
- the ability to patch code to change program flow or modify variables or registers during program execution
- hot patch and eventpoint control
- synchronous data monitoring
- loadable modules
- support of multi-threaded programs
- debugging of multiple processes
- dynamic memory debugging
- traversing linked lists
- searching segments of memory
- branch history
- per-thread debugging (protected, single thread, multiple threads)
- smart printing (customizing the way opaque or complex structures are displayed)
- timing functions or regions of code

Invoking NightView

- Execute NightView by issuing the following command:

nview &

at the command prompt or by double-clicking on the desktop icon.

NOTE

If you do not have desktop icons for the NightStar tools, run
/usr/lib/NightStar/bin/install_icons.

When we launch NightView, the NightView main window is presented.

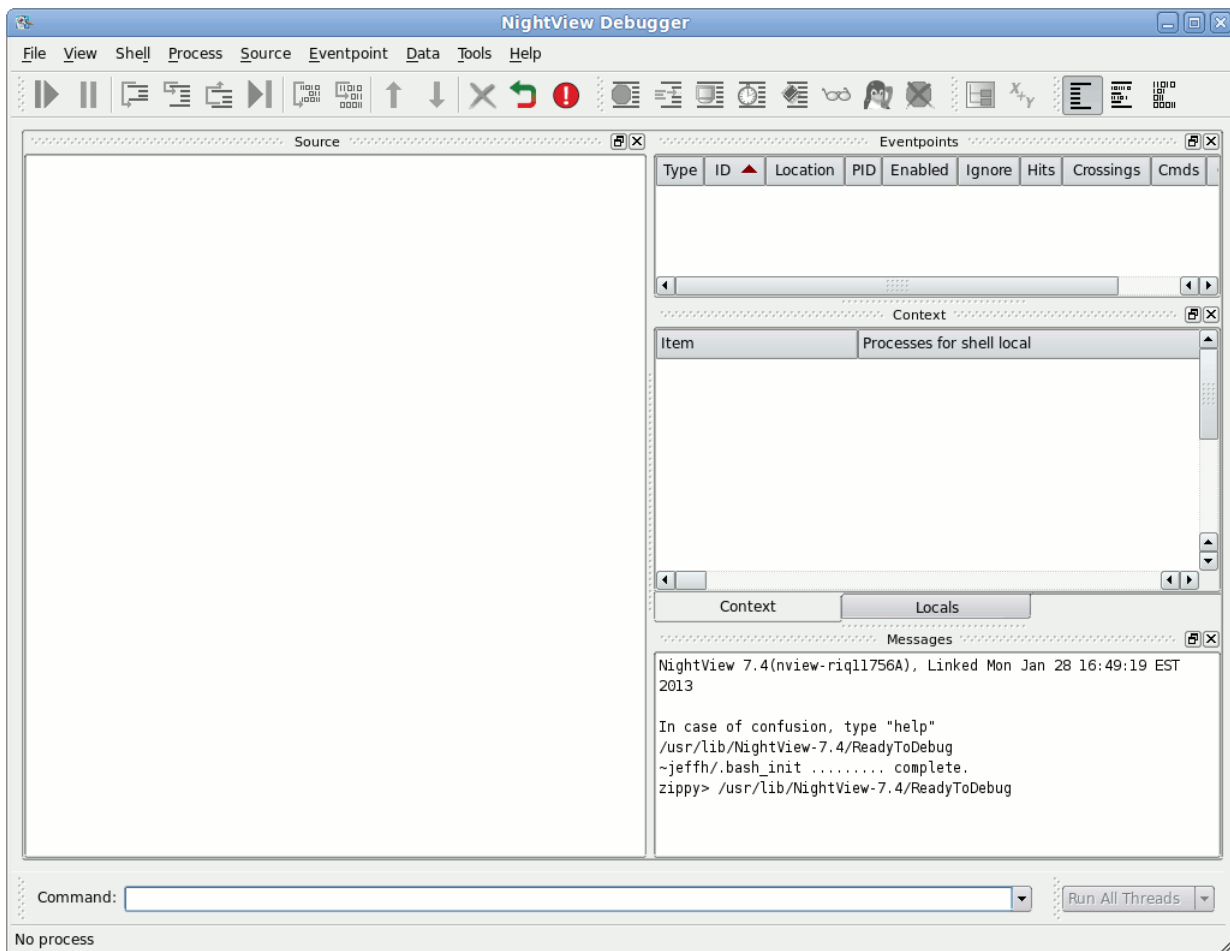


Figure 3-1. NightView Main Window

NOTE

If this is the first time you've invoked NightView since installing NightStar or upgrading to the latest version, you may see a welcome screen. You can disable the welcome screen for subsequent invocations using the checkbox in the lower left corner of that screen. If the screen appears, press the **NightView** button to proceed.

If you have used NightView before and have customized its configuration, you may want to load the default configuration to avoid confusion during the tutorial. To do this, go to the **File** menu and select **Load System Default Configuration**.

In our example, we'll be debugging a single application with multiple threads.

NOTE

If you have not yet created the **app** program, see "Building the Program" on page 1-4.

- Invoke our tutorial application in the NightView main window by selecting the **Process** menu, then **Run...** then in the **Run on local** dialog enter
./app
- Press **OK** to close the dialog and run the program.

Any output generated by the program will appear in the **Messages** panel in green.

When the **app** program is loaded for execution, NightView stops the program immediately after dynamic linking has finished its initial phase (but before any static constructor code might be executed) and displays the source for the main function in the source panel.

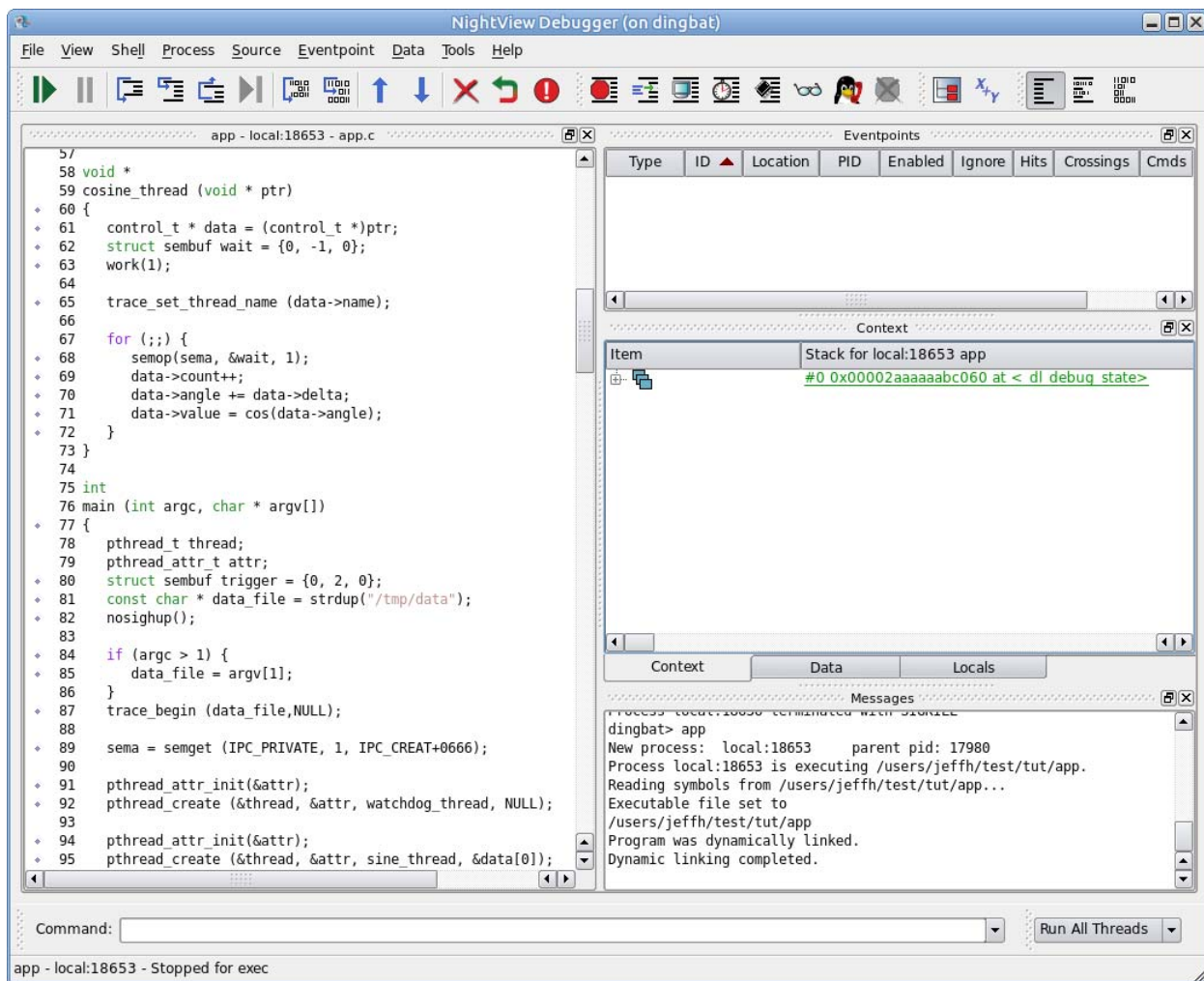


Figure 3-2. app Program Loaded

NightView supports debugging multiple processes as well as single and multi-threaded processes. In this tutorial, you will be debugging a single process with multiple threads.


Debugging Multiple Threads

Our application consists of the main thread and four additional ones created by the main thread.

- Allow the process to run by entering the following command in the Command panel:

```
resume
```

NOTE

Alternatively, press the Resume icon .

The second thread, the `watchdog_thread`, is a critical thread and must run unimpeded without missing its 50ms deadline; otherwise it will fail.

NOTE

If the watchdog thread is printing overrun messages continually in the message panel and you are not a privileged user (see “Setting Up User Privileges” on page 1-2) or the root user, this is because it cannot set its scheduling class to `SCHED_FIFO`. Consider running as a privileged user or as root.

We want to mark this thread as protected so NightView will not stop the thread when other threads stop.

- Right-click on the `watchdog_thread` in the Context panel and click the Protected Thread checkbox.

The context panel responds by showing the word protected on the `watchdog_thread` line. The protected attribute is a special NightView-maintained thread attribute.

The values of all NightView thread attributes are displayed in the thread list. Thread attributes can be useful to you. See **Thread Tags** in the **Concepts** chapter of the *NightView User's Guide* for more information.

In the next few sections, we will use breakpoints and other techniques that cause the process to stop.

- Enter the following command:

```
list sine_thread
```

- Set a breakpoint on line 52 by issuing the following command:

```
b 52
```

The process will run until one of the threads reaches the breakpoint on line 52, but the watchdog thread will not stop because it is protected -- all non-protected threads are stopped by NightView.

- Click on the **Context** tab to raise the **Context** panel.
- Click on the plus sign in the **Item** column to expand the thread displayed in green.
- Click on the plus sign to expand the first item in the walkback list that appeared as a result of the last step.

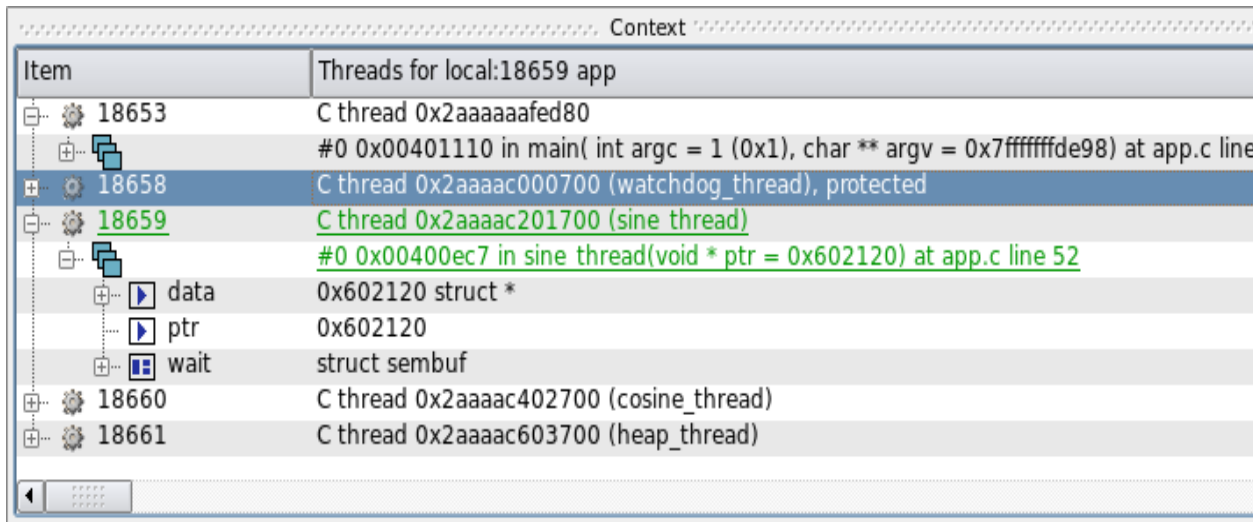


Figure 3-3. Context Panel With Stack Frames Expanded

Expanding an individual **Frame** in the walkback list shows all local variables for that frame. You can further expand composite variables and pointers to composite variables.

The source shown in the **Source** panel is that associated with the program counter of the thread which caused the process to stop. You can tell which thread you are stopped in by looking for the name of the thread's start routine in parenthesis. NightView automatically assigns names to threads based on the start routine which was passed to **pthread_create(2)**. Additionally, you can set the name of a thread inside NightView using the **set-thread-name** command.

You can switch to the context of other threads by clicking on the thread of interest. When you click on a thread, the source displayed in the NightView main window changes to the location where that thread is stopped.

Alternatively, you can use the **select-context** command and specify the thread name as shown in the Threads display in the **Context** panel or from the output of the **info threads** command:

```
info threads /v
select-context name="cosine_thread"
```

When thread names are not unique across threads, you can use the thread ID which is always unique. A thread ID is a hexadecimal number representing the thread -- it is

assigned by the threads library upon thread creation. The thread ID immediately follows the words “C thread” on each thread item in the **Context** panel.

- Switch to the context of the thread executing `sine_thread()` by clicking on it.

NightView provides a **Run Mode** which specifies how threads are resumed and stopped. By default, the **Run Mode** is **Run All Threads**. Thus when the application hits a breakpoint or is otherwise stopped by NightView, all non-protected threads in the application will stop. Similarly, when NightView resumes execution of a thread, all threads will resume execution.

If you change the **Run Mode** to **Run One Thread**, then it is the only one that runs when you resume a thread. All other stopped threads remain stopped.

On one of the toolbars, you will see an option list which represents the current run mode. By default, this item is the right-most widget in the toolbar of the main window..

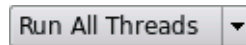




Figure 3-4. Run Mode Selector

- Change the mode to **Run One Thread** by clicking on the list and selecting that mode.
- Click the **Next** icon  several times until the green PC icon  stops on line 51, the call to `semop()`.

NOTE

If the PC icon arrow is gray instead of green, you are stopped somewhere without debug information and the `app.c` file will be displayed. If the system has debug information for the standard libraries, the arrow will be green and you will probably see the source file for the `semop` routine.

- Now click the **Next** icon one more time. Notice that the **Next** operation does not complete.

This is because we were only allowing a single thread to execute, and the thread is blocked in the `semop()` call, waiting for another thread to unblock it (the main thread).

- Press the **Interrupt** icon  to cancel the **Next** operation.

NOTE

Some versions of glibc on some distributions may be missing proper walkback information for the `semop(2)` routine, which is where the thread is stopped. In this case, the **walkback** and **interest** instructions below will not react as described below for this specific example.

Also, some systems may have debug versions of glibc installed, in which case NightView may show you source code inside `semop()`, or `semtimedop()`, or routines it calls. Regardless, you will likely be presented with a gray or green PC icon (triangular arrow), as described below, unless you are stopped at the lowest level, a system call.

A gray PC icon before the line number in the source panel represents the fact that we are positioned at a stack frame which is not the topmost stack frame and that the current frame is executing a subprogram call. A green PC icon means we are at the topmost stack frame.

By default, NightView hides uninteresting frames. If you desire to see all frames for all routines, even those that have no debug information, you can set your *interest threshold* to the keyword `min`:

```
interest threshold min
```

Once that command is issued, the walkback information shows all frames and you can position to any frame and debug at the assembly level if desired.

- Reset the **interest threshold** to zero via the following command:

```
interest threshold 0
```

- Change the Run Mode Selector to Run All Threads.


Rerunning the Process

Oftentimes while debugging a program, you want to rerun the program.

NightView automatically remembers all Eventpoint settings and reapplies them when the process runs again. However, it does not re-apply a thread's protected status.

But NightView has another method that can be used to set the protected attribute on the fly. This provides some automation in reapplying the protected status (see below).

First we will rerun the process.

- Re-initiate the program by pressing the ReRun icon  in the Process toolbar.

NOTE

Alternatively, you can issue the following command directly from the Command panel to initiate the process:

```
rerun
```

- Resume the process with the following command:

```
resume
```

Notice that our breakpoint was automatically reapplied and we hit the breakpoint, and all threads stopped, including the `watchdog_thread()`, because it is no longer protected.

We can apply a patchpoint to ensure its protected status is always reapplied when the thread starts execution.

- Scroll down to line 287 and click on the line.
- From the Eventpoint menu select Set Patchpoint.
- Click the Enable, disable after next hit radio button.
- Click the Set thread local tag values radio button.
- In the Thread Tags text box enter the text: `protected=1`
- Press OK.

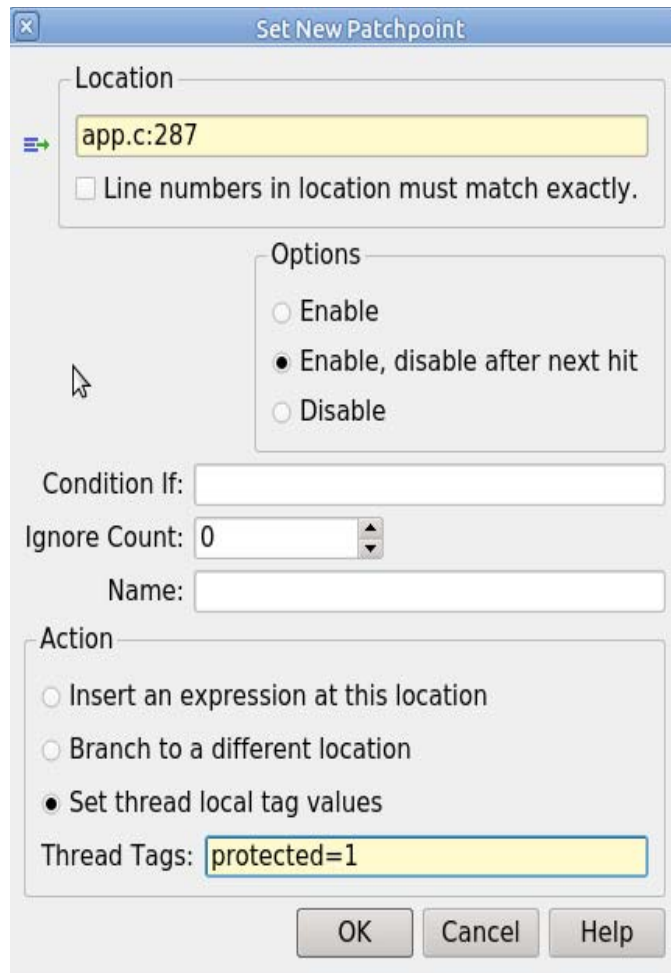


Figure 3-5. Set Patchpoint dialog with changes

Now as soon as the thread is created and executes line 287, it will set itself as protected. Since the protected attribute is set as part of a patchpoint, it will be retained on a **rerun** command.

- Rerun and resume the process again:

```
rerun  
resume
```

The process hits the breakpoint on line 52, but the watchdog did not stop and has the tag (protected) in the context window.

- Delete the breakpoint on line 52 by right-clicking on that breakpoint in the **Eventpoints** panel and selecting **Delete** or by issuing the following command:

```
clear app.c:52
```

- Resume execution of the process.

NOTE

A significant feature of NightView is the ability to execute most debugging operations without having to stop execution of the process.

The debugging operations in the several sections of this tutorial are done without stopping the process!

Traversing Linked Lists

NightView's data display panels allow you to view variables, indirect through pointers, and expand or collapse levels of detail. Variables are presented in a tree to facilitate viewing.

NightView provides two features which make viewing complex data structures easier: linked lists and filtering.

Our application has created a list of structures which are linked via a member of each structure. The variable **head** represents the start of this linked list.

For simplicity, we will remove the existing **Data** panel (if displayed) before proceeding with this section. By default you should only see the **Context** and **Locals** panels. If you do see a **Data** panel, remove it as described below:

- Raise the existing **Data** panel and then close it by clicking the close button in the upper-right of the panel's control area.
- Add the variable **head** to a new data panel by typing the following command:

```
data head
```

A new data panel now appears and contains the pointer variable **head**.

- Expand the pointer variable and the link pointer member of it, and several of its children.

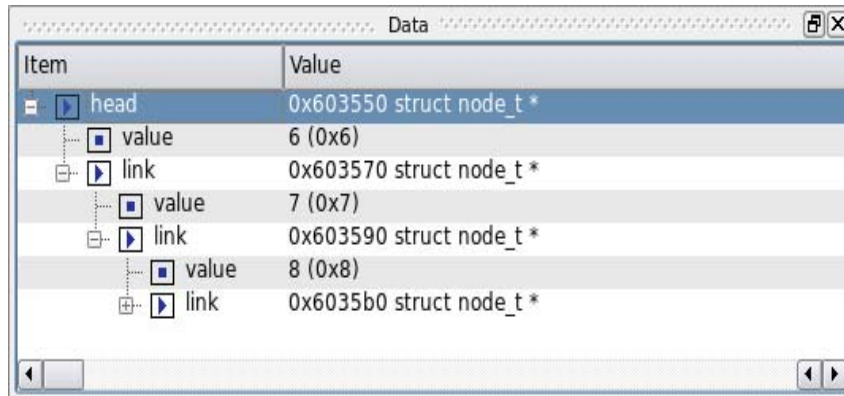


Figure 3-6. Pointer to Linked List Expanded

As shown in the figure above, each node in the linked list is nested under the previous node in the list. While this is a fine representation, it becomes cumbersome once you display more than just a few nodes.

As an alternative, you can tell NightView that the pointer is a member of a linked list.

- Right-click on the head variable and select **Treat As Pointer To Linked List...**

A small dialog is presented which allows you to specify the member of the structure which defines the next element in the list.

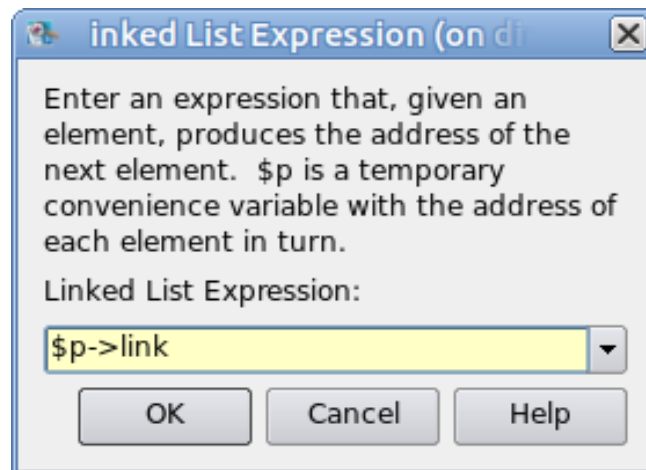


Figure 3-7. Dialog Selecting Linked List Component

NightView automatically populates a drop-down list with all members which have types appropriate for indicating a link in a list. In our case, it has correctly chosen the member which identifies the next node in the list.

- Press OK.

The head variable in the data panel is now displayed using an alternative method.

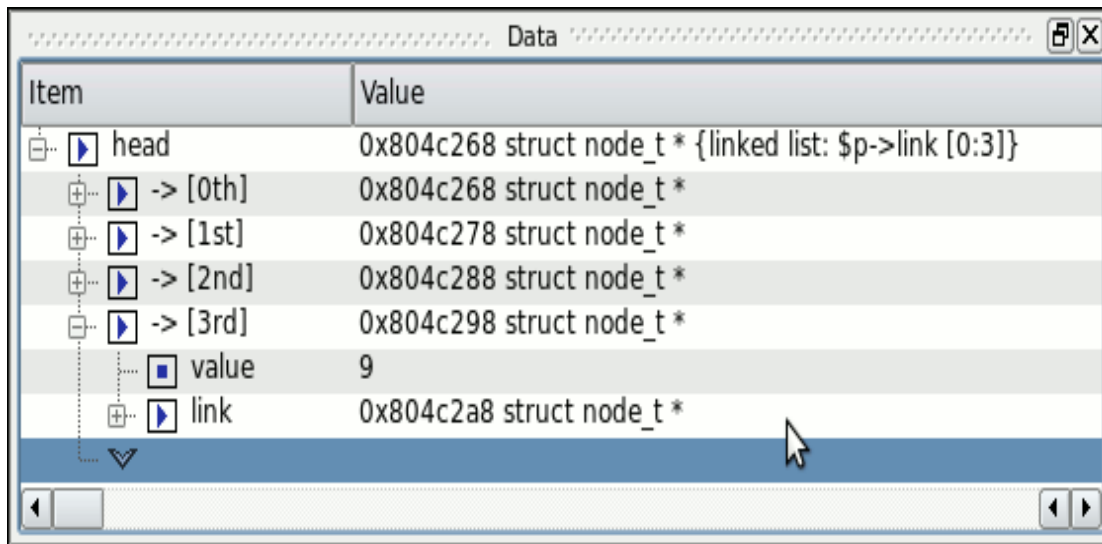


Figure 3-8. Pointer Variable Displayed As Linked List

In the figure above, the various nodes in the linked list are displayed at the same level and are numbered, starting from 0.

- Click on the guard symbol (blue chevron) several times until the “3rd” node is shown and then expand it to match the figure above.

NightView allows you to expand the list as long as the member that you selected above that defines the next item in the list is not NULL. You can also use the context-menu to tell NightView how many nodes in the list to display (as opposed to continuing to click on the guard symbol to extend the list).

Often when viewing a linked list you may want to identify a particular node in the list. We will use NightView’s filtering capability to do this.

- Right-click on the head variable and select Filter Elements with a Condition...

The following dialog appears which allows you to type in an expression which defines the nodes in the list to be shown.

Enter a condition expression. Only the elements that match the condition will be shown. Some temporary convenience variables are set as the condition is evaluated for each element in turn:

\$i has the index of the element. Example: `my_array[$i] < 5`
 \$p has the address of the element. Example: `*$p < 5`
 \$v refers to the element. Example: `$v < 5`

Clear the text field to remove the filter.

Condition Filter Expression:

`$p->value % 7 == 0` Clear

When looking for each filtered element, how many of the underlying elements should the filter check?

Search Limit: 1024

OK Cancel Help

Figure 3-9. Filter Dialog

The expression can include several special built-in variables which aid you in specifying the filter. The text in the dialog explains these variables: \$i, \$p, and \$v.

- Type the following text into the Condition Filter Expression field, as shown in the figure above, and then press OK.

`$p->value % 7 == 0`

We have told NightView to only show us nodes in the list whose member value is a multiple of seven.

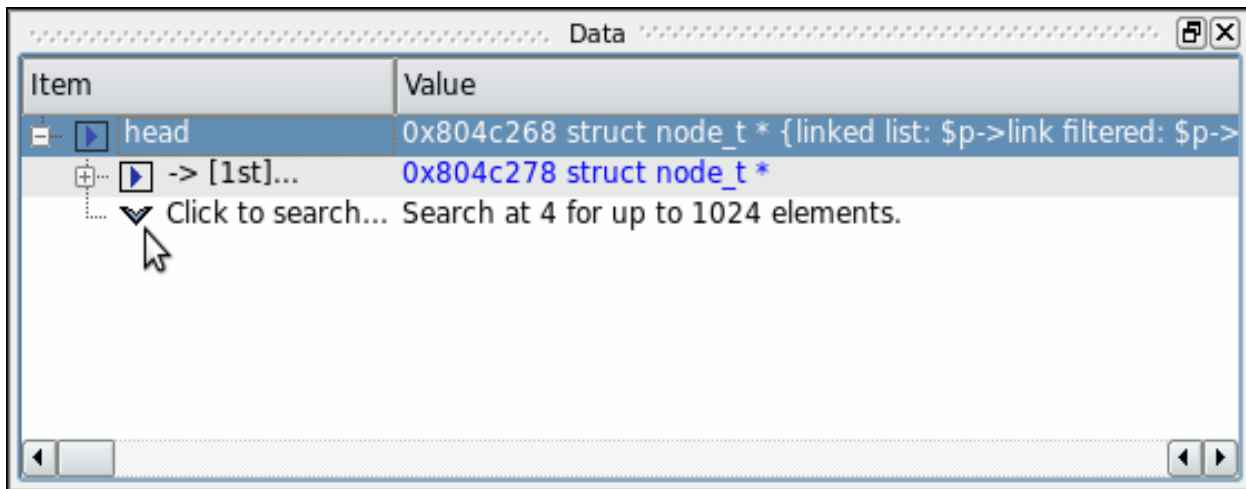



Figure 3-10. Filtered Linked List

Initially, the first node in the list matching the filter condition is shown -- it is node # 1, the second node in the list (node numbering starts at 0).

- Expand the filtered list by clicking the guard symbol  two times, and then expand all three filtered elements to match the figure below:

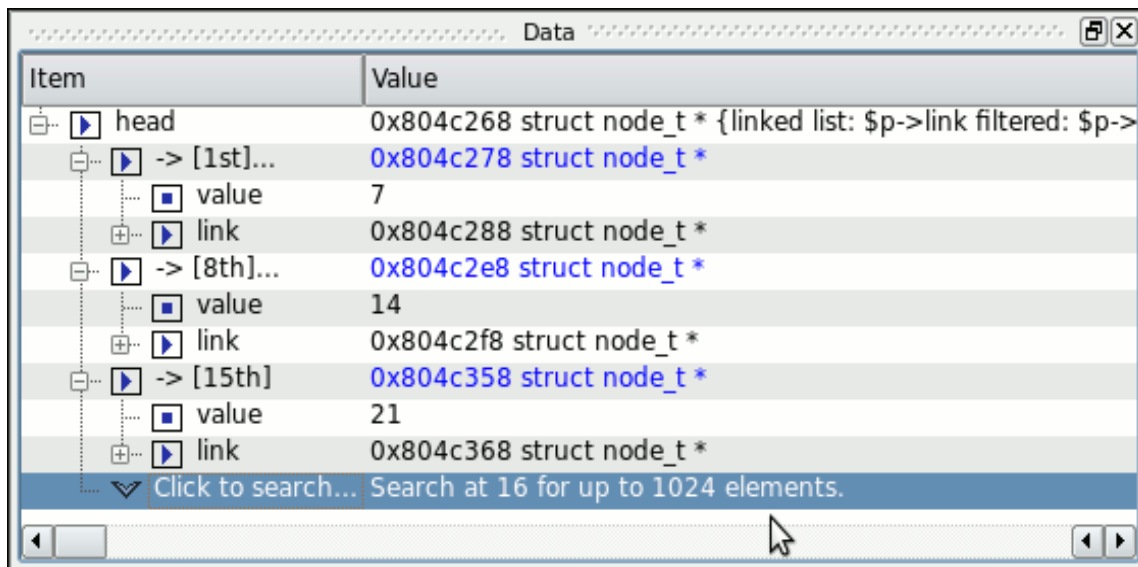


Figure 3-11. Filtered Linked List Expanded

See that all nodes shown in the list have a `value` member which is a multiple of seven, which satisfies the filter expression we specified above.

Notice that an ellipsis follows each node number when the next node in the list is not consecutive, indicating that there are gaps in the displayed list due to filtering. The description field of the head of the linked list also indicates filtering is active.

You can use NightView's filtering capability on arrays as well as linked lists. In fact, you can use it to search through memory for a particular value. Just add a pointer value to the data panel, tell NightView to treat it as an array using the context menu, and then apply a filter expression.


Using Monitorpoints

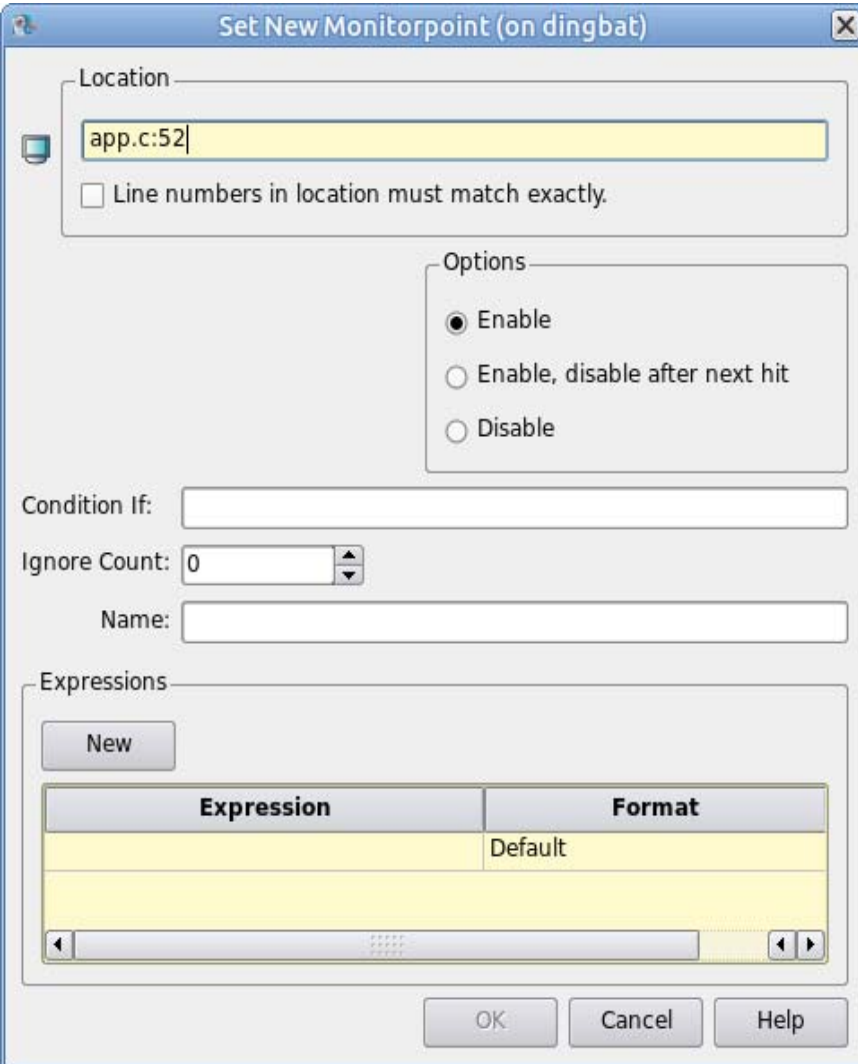
Monitorpoints provide a means of monitoring the values of variables in your program without stopping it. A monitorpoint is code inserted by the debugger at a specified location that will save the value of one or more expressions, which you specify. The saved values are then periodically displayed by NightView in a **Monitor** panel.

Unlike asynchronous sampling, monitorpoints allow you to view data which is synchronized with execution of a particular location in your application.

- Scroll back to line 52 in the source panel, then right-click on line 52 and select **Set eventpoint** from the context menu and select **Set Monitorpoint...** from the sub-menu.

NOTE

Alternatively, you could go to the **Eventpoint** menu and select the **Set Monitorpoint...** option, or click the **Set Monitorpoint** icon  from the toolbar to launch the **Set New Monitorpoint** dialog.



The dialog box is titled "Set New Monitorpoint (on dingbat)". It contains the following fields and options:

- Location:** A text field containing "app.c:52". Below it is a checkbox labeled "Line numbers in location must match exactly." which is unchecked.
- Options:** A group box containing three radio buttons: "Enable" (selected), "Enable, disable after next hit", and "Disable".
- Condition If:** An empty text field.
- Ignore Count:** A spinner box set to "0".
- Name:** An empty text field.
- Expressions:** A section with a "New" button and a table.

Expression	Format
	Default

At the bottom are "OK", "Cancel", and "Help" buttons.

Figure 3-12. Monitorpoint Dialog

- Ensure that the Location text field displays `app.c:52`, correcting it if need be.
- In the text field below the Expression column head, enter the following but do not press the Enter key yet:
`data->count`
- Scroll to the right then enter the following in the Label column:
`sine count`
- While still positioned in the cell under the Label column, press the Tab key. This positions you to the next row and allows you to continue adding expressions.

NOTE

If you have already left the cell and only one row is shown, press the **New** button.

- In the second row under the **Expression** column, type the following:

data->value

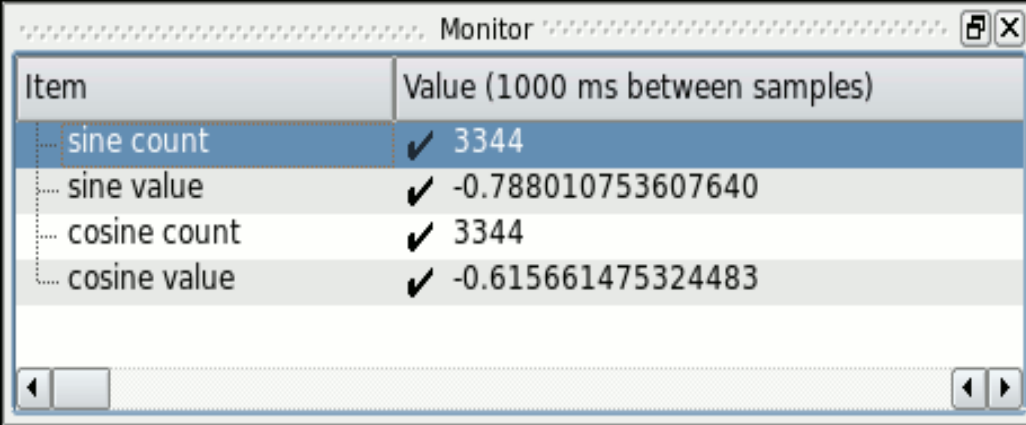
- Set its label value in the **Label** column, by typing the following:

sine value

- Press the OK button in the **Set New Monitorpoint** dialog.

A **Monitor** panel is created containing an entry for the expressions entered above.

- Likewise, set a monitorpoint on line 69 with the same expressions as in the previous monitorpoint, substituting **cosine** for **sine** in the **Label** fields.



Item	Value (1000 ms between samples)
sine count	✓ 3344
sine value	✓ -0.788010753607640
cosine count	✓ 3344
cosine value	✓ -0.615661475324483

Figure 3-13. NightView Monitor Panel

At this point, the data values in the **Monitor** panel change.

The values are sampled whenever line 52 or 69 are executed, respectively. NightView displays the latest set of values in the **Monitor** panel at a user-selectable rate.

Using Eventpoint Conditions and Ignore Counts

All eventpoints in NightView have optional *condition* and *ignore* attributes.

A *condition* is a user-supplied boolean expression of arbitrary complexity which is evaluated before the eventpoint is executed. Conditions can involve function calls in the user application.

Similarly, the *ignore* attribute is a count of the number of times to ignore an eventpoint before actually executing it.

Conditions and ignore counts are evaluated by the application itself via patched-in code and, as such, run at full application speed. Ignore-count values are consulted before condition-expression evaluation. Other debuggers evaluate the conditions and ignore counts from within the context of the debugger which takes significant time and can drastically affect the behavior of your program.

- Click the cell in the **Ignore** column of the row of the Eventpoint panel which describes the Monitorpoint for line 52.
- Change the value to 500 and press **Enter**.

The **Monitor** panel now indicates that the values for that monitorpoint have not been sampled by displaying a question mark before the value. When the ignore count reaches zero, the values will start updating again.

Finally, monitorpoints can include complex expressions that aren't just simple variables as long as the final expression resolves to a simple type; e.g., character, integer, floating point, or pointer.

- Enter the following commands in the **Command** panel of the NightView main window; terminate each command with the <enter> key:

```
monitor app.c:105
  p FunctionCall()
end monitor
```

A new item is added to the **Monitor** panel which represents the result of the function call `FunctionCall()` as executed by the user application each timeline 105 is crossed.

Using Patchpoints

Unlike breakpoints and monitorpoints, patchpoints allow you to modify the behavior of your program.

Patchpoints allow you to change program flow or modify variables or machine registers.

First, we will use a patchpoint to branch around some statements in our program.

NOTE

If the source file **app.c** is not displayed, issue the following command:


```
1 app.c:53
```

- Scroll the source file displayed in the NightView main window and right-click on line 53:

```
data->angle += data->delta
```

and select **Set eventpoint** from the context menu and select **Set Patchpoint...** from the sub-menu.

NOTE

Alternatively, you could go to the Eventpoint menu and select the **Set Patchpoint...** option, or click on the Set Patchpoint icon  in the toolbar to launch the Set New Patchpoint dialog.

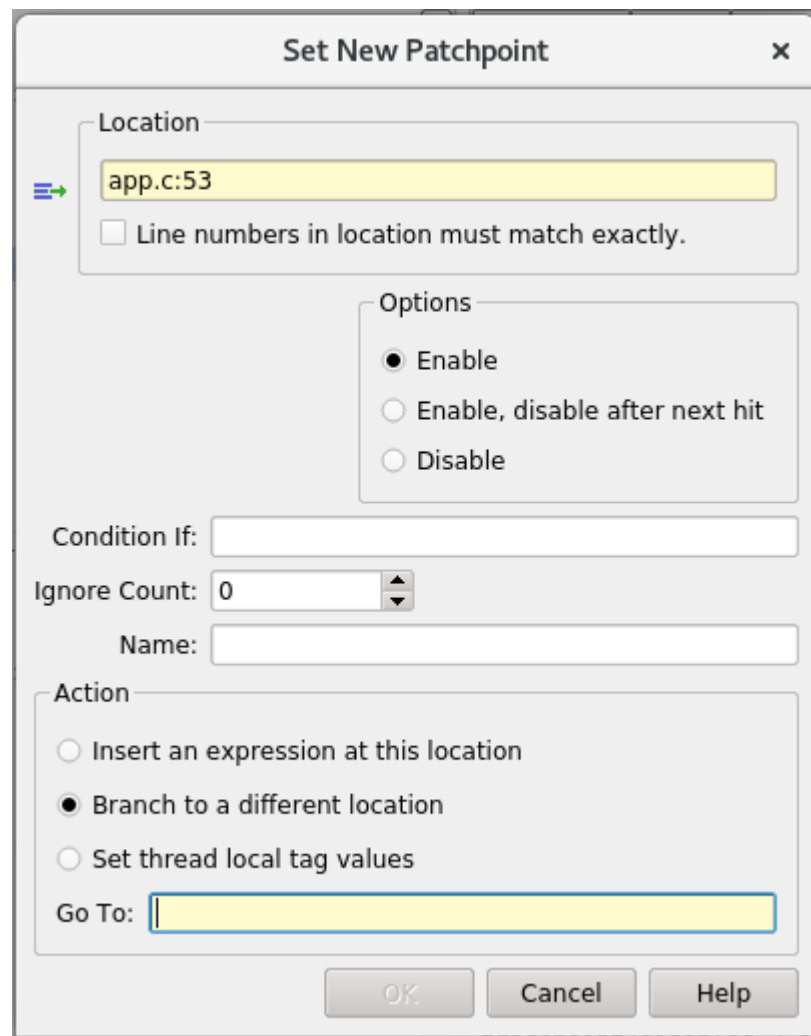


Figure 3-14. Patchpoint Dialog

- In the Location text area, ensure the text indicates `app.c:53`.
- Click on the Branch to a different location radio button in the lower portion of the dialog.
- In the Go To: text area, type:

`app.c:54`

then press the OK button.

This will effectively cause the application to skip execution of line 53, where it updates the angle used in the subsequent `sin()` call.

Note that the `sine` value in the Monitor panel stops changing, yet the associated `sine` count continues to change.

Alternatively, we can use patchpoints to change the value of expressions or variables.

- Type the following command in the **Command** panel of the NightView main window:

```
patch app.c:52 eval data->count -= 2
```

Note that the value of `sine count` is decrementing, because for each iteration, it continues to be incremented by 1, but it is now also decremented by 2.

We can disable the patchpoints without deleting them.

- Select both patchpoints in the **Eventpoints** panel (as indicated in the **Type** column by the word `Patch`), right-click and select **Disable** from the context menu.

The patches are disabled and the values shown in the **Monitor** panel return to their original behavior.

Using Tracepoints

This section covers NightView's integration with NightTrace.

A tracepoint is a specialized eventpoint which essentially patches a call to log a trace event with optional arguments.

Even if the application doesn't already use the NightTrace API, NightView can link in the required components and activate the tracing module. Our application already uses the NightTrace API, so this will not be necessary (see the **set-trace** command in the *NightView User's Guide* for more information on using tracepoints in applications which don't already use the NightTrace API).


Suppose that we were interested in measuring the performance of our cycles in the `sine_thread()` and `cosine_thread()` routines and that we also were interested in logging data values during the cycle.

- Scroll the source file displayed in the NightView main window and right-click on line 53:

```
data->angle += data->delta
```

and select **Set eventpoint** from the context menu and select **Set Tracepoint...** from the sub-menu.

NOTE

Alternatively, you could launch the dialog from the Eventpoint menu and select **Set Tracepoint...**, or click the **Set Tracepoint** icon  on the toolbar to launch the **Set New Tracepoint** dialog.

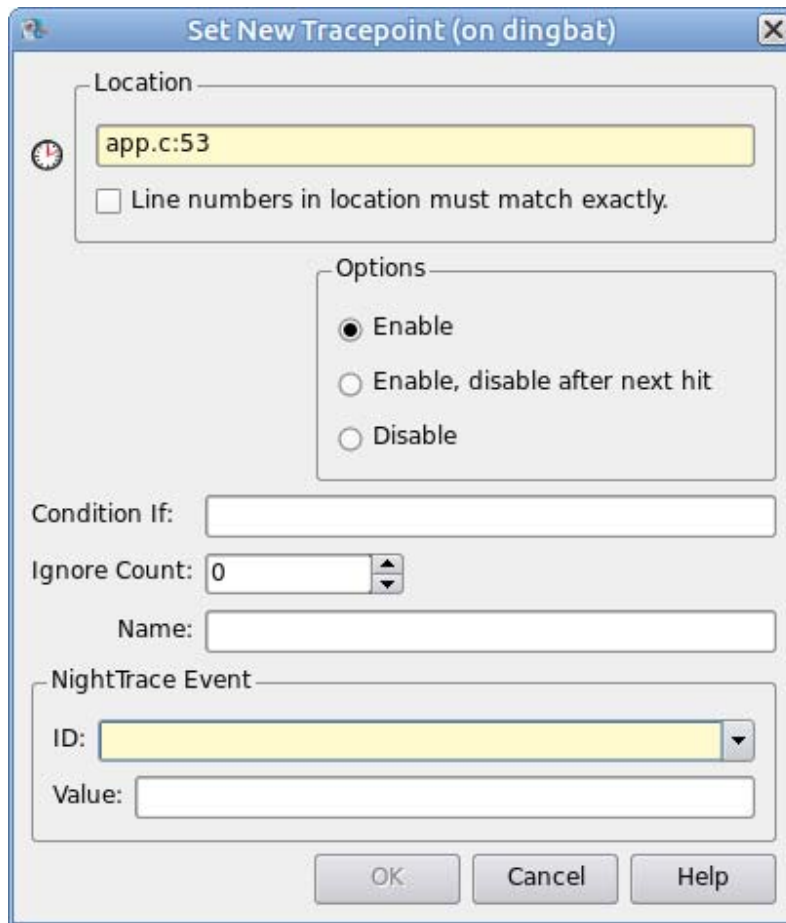


Figure 3-15. Tracepoint Dialog

- In the Location: text field ensure that **app.c:53** is displayed.
- In the NightTrace Event section's ID field, type the following:
1
- Press the OK button.

Similarly, we'll set additional tracepoints but we will also specify a value to be logged with the tracepoint.

- Set a tracepoint on line **app.c:51** and specify an Event ID of **2** and enter the following in the Value text field:
data->value
- Set a tracepoint on line **app.c:68** and specify an Event ID of **3** and enter the following in the Value text field:
data->value

Trace events can now be logged with the NightTrace tool.

NightTrace is a very powerful tool that allows you to mix trace events from different threads and process with kernel trace events. We will cover NightTrace in a subsequent chapter. For simpler needs, you can use NightView's lightweight timing points.

Using Lightweight Timing Points

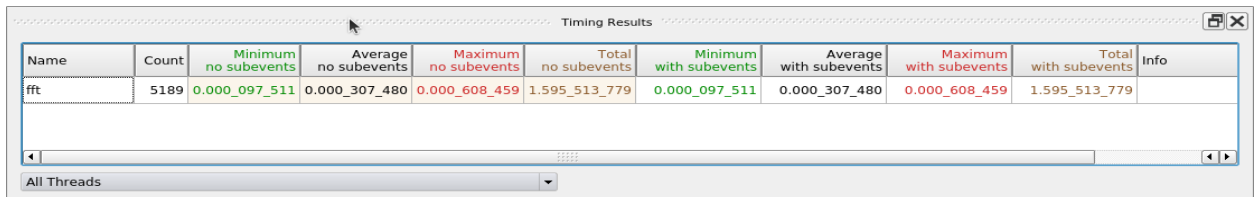
This section provides a mechanism to quickly and easily time regions of code from within the debugger. This is done by identifying start and end locations within executable code by PC address or line number, or simply by function name. These locations are patched with NightView code to record timestamps and calculate the time spent within those regions.

Each timing region supports an arbitrary number of **start** and **end** locations so that regions of code that can be entered or exited via many paths can be supported.

Let's time the **fft** function.

From the **Process** menu, select the **Set Timing for a Function...** option. In the **Set Timing for Function** dialog, enter **fft** in the Function Name area and click OK.

The **fft** function is now being timed. To view the results of these timing points, click the **View Menu** in the Toolbar and select the **New Timing Results Panel** option.



The screenshot shows a window titled "Timing Results" containing a table with the following data:

Name	Count	Minimum no subevents	Average no subevents	Maximum no subevents	Total no subevents	Minimum with subevents	Average with subevents	Maximum with subevents	Total with subevents	Info
fft	5189	0.000_097_511	0.000_307_480	0.000_608_459	1.595_513_779	0.000_097_511	0.000_307_480	0.000_608_459	1.595_513_779	

Below the table is a dropdown menu currently set to "All Threads".

Figure 3-16. Timing Results

Right-click the table and uncheck the **Hold Panel** option. You should see the values in the table updating during execution with the count increasing. The table now refreshes once per second. The table refresh is asynchronous from the execution of the **fft** function; it just displays the latest results.

Now add another Timing Region, this time based on line numbers. List the **fft** function in the command text box (**list fft**).

From the **Process** menu, select the **Set Timing Start or End Location**. Change the line location number from 85 to 86, change the Name to **fftNoCompute**, and click OK. Alternatively, you can right-click on the specified line, so we will do this for setting the end location timer. Scroll down to line 103, if necessary, right click on it to see the context menu and select **Set Timing Start or End Location....** A new dialog box will pop up. Since we've already applied a starting-time point, go ahead and select the **End** radio button under the bottom **Position** area and click OK.

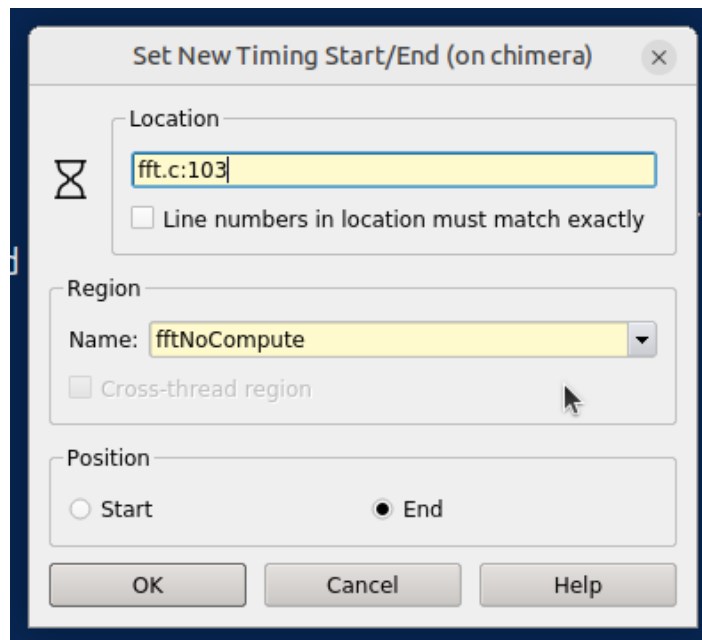


Figure 3-17. End Button

Let's open the Timing Region panel. Click View Menu in the tool bar and select New Timing Regions Panel. This will open a panel on the bottom area and show us the timing points we have inserted in the code. Here we'll have options to delete, clear the region, or update the timing points.

Timing Regions			
Region/Type	PID	Details/Location	
fft	2810447	per-thread	
start	2810447	0x00005555555556290	in fft() at fft.c line 85
end	2810447	0x00005555555556507	in fft() at fft.c line 104
fftNoCompute	2810447	per-thread	
start	2810447	0x000055555555562ab	in fft() at fft.c line 86
end	2810447	0x000055555555564db	in fft() at fft.c line 103

Figure 3-18. Timing Regions

Going back to the Timing Results Panel data: The Timing Results panel provides a table of the currently collected timing results. Each line in the table corresponds to a single timing region. The columns indicate the name of the region and individual timing data for that region. The set of columns is configurable from the context menu and can be rearranged by clicking and dragging columns. For multi-threaded applications, the results can be displayed for each thread separately, or for all threads combined. The patches have support for **subevents**. If two regions are defined such that one region encloses another, the enclosed region is considered a subevent of the enclosing region.

The Count column is the number of times the region was crossed (from start to end). The Minimum no subevents displays the minimum time for any single crossing of the region, excluding time spent in subevents. Average no subevents is the average time spent in the region, excluding time spent in subevents. Maximum no subevents is the maximum time for any single crossing of the region, excluding time spent in subevents.

Total no subevents is the total time spent in the region, excluding time spent in subevents. Minimum with subevents is minimum time for any single crossing of the region, including time spent in subevents. Similarly, the columns that say "with subevents" includes any time spent in nested regions. Info shows errors, warnings and other diagnostics.

There may be an underflow message in the Info sectioned column. This can indicate extraneous end events or missing start events. But it can also mean that events were added to the process during its execution and could reflect that execution was already within a region when the event was added as in this case.

Timing Results									
Name	Count	Minimum no subevents	Average no subevents	Maximum no subevents	Total no subevents	Minimum with subevents	Average with subevents	Maximum with subevents	Total with subevents
ft	182938	0.000_167_181	0.000_615_354	0.001_479_446	112_571_705_903	0.000_233_039	0.000_720_255	0.001_479_446	131_762_002_300
ftNoCompute	99749	0.000_067_897	0.000_192_389	0.000_377_388	19_190_587_716	0.000_067_897	0.000_192_389	0.000_377_388	19_190_587_716
current recursion depth: 1									
All Threads									

Figure 3-19. Timing Results with Subevents

For simplicity, we will now delete the timing regions. In the Timing Region panel, right click each region and select Delete.

Then delete both the Timing Regions panel and the Timing Results panel by clicking on the [x] icon at the upper right of each panel.

Heap Debugging

This section describes NightView's heap debugging features, which are quite useful.

However, if you wish to move immediately to the NightTrace section, skip to "Ready for NightTrace" on page 3-43. You can always come back to this section in the future.

Debugging dynamic memory problems can be difficult and extremely time-consuming. The word *heap* refers to a collection of allocated and freed memory typically controlled by the `malloc()` and `free()` utilities in the C language, or the `new` operator in C++.

NightView provides the unique ability to monitor and detect memory allocations, frees, and sets of user errors without requiring a non-standard allocator to be compiled or linked into your program.

One advantage of this is that often when you switch to a debugging allocator, the way blocks are allocated and freed changes -- often hiding the very bugs you're trying to find.

NightView offers a variety of settings and debugging levels that are useful in catching common heap-related errors. Some settings will change the behavior of the system allocator, affecting the size of allocated blocks and, ultimately, the address values returned.

Dynamic memory errors are typically detected in one of four ways:

- a check of the entire heap at a specified frequency in terms of the number of heap functions (e.g., `malloc`, `free`, `calloc`, etc.) called
- a check of an individual allocated block when `free` or `realloc` is called
- a check of the entire heap when a **heappoint** is crossed
- a check of the entire heap when a **heapcheck** command is issued

The frequency setting of the **heapdebug** command or **Debug Heap** window controls how often NightView should check for heap errors when a heap function is called. Setting the frequency to 1 causes NightView to check for heap errors on every heap operation.

A **heappoint** causes NightView to check for errors when the execution flows through the heappoint. An unlimited number of heappoints can be inserted into your program.

The check of an individual block when `free` or `realloc` is called is automatic.

All four mechanisms are useful. With the first three mechanisms, the heap error detection is executed at program application speed without context switching to the debugger.

Activating Heap Debugging

A limitation of heap debugging is that it requires that you activate the debugging before any allocations occur in your program and that several heap debugging commands only work if the entire process is stopped. If you attempt to activate the heap debugging features after allocations have already occurred, NightView will inform you of its inability to satisfy your request.

- If the **app** process is not currently being debugged (perhaps because you skipped this section previously), then load the process:

nview ./app

- Otherwise, if the application is still running under NightView, then we must disable the protected attribute of the `watchdog_thread` and prevent it from entering its real-time processing loop (since it will be stopped at various times in this section).
 - Delete the patchpoint on line 287 (right-click on the relevant event point in the **Event Points** panel and select **Delete**).
 - Rerun the process:

rerun

- Enter the following command to cause the `watchdog_thread` to essentially become idle:

patch 293 goto 309

- From the **Process** menu in the NightView main window, select the **Debug Heap...** option.

The **Debug Heap** window is shown.

- Select the **Enable heap debugging** checkbox at the top of the dialog.
- Press the **Medium** button in the **Debugging Level** area.
- Change the **Specify check heap freq** text field to 1.

The Debug Heap window should look similar to the following figure:

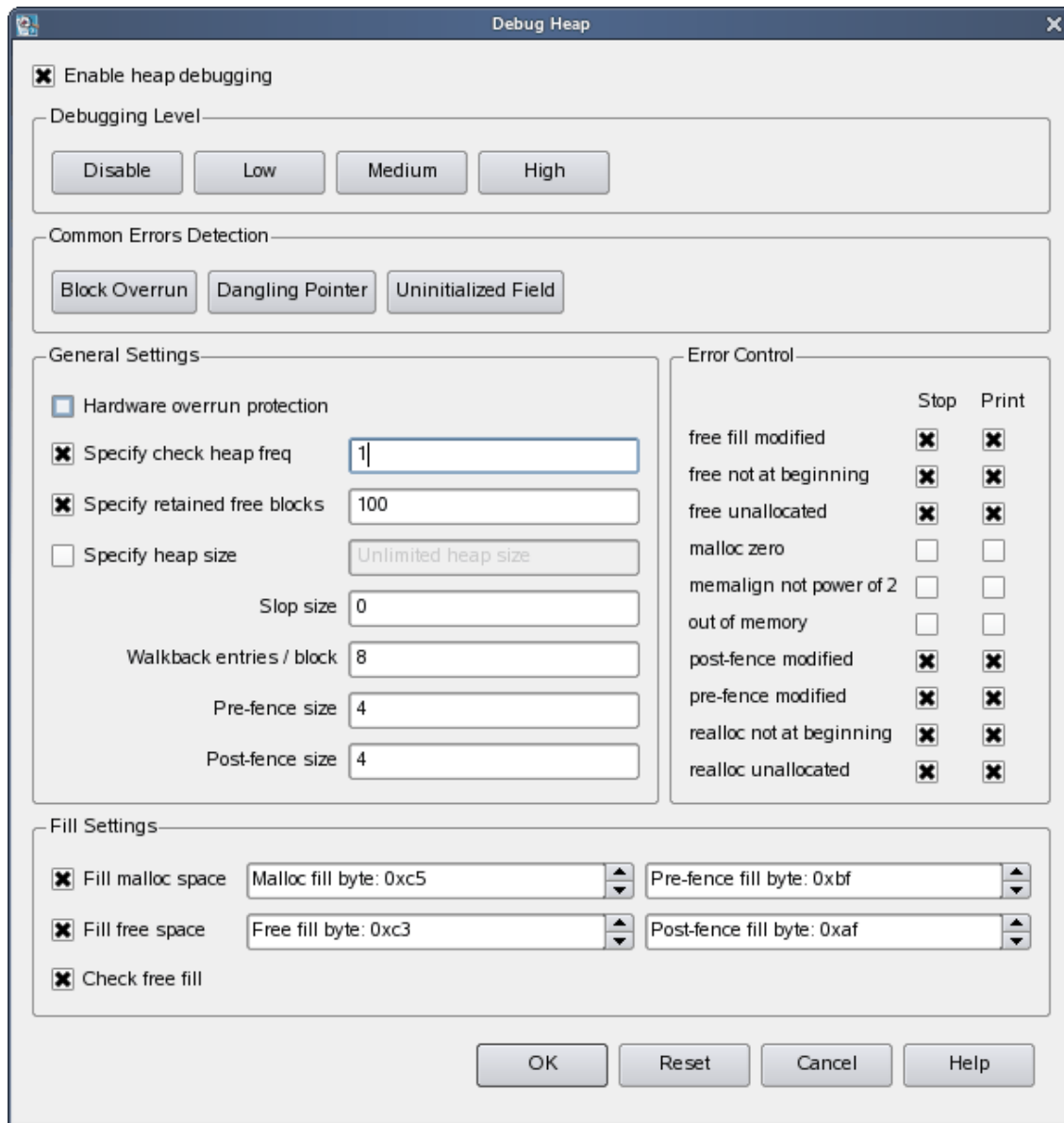


Figure 3-20. NightView Debug Heap Dialog

- Press the OK button to apply the changes and close the dialog.

These options instruct the debugger to activate heap debugging, retain freed blocks to detect certain kinds of errors, allocate some additional memory past the end of the requested size to detect errors, and stop the program when any heap error is detected.

- Click the Resume button  to resume process execution.

Setting up Heap Debugging Scenarios

The fourth thread created by the main program executes a routine called `heap_thread`.

This routine iteratively executes various dynamic memory operations based on the setting of the `scenario` variable. These operations are representative of common programming errors relating to dynamic memory.

Let's set a breakpoint on line 130.

- Scroll to line 130 in the source window:

```
sleep(5);
```

- Right-click anywhere on that line and select **Set simple breakpoint** from the context menu.

NOTE

Optionally, you could set a breakpoint on line 130 by using the **Eventpoint** menu and selecting **Set Breakpoint**, or by entering the following command in the **Command** panel of the NightView main window:

```
break app.c:130
```

The process will hit the breakpoint in the `heap_thread` and all threads will stop.

Scenario 1: Use of a Freed Pointer

A common error is to read or write a block of memory that has already been freed.

A way to detect this is to tell NightView to retain freed blocks and fill the freed blocks with a specific pattern. If the blocks are subsequently read, your application may more quickly discover the error since the contents are unexpected. If the blocks are modified, NightView can detect this.

By default, the `heap_thread` will not actually execute any of the five scenarios.

- To cause it to execute scenario 1, set the variable `scenario` to 1 by entering the following commands in the **Command** panel:

```
set scenario=1
resume
```

This causes the following snippet of code to be executed after a delay of 5 seconds:

```
ptr = alloc_ptr(1024,3);
free_ptr (ptr,2);
memset (ptr, 47, 64);
```

The last line represents usage of dynamically allocated space that has already been freed.

NightView will detect this at a heap point inserted by the user, or at a subsequent heap operation (based on the **frequency** setting of the **heapdebug** command), in this case on line 170.

NightView will stop the process once the heap error has been detected and issue a diagnostic similar to the following:

```
Heap errors in process local:3771:
  free-fill modified in free block (value=0x804a818)
#0 0x8048b6d in heap_thread(void*unused=0) at app.c line 170
```

The error refers to the fact that locations within the freed block were modified by the process after the block was freed.

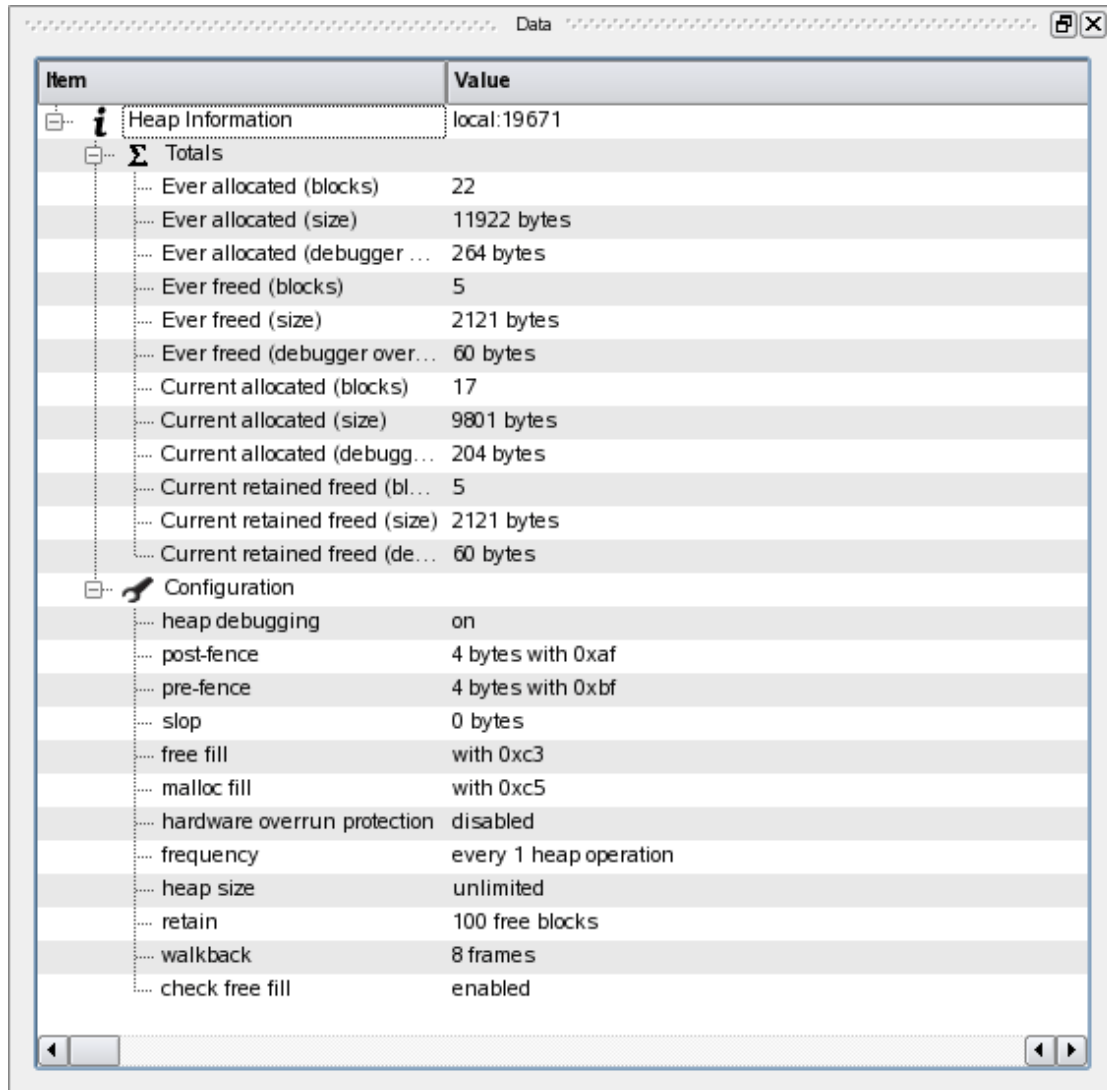
The **Data** panel is useful for displaying heap-related information as well as a variety of other attributes.

- From the **Data** menu select **Heap Information**.

A new tab may be created for the **Data** panel unless the application was still running. It is added to the NightView main window in the same location as the **Locals** and **Context** panels.

- Click on the **Data** tab (this is probably not necessary as it should be raised as the current tab already).
- Resize the first column (if necessary) by clicking on the divider between the column headings and dragging it to the right so that the items of interest below can be seen in their entirety.

- Expand the Configuration item under Heap Information in the Data panel to show the current **heapdebug** settings.
- Expand the Totals item under Heap Information to show summary statistics related to heap activity.



The screenshot shows the 'Data' panel in NightView. It contains a tree view on the left and a table on the right. The tree view has three main items: 'Heap Information' (with an information icon), 'Totals' (with a summation icon), and 'Configuration' (with a key icon). The table displays the following data:

Item	Value
Heap Information	local:19671
Totals	
Ever allocated (blocks)	22
Ever allocated (size)	11922 bytes
Ever allocated (debugger ...)	264 bytes
Ever freed (blocks)	5
Ever freed (size)	2121 bytes
Ever freed (debugger over...)	60 bytes
Current allocated (blocks)	17
Current allocated (size)	9801 bytes
Current allocated (debugg...)	204 bytes
Current retained freed (bl...)	5
Current retained freed (size)	2121 bytes
Current retained freed (de...)	60 bytes
Configuration	
heap debugging	on
post-fence	4 bytes with 0xaf
pre-fence	4 bytes with 0xbf
slop	0 bytes
free fill	with 0xc3
malloc fill	with 0xc5
hardware overrun protection	disabled
frequency	every 1 heap operation
heap size	unlimited
retain	100 free blocks
walkback	8 frames
check free fill	enabled

Figure 3-21. Heap Totals and Configuration

NOTE

In general, all information in the Data panel is updated whenever the process being debugged stops.

The values in the Totals section will vary from system to system.

- Collapse the Totals and Configuration items.
- Click on the tab labeled Locals.

The list of items in the **Locals** panel represents the local variables associated with the current frame being displayed. Note that the value of the variable `ptr` is displayed in red because it no longer contains a valid (allocated) heap address.

Expanding the `ptr` item reveals the `(heap info)` item. Expanding that item reveals additional information relating to the block that the pointer once referred to including:

- its state - **freed, but retained**
- its address range
- its size
- errors
- free and allocation information, which when expanded include walkback information relating to the routines which allocated and freed the block.

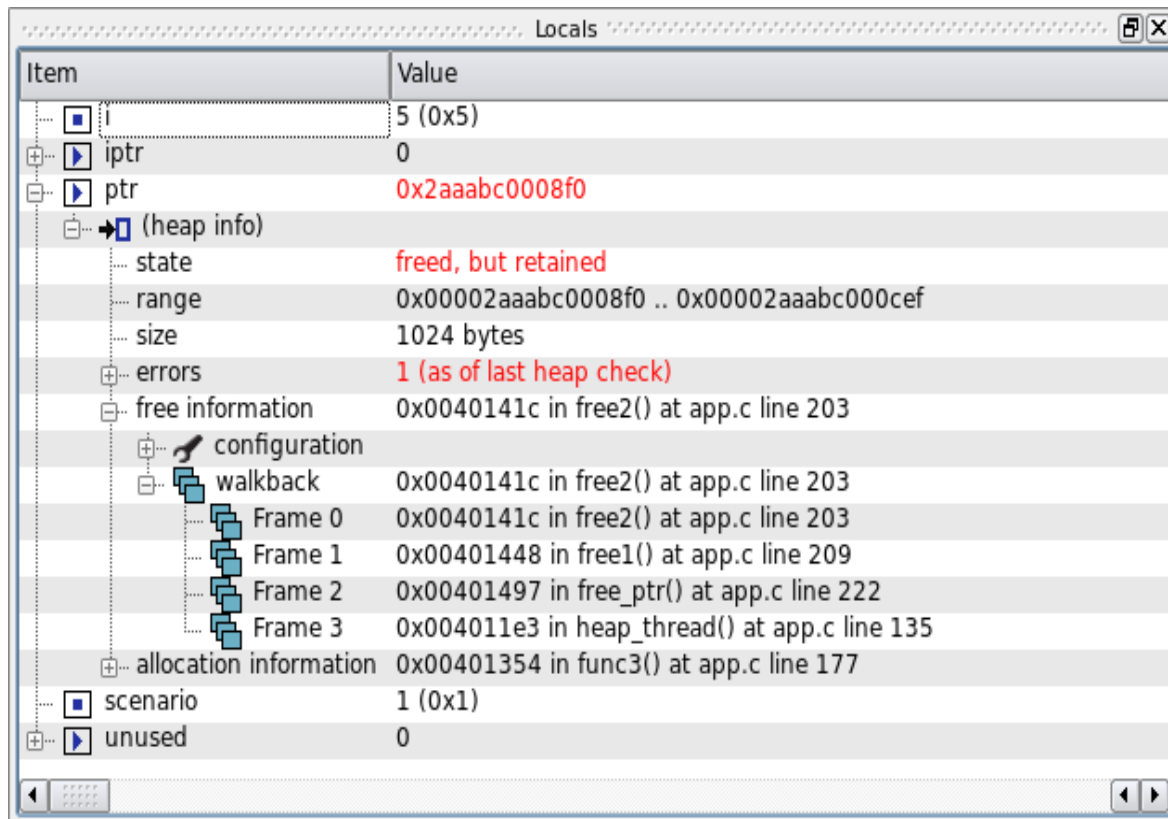


Figure 3-22. Heap Locals Info

Scenario 2: Freeing an Invalid Pointer Value

Another common error is to free a pointer multiple times or to free a value which doesn't actually refer to a heap block.

- Resume the process and let it reach the breakpoint on line 130:

```
resume
```

- Set the variable `scenario` to 2:

```
set scenario=2
resume
```

This causes the following snippet of code to be executed after a delay of 5 seconds:

```
ptr = alloc_ptr(1024,3);
free_ptr(ptr,2);
free(ptr);
```

NightView will detect the failure and print a diagnostic similar to the following:

```
Heap error in process local:3771: free called on freed or
unallocated block (value=0x804ac40)
#0 0x8048a78 in heap_thread(void*unused=0) at app.c line 142
```

NOTE

If you have glibc debugging information installed on the system, NightView may show you a different frame in the glibc allocator. If so, enter the command **up** until you are presented with a frame in `heap_thread`.

Another way of obtaining information about the heap block in question is to use the **info memory** command. It provides text about the information available in the **Locals** panel under the `ptr` item to the **Messages** panel of the NightView main window.

- Issue the following command in the **Command** panel:

```
info memory ptr
```

NightView will provide output similar to the following in the **Messages** panel:

```

Messages
info memory ptr

Memory map enclosing address 0x000015553c001030 for process local:139999:

Virtual Address Range          No. bytes
Comments
-----
0x000015553c000000 0x000015553c020fff      135168
    Readable,Writable

Allocator information for address 0x000015553c001030 for process local:139999:

freed, but retained
in block 0x000015553c001030 .. 0x000015553c00142f (1024 bytes)
no errors detected in block
free information:
    4 post-
fence bytes with 0xaf (fence range 0x000015553c001430 .. 0x000015553c001433)
    4 pre-
fence bytes with 0xbf (fence range 0x000015553c00102c .. 0x000015553c00102f)
free fill with 0xc3
malloc fill with 0xc5
walkback:
    0x0000000000401841 in free2() at app.c line 203
    0x000000000040186d in free1() at app.c line 209
    0x00000000004018bc in free_ptr() at app.c line 222
    0x0000000000401647 in heap_thread() at app.c line 141
allocation information:
    4 post-
fence bytes with 0xaf (fence range 0x000015553c001430 .. 0x000015553c001433)
    4 pre-
fence bytes with 0xbf (fence range 0x000015553c00102c .. 0x000015553c00102f)
free fill with 0xc3
malloc fill with 0xc5
walkback:
    0x0000000000401779 in func3() at app.c line 177
    0x00000000004017a2 in func2() at app.c line 182
    0x00000000004017da in func1() at app.c line 188
    0x000000000040189a in alloc_ptr() at app.c line 217
    0x0000000000401632 in heap_thread() at app.c line 140

```

Figure 3-23. info memory Command Output

Note that it reports no error in the block per se. The actual problem here is that a second attempt was made to free the block when it already had been freed previously.

In this case, the walkback information associated with the actual free is useful as you can quickly locate what code segment actually freed the block.

Scenario 3: Writing Past the End of an Allocated Block

Another common error is to allocate insufficient space or to write past the end of an allocated block.

- Resume the process and let it reach the breakpoint on line 130:

```
resume
```

- Set the variable scenario to 3:

```
set scenario=3
resume
```

This causes the following snippet of code to be executed after a delay of 5 seconds:

```
ptr = alloc_ptr(strlen(MyString),2);
strcpy (ptr, MyString); // oops -- forgot the zero-byte
```

NightView will detect the failure and print a diagnostic similar to the following:

```
Heap errors in process local:3771:
  post-fence modified in block (value=0x804b068)
  #0 0x8048b6d in heap_thread(void*unused=0) at app.c line 170
```

Note that the description of the variable `ptr` in the **Locals** panel does not indicate an invalid status. That is because `ptr` does point to a valid heap block.

However, expanding the (heap info) information for `ptr` and the errors list indicates that the block referenced by the `ptr` is invalid because the post-fence was modified.

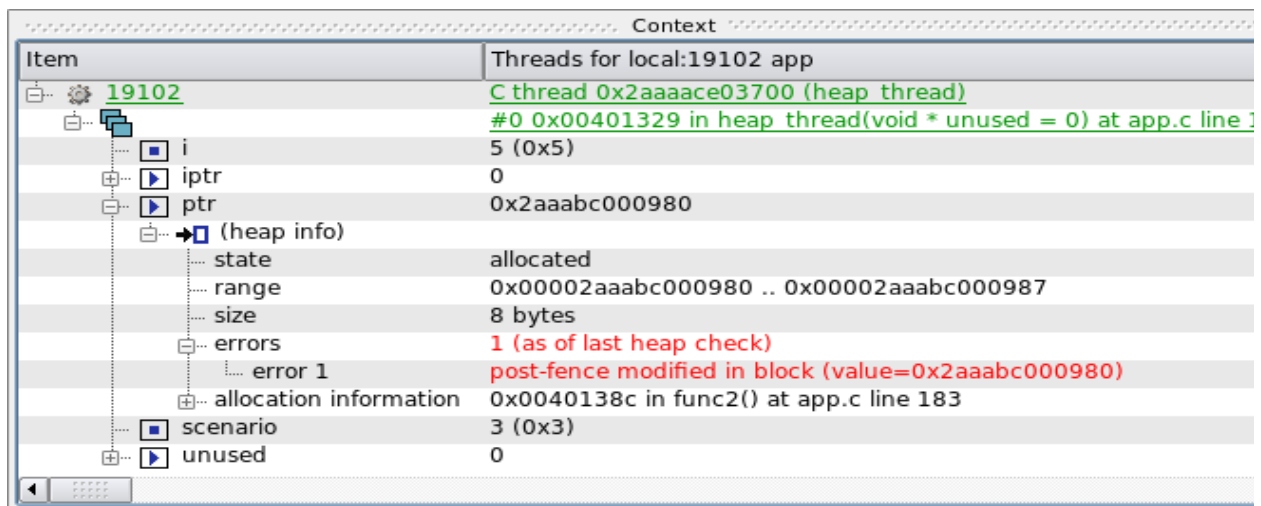


Figure 3-24. Heap Error Description

Scenario 4: Use of Uninitialized Heap Blocks

Another common error is forgetting to initialize dynamically allocated memory before using it. Code segments may assume that dynamically allocated memory is initialized to zero, as is the case with `calloc()` but not `malloc()`.

- Resume the process and let it reach the breakpoint on line 130:

```
resume
```

- Tell NightView to stop whenever a SIGSEGV is sent to the process and also set the variable `scenario` to 4:

```
handle sigsegv stop print pass  
set scenario=4  
resume
```

This causes the following snippet of code to be executed after a delay of 5 seconds:

```
iptr = (int**)alloc_ptr(sizeof(void*),2);  
if (*iptr) **iptr = 2778;
```

NightView will detect the failure and print a diagnostic similar to the following:

```
Process local:3771 received SIGSEGV  
#0 0x8048ad2 in heap_thread(void*unused=0) at app.c line 153
```

One heap debugging option instructs NightView to fill newly allocated, uninitialized space with a specific pattern to make it easier to detect use of uninitialized memory. The **Fill malloc space** field in the **Debug Heap** dialog that we used when enabling heap debugging specified the byte pattern to be `0xc5`.


- Issue the following command to view the content of the uninitialized memory block:

```
x/x iptr
```

A SIGSEGV signal is a fatal error so we must restart the process to continue the tutorial.

- Issue the following command:

```
kill
```

- Re-initiate the program by pressing the ReRun icon  in the Process toolbar:

NOTE

Alternatively, you can issue the following command directly from the **Command** panel to initiate the process:

```
rerun
```

NOTE

NightView automatically re-applies all eventpoint and heap control settings when it sees the subsequent execution of the program.

Scenario 5: Detection of Leaks

Another situation which may be indicative of error or inappropriate use of memory are leaks. In this instance, we define a leak as a dynamically allocated block of memory that is no longer referred to by any pointer in the program.

Detection of leaks is a *very expensive* process with respect to CPU utilization and intrusion on the user application. As such, leak detection is only executed when an explicit request is made from the user.

- Resume the process and let it reach the breakpoint on line 130:

```
resume
```

- Enter the following commands:

```
set scenario=5  
resume
```

This causes the following snippet of code to be executed after a delay of 5 seconds:

```
ptr = alloc_ptr(37,1);  
ptr = 0;
```

NightView does not detect the leak automatically, as mentioned above. The process will stop again when the breakpoint on line 130 is reached.

- At this time, specifically request a leak report by clicking the **Data** menu then selecting **Heap Leaks...** then check the **New Leaks** radio button, and press **OK** in the **Data Heap Leaks** dialog to add the item to the **Data** panel.

This operation causes NightView to analyze the program for leaks and displays a **Leak Sets** item in the **Data** panel. On small programs, this operation may appear to be insignificant, but for larger programs it can take some significant time.

- Click on the **Data** tab.
- Expand the **Leak Sets** item, if necessary.

An additional item is displayed for every leak set with a matching block size that was allocated with a matching walkback. Expansion of individual sets provides the common walkback shown for each allocation as well as expandable descriptions of each individual leaked block.

- Expand the leak set item with size 37 and then expand the walkback item associated with it.

Note the walkback indicating that it was allocated by the `heap_thread()` routine on line 157 of **app.c**.

Item	Value
Heap Information	local:19137
Totals	
Configuration	
Leak Sets	local:19137: new at heap operation 112
leak set, 48 bytes	1 block of 48 bytes, 0x00002aaaaace32c3 at <xt_get_default_xtconfig+21>
leak set, 37 bytes	1 block of 37 bytes, 0x004013c4 in func1() at app.c line 189
walkback	0x004013c4 in func1() at app.c line 189
Frame 0	0x004013c4 in func1() at app.c line 189
Frame 1	0x00401475 in alloc_ptr() at app.c line 217
Frame 2	0x004012a4 in heap_thread() at app.c line 157
blocks	
leak set, 32 bytes	1 block of 32 bytes, 0x00002aaaaace8487 at <xt_trace_register_thread+17>
leak set, 1 bytes	1 block of 1 bytes, 0x0040132d in heap_thread() at app.c line 170

Figure 3-25. Heap Leaks Display

NOTE

The Leak Sets display will vary depending on your system type. Concentrate on the leak set of 37 bytes as shown above.

The last frame shown may be in `heap_thread()`, or it may be in the pthread library (`start_thread()`), depending on whether you have pthread debugging files installed on your system.

NOTE

Unlike most items in the Data panel, the leak sets item is not automatically updated when the process stops. The description is a snapshot of the leaks at a certain moment in the execution of the program, and therefore it will remain unchanged even if additional leaks occur. To get updated information, request another leak report (go to the Data menu and select **Heap Leaks...**).

Scenario 6: Allocation Reports

NightView provides a detailed report of all allocated memory.

Construction of this report is a *very expensive* process with respect to CPU utilization and intrusion on the user application execution time. As such, allocation reports are only executed when an explicit request is made from the user.

- Set the variable `scenario` to 6:

```
set scenario=6
resume /one
```

This resumes only the `heap_thread` (because of the `/one` parameter to the `resume` command) and causes additional allocations to be made.

The thread will stop again when the breakpoint on line 130 is reached.

- At that time, specifically request an allocation report by clicking the **Data** menu and selecting **Still Allocated Blocks...** then click the **All Blocks** radio button, and press **OK** in the **Data Still Allocated Blocks** dialog to add the item to the **Data** panel.

This operation causes NightView to analyze the program and displays a `Still Allocated Sets` item in the **Data** panel. On small programs, this operation may appear to be insignificant, but for larger programs it can take some significant time.

- Resize the first column (if necessary) by clicking on the divider between the column headings and dragging it to the right so that the items of interest below can be seen in their entirety.
- Expand the `Still Allocated Sets` item, if necessary. An additional item is displayed for every allocation set with a matching block size that was allocated with a matching walkback. Expansion of individual sets provides the common walkback shown for each allocation as well as expandable descriptions of each individual leaked block.
- Expand the allocated set item with size 1048576 and then expand the walkback item associated with it.

Note the walkback indicates that it was allocated by the `func3()` function, which was initiated by a call to `alloc_ptr()` in the `heap_thread()` routine on line 162 of `app.c`.

Item	Value
Heap Information	local:19137
Leak Sets	local:19137: new at heap operation 112
Still Allocated Sets	local:19137: all at heap operation 218
?→ allocated set, 1048576 bytes	1 block of 1048576 bytes, 0x00401354 in func3() at app.c line 177
walkback	0x00401354 in func3() at app.c line 177
Frame 0	0x00401354 in func3() at app.c line 177
Frame 1	0x0040137d in func2() at app.c line 182
Frame 2	0x004013b5 in func1() at app.c line 188
Frame 3	0x00401475 in alloc_ptr() at app.c line 217
Frame 4	0x004012c1 in heap_thread() at app.c line 162
?→ blocks	
?→ allocated set, 8177 bytes	1 block of 8177 bytes, 0x00401354 in func3() at app.c line 177
?→ allocated set, 4564 bytes	1 block of 4564 bytes, 0x00401354 in func3() at app.c line 177
?→ allocated set, 1024 bytes	1 block of 1024 bytes, 0x0040138c in func2() at app.c line 183
?→ allocated set, 272 bytes	1 block of 272 bytes, 0x00002aaaaaabe8a4 at <_dl_allocate_tls+36>
?→ allocated set, 272 bytes	1 block of 272 bytes, 0x00002aaaaaabe8a4 at <_dl_allocate_tls+36>
?→ allocated set, 272 bytes	1 block of 272 bytes, 0x00002aaaaaabe8a4 at <_dl_allocate_tls+36>
?→ allocated set, 272 bytes	1 block of 272 bytes, 0x00002aaaaaabe8a4 at <_dl_allocate_tls+36>
?→ allocated set, 62 bytes	1 block of 62 bytes, 0x004013c4 in func1() at app.c line 189
?→ allocated set, 48 bytes	1 block of 48 bytes, 0x00002aaaaace32c3 at <xt_get_default_xtconfig+21>
?→ allocated set, 37 bytes	1 block of 37 bytes, 0x004013c4 in func1() at app.c line 189
?→ allocated set, 32 bytes	1 block of 32 bytes, 0x00002aaaaace8487 at <xt_trace_register_thread+17>

Figure 3-26. Still Allocated Blocks Display

NOTE

The data from the **Still Allocated Sets** will vary depending on your system. Concentrate on the allocated set of 1048576 bytes as shown above.

NOTE

Unlike most items in the **Data** panel, the **Still Allocated Sets** item is not automatically updated when the process stops. The description is a snapshot of the leaks at a certain moment in the execution of the program, and therefore it will remain unchanged even if additional items are allocated or freed. To update the information, request another allocation report (click the **Data** menu then select **Still Allocated Blocks...**).

Disabling Heap Debugging

- To disable all overhead associated with heap debugging, issue the following command:

```
heapdebug off
```

- Delete the breakpoint on line 130 by right-clicking on that breakpoint in the Eventpoints panel and selecting **Delete** or by issuing the following command:

```
clear app.c:130
```

This concludes the tutorial's topic on heap debugging.

Ready for NightTrace

This section assumes you have completed the steps in the section "Using Tracepoints" on page 3-23. If not, go back to that section and complete those steps and then return here.

- To launch NightTrace, open the **Tools** menu, select **NightTrace**, and then select **NightTrace Analyzer**.

The remaining sections of the tutorial do not use NightView, however, we want to keep the tracepoints patched into the executable. We will now detach the program from NightView but it will continue to execute and will retain all patchpoints and tracepoints.

- If the process is running, stop the process by typing the following into the Command panel:

```
stop /protected
```

- In the **Process** menu select **Detach**.
- In the **File** menu select **Exit NightView** to exit NightView.

NOTE

Normally, processes started from within NightView will be killed when NightView exits, even if they have been detached. This is because the shell that is used by NightView to invoke them sends them a SIGHUP signal. Our application ignores SIGHUP, so it can continue to execute.

Conclusion - NightView

This concludes the NightView portion of the NightStar RT Tutorial.

Using NightTrace

NightTrace is a graphical tool for analyzing the dynamic behavior of single and multiprocessor applications. NightTrace can log user-defined application data events from simultaneous processes executing on multiple CPUs or even multiple systems. NightTrace can also log kernel events such as individual system calls, context switches, machine exceptions, page faults and interrupts. By combining application events with kernel events, NightTrace presents a synchronized view of the entire system. Furthermore, NightTrace allows you to zoom, search, filter, summarize, and analyze those events in a wide variety of ways.

Using NightTrace, you can manage multiple user and kernel NightTrace daemons simultaneously from a central location. NightTrace provides you with the ability to start, stop, pause, and resume execution of any of the daemons under its management.

NightTrace lets you define and save a “session” consisting of one or more daemon definitions. These definitions include daemon collection modes and settings, daemon priorities and CPU bindings, and data output formats, as well as the trace event types that are logged by that particular daemon.

Invoking NightTrace

NightTrace was invoked during the last step of the Using NightView section.

If you skipped the Using NightView section, build the program (See “Building the Program” on page 1-4), run the program in NightView (**nview ./app**), and execute the steps in “Using Tracepoints” on page 3-23 before beginning this section of the tutorial (and resume execution of the process).

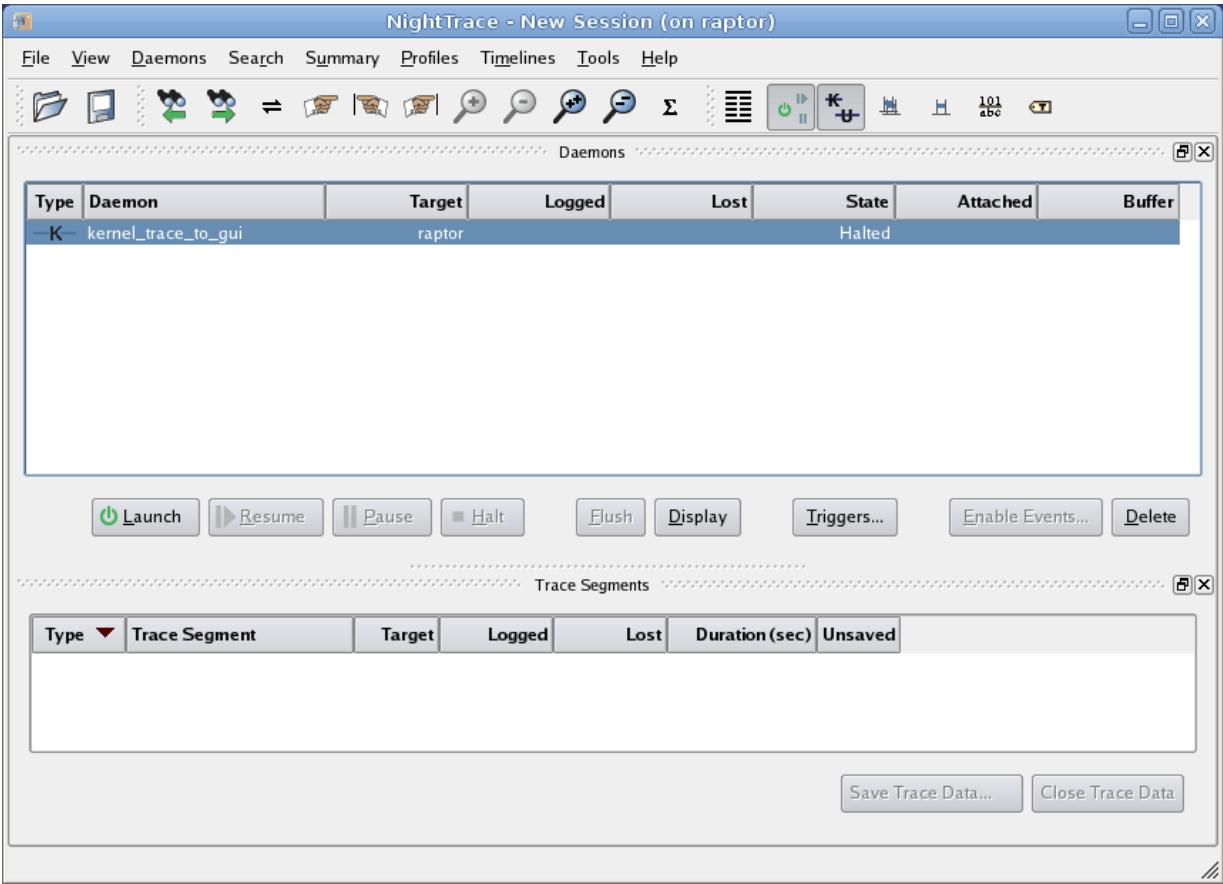


Figure 4-1. NightTrace Main Window

Below the menu bar and toolbar, the first page of the NightTrace main window contains the following two panels:

Table 4-1. NightTrace Panels

Panel	Description
Daemons	Shows the daemons configured.
Trace Segments	Shows each trace segment (contiguous collection of trace data).

The statistics on the **Daemons** panel indicate the number of raw events in the shared memory buffer used between the daemon and the user application and the number of raw events written to NightTrace by the daemon (under the **Buffer** and **Logged** columns, respectively).

The **Trace Segments** panel indicates the number of processed events that are currently available for immediate analysis through the **Events** panels and timelines.

NOTE

The number of events shown in the **Daemons** panel will normally differ from the number of events shown in the **Trace Segments** panel. The former are raw events and the latter are processed events. A processed event is often constructed from multiple raw events.

Configuring a User Daemon

NightTrace allows you to configure a user daemon to collect user trace events.

User trace events are generated by user applications that use the NightTrace API or by those inserted into a program by NightView.

We will configure a user daemon to collect the events that our **app** program logs.

To configure a user daemon based on a running application

- From the **Daemons** menu select Import... then Running Application.

The Import Daemon Definitions dialog is presented:

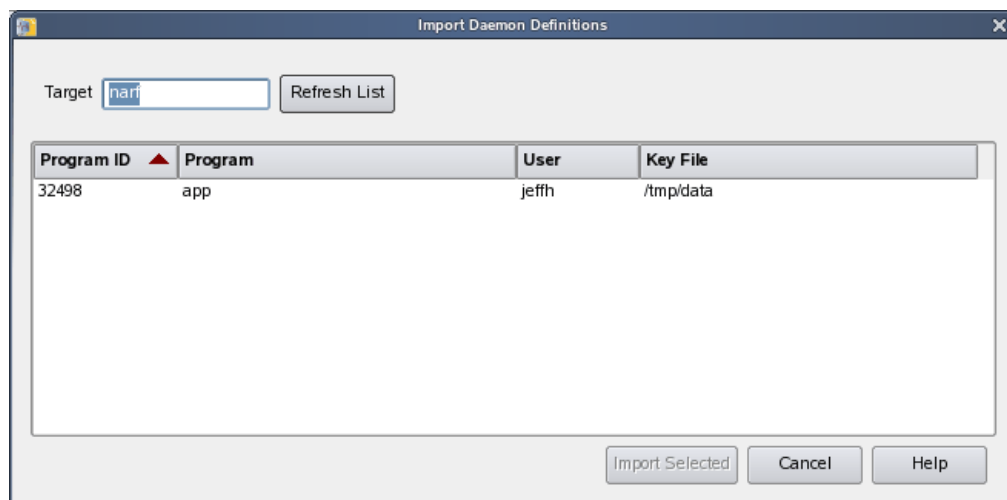


Figure 4-2. Import Daemon Definitions Dialog

The Import Daemon Definitions dialog allows you to define daemon attributes based on a running user application containing NightTrace API calls.

- Select the entry corresponding to the **app** application.
- Press the Import Selected button.

The Import Daemon Definitions dialog closes and a new user daemon is created and added to the Daemon Control Area in the NightTrace main window.

Streaming Live Data to the NightTrace GUI

NightTrace allows you to use a daemon to capture trace events and store them in a file for subsequent analysis or to stream the events directly into the graphical interface for live analysis.

Our daemon is configured for live streaming.

- Select the daemon labeled `app_data` from the **Daemons** panel in the NightTrace main window.
- Press the **Launch** button.
- Press the **Resume** button.

The daemon is now collecting events which are being generated by the **app** program from the tracepoints we inserted via NightView in “Using Tracepoints” on page 3-23.

In the **Daemons** panel, the count of events shown in the **Buffer** column will begin to change.

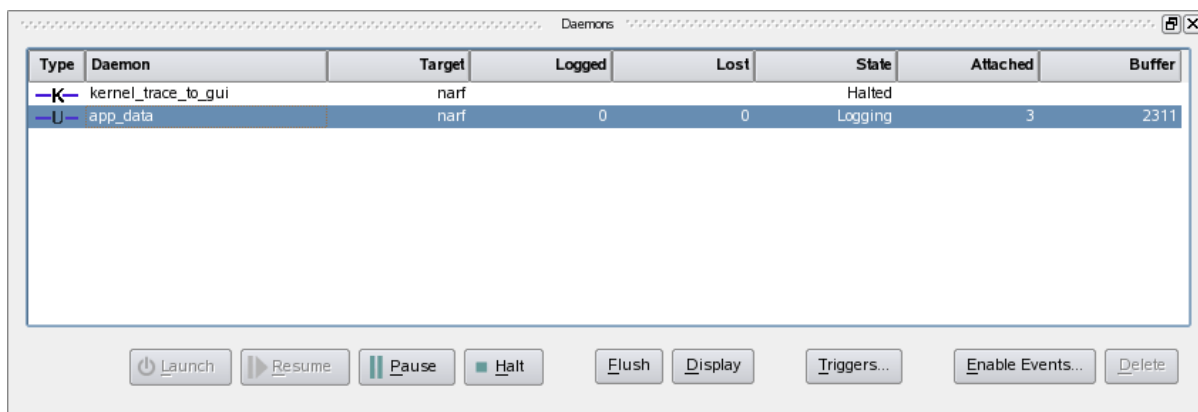


Figure 4-3. Logging Data

Halting the Daemon

Since the NightTrace portion of the tutorial is rather lengthy and may likely be a new experience for many users, we will halt the daemon to reduce memory consumption.

- Wait until the sum of the **Buffer** and **Logged** cells contains at least 10,000 and then halt the daemon by pressing the **Halt** button.

NOTE

Do not be concerned if the number of events shown in the **Trace Segments** panel is smaller than the number of events shown in the **Daemon Control Area** just before you

halted the daemon. The latter shows raw event counts whereas the Trace Segments panel shows processed event counts -- a processed event is often constructed from multiple raw events.

Viewing Events

This page is automatically generated by NightTrace and has individual rows for individual threads (those described via `set_thread_name(2)` in the NightTrace API (see the *NightTrace User's Guide* for details), as well as a row for any events that the process generated. Initially, these rows may be blank until you scroll to see events logged a bit later than the interval shown.

- Click on the newly-created tab labeled `app_data` that contains the Events panel and the timeline associated with those events.

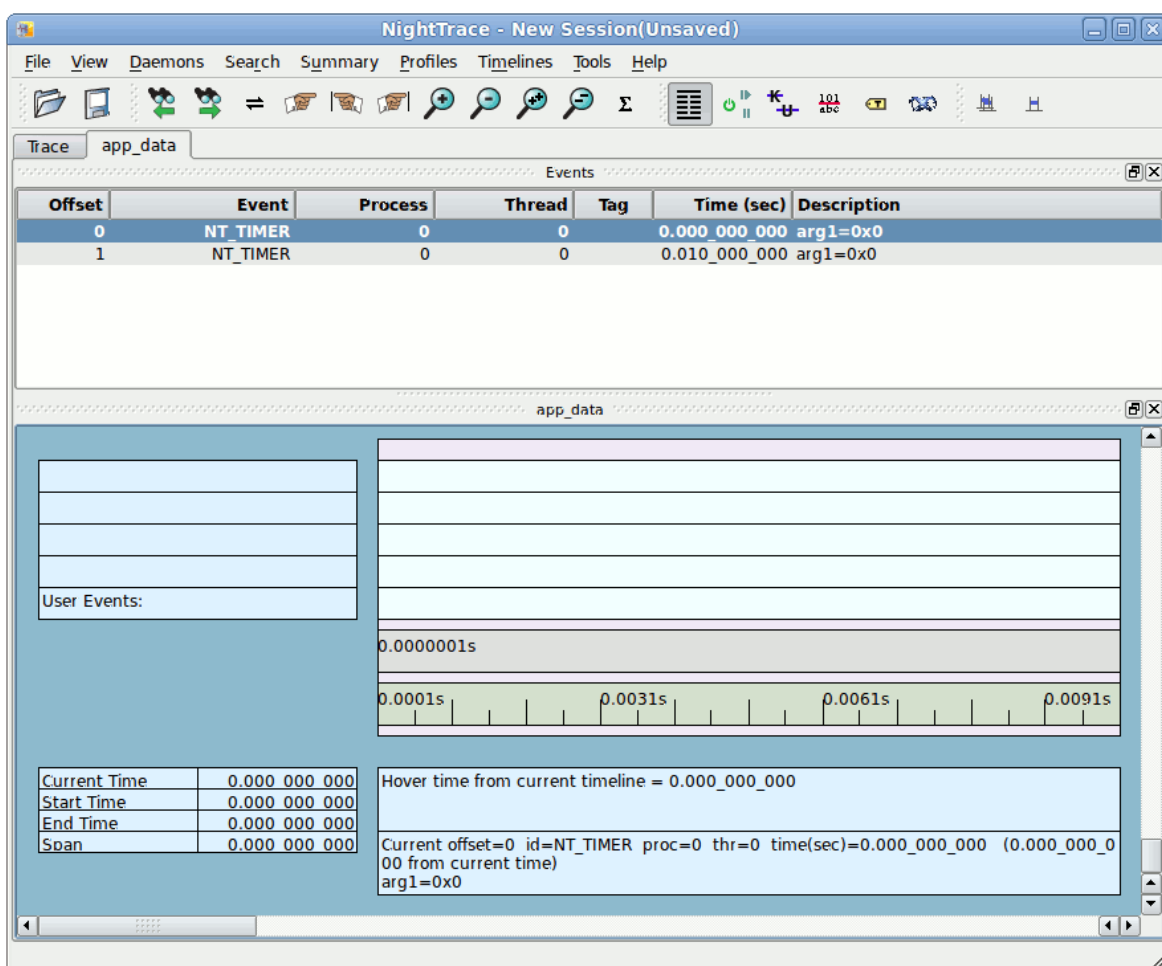


Figure 4-4. `app_data` Page

NOTE

If you have previously used NightTrace you may have saved Preferences which override the default colors as shown above. To change your preferences to use the colors shown above, go to the **File** menu, select **Preferences**, click on the **Timelines** tab, click the **Restore Defaults** button, check the **Apply to existing timelines** checkbox, and then click on **Save**.

Initially, the panels may be mostly blank.

You can force events to be flushed from the daemon buffer and output stream to be brought into the segment area for immediate viewing by zooming out on a timeline.

- Click anywhere in the display area containing the timelines.
- Press **Up** to zoom out.
- Press **Alt-Up** to zoom out completely.

The **Events** list will be populated with the events currently logged and the timeline will graphically display those events. If you don't see any events, press **Alt-Up** again.

- Click in the middle of the lower panel.

NOTE

To reverse the zoom behavior of the keys or to change the default zoom factor, go to the **File** menu and select **Preferences...** then make your desired adjustments in the **Zooming Controls** section of the dialog. Click **OK** to save and exit.

Using NightTrace Timelines

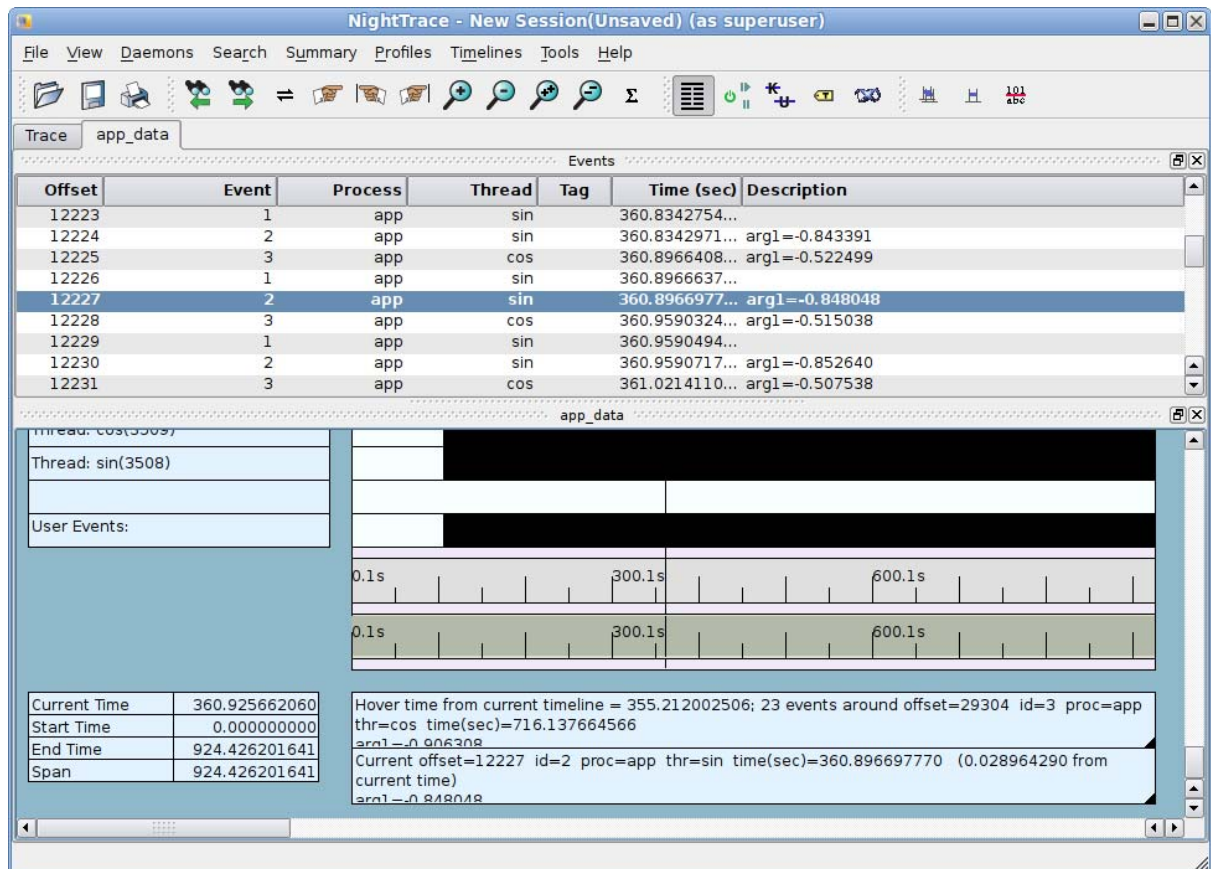


Figure 4-5. NightTrace Timeline

The timeline contains static and dynamic labels and event and state graphs.

By default, NightTrace detects the threads that have registered themselves through NightTrace API calls and creates individual labels and graphs for each thread.

Our application contains five threads, four of which have registered themselves with specific thread names: heap, sin, cos, and main. Rows for individual threads show only events logged by that thread. In addition, there is a user events graph near the bottom that shows events for all threads.

NOTE

You will see blank labels and graphs in your timeline. These are the labels and graphs for the main and heap threads which are not logging any events. The contents of the label are not shown until at least one event is logged by a thread.

If you see all blank labels, you likely didn't click in the middle of the timeline as instructed in the preceding step.

In "Using Tracepoints" on page 3-23 in the Using NightView section, we inserted tracepoints into the `sine` and `cosine` threads, which registered themselves as "sin" and "cos", respectively.

Zooming

Each vertical line in the graph represents at least one event. You can zoom in and zoom out to adjust the level of detail.

- Left click anywhere within the timeline
- Press the **Down** key repeatedly until you can see individual lines in the graph
- Press the **Up** key to zoom back out
- If you have a mouse wheel, move the wheel back and forth to zoom in and out

The vertical dashed line is the current timeline and is directly connected to the highlighted event in the **Events** panel.

Left-clicking the mouse in the display area moves the current timeline. The information in the **Event Detail** area below the timeline changes to reflect the event closest to the left of the current timeline.

Moving The Interval

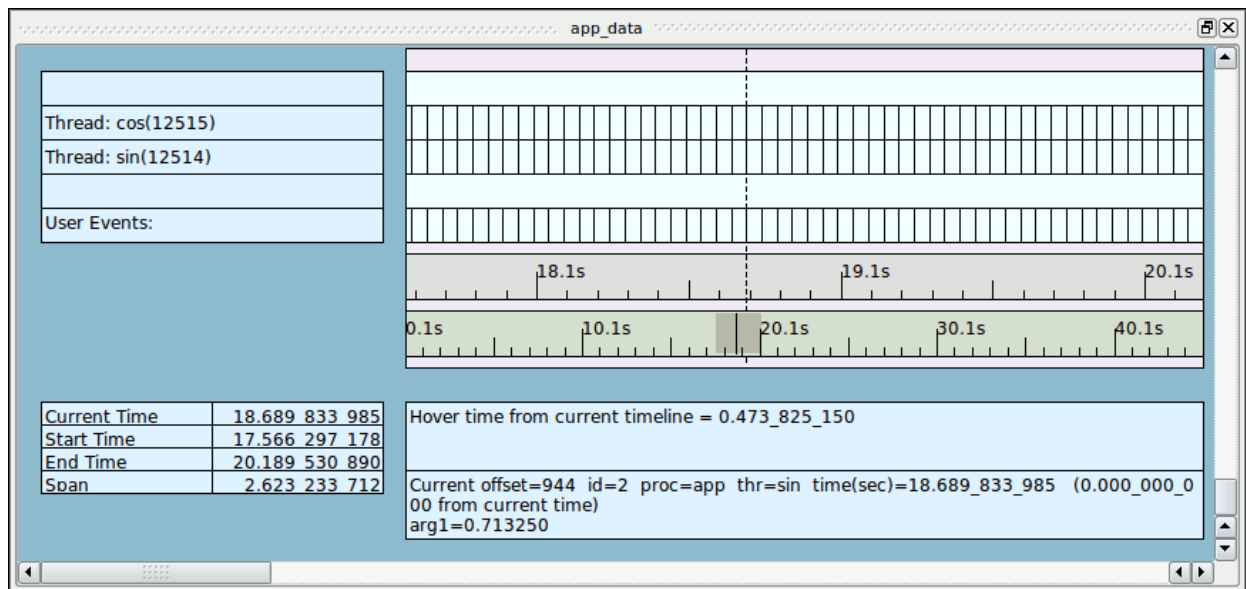


Figure 4-6. Timeline Interval Panel

By default, each timeline panel has two ruler rows positioned below the event graphs and above the descriptive boxes at the bottom of the panel. These contain numbers and hash marks that describe intervals of time.

The ruler on top indicates the timespan currently shown.

The ruler on the bottom indicates the timespan for all data currently available for viewing. This ruler is called the global ruler and has a gray area within it. The gray area represents the amount of the entire timespan that is currently shown in the panel. Thus zooming in will decrease the width of the gray area and zooming out will have the opposite effect.

NOTE

If you do not see a gray area, zoom out until you do.

There are several methods of moving through the entire timeline.

- Press the **Right** key

This causes the current timeline to go to the next event. If you are zoomed out too far, you may not notice the timeline moving. In this case, either zoom in or hold the Right key down until you can see the timeline move.

Alternatively, press the **Left** key to move the current timeline to the previous event.

- Press **Ctrl+Right**

This causes the displayed interval to move 25% of a section to the right by default. The section is the amount of time currently visible in the interval. Notice how the gray area in the global ruler moves.

Alternatively, pressing **Ctrl+Left** causes a shift one section to the left.

- Click midway between the gray area and the far-right portion of the global ruler.

Clicking anywhere in the global ruler causes the interval to shift to be centered at the selected time at the current zoom setting.

Thus to move the very beginning or end of the data set, you can click the beginning or end of the global ruler.

Using the Events Panel for Textual Analysis

Events						
Offset	Event	Process	Thread	Tag	Time (sec)	Description
941	2	app	sin		18.639_737_052	arg1=0.707107
942	3	app	cos		18.689_830_607	arg1=0.713250
943	1	app	sin		18.689_830_614	
944	2	app	sin		18.689_833_985	arg1=0.713250
945	1	app	sin		18.739_897_536	
946	3	app	cos		18.739_897_556	arg1=0.707107
947	2	app	sin		18.739_900_519	arg1=0.719340

Figure 4-7. Events Panel

The events shown in the **Events** panel are synchronized with the events shown in the timeline. The highlighted event indicates the current timeline.

- Click on a line in the **Events** panel.
- Press the **Down** key to move to the next event.
- Press the **Up** key to move to the previous event.

Whenever an event is selected or the current event line moves, the **Event Detail** area below the timeline on the right shows additional information about the event, if available.

- Press the **PageDown** key to advance to the next set of events.
- Press the **PageUp** key to shift to the previous set of events.

These actions only move the current timeline by the number of events that can be shown in the **Events** panel.

Customizing Event Descriptions

The event values we logged with the **tracepoint** commands in NightView were event IDs 1-3. We will customize the description of these events.

- Click on a row in the **Events** panel that shows event **Code** 1.
- Right-click that row and select **Edit Current Event Description...** from the context menu.

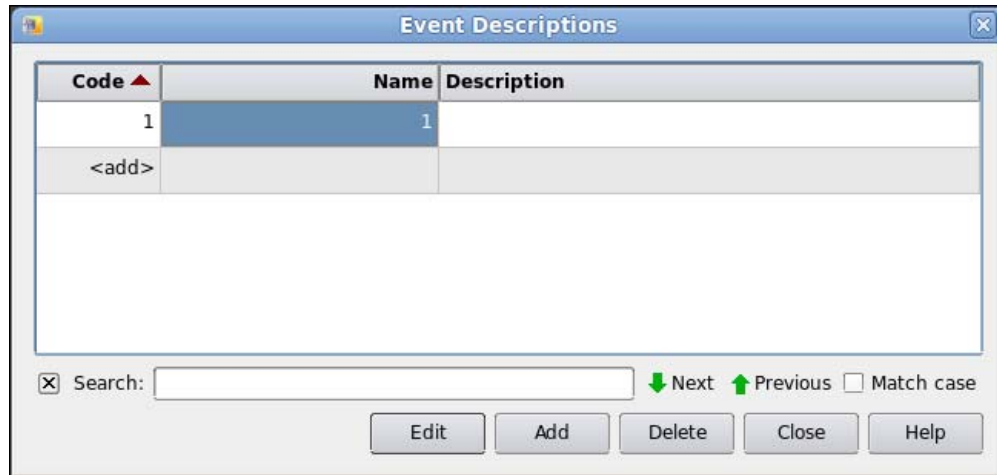


Figure 4-8. Add Event Description dialog

- Double-click the selected **Name** field and enter:

cycle_start

in the **Name** field.

- Double-click the table cell that contains **<add>**.
- Type 2 in the **Code** cell.
- Click in the **Name** cell for that row.
- In the **Name** text field enter:

cycle_end

- Press the **Close** button.

The descriptions of the events in the **Events** panel now correspond to the textual identifiers we assigned to them.

Events						
Offset	Event	Process	Thread	Tag	Time (sec)	Description
938	cycle_end	app	sin		18.589_677_378	arg1=0.700909
939	cycle_start	app	sin		18.639_734_687	
940	3	app	cos		18.639_736_625	arg1=0.719340
941	cycle_end	app	sin		18.639_737_052	arg1=0.707107
942	3	app	cos		18.689_830_607	arg1=0.713250
943	cycle_start	app	sin		18.689_830_614	
944	cycle_end	app	sin		18.689_833_985	arg1=0.713250
945	cycle_start	app	sin		18.739_897_536	
946	3	app	cos		18.739_897_556	arg1=0.707107
947	cycle_end	app	sin		18.739_900_519	arg1=0.719340
948	cycle_start	app	sin		18.789_970_202	
949	3	app	cos		18.789_972_729	arg1=0.700909
950	cycle_end	app	sin		18.789_972_991	arg1=0.725374

Figure 4-9. Events Panel with Event Descriptions displayed

Searching the Events List

We can use the search capabilities of NightTrace to search for a specific occurrence of an event or condition relating to an event or its arguments.

- In the Search menu select Power Search....

A dialog appears containing a list of defined profiles (currently empty) and several fields that allow you to define new profiles or edit existing ones:

Profiles (as superuser)

Type	Name	Status	Count	Last	Offset

Key / Value: Condition Reset Choose Profile...

Events: ALL Browse...

Exclude Events: NONE Browse...

Condition: TRUE

Processes: ALL Browse...

Threads: ALL Browse...

Output Script: /usr/lib/NightTrace/bin/event-summary.sh Browse...

CPUs: all (mask=all)

Name: cond

☒ Close dialog on summary or successful search

Add
Apply
Search Backward
Search Forward
Halt Search
Summarize All Events
Close
Help

Figure 4-10. Searching Using the Profiles Dialog

The top area of the dialog is the **Profiles** list area -- it shows all previously defined profiles.

The rest of the dialog provides mechanisms for defining or changing a profile, and common actions to act upon them.

- Press the **Browse...** button to the right of the **Events** field.



Figure 4-11. Browse Events Dialog

- Click in the **Search** text field and type **cycle**. The first event name that includes that word is shown. Ensure that **cycle_end** is selected in the event list, or press **Next** **Next** until it is. Then press the **Select** button.
- Enter the following text in the **Condition** text field of the **Profiles** panel:

arg_db1 > 0.8

- Enter the following text into the **Name** text field:

obtuse

- Press the **Add** button in the **Profiles** panel.

A profile called `obtuse` is now defined and appears in the **Profile Status List** area of the dialog.

- Press the **Search Forward** button at the bottom of the **Profiles** dialog.

The current timeline is moved to the next event that matched the search criteria, that being the end of a cycle when the sine value exceeded 0.8.

NOTE

If a dialog appears telling you that NightTrace has reached the end of the available dataset and asks you whether it should resume the search at the beginning, press **OK**.

If you didn't capture enough events, it is possible that your data set doesn't include a trace event when the sine value exceeded 0.8. If so, try adjusting the condition to < 0.0 .

- By default, NightTrace closes the **Profiles** dialog and returns you to the timeline as shown below:

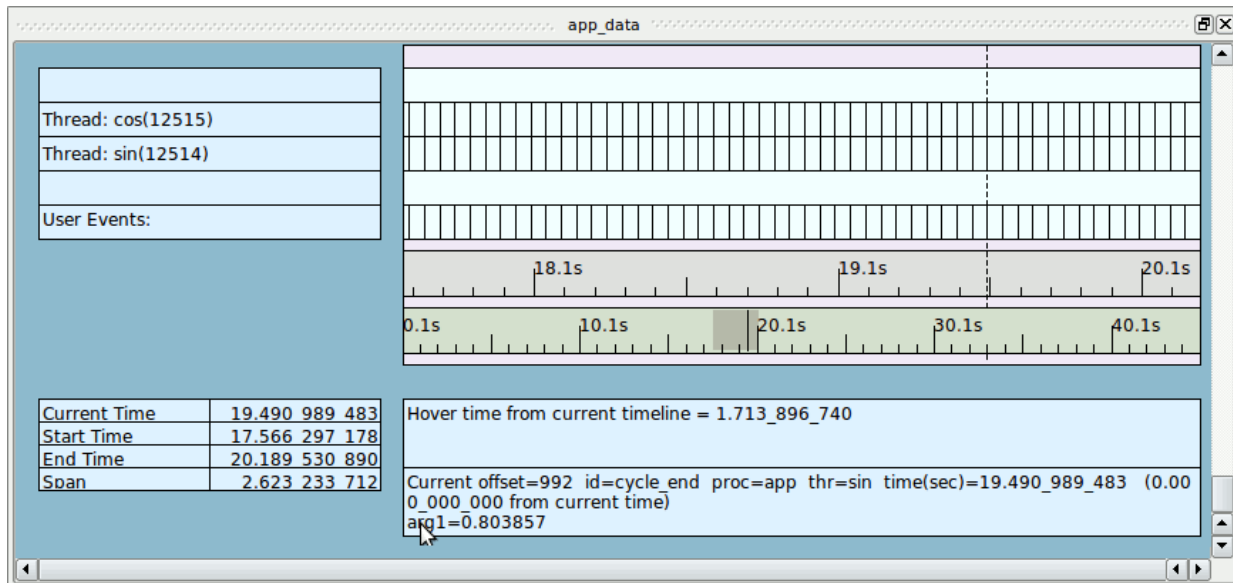


Figure 4-12. Timeline Panel After Search

- Verify that the current event listed in the **Events** panel indicates `arg1` with a value exceeding 0.8.

Events						
Offset	Event	Process	Thread	Tag	Time (sec)	Description
986	cycle_end	app	sin		19.390_874_186	arg1=0.793353
987	3	app	cos		19.440_928_124	arg1=0.615661
988	cycle_start	app	sin		19.440_929_571	
989	cycle_end	app	sin		19.440_931_735	arg1=0.798636
990	3	app	cos		19.490_987_163	arg1=0.608761
991	cycle_start	app	sin		19.490_987_360	
992	cycle_end	app	sin		19.490_989_483	arg1=0.803857
993	3	app	cos		19.541_048_819	arg1=0.601815
994	cycle_start	app	sin		19.541_048_848	
995	cycle_end	app	sin		19.541_051_516	arg1=0.809017
996	cycle_start	app	sin		19.591_126_040	
997	3	app	cos		19.591_126_446	arg1=0.594823
998	cycle_end	app	sin		19.591_128_256	arg1=0.814116

Figure 4-13. Events Panel After Search


Similarly, the timeline shows a description of the current event in the Event Detail area in the bottom portion of the panel.

- Move the mouse cursor to the event description box at the bottom of the panel and leave it there without moving it.

A tool-tip pops up with the full description of the event. This is useful when the description shown is truncated due to the size of the description box on the timeline page.

Using States

In addition to displaying individual events, NightTrace can display states.

- Click either of the Profiles icons on the toolbar. 

The Profiles dialog is displayed with the previously defined profile selected.

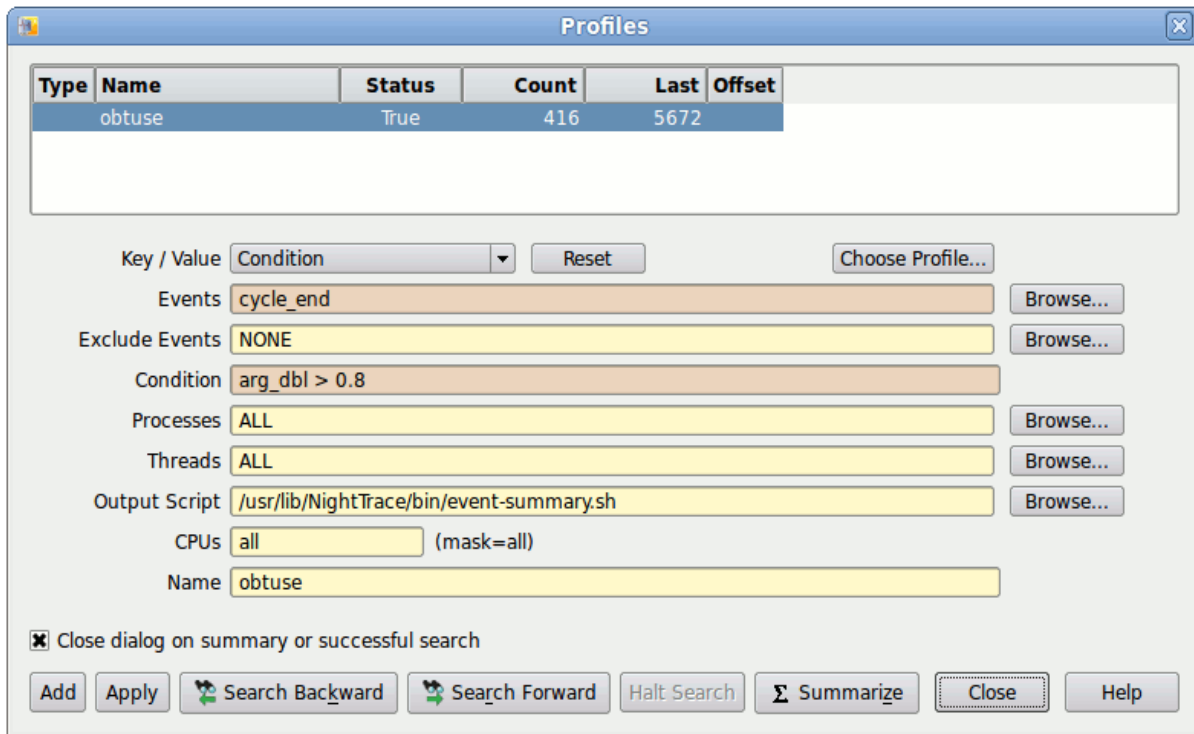


Figure 4-14. Profiles Dialog With Obtuse Profile Selected

- Press the Reset button.
- In the Key / Value option list select State.
- In the Start Events text area enter:
cycle_start
- In the End Events text field enter:
cycle_end
- In the Threads text field enter:
sin
- In the Name text field enter:
sine
- Press the Add button.
- Close the dialog.

A state named **sine** has now been defined and occurrences can be displayed in the graphs on the display page.

- Right-click anywhere in the display area and select **Edit Mode** from the context menu or press **Ctrl+E** to enter *edit mode*. (Black dots appear in the background indicating you are in Edit Mode.)

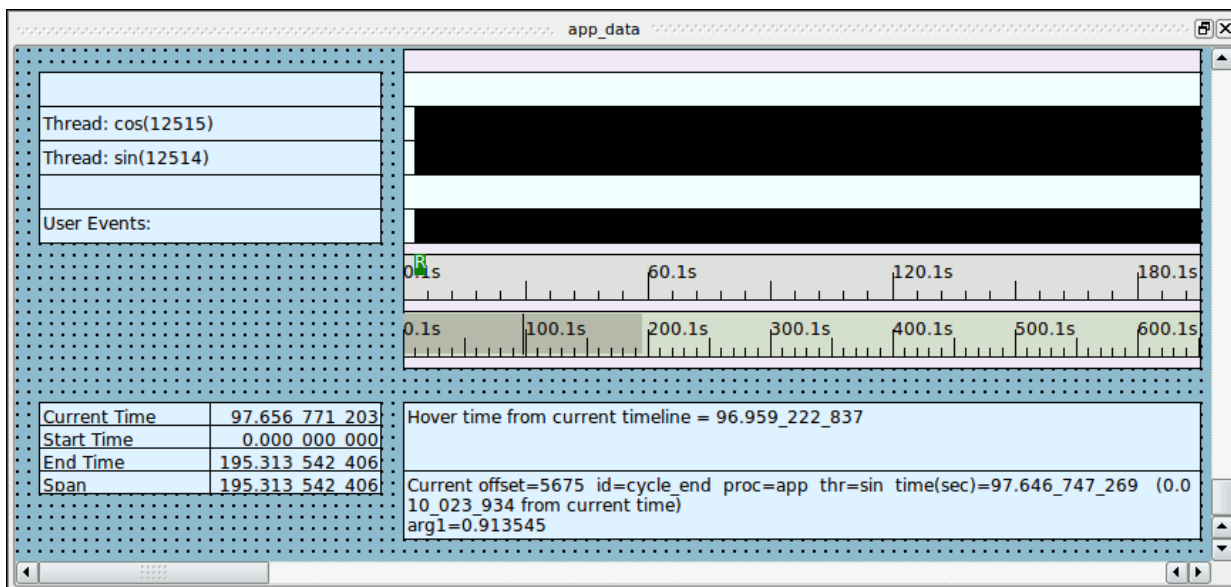


Figure 4-15. Timeline Editing

- Double-click on the graph associated with the row labeled “Thread: sin”. That graph is a row with vertical lines representing events inside the larger graph area, aligned with the label “Thread: sin”.

The Edit State Graph Profile dialog is displayed as shown below:

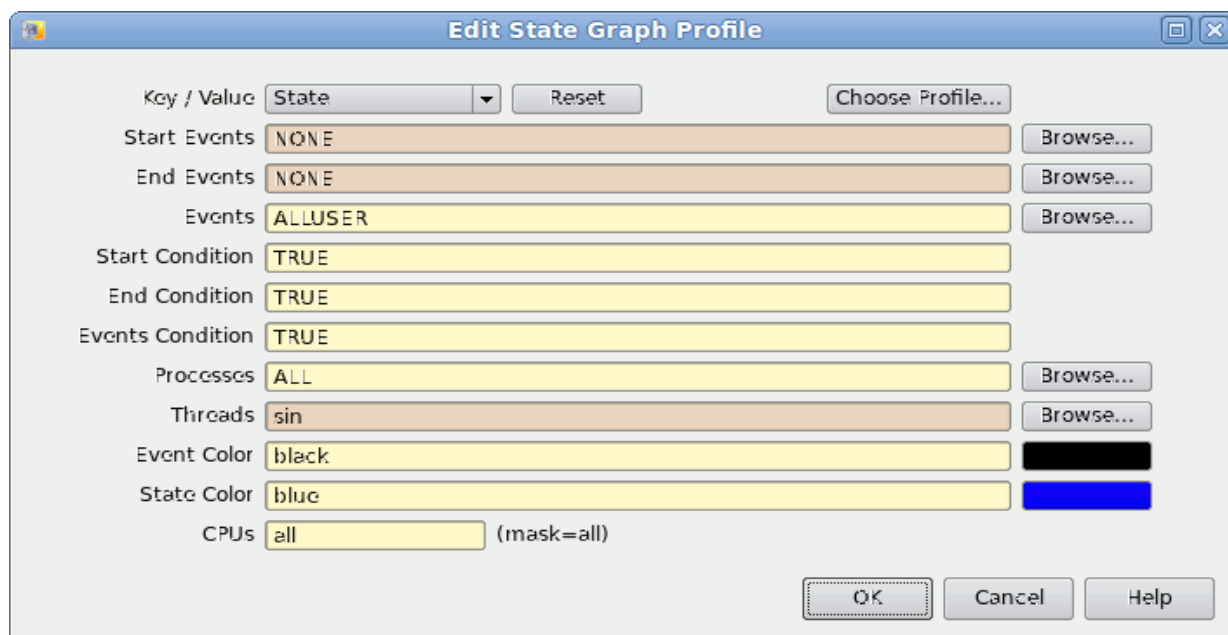
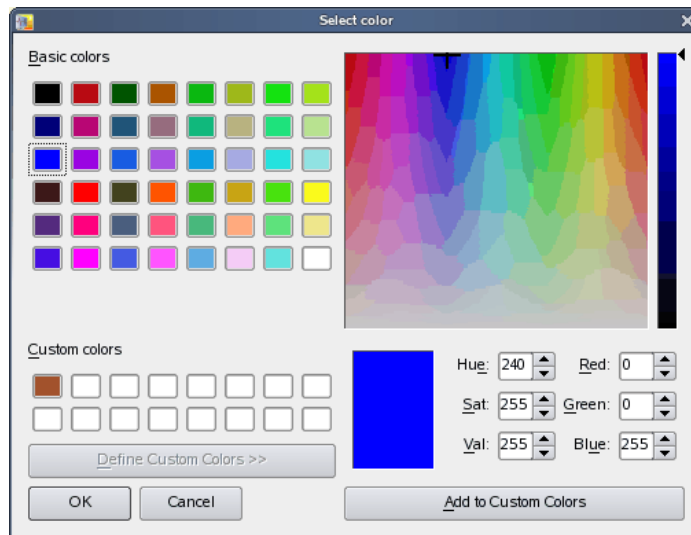


Figure 4-16. Edit State Graph Profile dialog

- From the **Key / Value** option list select **State**.
- Press the **Choose Profile...** button.

The **Choose Profile** dialog is displayed.


- Select the **sine** state from the list.
- Make sure the **Import by reference** checkbox is checked.
- Press **Select**.
- Click on the colored button to the right of the row labeled **State Color**.
The **Select color** dialog is presented.



- Select a pleasing color in the **Select color** dialog and press **OK**.
- Press **OK** in the **Edit State Graph Profile** dialog.
- Right-click anywhere in the display area and select **Edit Mode** from the context menu or press **Ctrl+E** to return to *view mode*.

The graph has now been configured to display the sine state as a solid bar in the lower portion of the state graph. Events will still be displayed as vertical black lines that extend over the entire vertical height of the graph.

It is likely that the display page has not changed in a significant way. This is because the **cycle_start** and **cycle_end** events occur so closely together in time that you cannot distinguish them at the current zoom setting.

- Click in the middle of the state graph.
- Zoom in using the mouse wheel or using the **Zoom In** icon  on the toolbar or the **Down** key until the two events can be distinguished and a state bar is shown.
- You may need to readjust the current timeline as you zoom in.

NOTE

If the **Down** key has no effect, press the **Num Lock** key and try again.

NOTE

The state may vanish at some zoom levels where it is still very small compared to the zoom level's scale. If so, just continue to zoom in and it will reappear.

The figure below displays an instance of the sine state.

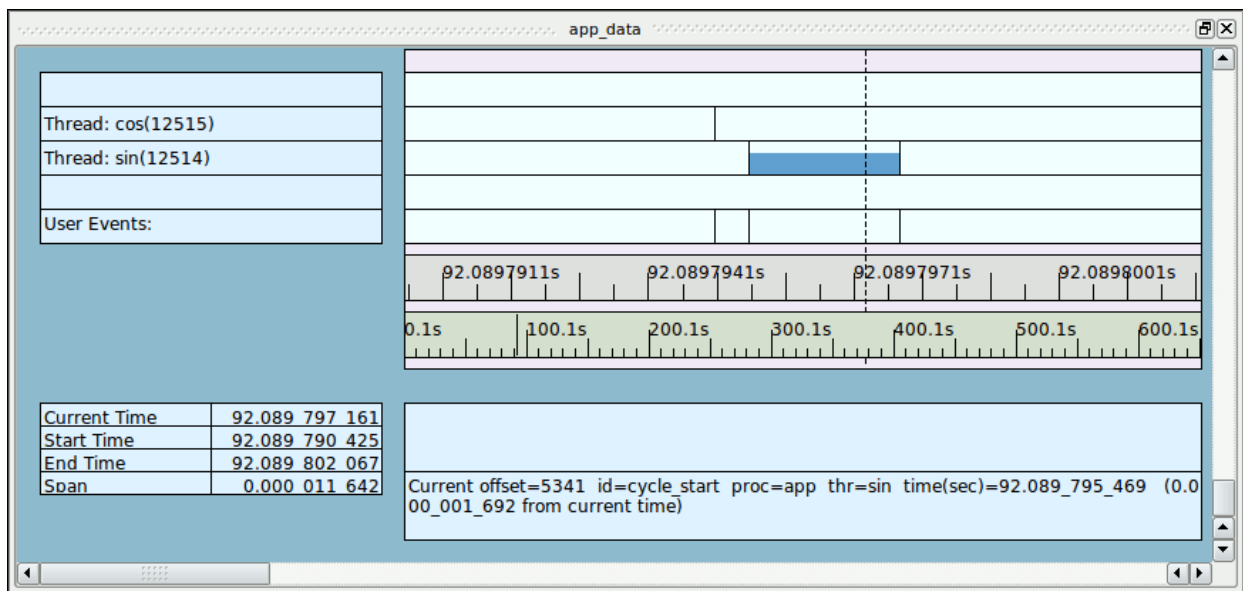


Figure 4-17. Sine State in Timeline

NOTE

If no states are visible, recheck the definition of the sine profile in the Profiles panel as described in “Using States” on page 4-15.

The activity shown in the **COS** row may be different than shown above.

Displaying State Duration

The duration of the most recently completed state can be displayed via a data box.

- Right-click anywhere in the display area on the page labeled `app_data` and select **Edit Mode** from the context menu or press **Ctrl+E** to enter *edit mode*.
- Right-click anywhere in the grid (the area with black dots in the background) and select **Add Data Box** option from the context menu.

The cursor will turn into a **+** character.

- Using the left mouse button, click an empty area in the left-side of the display page on the grid (outside of any currently displayed graph or data box -- i.e. only on an available area whose background shows the dotted grid) and drag the mouse to create the outline of the new data box -- release the mouse button.
- Double-click the data box. The **Edit Data Box Profile** dialog is presented.
- Enter the following into the Output field:

```
format ("cycle = %f ms", state_dur(sine)*1000.0)
```
- Press the **OK** button.
- Right-click anywhere in the display area and select **Edit Mode** from the context menu or press **Ctrl+E** to return to *view mode*.

The data box now displays the length of the most recently completed instance (with respect to the current timeline indicator) of the sine state in milliseconds.

Generating Summary Information

In addition to obtaining detailed information about specific events and states, summary information is easily generated.

- From the **Summary** menu select **Change Summary Profile...**
- From the **Profile Status List** table, select the profile matching the `sine` state.

It is likely that the `sine` profile is already selected. You can verify this by looking at the profile name shown in the **Name** text area near the bottom of the dialog.

- Press the **Summarize All Events** button.

A new page is created displaying the results of the summary.

sine (0 to 12222)

Summarizing from offset 0 thru offset 12222.

State Summary Results
=====

Number of states found: 4074

Maximum state duration: 0.000_015_549 at offset: 3371
 Minimum state duration: 0.000_001_041 at offset: 11526
 Average state duration: 0.000_001_945
 Total of state durations: 0.007_923_114

Number of state gaps found: 4073

Maximum state gap: 0.050_447_180 at offset: 9525
 Minimum state gap: 0.049_703_026 at offset: 4389
 Average state gap: 0.050_071_006
 Total of state gaps: 203.939_206_932

Offset ▲	End Offset	Duration (sec)	Gap (sec)	Event	CPU	Process	Thread	Time (sec)	T ▲
12220	12221	0.000_001_908	0.050_060_162	cycle_start		13581	sin	212.971_266_598	
12217	12218	0.000_002_345	0.050_071_084	cycle_start		13581	sin	212.921_204_091	
12214	12215	0.000_002_041	0.050_084_504	cycle_start		13581	sin	212.871_130_967	
12211	12212	0.000_002_212	0.050_041_545	cycle_start		13581	sin	212.821_044_251	
12208	12209	0.000_001_982	0.050_085_049	cycle_start		13581	sin	212.771_000_724	
12205	12206	0.000_002_143	0.050_083_553	cycle_start		13581	sin	212.720_913_532	
12202	12203	0.000_001_913	0.050_069_862	cycle_start		13581	sin	212.670_828_067	
12199	12200	0.000_002_120	0.050_069_816	cycle_start		13581	sin	212.620_756_085	
12196	12197	0.000_002_396	0.050_070_375	cycle_start		13581	sin	212.570_683_873	
12193	12194	0.000_002_440	0.050_116_065	cycle_start		13581	sin	212.520_611_057	
12190	12191	0.000_001_895	0.050_048_747	cycle_start		13581	sin	212.470_493_098	
12187	12188	0.000_001_987	0.050_055_088	cycle_start		13581	sin	212.420_442_363	
12184	12185	0.000_001_855	0.050_056_909	cycle_start		13581	sin	212.370_385_421	
12181	12182	0.000_001_945	0.050_088_050	cycle_start		13581	sin	212.320_326_566	
12178	12179	0.000_001_887	0.050_054_579	cycle_start		13581	sin	212.270_236_629	
12175	12176	0.000_002_151	0.050_096_382	cycle_start		13581	sin	212.220_179_898	
12172	12173	0.000_002_068	0.050_033_480	cycle_start		13581	sin	212.170_081_448	
12169	12170	0.000_002_066	0.050_086_616	cycle_start		13581	sin	212.120_045_902	
12166	12167	0.000_001_893	0.050_112_274	cycle_start		13581	sin	212.069_957_392	
12163	12164	0.000_001_936	0.050_007_152	cycle_start		13581	sin	212.019_843_183	
12161	12162	0.000_001_189	0.050_086_515	cycle_start		13581	sin	211.969_834_842	

Figure 4-18. Summary Results Page

The summary results page provides a number of columns of information including the state's starting and ending offsets, the state's duration, and the gap between a state and its most recent previous occurrence. You can click on the column headers to control how the list is sorted.

Double-clicking on a row in the list positions the current timeline to the beginning of that instance of the state and creates a tag at that position.

To go to the instance of the longest state duration, do the following:

- Click on the **Duration** header to select duration as the sort key.
- Click the **Duration** header until the sort order is largest to smallest. It is possible that the maroon chevron in the header may not properly indicate the sort direction, but it will be obvious from the list of values.
- The instance of the state with the longest duration is shown in the top row. Click on that row.

The current timeline is moved to that instance of the state, as shown in the **Events** and **Timeline** panels.

The minimum and maximum state occurrences are often of interest. However, a graphical display of state durations can be more enlightening.

- From the Summary menu select the Graph State Durations... option.
- Change the standard deviation value in the dialog to 0.
- Press the OK button.

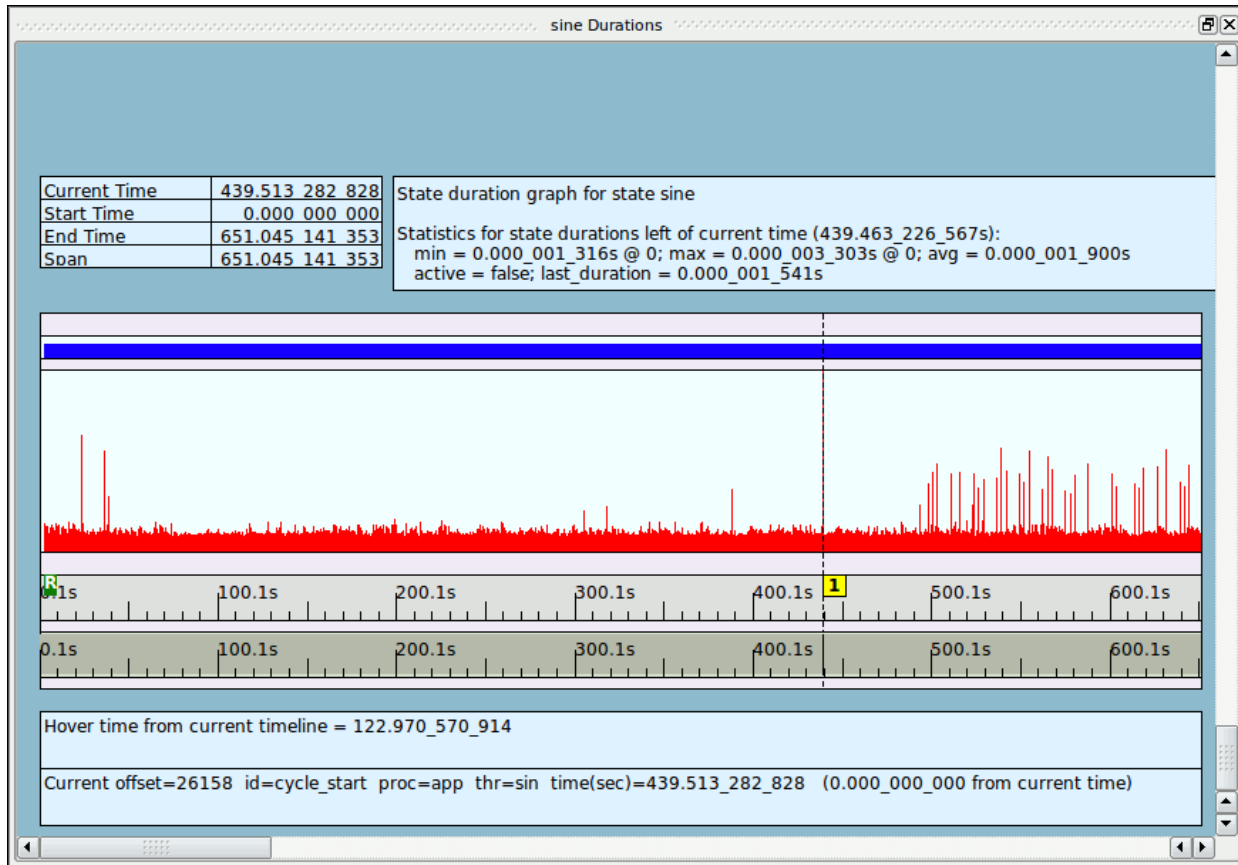


Figure 4-19. Summary Graph

A new page is created with a summary graph and a textual description of the instances of the state.

The row with blue shown indicates individual instances of the state. If the blue bar appears to be a single bar, zoom in until individual instances can be seen.

- Zoom all the way out by pressing **Alt+Up**.

A data graph is shown in the tall row beneath the row with blue state indicators.

Each red line indicates the duration of an instance of the state.

Sometimes a single occurrence of the state may be much longer than most occurrences. In such cases, the detail is obscured.

We can rebuild the page using a different standard deviation index.

- Right-click the tab that contains the summary and click **Delete Current Page**.
- From the **Summary** menu, select **Graph State Durations** and supply a value of 1 to the standard deviation request dialog.

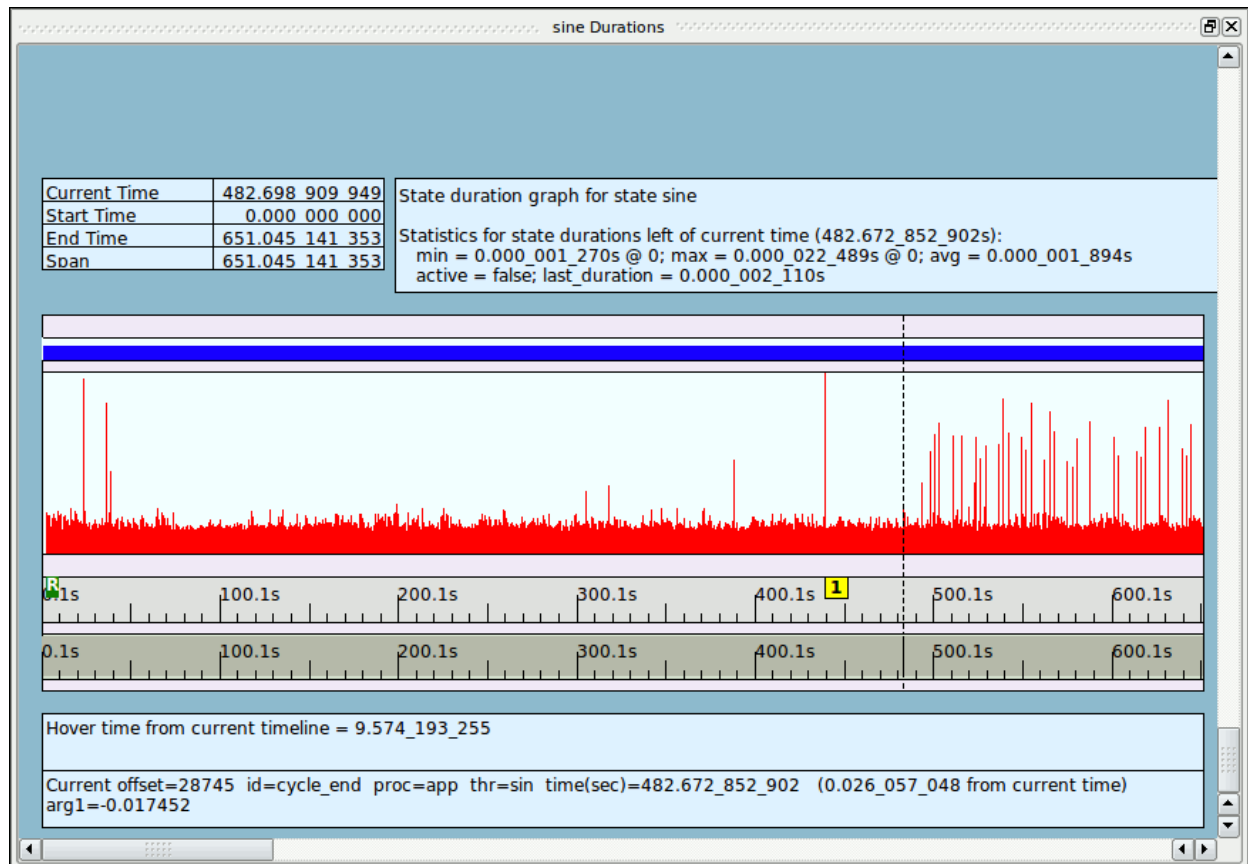


Figure 4-20. State Durations Graph Modified

The graph now shows more detail. The current timeline in the data graph is linked to the current timeline in all timelines and the **Events** panel. Clicking anywhere in the graph will move the current timeline in all such panels.

NOTE

Depending on various factors, selecting a standard deviation of 1 may actually have the opposite effect, obscuring detail even further. Experiment with standard deviation factors using the procedure above until you find one that is most useful (which may in fact be a factor of zero).

Defining a Data Graph

- Raise the `app_data` timeline page by clicking on its tab.
- Remove the **Events** panel by clicking the close box at the upper right-most portion of the panel's title bar.

The area that contains the individual rows with events is called the graph container. It has a pink background which you can see at the very top and the very bottom.

- Right-click anywhere in the display panel labeled `app_data` and select **Edit Mode** from the context menu or press **Ctrl+E** to enter *edit mode*.
- Click on the middle of the top horizontal line of the graph container.
- Move the mouse cursor so that it hovers over the middle of the top horizontal line of the column.
- When the cursor changes to two arrows pointing up and down, click and drag the upper boundary of the graph container upward to make space for the data graph.

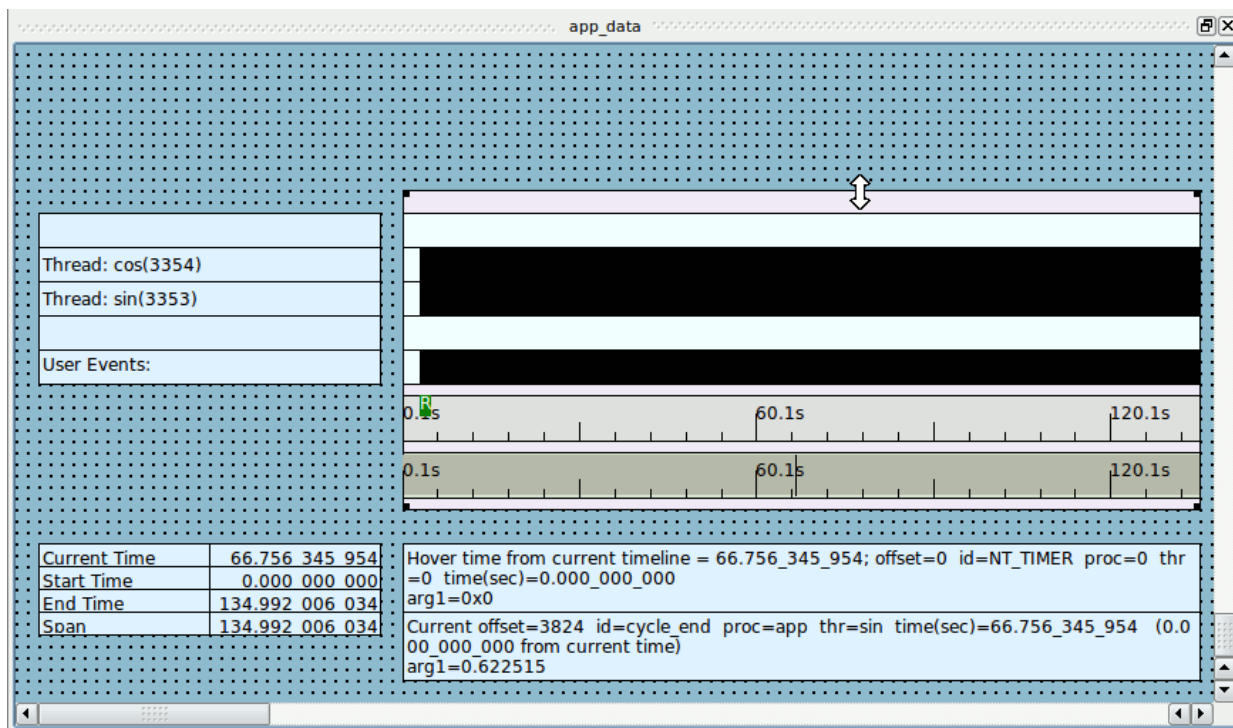


Figure 4-21. Timeline in Edit Mode

- Release the mouse button when sufficient space has been made (approximately an inch or more vertically).
- Click on the top horizontal line of the graph container.

- Right-click inside the graph container and select **Add to Selected Graph Container** from the context menu and select **Data Graph** from the sub-menu.

The cursor changes to a block plus sign.

- Click in the space created by the previous steps.

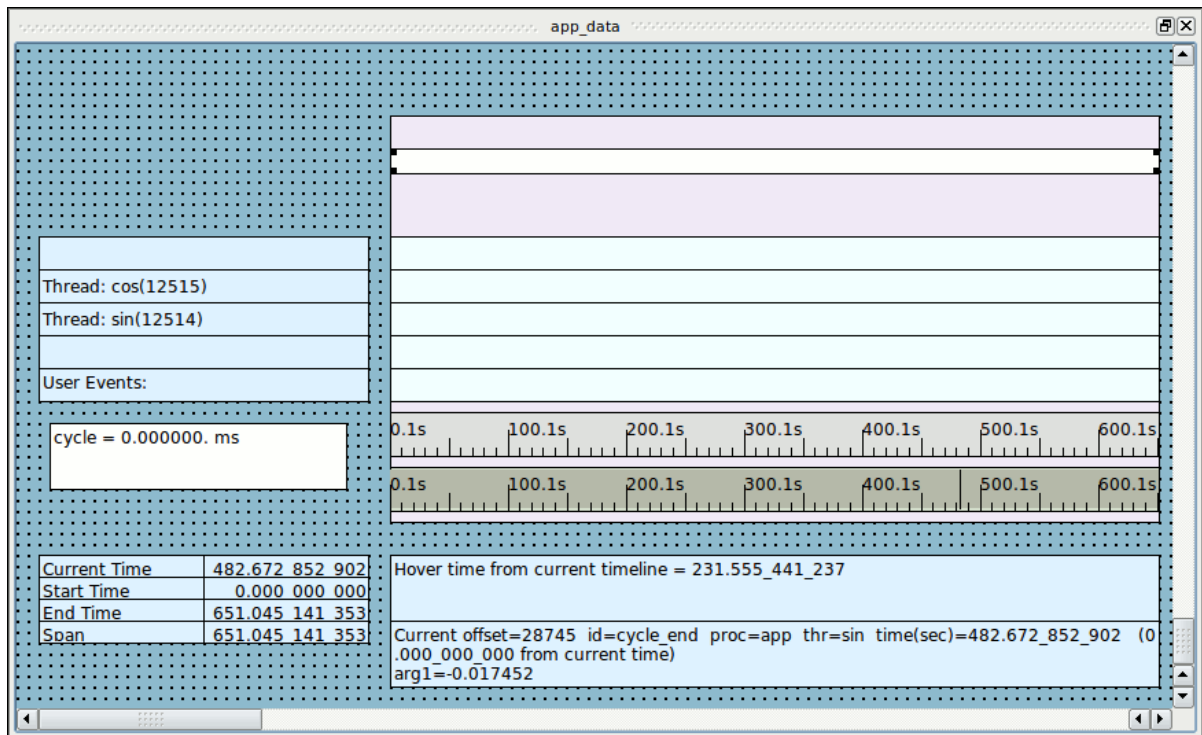


Figure 4-22. Adding a Data Graph

- Click inside the data graph you just inserted.
- Drag the top border to the top of the data graph and the bottom border to the bottom of the data graph so that it fills most of the available space in the top portion of the graph container.
- Double-click in the middle of the data graph.

The Edit Data Graph Profile dialog is presented.

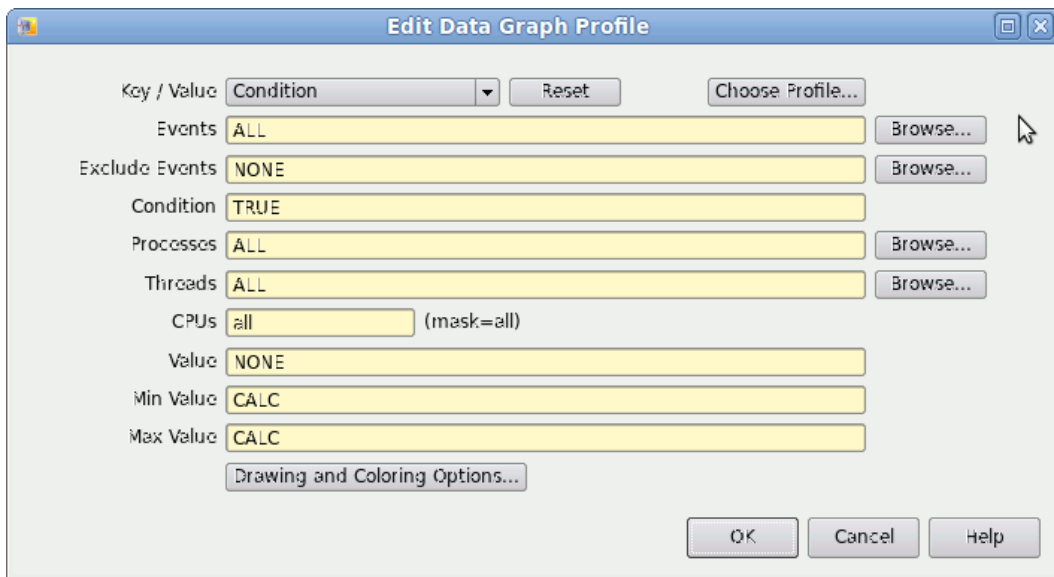


Figure 4-23. Edit Data Graph Profile Dialog

- In the Events text field enter:
cycle_end
- In the Value text field enter:
arg1_dbl
- Press OK to close the Edit Data Graph Profile dialog.
- Right-click inside the data graph and select Adjust Colors in Selected from the context menu and select Data Graph Value Color... from the sub-menu.
- Select a pleasing color from the Select color dialog for the data graph. Press OK to close the Select color dialog.
- Right-click anywhere in the display panel labeled app_data and select Edit Mode from the context menu or press Ctrl+E to return to view mode.
- Zoom the display to see the sine wave generated by the program.

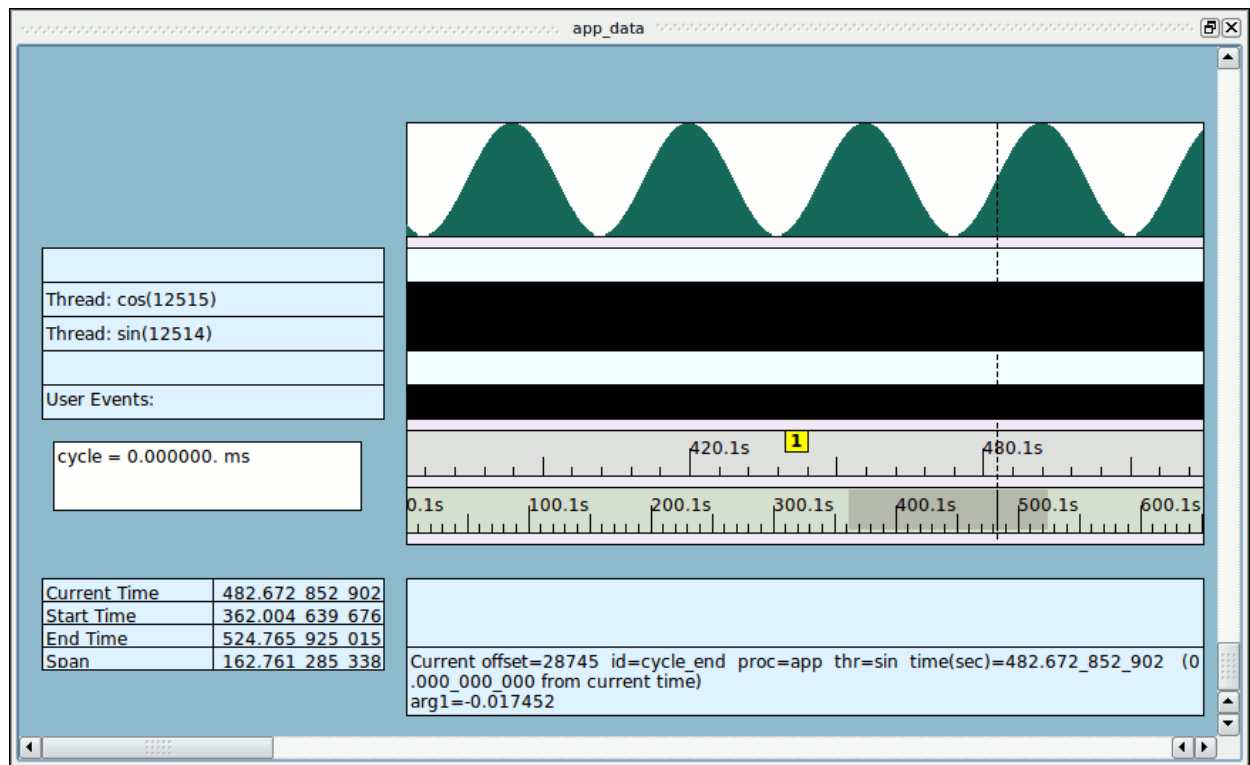


Figure 4-24. Display Page with Data Graph

Kernel Tracing

Kernel tracing provides amazing insight into the activities of the system and how applications interact with each other and the kernel.

In order to use kernel tracing you must be running a trace-enabled kernel.

Kernel names ending in **-trace** and **-debug** have kernel tracing enabled. You may check to see which kernel is running by using the following command:

```
uname -r
```

If you are not running a trace-enabled kernel, reboot now and select it from the GRUB menu at boot time. If you are unable to reboot your system at this time, please follow the beginning of the tutorial to load the pre-recorded kernel data as instructed.

- Click on the first tab of the NightTrace main window.
- Ensure the user daemon is halted; if not click on it in the Daemons panel and press the **Halt** button (which will only be sensitized if the daemon is still running).
- Select the `app_data` segment in the **Trace Segments** panel.
- Press the **Close Trace Data** button in the **Trace Segments** panel.

NightTrace will pop up a dialog warning you that the trace data has not been saved and will be discarded; the data does not need to be saved for this tutorial.

Obtaining Kernel Trace Data

If you are not running a trace-enabled kernel, skip this section and refer to the section “Using Prerecorded Kernel Data” on page 4-30.

- Double-click on the `kernel_trace_to_gui` entry in the **Daemons** panel on the first page of the NightTrace main window.

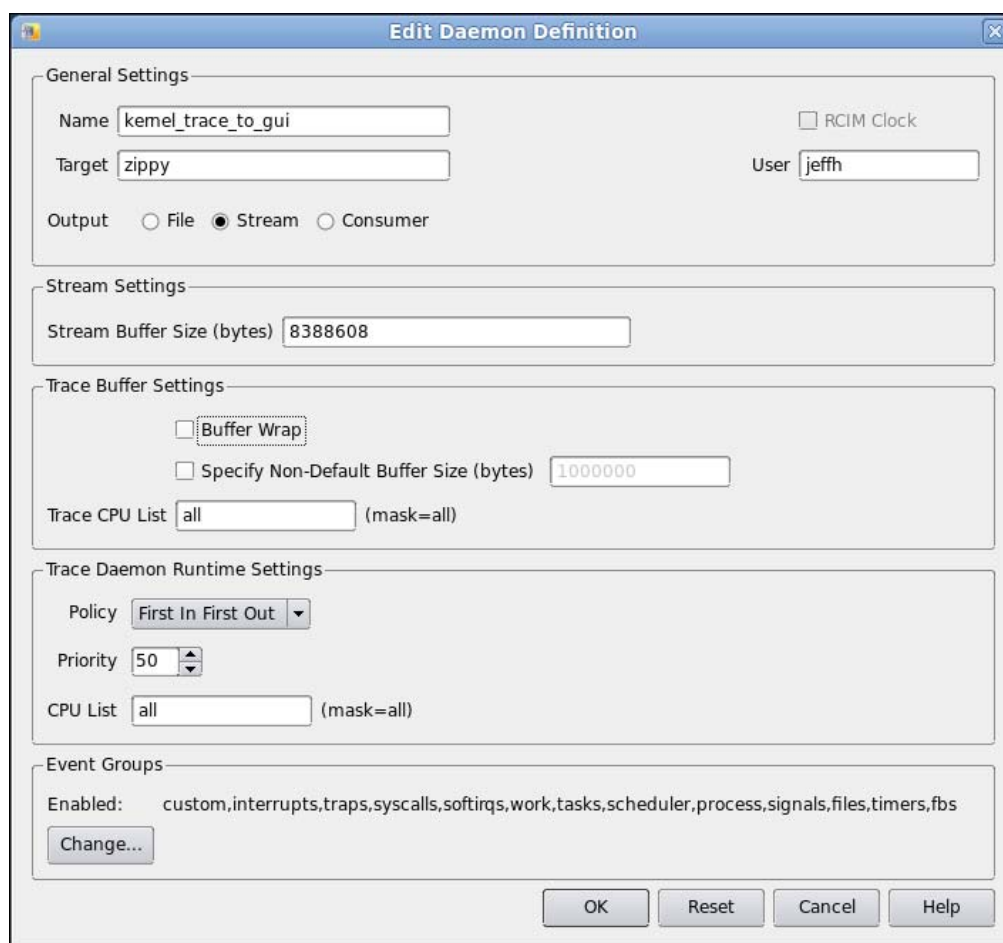


Figure 4-25. Edit Daemon Definition Dialog

NOTE

The Trace Buffer Settings and Event Groups may look different depending on which version of the RedHawk Kernel you are running; regardless, all versions have a Buffer Wrap setting.

We don't actually need to change anything at this time; this step was just for informational purposes to show you the attributes of kernel tracing you can control.

- Press OK.

Depending on system activity, huge amounts of kernel trace data can be generated in a relatively short period of time. Since operation of NightTrace is likely a new experience for many users, we will restrict the data flow to a manageable size for new users.

- Ensure that `kernel_trace_to_gui` is selected in the Daemon Control Area.
- Press the Launch button.

- Press the **Resume** button.
- Watch the daemon statistics in the Daemon Control Area; once at least 200,000 events are present in the **Logged** column, press the **Halt** button.

Skip the next section and jump directly to “Analyzing Kernel Data” on page 4-31.

Using Prerecorded Kernel Data

This section is provided only for those using the tutorial that have not booted a trace-enabled kernel.

If you collected live kernel trace data in the preceding section, skip to “Analyzing Kernel Data” on page 4-31.

The NightStar RT **tutorial** directory contains some pre-recorded kernel data which can be used in the section titled “Analyzing Kernel Data” on page 4-31.

- From the **File** menu in the NightTrace main window, select **Open Files...**
- In the file dialog’s **Selection** text field, type the following:

```
/usr/lib/NightStar/tutorial/.kernel.nts
```
- Press the **Enter** button.

Proceed to the next section.

Analyzing Kernel Data

NightTrace automatically generates a default kernel display page that is customized to the system from which the kernel data was captured.

- Click on the tab created in the NightTrace main window to display the newly-created kernel display page. The tab will have a name like <machine_name> Timeline.

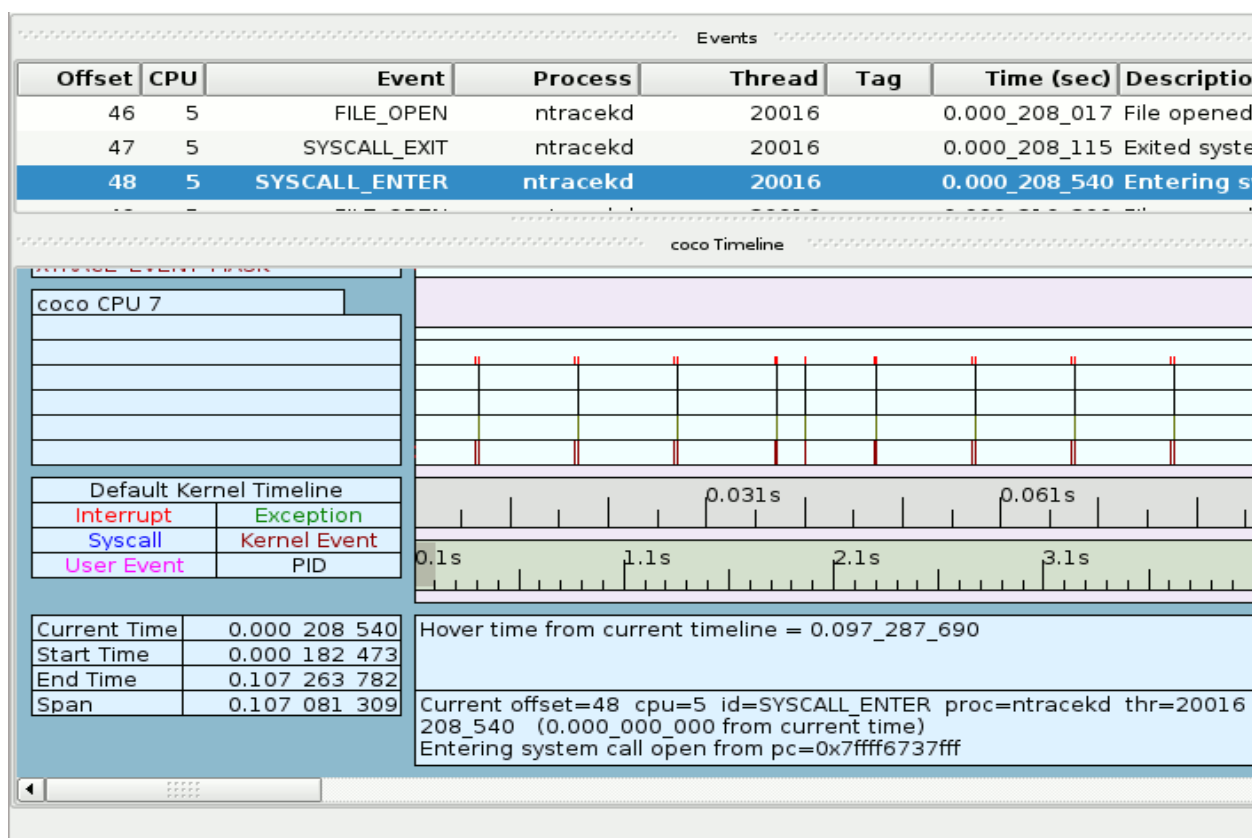


Figure 4-26. Kernel Display Page

NOTE

Your timelines may look significantly different if you have a different number of CPUs. Additional system activity can make the display vary as well. Do not be concerned about such differences at this step.

- Press Alt+Right to move to the end of the data set.
- Click in an active area and zoom in until detail can be seen.

For each CPU, the following information is displayed:

- interrupt activity (in red)

- machine exception activity (in green)
- system call activity (in blue)
- per-process CPU utilization (shown in a variety of colors)
- detailed kernel events (in dark red)

The data boxes on the left hand side of the display page are color coded to match the information they describe. Their contents change dynamically based on the position of the current timeline.

- Press **Ctrl+F** to bring up the **Profiles** dialog.
- Click the **Reset** button to the right of the **Key/Value** selection area.
- Press the **Browse...** button to the right of the **Processes** text field.

The **Select Processes** dialog is presented.

- Select the **app** process from the list of known processes.
- Press the **Select** button to close the **Select Processes** dialog.
- From the **Key / Value** option list select **System Call Enter Events**.

The **Select System Calls** dialog is presented.

- Select **nanosleep** from the list of system calls shown.

NOTE

On some distributions, such as Ubuntu 20.04, the name of the system call isn't "**nanosleep**", but "**clock_nanosleep**". If you fail to find **nanosleep** events, substitute **clock_nanosleep** for **nanosleep** in the following instructions.

- Press the **Select** button to close the **Select System Calls** dialog.

- Change the list of events in the **Events** text field to include only SYSCALL_RESUME. The Search/Summarize window should look like the following:

- Press the **Search Forward** button.

A new profile based on the information entered is added to the **Profile Status List** and the current timeline is changed to the next occurrence of a resumption of a suspended nanosleep system call in process **app**.

NOTE

If NightTrace fails to find an occurrence matching the sort criteria just entered, recheck the search criteria. It is likely that you may have skipped pressing the **Reset** button in the steps above. Ensure that the **Threads** text field indicates **ALL** and not **sin**.

- Click on the tab corresponding to the kernel display page.
- Click somewhere within the page in the background area to regain focus, (not within the grid, as that would change the current time indicator).

- Zoom in until detailed information is visible, similar to what is shown below:

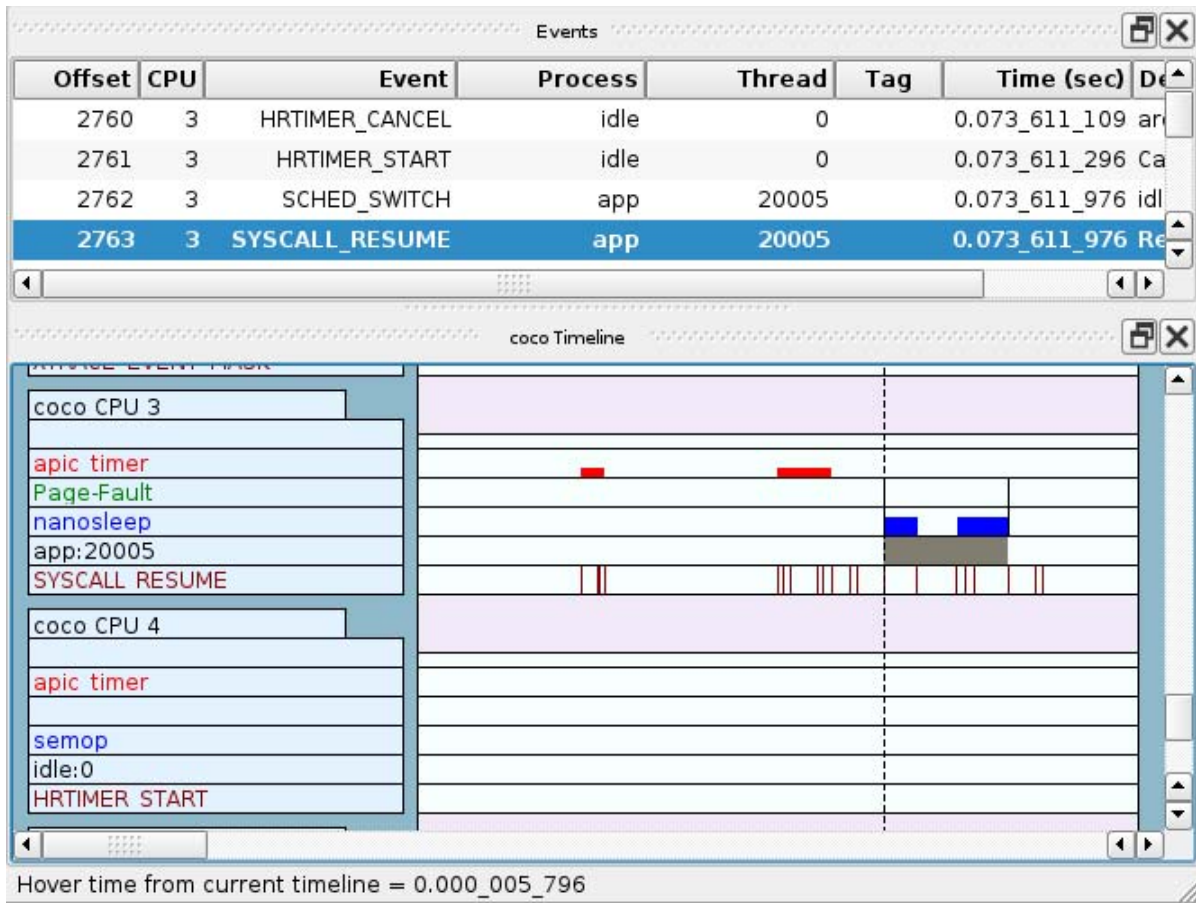


Figure 4-27. System Call Resume for Nanosleep

NOTE

Your timeline may look significantly different if you have a different number of CPUs. Additional system activity can make the display vary as well. Repeat the search a few times to find an occurrence that looks similar to the row that indicates the **app** process. You can repeat the last search by pressing the forward search icon on the toolbar or by pressing the **Ctrl+G**.

The red bar to the left of the current timeline indicates that an interrupt occurred. In this case, it was a `local_timer` interrupt.

The tall vertical black line spanning the system call and exception rows represents a context switch. The current timeline (dashed line spanning the entire rectangular display area) is likely overlaid with the context switch line at this zoom setting.

- Select the highlighted event in the **Events** panel. This is the event at the current timeline, which should be `SYSCALL_RESUME`.

The **Description** column in the **Events** panel for the currently highlighted event describes the event in more detail:

Events							
Offset	CPU	Event	Process	Thread	Tag	Time (sec)	Description
25617	0	PROCESS	idle	0		9.404_318_205	Wake process app (12383)
25618	0	IRQ_EXIT	idle	0		9.404_319_791	Interrupt handling for local_timer (IRQ=239) exited
25619	0	SCHEDCHANGE	app	main		9.404_321_346	idle switched out (runnable); app (12383) switched in
25620	0	SYSCALL_RESUME	app	main		9.404_321_347	Resuming system call nanosleep
25621	0	SYSCALL_EXIT	app	main		9.404_322_904	Exited system call nanosleep
25622	0	SYSCALL_ENTRY	app	main		9.404_346_638	Entering system call semop from pc=0x2ac4e21e6297
25623	0	PROCESS	app	main		9.404_350_035	Wake process app (12515)

Figure 4-28. Events Panel after Search

- While the current timeline is at the `SYSCALL_RESUME` event, press the Up key until you see another event for the same CPU.

The current timeline is changed to the preceding event and the text description indicates a context switch with text similar to the following:

```
idle switched out (runnable); app (12383) switched in
```

The blue bar represents system call activity. The data box to the left will describe the system call name for the system call at or to the left of the current timeline.

- Press **Ctrl+G** to go back to the `SYSCALL_RESUME` event.

In the instance shown in the screen shot above, shortly after the main thread returns from `nanosleep`, the main thread is exiting the `nanosleep` call on line 105 of **app.c**. It will then soon enter a `semop` system call to execute the `semop` library call on line 108.

NOTE

On some systems, the system call may be described as `ipc` or `semtimedop` instead of `semop`.

Mixing Kernel and User Data

If not running a trace-enabled kernel, skip this section and proceed to “Using the NightTrace Analysis API” on page 4-39.

- Click on the first tab of the NightTrace main window.
- Ensure the kernel daemon is halted by pressing the **Halt** button if it is sensitized (it should have been halted in a previous step).
- In the **Trace Segments** panel select the `kernel_trace_to_gui` segment and select the **Close Trace Data** menu option of the context menu.
- Select both daemons in the **Daemons** panel using **Click** and **Shift+Click** mouse and keyboard actions.

- Before proceeding, make absolutely sure both daemons are selected. If not, correct that.
- Press the **Launch** button.

Read the next four steps before proceeding, then execute them in order.

- Press the **Resume** button.
- Wait until the total of the **Buffer** and **Logged** cells reach ~2000 events. If you have a high number of CPUs on your system, you may be collecting massive amounts of kernel data. You can proceed to the next step before reaching the 2000 user events if that is the case.
- Press the **Flush** button.
- Press the **Halt** button.

Data from both the user application and the kernel have been captured and brought into NightTrace.

- From the **Summary** menu select **Change Summary Profile**.
- From the **Profile Status List** at the top of the page, select the **sine** profile.
- Press the **Summarize All Events** button.

The last action caused a new page to be created containing a summary of the **sine** state defined in “Generating Summary Information” on page 4-20.

- Click on the **Duration** header until it is sorted in descending order (you may have to click more than once).
- Click the cell containing the value of the duration in the first row.
- Click on the tab corresponding to the kernel display page.
- Click somewhere within the page to regain focus for the timelines, but only on the background -- clicking inside the grid would change the current timeline.
- Zoom in or out as required until you can clearly see the detail relating to the **sine** thread’s cycle.

In the graphic shown below, the **sine** thread was preempted by a kernel processing of an **rcim** interrupt.

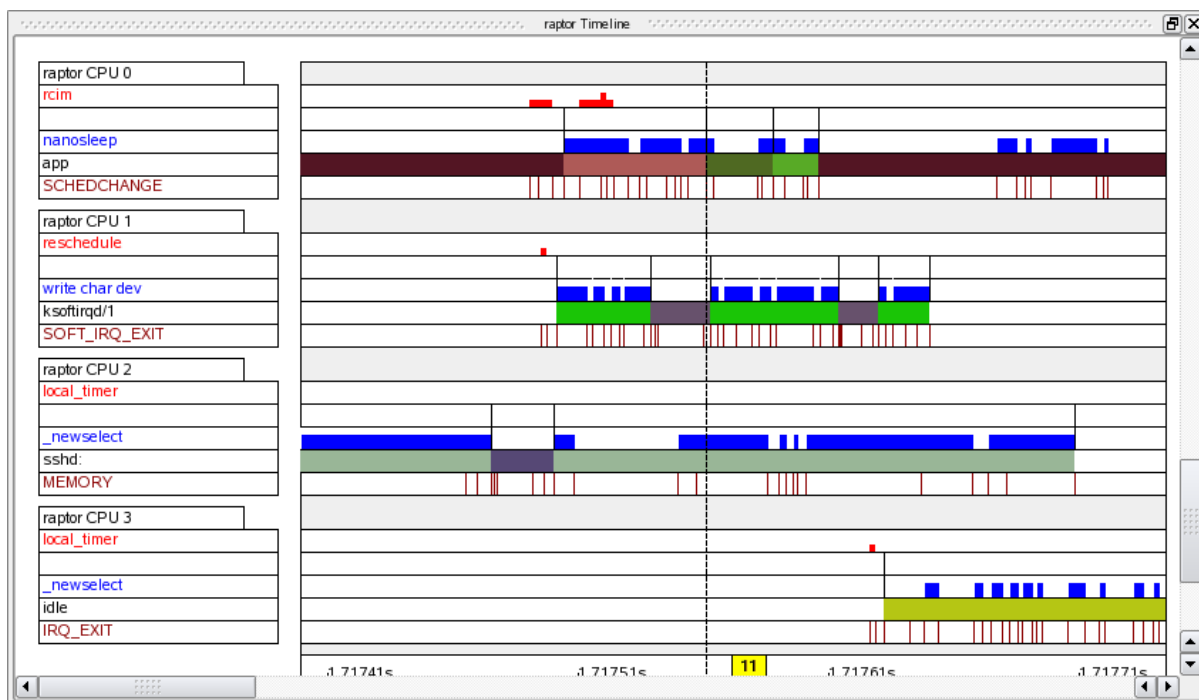


Figure 4-29. Longest Instance of State

The reason for the extended cycle in your trace data may be due to other circumstances.

- Was the `sine_thread()` preempted by another process?
- Did an interrupt occur during the cycle?
- Was there significant activity on the hyper-threaded sibling CPU which stole cycles from the CPU where the `sine` thread was executing?
- Did the application get a page fault or other machine exception?
- Did activity on a hyper-threaded sibling CPU interfere with the CPU where `app` is executing?

We have not yet tuned our application for real-time, so there may be many reasons for the extended cycle.

Some of these circumstances are discussed in more detail in “Overrun Detection and System Tuning” on page 7-10.

Machine exceptions include information detailing the type of exception, the faulting address (when applicable), and the PC at which the exception occurred.

- Type **Ctrl+F** while the kernel display page is selected.
- Press the **Reset** button.
- From the **Key / Value** option list select **Exception Enter Events**.
- Select **Page-Fault** from the list of exceptions.

- Press the **Select** button.
- Press the **Search/Forward** button.

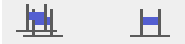
If a page fault is located, the current timeline is moved to the next occurrence of a page fault. The text area at the top of the kernel display page includes detailed information about the exception, including the PC at which the fault occurred and the faulting address.

Using the NightTrace Analysis API

NightTrace provides a powerful API which allows user applications to analyze pre-recorded trace data or to monitor and analyze live trace data.

You can write programs that define states and conditions and process events as they occur.

In this tutorial, we will instruct NightTrace to build an API program automatically.

- Click on the either of the two **Profiles** toolbar icons. 
- From the **Profile Status List** select the `sine` profile.
- Select **Search Forward**.
- From the **Profiles** menu select **Export to API Source...**

The following dialog is displayed:

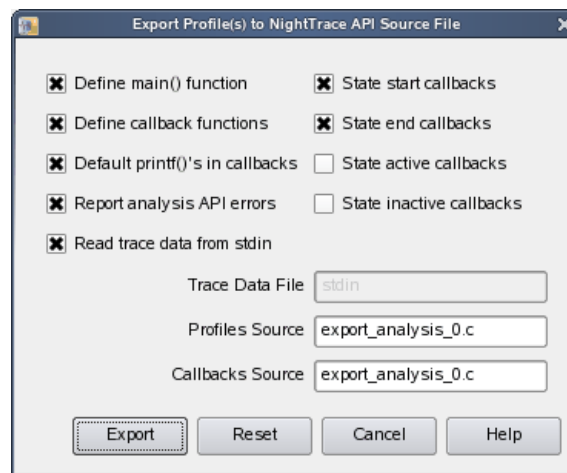


Figure 4-30. Export Profiles to NightTrace API Source File dialog

- Clear the **State start callbacks** checkbox.
- Press the **Export** button.
- From the **NightTrace** menu select the **Exit Immediately** menu item to exit NightTrace.

NightTrace has created an API program which listens for occurrences of the state defined by the **sine** profile and prints out some information for each instance.

- Build the API program using the following command:

```
cc -g export_analysis_0.c -lntrace_analysis
```

This program expects to consume live trace data.

You can configure a user daemon with the NightTrace GUI and have NightTrace launch the analysis program automatically.

Alternatively, you can use the command line user daemon program **ntraceud** to achieve the same effect.

- Type the following command:

```
ntraceud --stream --join /tmp/data | ./a.out
```

This command instructs **ntraceud** to start capturing trace data from a running application which is using the file **/tmp/data** as a handle. The **--stream** option indicates that instead of logging the data to the named file, it should be sent to **stdout**.

The application program may not immediately begin generating output because the data rate is fairly low and buffering is involved.

- To flush the current buffers for immediate consumption by the application, issue the following command in a different terminal session:

```
ntraceud --flush /tmp/data
```

NOTE

You may need to repeat that command several times over a period of a few seconds to allow the data to pass through system buffers.

Data similar to the following will appear on **stdout** in the terminal session where the analysis program was launched:

```
sine (end)offset 665 occur 333 code 2 pid 3399 time 16.628649 duration 0.000003
sine (end)offset 667 occur 334 code 2 pid 3399 time 16.678631 duration 0.000003
sine (end)offset 669 occur 335 code 2 pid 3399 time 16.728655 duration 0.000003
sine (end)offset 671 occur 336 code 2 pid 3399 time 16.778676 duration 0.000003
sine (end)offset 673 occur 337 code 2 pid 3399 time 16.828693 duration 0.000003
sine (end)offset 675 occur 338 code 2 pid 3399 time 16.878716 duration 0.000004
sine (end)offset 677 occur 339 code 2 pid 3399 time 16.928745 duration 0.000003
sine (end)offset 679 occur 340 code 2 pid 3399 time 16.978760 duration 0.000003
sine (end)offset 681 occur 341 code 2 pid 3399 time 17.028779 duration 0.000003
```

- Issue the following command to terminate the daemon:

```
ntraceud --quit-now /tmp/data
```

If you are not running a trace-enabled kernel, skip the remaining portion of this section and proceed to “**Conclusion - NightTrace**” on page 4-41. You can check to see which kernel you are running by issuing the following command: **uname -r**. Trace-enabled kernels normally end in **-trace**, although you may have built a custom kernel with tracing enabled and did not follow the naming convention.

Several sample API programs are provided with NightTrace.

- Type the following commands to build the watchdog example program:

```
cc -g -o watchdog \
/usr/lib/NightTrace/examples/c/analysis/watchdog.c \
-lntrace_analysis
```

This simple sample program watches for context switches on a specific CPU and prints the name of the process that is switching in.

This time the **ntracekd** kernel daemon will be used to capture 5 seconds of kernel data and stream the output to the **watchdog** program.

- Issue the following command:

```
ntracekd --stream --wait=5 /tmp/x | ./watchdog 1
```

The program will eventually generate output similar to the following:

```
context switch: 4.979350027      4 ksoftirqd/0
context switch: 4.979358275    2846 X
context switch: 4.983906074      0 idle
context switch: 4.983960385    2846 X
context switch: 4.994892976    3167 firefox-bin
context switch: 4.994989171    4492 ntfilterl
context switch: 4.995070736    4489 watchdog
context switch: 4.995092415    4492 ntfilterl
context switch: 4.995173214    4489 watchdog
context switch: 4.995188096    4492 ntfilterl
context switch: 4.995256175    4489 watchdog
context switch: 4.995270824    4492 ntfilterl
context switch: 4.995332743    4489 watchdog
context switch: 4.995355783    2846 X
context switch: 5.000351519      4 ksoftirqd/0
context switch: 5.000360675    2846 X
```

Shutting Down

- From the File menu of NightTrace, select **Exit Immediately** to terminate the NightTrace session.

Conclusion - NightTrace

This concludes the NightTrace portion of the NightStar RT Tutorial.

Using NightProbe

NightProbe is a graphical tool for viewing and modifying data from independently executing programs as well as recording data for subsequent analysis.

This chapter assumes you have already built the **app** program. If you have not built the program, do so using the instructions in “Building the Program” on page 1-4.

- Ensure no previous instances of app are running by issuing the following command:

```
killall -9 app
```

- Start the application afresh via the following command before proceeding:

```
./app &
```

Invoking NightProbe

Programs to be probed do not need to be instrumented with any special API calls. However, in order for NightProbe to refer to symbolic variable names, the program should be compiled with debug information (typically the **-g** compilation option).

NightProbe takes advantage of significant performance capabilities of the RedHawk kernel, eliminating intrusion on the process by sampling and modifying variables in other programs using direct memory fetches and stores.

- To invoke NightProbe from any NightStar Tool that is currently running, go to the **Tools** menu and select **NightProbe Monitor**. You may also invoke NightProbe by using the NightProbe desktop icon or by typing the following command at a shell command prompt:

```
nprobe &
```

The NightProbe main window is displayed.

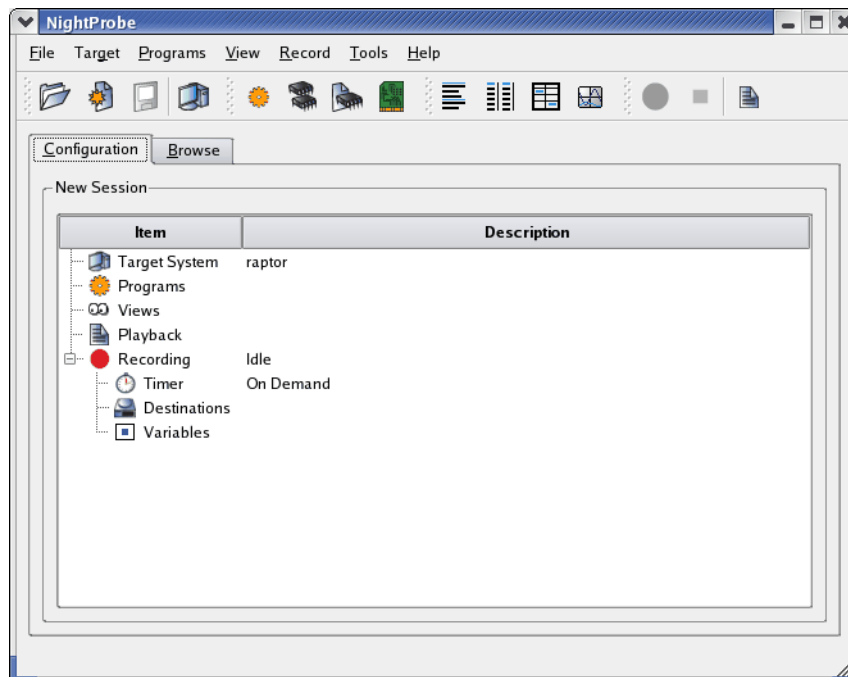


Figure 5-1. NightProbe Main Window

Selecting Processes

NightProbe has the ability to probe several kinds of resources, including programs, shared memory segments, memory mapped entities, and PCI devices.

- On the Configuration page right-click the Programs icon and select the Program... menu option.

The Program Selection dialog is presented:

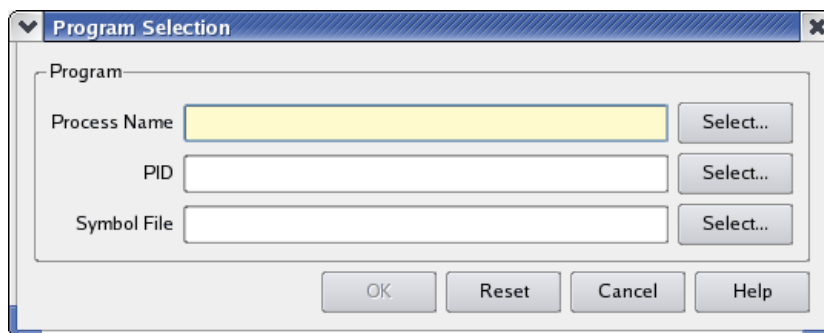


Figure 5-2. Program Selection Dialog

- Press the Select... button to the right of the PID field

The Process Selection dialog appears.

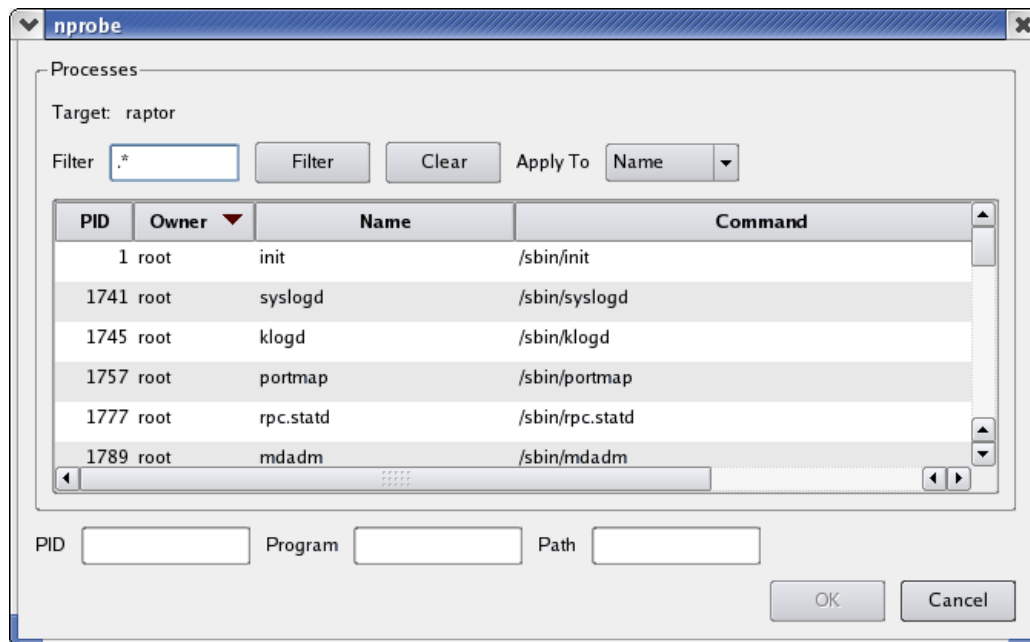


Figure 5-3. Process Selection Dialog

- Enter ^app in the Filter field and press the Enter key.

The list is filtered to only those process whose name starts with **app**.

- Locate the program in the table and select it. (If there is only one matching program it will already be selected.)
- Press Enter again to close the dialog.

The process ID associated with the **app** program is placed in the PID text field and the Process Name and Symbol File text fields are updated accordingly.

- Press OK to close the dialog.

The **app** program is added to the list of resources to be probed as is shown under the Programs item in the Configuration page and to the tree inside the Browse tab, which has automatically been raised.

Viewing Live Data

- Ensure that the Browse tab is raised; click on it if it is not.

The Live Browser is displayed inside the Browse tab.

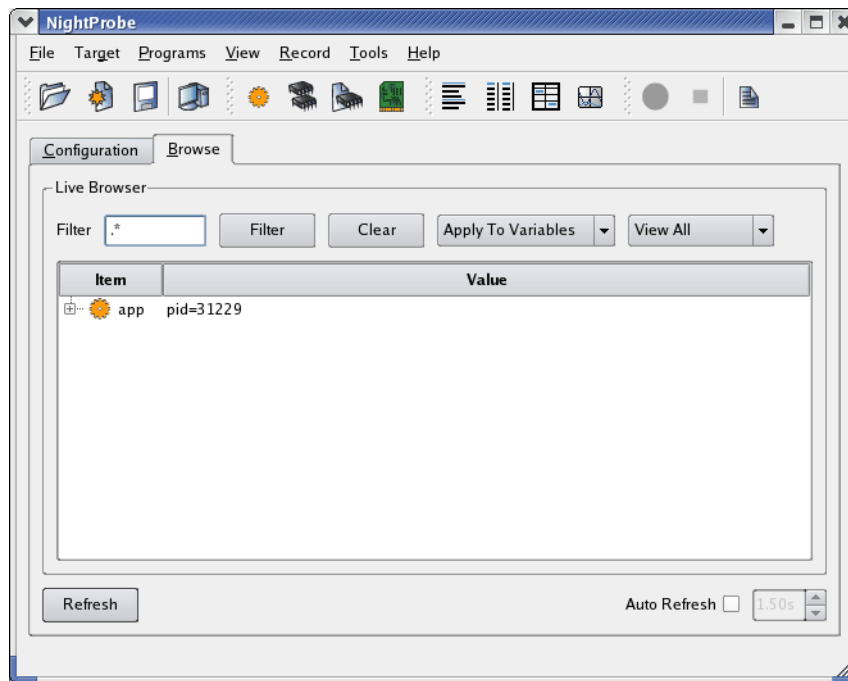


Figure 5-4. NightProbe Browse Panel

The **Browse** page serves two purposes. It allows you to browse your program to select variables of interest for recording or for viewing with alternative **View** panels.

It also provides you instant viewing of variables using the tree shown directly within the **Browse** page.

- Expand the **app** entry in the tree.

The items under a program's icon include all global variables as well as any nested scopes such as Ada packages, or functions that contain static data items.


Each variable item has an icon which indicates whether the variable is a scalar, a pointer, or a composite item such as an array or structure.

The data variable is a composite object and can be expanded.

- Expand the data variable.

Item	Value
app	pid=4640
f(x) add_link	
head	0x0804b220
data	
data[0]	
sema	294919
tail	0x0804e200
rate	50000000
ptrs	
state	run

Figure 5-5. Expanded Data Item

The downward pointing chevron  is the array subscript expansion icon. By clicking the icon, an additional component of the array is shown.

- Click the array expansion icon so that data[1] is shown
- Expand both structures displayed, data[0] and data[1].

In the **Browse** page, the current value of all variables shown in the tree is displayed whenever you press the **Refresh** button at the bottom of the page, and whenever an automatic refresh occurs as controlled by the **Auto Refresh** checkbox.

- Click the **Auto Refresh** checkbox.

This causes the display to automatically refresh at the rate shown in the spinbox to the right of the **Auto Refresh** checkbox.

Note the values of the count, angle, and value components of each component of the data array changing.

Modifying Variables

The app main program wakes each thread iteratively to do processing. The state variable controls whether or not this should occur.

Note that the current value of the state variable is the enumeration value `run`.

- Double-click the value of the state variable.

Item	Value
app	pid=31450
data	
data[0]	
name	0x08048e4c
count	23098
delta	8.726646259971648E-03
angle	2.015680753127312E+02
value	4.848096201641618E-01
data[1]	
name	0x08048e50
count	23098
delta	8.726646259971648E-03
angle	2.015680753127312E+02
value	8.746197071849462E-01
sema	1081359
rate	50000000
ptrs	
state	run

Figure 5-6. Variable Modification in Progress

The cell containing the value is frozen from updates and the current value is selected.

To change the value of a variable, all we need to do is supply a new value and commit the change to the program.

- Type the following in the cell:

hold


- Press the Enter key to commit the value to the program.

The value of the state variable is now hold which prevents the program from waking the threads for computation, as shown in the source code snippet from **app.c**:

```

103     for (;;) {
104         struct timespec delay = { 0, rate };
105         nanosleep(&delay, NULL);
106         work(random() % 1000);
107         if (state != hold) {
108             semop(sema.&trigger, 1);
109         }
110     }

```

- Change the value of the state variable back to run by double-clicking the value which is displaying hold and then using the option list chevron icon  (shown at the right-most part of the value row), select run and press Enter.

Selecting Variables for Recording and Alternative Viewing

Each variable has a **Mark** and a **Record** attribute. The **Mark** attribute, when set, indicates that the variable is of particular interest and may be viewed in other panels. The **Record** attribute specifies that the variable is to be included in recording sessions.

Double-clicking an item causes the icon color to turn red and sets its **Mark** and **Record** attributes. Alternatively, you can use an item's context menu to individually set its attributes.

- Double-click the count, angle, and value fields in the **Item** column from both `data[0]` and `data[1]` structures.
- Double-click the rate variable in the **Item** column.

The Browse page tree should look similar to the following:

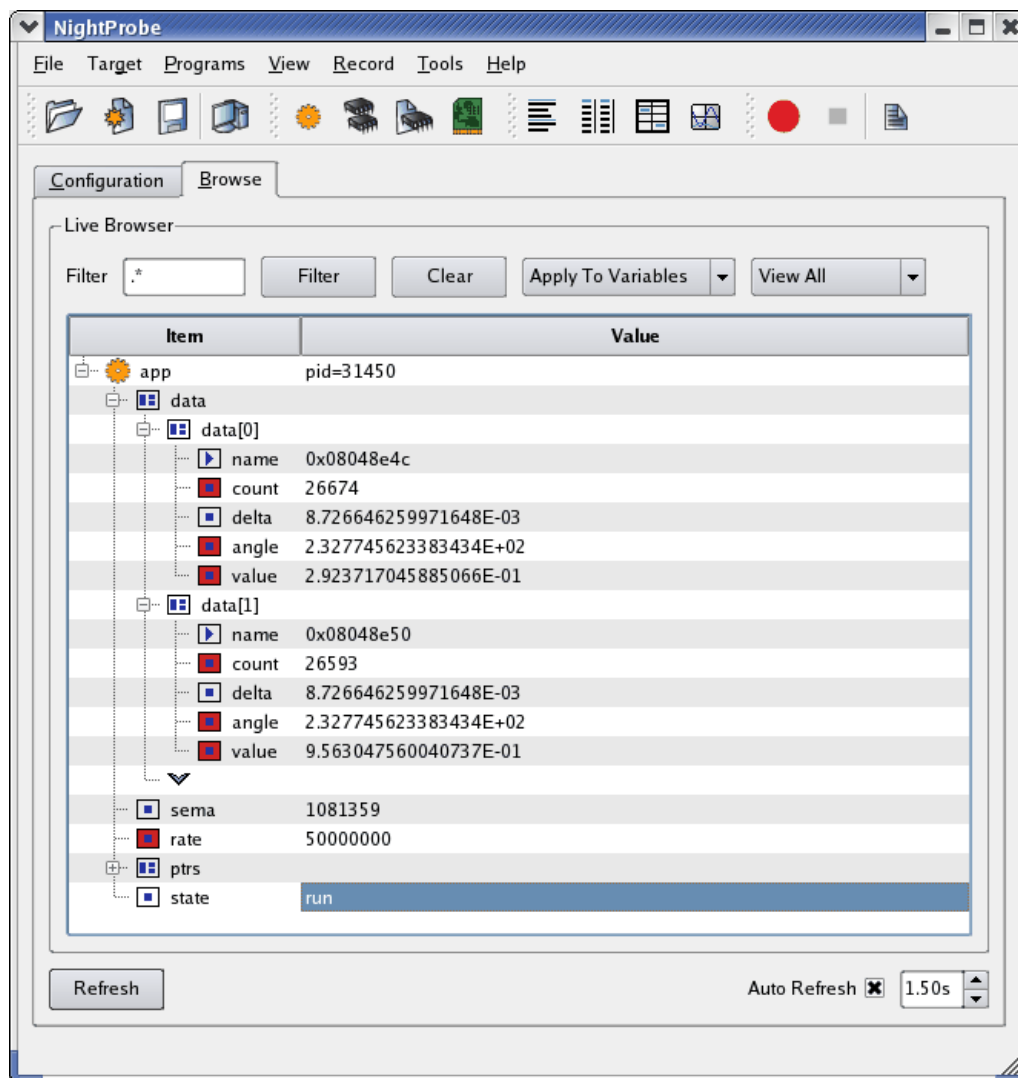


Figure 5-7. Mark and Record Attributes Set

Selection of Views

NightProbe provides various methods for viewing data:

- The Browse page
- List View
- Table View
- Spreadsheet View
- Graph View

Additionally, you can stream the output of a recording session to NightTrace or a user application for live analysis, or to a file for subsequent analysis within NightProbe.

Table View

The Table view provides a scrollable table with variables spread across the columns and rows containing the values of the variables, over time.

- From the View menu select the Table option.

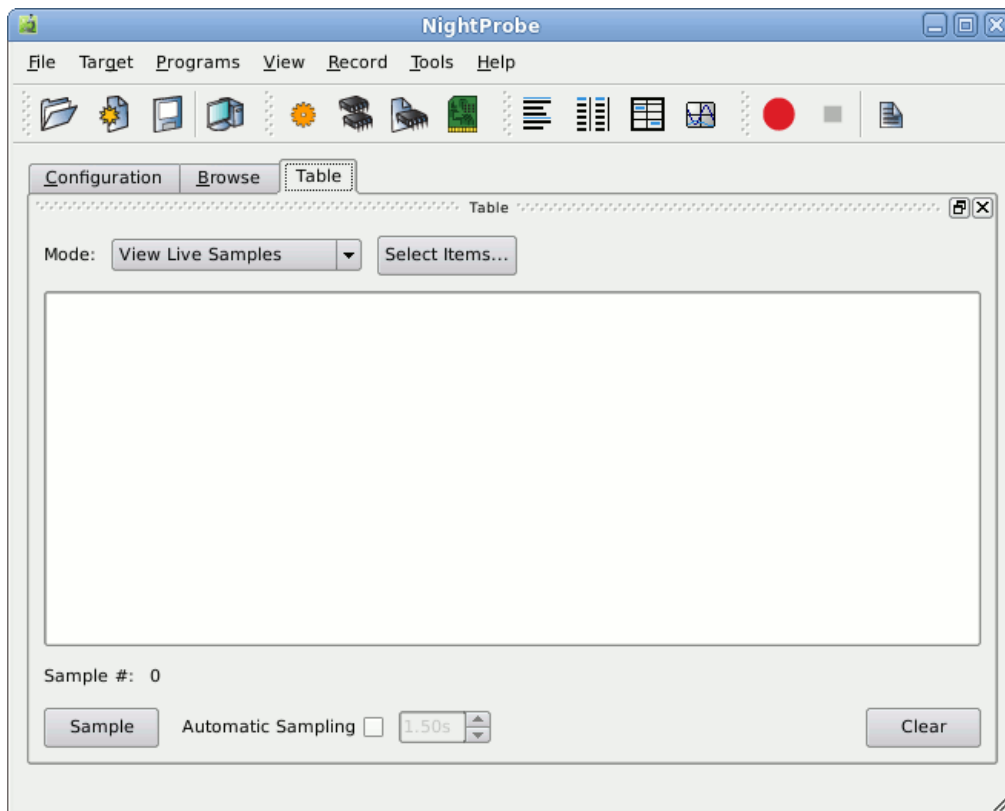


Figure 5-8. Table View

Initially, the table is empty. The first step is to select the items we wish to display in the table.

- Press the **Select Items...** button.

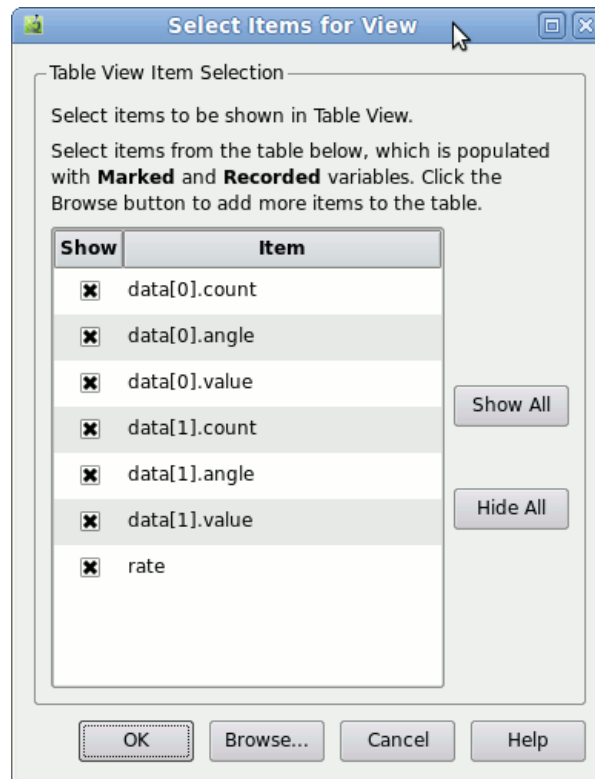


Figure 5-9. Item Selection Dialog

This dialog allows you to select items that have the Mark or Record attribute set.

By default, the dialog sets up defaults to display such variables.

- Hide all elements of the `data[1]` component by clicking their rows in the **Show** column to deselect them.
- Press the **OK** button.

The table now has five columns, one for the sample number and one for each of the variables we selected in the previous step.

- Check the **Automatic Sampling** checkbox

At the rate defined in the spinbox to the right of the Automatic Sampling checkbox, new samples are taken of the variables in the table.

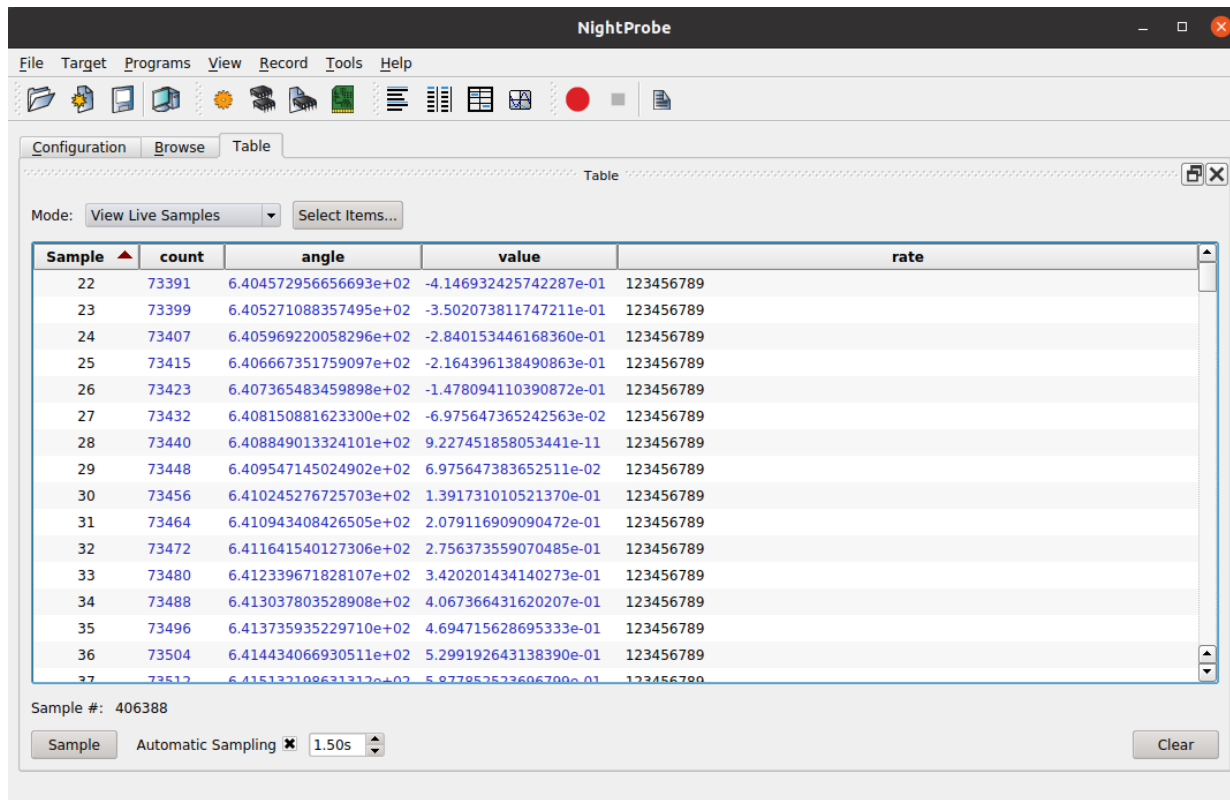


Figure 5-10. Table in Automatic Sampling Mode

Values are shown in blue if they have changed since the previous sample.

You can sort by variable value by clicking on a column header.

- Clear the Automatic Sampling checkbox
- Click on the column header for `data[0].value` and then click again so that the table is sorted from largest to smallest value.

The value shown at the top should be nearly 1.0 if enough samples have been taken (the value of `data[0].value` is that of a sine wave).

You can modify variables using the **Table** view in the same manner as described in “Modifying Variables” on page 5-5. The difference here is that the cell you click on is the value of a sample already taken. When you change the cell’s value, the variable’s value changes immediately within the program, but the cell reverts to the previously sampled value.

New samples will show the effects of the modification.

- Click on the **Sample** column header until it is sorted from smallest to largest.
- Check the Automatic Sampling checkbox.

- Click the scrollbar box and drag it all the way down to the bottom of the scrollbar and release.

New values will again be shown at the bottom of the table.

Graph View

The Graph panel presents individual variables as separate lines on a graph.

- Go to the **View** menu and select **Add New Page**.
- Go the **View** menu and select **Graph**.

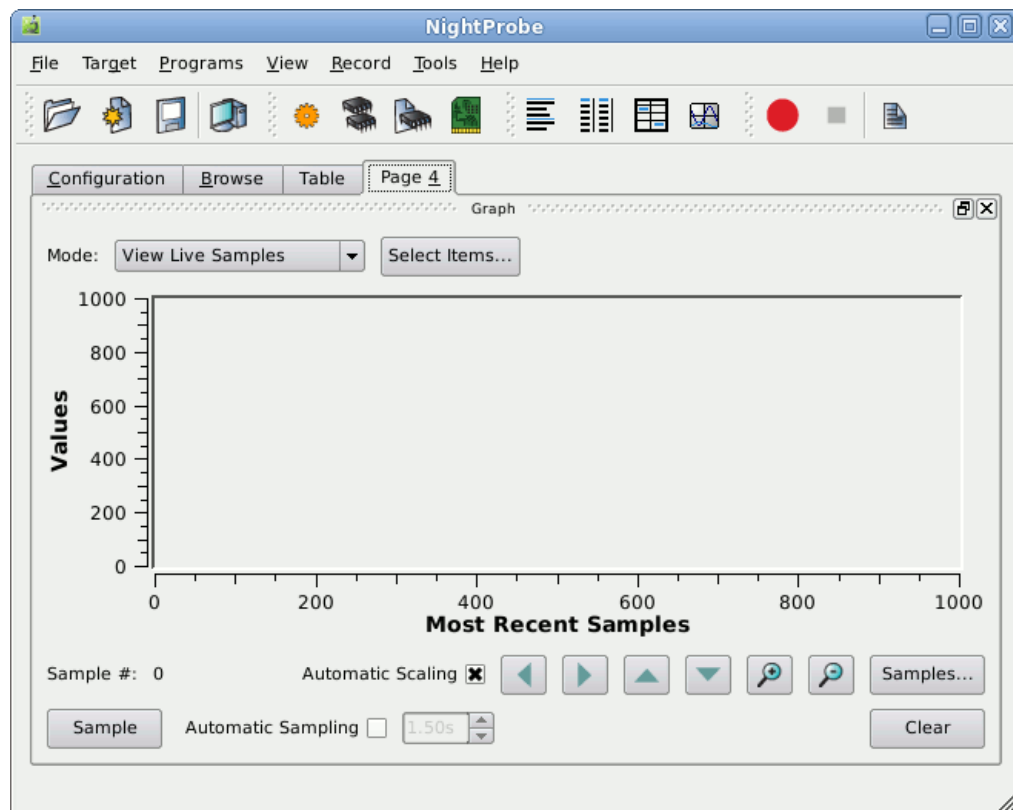


Figure 5-11. Graph Panel

Initially, the graph is empty.

- Press the **Select items...** button.

Unlike the table view, none of the items in the **Select Items for View** dialog are selected to be shown. Typically, only one or very few items are shown on a single graph.

- Mark the `data[0].value` and `data[1].value` items to show by clicking their respective rows in the **Show** column.
- Press the **OK** button.

- Ensure the Automatic Sampling checkbox is checked.
- Change the refresh rate to 1.0 seconds in the spinbox to the right of the Automatic Sampling checkbox.

Two lines begin to be plotted as shown below.

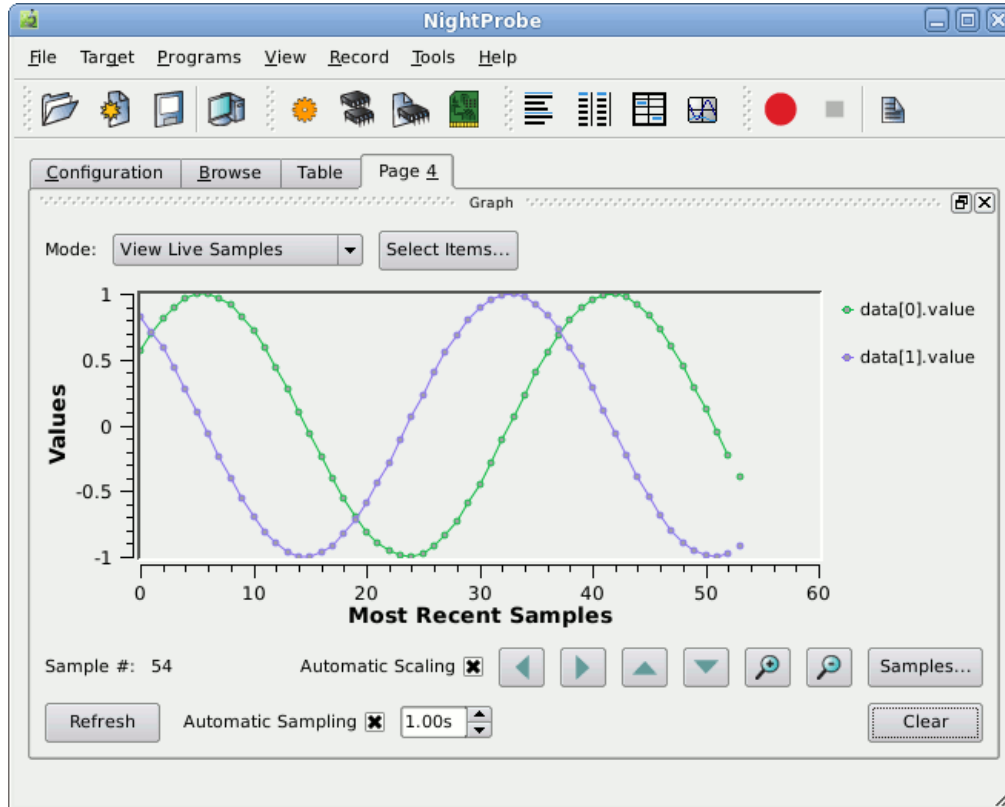


Figure 5-12. Graph Panel Actively Displaying Values

- Select the Edit... option from the context menu of one of the value items in the legend at the right-hand side of the graph panel. Right-click to activate the context menu.

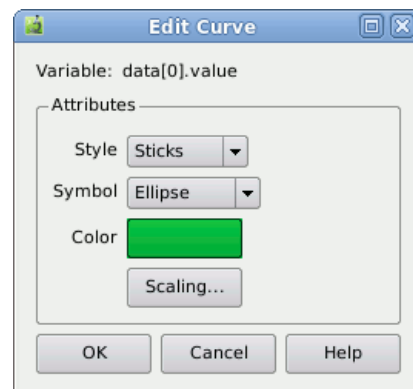


Figure 5-13. Edit Curve Attributes Dialog

- Select **Sticks** from the **Style** option list.
- Click on the colored block to activate a color selection dialog to change the color, if you want.
- Press the **OK** button to close the color selection dialog.
- Press the **OK** button to close the Edit Curve Attributes dialog.

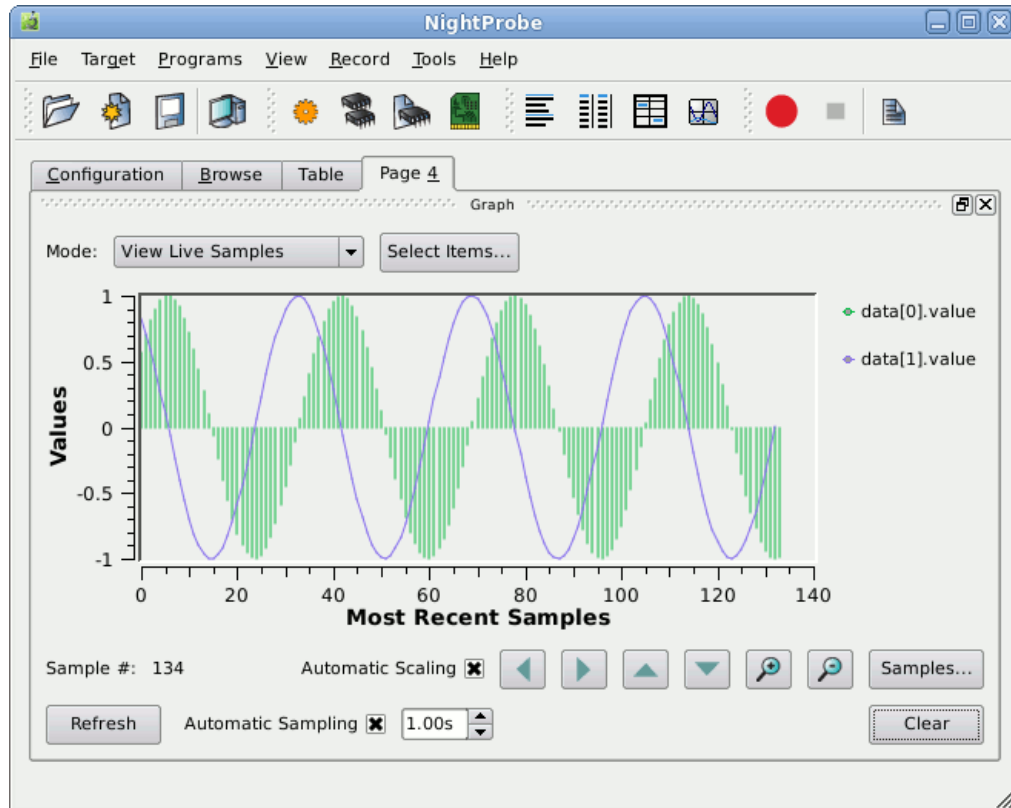


Figure 5-14. Graph Panel with Modified Curves

- Press the **Zoom In** icon repeatedly until the desired detail level is reached.

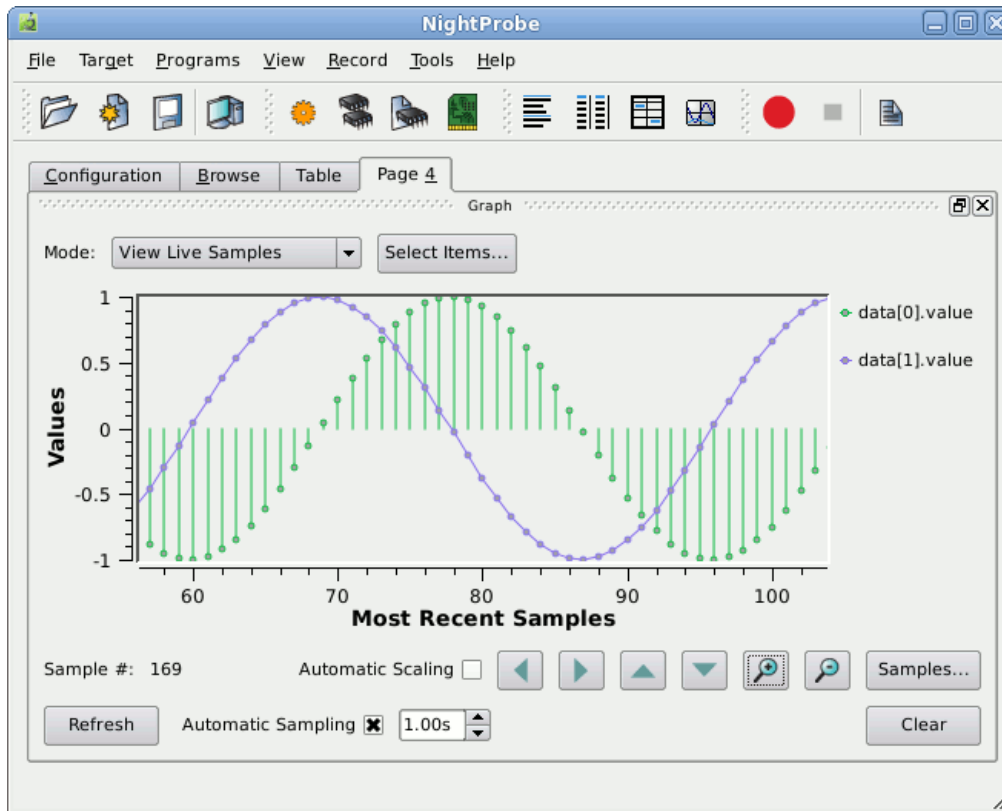


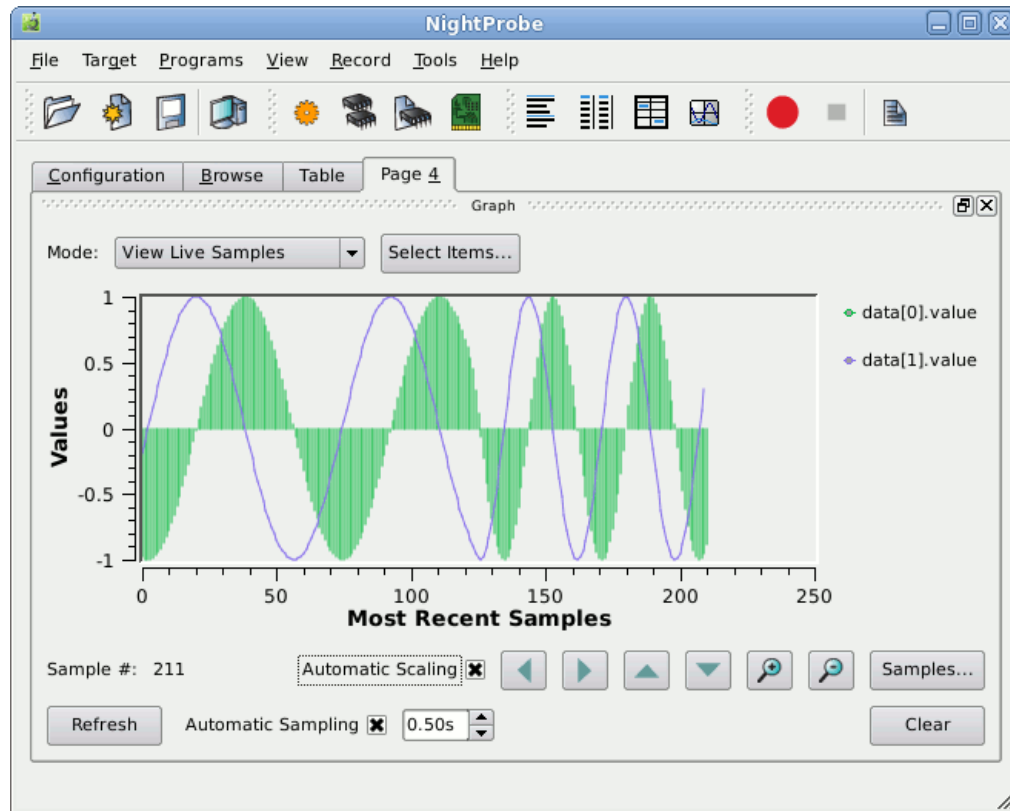
Figure 5-15. Graph Zoomed In

- Check the Automatic Scaling checkbox if it is not already checked.
- Change the refresh rate to 0.5 seconds

The program uses the rate variable to determine the frequency at which the threads are activated to do their calculations.

- Using the **Browse** page or the **Table** panel, change the value of the `rate` variable from 50000000 to 25000000.
- Click the **Page 4** tab to see the graph.

This change effectively doubles the frequency at which the threads operate, so the sine and cosine waves will change shape, as shown below.



Sending Probed Data to Other Programs

Data values may be recorded to files for subsequent processing, or may be recorded and streamed to NightTrace for live processing.

Similarly, you can send recorded data to any process of choice.

- Raise the Configuration page by clicking on its tab.

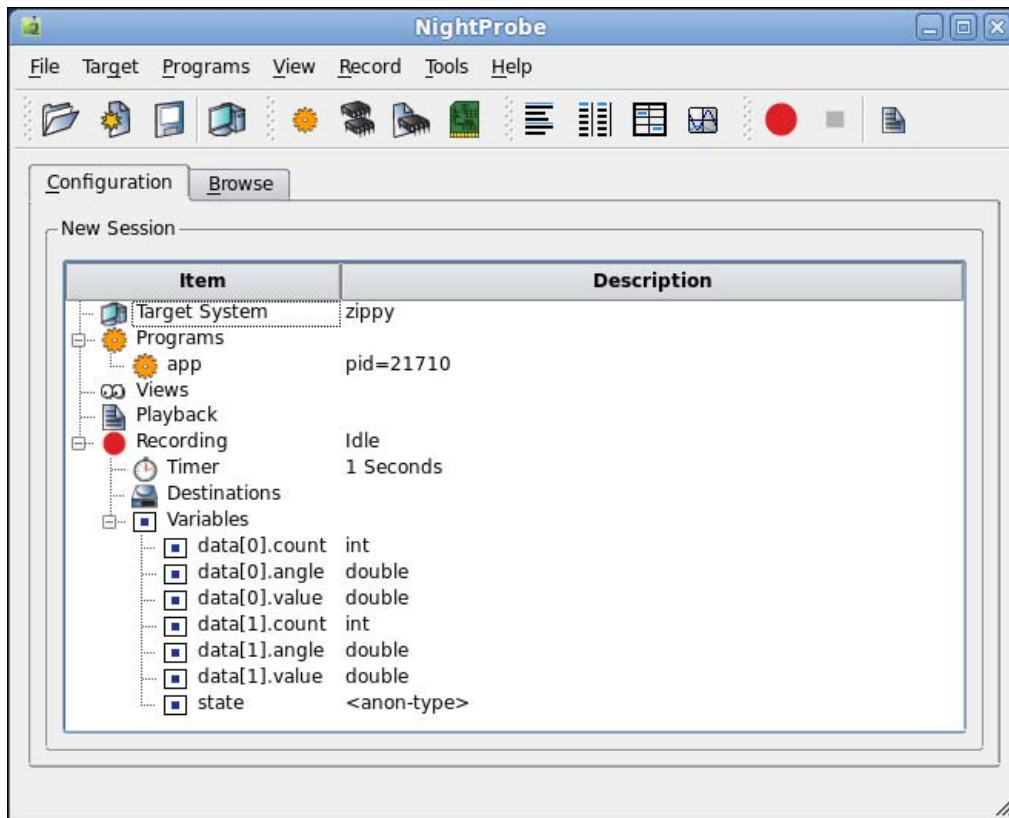


Figure 5-16. Recording area of Configuration Page

The Recording portion of the configuration tree indicates the Timing source for recording, the recording Destinations, and the list of variables whose Record attributes are set.

- Right-click on the Timer item in the Recording tree and select the Clock... option from the Timer sub-menu.

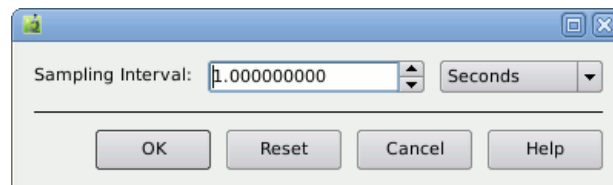


Figure 5-17. Clock Selection Dialog

This dialog controls the rate at which recording samples will be taken.

- Change the Sampling Interval value to 100.0.
- Change the units to Milliseconds using the Sampling Interval option list.
- Press the OK button.

The **Timer** item and description in the tree changes to reflect this activity.

The recording destination will be a user application.

- Right-click the **Destinations** item and select **To Program...**

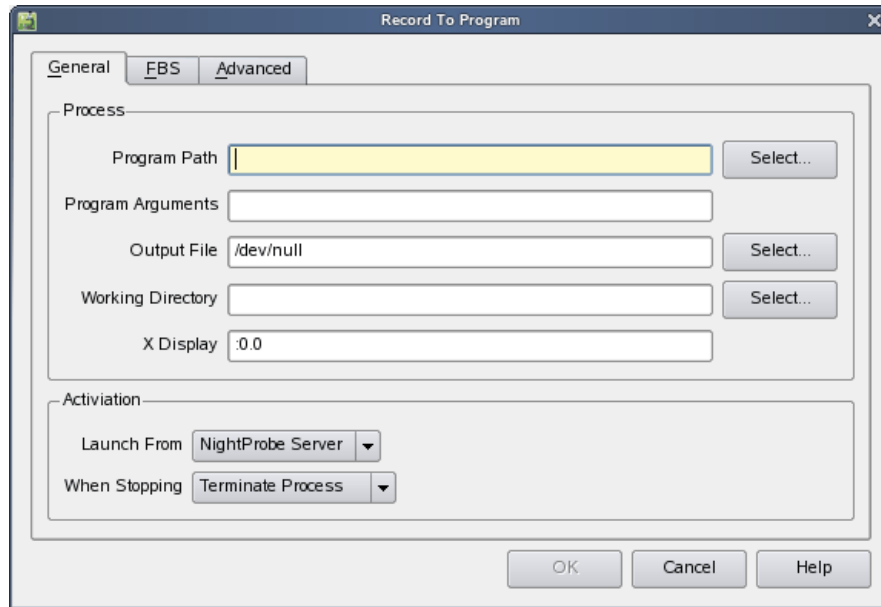


Figure 5-18. Record To Program Dialog

- Type **api** into the **Program Path** text field.
- Replace the **/dev/null** text in the **Output File** text field with the following.

/tmp/api.out

- Press the **OK** button.

A simple application which uses the NightProbe API to consume and print the values of recorded samples was copied into the **tutorial** directory in “Creating a Tutorial Directory” on page 1-4.


The API program should already have been built. If not, see “Building the Program” on page 1-4, type **make**, and then return here.

The Recording area of the Configuration page should look similar to the following.

Item	Description
Target System	narf
Programs	
app	pid=18128
Views	
Playback	
Recording	Idle
Timer	100 Milliseconds
Destinations	
api	/home/jeffh/work/tutorial/api
Variables	
data[0].count	int
data[0].angle	double
data[0].value	double
data[1].count	int
data[1].angle	double
data[1].value	double
rate	int

Figure 5-19. Recording Area of Configuration Page w/ Destination

Now that we have selected the variables to record, the recording timing source, and the recording destination, we can proceed to record samples and stream them to the **api** application.

- Press the Record icon  on the toolbar:
- View the output of the api program as samples are recorded and passed to it.
- Enter the following command in a terminal session:

```
tail -f /tmp/api.out
```


The program will generate output similar to the following:

```

gnome-terminal
File Edit View Terminal Tabs Help
item: count: 29072
item: angle: 253.701060
item: value: 0.694658
item: count: 29072
item: angle: 253.701060
item: value: -0.719340
item: rate: 50000000
Sample 479
item: count: 29074
item: angle: 253.718513
item: value: 0.681998
item: count: 29074
item: angle: 253.718513
item: value: -0.731354
item: rate: 50000000
Sample 480
item: count: 29076
item: angle: 253.735967
item: value: 0.669131
item: count: 29076
item: angle: 253.735967
item: value: -0.743145
item: rate: 50000000

```

Figure 5-20. Example Output of Graph Program

- Stop the recording process by pressing the Stop icon  on the Recording toolbar, then press Ctrl+C to gain terminal control.

For more information on the NightProbe API, refer to the “NightProbe API” chapter in the *NightProbe User’s Guide*.

Using Datamon to Modify Program Variables

The Data Monitoring Application Programming Interface is part of the NightStar RT tool set.

Data Monitoring allows you to specify executable programs that contain Ada, C, C++, or Fortran variables to be monitored, obtain and modify the values of selected variables by specifying their names, and obtain information about the variables such as their addresses, types, and sizes.

NOTE

Ada programs are only supported if compiled with the Concurrent MAXAda compiler which generates proper DWARF debug information.

Data Monitoring is a powerful capability with a rich API. It also allows you to obtain detailed symbolic and attribute information for variables in a program file. However, for our purposes, we will write a very simple program which changes the value of a single variable.

Refer to the *Data Monitoring Reference Manual* for more information about Data Monitoring.

The source code for our **set_rate** program follows:

```
#include <stdlib.h>
#include <stdio.h>
#include <datamon.h>

#define check(x) \
    if((x)) {fprintf(stderr, "%s\n", dm_get_error_string());exit(1);}

main(int argc, char * argv[])
{
    program_descriptor_t pgm;
    object_descriptor_t obj;
    char buffer[100];

    if (argc != 2) {
        fprintf (stderr, "Usage: set_rate integer-value\n");
        exit(1);
    }

    check(dm_open_program("app", 0, &pgm));
    check(dm_get_descriptor("rate", 0, pgm, &obj));
    check(dm_get_value(&obj, buffer, sizeof(buffer)));
    check(dm_set_value(&obj, argv[1]));

    printf ("rate: old_value=%s, new_value=%s\n", buffer, argv[1]);
}
```

The `dm_open_program` function initializes Data Monitoring on the specified process name and PID (in this case zero, which instructs the call to use any process matching the specified name).


The `dm_get_descriptor` call looks for the specified variable name and returns information about the variable. It also maps the underlying memory page of the variable in the **app** process into the monitoring process.

The `dm_get_value` and `dm_set_value` routines return and set the value of the variable using direct memory reads and writes; the **app** process is not affected in any other way than having the value of the rate variable changed.

The **set_rate.c** source file was copied into the current working directory during the activities in “Creating a Tutorial Directory” on page 1-4. The **set_rate** program should have already been built. If not, see “Building the Program” on page 1-4, type **make**, and then return here.

While this portion of the tutorial is in no way dependent on NightProbe itself, we will use NightProbe to see the effect of changing the rate variable using the Datamon API.

- Raise the **Graph** panel by clicking on the tab labeled **Page 4** in NightProbe.

- Use the **Pan Right** button  in the graph panel to move the viewport to the end of the graph set -- click the button repeatedly until the end of the graph is seen.
- Change the value of the `rate` variable in the **app** process by issuing the following command:

```
./set_rate 123456789
```

As shown in the source code above, the program prints the previous value of the `rate` variable and then sets it to the value specified as an argument to **set_rate**.

The sine and cosine waves change shape as shown in the **Graph** panel.

Conclusion - NightProbe

To terminate NightProbe operations, execute the following steps:

- Go to the **File** menu and select **Exit Immediately**.

This concludes the NightProbe portion of the NightStar RT Tutorial.

6

Using NightTune

NightTune is a graphical tool for analyzing and adjusting system activities.

This chapter assumes you have already built the **app** program and it is running. If you have not built the program and it isn't running, do so using the instructions in “Building the Program” on page 1-4 and execute the application before proceeding: **./app &**

Invoking NightTune

- Launch NightTune at a command prompt with the following command:

ntune &

Or, launch it by double-clicking on the NightTune desktop icon.

For some aspects of this tutorial, it will be necessary to execute NightTune as the **root** user or to ensure that your user account has appropriate privileges. See the “Setting Up User Privileges” on page 1-2 for more information.

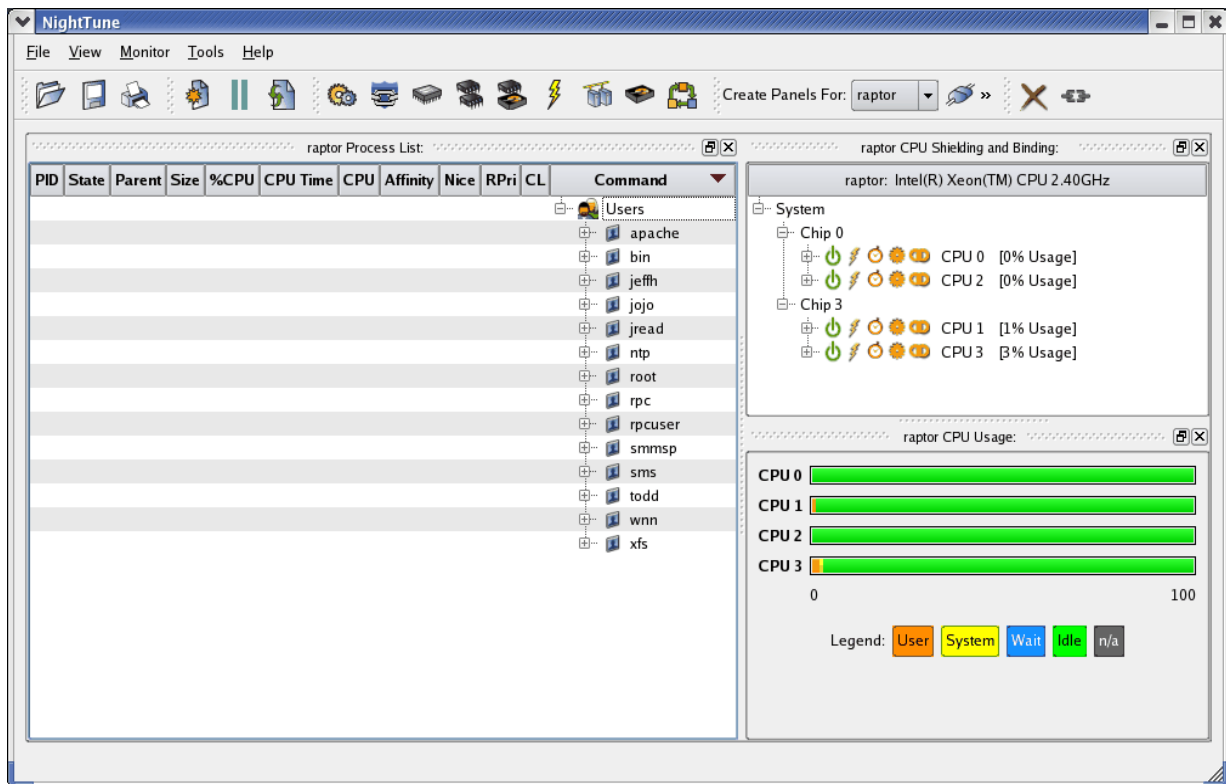


Figure 6-1. NightTune Initial Panels

NOTE

If you do not see the **Command** column in the **Process List** panel, increase the horizontal width of the window until you do.

If you have used NightTune before, you may have customized the layout of items within NightTune. You might want to reset the layout to the default one during this session, as is used in this tutorial. If so, go to the **File** menu and select **Load System Default Configuration**.

Monitoring a Process

First monitor the running **app** process.

- In the **Process List** panel, click anywhere within the panel and then type **Ctrl+F**.
- A **Find** bar appears at the bottom of the panel. Type **^app**, and the process list will automatically expand and the first process whose process name starts with the word “app” will be selected.

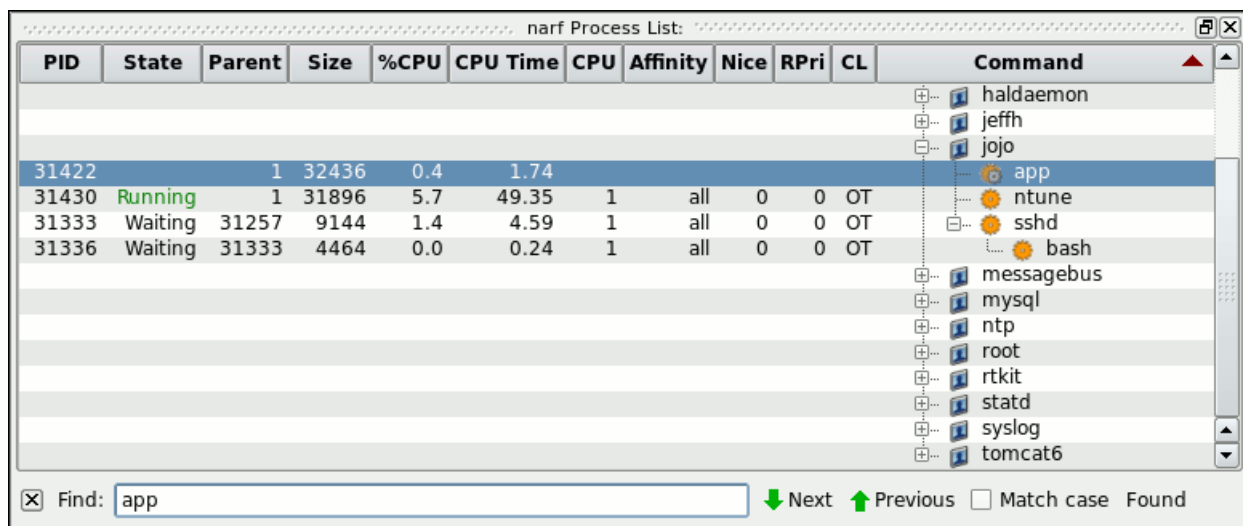



Figure 6-2. Expanded Process List

If the selected process is not your **app** process, press the **Next** icon  in the **Find** bar until the correct process is selected.

As shown below, notice that the icon associated with the **app** process has a small gray gear superimposed on the orange process icon. This indicates that process is multi-threaded.



- From the context menu associated with the **app** process, select the Show Threads option.

PID	State	Parent	Size	%CPU	CPU Time	CPU	Affinity	Nice	RPri	CL	Command
1685	Waiting	1610	12624	0.0	0.03	3	all	0	0	OT	ssh-agent
1	Waiting	0	27084	0.0	1.32	3	all	0	0	OT	init
2	Waiting	0	0	0.0	0.00	0	all	0	0	OT	kthreadd
5419	Waiting	2312	80852	0.0	0.01	0	all	0	0	OT	sudo
5820	Waiting	5764	80852	0.0	0.00	3	all	0	0	OT	sudo
5821	Waiting	5820	22016	0.0	0.07	0	all	0	0	OT	bash
5906	Waiting	5821	246392	99.9	77.58						app
5906	Waiting			0.0	0.05	1	all	0	0	OT	main
5907	Running			99.9	77.53	2	all	0	50	FF	watchdog_thread
5908	Waiting			0.0	0.00	1	all	0	0	OT	sin
5909	Waiting			0.0	0.00	0	all	0	0	OT	cos
5910	Waiting			0.0	0.00	0	all	0	0	OT	heap_thread
5911	Running	5821	103248	2.4	2.71	3	all	0	0	OT	ntune
483		1	247468	0.0	0.47						rtkit
											syslog
											rsyslogd

Figure 6-3. Process List with Threads

The panel shows characteristics of each thread and of the entire process. In particular, they include:

- the virtual memory size of the process
- the percentage and amount of CPU time used by each thread and by the whole process
- CPU on which each thread ran most recently
- CPU affinity for each thread (the set of CPUs on which the thread is allowed to run)
- scheduling characteristics of each thread
- the thread name, if it is being debugged by NightView, or, if the application is using the NightTrace API and names its threads via a call to `trace_set_thread_name(3x)`

The set of columns displayed can be modified by clicking the Display Fields option of the context menu for the panel, and then choosing individual fields by checking or unchecking their menu items.

Tracing System Calls

NightTune provides a handy interface for tracing system calls made by a process. This is essentially the same as using the `strace(1)` command, except that NightTune provides the output in a dialog which can be searched and controlled.

- From the context menu associated with the `sin` thread in the **app** program, select the **Trace System Calls...** option then press the start button, which is an aqua-colored right arrow.

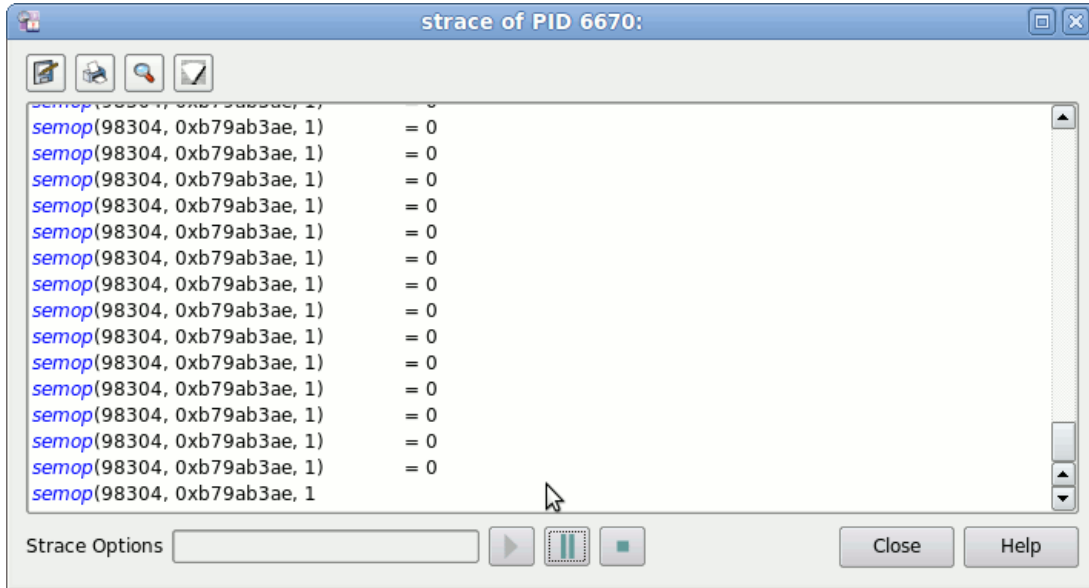


Figure 6-4. Strace Output of Thread

As shown in the figure above, the selected thread makes no system calls other than **semop(2)** which is associated with the line 51 of **api.c**, as shown in this code segment:

```

41 void *
42 sine_thread (void * ptr)
43 {
44     control_t * data = (control_t *)ptr;
45     struct sembuf wait = {0, -1, 0};
46     work(1);
47
48     trace_set_thread_name (data->name);
49
50     for (;;) {
51         semop(sema, &wait, 1);
52         data->count++;
53         data->angle += data->delta;
54         data->value = sin(data->angle);
55     }
56 }

```

- Press the **Close** button to stop the system call trace and close the dialog.

Process Details

NightTune provides detailed analysis of process attributes.

- Select the **Process Details...** option from the context menu of any thread in the **app** program.

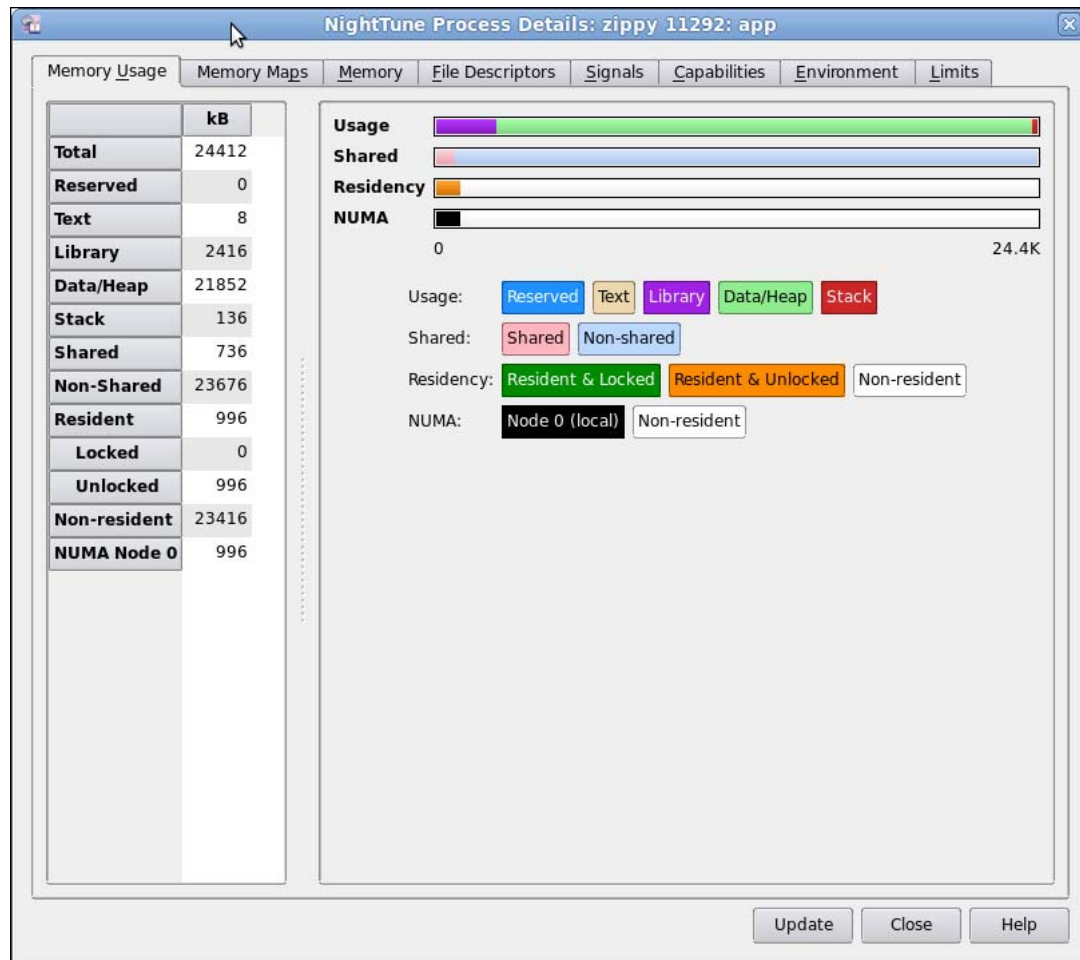


Figure 6-5. Process Details Dialog

All information displayed in this dialog is read-only in nature. You cannot make changes to process attributes using this dialog, except for locking memory pages on the **Memory** tab.

Eight tabbed pages provide detailed information about the process, including:

- Memory Usage
- Memory Maps
- Memory
- File Descriptors
- Signals
- Capabilities
- Environment
- Limits

The **Memory Usage** page provides summary information of the virtual and resident usage of memory in both textual and graphical panes.

Process Details - Memory Details

- Click on the **Memory** tab to raise that page.

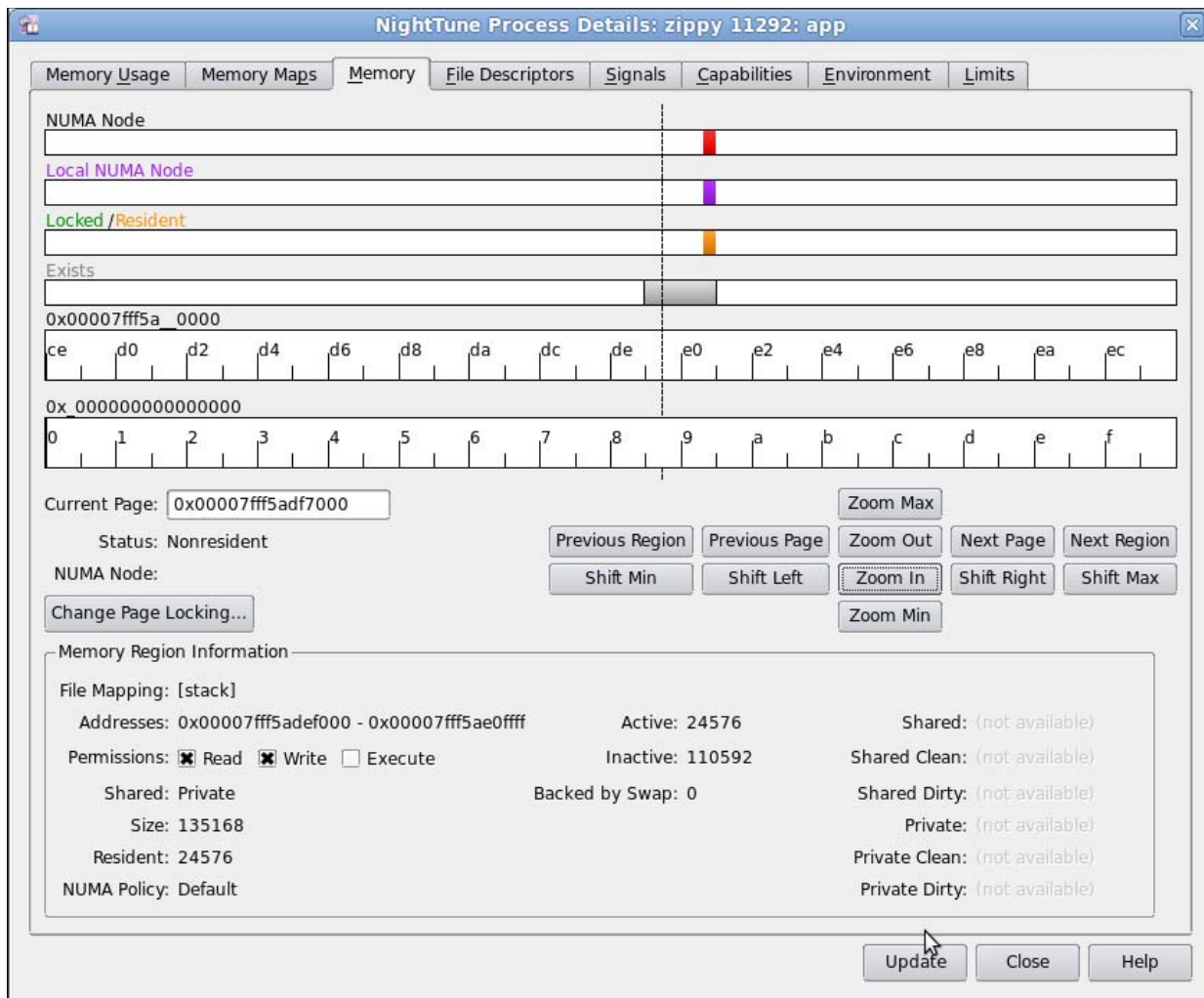


Figure 6-6. Process Memory Details Page

This dialog provides controls to allow you to get detailed memory information for any segment or page within the address space.

The controls in the graphical rows are similar to NightTrace in nature.

- Click anywhere on or above the rulers.
- Press **Alt+Up** keys to zoom out completely.

The process's entire address space is now displayed. Each segment of the memory address space that is associated with pages in your process is indicated by at least a single vertical black line in the **Exists** row.

- If no lines are visible in the middle portion of the display, click **Previous Region** and then **Zoom In** multiple times until you see something.
- Use the mouse wheel or the **Zoom In** button to zoom in until sufficient detail is available.

In the figure above, memory segments are shown as gray areas in the **Exists** row. The boundaries of memory segments are shown as vertical black lines. If the zoom factor is large enough, a memory segment may be portrayed as merely one or two vertical black lines.

Details about the memory segment are shown in the textual area in the bottom portion of the page.

The other rows show per-page information, including NUMA pools, and Locked and Resident attributes of the page.

NOTE

Locked and Resident information may not be available on all operating system versions. NUMA information is only applicable to systems supporting a Non-Uniform Memory Architecture and the information is only provided by some operating systems.

Alternatively, you can select a specific address by typing it into the **Current Page** text field.

See the *NightTune User's Guide* for more information on the **Memory** page.

Process Details - Signals

The Signals table displays attributes of signals.

Number ▲	Name	Pending	Shared Pending	Blocked	Ignored	Handled	Restart	Description
1	SIGHUP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Hangup
2	SIGINT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Interrupt
3	SIGQUIT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Quit
4	SIGILL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Illegal instruction
5	SIGTRAP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Trace/breakpoint trap
6	SIGABRT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Aborted
7	SIGBUS	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Bus error
8	SIGFPE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Floating point exception
9	SIGKILL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Killed
10	SIGUSR1	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	User defined signal 1
11	SIGSEGV	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Segmentation fault
12	SIGUSR2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	User defined signal 2
13	SIGPIPE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Broken pipe
14	SIGALRM	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Alarm clock
15	SIGTERM	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Terminated
16	SIGSTKFLT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Stack fault
17	SIGCHLD	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Child exited
18	SIGCONT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Continued
19	SIGSTOP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Stopped (signal)
20	SIGTSTP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Stopped
21	SIGTTIN	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Stopped (tty input)
22	SIGTTOU	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Stopped (tty output)
23	SIGURG	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Urgent I/O condition

Figure 6-7. Signals Page

The information shown includes indicators of signals currently pending or blocked by the application, as well as whether the application has a handler installed for a signal.

In the figure above, the application has a handler registered for SIGUSR2 and has ignored SIGHUP.

- Click on the **Close** button to close the dialog.

Process Details - File Descriptors

The **File Descriptors** tab lists all open file descriptors associated with the process, and provides a description of each.

The figure below shows the file descriptors in use by an **ntune** process.

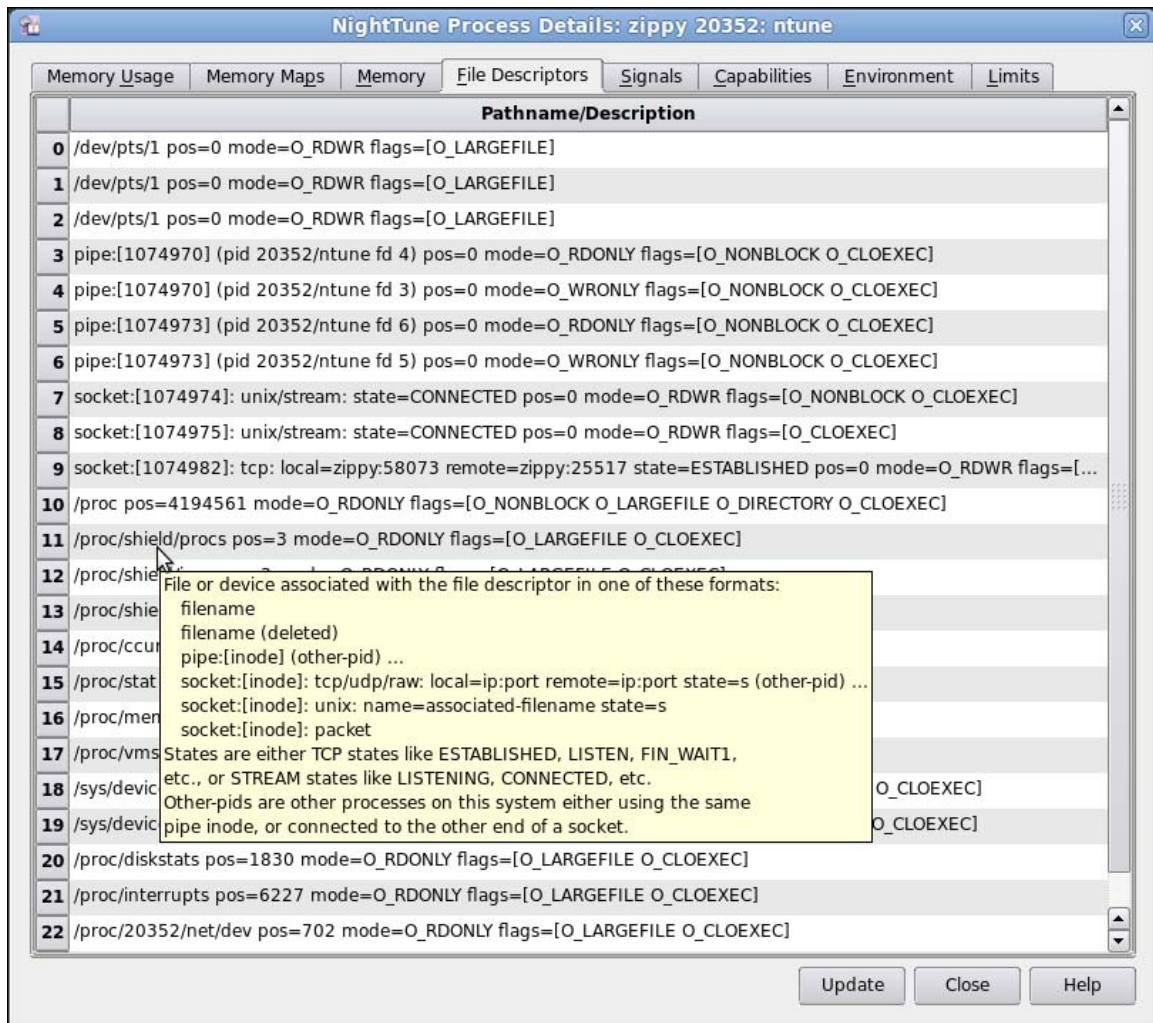


Figure 6-8. File Descriptors Page

The description includes the file name associated with a file descriptor (when relevant), connection information for a socket, and even identifies other processes using a pipe or socket when those processes are on the same system.

Changing Process Scheduling Parameters

It may be desirable to change the scheduling properties of a thread or process while it is running to see how that changes the behavior of an application. For instance, perhaps one thread is being starved of CPU time by other threads. You may wish to change its scheduling class to a real-time class and/or its priority to a higher priority.

- On the context menu associated with the `sin` thread in the **app** process, select **Process Scheduler...** .

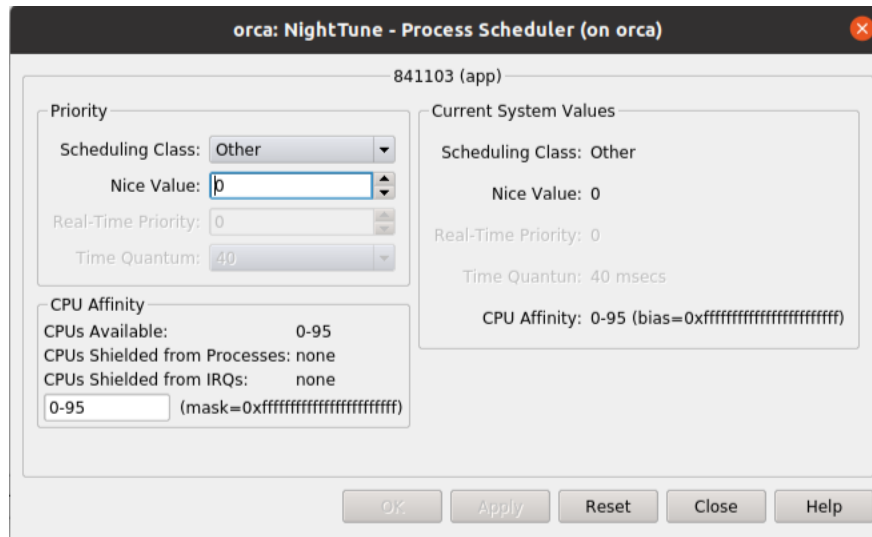


Figure 6-9. Process Scheduler Dialog

In this dialog, you can change the Scheduling Class, Nice Value, Real-time Priority, and/or Time Quantum. On multi-processor systems, you can also change the CPU Affinity. For each CPU on which the process or thread is allowed to run, the checkbox with the number of that CPU should be checked. See “Setting Process CPU Affinity” on page 6-11 for more on this topic.

NOTE

To change the Scheduling Class to Round Robin and change the Real-time Priority, NightTune must be run by the **root** user or your user account must have appropriate privileges as described in “Setting Up User Privileges” on page 1-2.

- From the Scheduling Class drop down list, select Round Robin.
- Change the Real-Time Priority to 3.
- Press the OK button.

The Process List panel now reflects these changes to the thread.

PID	State	Parent	Size	%CPU	CPU Time	CPU	Affinity	Nice	RPri	CL	Command
1685	Waiting	1610	12624	0.0	0.03	0	all	0	0	OT	ssh-agent
1	Waiting	0	27084	0.0	1.32	3	all	0	0	OT	init
2	Waiting	0	0	0.0	0.00	0	all	0	0	OT	kthreadd
5419	Waiting	2312	80852	0.0	0.01	0	all	0	0	OT	sudo
5820	Waiting	5764	80852	0.0	0.00	3	all	0	0	OT	sudo
5821	Waiting	5820	22016	0.0	0.07	0	all	0	0	OT	bash
5906	Waiting	5821	246392	99.9	113.37	1	all	0	0	OT	app
5907	Running			0.0	0.08	2	all	0	50	FF	main
5908	Waiting			0.0	0.01	1	all	0	3	RR	watchdog_thread
5909	Waiting			0.0	0.01	0	all	0	0	OT	sin
5910	Waiting			0.0	0.00	0	all	0	0	OT	cos
5911	Running	5821	103528	2.4	3.68	3	all	0	0	OT	heap_thread
483		1	247468	0.0	0.47						ntune
											rtkit
											syslog
											rsyslogd

Find: ^app Next Previous Match case Found

Figure 6-10. NightTune Process List with modified thread

For the modified thread, the CL (Scheduling Class) field displays the value RR (Round Robin), and the RPri (Real-Time Priority) field displays the value 3.

Setting Process CPU Affinity

This section only is applicable if the system running NightTune is a multi-processor system. If not, skip to “Conclusion - NightTune” on page 6-17.

The CPU Shielding and Binding panel shows the CPU hierarchy, shielding status (on Concurrent RedHawk Linux only), CPU usage, and process and IRQ bindings.

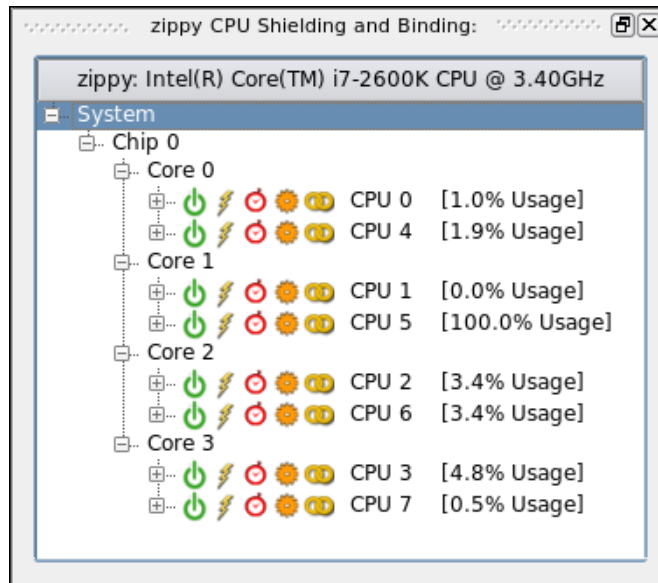


Figure 6-11. CPU Shielding and Binding Panel

The hierarchy is useful in visualizing the relationship of logical CPUs, especially in the presence of hyper-threaded and multi-core chips.

In the figure above, a single chip contains four physical cores which are hyper-threaded, totaling eight logical CPUs. Hyper-threaded CPUs share some physical resources between them, yet operate in all user-visible ways as independent processors. Multi-core CPUs also share physical resources between their siblings, but much less so than with the hyper-threaded technology.

A process or thread has a CPU affinity, which determines the set of CPUs on which it may execute. It may even be restricted such that it may run on only a single CPU. Often this is called *binding* the process or thread. “Changing Process Scheduling Parameters” on page 6-9 described one way to change the CPU affinity. In addition, the CPU Shielding and Binding panel can be used to bind a process or thread quickly.

- Select **Expand All** from the context menu associated with the **System** item in the panel

The tree expands with leaves for bound processes and interrupts for each CPU.

- While the cursor is positioned over one of the threads in the **app** process, press and hold the **left** mouse button, then drag the thread to one of the CPUs in the CPU Shielding and Binding panel and release the mouse button.

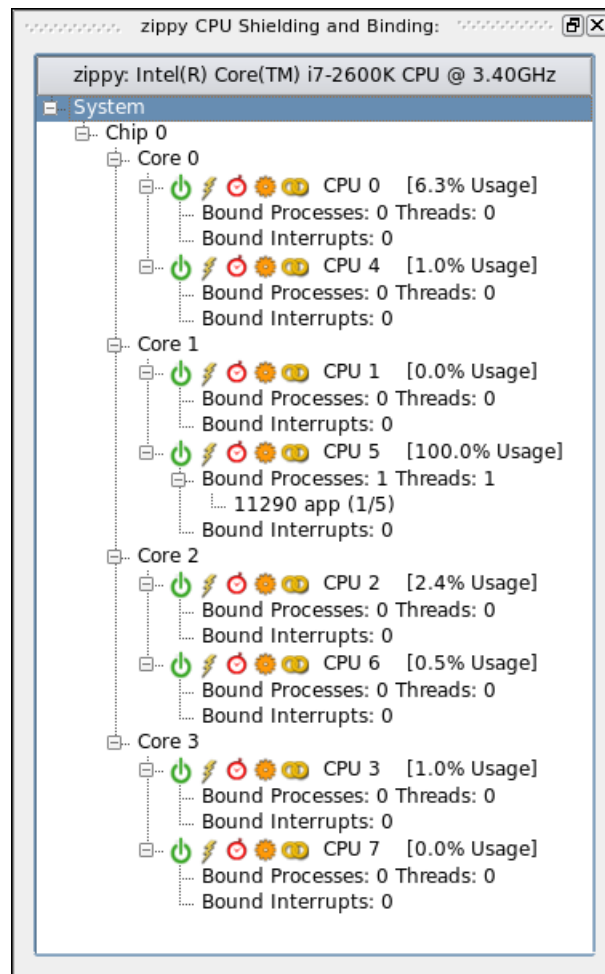


Figure 6-12. CPU Shielding and Binding Panel with Bound Thread


This action binds the selected thread to the particular CPU. That is, its CPU affinity is set to include only that single CPU. When a process's or thread's CPU affinity contains only a single CPU, that process or thread is listed in the CPU Shielding and Binding panel under the particular CPU's Processes tab. In the figure above, we bound the `sin` thread to CPU and there is one entry under CPU 5. Because only one thread was bound to CPU 1 in this example, the entry includes the suffix (1/5), indicating that only 1 of the 5 threads is bound to that CPU.

NOTE

Your system may have additional processes or interrupts bound to the CPU you selected.


The thread's new CPU affinity also is reflected in the Affinity field of the Process List panel. That field displays a bit mask in hexadecimal, where the low order bit represents CPU 0, the next bit represents CPU 1, etc. In this case, the value `0x20` indicates CPU 5.

NightTune also can unbind a process quickly.

- While the cursor is over the thread entry in the **CPU Shielding and Binding** panel, press and hold the **left** mouse button, then drag the item to the **Unbind** icon  at the upper right of the window (resembling a broken chain link) and release the mouse button.

The **Process List** panel will reflect that the thread is unbound once again.

You can also kill programs from within NightTune.

- In the **Process List** panel, click and drag the **app** program until it hovers over the **Kill** icon  and release the mouse.

Setting Interrupt CPU Affinity

The functionality described in this section only is available if NightTune was executed by the **root** user or your user account has appropriate privileges as described in “Setting Up User Privileges” on page 1-2. If this is not the case, skip to “Conclusion - NightTune” on page 6-17.

In addition to being able to set the CPU affinity of a process, NightTune can control the CPU affinity of an interrupt.

It may be desirable to change the CPU affinity of an interrupt. For instance, an interrupt may be occurring frequently and, depending on the CPU which handles it, may be interfering with an application running on that same CPU.

- Close the **Process List** panel by clicking on the box in the upper-right of its title bar.
- In its place, open the **Interrupt Detail Activity** panel: From the **Monitor** menu select the **Interrupt Detail Activity** option, then the **Text Pane** option from its sub-menu.
- If you have a large number of CPUs, a confirmation dialog box will pop up. Press **Yes** to continue.

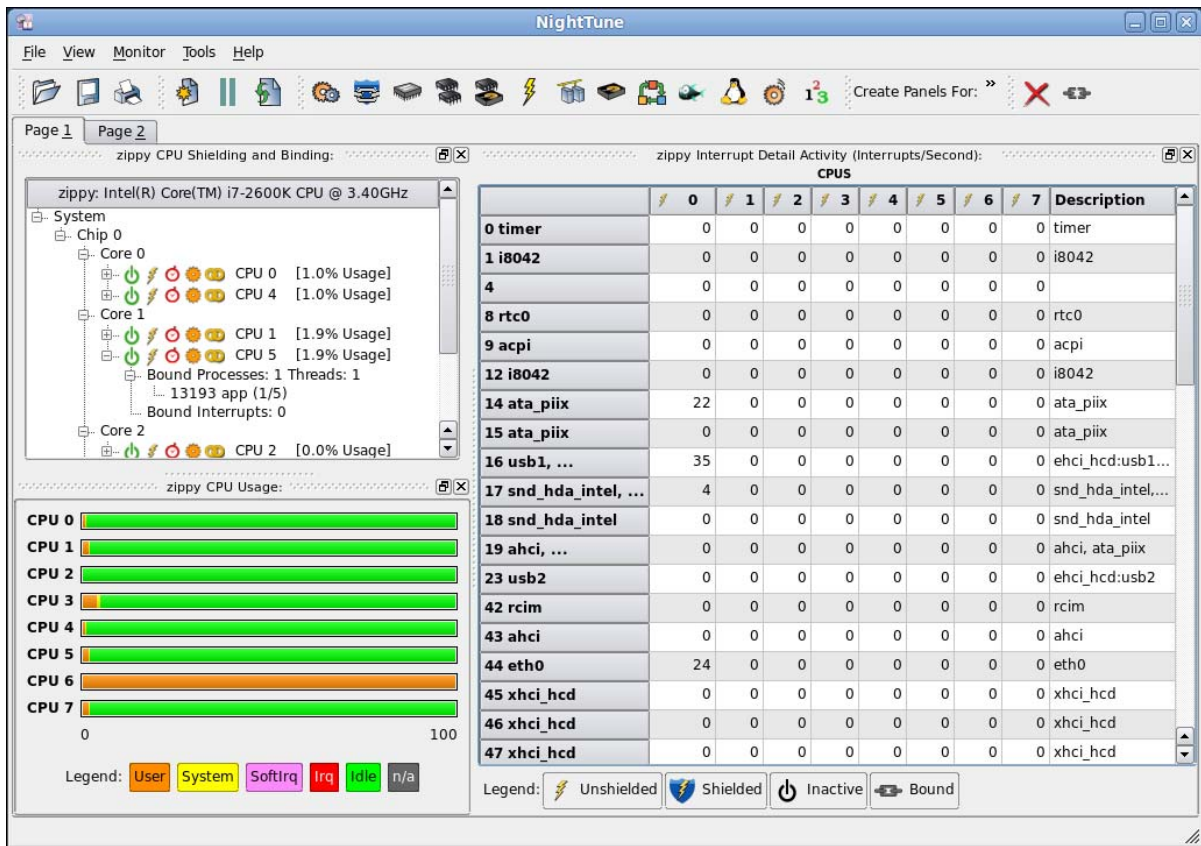


Figure 6-13. NightTune with Interrupt Detail Activity Panel

The panel shows the number of interrupts per second for each interrupt as handled on each CPU (if on a multi-processor system).

The chain link icon in the Interrupt Detail Activity panel indicates that an interrupt may be handled by that particular CPU. However, if an interrupt may be handled by all CPUs, then no icon appears for that interrupt. The same information is displayed in the Bound Interrupts items for each CPU in the CPU Shielding and Binding panel.

Some systems may employ IRQ balancing which automatically changes IRQ affinities over time. This interferes with attempts to control interrupt affinity manually. For purposes of this tutorial, ensure that IRQ balancing is currently disabled by executing the following command as the root user:

```
sudo systemctl stop irqbalance
```

To bind an interrupt to a single CPU, it may be dragged in much the same way as a process.

While the cursor is over an interrupt in the Interrupt Detail Activity panel, you may press and hold the **left** mouse button over any data cell (other than the title) in the row of an interrupt, then drag the interrupt to the particular CPU in the CPU Shielding and Binding panel. Similarly, while the cursor is over an interrupt in the Bound Interrupts list of a CPU in the CPU Shielding and Binding panel, you may press and hold the

left mouse button, then drag the interrupt to a different CPU in the CPU Shielding and Binding panel.

To change an interrupt's affinity to allow multiple CPUs, but possibly exclude one or more, select the **Set CPU Affinity...** option from the context menu of any interrupt row in the panel.

NOTE

As the root user or if your user permissions lack appropriate privileges, the **Set CPU Affinity...** option will not be present in the context menu.

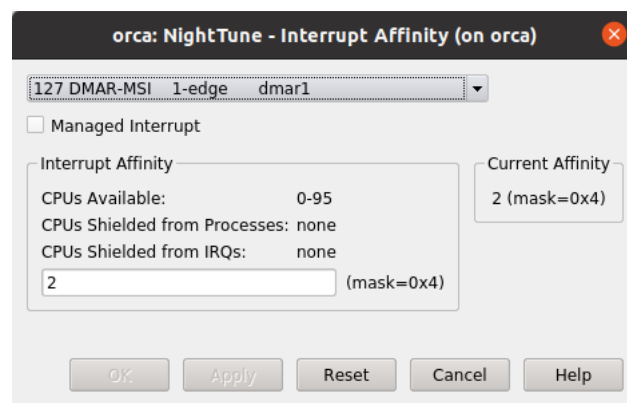


Figure 6-14. Interrupt Affinity Dialog

Specify which CPUs are allowed to handle the interrupt by entering their numbers in the text box. Comma-separate multiple CPUs, or use a dash to specify a range of CPUs (e.g., 7,9,11-13). The changes take effect when the OK or Apply button is pressed.

NOTE

For certain interrupts, such as NMI, it is impossible to control their CPU affinity.


Shielding CPUs for Maximum Determinism and Performance

NightTune allows you to easily shield specific CPUs from processes, interrupts, and shared resource interference from other CPUs.

This is demonstrated as part of the NightSim section in this tutorial. See “Overrun Detection and System Tuning” on page 7-10 for more information.

Conclusion - NightTune

The remaining portion of the tutorial is unrelated to the execution of the **app** program. Terminate the process by executing the following steps:

- If the **app** hasn't been killed, open the process list via the **Monitor** menu and select **Process List**, search for **app** and drag the process from the **Process List** panel using the left mouse button to the Kill icon  on the toolbar and release.
- To terminate NightTune go to the **File** menu and select **Exit**.

This concludes the NightTune portion of the NightStar RT Tutorial.

Using NightSim

NightSim is a graphical tool for scheduling multiple processes in a synchronized manner and monitoring their execution.

NightSim provides a graphical interface to the Frequency Based Scheduler utilities.

If you don't have the Frequency Based Scheduler installed on your system, this portion of the tutorial isn't applicable to you. Use the following command to see if the Frequency Based Scheduler is installed:

```
rpm -q ccur-fbsched (for RHEL/CentOS)
dpkg -l ccur-fbsched (for Ubuntu)
```

This chapter of the tutorial also uses a real-time clock interrupt source from the Real-Time Clock and Interrupt Module (RCIM) which is standard equipment on most Concurrent iHawk systems. If your system does not include an RCIM device, this portion of the tutorial isn't applicable to you. Use the following command to see if an RCIM is installed:

```
cat /proc/driver/rcim/status
```

If the file shown above does not exist, an RCIM does not exist on your system or your kernel has had the RCIM support removed. If no RCIM is installed, you can check to see if you have emulated RCIM devices installed:

```
fgrep rcim-emu /proc/devices
```

For some aspects of this section, it will be necessary to execute NightSim and NightTune as the **root** user or to ensure that your user account has appropriate privileges. See the "Setting Up User Privileges" on page 1-2 for more information.

Creating FBS Applications

It is trivial to modify cyclic applications so that they may be scheduled via NightSim.

A single API call is required.

The source code for our simplistic **work** application follows:

```
#include <fbsched.h>
int workload = 1000;
main()
{
    int data = 0;
    int i;
    volatile double d = 1.0;
    while (fbswait())>=0) {
        data = !data;
        for (i=0; i<workload; ++i) d = d/d;
    }
}
```

```
    }  
}
```

The call to `fbwait()` causes the process to block until its next scheduled cycle at which point it returns. The process then performs its workload and then loops to block in `fbwait()` until its next scheduled cycle.

The **work.c** source file was copied from `/usr/lib/NightStar/tutorial` into the current working directory in an earlier portion of this tutorial. The **work** program should have already been built. If not, see “Building the Program” on page 1-4, type **make**, and then return here.

Invoking NightSim

A NightSim configuration file has been prepared for this tutorial and should have been copied to your current working directory during the activities in the section entitled “Creating a Tutorial Directory” on page 1-4.

- Launch NightSim specifying the configuration file, as shown below:

```
nsim -c nsim.nsc &
```

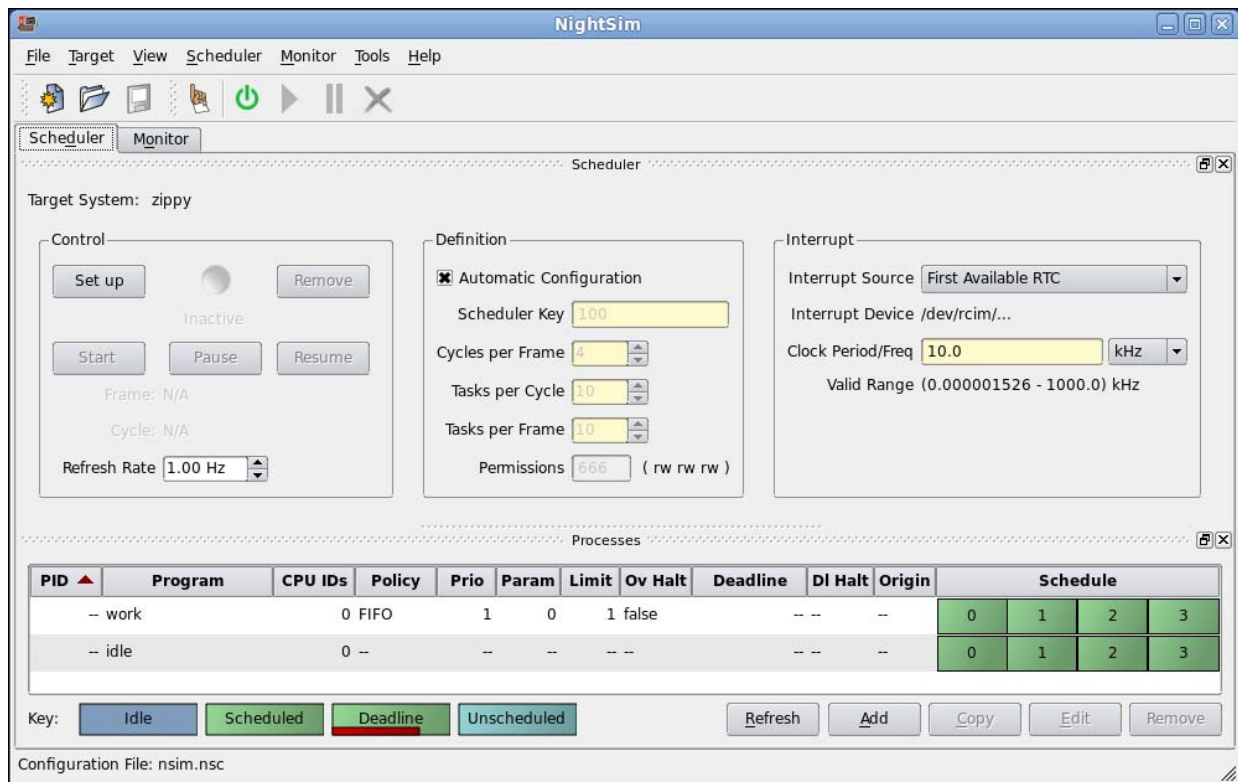


Figure 7-1. NightSim Initial Window

NOTE

If NightSim gives you an error about being unable to connect to the target system, ensure that the **hostname** of your system is part of your **/etc/hosts** file and has a proper IP address.

Creating a Scheduler

NightSim allows you to define the scheduling of multiple processes, using the following parameters:

- The scheduling source (usually an external interrupt)
- The rate at which the interrupts occur (for clock-based interrupts)
- The period at which a process is scheduled
- The CPU affinity, scheduling policy and priority of scheduled processes

Collectively, these parameters define a *scheduler*.

A cycle is defined as the time between the scheduling sources (interrupts).

A frame is defined by a fixed number of cycles. Frames are useful concepts in many cyclic applications where a series of discrete steps (cycles) must be executed in order before the entire algorithm (frame) repeats.

The scheduler configured by the **nsim.nsc** file specified on the command line in the previous section defined a scheduler with the following attributes visible on the main window:

- **Cycles Per Frame** -- four cycles per frame
- **Timing Source** - an interrupt source using the **First Available RTC** (real-time clock) of the Real-time Clock and Interrupt Module device (RCIM) or emulated real-time clock devices.
- **Clock Period** -- a cycle time of 100 microseconds (10.0 kHz)
- **Processes** -- a single process, **work**, scheduled to run on every cycle of the frame. To view the details of the attributes of the scheduled process, select the **./work** process in the process area at the bottom portion of the **Processes** panel and then press the **Edit...** button in the lower-right portion of the panel.

The **Edit Process** dialog is displayed.

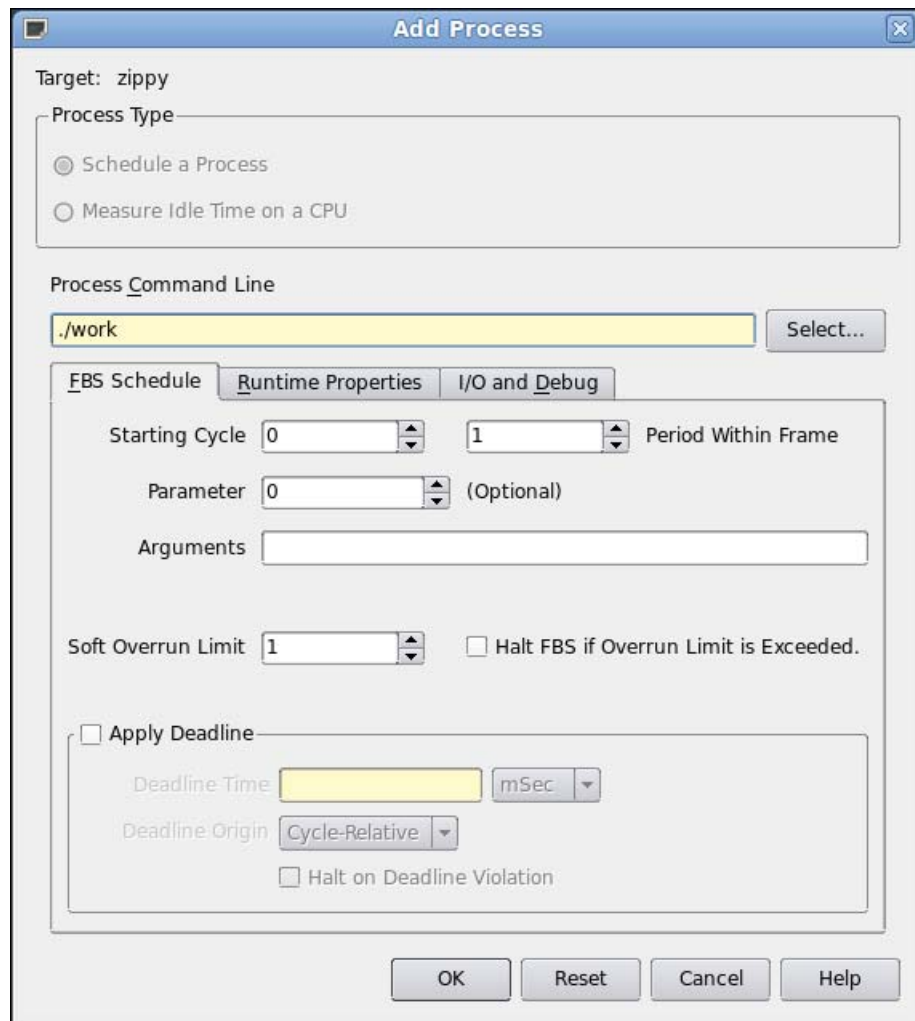


Figure 7-2. NightSim Edit Process Dialog

The FBS Schedule tab shows the starting cycle and period of the **work** process. The Starting Cycle defines the cycle within the frame where the process will begin its execution. The Period defines the frequency of execution, in cycles. A period value of 1 causes the application to execute every cycle in the frame.

- Click on the Runtime Properties tab in the dialog.

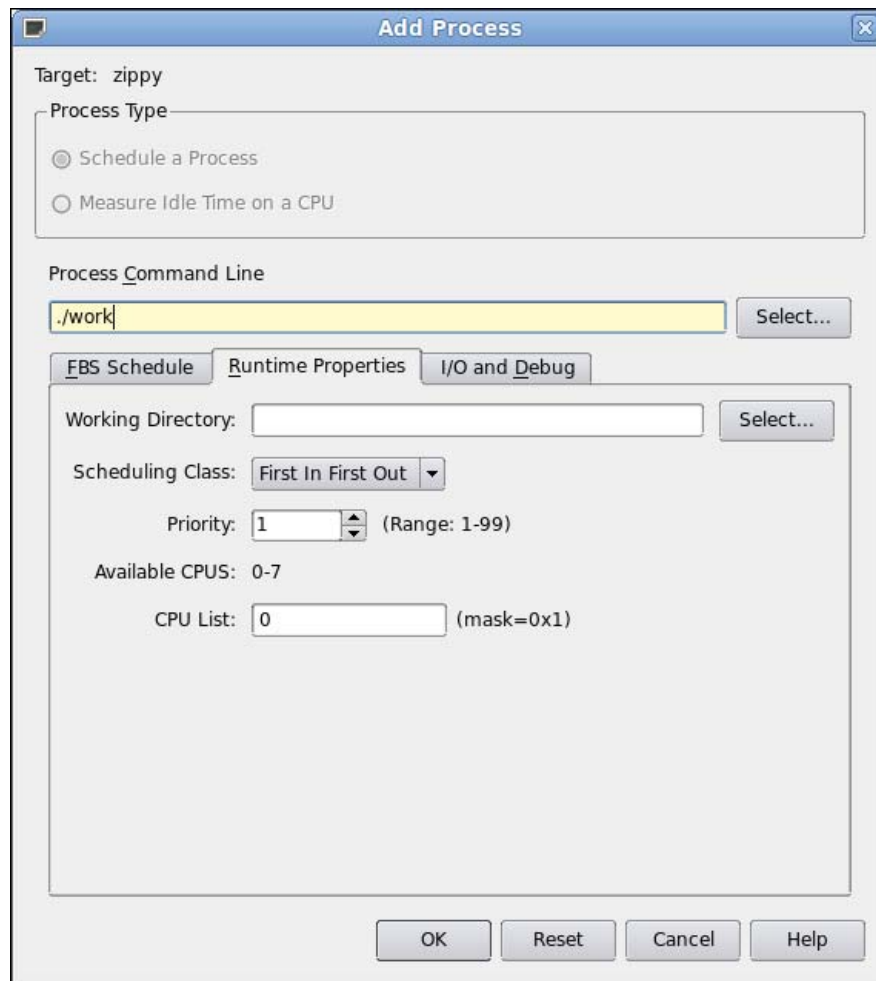


Figure 7-3. Runtime Properties Tab

NOTE

The CPU Bias description area of the Runtime Properties tab may vary depending on the number of CPUs on your system.

The Runtime Properties tab allows you to choose the CPU on which execution is allowed, the scheduling policy, and the scheduling priority of the process.

- Close the window by pressing the Cancel button.

Notice that in addition to the **work** process, the **idle** process is listed in the scheduling area of the NightSim window. We have registered the **idle** process so that we may subsequently monitor the amount of idle time available for each cycle. The **idle** process is not a process that is scheduled, but rather it is a placeholder used to represent idle cycles.

Running the Scheduler

- To start the scheduling of the process, in the **Control** area press the **Set up** button followed by the **Start** button.

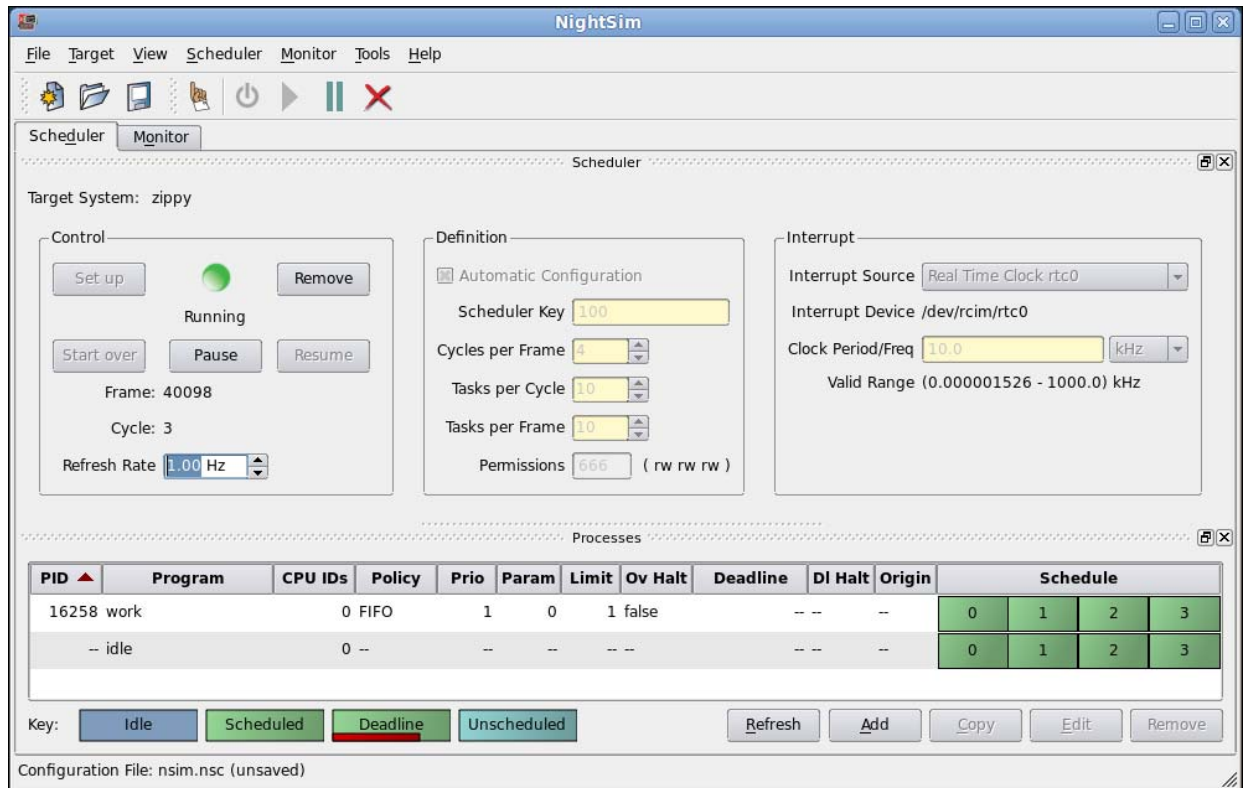


Figure 7-4. Scheduling Started

Note the Frame count begins to increase under the **Control** area as the **Cycle** oscillates between 0 and 3.

- To monitor the execution of the process, click on the **Monitor** tab near the top of the window.

The screenshot shows the NightSim Monitor Page - Metrics Panel. The table below displays the metrics for two processes. The Scheduler Status is 'Running' and the Frame is 205100.

PID	Program	Policy	Prio	CPU Bias	Start Cycle	Period	Iterations	% Used	Avg Time (us)	Last Time (us)	Total Time (us)	Soft Overruns	Overruns	Limit	Halt?	Deadline Violations
8458	work	FIFO	1	0x1	0	1	820504	33.4	33.356	33.955	2.737e+07	4563	249	1	false	0
0	idle	--	0	0x1	0	1	820800	50.3	50.302	0.000	4.129e+07	--	--	--	--	0

Scheduler Status: Running Frame: 205100 Cycle: 0

Figure 7-5. NightSim Monitor Page - Metrics Panel

The figure above isolates the **Metrics** panel from the rest of the NightSim window in order to make the panel more readable in this manual.

The NightSim Monitor **Metrics** panel provides statistics about each individual process on the scheduler. It includes the PID, program name, CPU bias, number of cycles executed, the CPU times related to per cycle execution, counts of overruns, and the average percent-age of the frame used by each process. Additional statistics can be selected for display via the **Select Fields...** option item of the table's context menu.

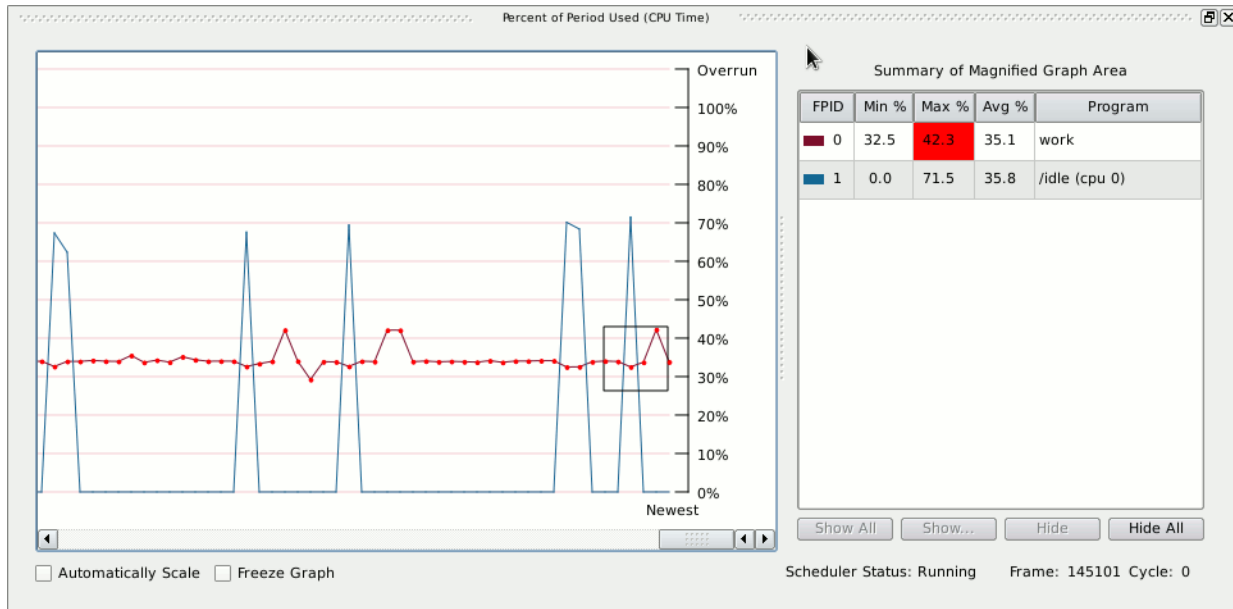


Figure 7-6. NightSim Monitor Page - Percent of Period Used Panel

The lower half of the page shows the **Percent of Period Used (CPU Time)** graph, which has been extracted in the figure above. There is a line for each process on the scheduler; the percent of time used (**CPU time**) during the last cycle is plotted over time. If an application overruns its timeslot, a red dot is shown on the graph.

Points that fall within the square magnifier are detailed in the table to the right.

You can move and resize the magnifier by clicking it and grabbing a line or corner and moving the mouse. Make sure the magnifier is covering the execution of the work process, moving and resizing it if necessary.

Important

A process can overrun its deadline even if it doesn't use more than 100% of its allotted CPU time -- other processes could be interfering with it or it may be waiting on I/O, etc. In fact, this is often the case before we tune the system for best performance, which we do later in this chapter (See "Overrun Detection and System Tuning" on page 7-10).

Watch the **Last Time** column. The values displayed are the CPU time used by each process for their last cycle's execution in microseconds. The values attributed to the **idle** process indicate the remaining CPU time available within the cycle.

We will adjust the workload of the **work** process and see the effects shown in the Night-Sim Monitor window.

Using Datamon to Modify Program Variables

The Data Monitoring Application Programming Interface is part of the NightStar RT tool set.

Data monitoring allows you to specify executable programs that contain Ada, C, C++, or Fortran variables to be monitored, obtain and modify the values of selected variables by specifying their names, and obtain information about the variables such as their addresses, types, and sizes.

Data monitoring is a powerful capability with a rich API. However, for our purposes, we will write a very simple program which changes the value of a single variable.

Refer to the *Data Monitoring Reference Manual* for more information about data monitoring.

The source code for our **set_workload** program follows:

```
#include <stdlib.h>
#include <stdio.h>
#include <datamon.h>

#define check(x) \
    if((x)) {fprintf(stderr, "%s\n", dm_get_error_string());exit(1);}

main(int argc, char * argv[])
{
    program_descriptor_t pgm;
    object_descriptor_t obj;
    char buffer[100];

    if (argc != 2) {
        fprintf (stderr, "Usage: set_workload integer-value\n");
        exit(1);
    }

    check(dm_open_program("work",0,&pgm));
    check(dm_get_descriptor("workload",0,pgm,&obj));
    check(dm_get_value(&obj,buffer,sizeof(buffer)));
    check(dm_set_value(&obj,argv[1]));

    printf ("workload: old_value=%s, new_value=%s\n", buffer, argv[1]);
}
```

The `dm_open_program` function initializes Data Monitoring on the specified process name and PID (in this case zero, which instructs the call to use any process matching the specified name).

The `dm_get_descriptor` call looks for the specified variable name and returns information about the variable. It also maps the underlying memory page of the variable in the **work** process into the monitoring process.

The `dm_get_value` and `dm_set_value` routines return and set the value of the variable using direct memory reads and writes; the **work** process is not affected in any other way than having the value of the workload variable changed.

The **set_workload.c** source file was copied into the current working directory during the activities in “Creating a Tutorial Directory” on page 1-4. The **set_workload** program should have already been built. If not, see “Building the Program” on page 1-4, type **make**, and then return here.

- Change the value of the workload variable in the **work** process by issuing the following command:

```
./set_workload 0
```

As shown in the source code above, the program prints the previous value of the workload variable and then sets it to the value specified as an argument to **set_workload**.

The Last Time field for **./work** is affected by the reduced workload as shown in the NightSim Monitor window.

- Experiment with various values of workload using the **set_workload** program until the average Last Time (us) for **./work** is approximately 50 microseconds. You may want to select Clear Performance Data from the Monitor menu after each adjustment, or just look at the graph and stop adjusting when the work line is near 50%.

Overrun Detection and System Tuning

A scheduling *overrun* occurs when a process’s next cycle begins but it has not yet finished execution of its previous cycle.

The NightSim Monitor window includes overrun counts for each process.

It is likely that several overruns have occurred for the **work** process.

NOTE

If overruns have not yet occurred, place some additional load on the system. Running the following command in a separate terminal session should have the desired effect:

```
find / -print
```

If no overruns are observed, run the following command:

```
run -b0 -P50 -sfifo find / -print
```

The NightTrace tool, as described in a previous chapter, is well suited for determining the specific cause of process overruns. NightTrace kernel tracing provides a detailed view of system activity on all CPUs, including process context switches, interrupts, system calls, and machine exceptions.

For brevity, we will assume that the overruns are due to additional activities unrelated to the scheduler occurring on the CPU where **work** executes.

We will use NightTune to shield the CPU associated with our scheduler from other activities.

NOTE

If your system only has a single CPU, the remaining portion of this section is inapplicable. Skip to “Shutting Down the Scheduler” on page 7-15 in this case.

- Press **Ctrl+c** to gain terminal control.
- Launch NightTune using the **ntune.config** file that was copied into the current working directory during the activities in “Creating a Tutorial Directory” on page 1-4:

```
ntune -c ./ntune.config &
```

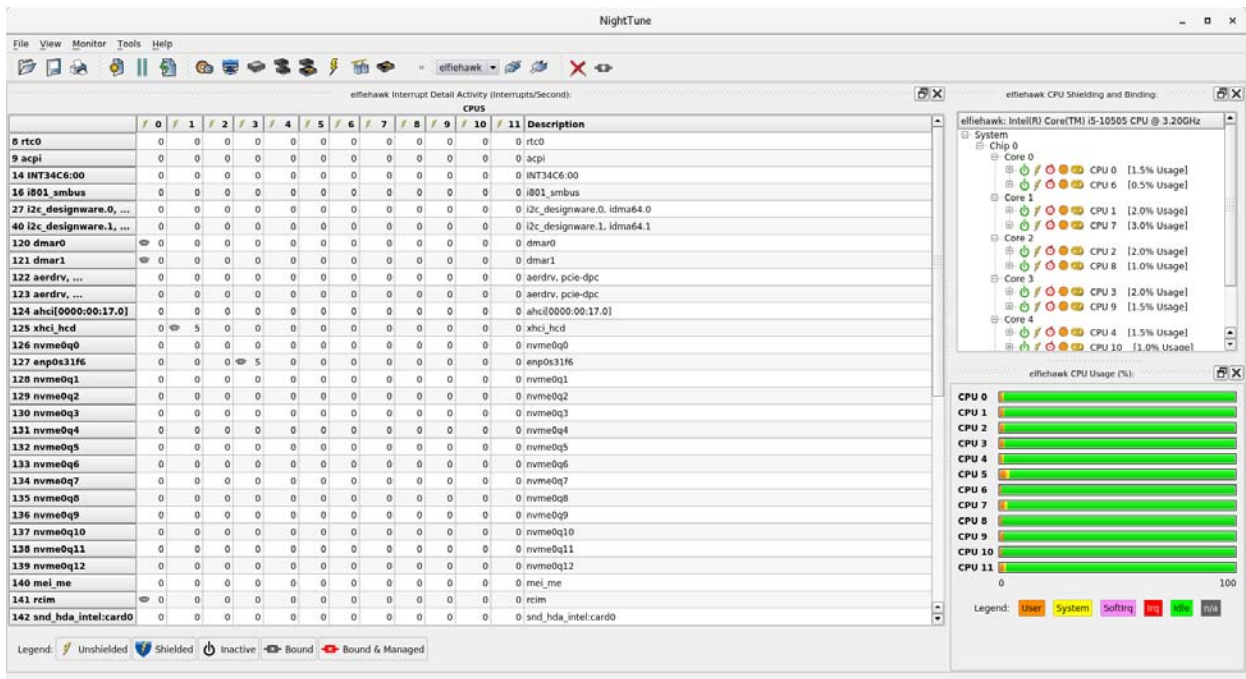


Figure 7-7. NightTune with Interrupt and CPU Shielding & Binding Panels

A NightTune window appears which displays interrupt activity and the shielding and bound status of all CPUs.

- In the CPU Shielding & Binding panel, right-click on **System** and select **Expand All** from the context menu.

Note that **work** process is listed in the Bound Processes list of CPU 0. We'll want to click the work process to move it to CPU 1. We want to do this because some interrupts are automatically bound to CPU 0.

- Drag and drop the **work** process from CPU 0 to CPU 1.

The **work** process is now bound to CPU 1.

- Locate the cell in the Interrupt Detail Activity panel in the Description panel which contains the word "rcim".

NOTE

If you don't have an RCIM and are using an emulated real-time clock, skip this portion.

You may have to resize the NightTune window and/or the Interrupt Detail Activity panel to see the Description header.

- While the cursor is positioned in the Interrupt panel over the cell in the Description column which contains the word "rcim", press and hold the left mouse button, then drag the interrupt onto the CPU 1 row in the CPU Shielding and Binding panel, and release the mouse button. The **rcim** interrupt is now bound to CPU 1.

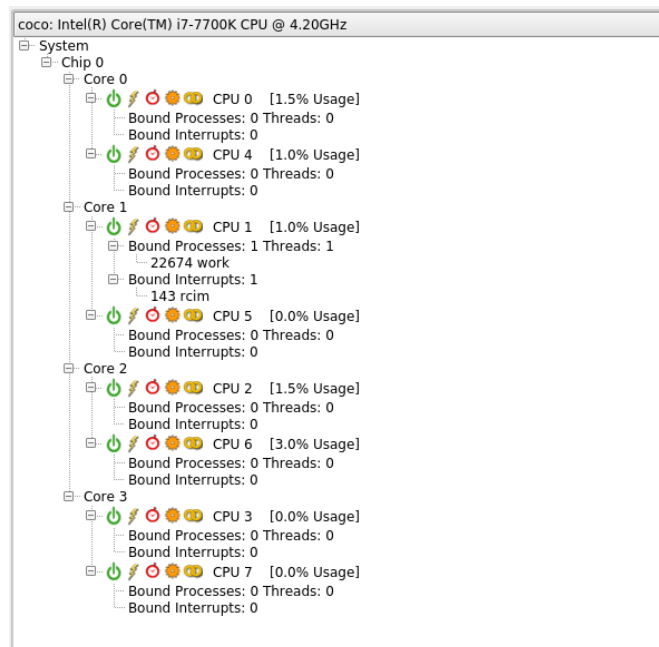


Figure 7-8. Process and Interrupt Bound to CPU 1 (not CPU 0)

- Right-click anywhere in the CPU Shielding and Binding panel and select the Change Shielding... option from the context menu.

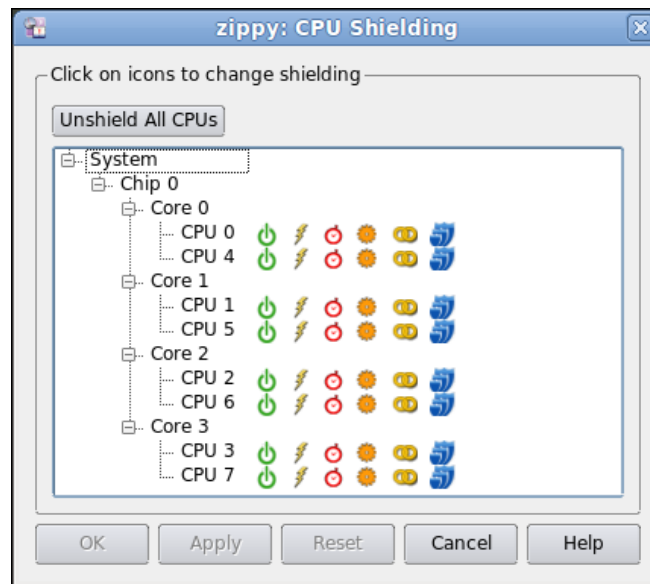



Figure 7-9. Change Shielding Dialog (unshielded)

NOTE

The CPU layout shown here may be different than that of your actual system.

- Click the Maximize Shielding icon  in the CPU 1 line (the maximize shielding icon is the right-most icon with three overlapping shield figures).

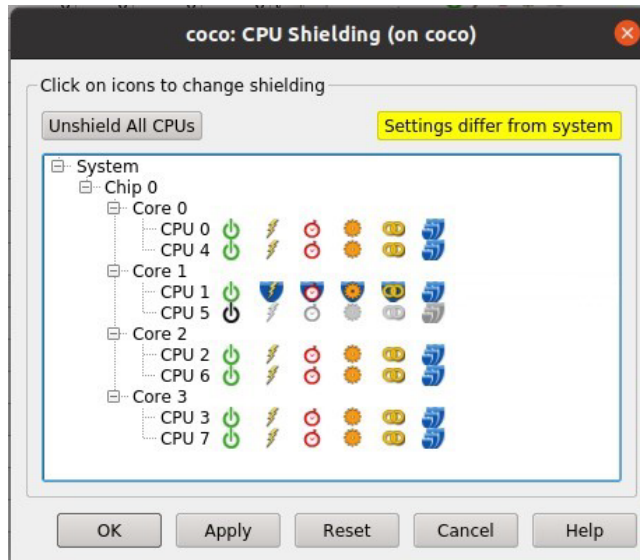


Figure 7-10. Shielding Changes Pending

The CPU 1 line changes its display to indicate that all processes and interrupts other than **work** and **rcim** will be shielded from CPU 1. Additionally, the sibling hyper-threaded CPU (in this case CPU 5) is marked down so that hyper-threaded execution on CPU 5 does not interfere with CPU 1.

NOTE

The hyper-threaded sibling of CPU 1 may be a logical CPU number other than CPU 5.

NOTE

Your system may not support hyper-threading or it may not have hyper-threading enabled in which case the CPUs are not displayed in hyper-threaded groups.

NOTE

If your system does have hyperthreaded CPUs it is possible that NightTune cannot mark the sibling of CPU 1 down. This can occur if there are other processes or interrupts that are bound to the sibling CPU. In this case, you can try to unbind them using the context menu inside the CPU Shielding and Binding panel, but be aware that some interrupts cannot be unbound (e.g. the hpet interrupt on some systems).

- Press the **OK** button to activate the shielding changes.

Return to the NightSim Monitor window and watch the **Overrun** column. It is likely that overruns have ceased to occur.

- Clear the overrun count by going to the **Monitor** menu and selecting the **Clear Performance Data** option. This action resets all the statistics to zero.

Watch the **Overrun** column to see if any overruns still occur.

If the system is properly configured, the scheduler should continue to execute without any overruns on the shielded CPU.

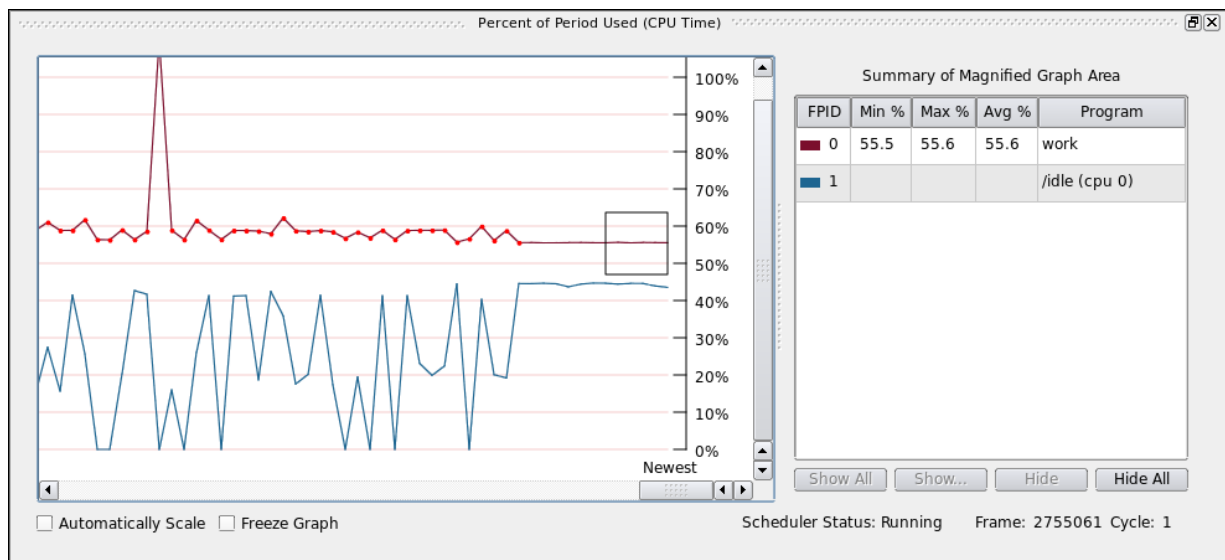


Figure 7-11. NightSim Percentage of Period Panel - Shielded CPU

In the figure above, you can see when the process overruns stopped, due to the shielding activities we took in NightTune.

Shutting Down the Scheduler

- Return to the **Scheduler** page and press the **Remove** button to terminate the scheduler. Press **Yes** if presented with a dialog which asks whether to kill the processes associated with the scheduler.
- Exit NightSim by going to the **File** menu and selecting **Exit**. A dialog asking whether or not to save changes to **nsim.nsc** may appear; if so, press **No**.
- You may also wish to clear the shielding attributes for CPU 1 and return the system to its previous state using NightTune.

- Exit NightTune by going to the **File** menu and selecting **Exit**.

This concludes the NightSim portion of the NightStar RT Tutorial.

A

Tutorial Files

The following sections show the source listings for the files used in the *NightStar RT Tutorial*.

- `api.c`
- `app.c`
- `fft.c` (not shown here)
- `function.c`
- `report.c`
- `set_workload.c`
- `set_rate.c`
- `work.c`
- `worker.c`

`api.c`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <nprobe.h>

int cycles = 0;
int overruns = 0;
char * sample;

// Perform the work of consuming a single Data Recording
// sample from NightProbe.
int
work (FILE * ofile, np_handle h, np_header * hdr) {
    np_item * i;
    int status;
    int which;
```

```

        // Read one sample, which may contain data for multiple
processes
        // and variables.
        //
        status = np_read (h, sample);
        if (status <= 0) {
            return status;
        }

        cycles++;

        fprintf (ofile, "Sample %d\n", cycles);
        for (i = hdr->items; i; i = i->link) {
            char buffer [1024];
            sprintf (buffer, "item: %s:", i->name);
            fprintf (ofile, "%-30s", buffer); // Nice formatting :-
        )

        // Display the value of each item.
        // For arrays, format each individual item.
        //
        for (which = 1; which <= i->count; ++which) {
            char * image = np_format (h, i, sample, which);

            if (image != NULL) {
                fprintf (ofile, " %s", image);
            } else {
                fprintf (ofile, "\n<error: %s>\n", np_error (h));
                return -1;
            }

            free (image);
        }
        fprintf (ofile, "\n");
    }
    fflush (ofile);

    return 1;
}

```

```

int
main (int argc, char *argv[])
{
    np_handle h;
    np_header hdr;
    np_process * p;
    np_item * i;
    int fd;
    int status;
    FILE * ofile = stdout;

```

```

fd = 0; // stdin

status = np_open (fd, &hdr, &h);
if (status) {
    fprintf (stderr, "%s\n", np_error (h));
    exit(1);
}

sample = (char *) malloc(hdr.sample_size);
if (sample == NULL) {
    fprintf (stderr, "insufficient memory to allocate
sample buffer\n");
    exit(1);
}

for (p = hdr.processes; p; p = p->link) {
    if (p->pid >= 0) {
        fprintf (ofile, "process: %s (%d)\n", p->name, p-
>pid);
    } else {
        fprintf (ofile, "resource: %s (%s)\n", p->name, p-
>label);
    }
}
fprintf (ofile, "\n");

for (i = hdr.items; i; i = i->link) {
    fprintf (ofile, "item: %s (%s), size=%d bits, count=%d,
type=%d\n",
        i->name, i->process->name, i->bit_size, i-
>count, i->type);
}
fprintf (ofile, "\n");

for (;;) {
    status = work (ofile, h, &hdr);
    if (status <= 0) break;
}

fprintf (ofile, "Data Recording done: %d cycles fired, %d
overruns\n",
    cycles, overruns);

if (ofile != stdout) {
    fclose (ofile);
}

if (status < 0) {
    fprintf (stderr, "%s\n", np_error (h));
}

np_close (h);

```

```
    // At this point, file descriptor 0 remains open, but is  
no    // longer a NightProbe Data File/Stream.  
}
```

app.c

```

#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <ntrace.h>
#include <math.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/prctl.h>

static void    add_link (void);
static void * heap_thread (void * ptr);
static void    nosighup (void);
static void * watchdog_thread (void * ptr);
extern void    work (int control);

typedef struct {
    char * name;
    int    count;
    double delta;
    double angle;
    double value;
} control_t;

control_t data[2] = { { "sin", 0, M_PI/360.0, 0.0, 0.0 },
                      { "cos", 0, M_PI/360.0, 0.0, 0.0 } };

enum { run, hold } state;

int rate = 50000000;
int sema;

extern double
FunctionCall(void)
{
    return data[0].value + data[1].value;
}

void *
sine_thread (void * ptr)
{
    control_t * data = (control_t *)ptr;
    struct sembuf wait = {0, -1, 0};
    work(1);

    trace_set_thread_name (data->name);

    for (;;) {

```

```

        semop(sema, &wait, 1);
        data->count++;
        data->angle += data->delta;
        data->value = sin(data->angle);
    }
}

void *
cosine_thread (void * ptr)
{
    control_t * data = (control_t *)ptr;
    struct sembuf wait = {0, -1, 0};
    work(1);

    trace_set_thread_name (data->name);

    for (;;) {
        semop(sema, &wait, 1);
        data->count++;
        data->angle += data->delta;
        data->value = cos(data->angle);
    }
}

int
main (int argc, char * argv[])
{
    pthread_t thread;
    pthread_attr_t attr;
    struct sembuf trigger = {0, 2, 0};
    const char * data_file = strdup("/tmp/data");
    nosighup();

    if (argc > 1) {
        data_file = argv[1];
    }
    trace_begin (data_file, NULL);

    sema = semget (IPC_PRIVATE, 1, IPC_CREAT+0666);

    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, watchdog_thread, NULL);

    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, sine_thread, &data[0]);

    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, cosine_thread, &data[1]);

    pthread_attr_init(&attr);
    pthread_create (&thread, &attr, heap_thread, NULL);

    for (;;) {
        struct timespec delay = { 0, rate } ;

```

```

        nanosleep(&delay,NULL);
        work(random() % 1000);
        if (state != hold) {
            semop(sema,&trigger,1);
        }
    }

    trace_end ();
}

void * ptrs[5];

static void *
heap_thread (void * unused)
{
    int i = 5;
    int scenario = -1;
    void * ptr;
    int * * iptr;
    extern void * alloc_ptr (int size, int swtch);
    extern void free_ptr (void * ptr, int swtch);

    trace_set_thread_name("heap_thread");

    for (;;) {
        sleep (5);
        switch (scenario) {
            case 1:
                // Use of freed pointer
                ptr = alloc_ptr(1024,3);
                free_ptr(ptr,2);
                memset (ptr, 47, 64);
                break;
            case 2:
                // Double-free
                ptr = alloc_ptr(1024,3);
                free_ptr(ptr,2);
                free(ptr);
                break;
            case 3:
                // Overwriting past end of an allocated block
#define MyString "mystring"
                ptr = alloc_ptr(strlen(MyString),2);
                strcpy (ptr,MyString); // oops -- forgot the zero-
byte
                break;
            case 4:
                // Uninitialized use
                iptr = (int * *) alloc_ptr(sizeof(void*),2);
                if (*iptr) **iptr = 2778;
                break;
            case 5:
                // Leak -- all references to block removed
                ptr = alloc_ptr(37,1);

```

```

        ptr = 0;
        break;
    case 6:
        // Some more allocations we will check on...
        ptrs[0] = alloc_ptr(1024*1024,3);
        ptrs[1] = alloc_ptr(1024,2);
        ptrs[2] = alloc_ptr(62,1);
        ptrs[3] = alloc_ptr(4564,3);
        ptrs[4] = alloc_ptr(8177,3);
        break;
    }

    (void) malloc(1);
    scenario = 0;
}

void * func3 (int size, int count)
{
    return malloc(size);
}

void * func2 (int size, int count)
{
    if (--count > 0) return func3(size,count);
    return malloc(size);
}

void * func1 (int size, int count)
{
    if (--count > 0) return func2(size,count);
    return malloc(size);
}

void free3 (void * ptr, int count)
{
    free(ptr);
}

void free2 (void * ptr, int count)
{
    if (--count > 0) {
        free3(ptr,count);
        return;
    }
    free(ptr);
}

void free1 (void * ptr, int count)
{
    if (--count > 0) {
        free2(ptr,count);
        return;
    }
}

```



```

        free(ptr);
    }

void * alloc_ptr (int size, int count)
{
    return func1(size,count);
}

void free_ptr (void * ptr, int count)
{
    free1(ptr,count);
}

void work (int control)
{
    volatile double calculations[2048];
    volatile double d = 0.0;
    int i;
    for (i=0; i<2048; i+=512) {
        calculations[i] = 3.14159;
    }
    for (i=0; i<control*10; ++i) {
        d = d*d;
        calculations[i%2048] = d;
    }
    add_link();
}

struct node_t {
    int value;
    struct node_t * link;
};

struct node_t * head;
struct node_t * tail;

static void add_link (void)
{
    static int count;
    count++;
    if (count > 5 && count < 1000) {
        struct node_t * n = (struct
node_t*)malloc(sizeof(struct node_t));
        n->value = count;
        n->link = NULL;
        if (tail) {
            tail->link = n;
        } else {
            head = n;
        }
        tail = n;
    }
}

#include <signal.h>

```

```

static void nosighup (void)
{
    struct sigaction ignore;
    ignore.sa_flags = 0;
    ignore.sa_handler = SIG_IGN;
    sigemptyset(&ignore.sa_mask);
    sigaction(SIGHUP,&ignore,NULL);
}

#include <time.h>
#include <sched.h>
#include <stdio.h>
void * watchdog_thread (void * unused)
{
    double deadline = 0.050;
    struct timespec ts;
    double last, now;
    int overrun_is_fatal = 0;
    struct sched_param param;
    extern void fft();

    // prctl(PR_SET_NAME,"watchdog_thread"); or
    trace_set_thread_name("watchdog_thread");

    param.sched_priority = 50;
    if (sched_setscheduler(0,SCHED_FIFO,&param) != 0) {
        printf("Warning: sched_setscheduler failed:
%s\n",strerror(errno));
    }
    clock_gettime(CLOCK_REALTIME,&ts);
    last = (double)ts.tv_sec + ((double)ts.tv_nsec)/
1000000000.0;
    for (;;) {
        clock_gettime(CLOCK_REALTIME,&ts);
        now = (double)ts.tv_sec + ((double)ts.tv_nsec)/
1000000000.0;
        if (now-last > deadline) {
            printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
            printf("Deadline missed by %f seconds!!!!\n",(now-
last)-deadline);
            printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
            if (overrun_is_fatal) break;
        }
        last = now;
        fft();
        usleep(10000);
    }

    for (;;) {
        sleep(1);
    }
}

```

function.c

```
double
FunctionCall(void)
{
    static double counter;
    return counter++;
}
```

report.c

```
#include <stdio.h>

void report (char * caller, double value)
{
    static int count;

    if (++count % 40) printf ("The value from %s is %f\n",
caller, value);
}
```

set_workload.c

```
#include <stdlib.h>
#include <stdio.h>
#include <datamon.h>

#define check(x) if((x)) {fprintf(stderr, "%s\n",
dm_get_error_string());exit(1);}

int
main(int argc, char * argv[])
{
    program_descriptor_t pgm;
    object_descriptor_t obj;
    char buffer[100];

    if (argc != 2) {
        fprintf (stderr, "Usage: set_workload integer-
value\n");
        exit(1);
    }
```

```

    }

    check(dm_open_program("work",0,&pgm));
    check(dm_get_descriptor("workload",0,pgm,&obj));
    check(dm_get_value(&obj,buffer,sizeof(buffer)));
    check(dm_set_value(&obj,argv[1]));

    printf ("workload: old_value=%s, new_value=%s\n", buffer,
argv[1]);
}

// Workaround for incompatible change in glibc extern void
cfree(void* x) { free(x); }

```

set_rate.c

```

#include <stdlib.h>
#include <stdio.h>
#include <datamon.h>

#define check(x) if(!(x)) {fprintf(stderr, "%s\n",
dm_get_error_string());exit(1);}

int
main(int argc, char * argv[])
{
    program_descriptor_t pgm;
    object_descriptor_t  obj;
    char buffer[100];

    if (argc != 2) {
        fprintf (stderr, "Usage: set_rate: integer-value\n");
        exit(1);
    }

    check(dm_open_program("app",0,&pgm));
    check(dm_get_descriptor("rate",0,pgm,&obj));
    check(dm_get_value(&obj,buffer,sizeof(buffer)));
    check(dm_set_value(&obj,argv[1]));

    printf ("rate: old_value=%s, new_value=%s\n", buffer,
argv[1]);
}

// Workaround for incompatible change in glibc extern void
cfree(void* x) { free(x); }

```

work.c

```

#include <fbsched.h>

int workload = 1000;

int
main
{
    int data = 0;
    int i;
    volatile double d = 1.0;

    while(fbswait()>=0) {
        data = !data;
        for (i=0; i<workload; ++i) d = d/d;
    }
}

```

worker.c

```

#include <time.h>
#include <stdio.h>

static int elapsed(struct timespec*,struct timespec*);

int outer = 50;
int inner = 10;
int threshold = 200;
int usecs;
int overruns;

double
work(void)
{
    volatile double d = 0.0;
    int i,j;
    for (i=0; i<outer; ++i) {
        for (j=0; j<inner; ++j) {
            d *= d;
        }
    }
}

int
main()
{
    struct timespec start;

```

```
struct timespec stop;
for (;;) {
    struct timespec t = {0,100000000};
    nanosleep(&t,0);
    clock_gettime(CLOCK_REALTIME,&start);
    work();
    clock_gettime(CLOCK_REALTIME,&stop);
    usecs = elapsed(&stop,&start);
    if (usecs > threshold) {
        printf ("Overrun %d\n",++overruns);
    }
}

static
int
elapsed (struct timespec * stop, struct timespec * start)
{
    int sec = stop->tv_sec - start->tv_sec;
    int nsec;
    if (stop->tv_nsec < start->tv_nsec) {
        sec--;
        nsec = 1000000000-(start->tv_nsec-stop->tv_nsec);
    } else {
        nsec = stop->tv_nsec-start->tv_nsec;
    }
    return sec * 1000000 + nsec/1000;
}
```