Concurrent Fortran 95 Tutorial

CONCURRENT COMPUTER CORPORATION™

0890498-000

February 2002

Revision History:	Level:	Effective With:
Original Release -- February 2002	000	PowerMAX OS 4.3

Preface

General Information

Concurrent Fortran 95 utilizes the Numerical Algorithms Group's F95 compiler and Concurrent's C/C++ compiler to produce highly optimized object code tailored to Concurrent systems running PowerMAX OS™.

Scope of Manual

This manual is a tutorial for Concurrent Fortran 95. In this tutorial, we will compile and link a Fortran program and then document its usage with the NightView™ symbolic debugger, the NightSim™ frequency-based scheduler, and the NightTrace™ event analyzer.

Structure of Manual

This manual consists of one chapter which is the tutorial for Concurrent Fortran 95.

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:

0890395	*NightView User's Guide*
0890398	*NightTrace Manual*
0890458	*NightSim User's Guide*

Contents

1 Using Concurrent Fortran 95 with NightStar Tools

Concurrent Fortran 95 compiles Fortran source using C as its intermediate language. The Fortran source is first translated to its equivalent in C and that resultant C code is then compiled using the Concurrent C/C++ compiler.

Because of this, certain considerations must be taken into account. In the generated C code, an underscore ("_") is appended to the names of all variables and function calls. This must be taken into consideration when using any of the NightStar tools which reference the variables or function names in the Fortran source.

This tutorial will demonstrate the interaction of a Fortran program with the various NightStar tools including the NightView™ symbolic debugger, NightTrace™ event analyzer, and NightSim™ frequency-based scheduler.

Overview

This is a demonstration of the Concurrent Fortran 95 compiler and its interactions with various NightStar tools, including:

- NEdit
- NightSim
- NightView
- NightTrace

integrating them together into one cohesive example.

Please see "Before you begin" on page 1-1 for some important recommendations and considerations.

Before you begin

For the sections of the tutorial that use the NightSim Scheduler and the NightView Source-Level Debugger, this tutorial requires that the user have the following privileges:

- P_CPUBIAS
- P_PLOCK
- P_RTIME

A convenient way to associate privileges with users is through the use of roles. A role is simply a named description of a set of privileges that have been registered for certain executable files, such as the shell. The system administrator creates roles and assigns users to them. During the login process, users can request that their shell be granted the privileges associated with their role. Such a request takes the form of an invocation of the **`tfadmin(1M)`** command. Once privileges have been granted to the user's shell, subsequently spawned processes automatically inherit those privileges.

The following commands create a role and register all the privileges required by this tutorial to three commonly used shells (**`sh`**, **`ksh`**, and **`csh`**). The PowerMAX OS system administrator should issue the following commands once.

```
/usr/bin/adminrole -n NSTAR_USERS
/usr/bin/adminrole -a sh:/usr/bin/sh:cpubias:plock:rtime NSTAR_USERS
/usr/bin/adminrole -a ksh:/usr/bin/ksh:cpubias:plock:rtime NSTAR_USERS
/usr/bin/adminrole -a csh:/usr/bin/csh:cpubias:plock:rtime NSTAR_USERS
```

The following command assigns an example user (JoeUser) to the NSTAR_USERS role. The system administrator should issue the following command once.

```
/usr/bin/adminuser -n -o NSTAR_USERS JoeUser
```

JoeUser is now allowed to request that the above privileges be granted to his shell (assuming JoeUser utilizes either the **`sh`**, **`ksh`**, or **`csh`** shell, as these are the only shell commands registered in the NSTAR_USERS role). However, by default, these privileges are not granted. He must explicitly make the request by initiating a new shell with the **`tfadmin(1M)`** command. For convenience, it is recommended that the following command be added to the end of his **`.profile`** (or **`.login`** for csh users) file. (This file is executed during initialization of the login shell).

```
exec /sbin/tfadmin NSTAR_USERS: shell
```

where *shell* is the shell of your choice (**`sh`**, **`ksh`**, or **`csh`**).

Proceed to “Getting Started” on page 1-3 to begin the tutorial.

Getting Started

We will start by creating a directory in which we will do all our work.

To create a working directory

- 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
  ```

Using NEdit

Next, we will create one of the source files that will be used by our example program. We will do this using the NEdit Editor. Although other editors may be used, NEdit comes with PowerMAX OS and thus will be demonstrated in this tutorial.

Let’s open the NEdit editor.

To start NEdit

- From the command line in a terminal window, type the following command:

```
nedit
```

The NEdit Editor will be opened, ready to accept input.

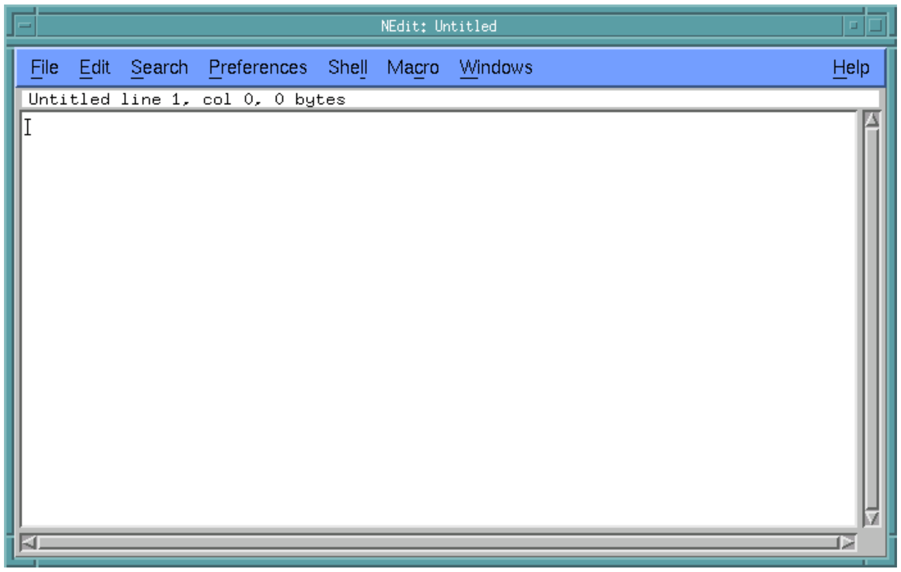


Figure 1-1. NEdit Editor

We will enter the source file for our example program. This program is written in Fortran and is shown on the following pages:

```
MODULE do_work_module

   REAL , DIMENSION(:), ALLOCATABLE :: results

CONTAINS

   SUBROUTINE do_work(iteration_count)

   INTEGER i
   REAL , POINTER :: real_ptr => NULL()

      ALLOCATE(real_ptr)
      real_ptr = iteration_count * 2.549
      DO i = 1, 500
         ALLOCATE(results(i))
         DO j = 1, i
         results(j) = i * real_ptr
         END DO
         DEALLOCATE(results)
      END DO
      DEALLOCATE(real_ptr)

   RETURN
   END SUBROUTINE do_work

END MODULE do_work_module


MODULE tracing_module

CONTAINS

   SUBROUTINE start_tracing

      !  trace_start() takes a trace-event file name as an argument.
      !  The ntraceud daemon writes the trace events logged by the
      !  NightTrace library to the trace-event file. The trace_start()
      !  routine must be called by a process to attach to the shared
      !  memory buffer used by the NightTrace library and the ntraceud
      !  daemon.

      !  trace_open() opens the current thread of execution for
      !  tracing. This routine is required for NightTrace to identify
      !  the process logging the trace events.


      INTEGER trace_start, rc_trace_start
      INTEGER trace_open_thread, rc_trace_open_thread

      rc_trace_start = trace_start(“prog.trace.data”)
      rc_trace_open_thread = trace_open_thread(“abc”)

   RETURN
   END SUBROUTINE start_tracing

   SUBROUTINE end_tracing

      !  The  trace_close_thread()  routine  is  used  to  close  the
      !  currently  running  thread and disable it from logging trace
      !  events.

      !  The trace_end() routine disables the  trace  mechanism,
         !  detaches the shared memory buffer, and frees all resources
         !  allocated for tracing.


         INTEGER trace_close_thread, rc_trace_close_thread
         INTEGER trace_end, rc_trace_end

         rc_trace_close_thread = trace_close_thread()
         rc_trace_end = trace_end()

      RETURN
      END SUBROUTINE end_tracing

   END MODULE tracing_module


   PROGRAM prog

      USE do_work_module
      USE tracing_module

      INTEGER istat
      INTEGER i

         i = 0
         CALL start_tracing            ! contained in the tracing_module
         CALL fbswait(istat)
         DO WHILE (istat .GE. 0)
            CALL do_work(i)            ! contained in the do_work_module
            CALL fbswait(istat)
            i = i + 1
         END DO
         CALL end_tracing              ! contained in the tracing_module

   END PROGRAM prog
```

This program utilizes the `fbswait` service. `fbswait` causes the calling process to go to sleep. The process will be awakened by a frequency-based scheduler at the process's scheduled frequency. At that point, it will enter the loop. The subroutine `do_work` will do some calculations. When `do_work` returns from its processing, the program will encounter another `fbswait` call which will cause the program to sleep until the frequency-based scheduler allows it continue.

To save an untitled file using the NEdit Editor

- Select Save from the File menu. This will open a file dialog.
- Ensure the Directory is the same as the one you created in “Getting Started” on page 1-3.
- Enter the name **`prog.f95`** in the Save File As field.
- Press OK.

Now that we have saved the file, we may exit our NEdit session.

To exit NEdit

- Select Exit from the File menu.

Using the Concurrent Fortran 95 compiler

Concurrent Fortran 95 utilizes the Numerical Algorithms Group's F95 compiler and Concurrent's C/C++ compiler to produce highly optimized object code tailored to Concurrent systems running PowerMAX OS^{TM}.

To compile the Fortran program

- Open a terminal window and position yourself in the working directory you created in "Getting Started" on page 1-3.
- Execute the following command:

```
f95 -g -o prog prog.f95 -lntrace -lud -lF77rt
```

In order to debug the program using the NightView Source Level Debugger, we need to compile the program with debug information so we specify the **-g** compile option.

We specify the name of the resultant output file using the **-o** compile option (in this example, our executable will be named **prog**).

In order to generate trace data when we run the program and then subsequently analyze it using the NightTrace Analyzer, we specify the compile options:

```
-lntrace -lud -lF77rt
```

At this point, we have a directory, **tutorial**, that has within it a Fortran executable, **prog**, and its corresponding source file, **prog.f95**. Full debug information will be generated for the program and tracing functionality has been included so that we may gather tracing data for later analysis.

Viewing the intermediate C code

Concurrent Fortran 95 compiles Fortran source using C as its intermediate language. The Fortran source is first translated to its equivalent in C and that resultant C code is then compiled using the Concurrent C/C++ compiler.

This intermediate source can be viewed by using the **-S** compile option to **f95**. For instance,

```
f95 -S prog.f95
```

will generate a file named **prog.c** which consists of the Fortran source translated to C. (References to the Fortran source appear throughout the C code.)

Because of this, certain considerations must be taken into account. In the generated C code, an underscore ("_") is appended to the names of all variables and function calls. This must be taken into consideration when using any of the NightStar tools which reference the variables or function names in the Fortran source. Some of these points will be addressed in the following sections.

To view the intermediate C code

- Open a terminal window and position yourself in the working directory you created in “Getting Started” on page 1-3.
- Execute the following command:

  ```
  f95 -S prog.f95
  ```

  This will generate a file named **`prog.c`**. The following code fragment shows a portion of that file:

```
# line 1 "prog.f95"
#include <f95.h>
typedef struct { Real *addr; Integer3 offset; Triplet dim[1]; }
AAType1;
# line 1 "prog.f95"
AAType1 do_work_module_MP_results;
extern void do_work_module_MP_do_work();
# line 26 "prog.f95"
# line 7 "prog.f95"
void do_work_module_MP_do_work(iteration_count_)
     Integer *iteration_count_;
{
Integer Tmp1;
Integer Tmp2;
Integer Tmp3;
register Integer j_;
register Integer i_;
# line 7 "prog.f95"
static Real *real_ptr_ = (Real *)0;
# line 12 "prog.f95"
real_ptr_ = ((Real *)__NAGf90_Allocate_s(4,(Integer *)0));
# line 12 "prog.f95"
;
# line 13 "prog.f95"
 *real_ptr_ =  *iteration_count_*2.549000025e+00f;
# line 14 "prog.f95"
for(i_ = 1;i_ <= 500;i_++) {
# line 15 "prog.f95"
if (do_work_module_MP_results.addr)
__NAGf90_already_allocated("RESULTS");
# line 15 "prog.f95"
do_work_module_MP_results.offset = 0;
# line 15 "prog.f95"
do_work_module_MP_results.dim[0].lower = 1;
# line 15 "prog.f95"
Tmp1 = i_;
# line 15 "prog.f95"
if (Tmp1<0) Tmp1 = 0;
# line 15 "prog.f95"
do_work_module_MP_results.dim[0].extent = Tmp1;
# line 15 "prog.f95"
do_work_module_MP_results.dim[0].mult = 1;
# line 15 "prog.f95"
```

Using NightSim

NightSim is a tool for scheduling and monitoring real-time applications which require predictable, repetitive process execution. NightSim provides a graphical interface to the PowerMAX OS frequency-based scheduler and performance monitor. With NightSim, application builders can control and dynamically adjust the periodic execution of multiple coordinated processes, their priorities, and their CPU assignments. NightSim's performance monitor tracks the CPU utilization of individual processes and provides a customizable display of period times, minimums, maximums, and frame overruns. For more information on NightSim, refer to the *NightSim User's Guide* (0890480).

Invoking NightSim

Because our program uses the frequency-based scheduler, we will use the NightSim Scheduler to schedule the process.

To invoke the NightSim Scheduler

- From the command line, type the following command:

  ```
  nsim &
  ```

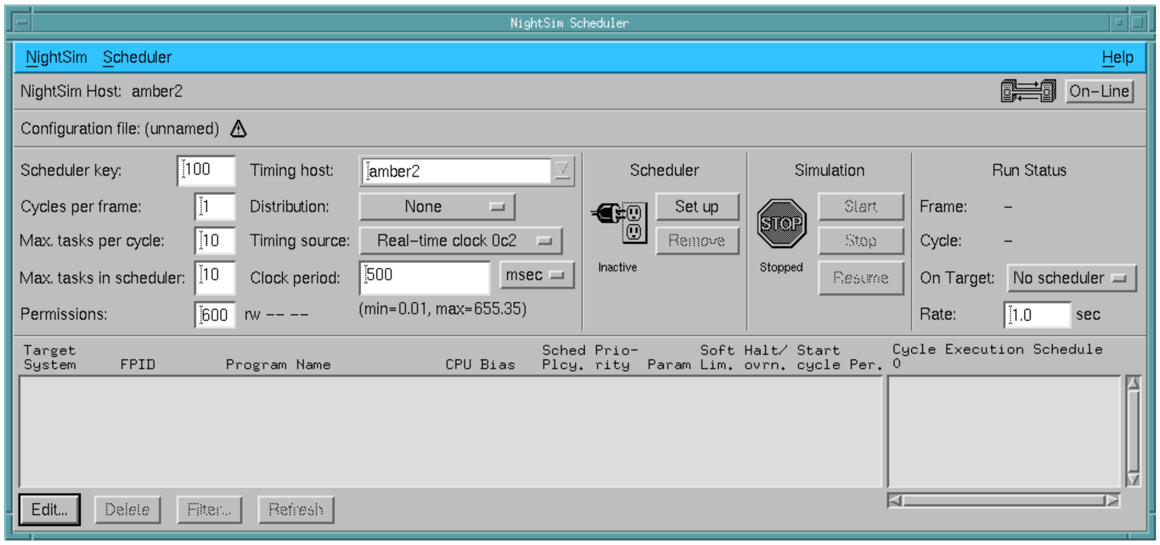
  The NightSim Scheduler will be opened, ready to be configured.

NOTE

We specify the **&** so that the NightSim session runs in the background.

Configuring the Scheduler

The NightSim Scheduler window is opened, ready for us to configure it for our particular simulation.



NightSim Host: amber2

Configuration file: (unnamed)

Scheduler key: 100
Timing host: amber2
Cycles per frame: 1
Distribution: None
Max. tasks per cycle: 10
Timing source: Real-time clock 0c2
Max. tasks in scheduler: 10
Clock period: 500 msec
Permissions: 600 rw -- --
(min=0.01, max=655.35)

Scheduler: Inactive

Simulation: Stopped

Run Status
Frame: –
Cycle: –
On Target: No scheduler
Rate: 1.0 sec

```
Target                                                   Sched Prio-        Soft Halt/ Start     Cycle Execution Schedule
System      FPID          Program Name          CPU Bias  Plcy. rity  Param Lim. ovrn. cycle Per. 0
```

Figure 1-2. NightSim Scheduler

To configure a NightSim Scheduler

- Specify a Scheduler key. The key is a user-chosen numeric identifier with which the scheduler will be associated. For our example, we will use 100.
- Specify the Cycles per frame. This field allows you to specify the number of cycles that compose a frame on the specified scheduler. We will use the value 1.
- Specify the Max. tasks per cycle. This field allows you to specify the maximum number of processes that can be scheduled to execute during one cycle. Enter 10 for our example.
- Specify the Max. tasks in scheduler. This field allows you to specify the maximum number of processes that can be scheduled on the specified scheduler at one time. For our example, we will specify the value 10.
- Enter the name of a PowerMAX OS system which will act as the Timing host for the simulation. You may use the drop down list associated with this field for the names of systems previously used as timing hosts. For our example, we will enter amber2, a Turbo Hawk system.

NOTE

When NightSim is operating in On-Line mode, an attempt will be made to communicate with the system specified as the timing host. The user may experience a slight delay and the message Talking to Server... will appear in the Configuration File Name Area of the NightSim Scheduler as this occurs. See the *NightSim User's Guide* (0890480) for more information.

- Select a Timing source from the list provided. This list contains the set of devices available on the timing host. We will use Real-time clock 0c2.

NOTE

> Do not use Real-time clock 0c0 for the Timing source as it is typically used by system utilities and could cause unwanted effects if used. See **hrtconfig(1)** for more information

Since we are using the real-time clock on the target system, we need to specify the clock period. For our simulation, we would like the real-time clock to “fire” every .5 seconds (or 500 milliseconds).

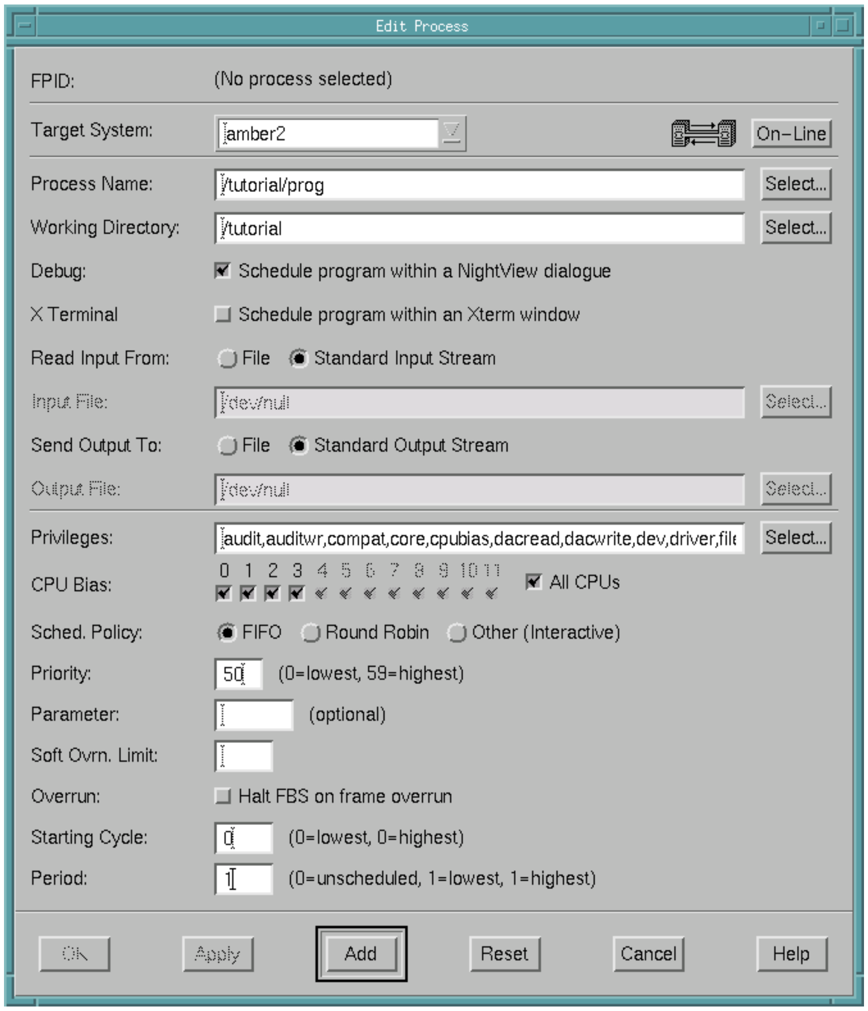
IMPORTANT

> The following steps should be performed in the order presented below to ensure the correct value for the clock period.

- Choose the msec from the drop-down list next to the Clock period field.
- Specify Clock period. For our example, we will specify 500 for the number of milliseconds.

Scheduling a process

Once we have properly configured the Scheduler, we can add a process to the frequency-based scheduler.



Edit Process

FPID:	(No process selected)	
Target System:	amber2	On–Line
Process Name:	/tutorial/prog	Select...
Working Directory:	/tutorial	Select...
Debug:	[✓ Schedule program within a NightView dialogue]	
X Terminal	Schedule program within an Xterm window	
Read Input From:	File [✓ Standard Input Stream]	
Input File:	/dev/null	Select...
Send Output To:	File [✓ Standard Output Stream]	
Output File:	/dev/null	Select...
Privileges:	audit,auditwr,compat,core,cpubias,dacread,dacwrite,dev,driver,file	Select...
CPU Bias:	0 1 2 3 4 5 6 7 8 9 10 11 (✓ 0 1 2 3) [✓ All CPUs]	
Sched. Policy:	[✓ FIFO] Round Robin Other (Interactive)	
Priority:	50 (0=lowest, 59=highest)	
Parameter:	(optional)	
Soft Ovrn. Limit:		
Overrun:	Halt FBS on frame overrun	
Starting Cycle:	0 (0=lowest, 0=highest)	
Period:	1 (0=unscheduled, 1=lowest, 1=highest)	

Figure 1-3. NightSim Edit Process

To add a process to the frequency-based scheduler

- Press the Edit… button on the NightSim Scheduler window. This will bring up the Edit Process window.
- Press the Select… button next to the Process Name field. This brings up the Select a Program dialog.
    - Either type the full pathname to your working directory, **`tutorial`**, in the Directory field, or maneuver to that directory using the items in the Directories list.

    - Choose the program we wish to schedule from the Files list. For our example, we will select prog from the list.
    - Press OK to select the program.

- Ensure that the Working Directory is the same directory that contains our program (the directory of the Process Name selected in the previous step).
- Check the Schedule program within a NightView dialogue checkbox. This will bring the program up in the NightView debugger before the program executes, allowing us to set *tracepoints* so that we may generate trace data when the program executes.
- Specify the Priority for this process. The range of priority values that you can enter is governed by the scheduling policy specified. NightSim displays the range of priority values that you can enter next to the Priority field. Higher numerical values correspond to more favorable scheduling priorities. For our example, we will give the process a priority of 50.
- Select Starting Cycle. This field allows you to specify the first minor cycle in which the specified program is to be wakened in each major frame. We will choose the lowest value, 0, for our example.
- Select Period. This field allows you to establish the frequency with which the specified program is to be wakened in each major frame. Enter the number of minor cycles representing the frequency with which you wish the program to be wakened. For our example, we will specify a period of 1, indicating that the specified program is to be wakened every minor cycle.
- Press Add to add the process to the frequency-based scheduler.
- Press the Close button to dismiss the Edit Process window.

Activating user tracing and kernel tracing

At this point in the tutorial, we are about to create the scheduler configured according to the parameters we just specified and allow the program to run. However, we would like to generate trace data from this program while it is running so we need to start the NightTrace user daemon to log user trace events as well as KernelTrace which will collect data about the execution time of interrupts, exceptions, system calls, context switches, and I/O to various devices.

To activate the NightTrace user daemon

- Open a terminal window and position yourself in the working directory you created in “Getting Started” on page 1-3.

IMPORTANT

It is essential that you are positioned in the working directory that is associated with the user program being scheduled with NightSim. The NightTrace user daemon will communicate with the user program based on the file argument supplied in the next step.

- Invoke the NightTrace user daemon. We issue the **`ntraceud`** command which takes as an argument the name of a file in which to save the trace data. This file should be named *program_name*`.trace.data`, where *program_name* is the name of the program generating the trace data.

NOTE

By default, **`ntraceud`** requires write access to system SPL devices, e.g. **`/dev/spl`**, **`/dev/spl1`**, etc. On most systems, these devices are only writeable by the root user; therefore, you should run the **`ntraceud`** command as root.

However, since the use of SPL devices is not strictly necessary for tracing single-threaded user applications (although, for optimal real-time performance it is recommended), the **`-ipldisable`** option to **`ntraceud`** is acceptable.

Since the application in this tutorial is single-threaded, you may use the **`-ipldisable`** option as indicated below.

For our example, we will issue the following command:

```
ntraceud -ipldisable prog.trace.data
```

Now we can activate kernel tracing.

To activate kernel tracing

- Open a terminal window and position yourself in the working directory you created in “Getting Started” on page 1-3.
- Invoke the KernelTrace utility. We issue the **`ktrace`** command which can take a number of arguments.

NOTE

The KernelTrace utility requires root access in order to run.

We will use the **`-o`** option which specifies the name of a file in which to save the kernel trace data.

When generating kernel trace data, the resultant file can grow extremely large very quickly. In order to circumvent any problems that may arise from the output file growing extremely large, we will use the **`-bufferwrap`** option which limits the size of the output file. Specifying a value of 50 to this option will limit the size of the resulting output file to a little over 2 megabytes.

NOTE

Due to a problem with the **`-bufferwrap`** option, user and kernel data may not appear synchronized when viewing the trace data in subsequent steps. This problem has been fixed in the **`ktrace`** and **`ntfilter`** commands in PowerMAX OS 4.3 Patch Set 6 (**`trace-004`** and **`base-006`**). If these packages are not installed on your system, you may omit the **`-bufferwrap`** option. However, be aware that the kernel trace file may grow extremely large in a short period of time.

So, for our example, we will issue the following command, as the root user:

```
ktrace -bufferwrap 50 -o prog.ktrace.data
```

You should see output similar to the following:

```
locking into memory
setting priority to RT 59
open /dev/trace
initialize
set trace event time stamp source to Motorola Time Base
Register
gather trace point data
```

Setting up the scheduler

To set up the scheduler

- In the NightSim Scheduler window, press the `Set up` button.

  This action:

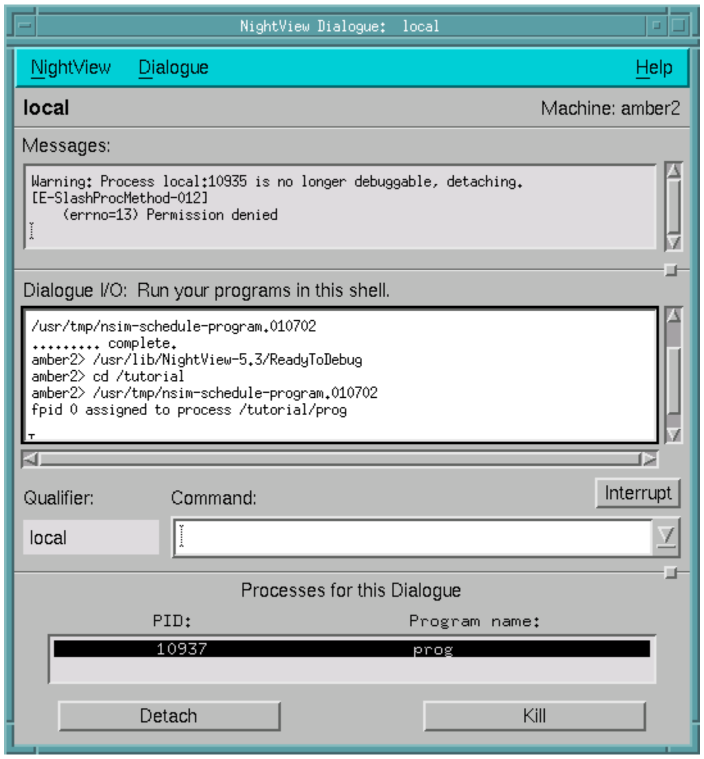
  - creates a scheduler that is configured according to the parameters we specified
  - schedules the processes that we have added to the NightSim Scheduler window and starts them running up to the first `fbswait` call, and
  - attaches the timing source to the scheduler.

Because we have specified the Schedule program within a NightView dialogue option when we added this process to the frequency-based scheduler (see “To add a process to the frequency-based scheduler” on page 1-12), the NightView Source Level Debugger will be started.

Using NightView

NightView is a graphical source-level debugging and monitoring tool specifically designed for real-time applications. NightView can monitor, debug, and patch multiple real-time processes running on multiple processors with minimal intrusion. In addition to standard debugging capabilities, NightView supports application-speed eventpoint conditions, hot patches, synchronized data monitoring, exception handling and loadable modules.

Because we have specified the Schedule program within a NightView dialogue option when we added this process to the frequency-based scheduler (see “To add a process to the frequency-based scheduler” on page 1-12), we are presented with a NightView Dialogue Window as well as a Principal Debug Window with the execution of the program stopped.



```
Warning: Process local:10935 is no longer debuggable, detaching.
[E-SlashProcMethod-012]
    (errno=13) Permission denied
```

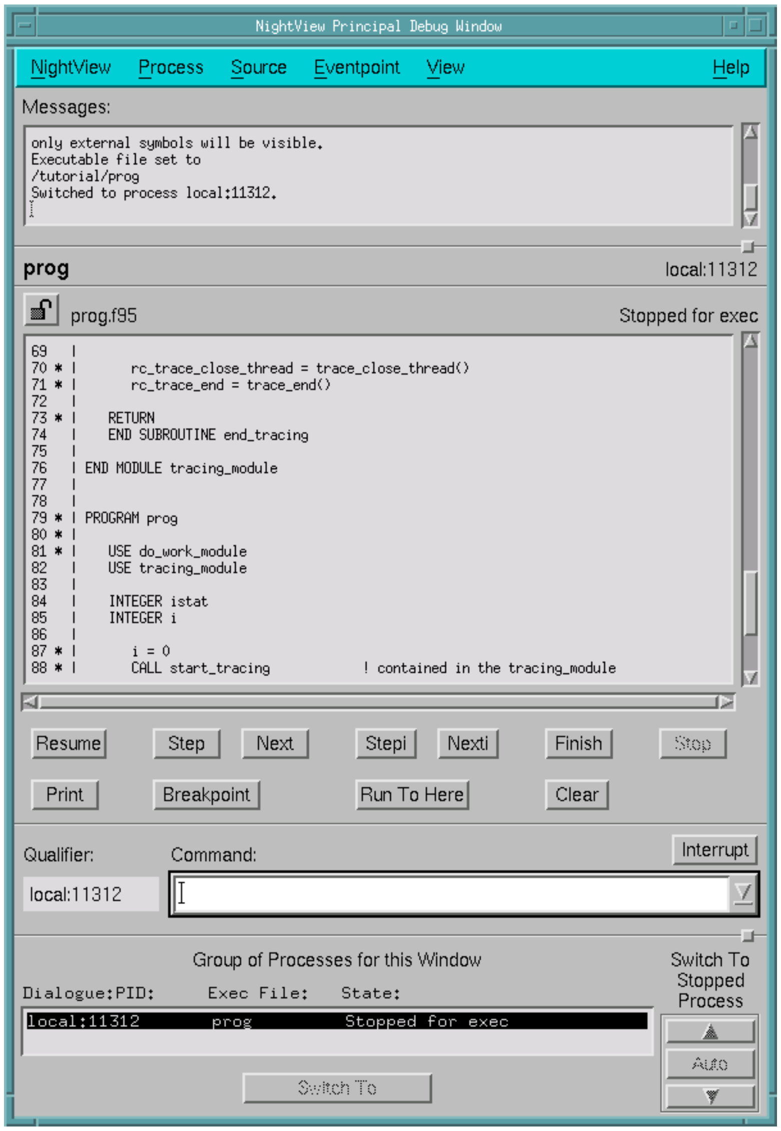
```
/usr/tmp/nsim-schedule-program.010702
......... complete.
amber2> /usr/lib/NightView-5.3/ReadyToDebug
amber2> cd /tutorial
amber2> /usr/tmp/nsim-schedule-program.010702
fpid 0 assigned to process /tutorial/prog
```

Figure 1-4. NightView Dialogue

During initialization, you will see a message similar to the following:

```
Warning: Process local:11749 is no longer debuggable,
detaching.
[E-SlashProcMethod-012]
    (errno=13) Permission denied
```

This is an anomaly caused by an intermediate process which schedules the user program. You may ignore this warning.



```
only external symbols will be visible.
Executable file set to
/tutorial/prog
Switched to process local:11312.
```

```

      rc_trace_close_thread = trace_close_thread()
      rc_trace_end = trace_end()

   RETURN
   END SUBROUTINE end_tracing

END MODULE tracing_module


PROGRAM prog

   USE do_work_module
   USE tracing_module

   INTEGER istat
   INTEGER i

      i = 0
      CALL start_tracing            ! contained in the tracing_module
```

```
Dialogue:PID:     Exec File:    State:
local:11312       prog          Stopped for exec
```

Figure 1-5. NightView Principal Debug Window

Adding a tracepoint in the program

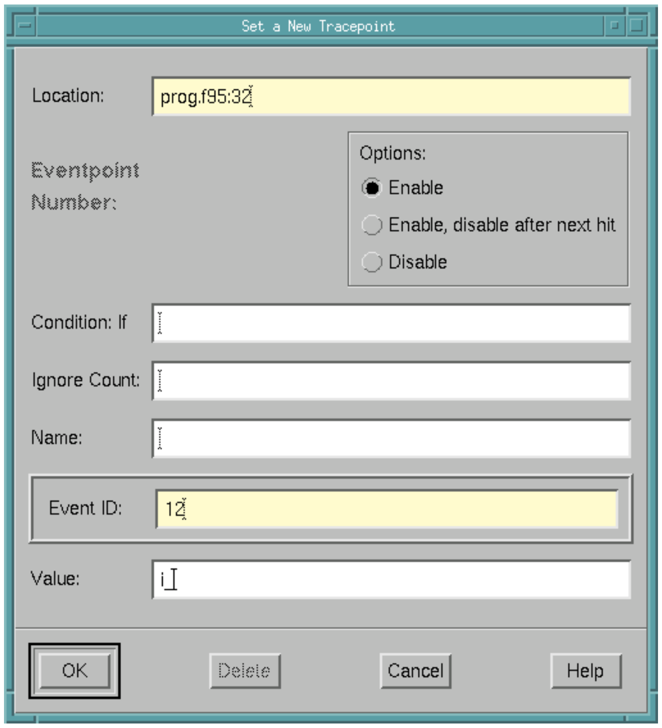
Since we would like to generate user trace data, but did not place any calls within the code before our program was compiled, we can use NightView to insert a *tracepoint* in the code. A tracepoint is a call to one of the `ntrace(3X)` library routines for recording the time when execution reached the tracepoint.

To add a tracepoint in a program

- In the NightView Principal Debug Window, click on the line:

```
CALL do_work(i)
```

- Select `Set Tracepoint...` from the `Eventpoint` menu. This will open the `Set a New Tracepoint` dialog.



Location: prog.f95:32
Eventpoint Number:
Options: [✓ Enable] Enable, disable after next hit Disable
Condition: If
Ignore Count:
Name:
Event ID: 12
Value: i_

Figure 1-6. Setting a new tracepoint

- Enter the `12` for the `Event ID`. Each trace event has a user-defined trace event ID. This ID is a valid integer in the range reserved for user trace events (0-4095, inclusive). We have chosen 12 for this example.

- Enter `i_` in the `Value` field. This will log the value of the variable `i` as `arg1` in the trace file every time this tracepoint is encountered.

IMPORTANT

Note the underscore appended to the name of the Fortran variable `i`. When debugging a Concurrent Fortran 95 program, the Fortran source (not the generated C code) will appear in the NightView Source-Level Debugger. However, NightView uses the generated C code as its underlying source for debugging. As such, an underscore ("_") must be appended to variables or function names that are referenced. See “Viewing the intermediate C code” on page 1-7 for more information.

- Press `OK`.

NOTE

You may have also entered the following command in the `Command` field of the NightView Principal Debug Window:

`tracepoint 12 at` *line_number* **`value=i_`**

where *line_number* coincides with the line:

```
CALL do_work(i)
```

See **`tracepoint`** for details on the use of this command.

Inserting a monitorpoint

NightView allows the use of *monitorpoints* while debugging a process. Monitorpoints allow you to monitor the value of one or more variables without interrupting the execution of your program.

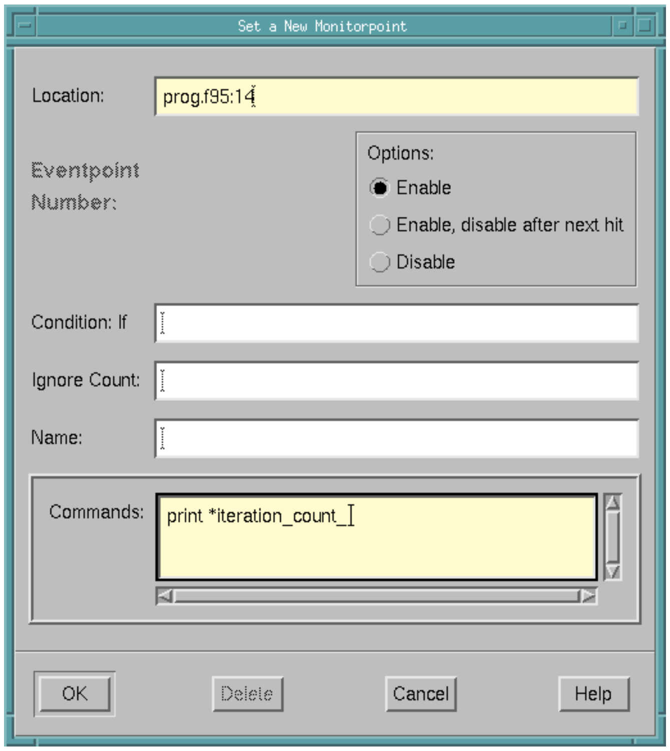
In our example, we will insert a monitorpoint in the `do_work` subroutine contained in the `do_work_module`.

To insert a monitorpoint in a program

- In the NightView Principal Debug Window, click on the line:

  ```
  real_ptr = iteration_count * 2.549
  ```

  which appears in the `do_work` subroutine in the `do_work_module`.
- Select `Set Monitorpoint…` from the `Eventpoint` menu. This will open the `Set a New Monitorpoint` dialog.



Location: prog.f95:14

Options: [✓ Enable] Enable, disable after next hit Disable

Condition: If

Ignore Count:

Name:

Commands: print *iteration_count_

Figure 1-7. Setting a new monitorpoint

- Enter the expression:

  ```
  print *iteration_count_
  ```

  in the `Commands` field.

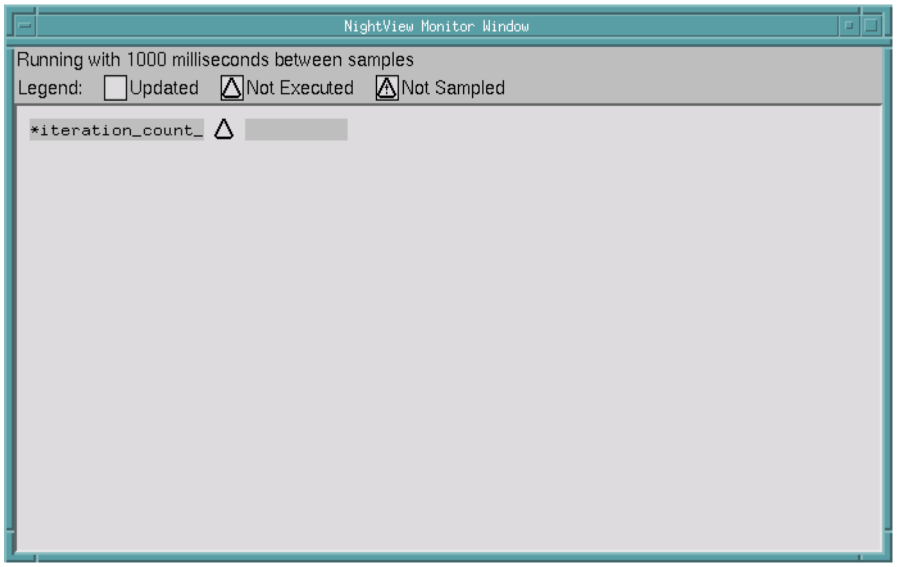
IMPORTANT

Note the underscore appended to the name of the Fortran variable `iteration_count`. When debugging a Concurrent Fortran 95 program, the Fortran source (not the generated C code) will appear in the NightView Source-Level Debugger. However, NightView uses the generated C code as its underlying source for debugging. As such, an underscore ("_") must be appended to variables or function names that are referenced. See “Viewing the intermediate C code” on page 1-7 for more information.

Also, because arguments to Fortran functions and subroutines are passed by reference, `iteration_count` is actually a pointer (see “Viewing the intermediate C code” on page 1-7). As such, we must prepend a `*` to `iteration_count` to access the value of the variable at the memory address stored in `iteration_count`.

- Press OK.

This will open a NightView Monitor Window which will display the value of `iteration_count` while the program is running.



Running with 1000 milliseconds between samples

*iteration_count_

Figure 1-8. NightView Monitor Window

NOTE

You may have also entered the following commands in the Command field of the NightView Principal Debug Window:

`monitorpoint at` *line_number*
`print *iteration_count_`
`end monitor`

where *line_number* coincides with the line:

```
real_ptr = iteration_count * 2.549
```

See **`monitorpoint`** for details on the use of this command.

Resuming execution

Now it's time to let the program run and generate some trace data from the tracepoint we just entered.

To resume execution in NightView

- Press the Resume button in the NightView Principal Debug Window.

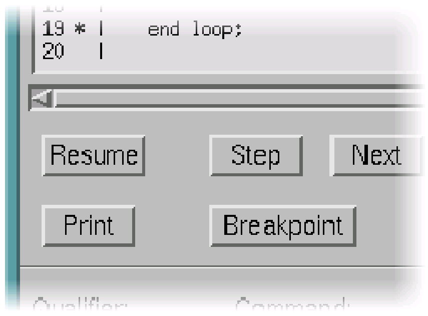

```
19 * |    end loop;
20   |
```

Figure 1-9. Resuming execution

Starting the simulation

Now we need to go back to our NightSim Scheduler window and start the simulation. When you click on the Start button, NightSim carries out the following actions:

- Attaches the timing source to the scheduler if not already attached or if the timing source has been changed
- If a real-time clock is being used as the timing source, sets the clock period in accordance with the value entered in the Clock period field in the Scheduler Configuration Area
- Starts the simulation with the values of the *minor cycle*, *major frame*, and *overrun* counts set to zero

To start a simulation in NightSim

- Press the Start button on the NightSim Scheduler window.

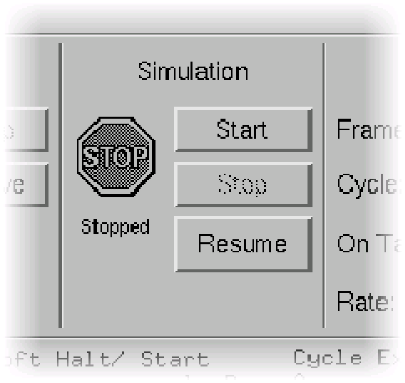


Figure 1-10. Starting the simulation

Once the simulation is started, note the value of `iteration_count` incrementing in the NightView Monitor Window. See “Inserting a monitorpoint” on page 1-20 for details.

Inserting a patchpoint

NightView allows the use of *patchpoints* while debugging a process. Patchpoints are locations in the debugged process where a *patch*, usually an expression that alters the behavior of the process, is inserted.

In our example, we will insert a patchpoint in the loop in program `prog` to change the value of the `istat` variable in order to exit the loop:

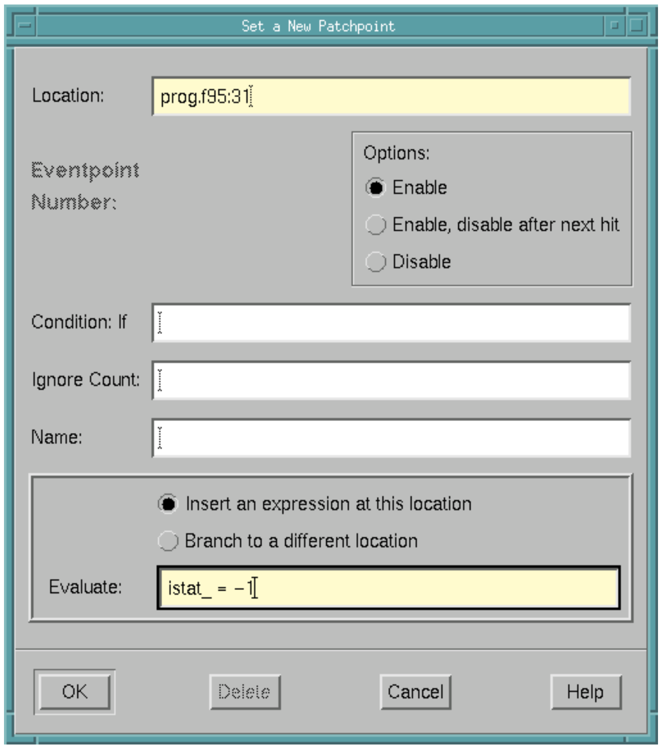
```
DO WHILE (istat .GE. 0)
   CALL do_work(i)            ! contained in the do_work_module
   CALL fbswait(istat)
   i = i + 1
END DO
```

To insert a patchpoint in a program

- In the NightView Principal Debug Window, click on the line:

  ```
  DO WHILE (istat .GE. 0)
  ```

- Select `Set Patchpoint...` from the `Eventpoint` menu. This will open the `Set a New Patchpoint` dialog.



Location: prog.f95:31

Eventpoint Number:

Options:
[✓ Enable]
Enable, disable after next hit
Disable

Condition: If

Ignore Count:

Name:

[✓ Insert an expression at this location]
Branch to a different location

Evaluate: istat_ = −1

Figure 1-11. Setting a new patchpoint

- Enter the expression:

  ```
  istat_ = -1
  ```

  in the `Evaluate` field.

IMPORTANT

> Note the underscore appended to the name of the Fortran variable `istat`. When debugging a Concurrent Fortran 95 program, the Fortran source (not the generated C code) will appear in the NightView Source-Level Debugger. However, NightView uses the generated C code as its underlying source for debugging. As such, an underscore ("_") must be appended to variables or function names that are referenced. See “Viewing the intermediate C code” on page 1-7 for more information.

- Press `OK`.

  When this patchpoint is encountered during the execution of the program, the value of the Fortran variable `istat` will be set to -1, breaking out of the loop, thereby terminating the program.

NOTE

You may have also entered the following command in the Command field of the NightView Principal Debug Window:

```
patchpoint at line_number eval istat_ = -1
```

where *line_number* coincides with the line:

```
DO WHILE (istat .GE. 0)
```

See **`patchpoint`** for details on the use of this command.

Halting user tracing and kernel tracing

Now that our program has finished, we can exit the KernelTrace utility and stop the NightTrace user daemon.

To halt kernel tracing

- In the terminal window where you invoked the KernelTrace utility (see "To activate kernel tracing" on page 1-14), press Ctrl-C.

  You should see the message:

  ```
  terminating
  ```

To halt the NightTrace user daemon

- In the terminal window where you invoked the NightTrace user daemon (see "To activate the NightTrace user daemon" on page 1-13), enter the following command:

  ```
  ntraceud -quit program_name.trace.data
  ```

  where *program_name* is the name of the program generating the trace data. So, for our example, we will issue the following command:

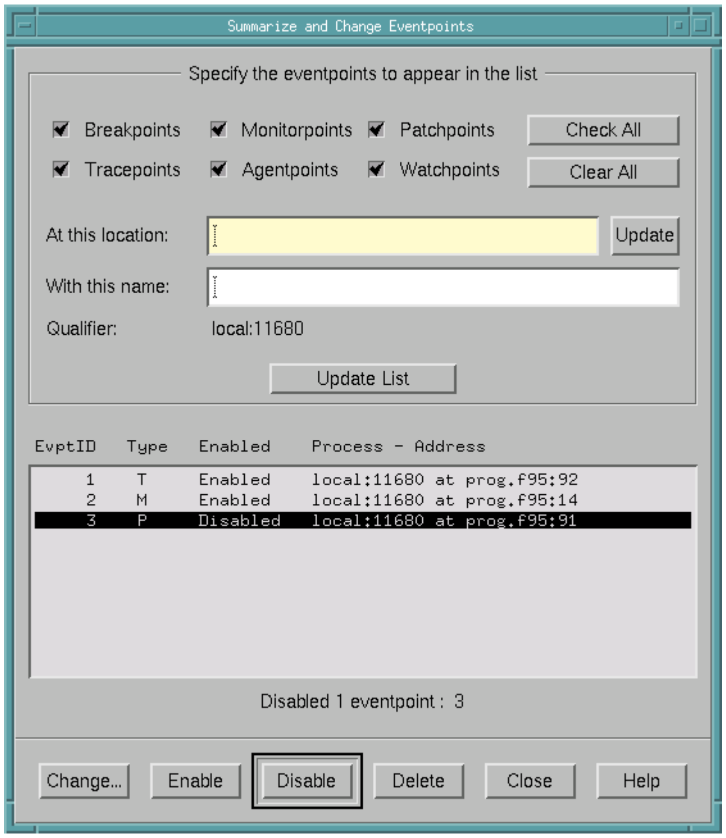
  ```
  ntraceud -quit prog.trace.data
  ```

Disabling the patchpoint

Before we exit NightView, we should disable the patchpoint that we set in "Inserting a patchpoint" on page 1-24. NightView retains knowledge of all eventpoints for a particular program in a current session and will reinitialize them if that program is re-run. If not disabled, the patchpoint in our program will be encountered immediately if our program is re-run under the current session of NightView, causing us to exit the loop and terminate the program.

To disable a patchpoint in NightView

- Select Summarize/Change… from the Eventpoint menu.
- Select the patchpoint from the list of eventpoints (listed with a P in the Type column).


Summarize and Change Eventpoints
Specify the eventpoints to appear in the list
Check All
Clear All
Update
Update List
Change...
Enable
Disable
Delete
Close
Help

✓ Breakpoints ✓ Monitorpoints ✓ Patchpoints
✓ Tracepoints ✓ Agentpoints ✓ Watchpoints

At this location:
With this name:
Qualifier: local:11680

```
EvptID  Type  Enabled   Process - Address
    1    T    Enabled   local:11680 at prog.f95:92
    2    M    Enabled   local:11680 at prog.f95:14
    3    P    Disabled  local:11680 at prog.f95:91
```

Disabled 1 eventpoint : 3

Figure 1-12. Disabling a patchpoint

- Press Disable.
- Press Close.

Exiting the program

NightView suspends the process it is debugging before it exits. We may allow the process to complete its termination by resuming its execution.

To resume execution in NightView

- Press the Resume button in the NightView Principal Debug Window.

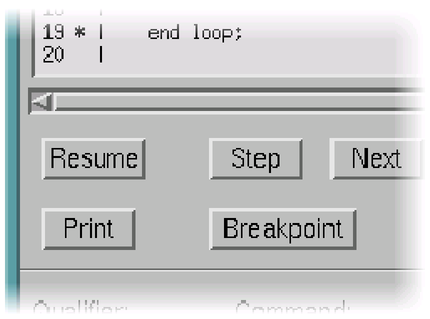


Figure 1-13. Resuming execution

Removing the scheduler

To remove the scheduler

- In the NightSim Scheduler window, press the `Remove` button.

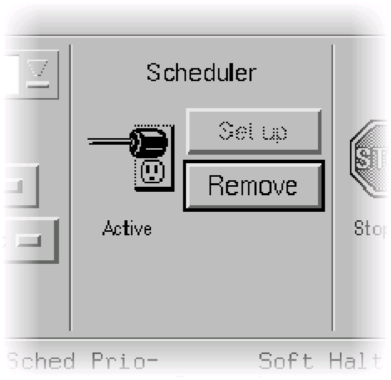
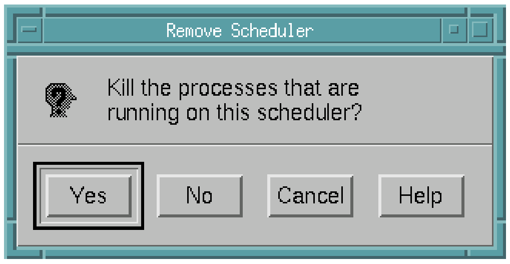


Figure 1-14. Removing the scheduler

You will be presented with the following dialog:



Kill the processes that are running on this scheduler?

Figure 1-15. Removing the scheduler

- Press `Yes` to kill the processes that are currently scheduled on the scheduler.

Using NightTrace

NightTrace is a graphical tool for analyzing the dynamic behavior of single and multiprocessor applications. NightTrace can log application data events from simultaneous processes executing on multiple CPUs or even multiple systems. NightTrace combines application events with PowerMAX OS events and presents a synchronized view of the entire system. NightTrace allows users to zoom, search, filter, summarize, and analyze events in a wide variety of ways. PowerMAX OS events include individual system calls, context switches, machine exceptions, page faults and interrupts. Application events are defined by the user allowing logging of the data items associated with each event.

We may use NightTrace to analyze the trace data that we gathered during the execution of our program but first we will need to convert the files so that they may be used by NightTrace.

Converting kernel trace event files

To convert kernel trace event files

- On the PowerMAX OS system where you invoked the KernelTrace utility (see “To activate kernel tracing” on page 1-14), enter the following command:

  ntfilter -v < *raw_kernel_file* **>** *filtered_kernel_file*

  where *raw_kernel_file* is the file we specified using the **`-o`** option to **`ktrace`** and *filtered_kernel_file* is the name of the resultant output file from **`ntfilter`**.

  So, for our example, we will issue the following command:

  ```
  ntfilter -v < prog.ktrace.data > prog.ntrace.kernel
  ```

  The converted KernelTrace trace event file will then be saved to the file **`prog.ntrace.kernel`**. The **`-v`** option creates a **`vectors`** files that will be specified to NightTrace along with the converted KernelTrace trace event file. The **`vectors`** file is generated dynamically because it is system-configuration dependent. Without a **`vectors`** file, NightTrace will not be able to display the names of the system processes, interrupts, and exceptions that occurred during kernel tracing.

  See “Converting KernelTrace Trace Event Files with ntfilter” in the *NightTrace Manual* (0890398) for more detailed information about this process.

Invoking NightTrace

Now that all our files are created and converted, we may invoke NightTrace and analyze the results.

To invoke NightTrace

- In the working directory you created in “Getting Started” on page 1-3, enter the following command

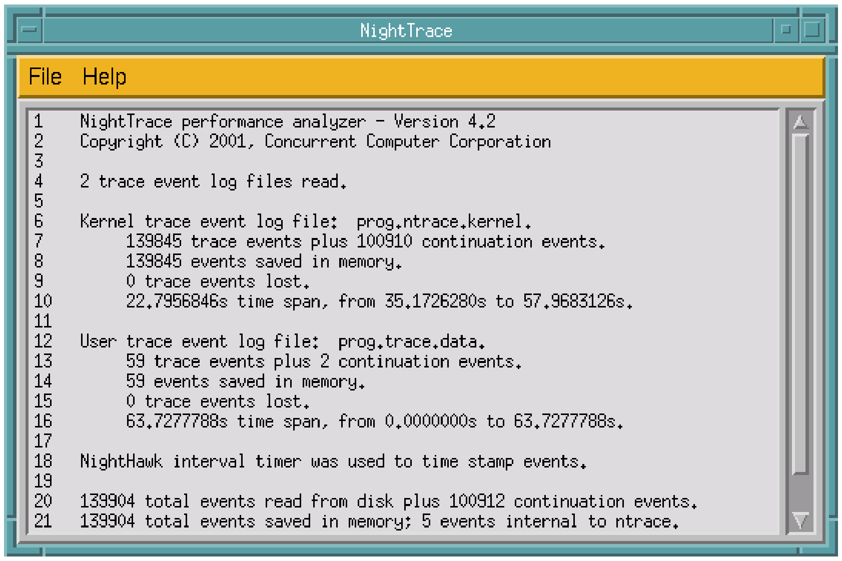
  ```
  ntrace prog.ntrace.kernel prog.trace.data vectors
  ```

  This will start the NightTrace Analyzer and pass to it:

prog.ntrace.kernel	the file created by “Converting kernel trace event files” on page 1-30
prog.trace.data	the file created by “To activate the Night-Trace user daemon” on page 1-13
vectors	a file created by “Converting kernel trace event files” on page 1-30 which allows NightTrace to display the names of the system processes, interrupts, and exceptions that occurred during kernel tracing.

  See ntrace Arguments for more information about invoking NightTrace.

  NightTrace will present the NightTrace window which is shown below:



```
1   NightTrace performance analyzer - Version 4.2
2   Copyright (C) 2001, Concurrent Computer Corporation
3
4   2 trace event log files read.
5
6   Kernel trace event log file:  prog.ntrace.kernel.
7        139845 trace events plus 100910 continuation events.
8        139845 events saved in memory.
9        0 trace events lost.
10       22.7956846s time span, from 35.1726280s to 57.9683126s.
11
12  User trace event log file:  prog.trace.data.
13       59 trace events plus 2 continuation events.
14       59 events saved in memory.
15       0 trace events lost.
16       63.7277788s time span, from 0.0000000s to 63.7277788s.
17
18  NightHawk interval timer was used to time stamp events.
19
20  139904 total events read from disk plus 100912 continuation events.
21  139904 total events saved in memory; 5 events internal to ntrace.
```

Figure 1-16. NightTrace Main window

For more information on the NightTrace window, see ntrace Global Window in the *NightTrace Manual* (0890398).

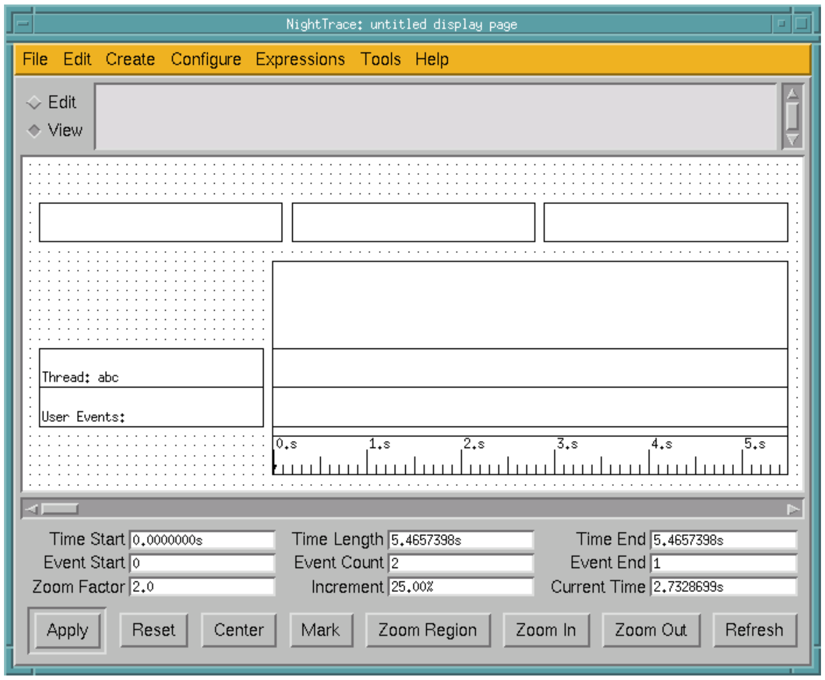
Creating a default page

In order to view our user trace events, we need to create a default page.

To create a default page

- In the NightTrace window, select Default Page from the File menu.

  This will create a Default Page as shown below:



Thread: abc
User Events:

```
Time Start   0.0000000s    Time Length  5.4657398s    Time End      5.4657398s
Event Start  0             Event Count  2             Event End     1
Zoom Factor  2.0           Increment    25.00%        Current Time  2.7328699s
```

Figure 1-17. NightTrace default page

For more information on display pages, see The Display Page in the *NightTrace Manual* (0890398).

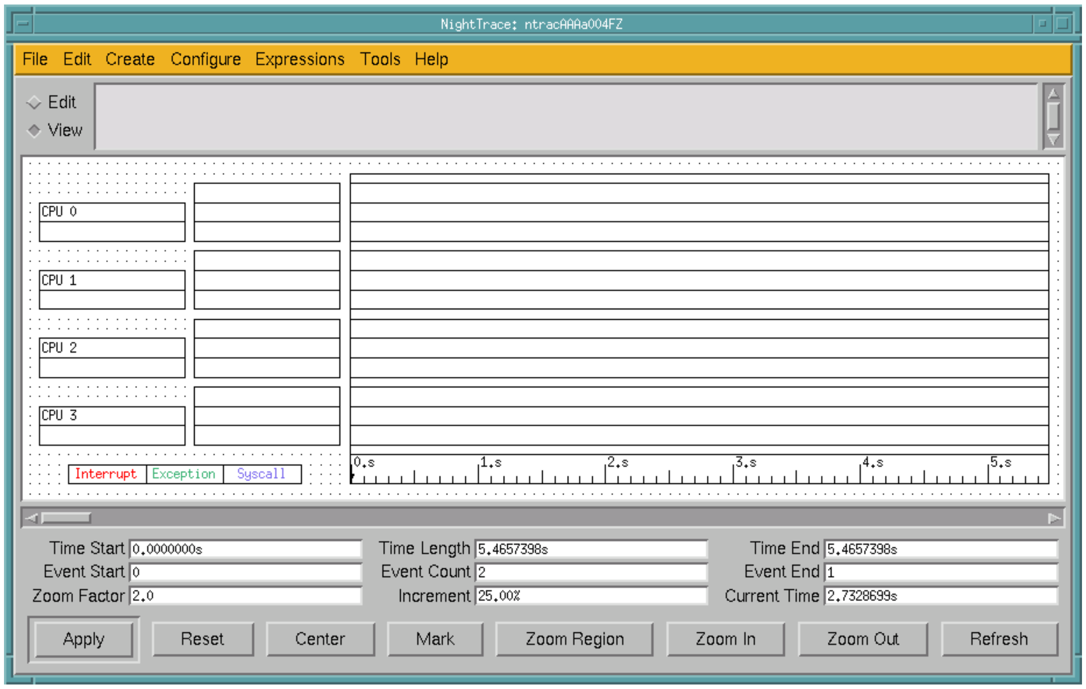
Creating a default kernel page

In order to view our kernel trace events, we need to create a default kernel page.

To create a default kernel page

- In the NightTrace window, select Default Kernel Page from the File menu.

  This will create a Default Kernel Page as shown below:



Time Start 0.0000000s
Event Start 0
Zoom Factor 2.0
Time Length 5.4657398s
Event Count 2
Increment 25.00%
Time End 5.4657398s
Event End 1
Current Time 2.7328699s

Figure 1-18. Default Kernel Page

For more information on the Default Kernel Page, see Kernel Display Pages in the *Night-Trace Manual* (0890398).

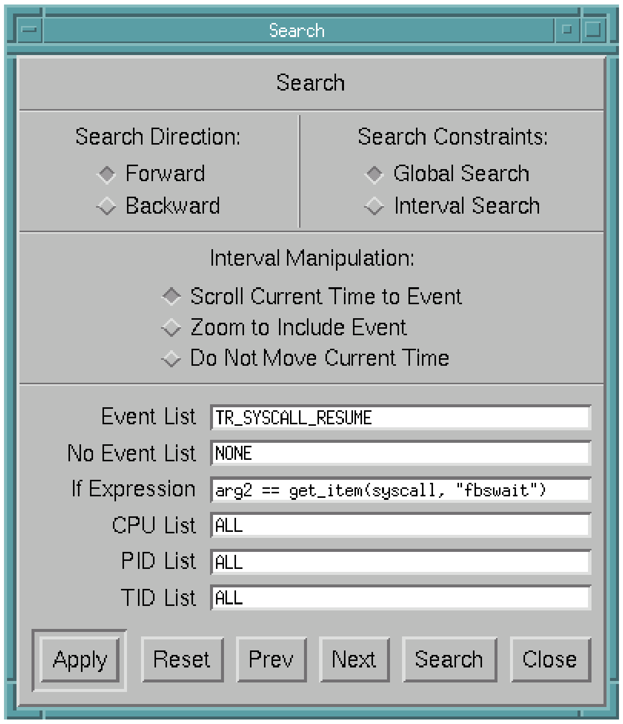
Searching for a kernel trace event

Now that we have loaded our data into NightTrace and created the appropriate display pages, we can search for the system call that corresponds to the `fbswait` call made in our program (see “Using NEdit” on page 1-4).

To search for a kernel trace event

- Select `Search...` from the `Tools` menu of the kernel display page (see “Creating a default kernel page” on page 1-32).

  You will be presented with the following dialog:



Event List TR_SYSCALL_RESUME
No Event List NONE
If Expression arg2 == get_item(syscall, "fbswait")
CPU List ALL
PID List ALL
TID List ALL

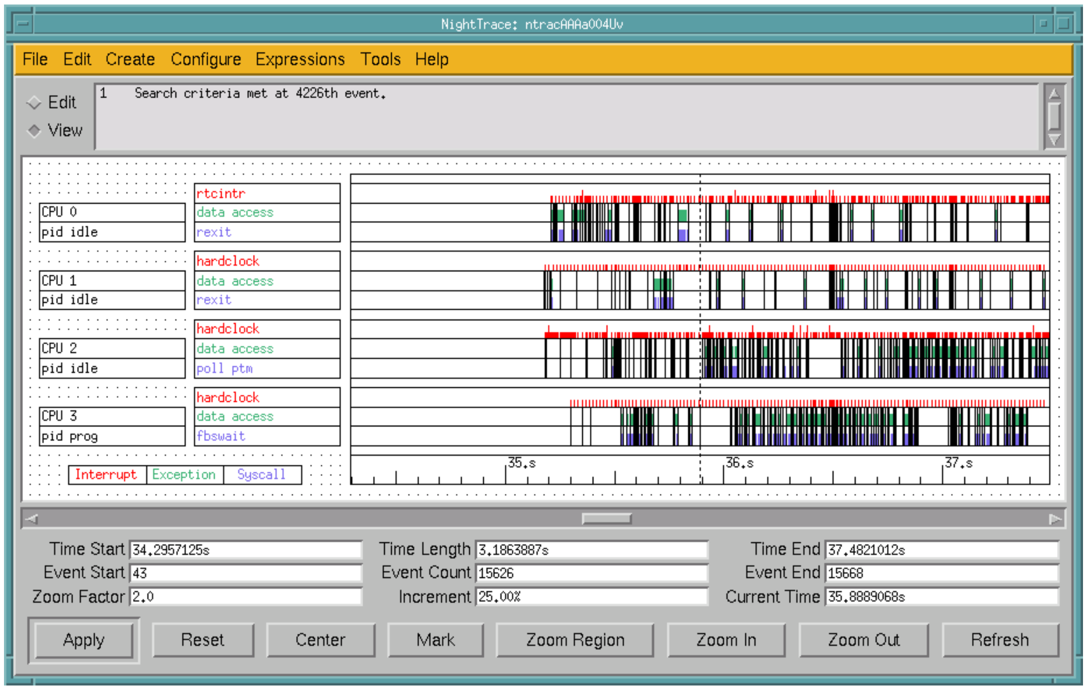
Figure 1-19. Searching for a kernel trace event

- Enter TR_SYSCALL_RESUME in the Event List field. This trace event is logged whenever a system call (syscall) is resumed (i.e., the process that caused the syscall to occur, which was switched out before the syscall could be completed, is switched back in).
- Enter arg2 == get_item(syscall, "fbswait") in the If Expression field. The `fbswait` system call corresponds to the `fbswait` call we made in our Fortran program.
- Press Apply.
- Press Search.

NightTrace will set the current time to that of the first logged kernel trace event that matches the specified search criteria, positioning the grid on the kernel display page accordingly. This is shown in the figure below. Note the Current Time. In our example, it is set to 72.1783521 seconds.

NOTE

Since we specified the **`-bufferwrap`** option to **`ktrace`** (see "To activate kernel tracing" on page 1-14), it is likely that the earlier trace events may have been overwritten by buffer wraparound during the execution of the program. Hence, we may not actually see the *first* actual kernel trace event that corresponds to our search criteria. However, this is sufficient for our example.

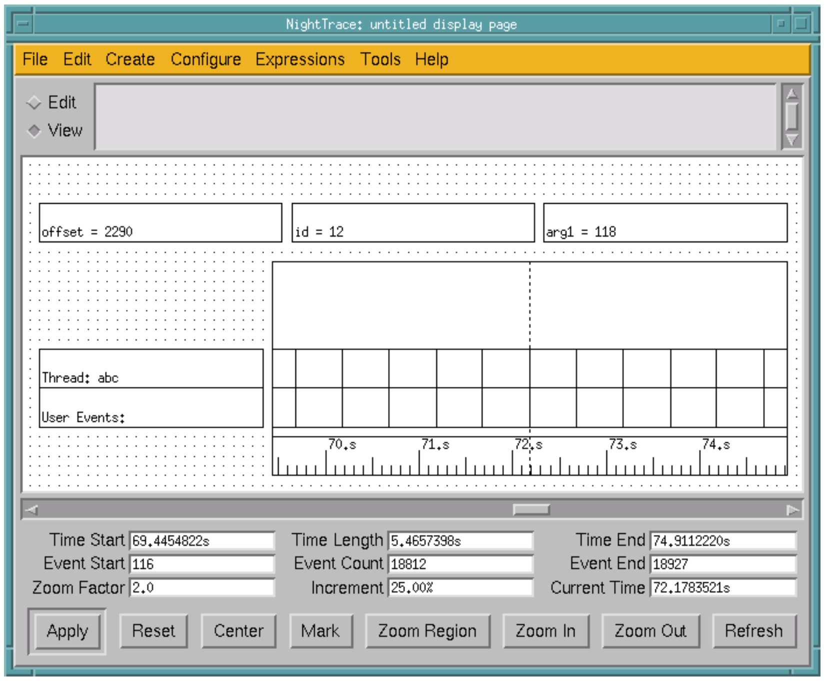


```
1    Search criteria met at 4226th event.
```

Time Start 34.2957125s
Event Start 43
Zoom Factor 2.0
Time Length 3.1863887s
Event Count 15626
Increment 25.00%
Time End 37.4821012s
Event End 15668
Current Time 35.8889068s

Figure 1-20. First kernel trace event

In addition to setting the current time and repositioning the grid on the kernel display page when the search for the kernel trace event was performed, NightTrace will automatically set the current time and reposition the display page that contains the user trace events as well. This is shown in the following figure.


NightTrace: untitled display page
File Edit Create Configure Expressions Tools Help
Edit
View
70.s
71.s
72.s
73.s
74.s
Apply Reset Center Mark Zoom Region Zoom In Zoom Out Refresh

offset = 2290
id = 12
arg1 = 118
Thread: abc
User Events:

Time Start 69.4454822s
Event Start 116
Zoom Factor 2.0
Time Length 5.4657398s
Event Count 18812
Increment 25.00%
Time End 74.9112220s
Event End 18927
Current Time 72.1783521s

Figure 1-21. NightTrace display page repositioned accordingly

Searching for a user trace event

Now that we have found the first logged kernel trace event, we can search for the user trace events that we logged using NightView (see “Adding a tracepoint in the program” on page 1-18).

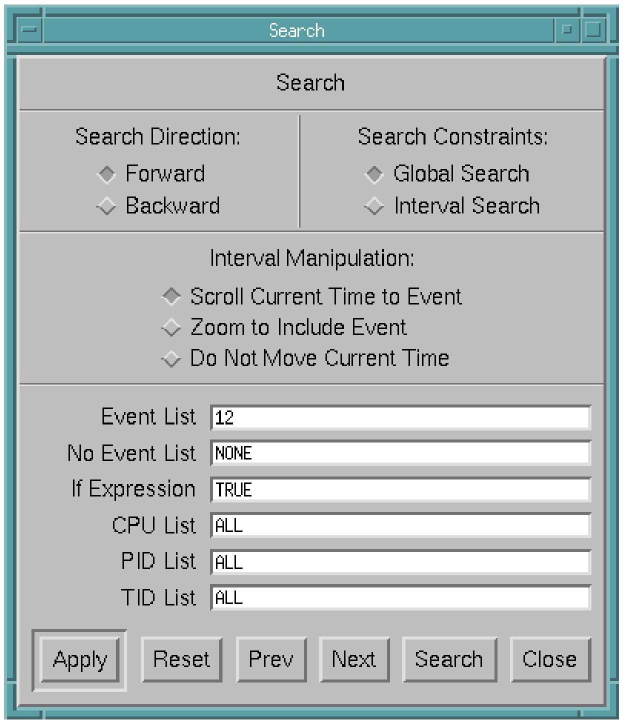
To search for a user trace event

NOTE

You may use the same search dialog that you used in the previous step, “Searching for a kernel trace event” on page 1-33.

- Select Search... from the Tools menu of the display page containing the user trace events (see “Creating a default page” on page 1-32).

  You will be presented with the following dialog:



Search Direction: [✓ Forward] Backward
Search Constraints: [✓ Global Search] Interval Search
Interval Manipulation: [✓ Scroll Current Time to Event] Zoom to Include Event Do Not Move Current Time

Event List	12
No Event List	NONE
If Expression	TRUE
CPU List	ALL
PID List	ALL
TID List	ALL

Figure 1-22. Searching for a user trace event

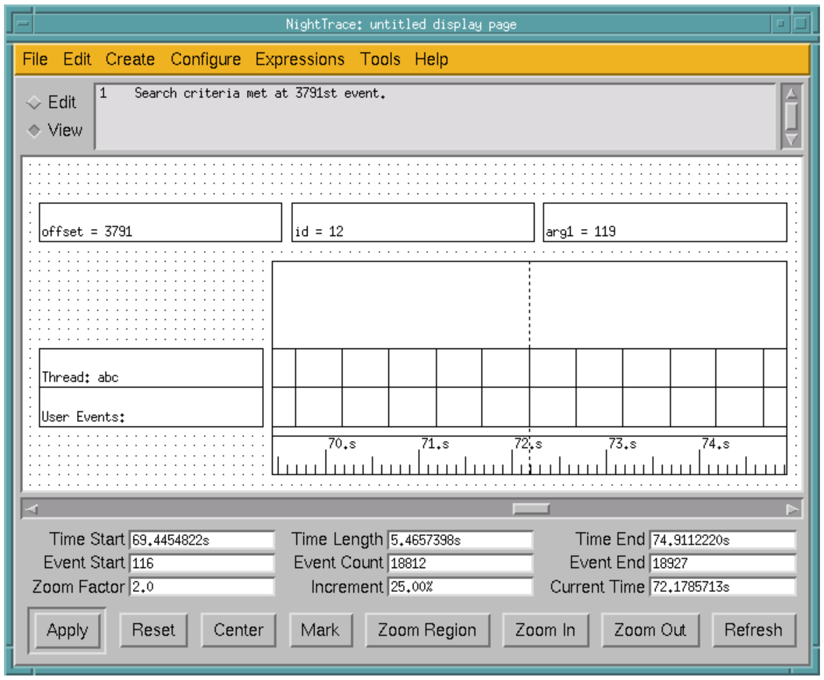
- Enter 12 in the Event List field. This corresponds to the Event ID for the tracepoint we specified in NightView (see “Adding a tracepoint in the program” on page 1-18).
- Ensure that the value of the If Expression field is TRUE.
- Press Apply.
- Press Search.

NightTrace will set the current time to the first user trace event after the current time that matches the specified search criteria, positioning the grid on the kernel display page accordingly. This is shown in the figure below. Note the Current Time now. In our example, it is set to 72.1785713 seconds, 0.0002192 seconds after the `fbswait` system call we found in “Searching for a kernel trace event” on page 1-33.

You can alternately search between the kernel display page (see “To search for a kernel trace event” on page 1-33) and the display page which contains the user trace events (see “To search for a user trace event” on page 1-36) to see that an `fbswait` system call always precedes the user trace event that we logged, which is what we would expect.

NOTE

If you used the same search dialog as you used for searching for a kernel trace event, you may use the `Prev` button on the search dialog for the previous search criteria. You can alternate between searching for user trace events and kernel trace events using this functionality.


NightTrace: untitled display page
File Edit Create Configure Expressions Tools Help
Edit
View
70.s
71.s
72.s
73.s
74.s
Apply
Reset
Center
Mark
Zoom Region
Zoom In
Zoom Out
Refresh

```
1    Search criteria met at 3791st event.
```

offset = 3791

id = 12

arg1 = 119

Thread: abc

User Events:

Time Start	69.4454822s	Time Length	5.4657398s	Time End	74.9112220s
Event Start	116	Event Count	18812	Event End	18927
Zoom Factor	2.0	Increment	25.00%	Current Time	72.1785713s

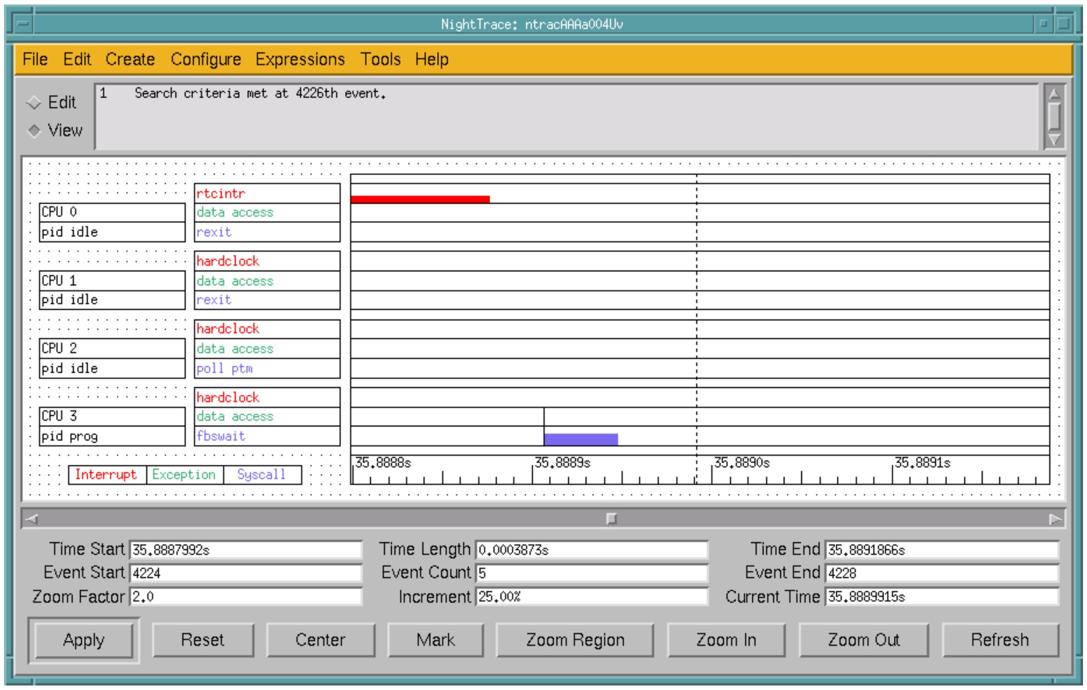
Figure 1-23. NightTrace display page

Zooming in

To zoom in:

- You may use the `Zoom In` button on the NightTrace Analyzer to see more details.

  For our example, we zoomed in on our kernel display page 13 times to see the following level of detail.



```
1    Search criteria met at 4226th event.
```

Time Start 35.8887992s
Event Start 4224
Zoom Factor 2.0
Time Length 0.0003873s
Event Count 5
Increment 25.00%
Time End 35.8891866s
Event End 4228
Current Time 35.8889915s

Figure 1-24. Zoomed in kernel display page

In the above figure, the first bar (red) listed for CPU 0 indicates the real-time clock interrupt for this cycle. The first bar (blue) listed for CPU 3 shows the target process **`prog`** exiting the `fbswait` call in the Fortran code. The current time line is positioned at the user trace event that we previously searched for.

Looking at the other display page (which shows our user trace events), we can see the user trace event inserted through NightView (see "Adding a tracepoint in the program" on page 1-18). Note that both displays are synchronized in time (the current time line represents the same instant in time on both display pages). You may middle-click on the line representing the user trace event to see more detailed information.

NOTE

Due to a problem with the **`-bufferwrap`** option to the **`ktrace`** command, user and kernel data may not appear synchronized. This problem has been fixed in the **`ktrace`** and **`ntfilter`** commands in PowerMAX OS 4.3 Patch Set 6 (**`trace-004`** and **`base-006`**). See "To activate kernel tracing" on page 1-14 for more information.

Conclusion

This concludes our tutorial for using the Concurrent Fortran 95 compiler with the Night-Star tools. We hope that we have given you a sufficient overview of the various tools and the interactions between them.