NightStar RT Tutorial

Version 4.1

(RedHawk™ Linux®)

concurrent

0898009-071
April 2008

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

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

NightStar's integrated help system is based on Qt's Assistant from Trolltech.

Preface

General Information

NightStar RT™ allows users running RedHawk 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	*NightView™ User's Guide*
0898398	*NightTrace™ User's Guide*
0898465	*NightProbe™ User's Guide*
0898480	*NightSim™ User's Guide*
0898515	*NightTune™ User's Guide*

Contents

Chapter 5 Using NightProbe

Chapter 6 Using NightTune

Chapter 7 Using NightSim

Appendix A Tutorial Files

Illustrations

1
Overview

NightStar RT™ is an integrated set of debugging tools for developing time-critical Linux® applications. NightStar RT tools run at application speed with minimal intrusion, thus preserving execution behavior and determinism. Users can quickly and easily debug, monitor, analyze, and tune their applications.

NightStar RT graphics-based tools reduce test time, increase productivity, and lower development costs. Time-critical applications require debugging tools that can handle the complexities of multiple processors, multi-task interaction, and multithreading. NightStar RT advanced features enable system builders to solve difficult problems quickly.

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 run as the root user, where indicated within this tutorial, or obtain appropriate privileges as detailed in the “Setting Up User Privileges” on page 1-2.

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
```

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.

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 list of files to modify is dependent on the list of methods that will be used to access the system. The following table presents a recommended configuration that will grant capabilities to users of the services most commonly employed in accessing a system.

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

/etc/pam.d File	Affected Services	Comment
remote	telnet rlogin rsh (when used w/o 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.
login	local login (e.g. console) telnet* rlogin* rsh* (when used w/o 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 with a command)	e.g. **rsh system_name a.out**
sshd	ssh	You must also edit **/etc/ssh/sshd_config** and ensure that the following line is present: **UsePrivilegeSeparation no**
gdm	gnome sessions	
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:

```
service sshd restart
```

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:

```
/usr/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 copied to **/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:

```
main()
{
   pthread_t thread;
   pthread_attr_t attr;
   struct sembuf trigger = { 2, 0, 0 };

   trace_begin ("/tmp/data",NULL);
   trace_open_thread ("main");

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

   ptrace_attr_init (&attr);
   Pthread_create (&thread, &attr, sine_thread, &data[0]);

   ptrace_attr_init (&attr);
   Pthread_create (&thread, &attr, cosine_thread, &data[1]);

   ptrace_attr_init (&attr);
   Pthread_create (&thread, &attr, heap_thread, NULL);

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

}
```

The program creates three threads and then enters a loop which cyclically activates each of two threads based on a common timeout. The third thread, `heap_thread`, runs asynchronously.

To build the executable

From the local **`tutorial`** directory, enter the following command:

```
cc -g -o app app.c -lntrace_thr -lpthread -lm
```

NOTE

The NightStar RT tools require that the user application is built with DWARF debugging information in order to read symbol table information from user application program files. For this reason, the **`-g`** compile option is specified. However, the tools can be used to debug programs without symbols with reduced functionality.

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.

Please read this chapter before proceeding to the first steps in using the tools, which follows in “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:

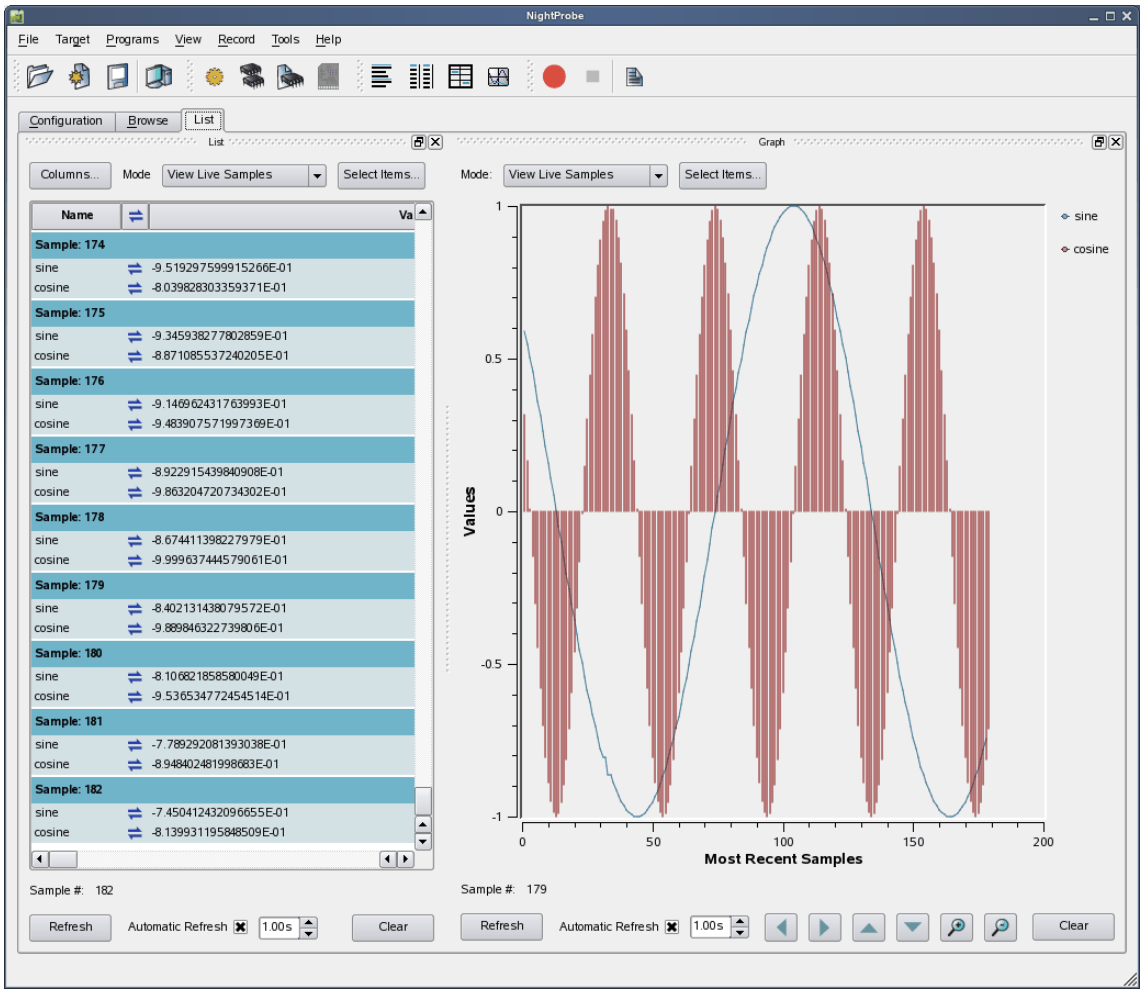


Figure 2-1. Viewing Page with List & Graph Panels

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.

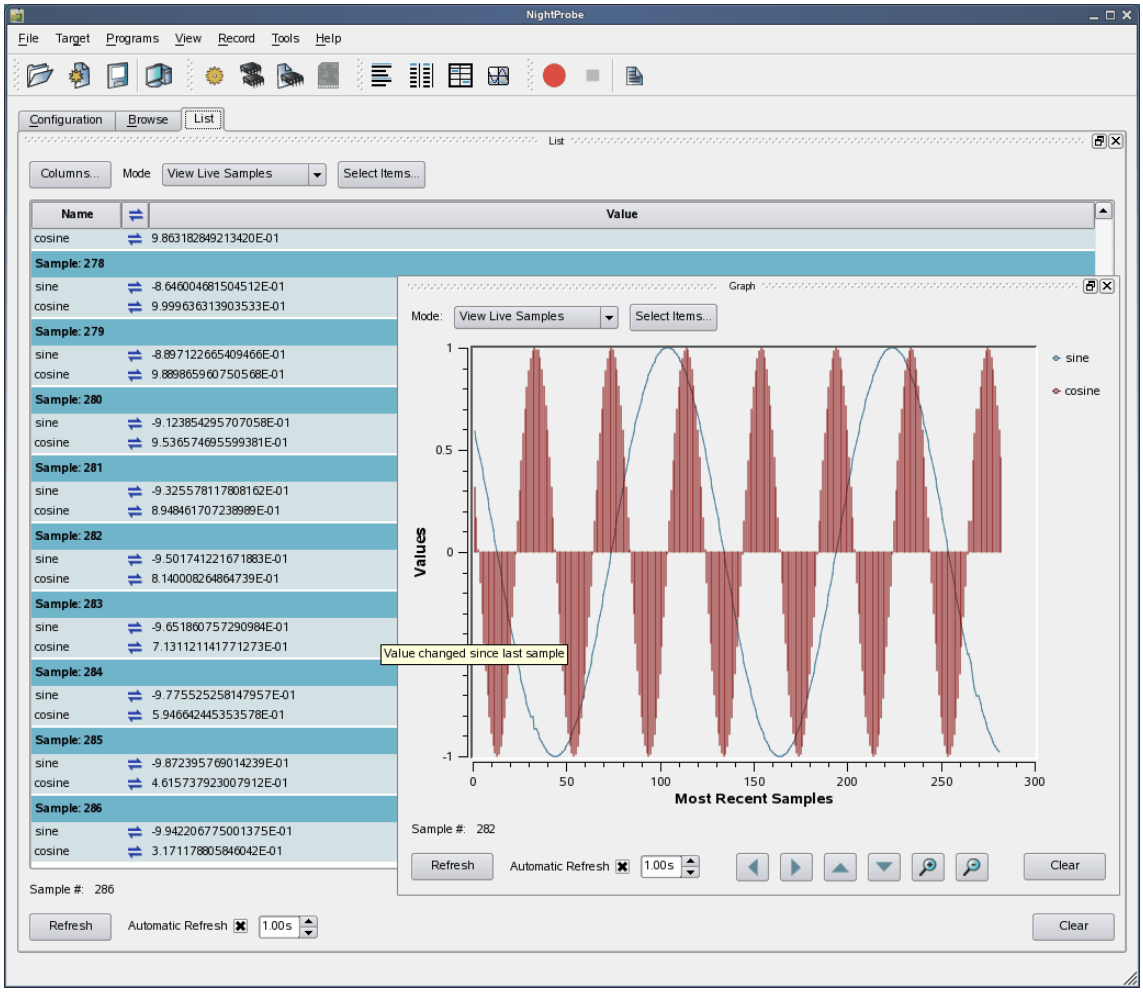


Figure 2-2. Panel Detaches from Page

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.

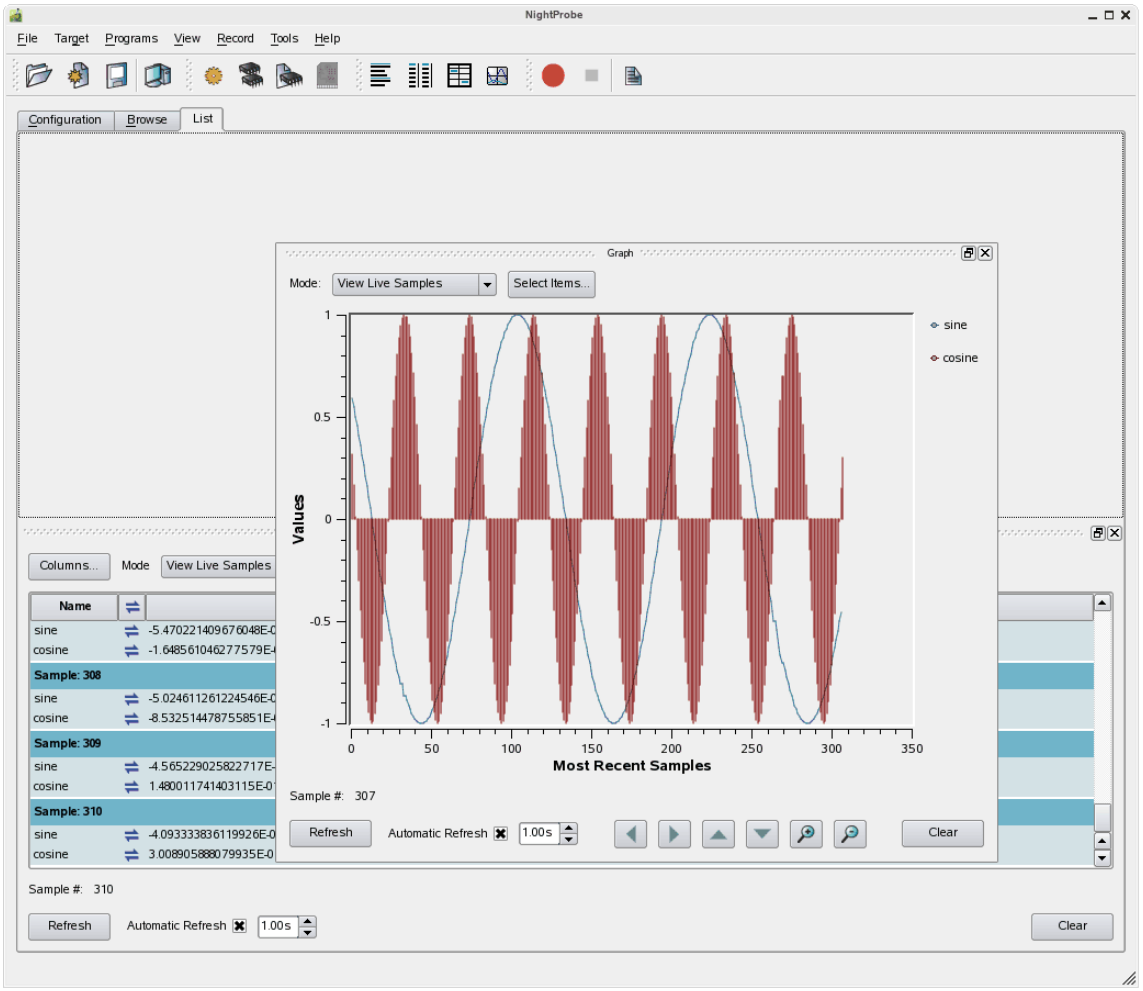


Figure 2-3. Panel Movement in Progress

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.

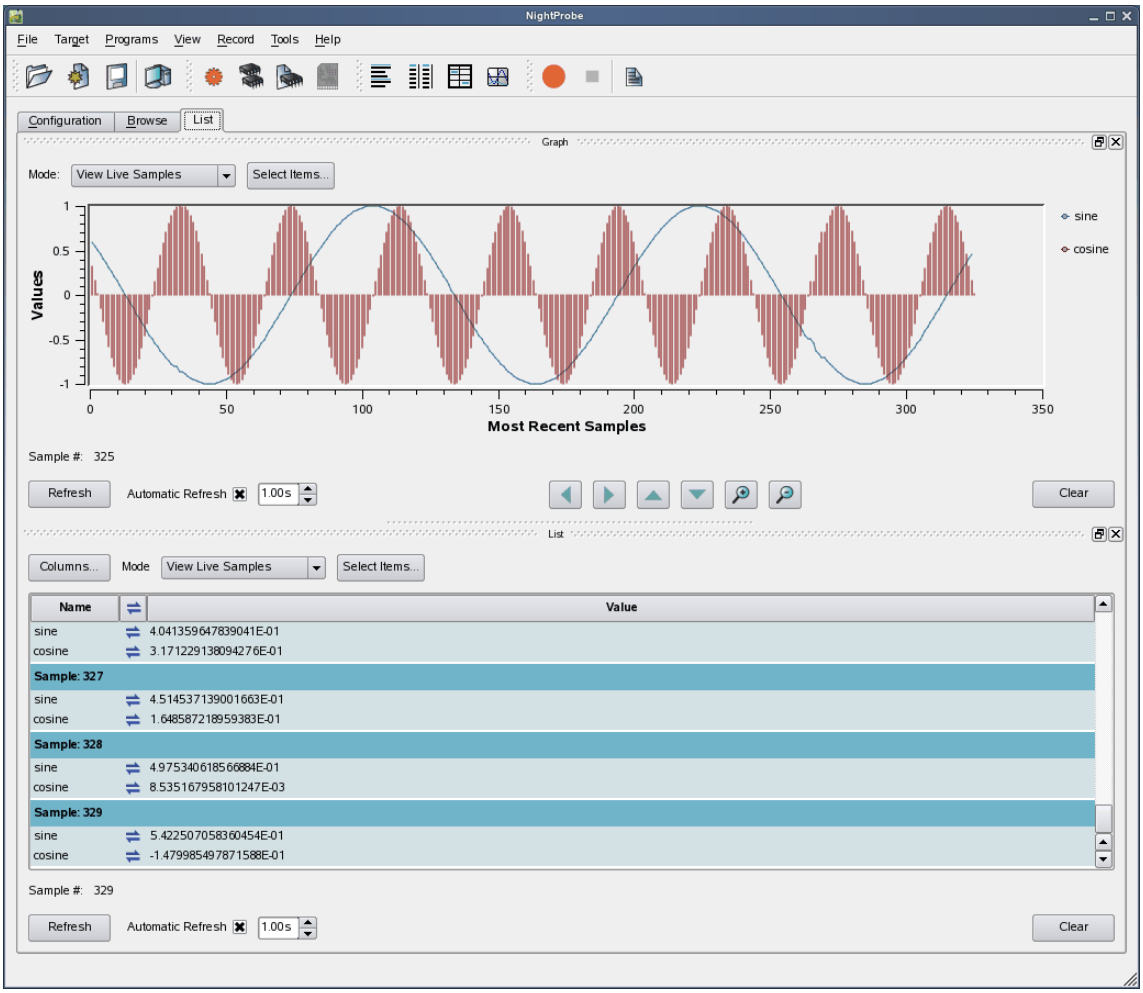


Figure 2-4. Graph Panel on Top of List Panel

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.

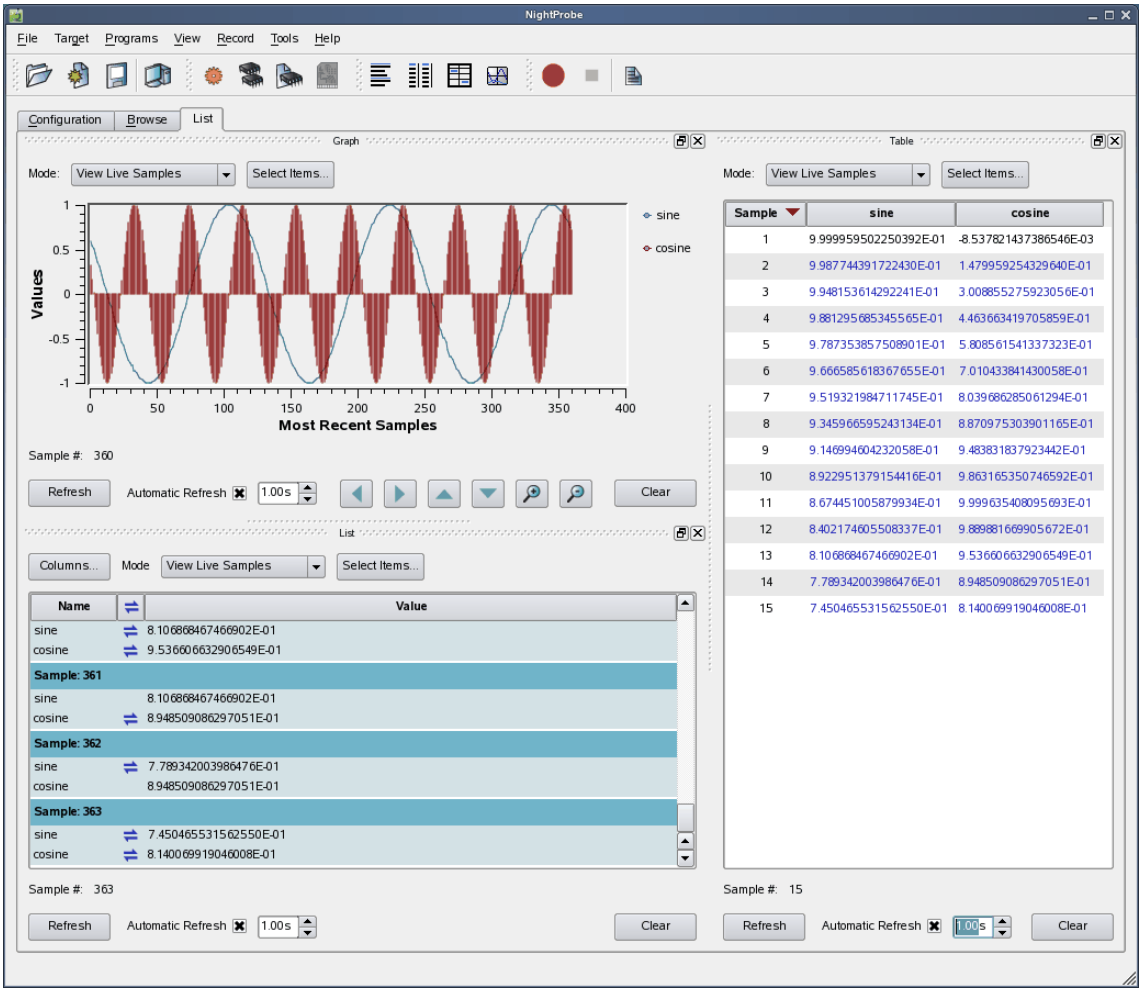


Figure 2-5. Table View added to Page

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.

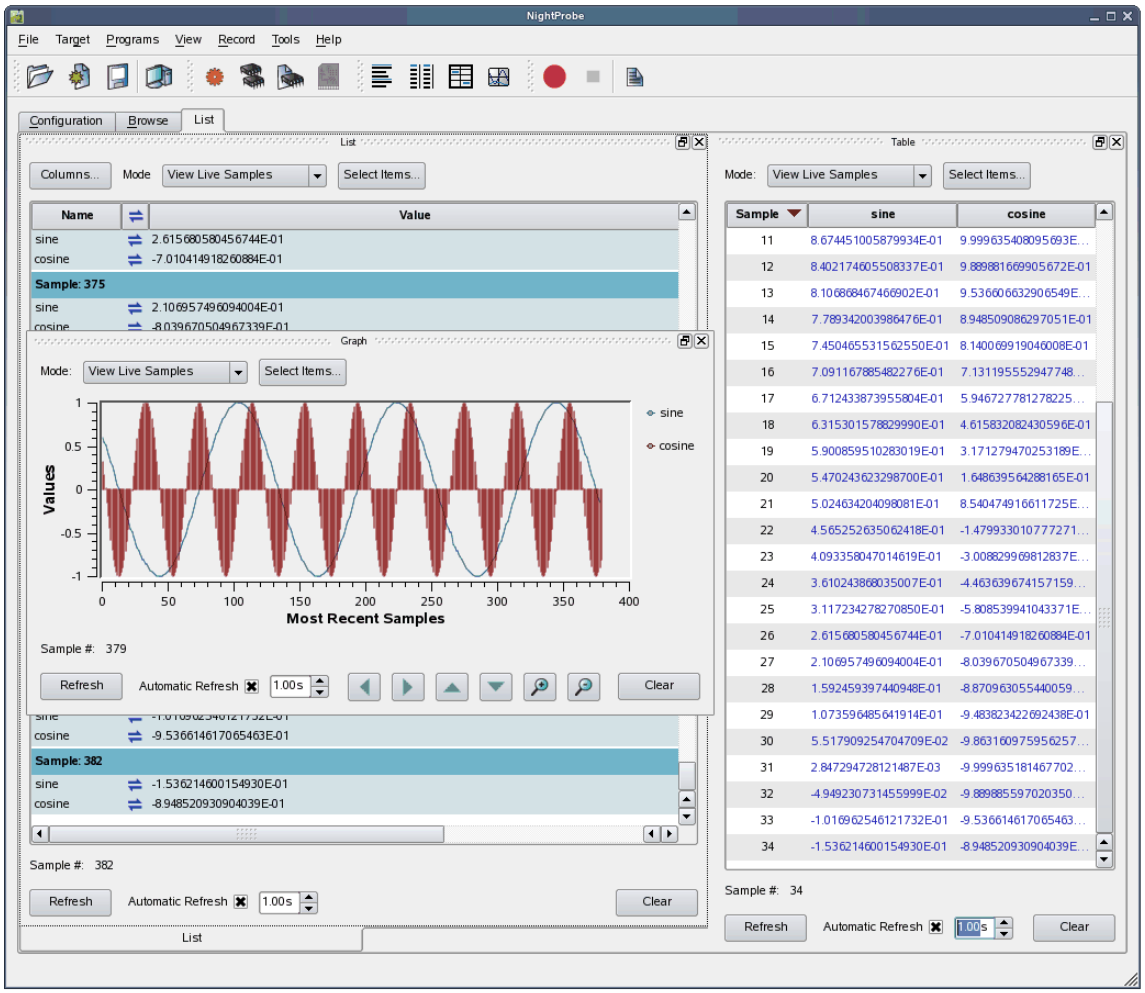
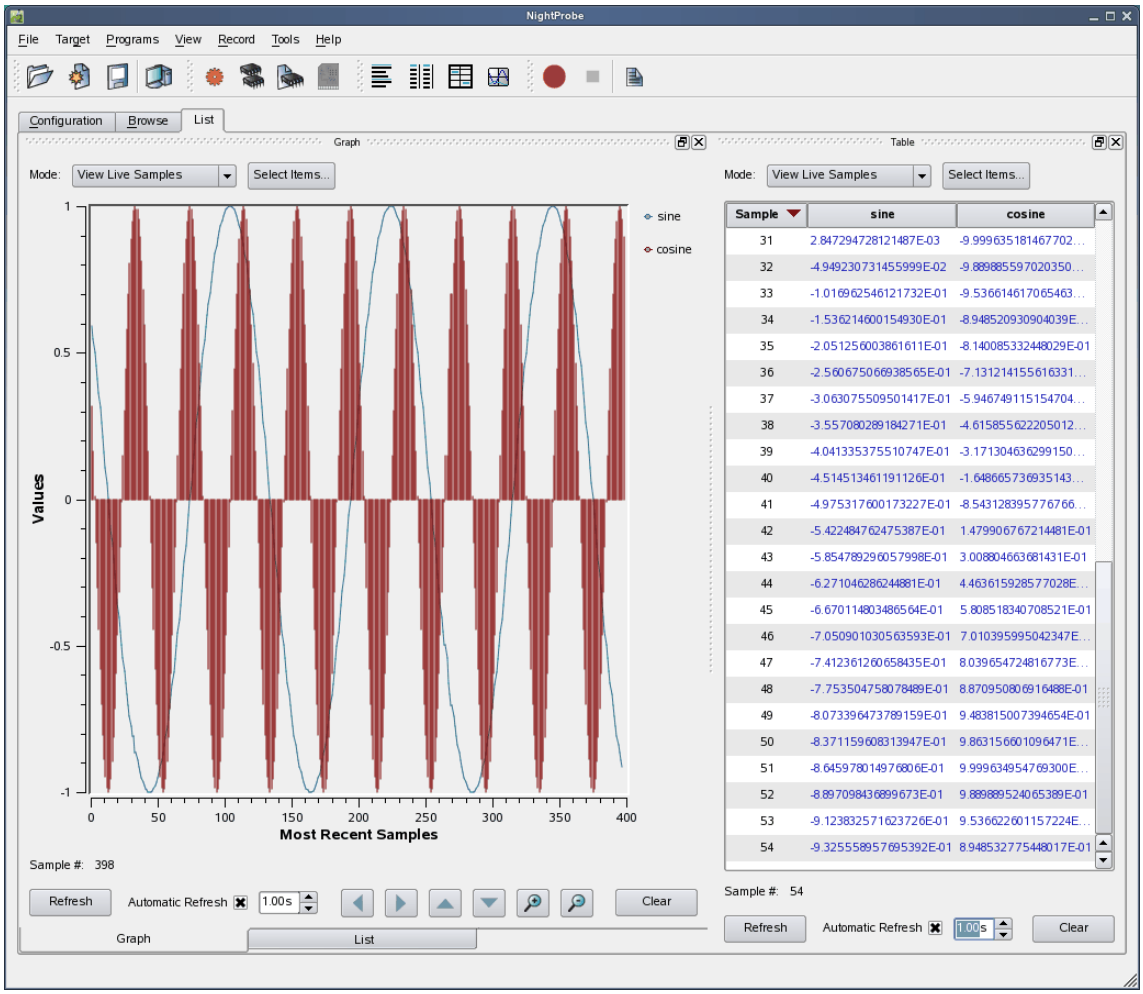


Figure 2-6. Panel in Motion Creating Tab

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.



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.

3
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
- hardware-assisted address traps (watchpoints)
- 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 memory 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

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.

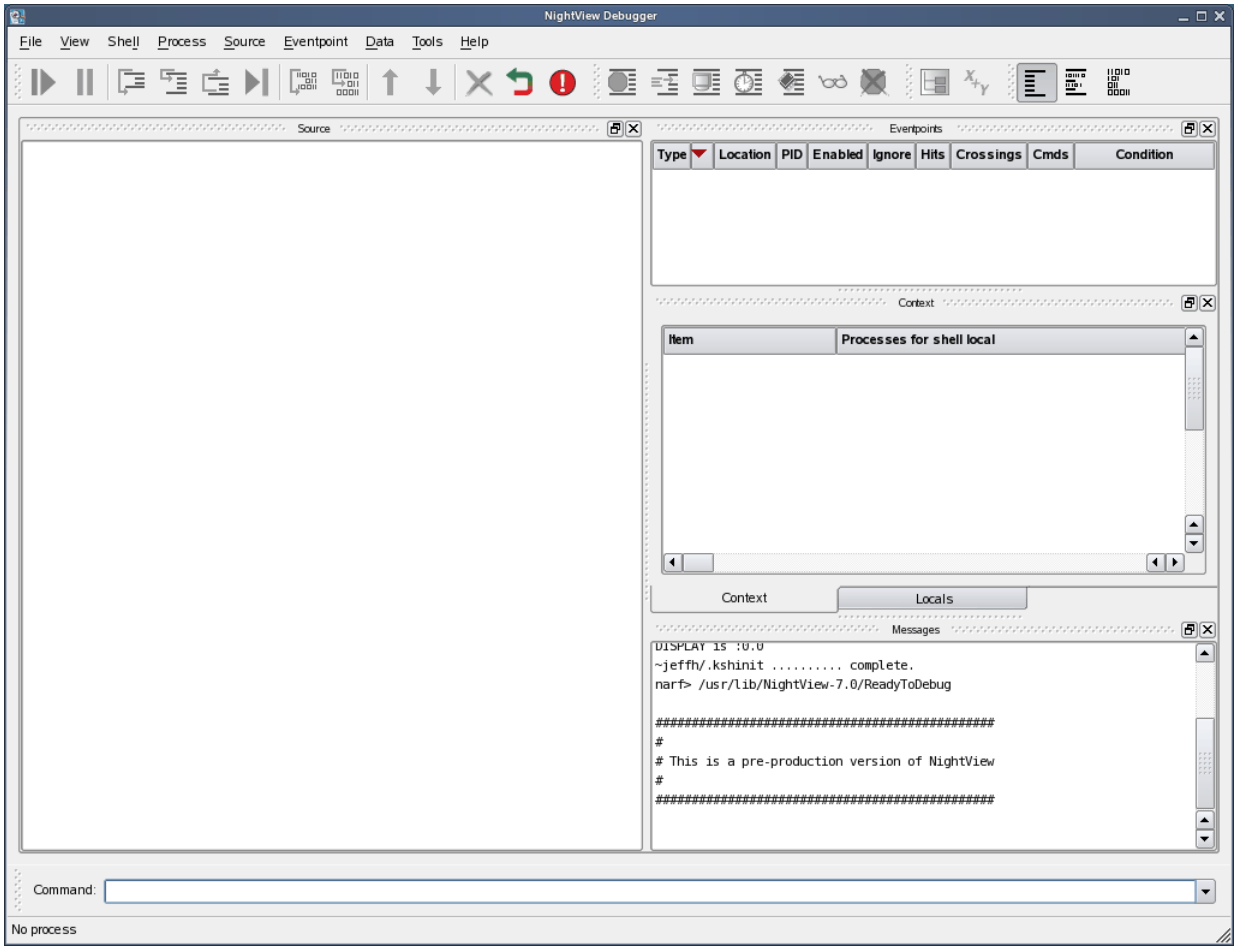


Figure 3-1. NightView Main Window

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. If so, press the NightView button to proceed.

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

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 Run… from the Process menu and entering:

  ```
  ./app &
  ```

  in the text field of the Run on local dialog.

- Press OK to close the dialog and run the program.

IMPORTANT

Ensure that you used the & character to put the program into the background. The command is echoed in the Messages panel. If you forgot to specify the & character, type **kill** in the Command field and then run the process again.

It is not normally necessary to run a process in the background when debugging under NightView; in fact, it is an unusual thing to do. We do this in the tutorial because we will detach from the process and close NightView at the end of this chapter; yet we want our **app** program to continue executing for the remainder of the tutorial.

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

When the **app** program begins to execute, NightView displays the source in the source panel and stops the program at the first line of code.

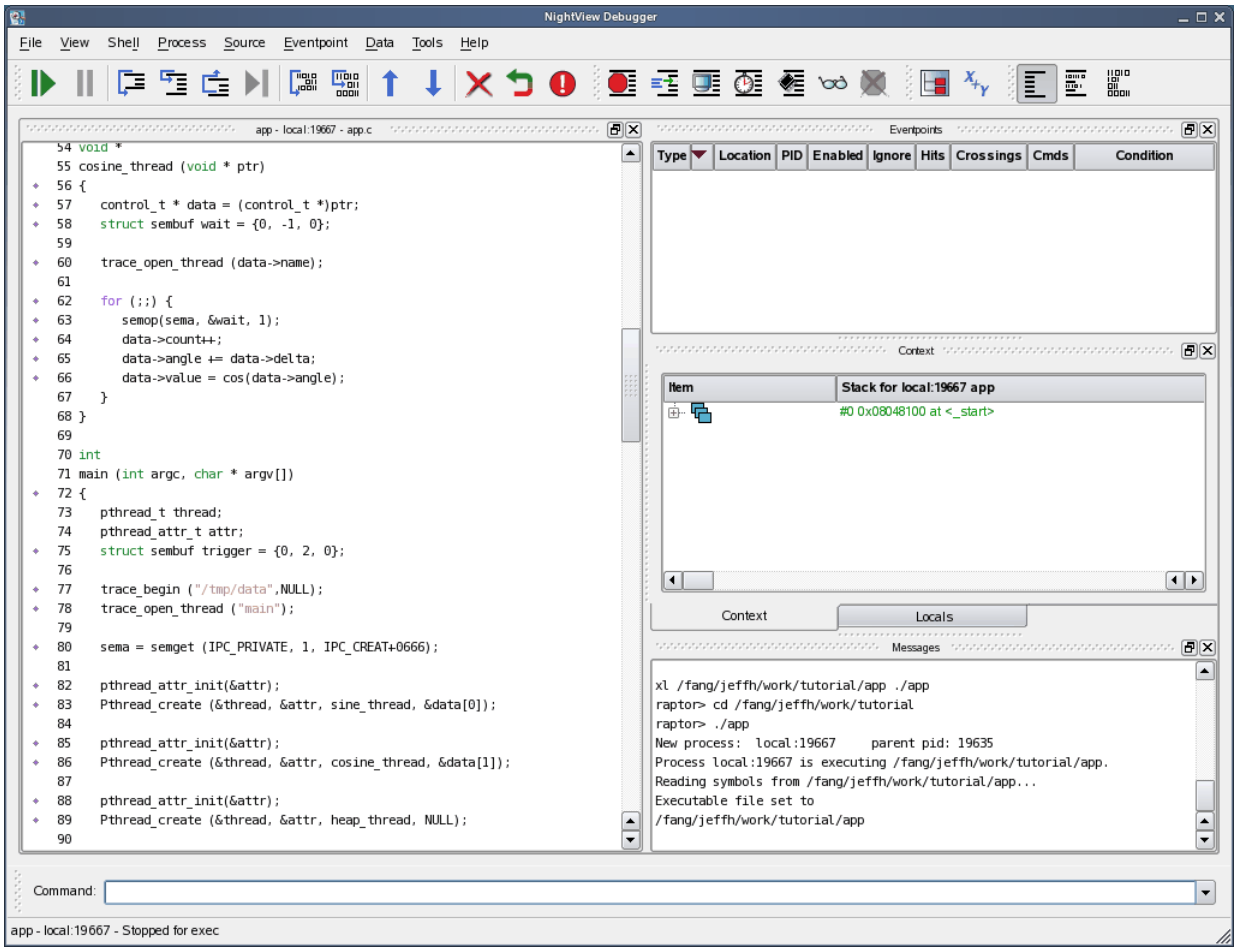


Figure 3-2. app Program Loaded

IMPORTANT

Do not resume execution of the program at this time.

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

Heap Debugging

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.

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 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 utility routine 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 process executes instructions where the heappoint is inserted. 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

One limitation of heap debugging is that it requires that you activate the debugging before any allocations occur in your program. 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.

NOTE

If you have mistakenly resumed execution of the program already, kill the program and restart it in the NightView main window. Type `kill` in the Command area and press Enter. Type **`./app &`** and Enter in the dialog shell.

- Select the Debug Heap... menu option from the Process menu in the NightView main window.

  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:

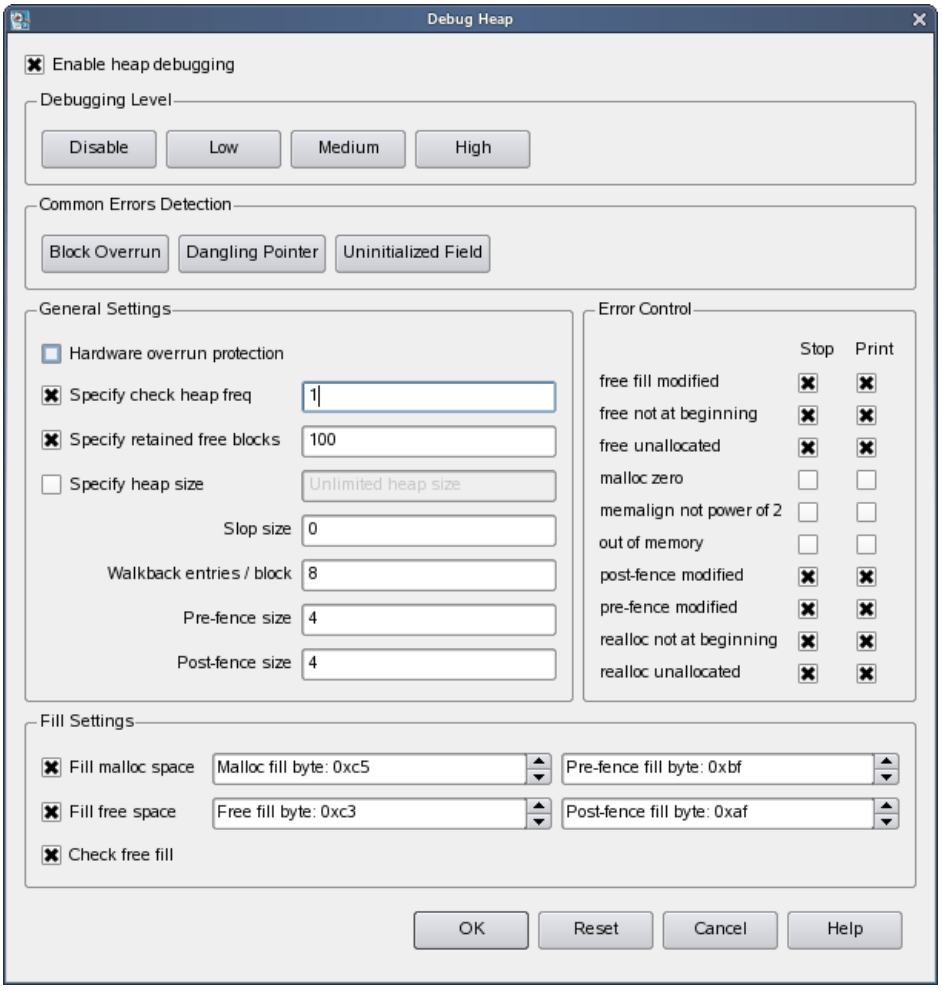


Figure 3-3. NightView Debug Heap Dialog

- 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.

Controlling the app Program

The third 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 user errors relating to dynamic memory.

Let's set a breakpoint on line 115.

- Scroll to line 115 in the source window:

```
sleep(5);
```

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

NOTE

Optionally, you could set a breakpoint on line 115 by using either the Set Breakpoint menu item from the Eventpoint menu or enter the following command in the Command panel of the NightView main window:

```
break app.c:115
```

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 subsequently written, NightView can detect this.

- Resume the process and let it reach the breakpoint on line 115 by pressing the Resume icon on the Process toolbar:

NOTE

Alternatively, you can resume the process by typing **`resume`** into the Command field:

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 field:

```
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 heappoint inserted by the user, or at a subsequent heap operation (based on the **`frequency`** setting of the **`heapdebug`** command), in this case on line 155.

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 155
```

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.

- Select Heap Information from the Data menu.

  The Data panel is added to the NightView main window in the same location as the Locals and Context panels. A new tab will be created for the Data panel.

- Click on the newly-created Data tab.

- 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.

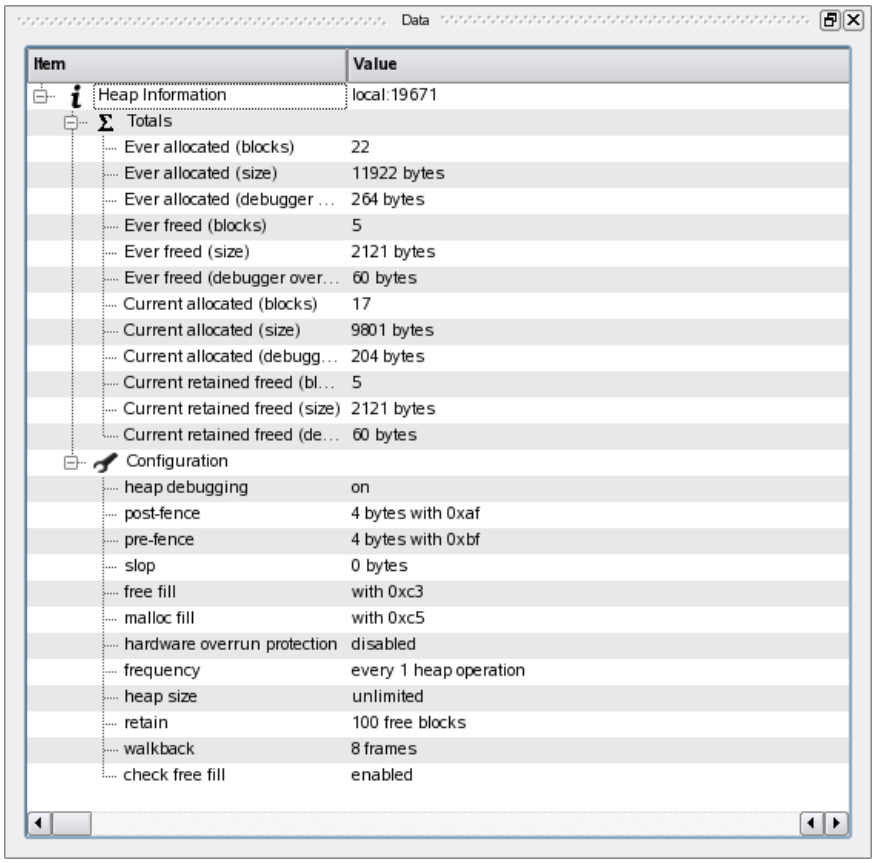


Figure 3-4. Heap Totals and Configuration

NOTE

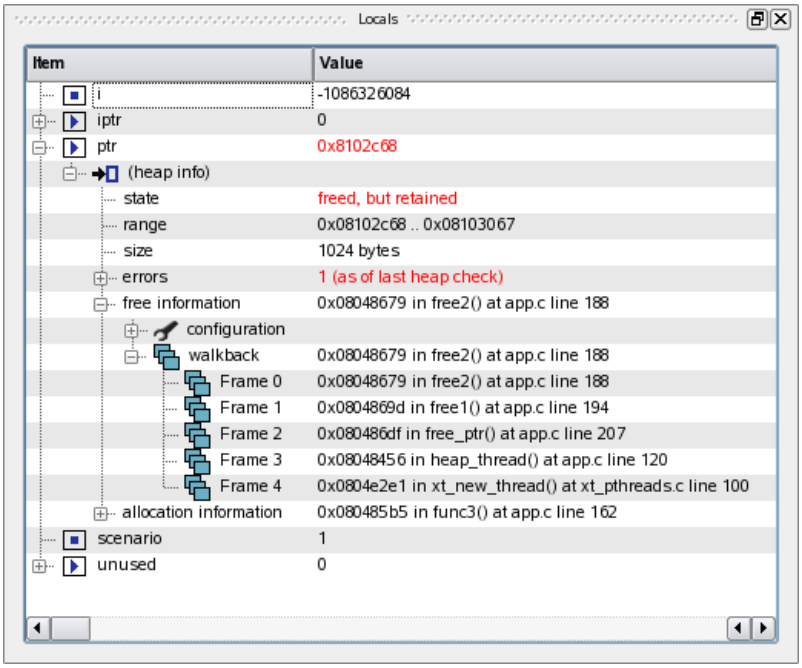
In general, all information in the Data panel is updated whenever the process being debugged stops.

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

The list of items in the `Locals` panel changes each time the process stops to represent 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



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 115:

  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 127
```

Another way of obtaining information about the heap block in question is to use the **info memory** command. It provides textual output of 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:

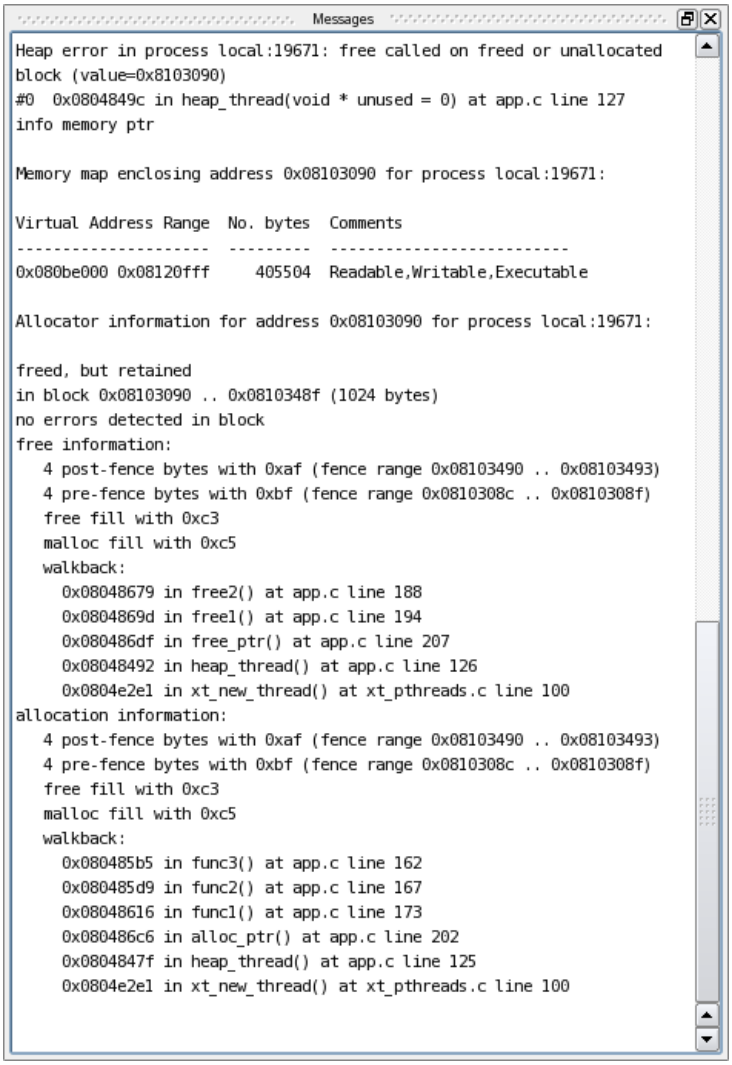


Figure 3-5. info memory Command Output

Note that there 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 115:

```
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 155
```

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.

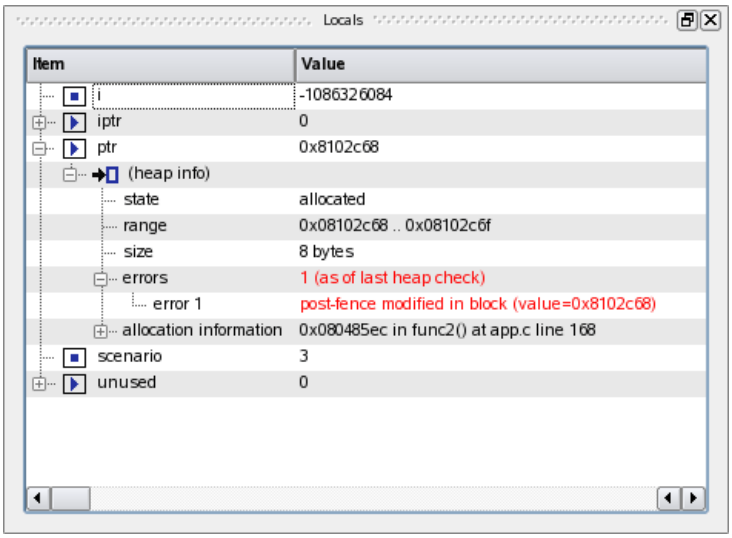


Figure 3-6. Heap Error Description

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 115:

  resume

- Tell NightView to stop whenever a SIGSEGV is sent to the procesa 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(int*),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 138
```

The `malloc_fill` setting of the **heapdebug** command causes NightView to fill blocks being allocated with a specific byte pattern, in this case `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 field 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 115:

  ```
  resume
  ```

- Set the variable `scenario` to 5:

  ```
  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 115 is reached.

- At that time, specifically request a leak report by selecting Heap Leaks… from the Data menu, 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 142 of **`app.c`**.

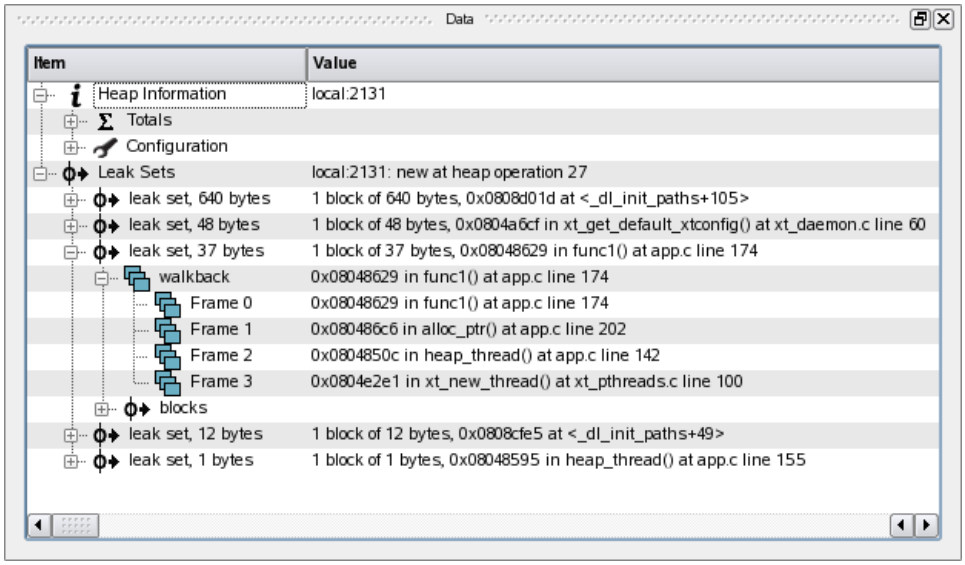


Figure 3-7. Heap Leaks Display

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 (select Heap Leaks… from the Data menu).

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
```

This causes additional allocations to be made.

The process will stop again when the breakpoint on line 115 is reached.

- At that time, specifically request an allocation report by selecting Still Allocated Blocks... from the Data menu, 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 indicating that it was allocated by the `heap_thread()` routine on line 147 of **`app.c`**.

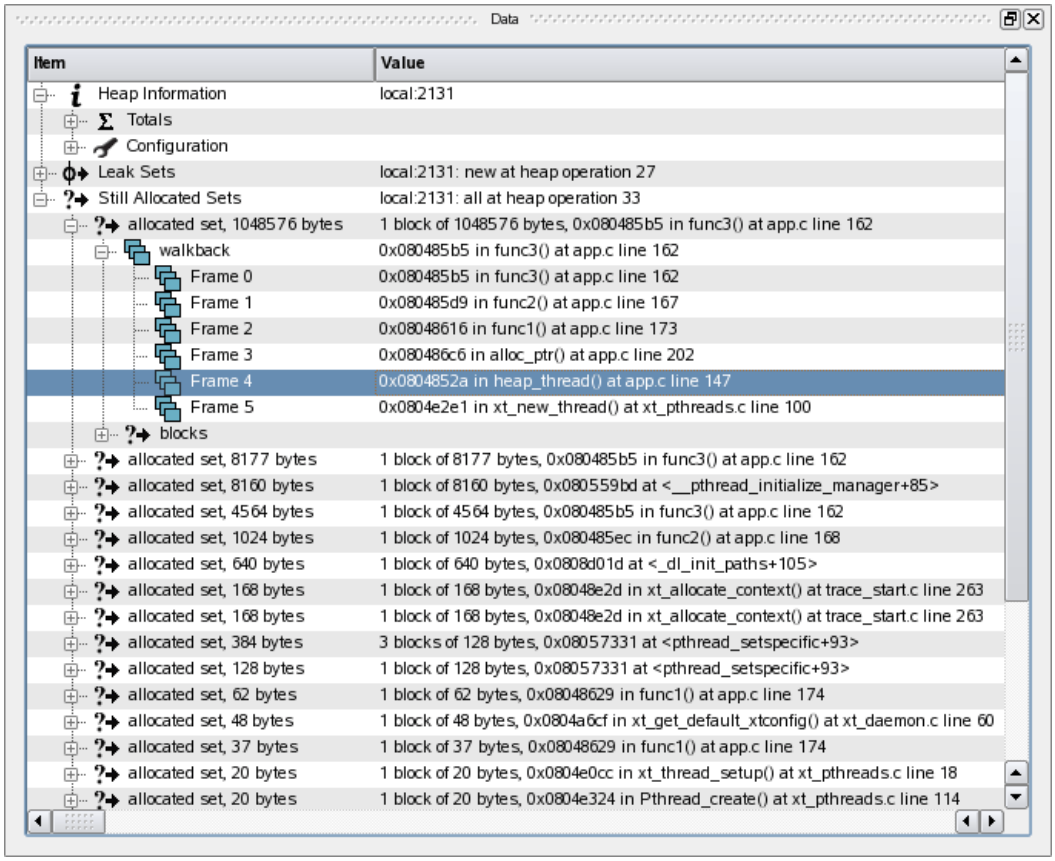


Figure 3-8. Still Allocated Blocks Display

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 (select Still Allocated Blocks... from the Data menu).

Disabling Heap Debugging

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

```
heapdebug off
```

This concludes the tutorial's topic on heap debugging. We will now continue on to other capabilities of NightView.

Debugging Multiple Threads

At this point in the tutorial the user application should be stopped at line 115 in **`app.c`**.

NOTE

If the application is not stopped at line 115, set a breakpoint on line 115 in **`app.c`** and resume the process until it stops on that line number. Refer to the previous sections for instructions on setting breakpoints and resuming the process.

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

When the application hits a breakpoint or is otherwise stopped by NightView, all threads in the application will stop. Similarly, when NightView resumes execution of a thread, all threads will resume execution.

- Click on the Context tab to raise the Context panel.
- Expand the thread which is displayed in green.
- Expand the first item in the walkback list that appeared as a result of the last step

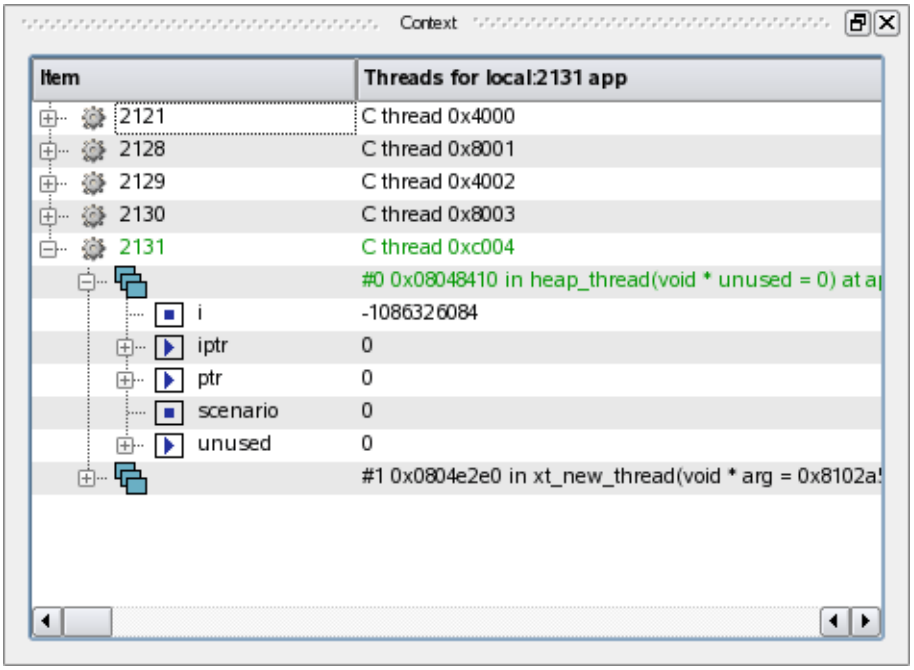


Figure 3-9. Context Panel With Stack Frames Expanded

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

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 thread ID displayed in green in the Context panel. A thread ID is a hexadecimal number representing the thread -- it is assigned by the threads library upon thread creation.

You can switch to the context of other threads by clicking on the thread of interest.

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

```
info threads /v
select-context thread=0xb7245bb0
```

- Switch to the context of the thread executing `sine_thread()` by clicking on it (it is usually the third thread from the top).

  The source displayed in the NightView main window changes to line 46 on a call to `semop()`.

NOTE

> It is possible that the context of the thread in question could be executing on any line in the range of 45-49.

The gray triangular arrow 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.

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
  ```

- Delete the breakpoint on line 115 by right-clicking on that breakpoint in the Eventpoints panel and selecting Delete or by issuing the following command:

  ```
  clear app.c:115
  ```

  before proceeding to the next section.

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.

- Right-click on line 46 and select Set eventpoint from the pop-up menu and select Set Monitorpoint… from the sub-menu.

NOTE

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

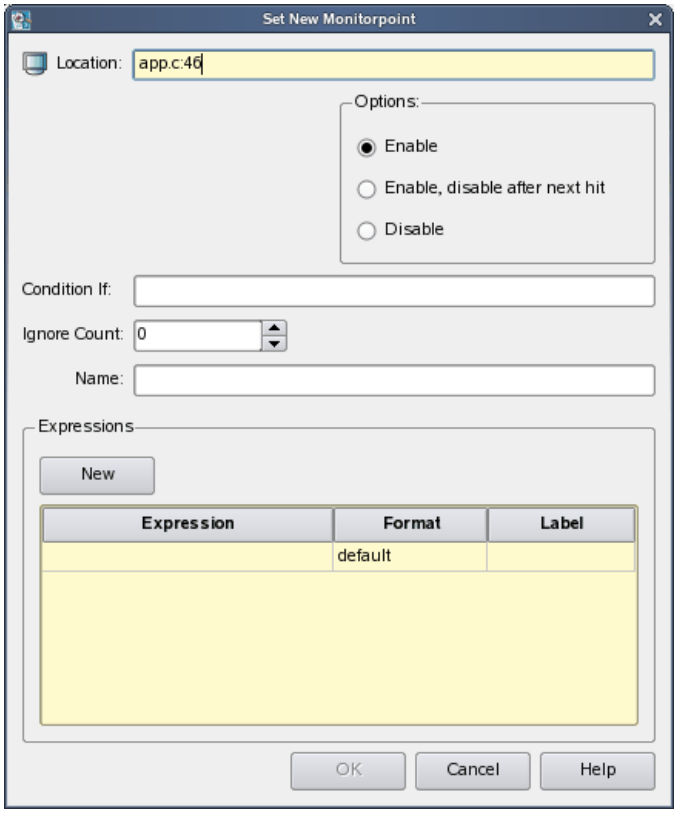


Figure 3-10. Monitorpoint Dialog

- Ensure that the Location text field has `app.c:46`, correcting if it need be.

- Enter the following:

  ```
  data->count
  ```

  in the text field below the Expression column head, but do *not* press the Enter key yet.

- You can control the format in which the value is displayed by clicking the option list under the format column. Select hexadecimal from the list.

- 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, simply click in the next row under the Expression column.

- 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 there:

  ```
  sine value
  ```

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

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

- Likewise, set a monitorpoint on line 63 with the same commands as in the previous monitorpoint, substituting `cosine` for `sine` in the optional `id` parameter.

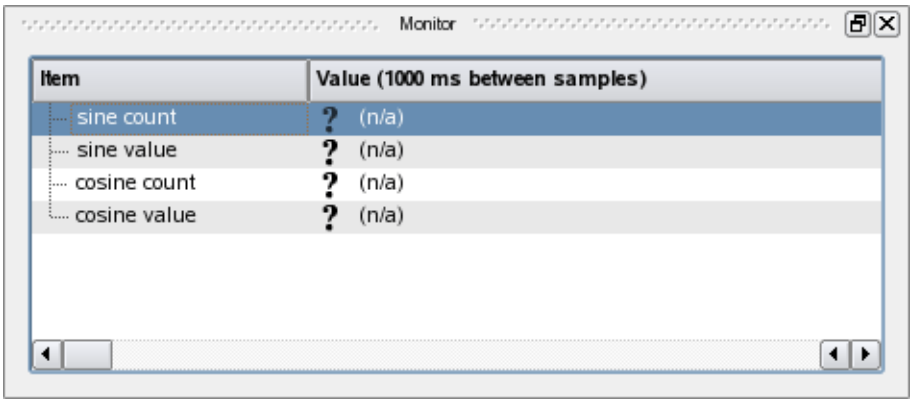


Figure 3-11. NightView Monitor Panel

- Resume execution of the process.

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

  The values are sampled whenever line 46 or 63 are executed, respectively. NightView displays the latest set of values in the Monitor panel at a user-selectable rate.

NOTE

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

All subsequent debugging operations in this tutorial can be done without stopping the process!

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. 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 Count column of the first row of the Eventpoint panel.
- Enter 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.

- Enter the following commands in the Command field of the NightView main window:

```
monitor app.c:93
   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 time line 93 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:

```
l app.c:48
```

- Scroll the source file displayed in the NightView main window and right-click on line 48:

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

  and select Set eventpoint from the pop-up menu and select Set Patchpoint… from the sub-menu.

NOTE

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

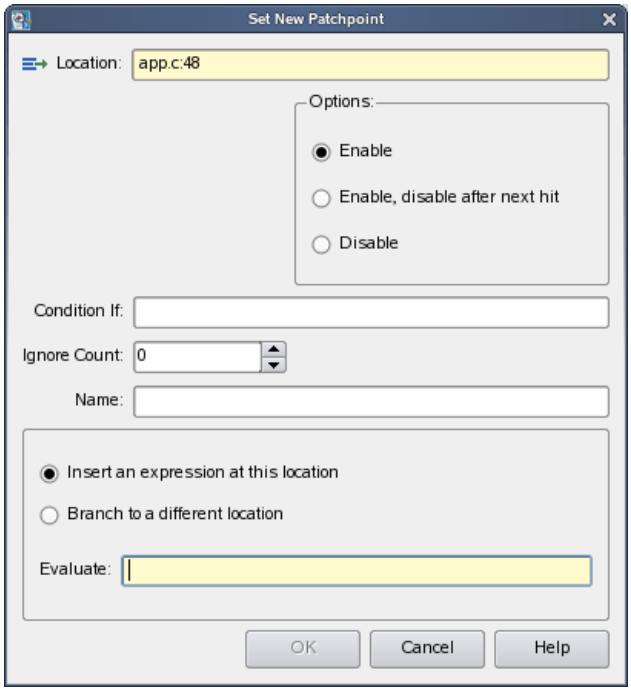


Figure 3-12. Patchpoint Dialog

- In the Location text area, ensure the text indicates `app.c:48`.
- 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:49`

  then press the OK button.

This will effectively cause the application to skip execution of line 48, 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` value 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:49 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 now also is 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 pop-up menu.

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

Adding and Replacing Functions Dynamically

NightView provides the ability to dynamically add new functions to the application being debugged, as well as to replace existing functions.

- In a terminal session outside of NightView, compile the **`report.c`** source file which was copied into your current directory in the initial steps of this tutorial:

  ```
  cc -g -c report.c
  ```

- Load the new module into the program using the following command in the Command panel of the NightView main window:

  ```
  load report.o
  ```

NOTE

The source displayed in the NightView main window may change as a result of the **`load`** command. This annoyance will be addressed in the future.

We have added a simple function which prints information to **`stdout`**. The function could have been arbitrarily complex and referenced any variable in the application. The only limitation is that the function cannot reference symbols that are absent from the module being loaded and are not already in the user application.

- Issue the following command to see the source code for the function `report()`:

  ```
  l report.c
  ```

You will see that the `report()` function expects a pair of arguments whose types are `char *` and `double`, respectively.

- Go back to the application source file by issuing the following command:

  ```
  l app.c
  ```

We will install a new patchpoint which will call the newly added function.

- Set a patchpoint on line app.c:63 with the following expression:

  ```
  report("cos",data->value)
  ```

The program is now generating output to **`stdout`** in the Messages panel of the NightView main window as calls to the `report()` function are executed.

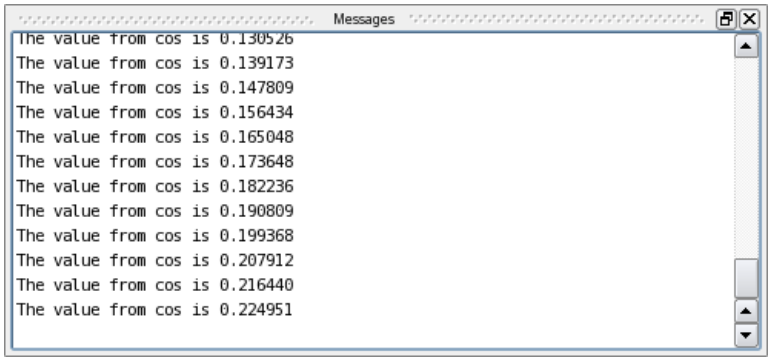


Figure 3-13. Result of Patching in Call to Newly Loaded Function

- Disable the patchpoint that was just added by clicking its Enabled cell in the Eventpoint panel and selecting Disable. You must click outside of the cell in order for this to take effect.

Finally, we will replace a function that already exists in the application.

- In a terminal session outside of NightView, list the contents of the source file **`function.c`** which was copied into your current directory in the initial steps of this tutorial, and compile it with the following commands:

```
cat function.c
cc -g -c function.c
```

- Now load the replacement code by entering the following command in the Command panel of the NightView main window:

```
load function.o
```

Note how the Monitor panel value for the `FunctionCall()` value no longer pertains to the value computed by the application, but rather is a monotonically increasing number as per the source file **`function.c`**.

- Return the NightView main window source panel to the **`app.c`** source file via the following command:

```
l app.c:40
```

Using Tracepoints

The last portion of NightView we will cover in this tutorial is integration with NightTrace.

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

The current limitation on tracepoints is that the application must already have linked with the NightTrace API library and has made a single API call to initiate tracing.

Our application satisfies this requirement.

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 48:

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

  and select Set eventpoint from the pop-up menu and select Set Tracepoint… from the sub-menu.

NOTE

Alternatively, you could launch the dialog by selecting Set Tracepoint… from the Eventpoint menu or click on the Set Tracepoint icon on the toolbar to launch the Set New Tracepoint dialog.

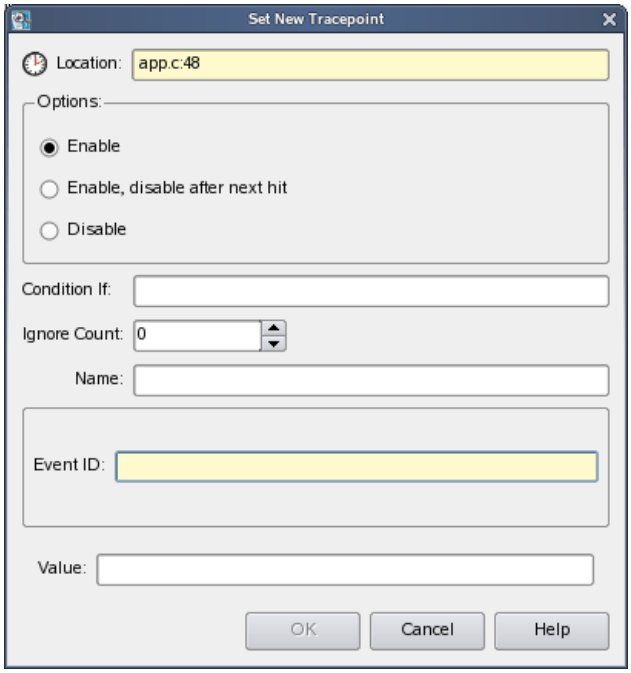


Figure 3-14. Tracepoint Dialog

- In the Location: text field ensure that **`app.c:48`** is displayed.
- In the Event 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:46`** 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:63`** 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 which is described in the next section of this tutorial.

- Launch NightTrace by selecting the NightTrace Analyzer menu item from the Tools menu of the NightView main window.

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.

- Stop the processes by typing the following into the Command field:

```
stop
```

- Select the Detach option from the Process menu
- Select the Exit option from the File menu to exit NightView.

Conclusion - NightView

This concludes the NightView portion of the NightStar RT Tutorial.

4
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 users to zoom, search, filter, summarize, and analyze those events in a wide variety of ways.

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

NightTrace users can 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, execute the steps in “Using Tracepoints” on page 3-31 before beginning this section of the tutorial.

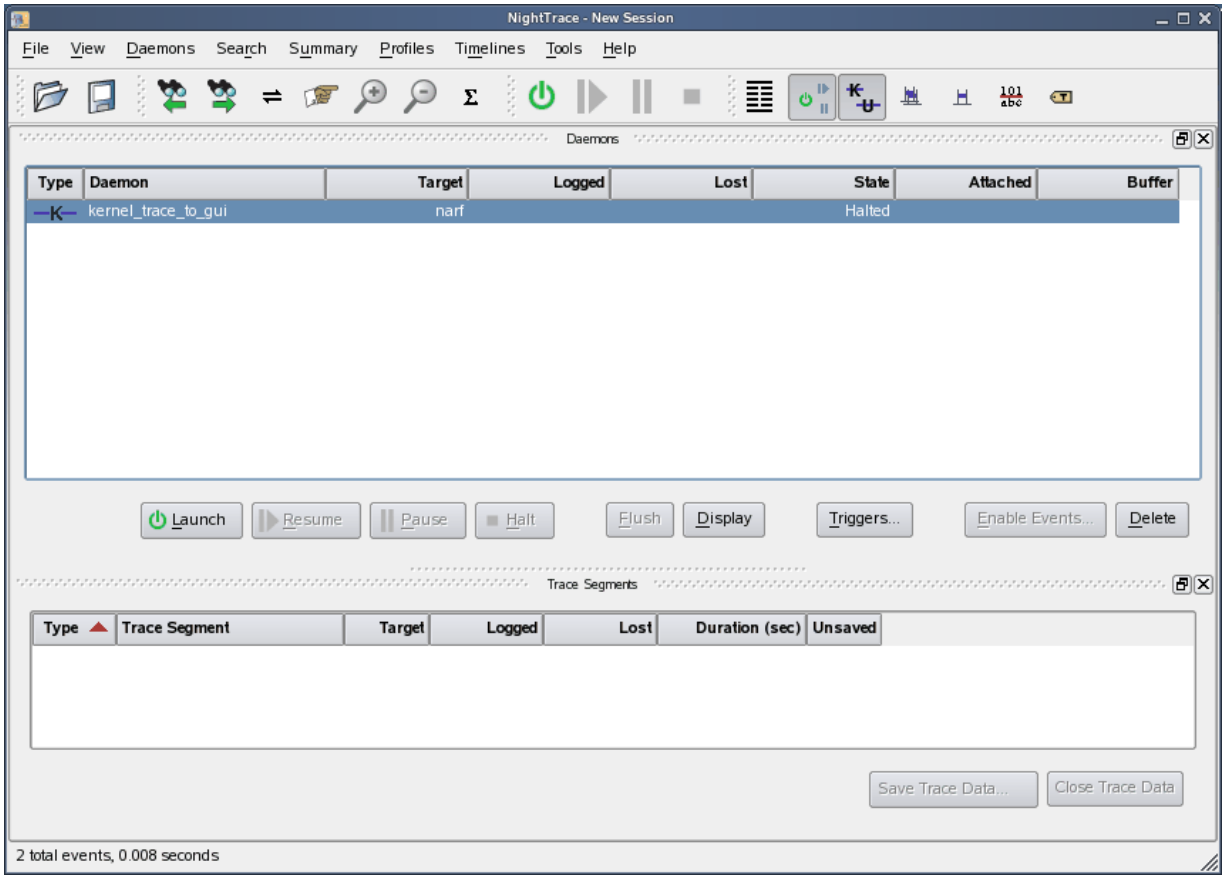


Figure 4-1. NightTrace main window

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

Table 4-1. NightTrace Panels

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, which have not been shown yet.

NOTE

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

Configuring a User Daemon

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

User trace events are generated by user applications that use the NightTrace API.

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

- Select the Import… menu item from the Daemons menu.

  The Import Daemon Definitions dialog is presented:

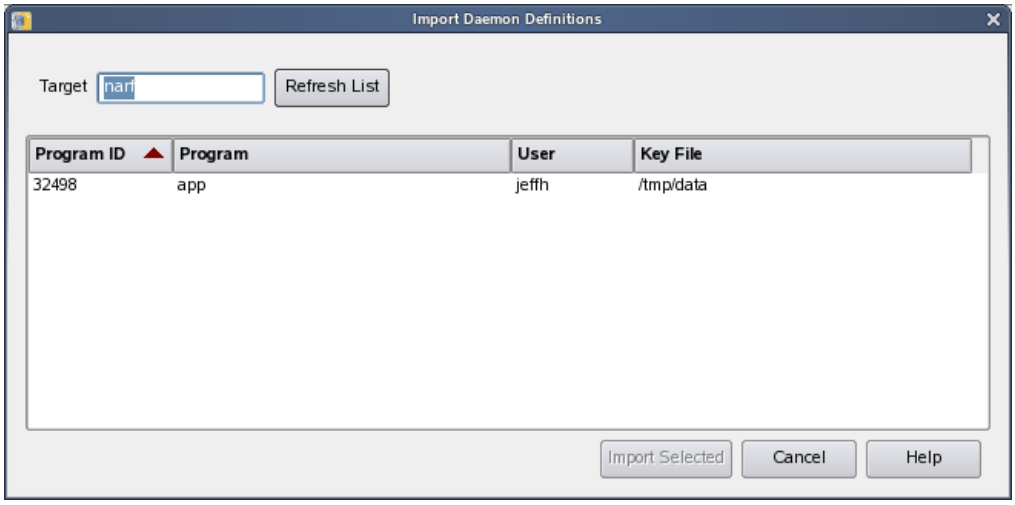


Figure 4-2. Import Daemon Definitions Dialog

The Import Daemon Definitions dialog allows the user 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 in “Using Tracepoints” on page 3-31.

In the Daemons panel, the counts in the Logged and Buffer columns will begin to change.

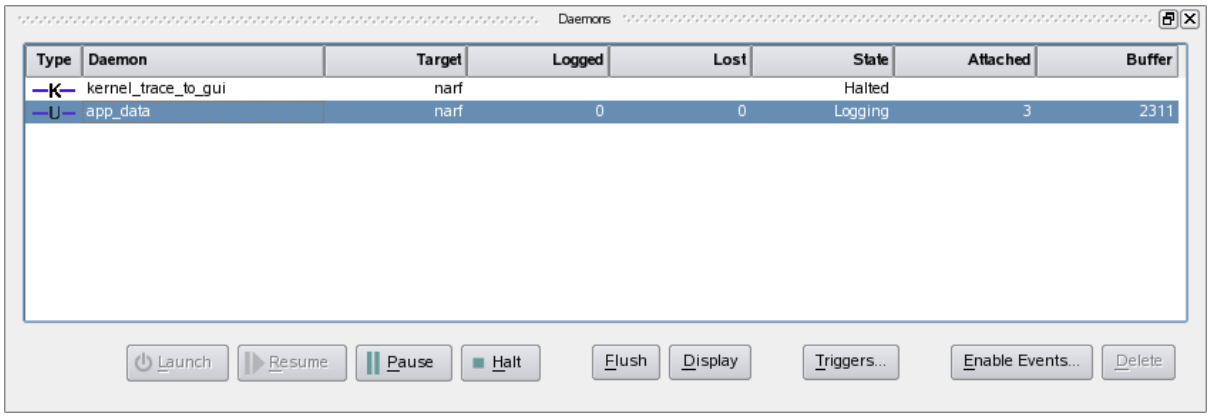


Figure 4-3. Logging Data

NOTE

A tabbed page is created in the NightTrace main window when Launch is pressed. This page is an automatically customized page containing a list of the events logged and a timeline for graphical representation of those events.

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

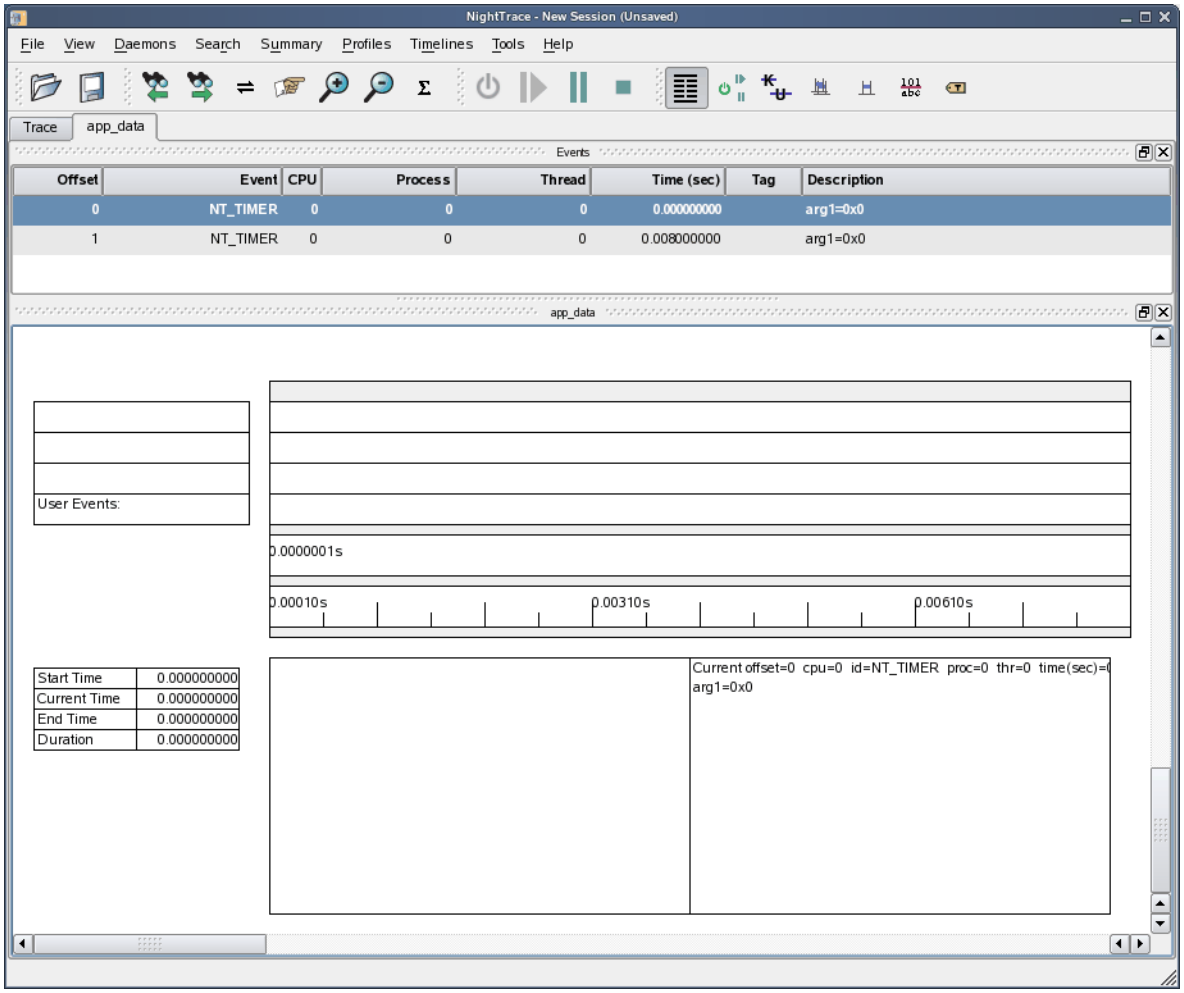


Figure 4-4. app_data Page

Initially, the panels will 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 all the way out in a timeline.

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

  The Events list will be populated with the events currently logged and the timeline will graphically display those events.

NOTE

If you plan to leave the tutorial for an extended period of time before returning, press the Pause button to temporarily prevent the collection of trace points. When you return, press the Resume button.

Using NightTrace Timelines

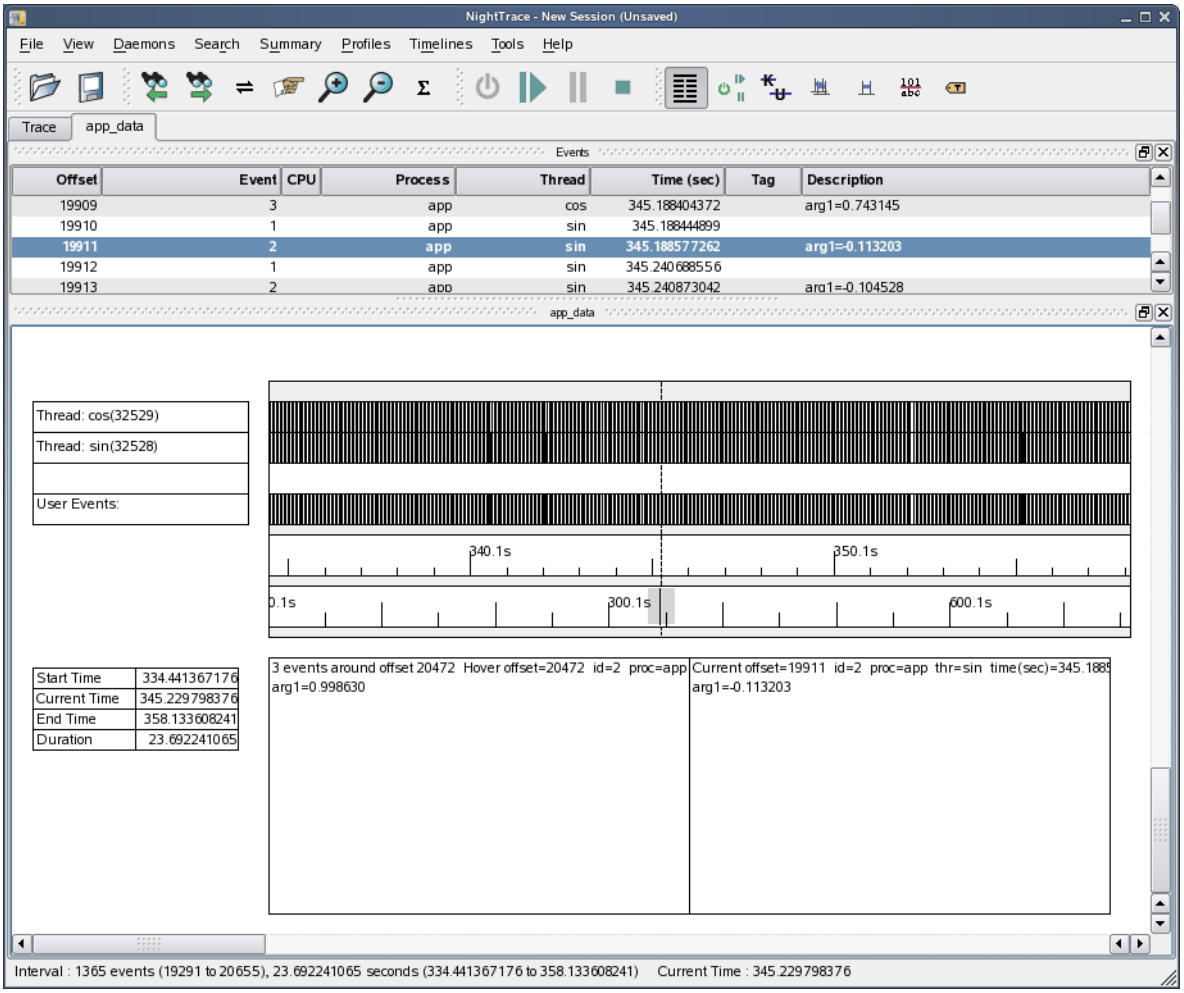


Figure 4-5. NightTrace Timeline

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. In addition, there is a user events graph near the bottom that shows events for threads.

NOTE

You may see a blank label and graph in your timeline. This is likely the label and graph for the main thread. The contents of the label are not shown until at least one event is logged by the main thread. In our application, the main thread does not log events so the row will remain blank.

In “Using Tracepoints” on page 3-31 in the Using NightView section, we inserted tracepoints into the `sine` thread, which registered itself with the string “`sin`”.

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.

- Press the DownArrow key repeatedly until you can see individual lines in the graph
- Press the UpArrow 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 on the right side changes to reflect the event closest to the left of the current timeline.

Moving The Interval

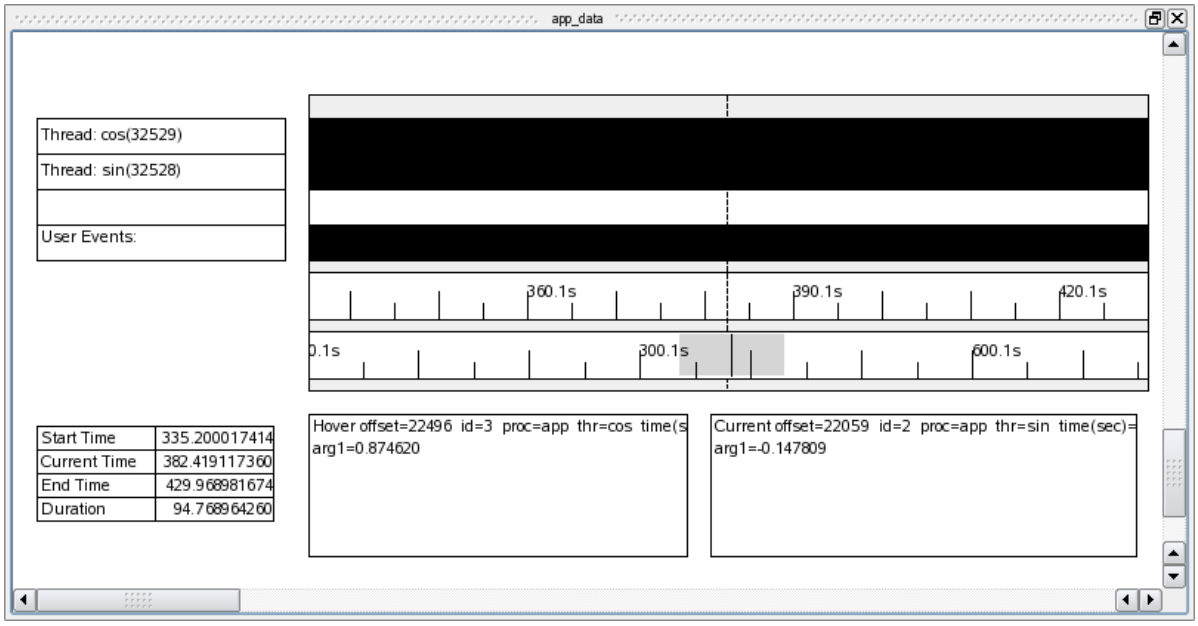


Figure 4-6. Timeline Interval Panel

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.

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 control 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 RightArrow 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 out or hold the Right key down until you can see the timeline move.

  Alternatively, pressing the LeftArrow key causes the current timeline to go to the previous event.

- Press Ctrl+RightArrow

  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 are in the control ruler moves.

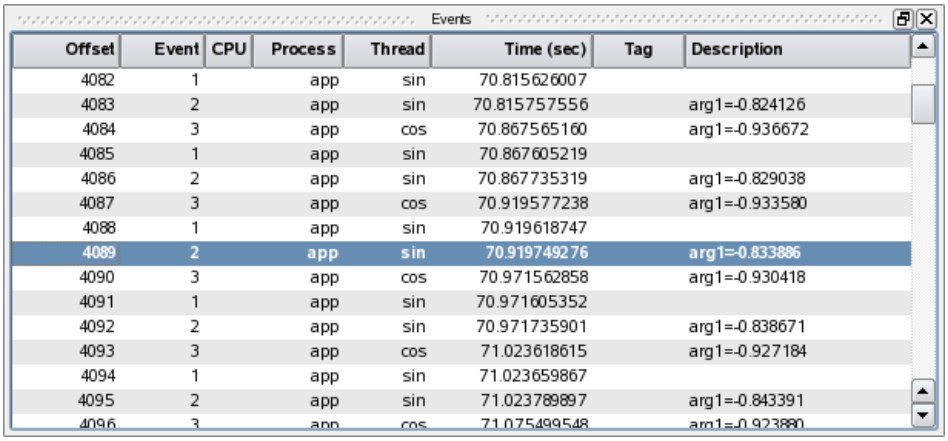
  Alternatively, pressing Ctrl+LeftArrow causes a shift one section to the left.

- Click midway between the gray area and the right hand portion of the control ruler

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

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

Using the Events Panel for Textual Analysis



Offset	Event	CPU	Process	Thread	Time (sec)	Tag	Description
4082	1		app	sin	70.815626007		
4083	2		app	sin	70.815757556		arg1=-0.824126
4084	3		app	cos	70.867565160		arg1=-0.936672
4085	1		app	sin	70.867605219		
4086	2		app	sin	70.867735319		arg1=-0.829038
4087	3		app	cos	70.919577238		arg1=-0.933580
4088	1		app	sin	70.919618747		
4089	2		app	sin	70.919749276		arg1=-0.833886
4090	3		app	cos	70.971562858		arg1=-0.930418
4091	1		app	sin	70.971605352		
4092	2		app	sin	70.971735901		arg1=-0.838671
4093	3		app	cos	71.023618615		arg1=-0.927184
4094	1		app	sin	71.023659867		
4095	2		app	sin	71.023789897		arg1=-0.843391
4096	3		app	cos	71.075499548		arg1=-0.923880

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 DownArrow key to advance to the next event.
- Press the UpArrow key to advance 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 to advance to the next set of events.
- Press the PageUp to shift to the previous set

  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 Event panel that shows event ID **`1`**.

- Right-click that row and select Edit Current Event Description… from the context menu.

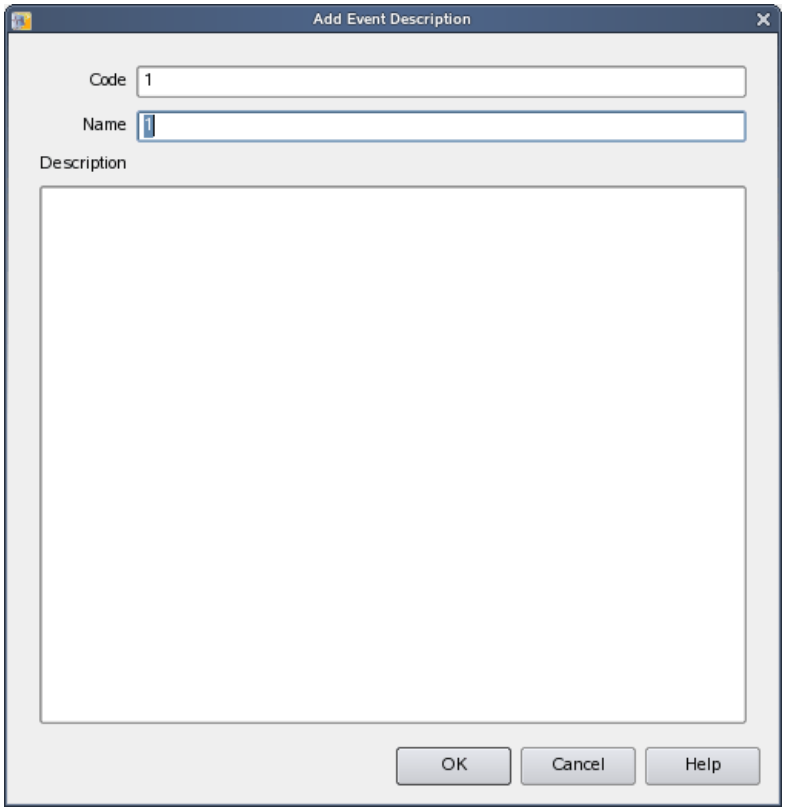


Figure 4-8. Add Event Description dialog

- Enter:

  `cycle_start`

  in the Name field.

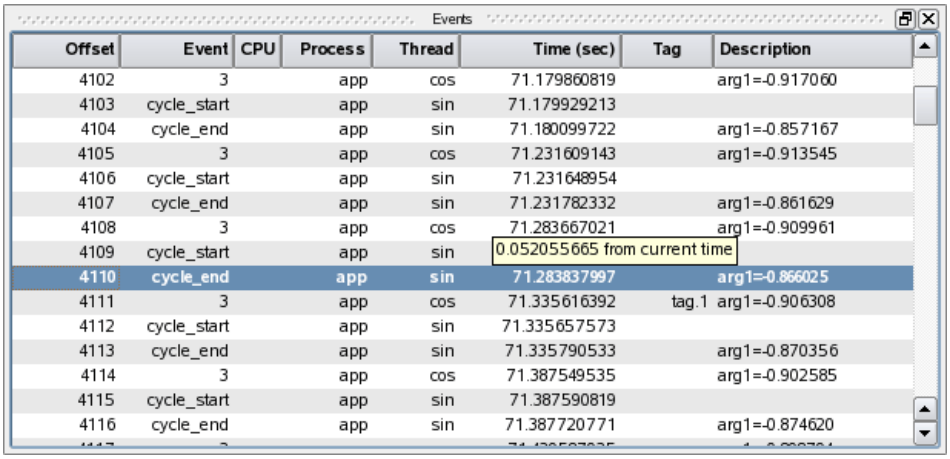
- Press OK.
- Right-click on an entry whose value in the Event column has the value **2**.
- Select Edit Current Event Description… from the context menu.
- Enter:

  `cycle_end`

  in the Name text field.

- Press the OK button.

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



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.

- Select the Change Search Profile... menu item from the Search menu in the NightTrace main window or press Ctrl+F.

A new page is created containing the Profile Status List panel and the Profile Definition panel:

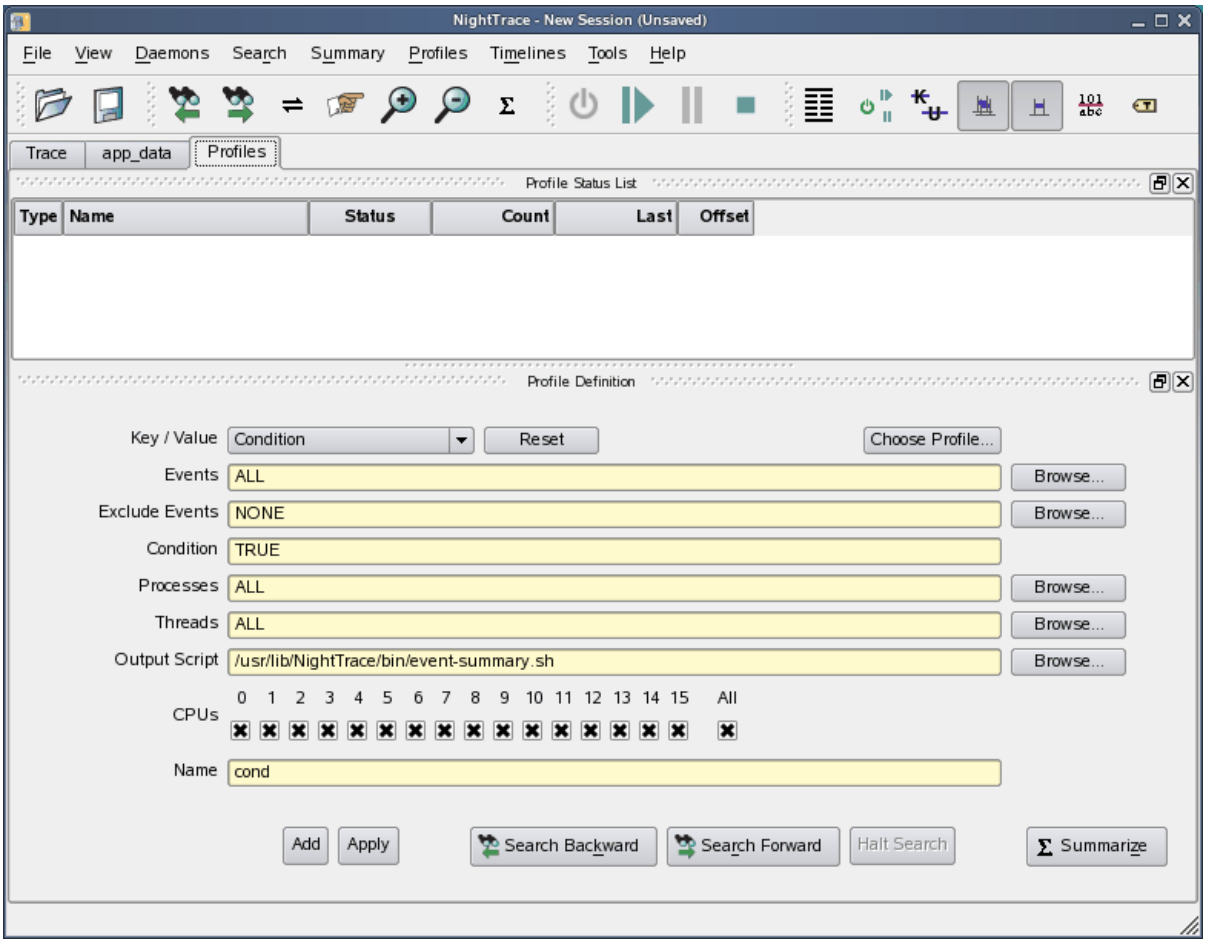


Figure 4-9. Searching using the Profiles panel

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

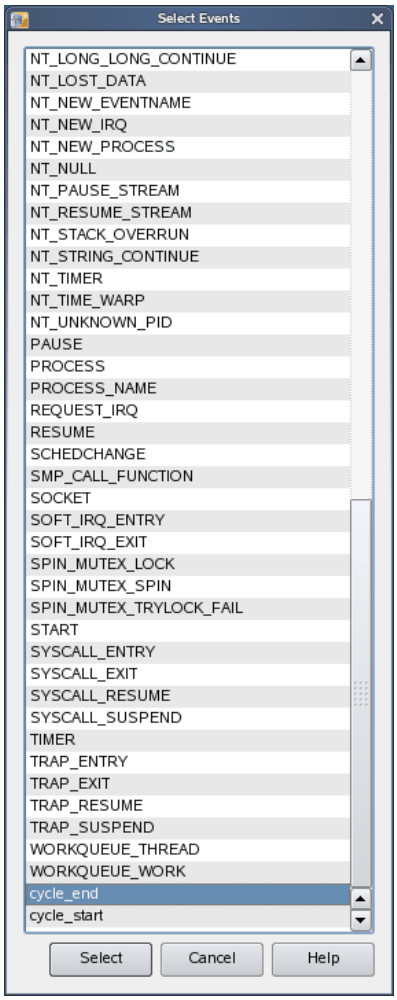


Figure 4-10. Browse Events Dialog

- Scroll the list until **`cycle_end`** is visible and select it; or click in the list and repeatedly press the c key until it becomes selected. Then press the Select button.

- Enter the following text in the Condition text field of the Profile panel:

  ```
  arg_dbl > 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 panel.

- Press the Search Forward button at the bottom of the Profiles panel.

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

- Click on the tab labeled `app_data` and verify that the current event listed in the Events panel indicates `arg1` with a value exceeding 0.8.

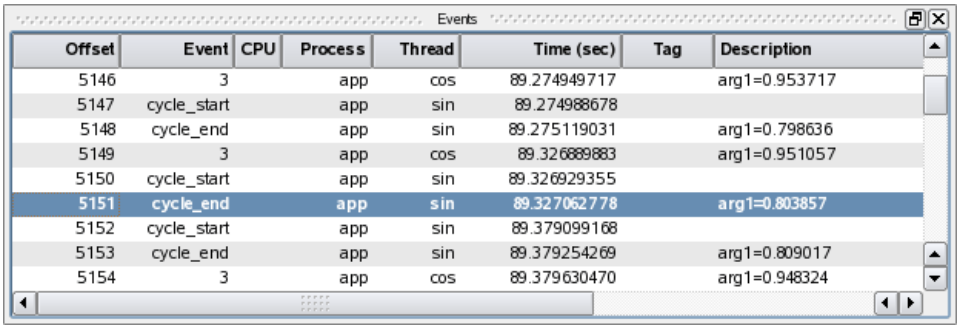


Figure 4-11. Events Panel After Search

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 in the lower right portion of the panel and leave it there without moving it

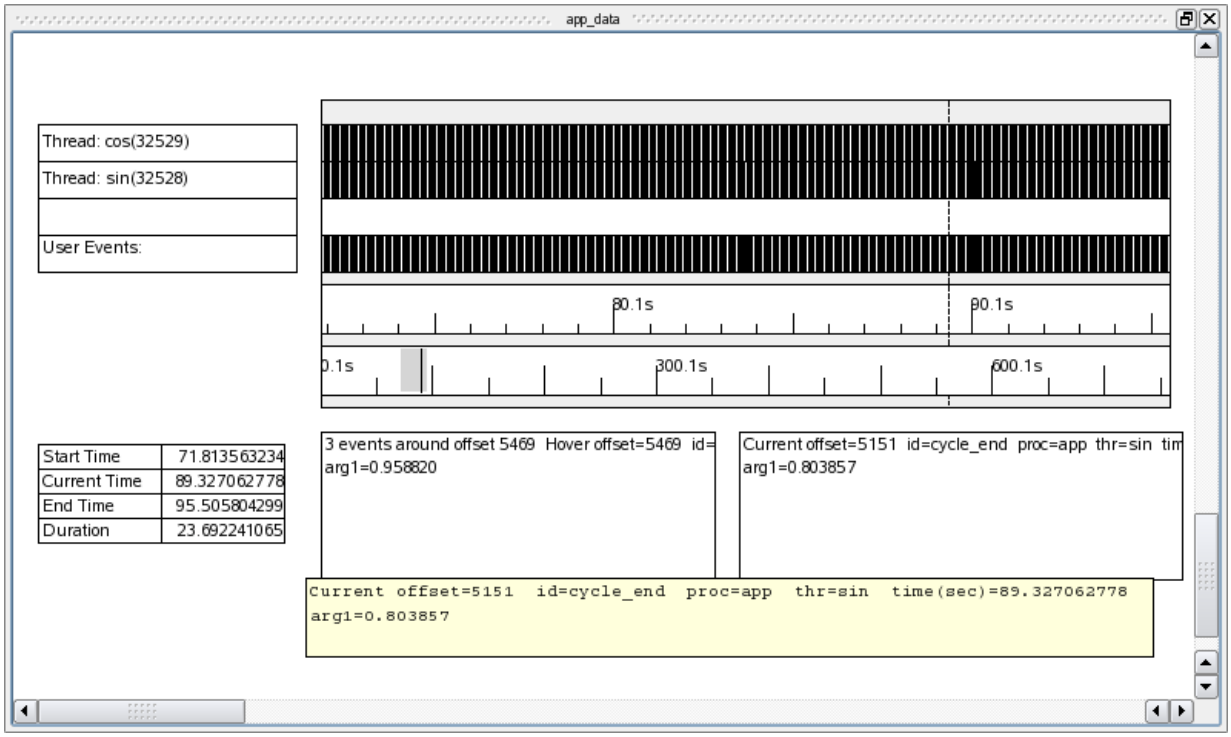


Figure 4-12. Timeline Panel w/ Tool Tip

A tool tip is also displayed which describes the current event.

NOTE

It is possible that the search will fail if an insufficient number of events have been brought into live analysis. If this occurs, bring in more events using the Event list scroll bar and retry the search by pressing the forward search icon on the tool bar.

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 usage.

Examine the daemon statistics in the Daemon Control Area on the first page. If the application has logged over 100,000 events, halt the daemon by pressing the Halt button to reduce memory usage as we slowly move through the NightTrace portion of the tutorial.

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.

If it has not reached this stage yet, you may leave the daemon running and occasionally glance at the statistics. If NightTrace becomes unresponsive or slows down as the event counts reach into the millions, halt the daemon. NightTrace has a configurable memory consumption limit that will automatically halt the daemon when the limit is reached; a dialog will be presented informing the user when this occurs.

Using States

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

- Click on the Profiles tab created in “Searching the Events List” on page 4-12.

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

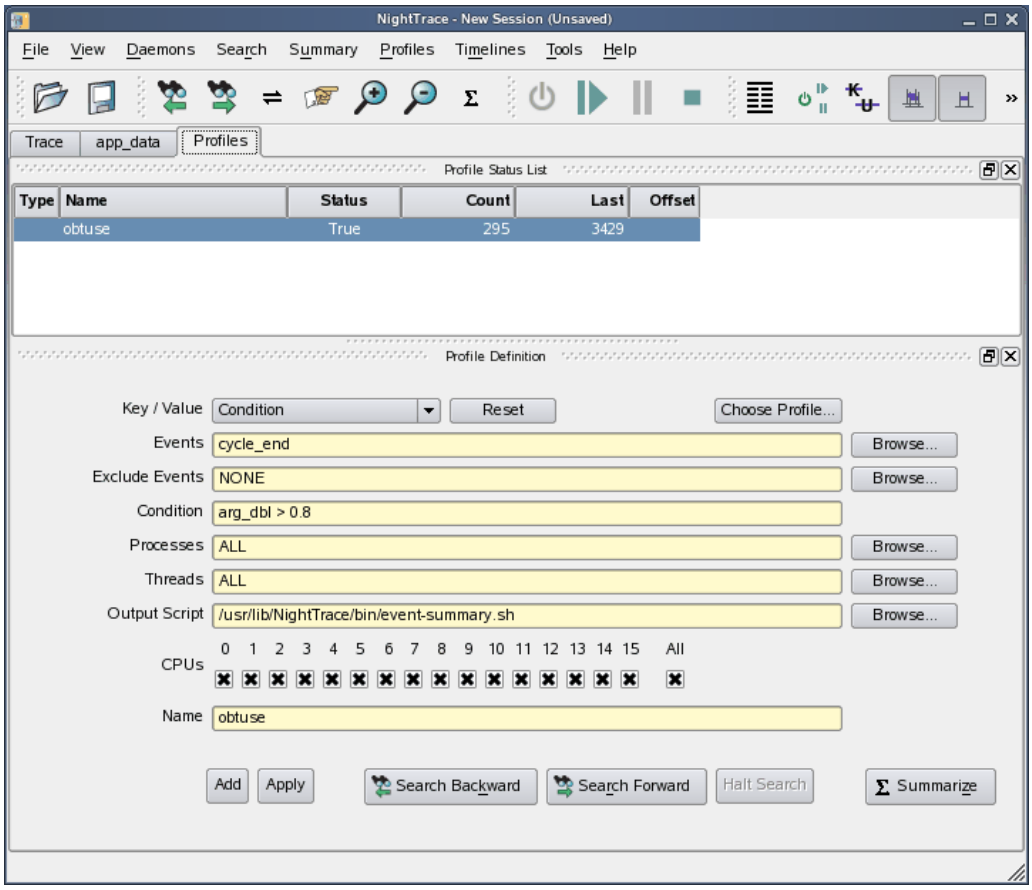


Figure 4-13. Profiles Panel With Obtuse Profile Selected

- Press the Reset button.
- Select State in the Key / Value option list.
- Enter:

  `cycle_start`

  in the Start Events text area
- Enter:

  `cycle_end`

  in the End Events text field.
- Enter:

  `sin`

  in the Threads text field.

- Enter:

  `sine`

  in the Name text field.

- Press the Add button.

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

- Click on the tab labeled `app_data` to show the timeline.
- Right-click anywhere in the display area and select Edit Mode from the context menu or press Ctrl-E to enter *edit mode*.

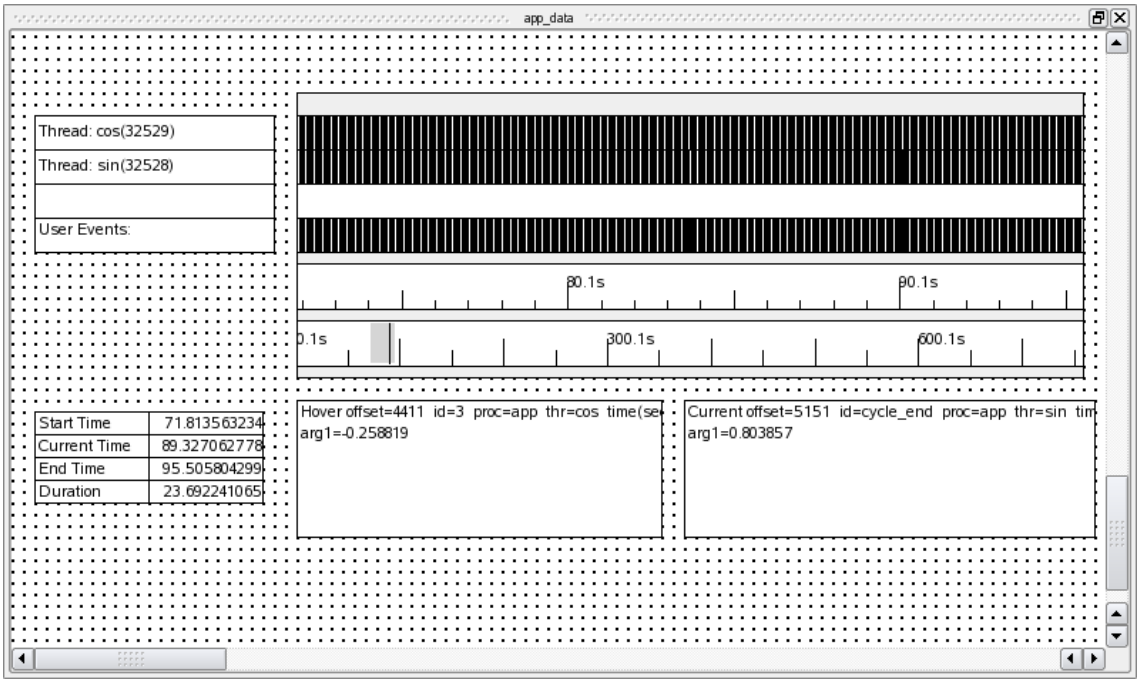


Figure 4-14. Timeline Editing

- 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:

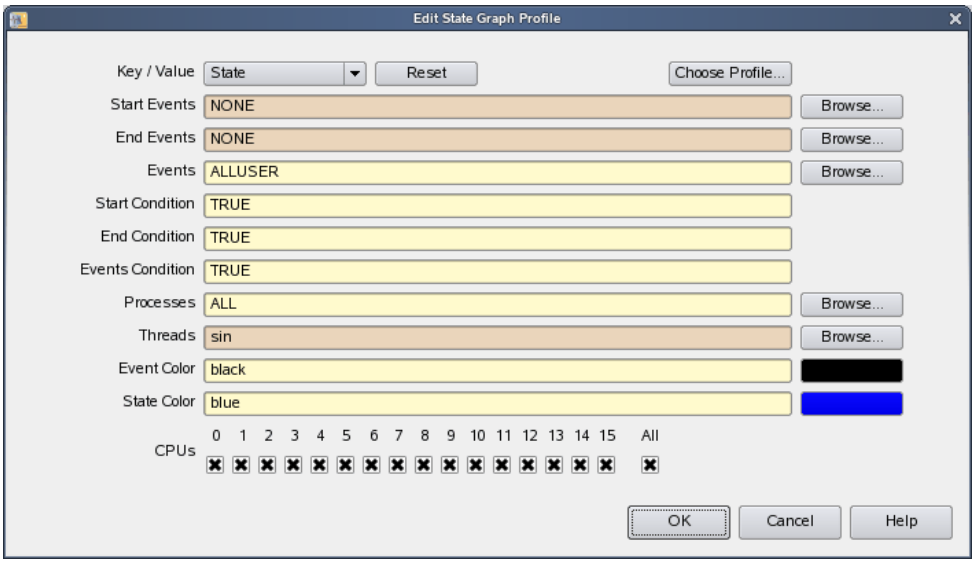
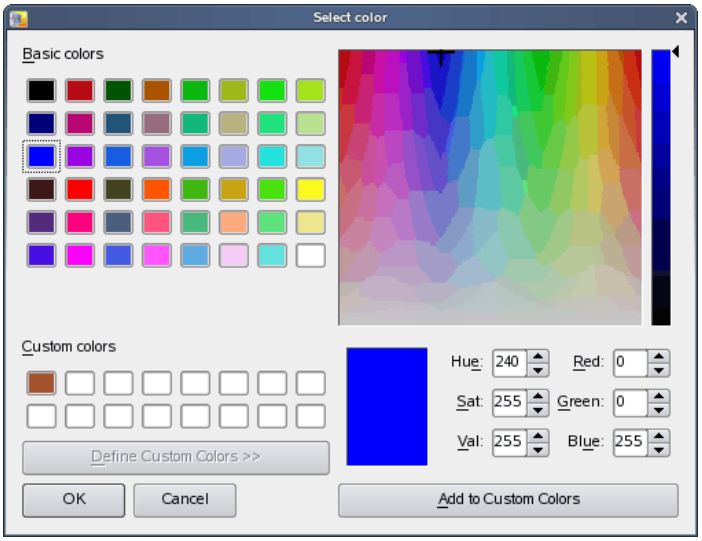


Figure 4-15. Edit State Graph Profile dialog

- Select State from the Key / Value option list.
- 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.
- Press OK in the Edit State Graph Profile dialog.
- Right-click on the graph associated with the row labeled "Thread: sin" and select Adjust Colors in Selected from the pop-up menu and select

State Color... from the sub-menu. The Select color dialog is presented.



- Select a pleasing color in the Select color dialog and press OK.
- Right-click anywhere in the display area and select Edit Mode from the pop-up 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 Arrow 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 Arrow 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.

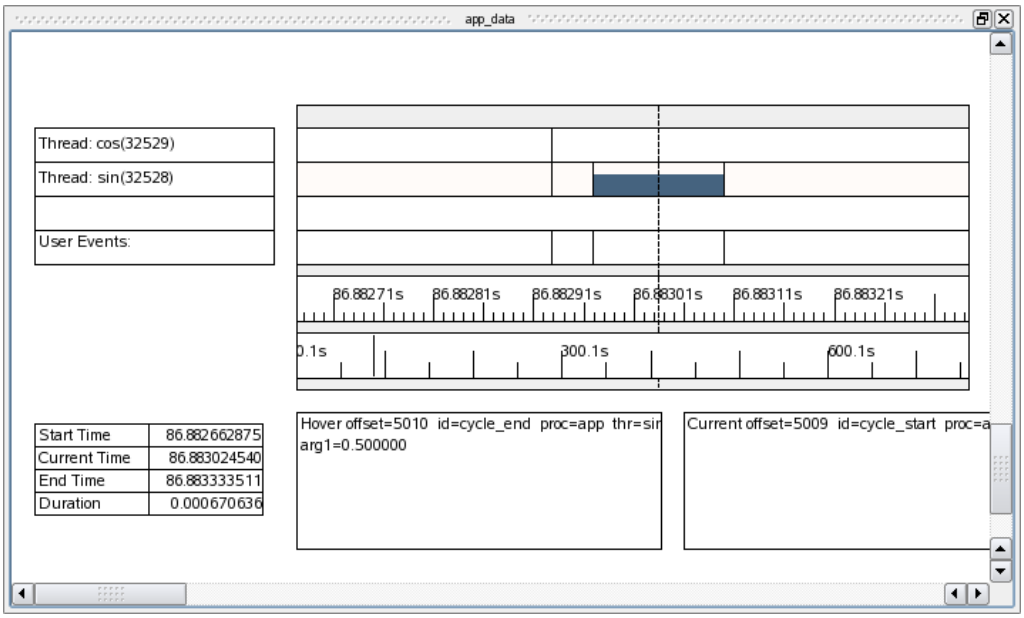


Figure 4-16. Sine State in Timeline

NOTE

If no states are visible, recheck the definition of the sine profile in the Profiles panel as described in “Using States” on page 4-16.

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 pop-up menu or press Ctrl-E to enter *edit mode*.
- Right-click anywhere in the grid and select Add Data Box option from the pop-up menu.

  The cursor will turn into a + character.
- Using the left mouse button, click an area in 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 pop-up menu or press Ctrl-E to return to *view mode*.

The data box now displays the length of the most recently completed instance 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.

- Select the Change Summary Profile... menu item from the Summary menu.
- Select the profile matching the `sine` state from the list of profiles shown in the Profile Status List panel.

  It is likely that the `sine` profile is already selected. Check the profile name shown in the Name text area near the bottom of the dialog.

- Press the Summarize button.

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

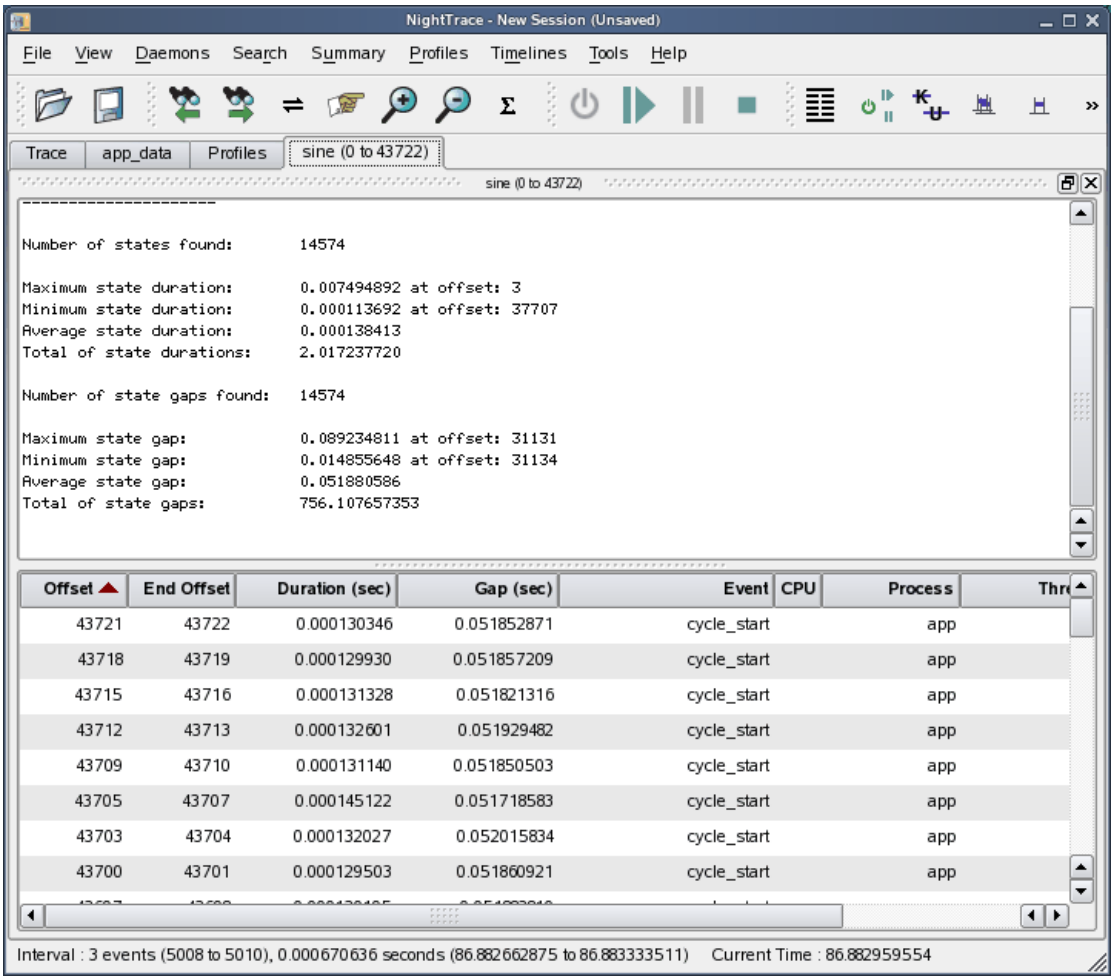


Figure 4-17. Summary Results Page

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 again on the Duration header if the current sort order is smallest to largest
- The instance of the state with the longest duration is shown in the top row
- Double 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.

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

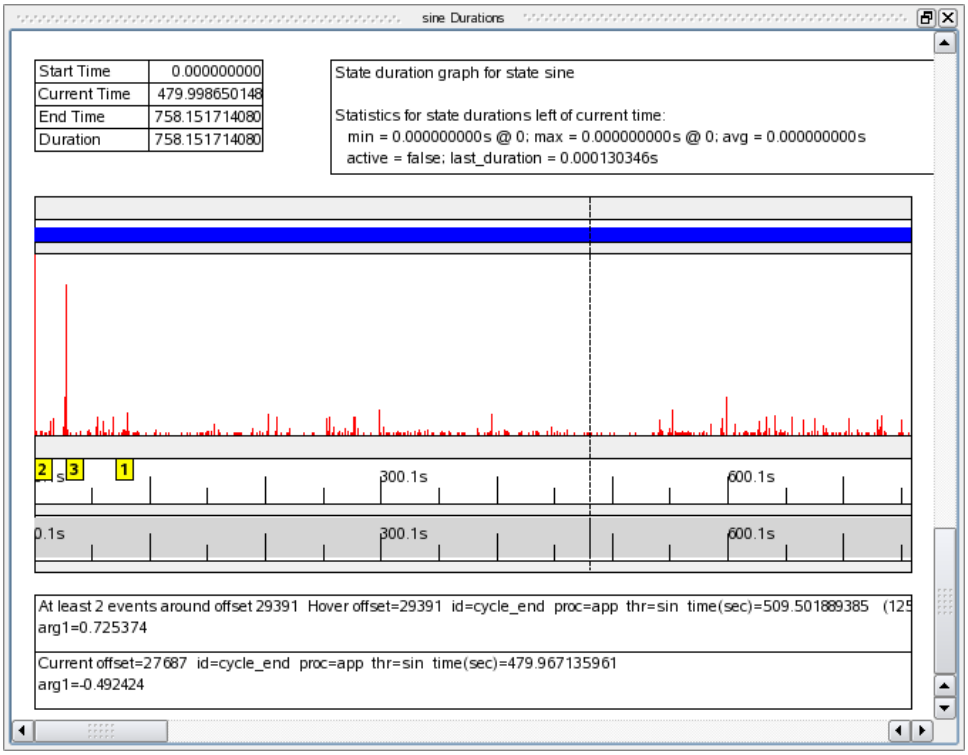


Figure 4-18. Summary Graph

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+UpArrow.

  A data graph is shown in the wide column 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.

- Click anywhere in the data graph and enter Edit mode by pressing Ctrl+E.
- Double-click anywhere in the data graph.

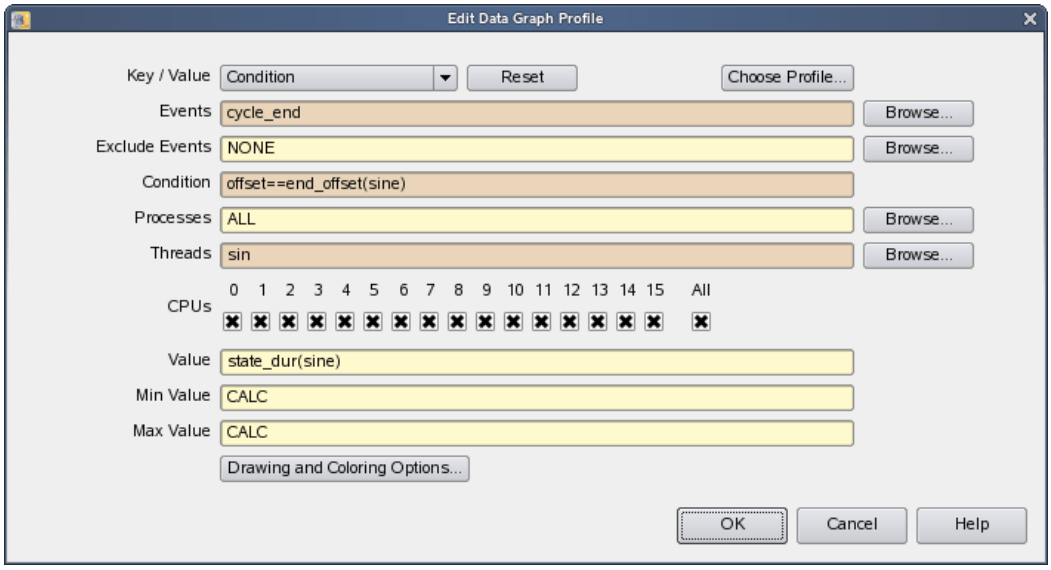


Figure 4-19. Data Graph Profile Dialog

- Change the Max Value text field to 0.001 or a value representative of most of typically long state durations (refer to the sorted list of state durations in “Generating Summary Information” on page 4-22).
- Press the OK button.

- Return from Edit mode by pressing Ctrl+E.

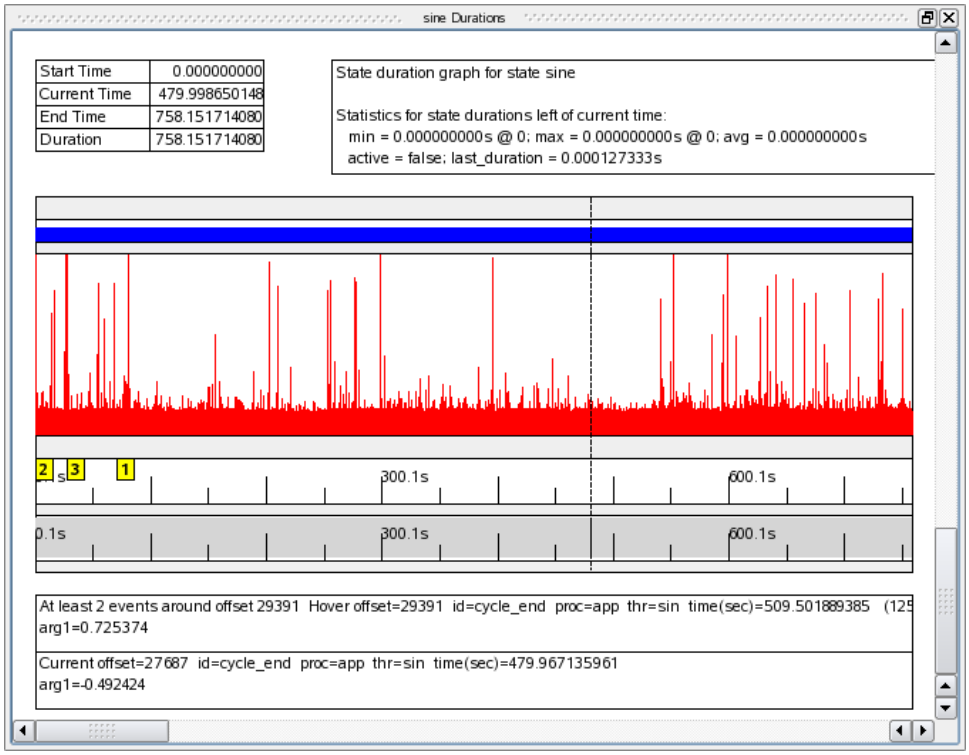


Figure 4-20. Modified Data Graph

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.

Defining a Data Graph

The area containing the timelines has a blank area above the graphs for each of the threads in the program. We will now add a data graph in this area.

- 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.
- Right-click anywhere in the display panel labeled `app_data` and select Edit Mode from the pop-up menu or press Ctrl-E to enter *edit mode*.
- Click on the middle of the upper horizontal line of the column containing the graphs in the panel.
- Move the mouse cursor so that it hovers over the middle of the upper horizontal line of the column..

- When the cursor changes to two arrows pointing up and down, click and drag the upper boundary of the column upward to make space for the data graph.

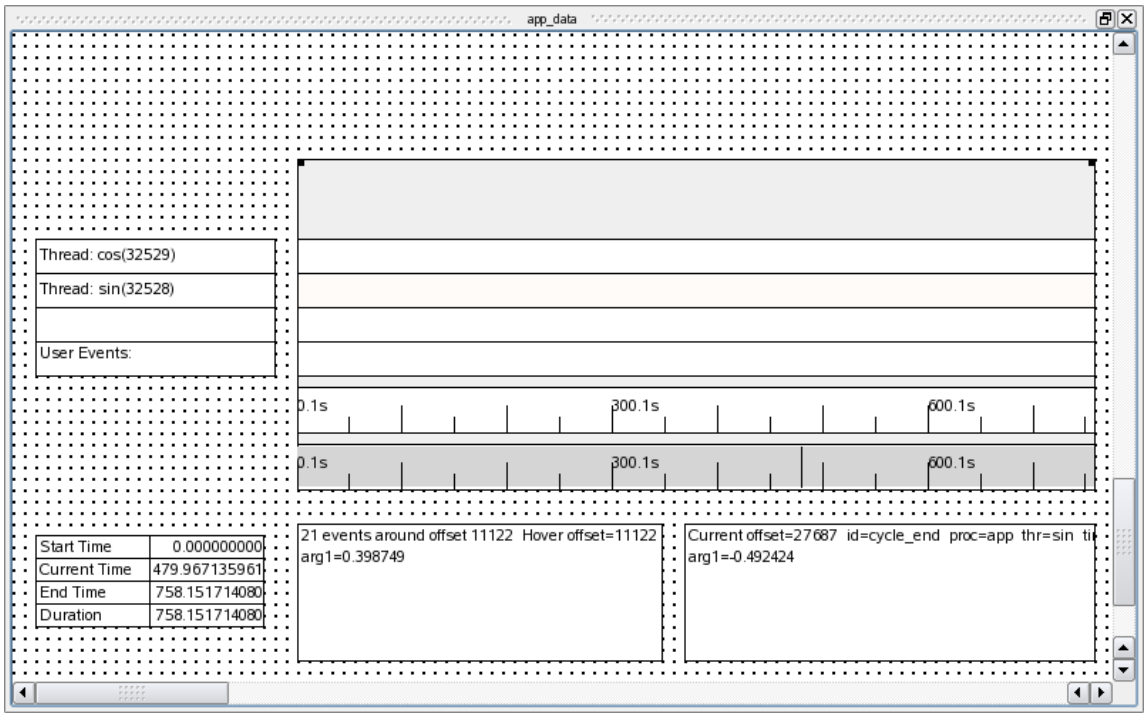


Figure 4-21. Resizing in Progress

- Release the mouse button when sufficient space has been made (approximately an inch or more vertically).
- Click on the upper horizontal line of the column.
- Right-click inside the graph container and select Add to Selected Graph Container from the pop-up 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.

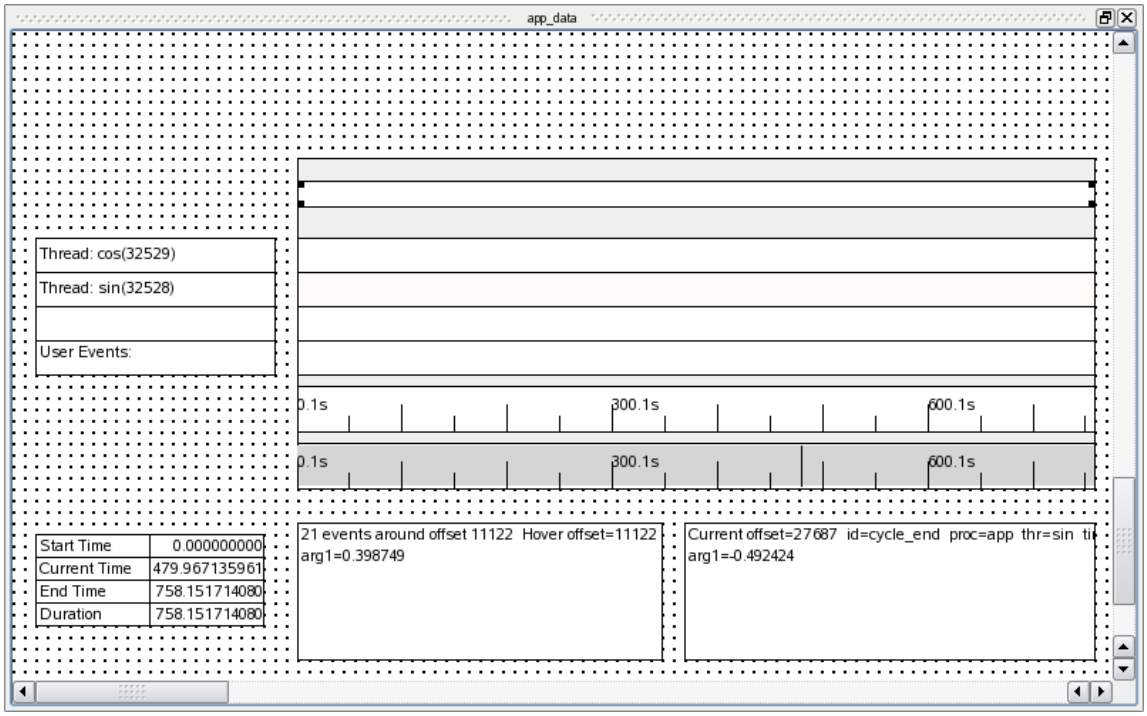


Figure 4-22. Adding a Data Graph

- Click inside data graph you just inserted.
- Drag the top border to the top of the graph container and the bottom border to the bottom of the graph container so that the data graph fills the graph container you created.
- Click and drag the upper and lower lines of the newly inserted data graph to fill the available space.
- Double-click in the middle of the data graph.

The Edit Data Graph Profile dialog is presented.

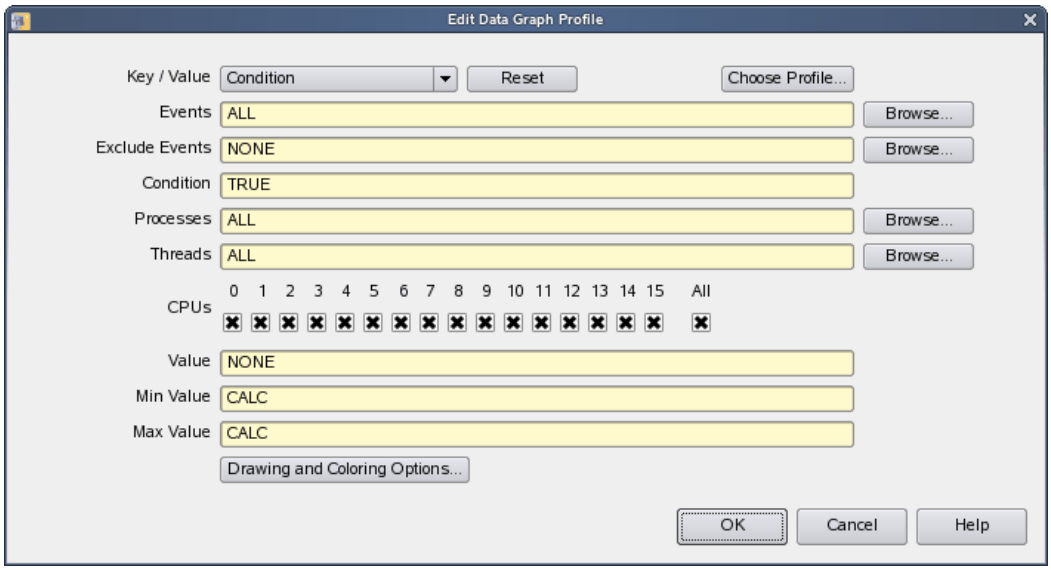


Figure 4-23. Edit Data Graph Profile Dialog

- Enter:

  `cycle_end`

  in the Events text field.

- Enter:

  `arg1_dbl`

  in the Value text field.

- Press OK to close the Edit Data Graph Profile dialog.

- Right-click inside the data graph and select Adjust Colors in Selected from the pop-up menu and select Data Graph Value Color... from the sub-menu.

- Select a pleasing color from the Select color dialog for the data graph. Click OK to close the Select color dialog.

- Right-click anywhere in the display panel labeled `app_data` and select Edit Mode from the pop-up menu or press Ctrl-E to return to *view mode*.

- Zoom the display to see the sine wave generated by the program.

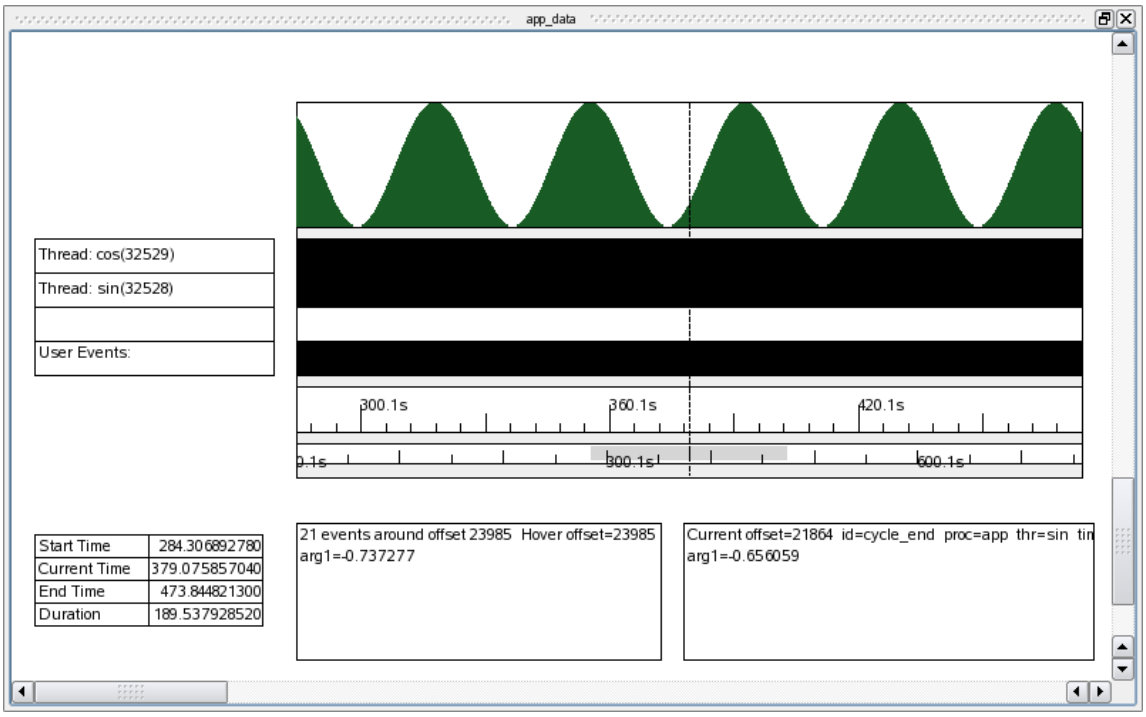


Figure 4-24. Display Page with Data Graph

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.

Kernels 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 tutorial and load the pre-recorded kernel data as instructed.

- Click on the first tab of the NightTrace main window.
- Ensure the user daemon is stopped by pressing the Halt button in the Daemon Control Area if it is sensitized.
- Select the `app_data` segment in the Trace Segments panel.

CAUTION

If the trace segment was not removed it is likely that you selected the `app_data` line from the Daemon Definition Area and not the Trace Segments panel.

- Select the Close Trace Data option from the Trace Segments panel.

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 4-33.

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

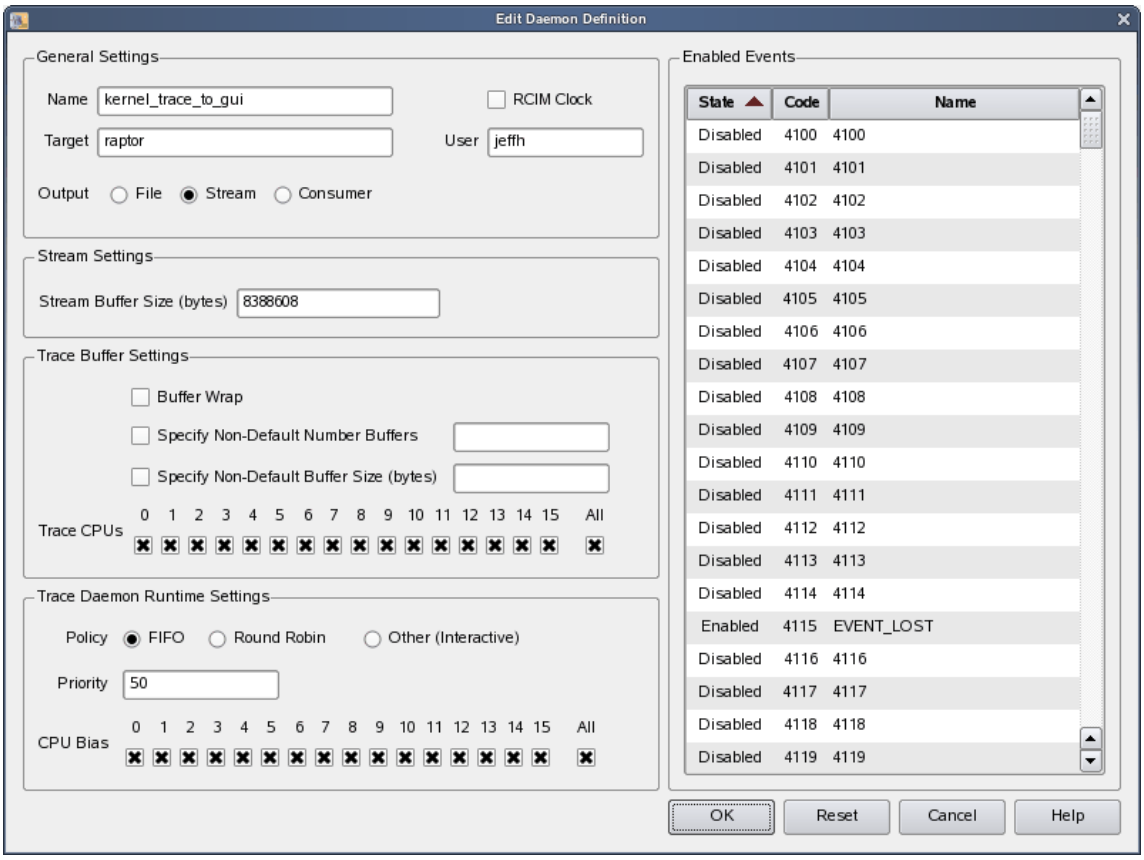


Figure 4-25. Edit Daemon Definition Dialog

- Check the Buffer Wrap checkbox in the Trace Buffer Settings section of the Edit Daemon Definition dialog.
- Press OK.

The kernel daemon is now configured to run in bufferwrap mode. This means that kernel events are collected in kernel memory buffers and are not passed to NightTrace except by explicit flush operations.

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 40,000-50,000 events are present in the Buffer column, press the Flush button and then the Halt button.

Skip the next section and jump directly to “Analyzing Kernel Data” on page 4-33.

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 4-33.

The NightStar RT **`tutorial`** directory contains some pre-recorded kernel data which can be used in the section titled “Analyzing Kernel Data” on page 4-33.

- Select the Open Trace File… menu item from the NightTrace menu in the NightTrace main window.
- Type the following into the file dialog in the Selection text field:

  `/usr/lib/NightStar/tutorial/.kernel-data`

- Press the OK 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.
- Zoom out until the data and state graphs are populated with events.

- Click in an active area and zoom in until detail can be seen.

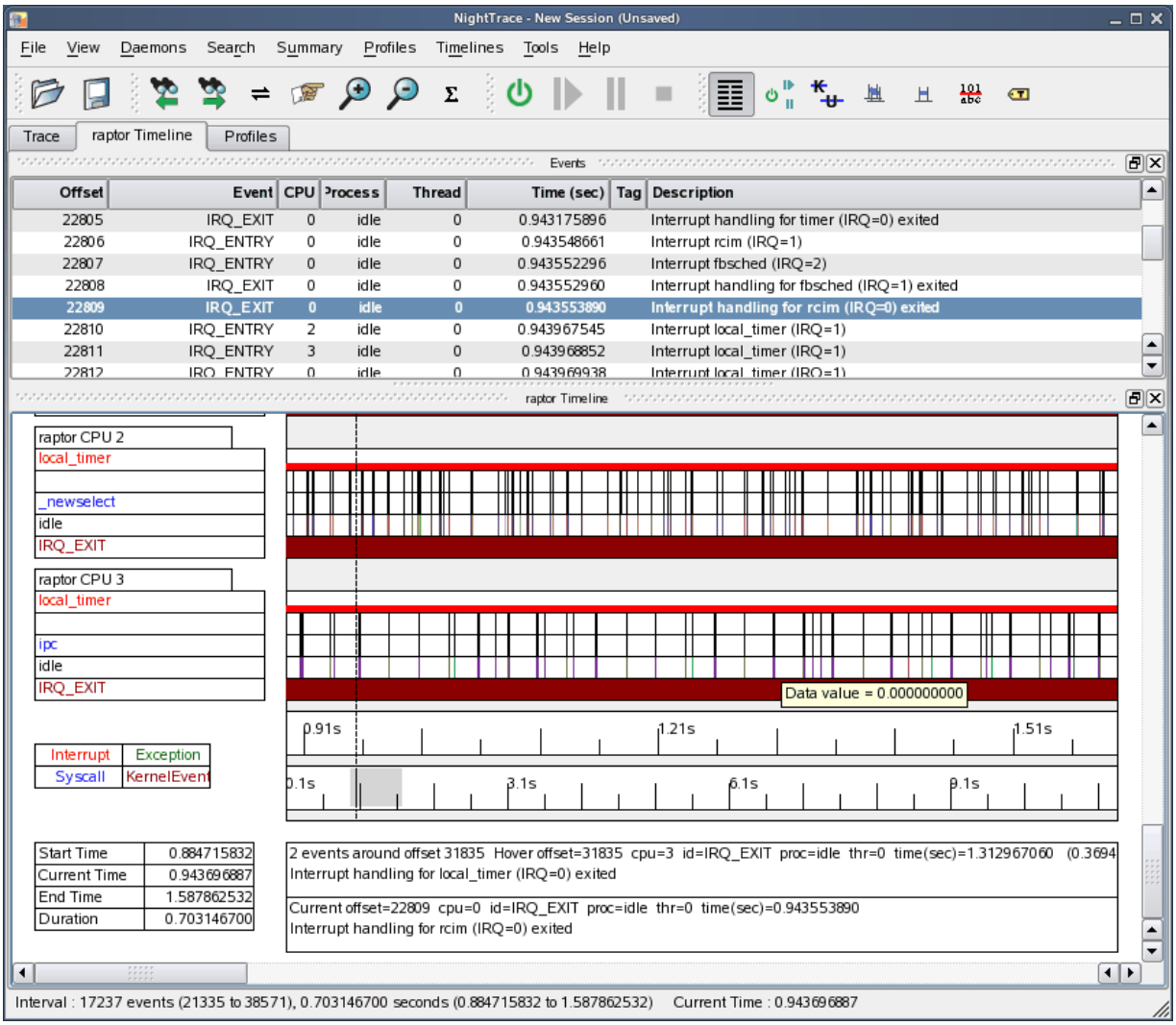


Figure 4-26. Kernel Display Page

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.

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 switch to the Profiles panel.
- 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.
- Select the System Call Enter Events option from the Key / Value option list.

  The Select System Calls dialog is presented.

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

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

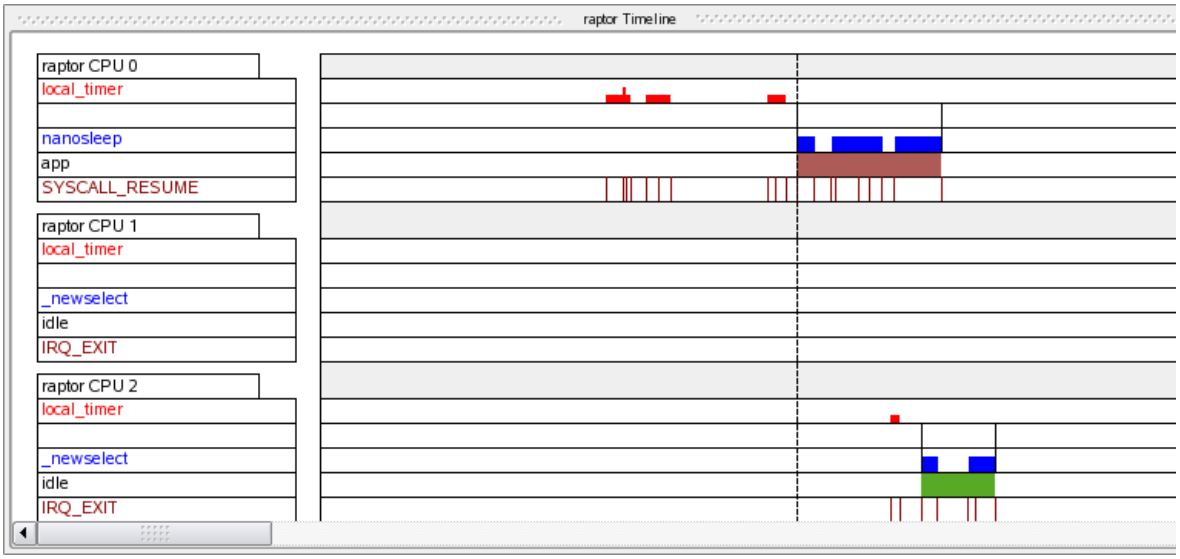


Figure 4-27. System Call Resume for Nanosleep

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 which 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 with:

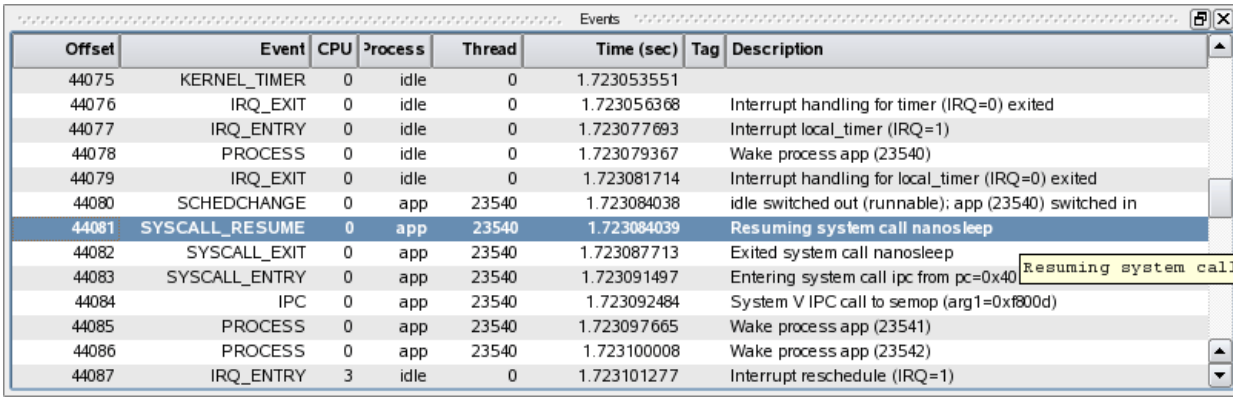


Figure 4-28. Events Panel after Search

- While the current timeline is at the SYSCALL_RESUME event, press the Up Arrow key.

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 (5336) 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 time line.

- Press the Ctrl-G key to advance back to the SYSCALL_RESUME event.

In the instance shown in the screen shot above, shortly after the sine thread returns from nanosleep, the main thread is exiting the nanosleep call on line 93 of **app.c**. It then enters an ipc system call to execute the semop library call on line 94.

NOTE

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

Mixing Kernel and User Data

If you are not running a trace-enabled kernel, skip this section and proceed to "Using the NightTrace Analysis API" on page 4-41.

- 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).

- Select the `kernel_trace_to_gui` segment in the Trace Segments panel and select the Close Trace Data menu option of the context menu.
- Select both daemons in the Daemon Control Area using Click and Shift+Click mouse and keyboard actions.
- Press the Launch button.

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

- Press the Resume button.
- Wait about 2 seconds.
- Press the Flush button.
- Press the Halt button.

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

- Click on the Profiles tab.
- Select the `sine` profile from the Profile Status List at the top of the page.
- Press the Summarize icon on the toolbar.

The last action caused a new page to be created containing a summary of the `sine` state defined in “Generating Summary Information” on page 4-22.

The current timeline is automatically positioned to the longest instance of the state.

- Click on the tab corresponding to the kernel display page.
- 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 a `rcim` interrupt.

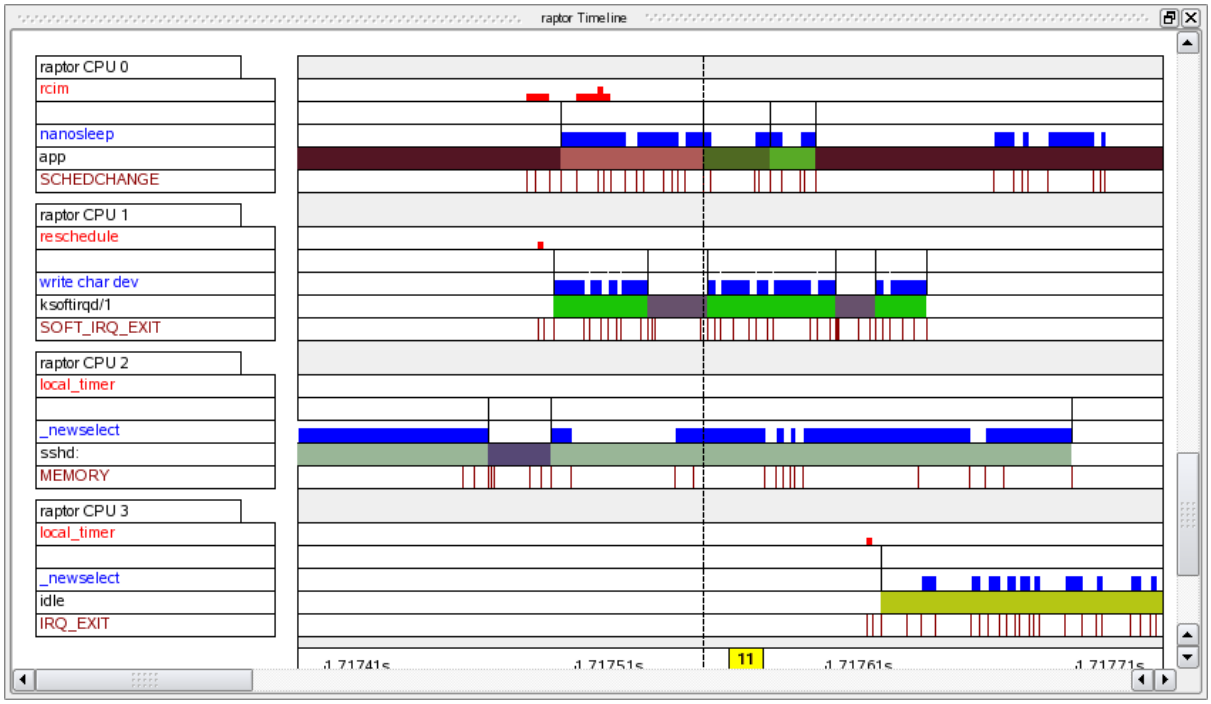


Figure 4-29. Longest Instance of State

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.

Some of these circumstances are discussed in more detail in “Overrun Detection and System Tuning” on page 7-9.

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.
- Select Exception All Events from the Key / Value option list.
- 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.

You can use NightView to see the actual line number of programs (if they have debugging information) based on the PC information with a command like: **`list *`***pc-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.

Users 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 Profiles tab.
- Select the `sine` profile from the Profile Status List.
- Select the Export to API Source... menu item from the Profiles menu.

The following dialog is displayed:

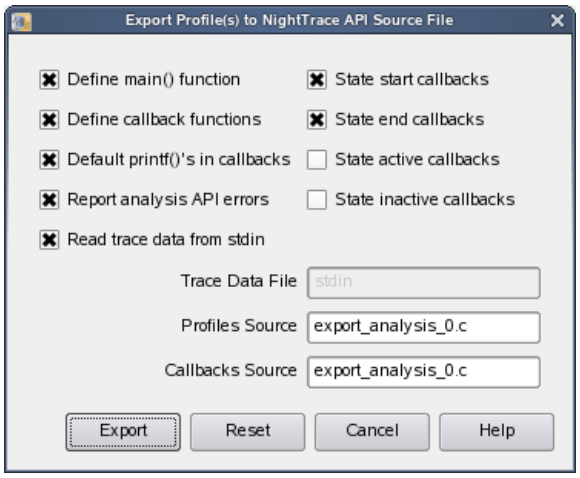


Figure 4-30. Export Profiles to NightTrace API Source File dialog

- Clear the State start callbacks checkbox.
- Press the Export button.
- Select the Exit Immediately menu item from the NightTrace menu 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 daemon, skip the remaining of this section and proceed to "Conclusion - NightTrace" on page 4-43.

Several sample API programs are provided with NightTrace.

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

```
cp /usr/lib/NightTrace/examples/watchdog.c .
cc -g -o watchdog 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 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
```

Conclusion - NightTrace

This concludes the NightTrace portion of the NightStar RT Tutorial.

5
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 and it is running under the control of NightView. If you have not built the program, do so using the instructions in "Building the Program" on page 1-4 and execute the application 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.

Invoke NightProbe by selecting NightProbe Monitor from the Tools menu of any of the NightStar Tools currently running. You may also invoke NightProbe by using the NightProbe desktop icon or type the following command:

```
nprobe &
```

at a command prompt.

The NightProbe main window is displayed.

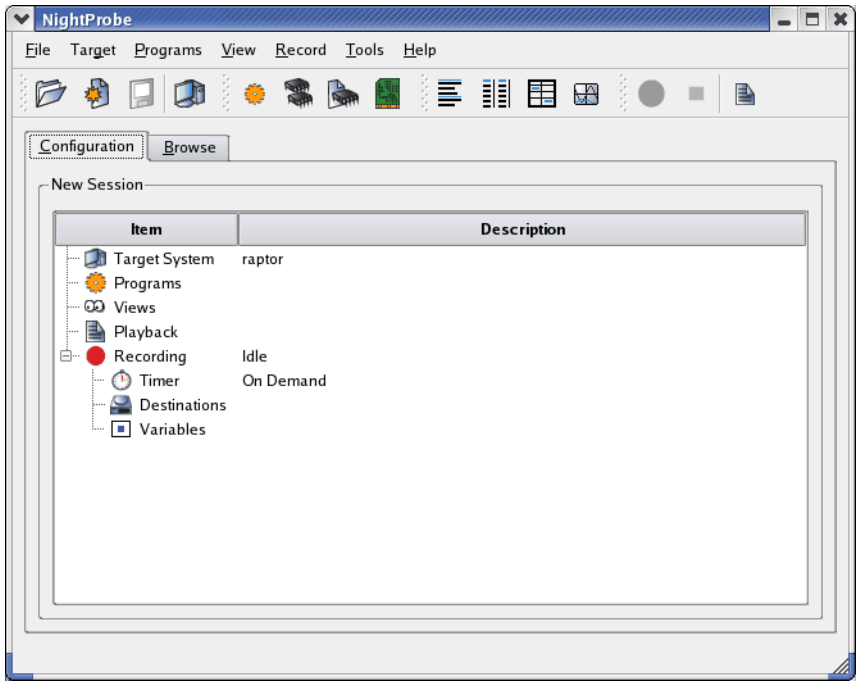


Figure 5-1. NightProbe Main Window

Selecting Processes

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

- Right-click the Programs icon on the Configuration page and select the Program… menu option.

The Program Selection dialog is presented:

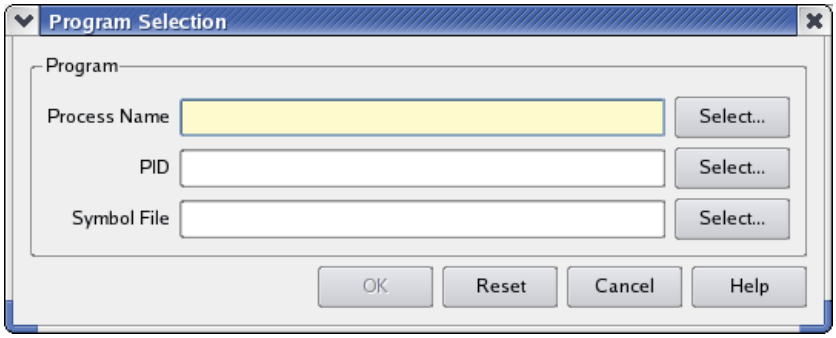


Figure 5-2. Program Selection Dialog

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

The Process Selection dialog will appear.

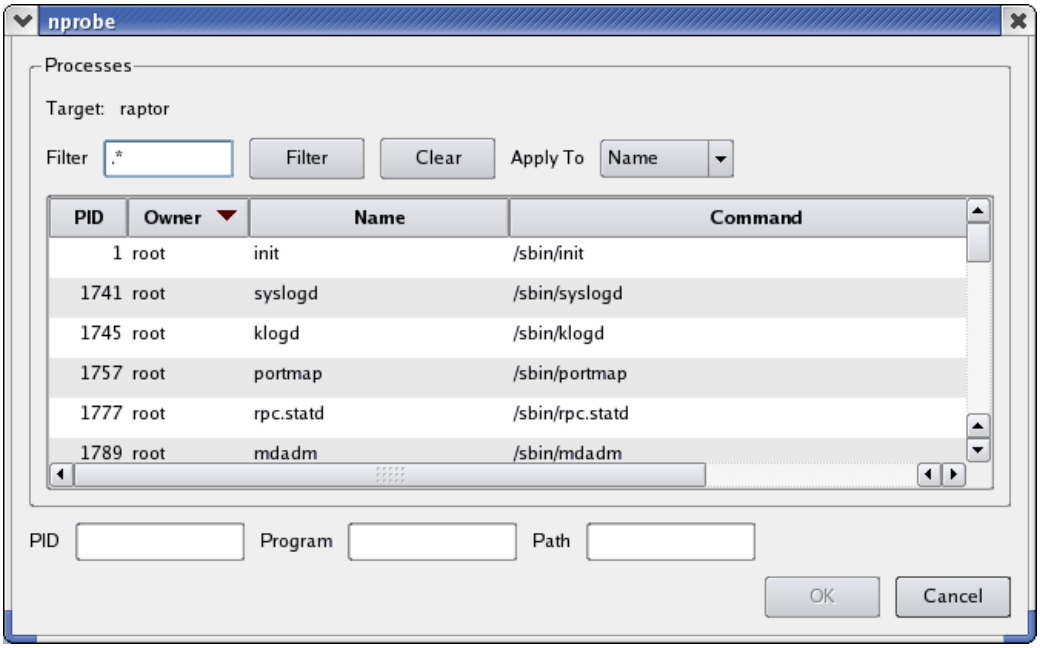


Figure 5-3. Process Selection Dialog

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

The list is filtered to only those process whose name includes **app** and an entry should be selected in the table.

- Ensure that a single item appears in the table and press Enter again to close the dialog. If multiple items appear in the table, double-click on the **app** process associated with your user name.

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 Enter 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.

Viewing Live Data

- Click on the Browse tab in the NightProbe main window.

The Live Browser is displayed.

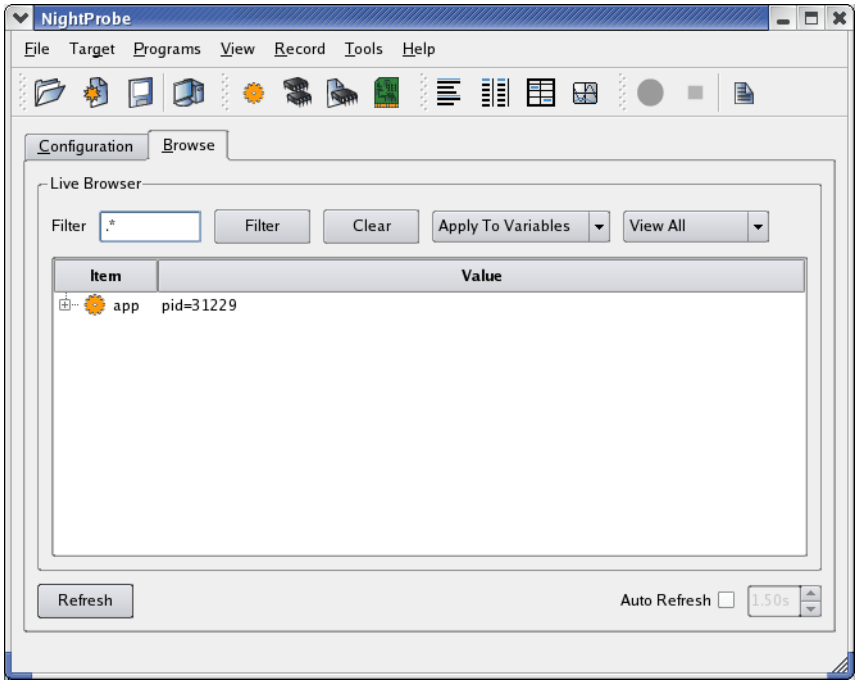


Figure 5-4. NightProbe Browse Panel

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.

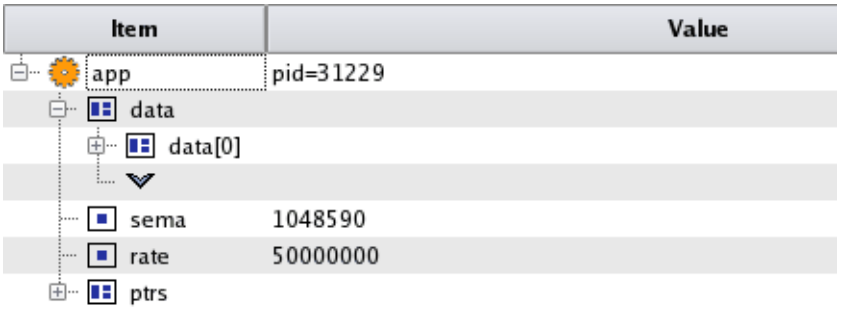


Figure 5-5. Expanded Data Item

The downward pointing arrow head 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, whenever an automatic refresh occurs as controlled by the Automatic refresh checkbox, or when the page receives or loses focus.

- Click the Automatic Refresh checkbox.

This causes the display to automatically refresh at the rate shown in the spinbox to the right of the Automatic 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 this should occur or not.

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

Double-click the value of the state variable.

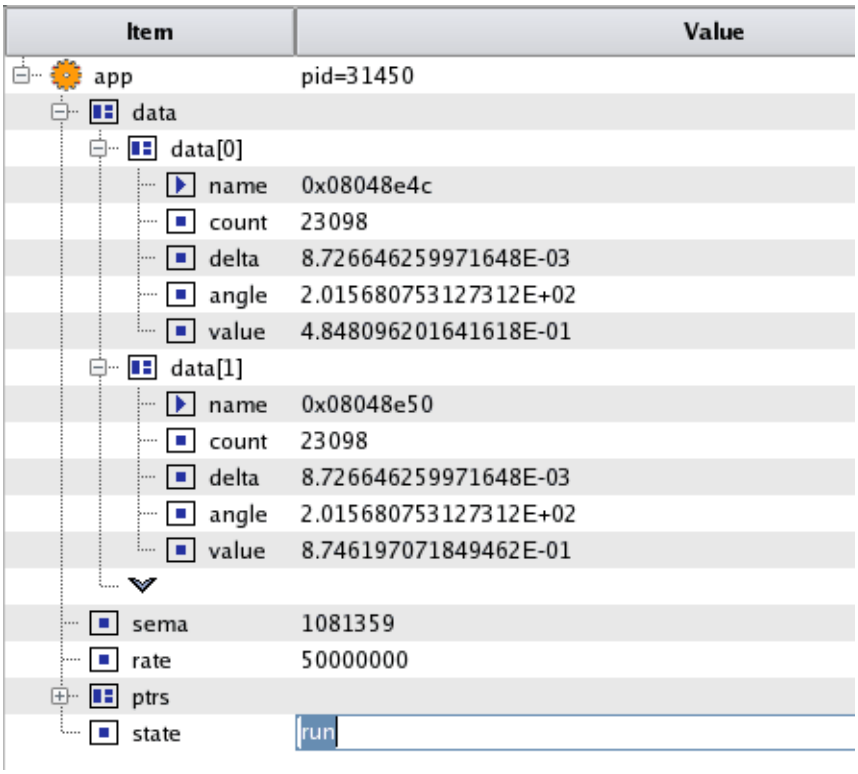


Figure 5-6. Variable Modification in Progress

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`**:

```
94      for (;;) {
95         struct timespec delay = { 0, rate };
96         nanosleep(&delay,NULL);
97         if (state != hold) semop(sema.&trigger,1);
98      }
```

- Change the value of the state variable back to `run`

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 attributes specifies that the variable is to be included in recording sessions.

Double-clicking an item causes the color to turn a reddish color 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 from both `data[0]` and `data[1]` structures.
- Double-click the `rate` variable.

The Browse page tree should look similar to the following:

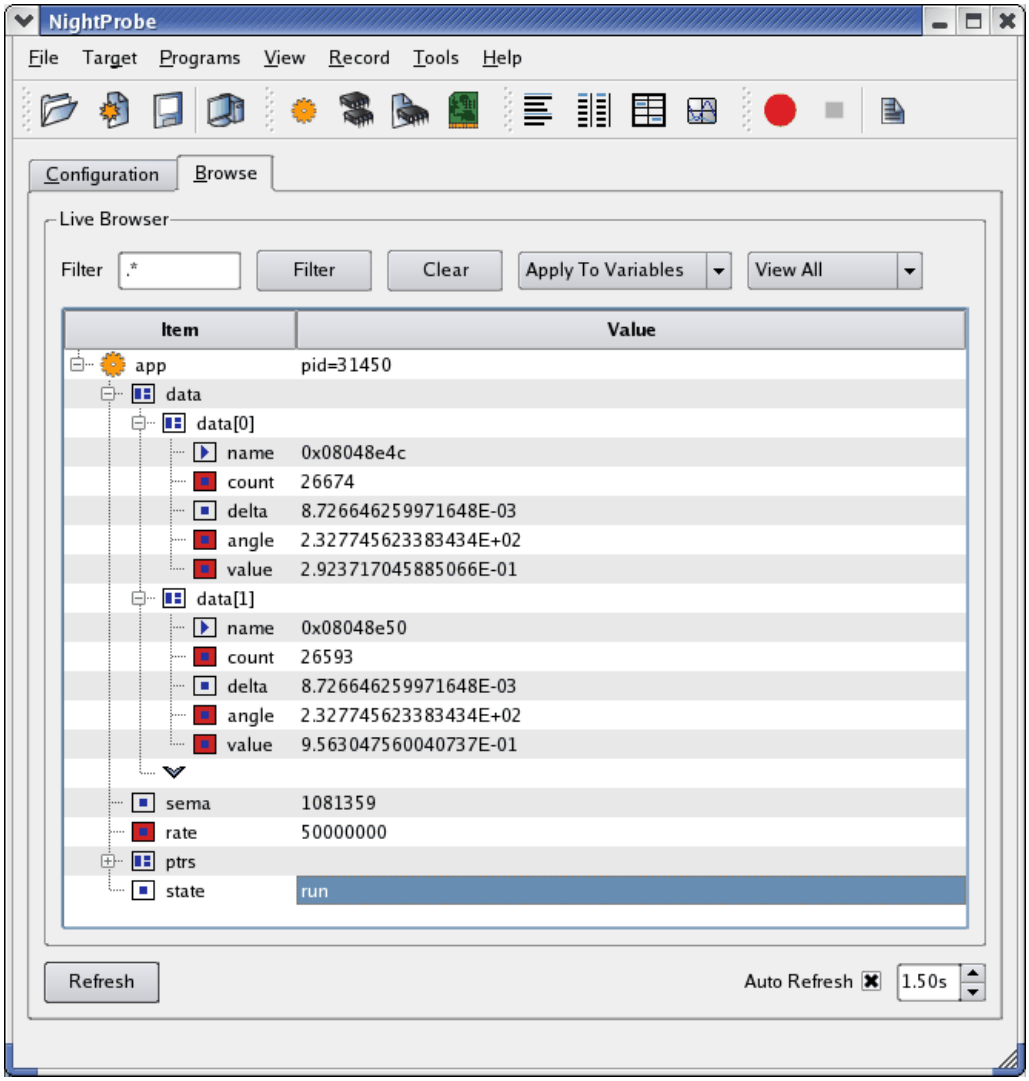


Figure 5-7. Mark and Record Attributes Set

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

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

- Select the Table option from the View menu.

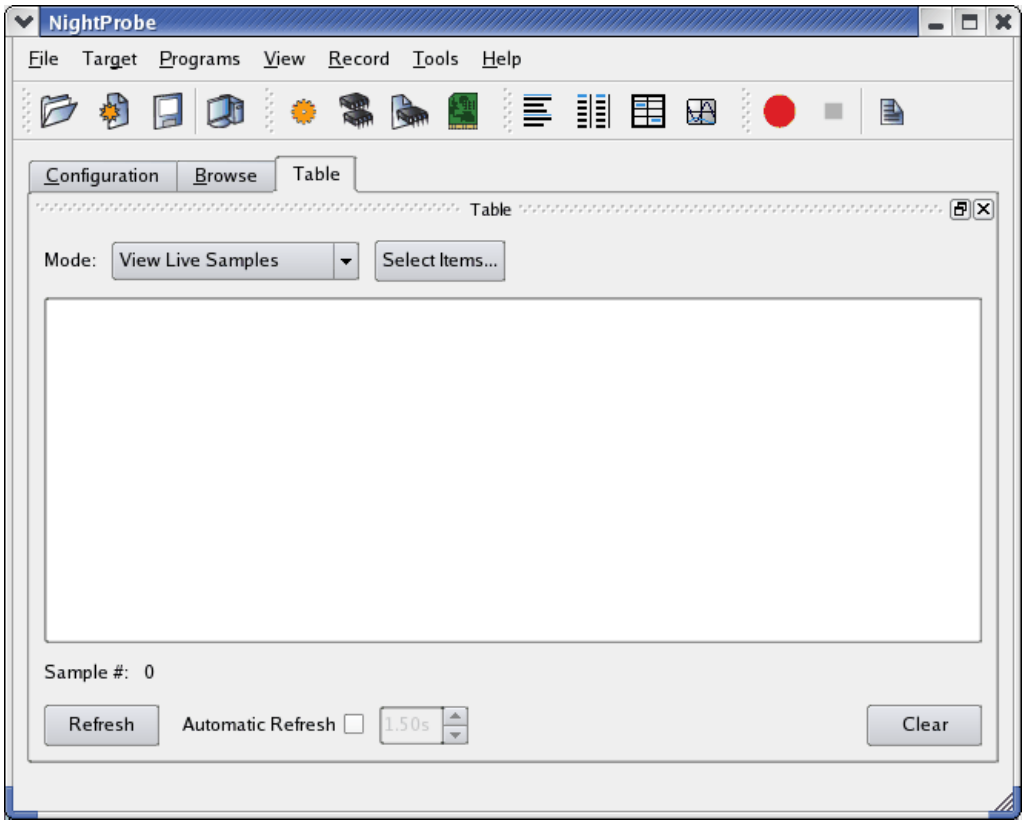


Figure 5-8. Table View

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.

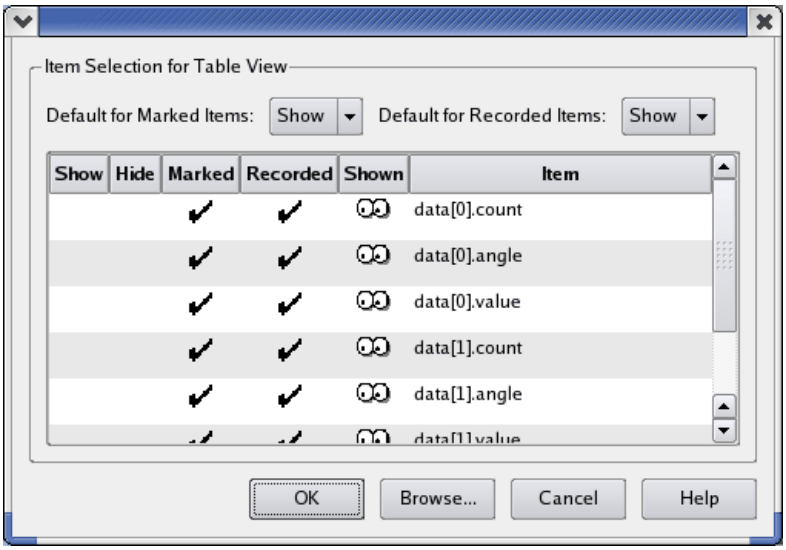


Figure 5-9. Item Selection Dialog

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 Hide column.
- 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 Refresh checkbox

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

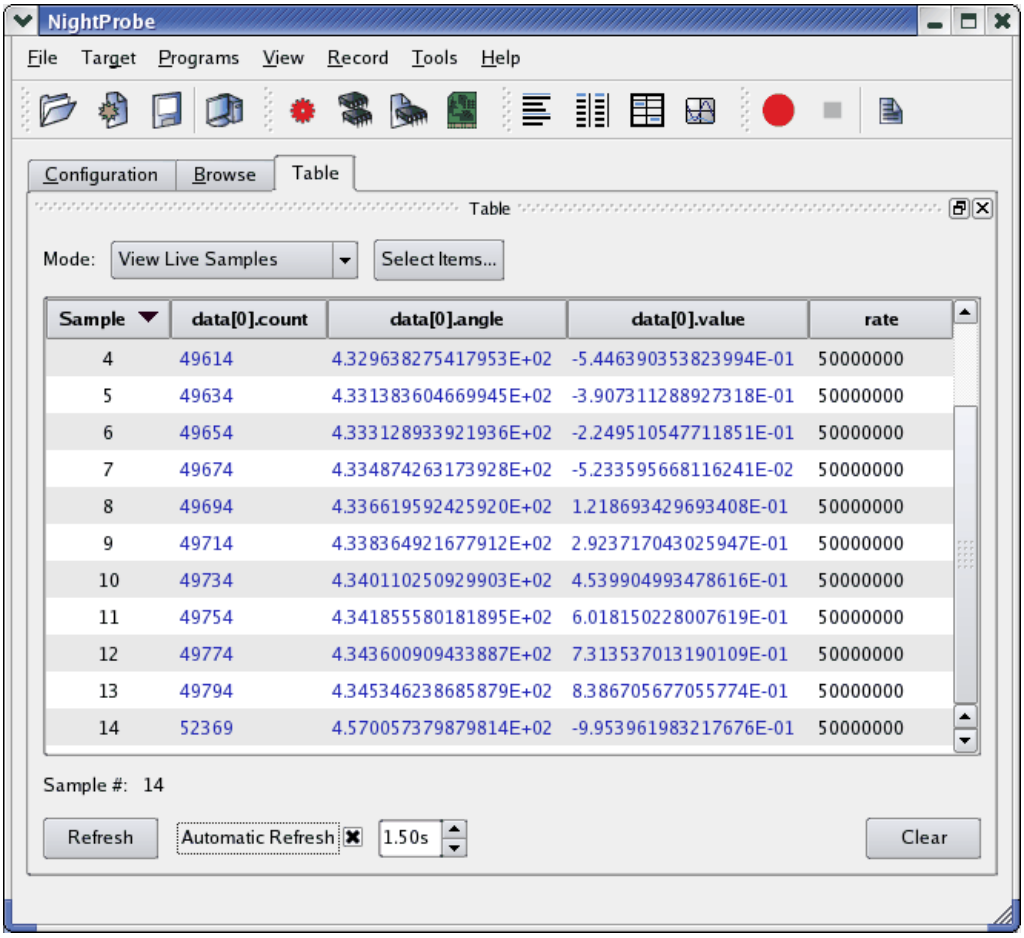


Figure 5-10. Table in Auto Refresh Mode

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 Refresh 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.

Graph View

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

- Select the Add New Page option from the View menu.
- Select the Graph option from the View menu.

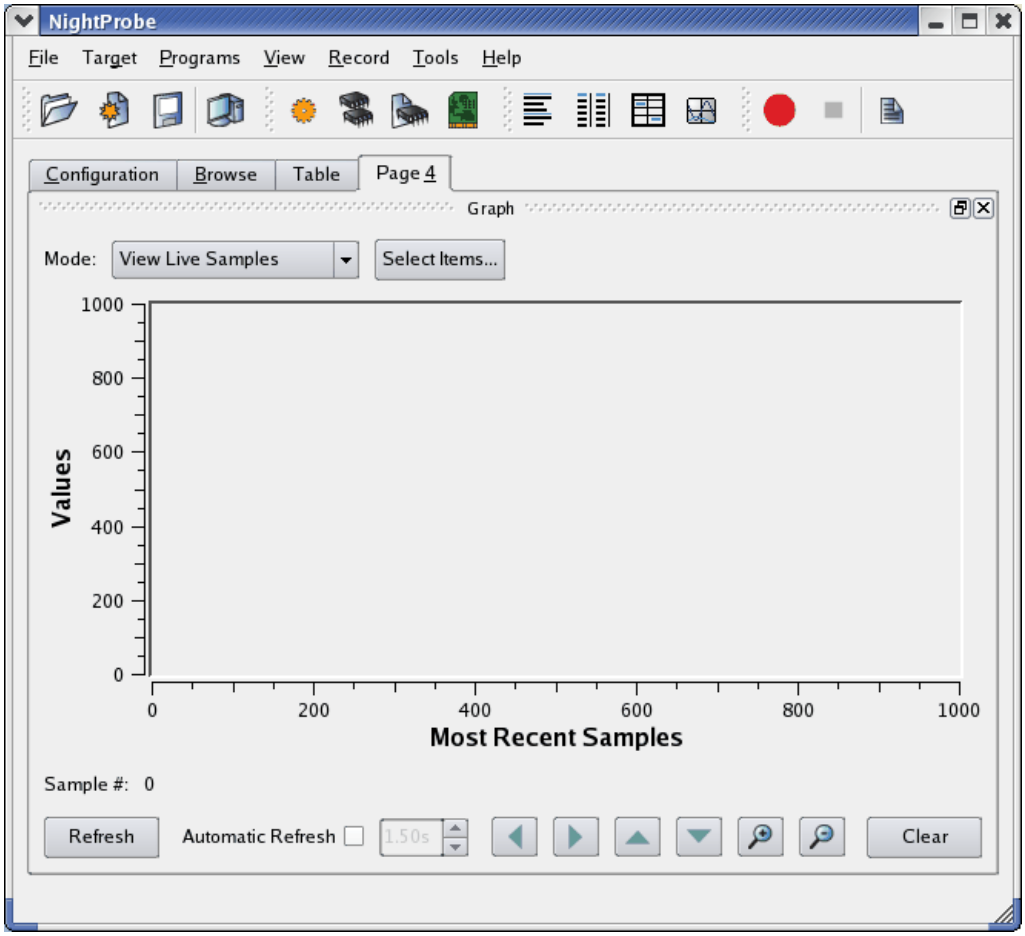


Figure 5-11. Graph Panel

Initially, the graph is empty.

- Press the Select items... button.

Unlike the table view, none of the items in the Select Item 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 as Shown by clicking their respective rows in the Show column.
- Press the OK button.
- Check the Automatic Refresh checkbox.

- Change the refresh rate to 1.0 seconds in the spinbox to the right of the Automatic Refresh checkbox.

Two lines begin to be plotted as shown below.

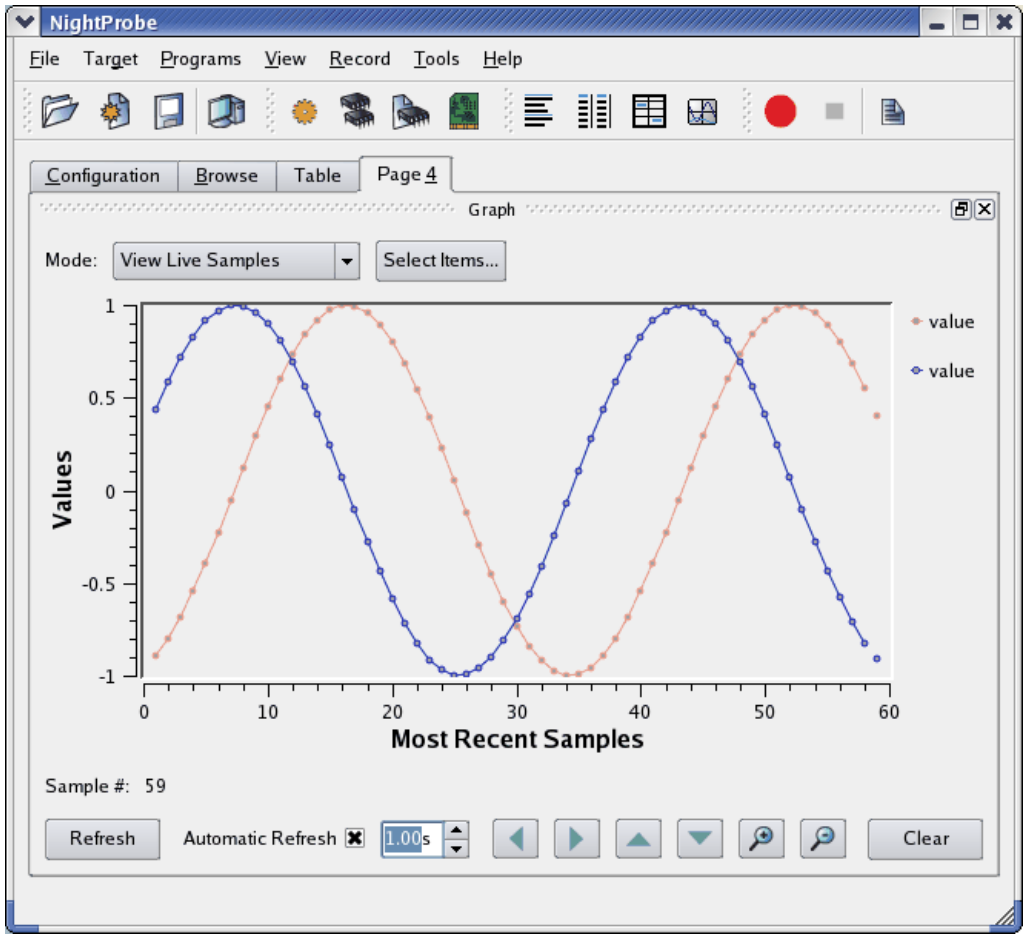


Figure 5-12. Graph Panel Actively Displaying Values

- 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-clicking activates

the context menu).

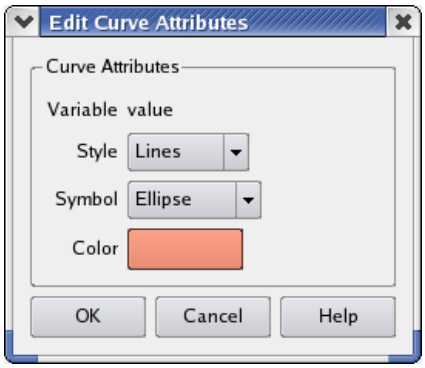


Figure 5-13. Edit Curve Attributes Dialog

- Select Sticks from the Style option list.
- Click on the colored block to activate a color selection dialog to change the color.
- Press the OK button to close the color selection dialog.

- Press the OK button to close the Edit Curve Attributes dialog.

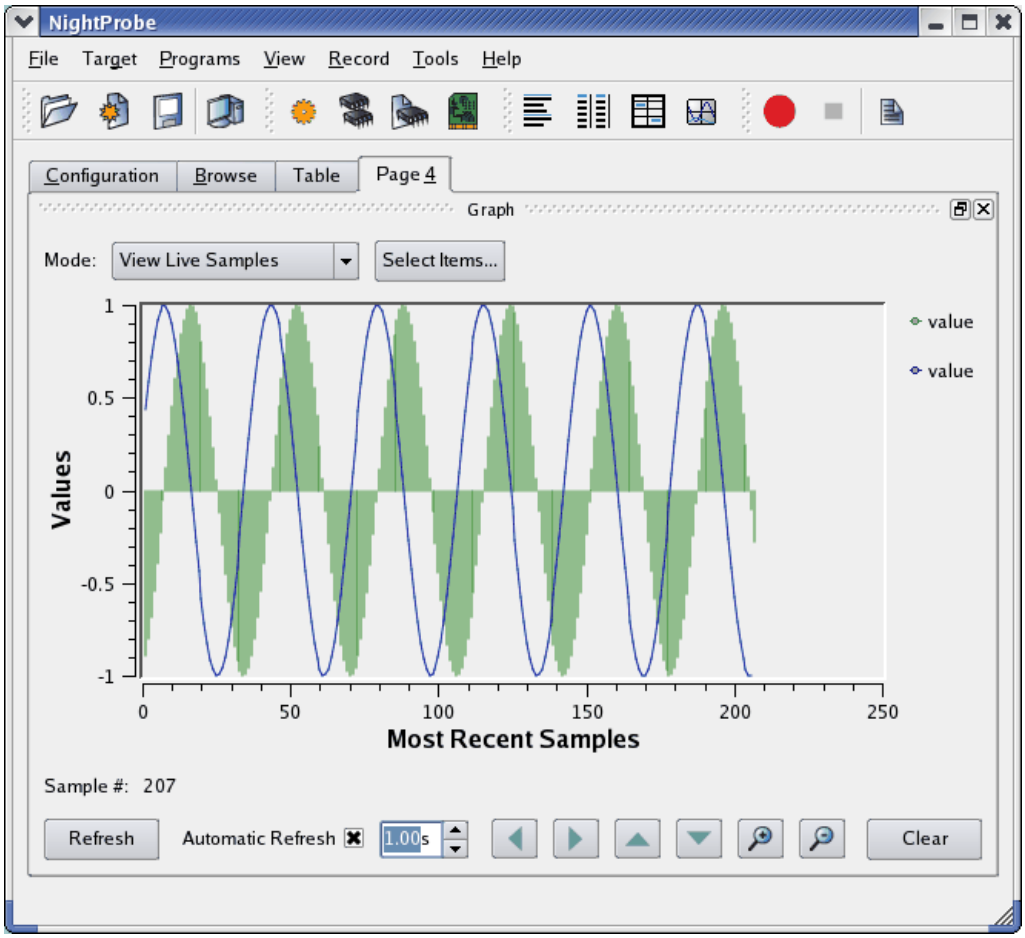


Figure 5-14. Graph Panel with Modified Curves

You can zoom in and out on the graph to see more detail.

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

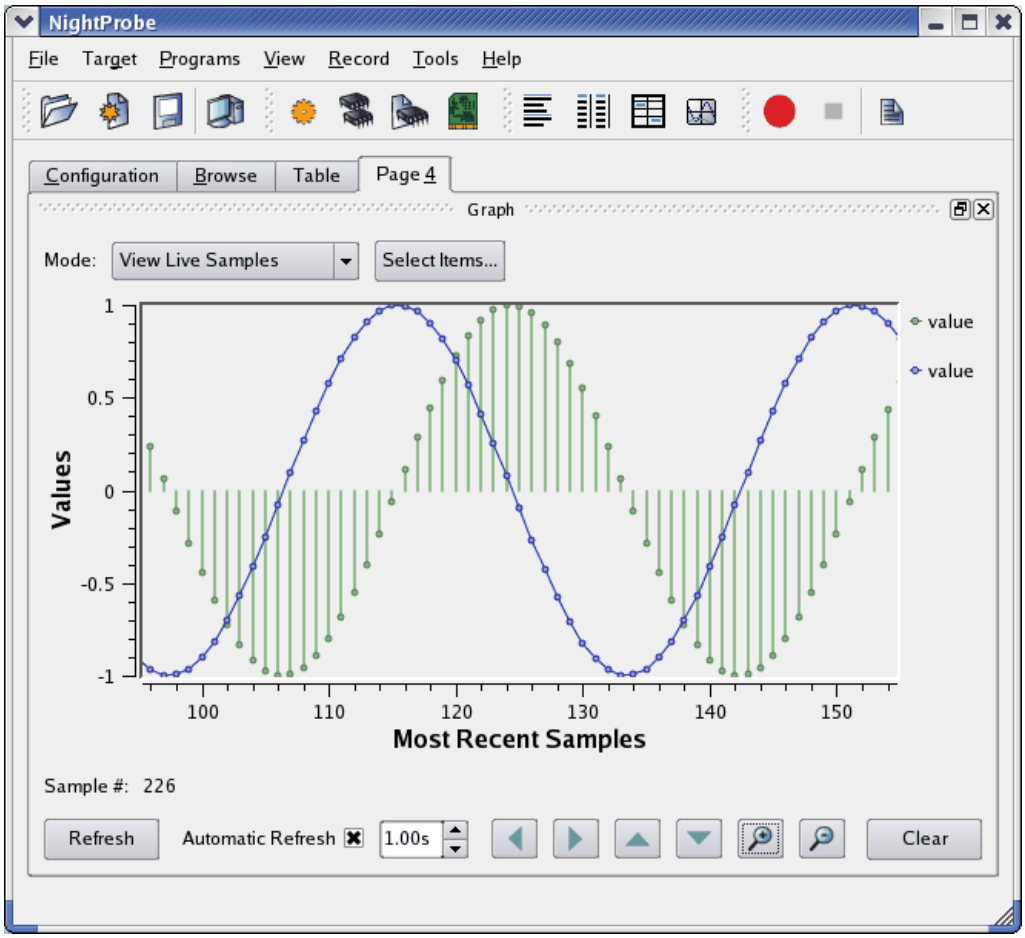


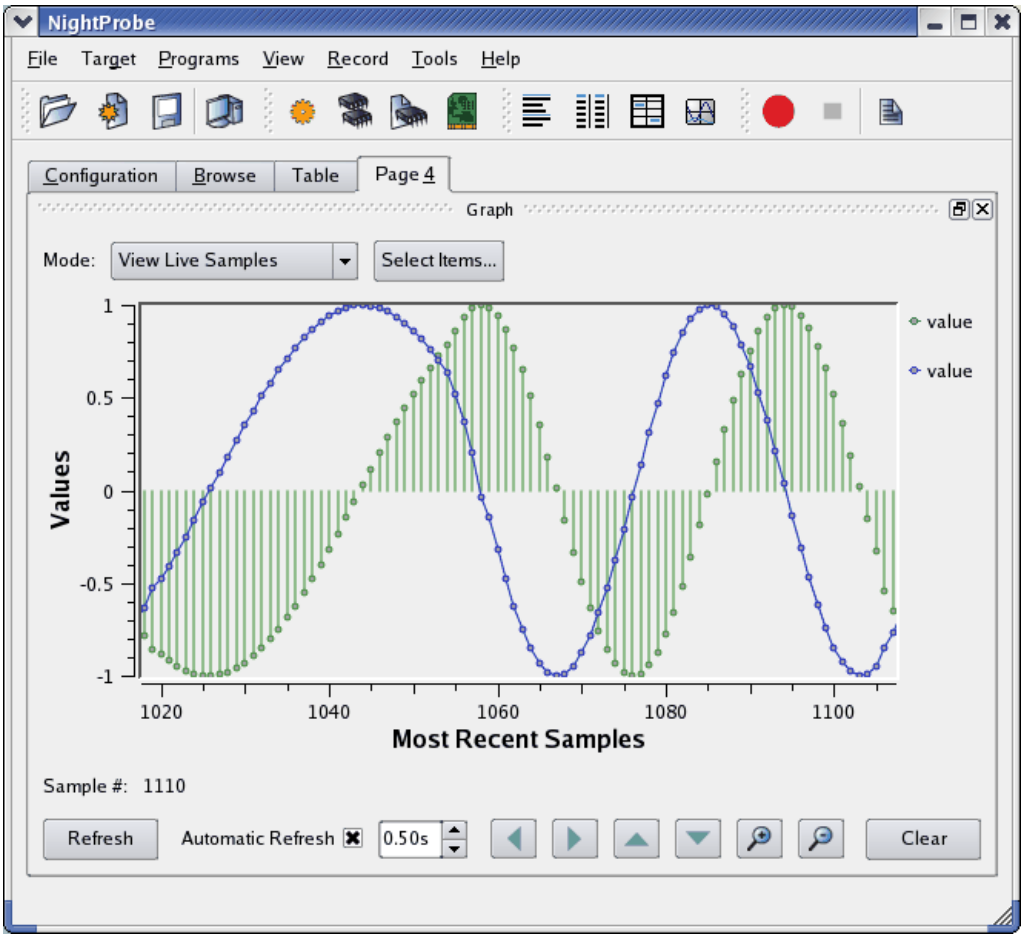
Figure 5-15. Graph Zoomed In

- 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`.

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



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.

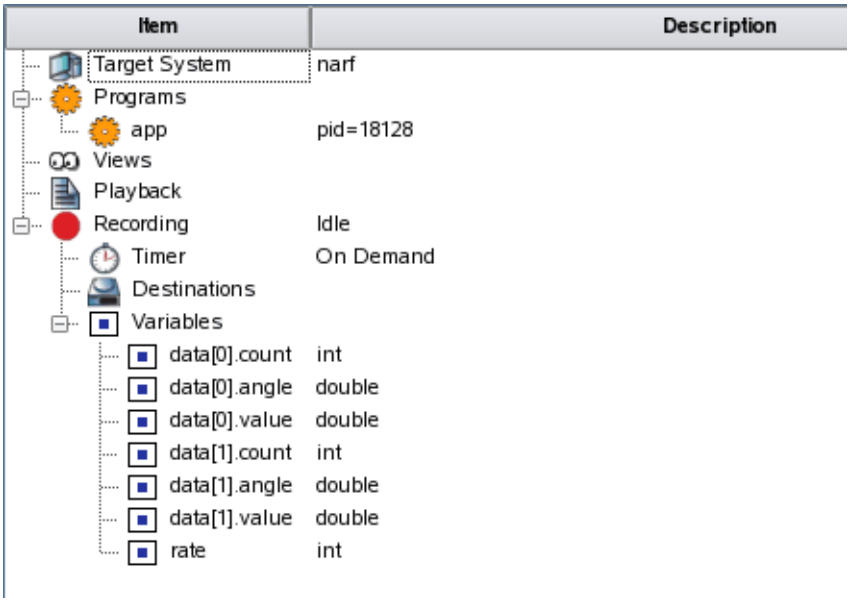


Figure 5-16. Recording area of Configuration Page

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.

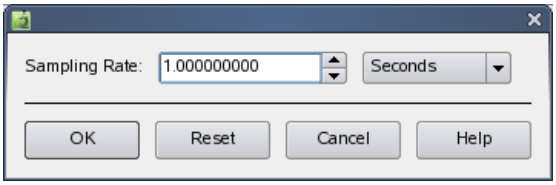


Figure 5-17. Clock Selection Dialog

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

- Change the units to Milliseconds from the option list Sampling Rate option list.
- Change the Sampling Rate value to 100.0.
- 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...

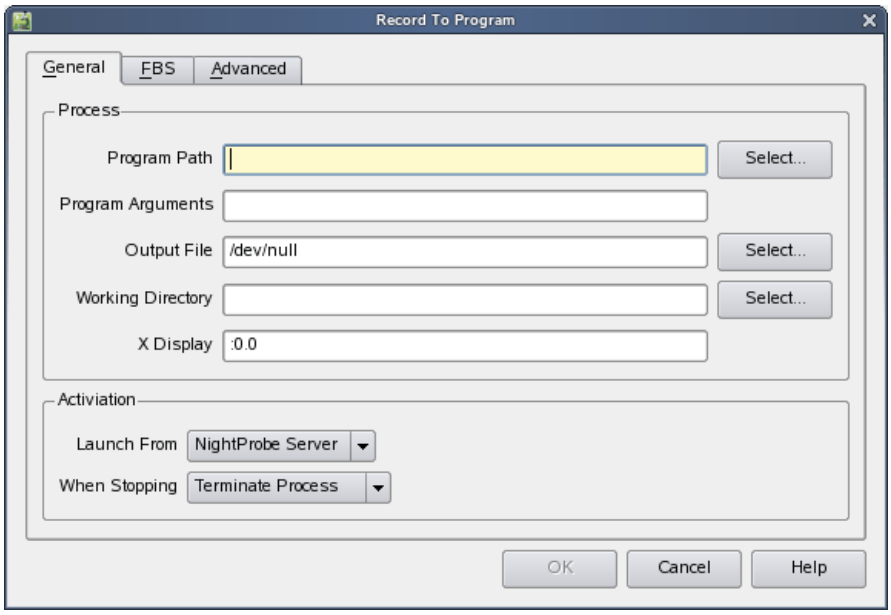


Figure 5-18. Record To Program Dialog

- 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 1-3.

- Type the following command in your terminal session to build the program:

  ```
  cc -g -o api api.c -lnprobe
  ```

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

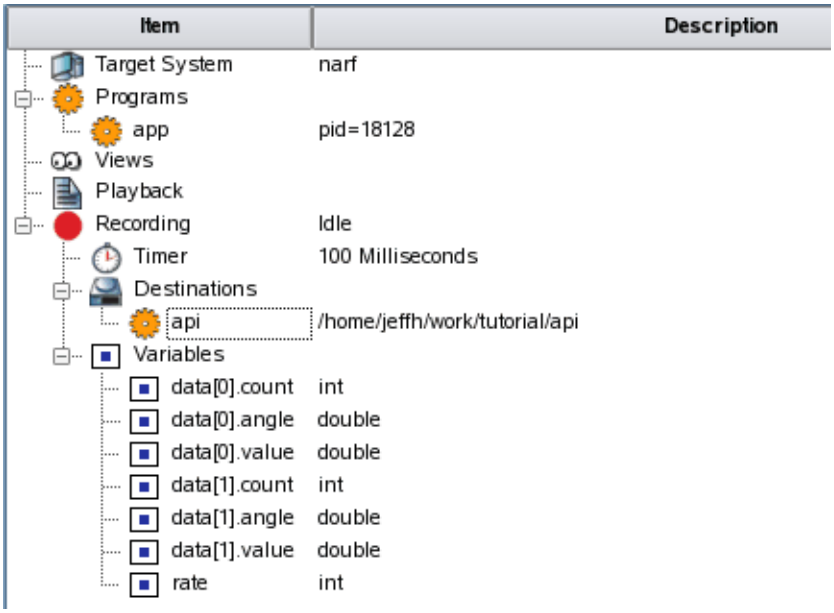


Figure 5-19. Recording Area of Configuration Page w/ Destination

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:

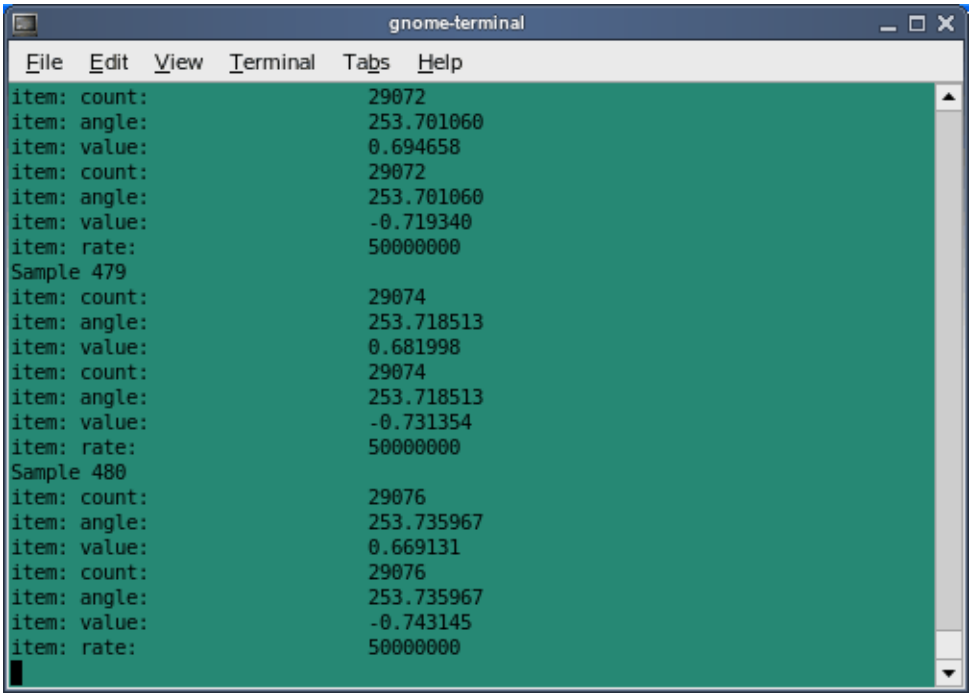


Figure 5-20. Example Output of Graph Program

- Stop the recording process by pressing the Stop icon on the Recording toolbar:

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, 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 1-3.

- Compile the program using the following command:

```
cc -g -o set_rate set_rate.c -ldatamon -lccur_rt
```

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 labelled Page 4 in Night-Probe.
- 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:

- Select the Exit Immediately option from the File menu

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, do so using the instructions in “Building the Program” on page 1-4 and execute the application before proceeding:

Invoking NightTune

NightTune can be launched with the following command at a command prompt:

```
ntune &
```

Or it may be launched 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.

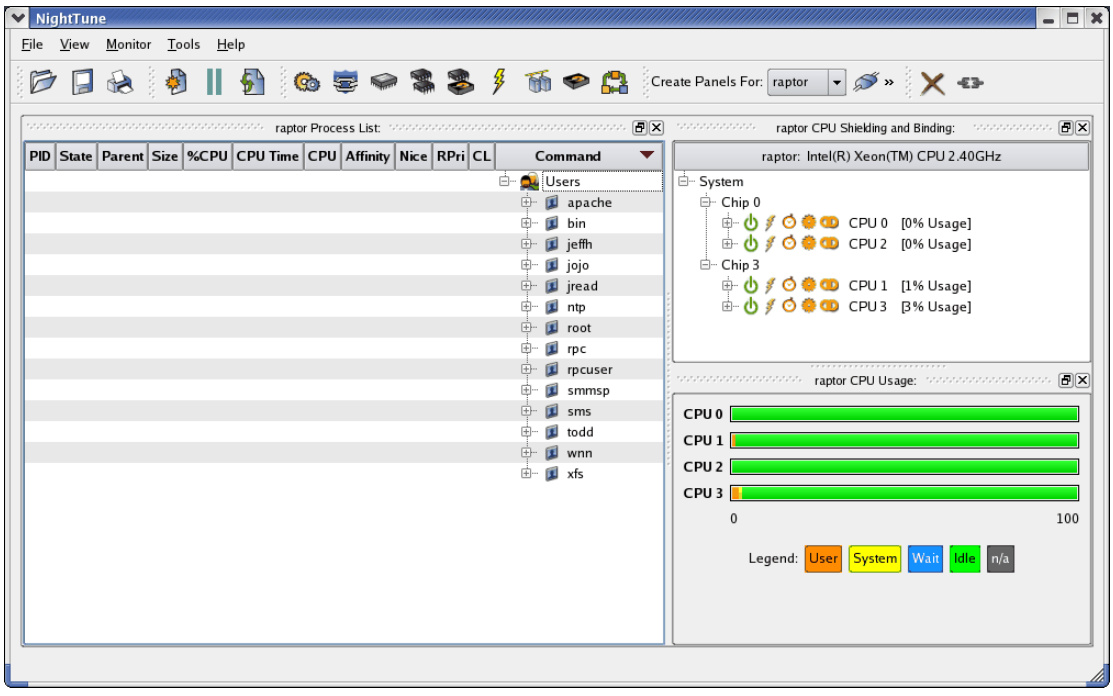


Figure 6-1. NightTune initial panels

Monitoring a Process

First monitor the running **app** process.

- In the Process List panel on the left side of the window, select Expand All from the context menu associated with the item matching your user ID.
- Locate the **app** process that appears under your users's processes.

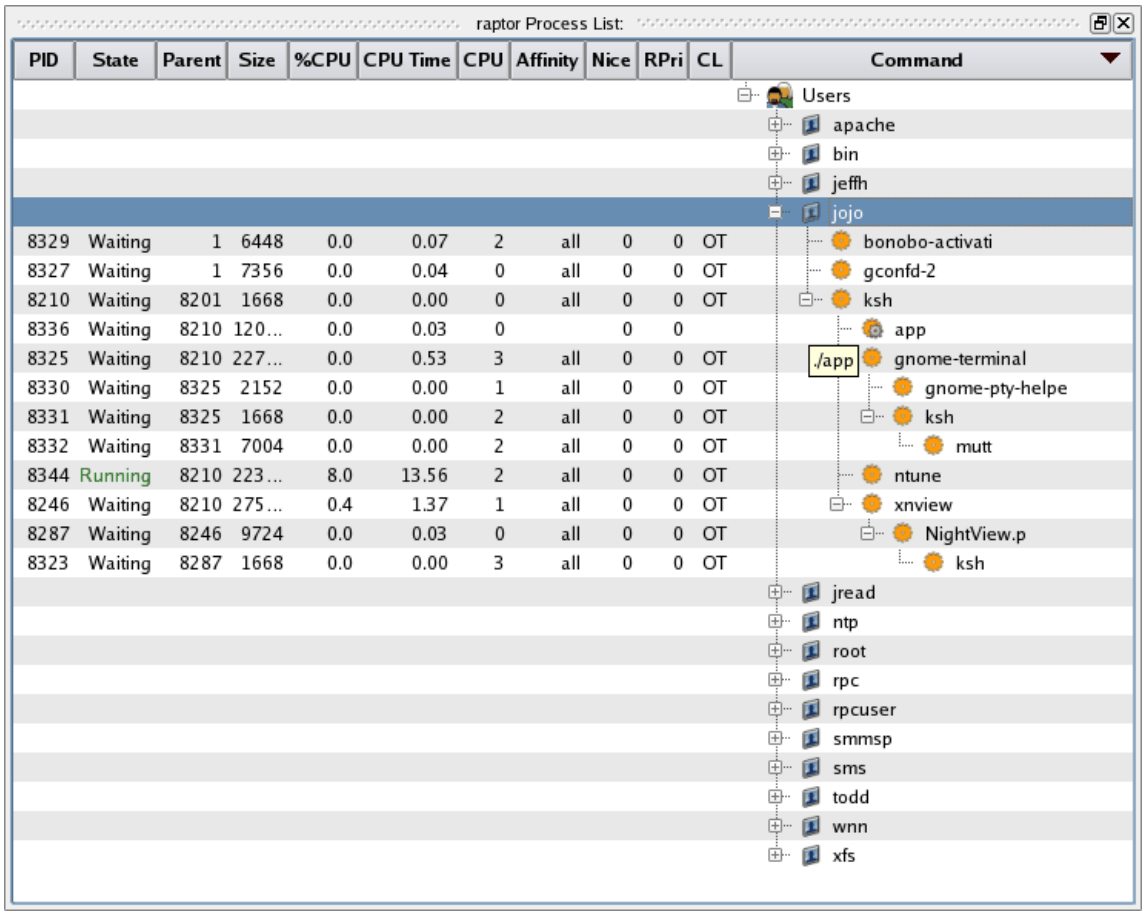


Figure 6-2. Expanded Process List

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.

- Select the Show Threads option from the context menu associated with the **app** process.

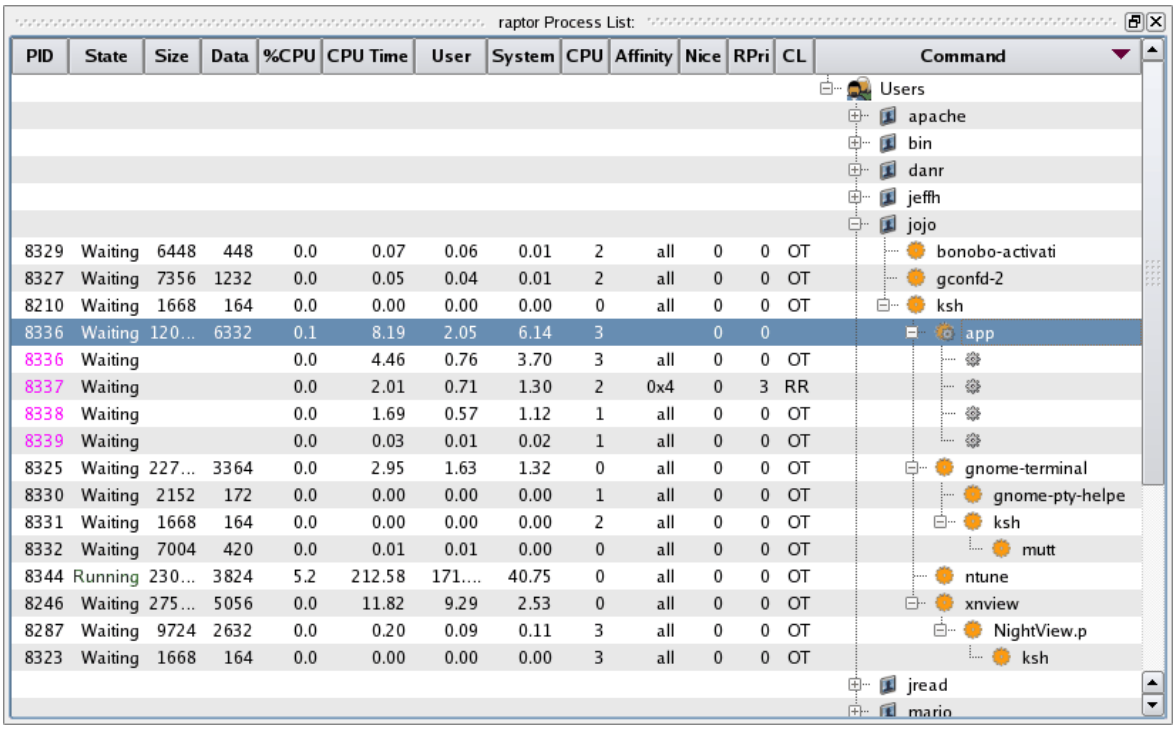


Figure 6-3. Process List with Threads

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 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.

- Select the Trace System Calls... option from the context menu associated with the second thread in the **app** program.

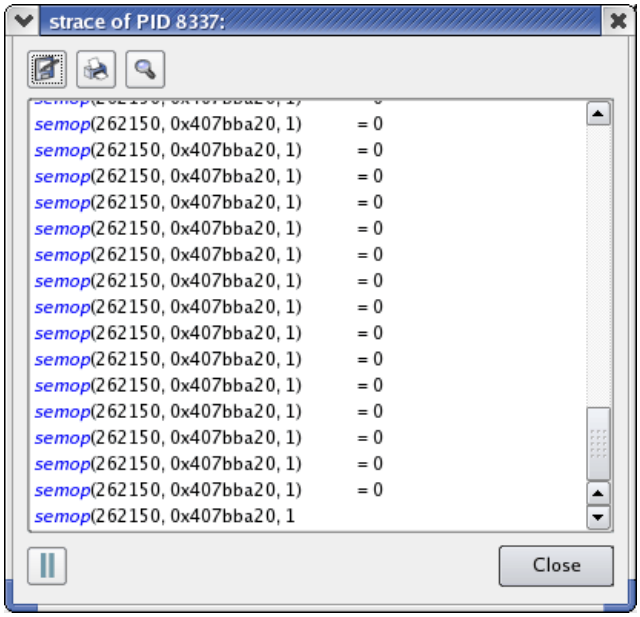


Figure 6-4. Strace Output of Thread

As shown in the figure above, the selected thread makes no system calls other than **semop(2)** which is associated with the line 46 of **api.c**, as shown in this code segment:

```
36 static
37 void *
38 sine_thread (void * ptr)
39 {
40    control_t * data = (control_t *)ptr;
41    struct sembuf wait = {0, -1, 0};
42
43    trace_open_thread (data->name);
44
45    for (;;) {
46       semop(sema, &wait, 1);
47       data->count++;
48       data->angle += data->delta;
49       data->value = sin(data->angle);
50    }
51 }
```

- 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.

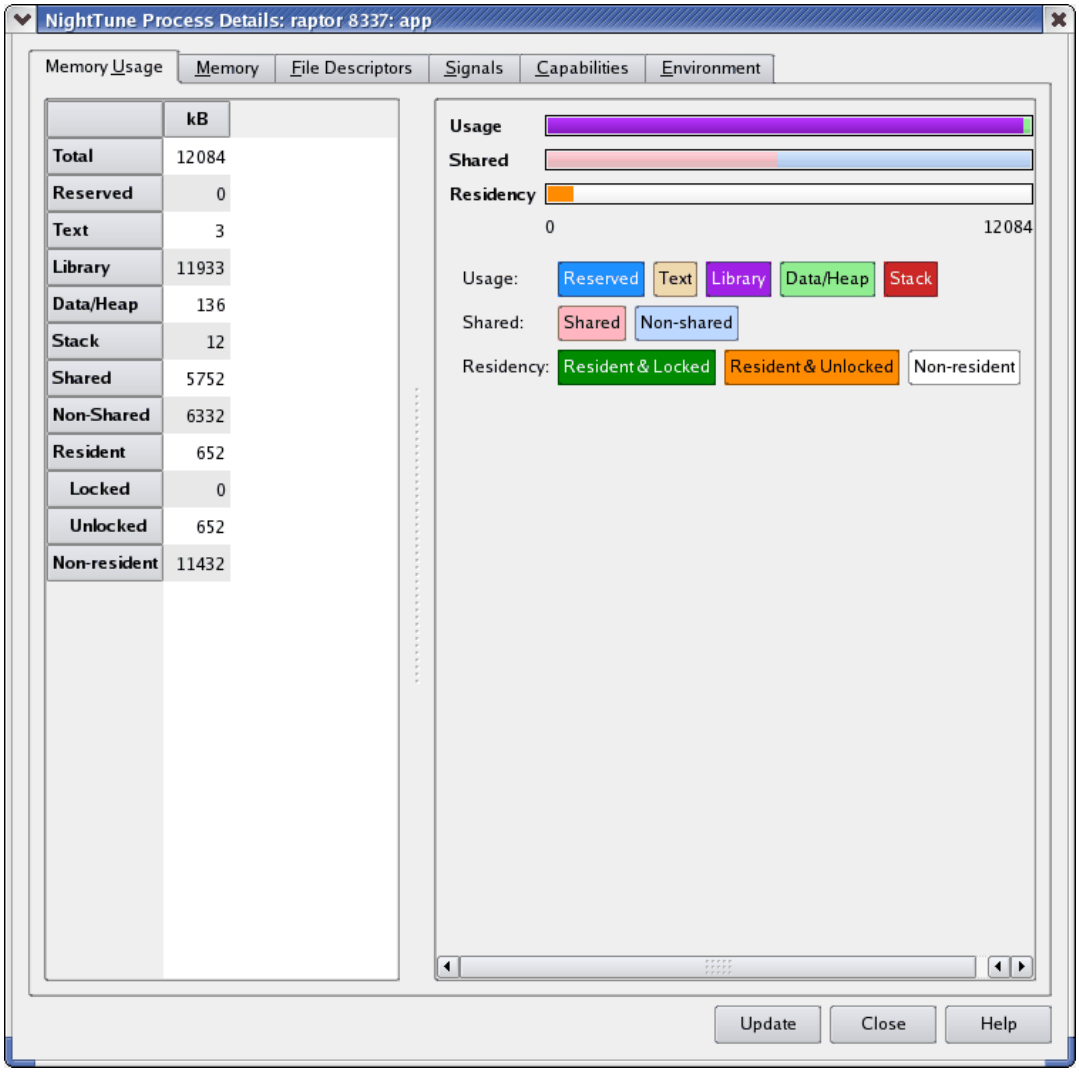


Figure 6-5. Process Details Dialog

All information displayed in this dialog is read-only in nature. You cannot make changes to process attributes using this dialog.

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

- Memory Usage
- Memory details

- File Descriptors
- Signals
- Capabilities
- Environment

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.

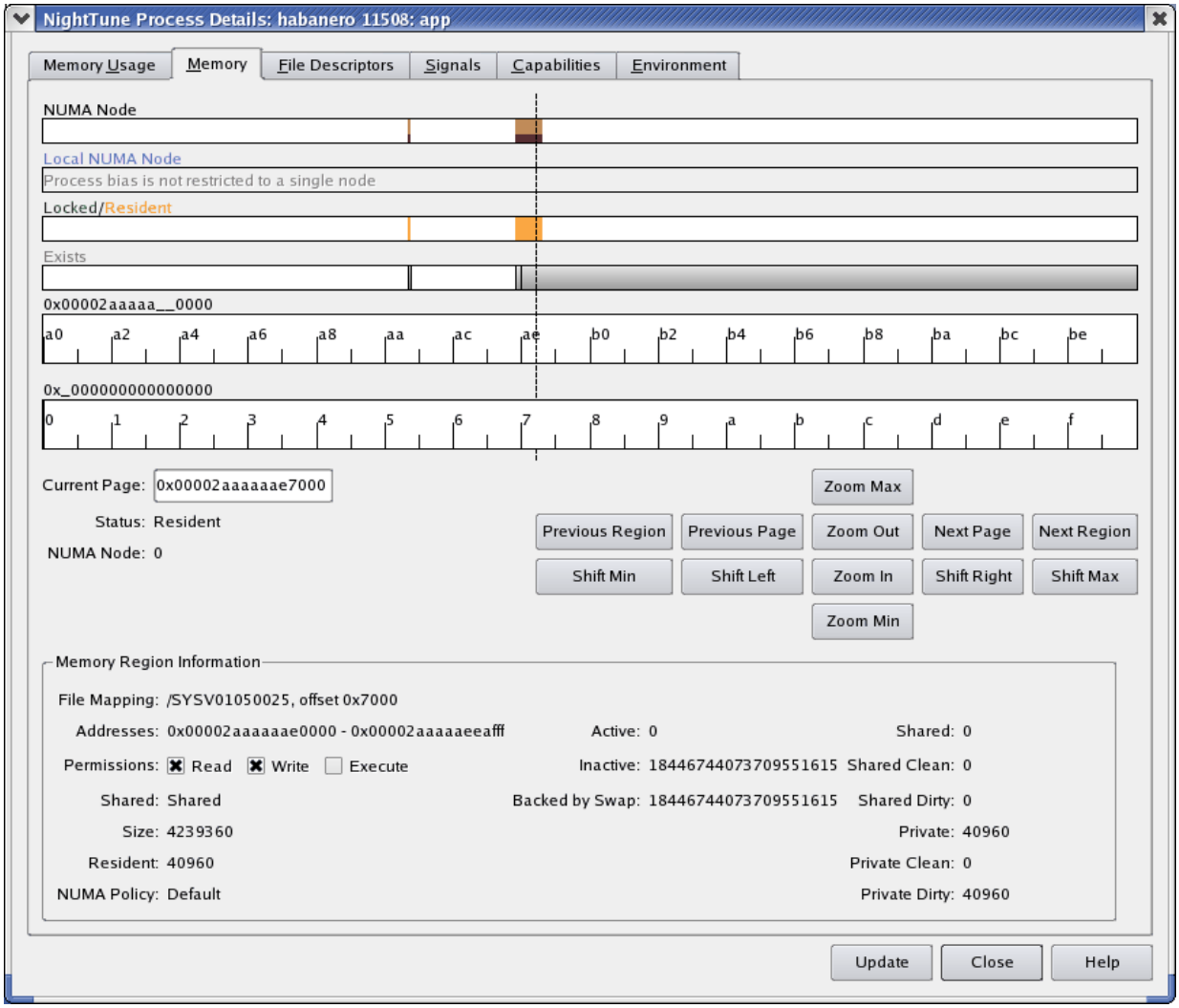


Figure 6-6. Process Memory Details Page

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+UpArrow 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.

- Click on one of these lines
- 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 - File Descriptors

The File Descriptors page 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.

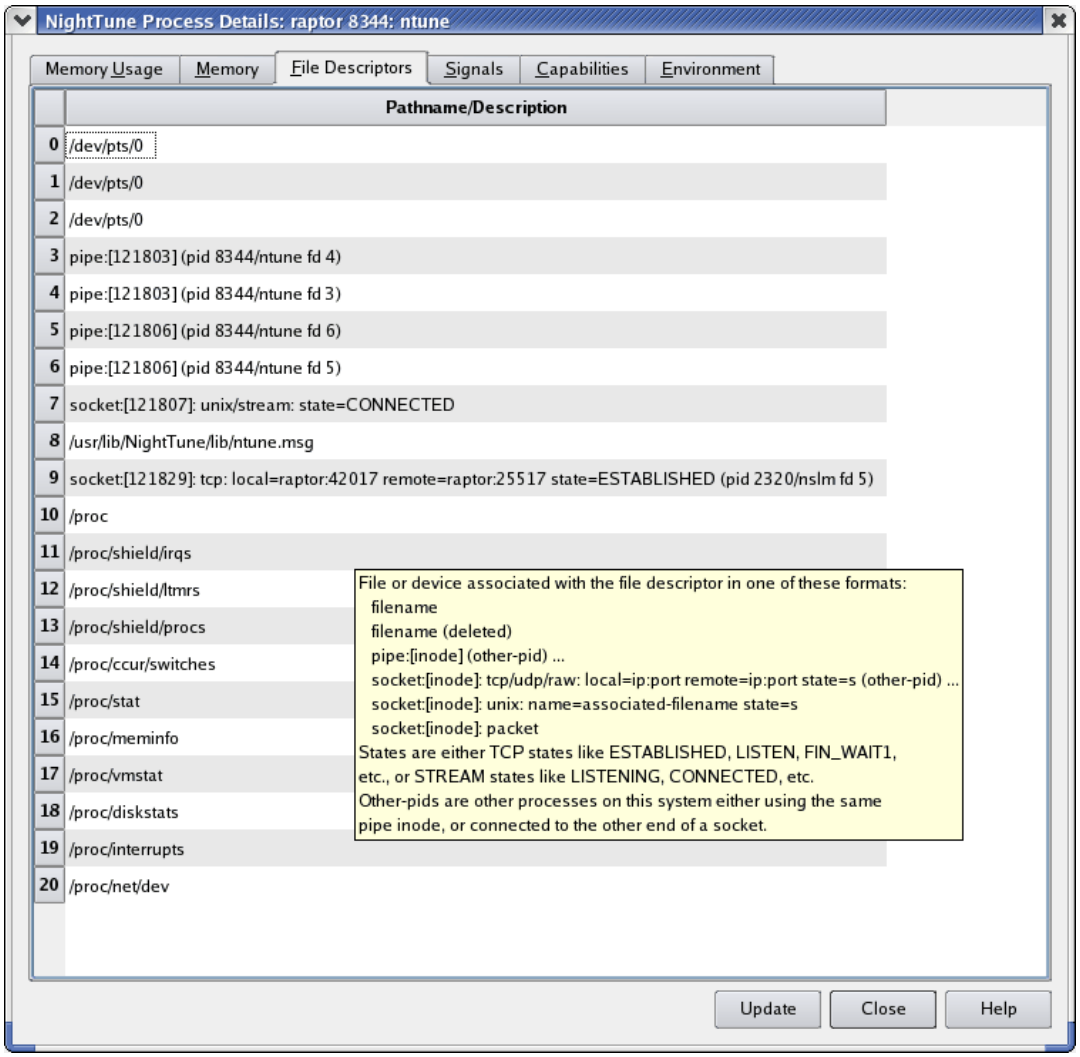


Figure 6-7. File Descriptors Page

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.

Process Details - Signals

The Signals table displays attributes of signals.

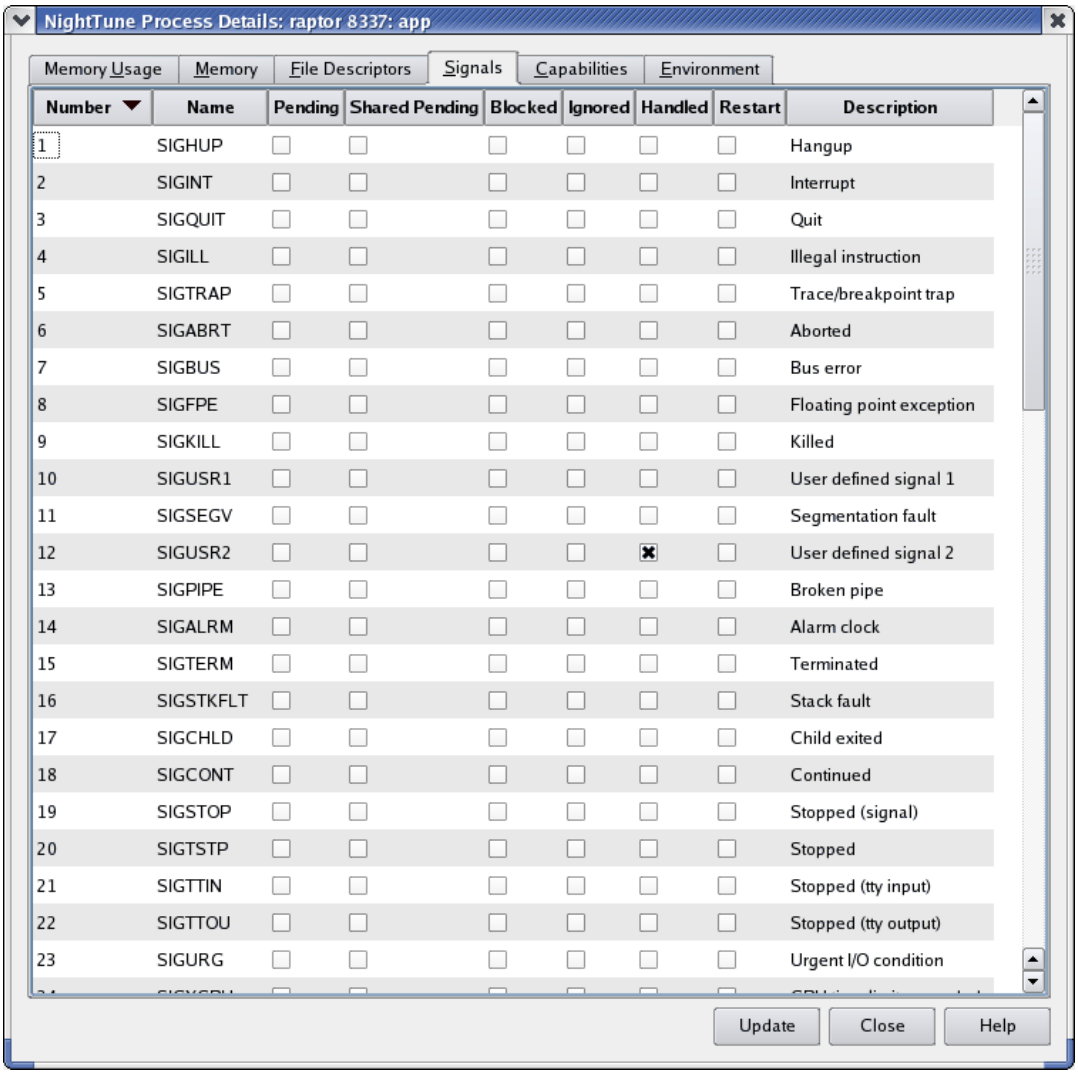


Figure 6-8. Signals Page

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.

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.

- Select the Process Scheduler... option of the context menu associated with a thread in the **app** process.

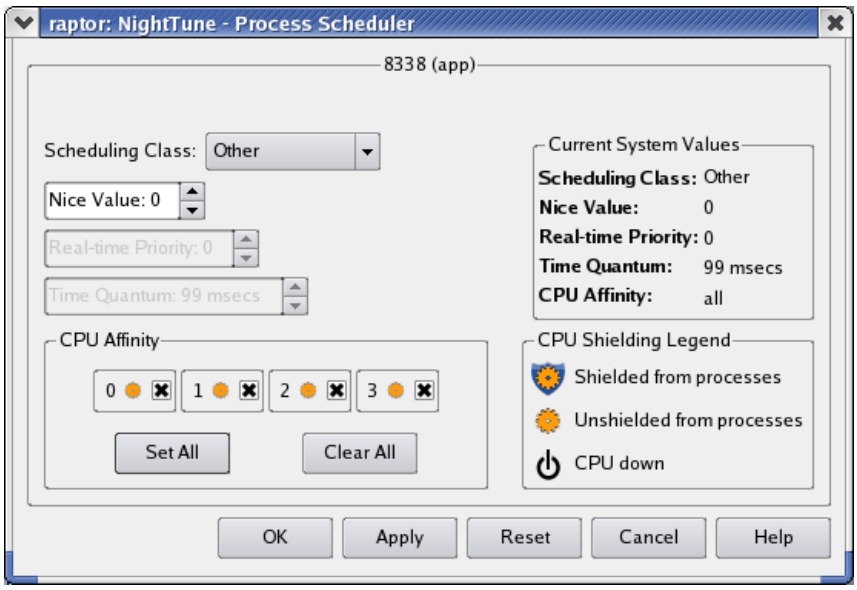


Figure 6-9. Process Scheduler Dialog

In this dialog, it is possible to change the Scheduling Class, Nice Value, Real-time Priority, and/or Time Quantum. On multi-processor systems, it also is possible to 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.

- Change the Scheduling Class to Round Robin by selecting that from a drop down list.
- Change the Real-time Priority to 3.
- Press the OK button.

NOTE

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

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

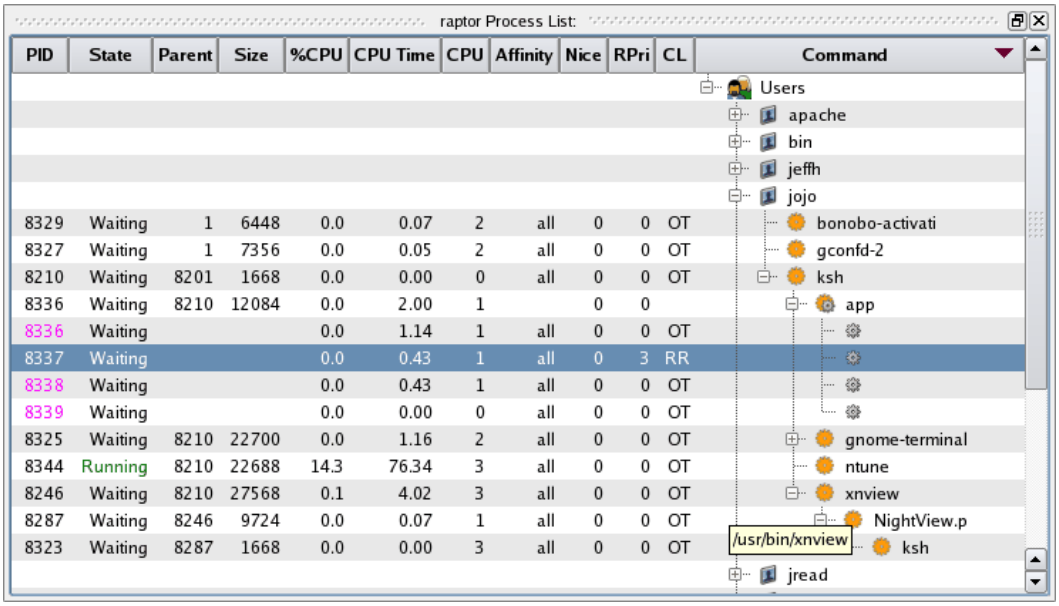


Figure 6-10. NightTune Process List with modified thread

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, CPU usage, and process and IRQ bindings.

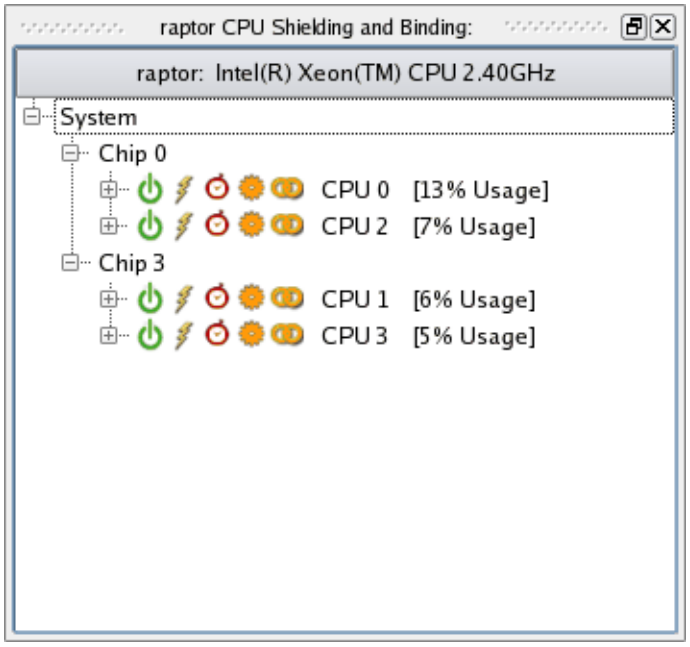


Figure 6-11. CPU Shielding and Binding Panel

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, two chips each contain two local CPUs which are hyper-threaded siblings of each other. 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 6-10 described one way to change the CPU affinity. In addition, the CPU Status 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.

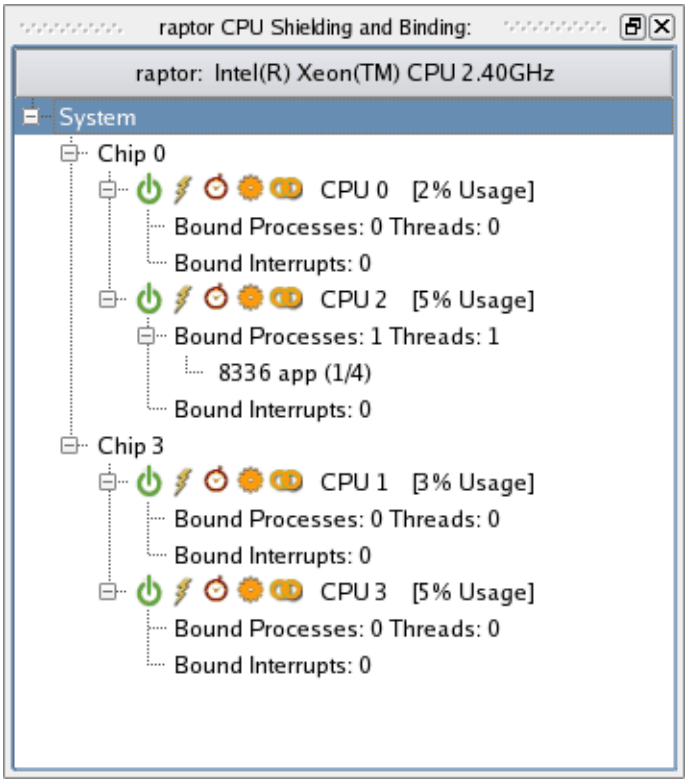


Figure 6-12. CPU Shielding and Binding Panel with Bound Thread

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' or thread's CPU affinity contains only a single CPU, that process or thread is listed in the CPU Status panel under the particular CPU's Processes tab. In this case, there is one entry under CPU 1. Because only one thread was bound to CPU 1 in this example, the entry includes the suffix (1/4), indicating that only 1 of the 4 threads is bound to that CPU.

The thread's new CPU affinity also is reflected in the Affinity field of the Process Monitor 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 0x1 has only the lowest bit turned on, indicating CPU 0.

NightTune also can unbind a process quickly.

- While the cursor is over the thread entry in the CPU Status 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.

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 right-hand most box in its title bar.
- In its place, open the Interrupt Activity panel by selecting the Interrupt Activity option from the Monitor menu and then the Text Pane option from its sub-menu.

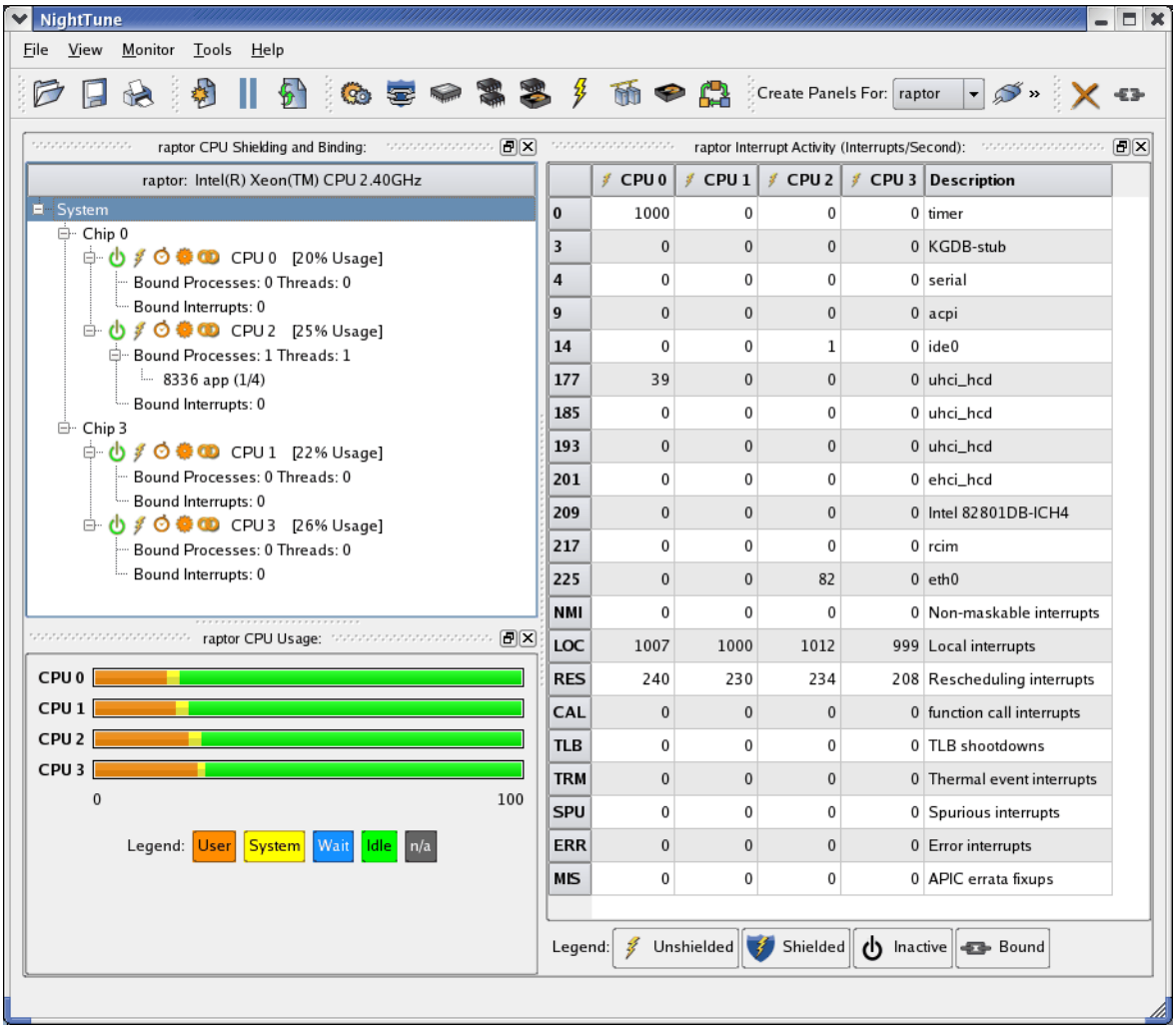


Figure 6-13. NightTune with Interrupt Activity Panel

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 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:

```
/sbin/service irqbalance stop
```

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 Activity panel, you may press and hold the *left* mouse button, 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 middle 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

If you are not running as the root user or your user lacks appropriate privileges, the Set CPU Affinity... option will not be present in the context menu.

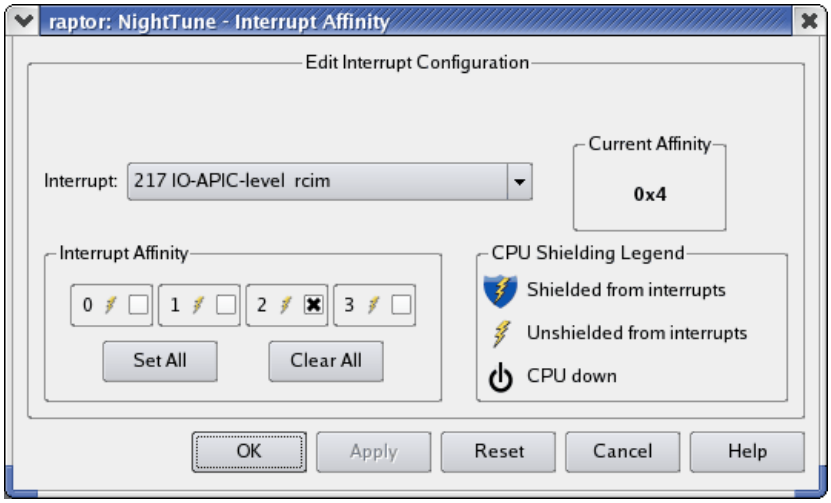


Figure 6-14. Interrupt Affinity Dialog

For each CPU on which the interrupt is allowed to be handled, the checkbox with the number of that CPU should be checked. 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 7-9 for more information.

Conclusion - NightTune

The remaining portion of the tutorial is unrelated to the execution of the **app** program. Terminate the program by executing the following steps:

- Select the Add Page option from the View menu.
- Add a Process List panel using the Monitor menu and locate the **app** process under your user name in the panel.
- Drag the **app** process using the left mouse button to the Kill icon on the toolbar and release.

- Terminate NightTune by selecting Exit from the File menu.

This concludes the NightTune portion of the NightStar RT Tutorial.

7
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
```

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 all 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.

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 wave 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 `fbswait()` 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 `fbswait()` until its next scheduled cycle.

The **`wave.c`** source file was copied from **`/usr/lib/NightStar/tutorial`** into the current working directory in an earlier portion of this tutorial.

Compile and link the application using the following command:

```
cc -g -o wave wave.c -lccur_fbsched -lccur_rt
```

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 1-3.

Launch NightSim specifying the configuration file, as show below:

```
nsim -f nsim.config -offline &
```

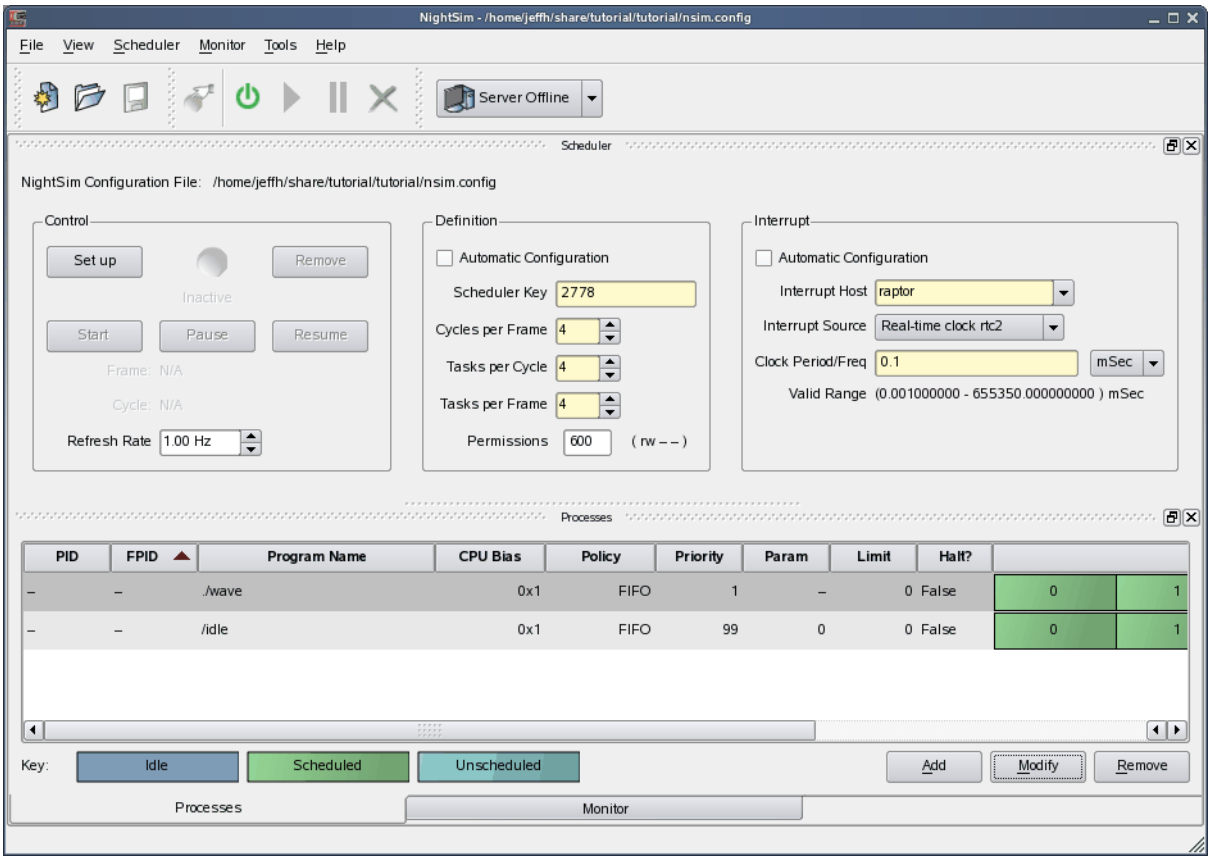


Figure 7-1. NightSim Initial Window

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.config`** 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 RTC2 of the Real-time Clock and Interrupt Module device (RCIM)
- Clock Period -- a cycle time of 100 microseconds
- Processes -- a single process, wave, schedule to run on every cycle of the frame

To view the details of the attributes of the scheduled process, select the **`./wave`** process in the process area at the bottom portion of the Processes panel and then press the Modify… button in the lower right- hand area of the panel.

The Edit Process dialog is displayed.

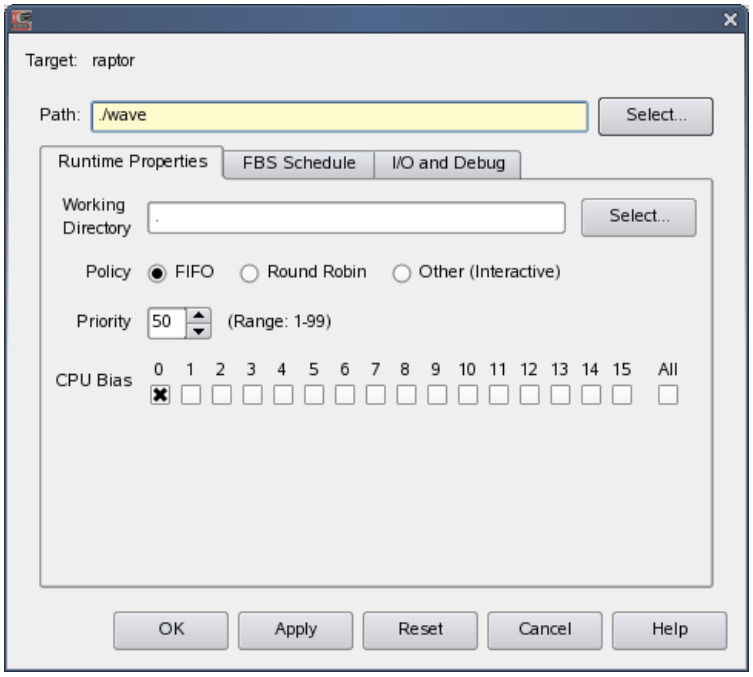


Figure 7-2. NightSim Edit Process Dialog

NOTE

The CPU Bias description area of the Process Tab may vary depending on the number of CPUs on your system.

The Runtime Properties tab allows you to chose the CPU on which execution is allowed, the scheduling policy, and the scheduling priority of the process.

Click on the FBS Schedule tab in the dialog.

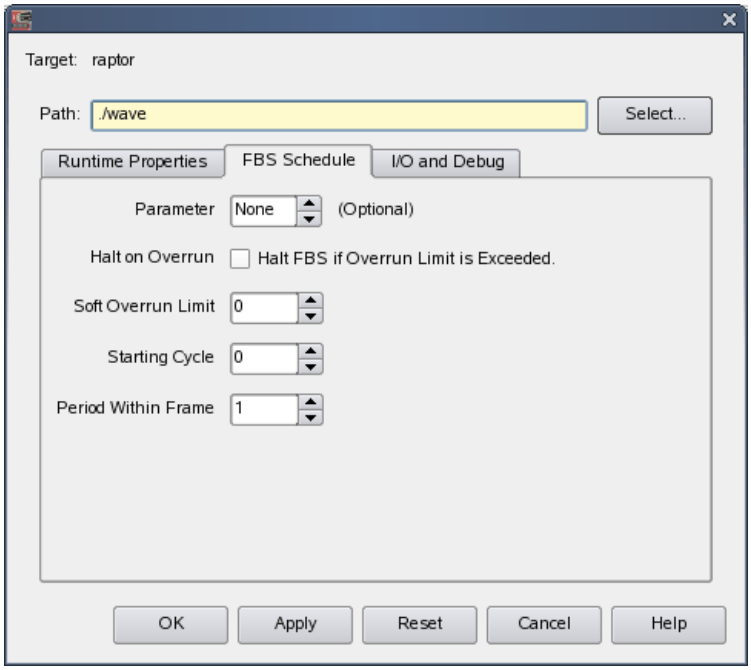


Figure 7-3. FBS Scheduler Tab

The FBS Schedule tab shows the starting cycle and period of the **`wave`** process. The Staring 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.

Close the Edit Process window by pressing the Cancel button.

Notice that in addition to the **`wave`** 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, press the Setup button followed by the Start button in the Control area.

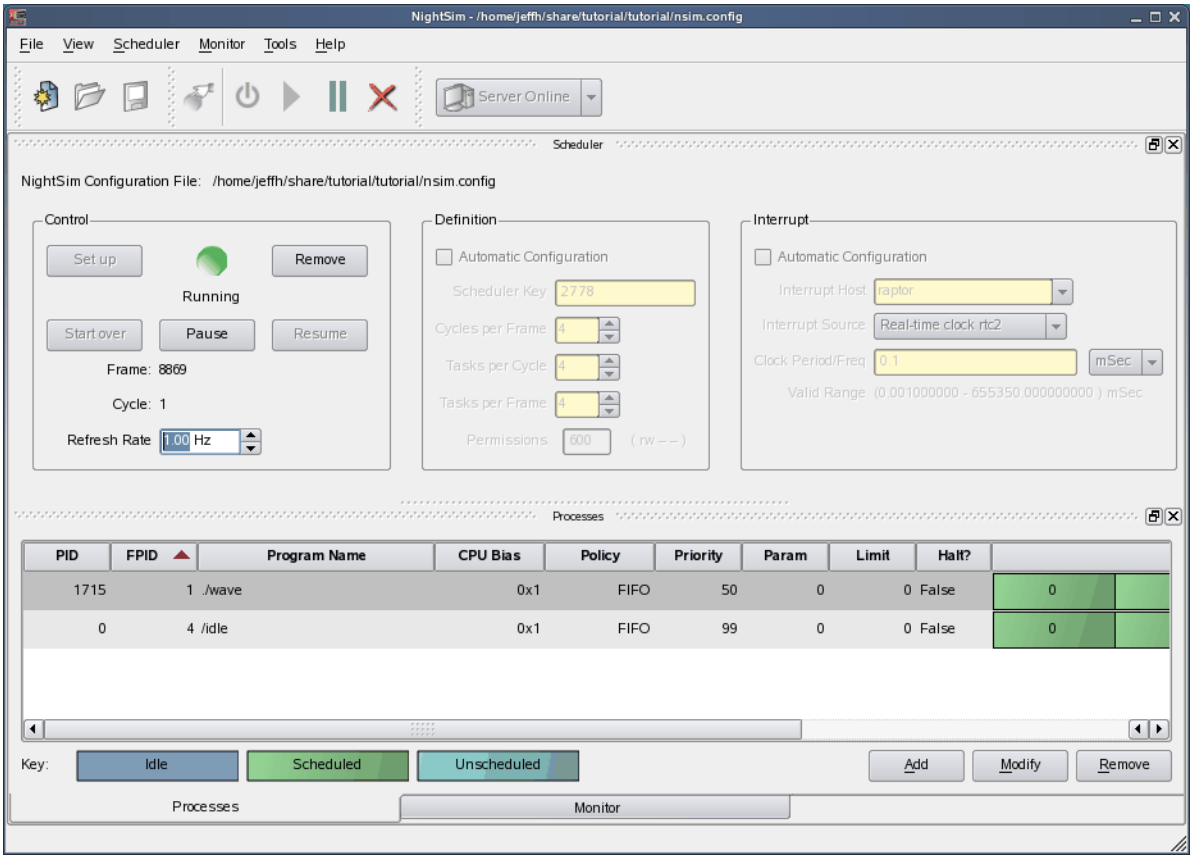


Figure 7-4. Scheduling Started

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 bottom of the window.

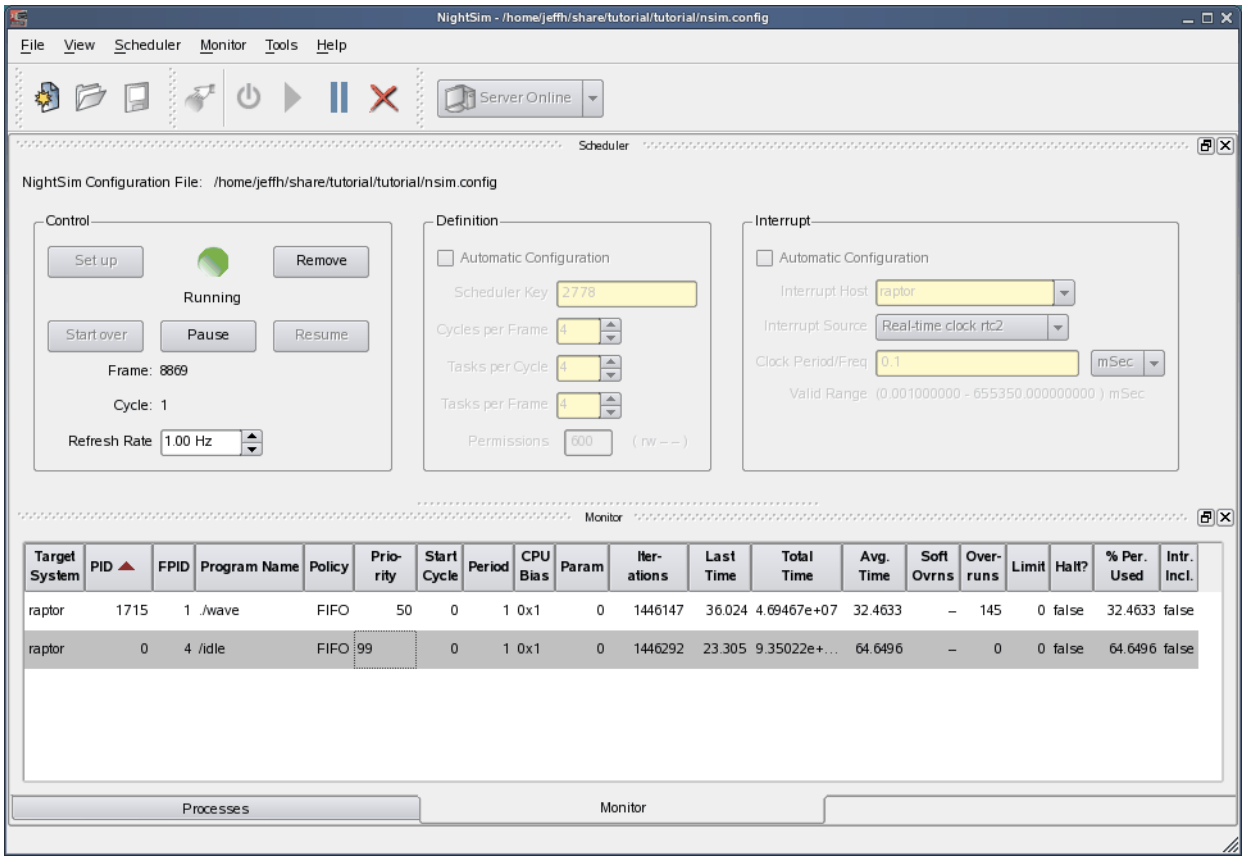


Figure 7-5. NightSim Monitor Page

The NightSim Monitor page 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 percentage of the frame used by each process. Additional statistics can be selected for display via the Display Monitor Columns option item of the Monitor menu.

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 **`wave`** process and see the effects shown in the NightSim 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, 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("wave",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 wave 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 **wave** 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 1-3.

Compile the program using the following command:

```
cc -g -o set_workload set_workload.c -ldatamon -lccur_rt
```

Change the value of the `workload` variable in the **wave** 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 **`./wave`** is affected by the reduced workload as shown in the NightSim Monitor window.

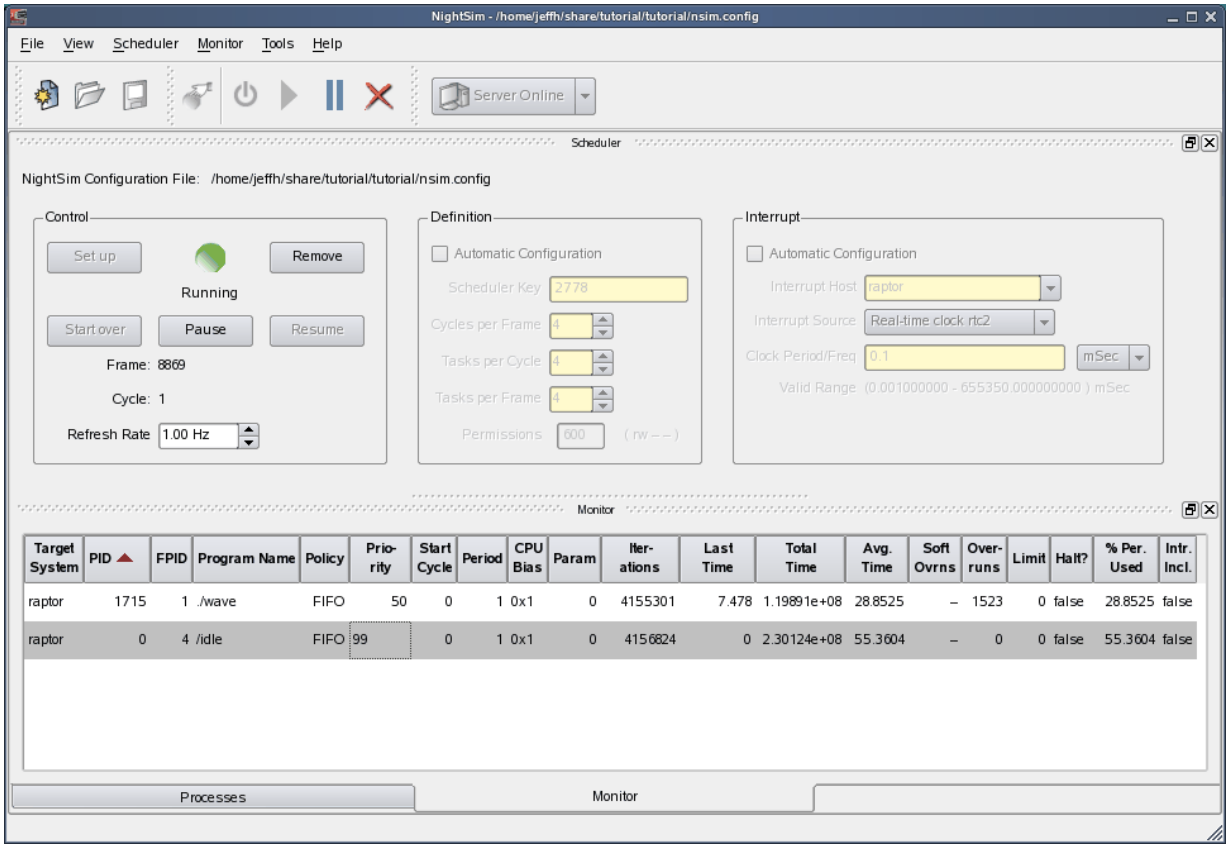


Figure 7-6. NightSim Monitor Window -- Reduced Workload

Experiment with various values of `workload` using the **`set_workload`** program until the average Last Cycle time for **`./wave`** is approximately 50 microseconds.

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 wave 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
```

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 cause of the overruns is due to additional activities unrelated to the scheduler are occurring on the CPU where **`wave`** executes.

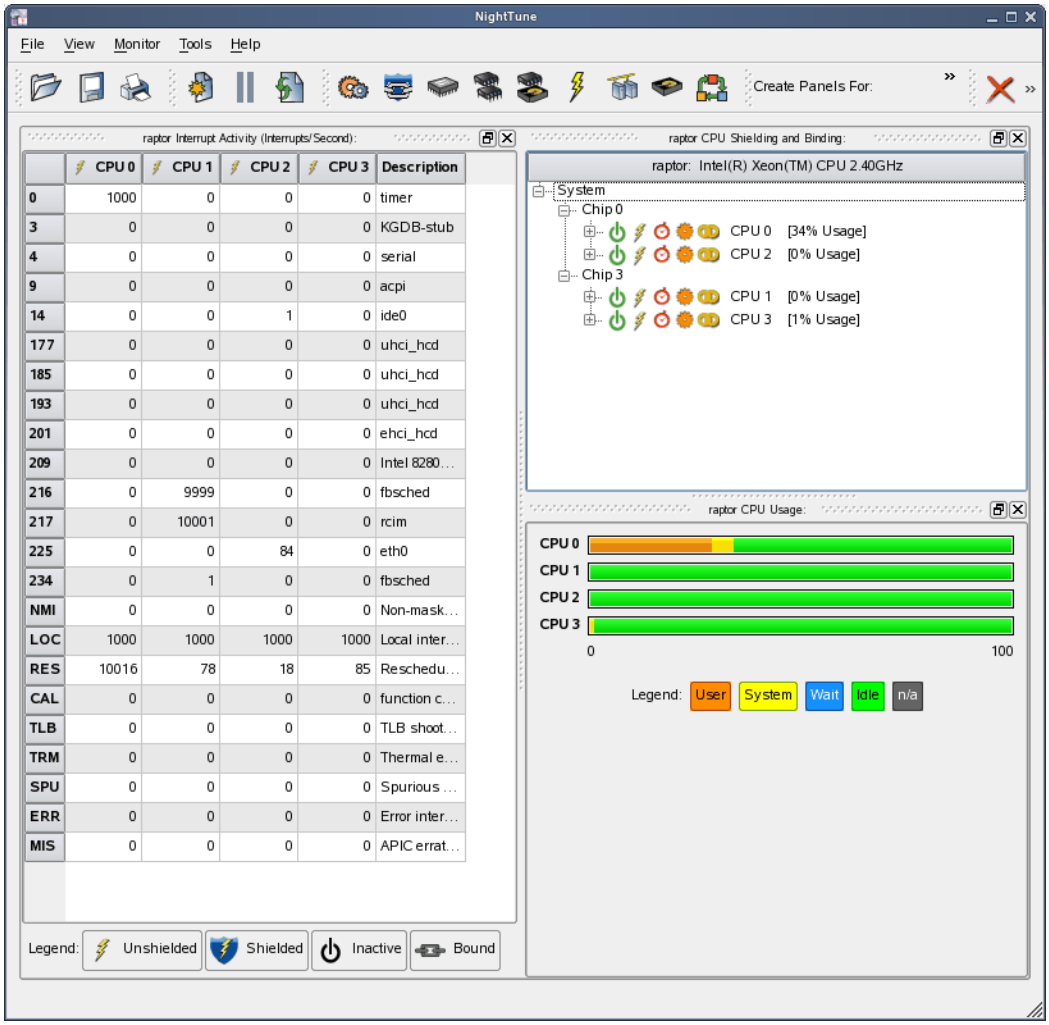
We will use NightTune to shield the CPU associated with our scheduler from other activities.

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 7-14 in this case.

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 1-3:

```
ntune -c ./ntune.config &
```



	CPU 0	CPU 1	CPU 2	CPU 3	Description
0	1000	0	0	0	timer
3	0	0	0	0	KGDB-stub
4	0	0	0	0	serial
9	0	0	0	0	acpi
14	0	0	1	0	ide0
177	0	0	0	0	uhci_hcd
185	0	0	0	0	uhci_hcd
193	0	0	0	0	uhci_hcd
201	0	0	0	0	ehci_hcd
209	0	0	0	0	Intel 8280...
216	0	9999	0	0	fbsched
217	0	10001	0	0	rcim
225	0	0	84	0	eth0
234	0	1	0	0	fbsched
NMI	0	0	0	0	Non-mask...
LOC	1000	1000	1000	1000	Local inter...
RES	10016	78	18	85	Reschedu...
CAL	0	0	0	0	function c...
TLB	0	0	0	0	TLB shoot...
TRM	0	0	0	0	Thermal e...
SPU	0	0	0	0	Spurious ...
ERR	0	0	0	0	Error inter...
MIS	0	0	0	0	APIC errat...

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.

Right-click on the System icon in the CPU Shielding & Binding panel and select Expand All from the context menu.

Note that **`wave`** process is listed in the Bound Processes list of CPU 0.

Take the following actions to bind the RCIM interrupt to CPU 0 and shield CPU 0 from all other activities:

- 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 0

row in the CPU Shielding and Binding panel, and release the mouse button. The **rcim** interupt is now bound to CPU 0.

- Right-click the row corresponding to CPU 0 in the CPU Shielding and Binding panel and select the Change Shielding... option from the context menu.

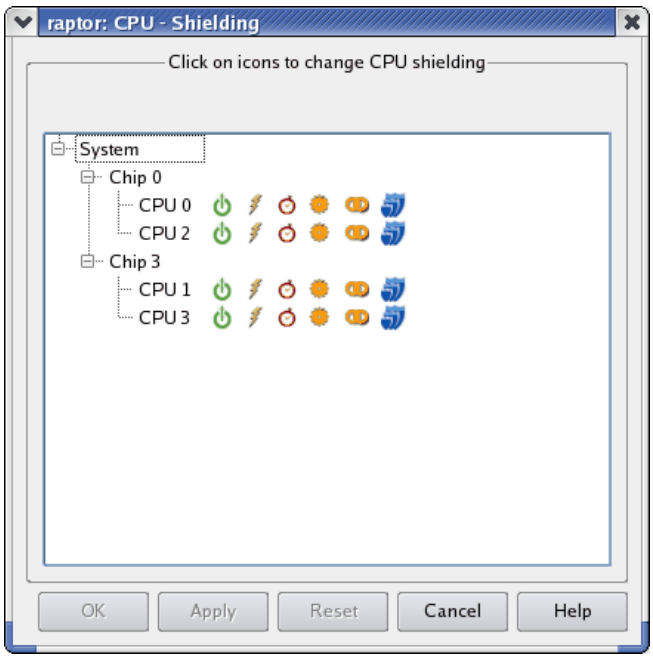


Figure 7-8. Change Shielding Dialog

- Click the Maximize Shielding icon in the CPU 0 line (the maximize shielding icon is the right-most icon with three overlapping shield figures).

The CPU 0 line changes its display to indicate that all processes and interrupts other than **wave** and **rcim** will be shielded from CPU 0. Additionally, the sibling hyper-threaded

CPU (in this case CPU 2 as shown below CPU 0) is marked down so that hyper-threaded execution on CPU 2 does not interfere with CPU 0.

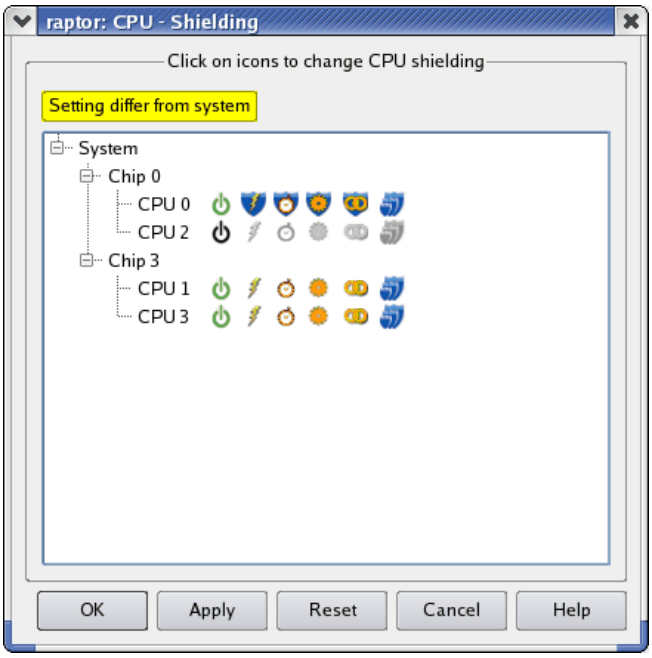


Figure 7-9. Shielding Changes Pending

NOTE

The hyperthreaded sibling of CPU 0 may be a logical CPU number other than CPU 2.

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.

- 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 selecting the Clear Performance Data option item from the Monitor menu. This action resets all the statistics to zero.

Watch the Overrun column to see if any overruns still occur.

Experiment with the **`./set_workload`** program to make the workload of the **`./wave`** application such that only 35 microseconds are left for **`/idle`** processing.

If the system is properly configured, the scheduler should continue to execute without any overruns on the shielded CPU.

Shutting Down the Scheduler

Return to the NightSim window and press the Remove button to terminate the scheduler. Press Yes when presented with the dialog which asks whether to kill the processes associated with the scheduler.

Exit NightSim by selecting the Exit menu item from the File menu. A dialog asking whether or not to save changes to **`nsim.config`** may appear; if so, press No.

You may also wish to clear the shielding attributes for CPU 0 and return the system to its previous state using NightTune.

Exit NightTune by selecting the Exit from the File menu.

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**
- **function.c**
- **report.c**
- **set_workload.c**
- **set_rate.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>

static void * heap_thread (void * ptr);

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;
}

static
void *
sine_thread (void * ptr)
{
   control_t * data = (control_t *)ptr;
   struct sembuf wait = {0, -1, 0};

   trace_open_thread (data->name);

   for (;;) {
      semop(sema, &wait, 1);
      data->count++;
      data->angle += data->delta;
      data->value = sin(data->angle);
   }
}
```

```
static
void *
cosine_thread (void * ptr)
{
   control_t * data = (control_t *)ptr;
   struct sembuf wait = {0, -1, 0};

   trace_open_thread (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};

   trace_begin ("/tmp/data",NULL);
   trace_open_thread ("main");

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

   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);
      if (state != hold) semop(sema,&trigger,1);
   }

   trace_end () ;
}

void * ptrs[5];

static
void *
heap_thread (void * unused)
{
```

```
   int i;
   int scenario = -1;
   void * ptr;
   int * * iptr;

   extern void * alloc_ptr (int size, int swtch);
   extern void free_ptr (void * ptr, int swtch);

   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'll 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);
}
```

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);}

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("wave",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]);
}
```

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);}

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]);
}
```