**sgi**®

REACT™ Real-Time for Linux®
Programmer's Guide

# New Features in this Guide

This revision contains the following:

- Support for SGI Altix XE systems.

  **Note:** This release provides support for interrupt response guarantee and external interrupt processing on the Altix XE platform.

- External interrupt device access can now be coordinated through the use of the `flock`(2) system call.

- Removal of the concept of temporarily disabling or enabling. Now REACT is either enabled or disabled. You should control REACT only through the `reactconf.pl` script; you should not use the `chkconfig` command or the `/etc/init.d/sgi_react` script. See "Disabling and Enabling REACT" on page 117.

# Record of Revision

| Version | Description |
|---------|-------------|
| 001 | February 2005<br>Original publication to support REACT real-time for Linux 4.0 |
| 002 | July 2005<br>Revision to support REACT real-time for Linux 4.2 |
| 003 | December 2005<br>Revision to support REACT real-time for Linux 4.3 |
| 004 | July 2006<br>Revision to support REACT real-time for Linux 5.0 |
| 005 | February 2007<br>Revision to support REACT real-time for Linux 5.1 |

# Contents

# Figures

# Tables

# Examples

# About This Guide

A *real-time program* is one that must maintain a fixed timing relationship to external hardware. In order to respond to the hardware quickly and reliably, a real-time program must have special support from the system software and hardware. This guide describes the facilities of REACT real-time for Linux.

## Audience

This guide is written for real-time programmers. You are assumed to be:

- An expert in the C programming language

- Knowledgeable about the hardware interfaces used by your real-time program

- Familiar with system-programming concepts such as interrupts, device drivers, multiprogramming, and semaphores

You are not assumed to be an expert in Linux system programming, although you do need to be familiar with Linux as an environment for developing software.

## What This Guide Contains

This guide contains the following:

- Chapter 1, "Introduction" on page 1 describes the important classes of real-time programs, emphasizing their performance requirements

- Chapter 2, "Linux and REACT Support for Real–Time Programs" on page 9 provides an overview of the real-time support for programs in Linux and REACT

- Chapter 4, "CPU Workload" on page 37 describes how you can isolate a CPU and dedicate almost all of its cycles to your program's use

- Chapter 5, "Using the Frame Scheduler" on page 55 describes how to structure a real-time program as a family of independent, cooperating activities, running on multiple CPUs, scheduled in sequence at the frame rate of the application

- Chapter 6, "Disk I/O Optimization" on page 91 describes how to set up disk I/O to meet real-time constraints, including the use of asynchronous I/O

- Chapter 7, "PCI Devices" on page 93 discusses the Linux PCI interface

- Chapter 8, "User-Level Interrupts" on page 95 discusses the facility that is intended to simplify the creation of device drivers for unsupported devices

- Chapter 9, "REACT System Configuration" on page 105 discusses the `reactconf.pl` configuration script that is provided to assist you in setting up REACT

- Chapter 11, "Troubleshooting" on page 129 discusses diagnostic tools that apply to real-time applications and common problems

- Appendix A, "Example Application for SGI Altix and SGI Altix XE" on page 133 provides excerpts of application modules to be used with REACT

- Appendix B, "High-Resolution Timer Example" on page 173 demonstrates the use of SGI high-resolution timers

- Appendix C, "Sample User-Level Interrupt Program" on page 179 contains a sample program that shows how user-level interrupts are used

- Appendix D, "Reading MAC Addresses Sample Program" on page 185 provides a sample program for reading the MAC address from an Ethernet card

- Appendix E, "IRIX Differences" on page 189 summarizes some differences between REACT for IRIX and REACT for Linux

## Related Publications and Sites

The following may be useful:

- Available from the online SGI Technical Publications Library:

    - The user guide for your SGI system

    - *Linux Configuration and Operations Guide*

    - *Linux Device Driver Programmer's Guide-Porting to SGI Altix Systems*

    - *SGI L1 and L2 Controller Software User's Guide*

    - *SGI ProPack 5 for Linux Service Pack 1 Start Here*

    - *SUSE LINUX Enterprise Server for SGI Altix Systems*

- *Porting IRIX Applications to SGI Altix Platforms: SGI ProPack for Linux*

- *The Linux Programmer's Guide* (Sven Goldt, Sven van der Meer, Scott Burkett, Matt Welsh)

- *The Linux Kernel* (David A Rusling)

- *Linux Kernel Module Programming Guide* (Ori Pomerantz)

- *Linux Device Drivers*, third edition, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, February 2005 (ISBN: 0-596-00590-3):

  http://www.oreilly.com/catalog/linuxdrive3/

For more information about the SGI Altix series, see the following sites:

- http://www.sgi.com/products/servers/altix

- http://www.sgi.com/products/servers/altix/350

## Conventions

The following conventions are used throughout this document:

| Convention | Meaning |
|---|---|
| [ ] | Brackets enclose optional portions of a command or directive line. |
| command | This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures. |
| ... | Ellipses indicate that a preceding element can be repeated. |
| manpage(*x*) | Man page section identifiers appear in parentheses after man page names. |
| **user input** | This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.) |

| | |
|---|---|
| *variable* | Italic typeface denotes variable entries and words or concepts being defined. |

## Obtaining Publications

You can obtain SGI documentation as follows:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.

- You can view release notes on your system by accessing the README file(s) for the product. This is usually located in the /usr/share/doc/*productname* directory, although file locations may vary.

- You can view man pages by typing man *title* at a command line.

## Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  SGI
  Technical Publications
  1200 Crittenden Lane, M/S 535
  Mountain View, California 94043–1351

SGI values your comments and will respond to them promptly.

# Introduction

This chapter discusses the following:

- "Real-Time Programs" on page 1

- "Real-Time Applications" on page 1

- "REACT Real-Time for Linux" on page 6

- "REACT Installation Requirements" on page 7

## Real-Time Programs

A *real-time program* is any program that must maintain a fixed, absolute timing relationship with an external hardware device. A *hard real-time program* is a program that experiences a catastrophic error if it misses a deadline. A *firm real-time program* is a program that experiences a significant error if it misses a deadline but is able to recover from the error and can continue to execute. A *soft real-time program* is a program that can occasionally miss a deadline with only minor adverse effects.

A *normal-time program* is a correct program when it produces the correct output, no matter how long that takes. Normal-time programs do not require a fixed timing relationship to external devices. You can specify performance goals for a normal-time program (such as "respond in at most 2 seconds to 90% of all transactions"), but if the program does not meet the goals, it is merely slow, not incorrect.

## Real-Time Applications

The following are examples of real-time applications:

- "Simulators and Stimulators" on page 2

- "Data Collection Systems" on page 4

- "Process Control Systems" on page 5

## Simulators and Stimulators

A *simulator* or a *stimulator* maintains an internal model of the world. It receives control inputs, updates the model to reflect them, and outputs the changed model. It must process inputs in real time in order to be accurate. The difference between them is that a simulator provides visual output while a stimulator provides nonvisual output. SGI systems are well-suited to programming many kinds of simulators and stimulators.

Simulators and stimulators have the following components:

- An internal model of the world, or part of it; for example, a model of a vehicle traveling through a specific geography, or a model of the physical state of a nuclear power plant.

- External devices to supply control inputs; for example, a steering wheel, a joystick, or simulated knobs and dials. (This does not apply to all stimulators.)

- An operator (or hardware under test) that closes the feedback loop by moving the controls in response to what is shown on the display. A *feedback loop* provides input to the system in response to output from the system. (This does not apply to all stimulators.)

Simulators also have the external devices to display the state of the model; for example, video displays, audio speakers, or simulated instrument panels.

The real-time requirements vary depending on the nature of these components. The following are key performance requirements:

- *Frame rate* is the rate at which the simulator updates the display, whether or not the simulator displays its model on a video screen. Frame rate is given in cycles per second (*hertz*, abbreviated *Hz*). Typical frame rates run from 15 Hz to 60 Hz, although rates higher and lower than these are used in special situations.

  The inverse of frame rate is *frame interval*. For example, a frame rate of 60 Hz implies a frame interval of 1/60 second, or 16.67 milliseconds (ms). To maintain a frame rate of 60 Hz, a simulator must update its model and prepare a new display in less than 16.67 ms.

- *Transport delay* is the number of frames that elapses before a control motion is reflected in the display. When the transport delay is too long, the operator perceives the simulation as sluggish or unrealistic. If a visual display in a simulator lags behind control inputs, a human operator can become physically ill. In the case where the operator is physical hardware, excessive transport delay can cause the control loop to become unstable.

## Aircraft Simulators

Simulators for real or hypothetical aircraft or spacecraft typically require frame rates of 30 Hz to 120 Hz and transport delays of 1 or 2 frames. There can be several analogue control inputs and possibly many digital control inputs (simulated switches and circuit breakers, for example). There are often multiple video display outputs (one each for the left, forward, and right "windows") and possibly special hardware to shake or tilt the "cockpit." The display in the "windows" must have a convincing level of detail.

## Ground Vehicle Simulators

Simulators for automobiles, tanks, and heavy equipment have been built with SGI systems. Frame rates and transport delays are similar to those for aircraft simulators. However, there is a smaller world of simulated "geography" to maintain in the model. Also, the viewpoint of the display changes more slowly, and through smaller angles, than the viewpoint from an aircraft simulator. These factors can make it somewhat simpler for a ground vehicle simulator to update its display.

## Plant Control Simulators

A simulator can be used to train the operators of an industrial plant such as a nuclear or conventional power-generation plant. Power-plant simulators have been built using SGI systems.

The frame rate of a plant control simulator can be as low as 1 or 2 Hz. However, the number of control inputs (knobs, dials, valves, and so on) can be very large. Special hardware may be required to attach the control inputs and multiplex them onto the PCI bus. Also, the number of display outputs (simulated gauges, charts, warning lights, and so on) can be very large and may also require custom hardware to interface them to the computer.

## Virtual Reality Simulators

A virtual reality simulator aims to give its operator a sense of presence in a computer-generated world. A difference between a vehicle simulator and a virtual reality simulator is that the vehicle simulator strives for an exact model of the laws of physics, while a virtual reality simulator typically does not.

Usually the operator can see only the simulated display and has no other visual referents. Because of this, the frame rate must be high enough to give smooth, nonflickering animation; any perceptible transport delay can cause nausea and

disorientation. However, the virtual world is not required (or expected) to look like the real world, so the simulator may be able to do less work to prepare the display than does a vehicle simulator

SGI systems, with their excellent graphic and audio capabilities, are well suited to building virtual reality applications.

### Hardware-in-the-Loop Simulators

The operator of a simulator need not be a person. In a *hardware-in-the-loop* (HWIL) simulator, the human operator is replaced by physical hardware such as an aircraft autopilot or a missile guidance computer. The inputs to the system under test are the simulator's output. The output signals of the system under test are the simulator's control inputs.

Depending on the hardware being exercised, the simulator may have to maintain a very high frame rate, up to several thousand Hz. SGI systems are excellent choices for HWIL simulators.

### Control Law Processor Stimulator

An example of a *control law processor* is one that simulates the effects of Newton's law on an aircraft flying through the air. When the rudder is turned to the left, the information that the rudder had turned, the velocity, and the direction is fed into the control law processor. The processor calculates and returns a response that represents the physics of motion. The pilot in the simulator cockpit will feel the response and the instruments will show the response. However, a human did not actually interact directly with the processor; it was a machine-to-machine interaction.

### Wave Tank Stimulator

A wave tank simulates waves hitting a ship model under test. The stimulator must "push" the water at a certain rhythm to keep the waves going. An operator may adjust the frequency and amplitude of the waves, or it could run on a preprogrammed cycle.

## Data Collection Systems

A *data collection system* receives input from reporting devices (such as telemetry receivers) and stores the data. It may be required to process, reduce, analyze, or

compress the data before storing it. It must respond in real time to avoid losing data. SGI systems are suited to many data collection tasks.

A data collection system has the following major parts:

- Sources of data such as telemetry (the PCI bus, serial ports, SCSI devices, and other device types can be used).

- A repository for the data. This can be a raw device (such as a tape), a disk file, or a database system.

- Rules for processing. The data collection system might be asked only to buffer the data and copy it to disk. Or it might be expected to compress the data, smooth it, sample it, or filter it for noise.

- Optionally, a display. The data collection system may be required to display the status of the system or to display a summary or sample of the data. The display is typically not required to maintain a particular frame rate, however.

The first requirement on a data collection system is imposed by the *peak data rate* of the combined data sources. The system must be able to receive data at this peak rate without an *overrun*; that is, without losing data because it could not read the data as fast as it arrived.

The second requirement is that the system must be able to process and write the data to the repository at the *average data rate* of the combined sources. Writing can proceed at the average rate as long as there is enough memory to buffer short bursts at the peak rate.

You might specify a desired frame rate for updating the display of the data. However, there is usually no real-time requirement on display rate for a data collection system. That is, the system is correct as long as it receives and stores all data, even if the display is updated slowly.

## Process Control Systems

A *process control system* monitors the state of an industrial process and constantly adjusts it for efficient, safe operation. It must respond in real time to avoid waste, damage, or hazardous operating conditions.

An example of a process control system would be a power plant monitoring and control system required to do the following:

- Monitor a stream of data from sensors

- Recognize a dangerous situation has occurred

- Visualize the key data, such as by highlighting representations of down physical equipment in red and sending audible alarms

The danger must be recognized, flagged, and responded to quickly in order for corrective action to be taken appropriately. This entails a real-time system. SGI systems are suited for many process control applications.

## REACT Real-Time for Linux

REACT provides the following:

- SGI Linux Trace debug kernel to provide trace information for analyzing the impact of kernel operations on application performance.

- reactconf.pl system configuration script to help you easily generate a real-time system.

- User-level interrupts to allow you to handle hardware interrupts from a user process.

- A frame scheduler that makes it easier to structure a real-time program as a family of independent, cooperating activities that are running on multiple CPUs and are scheduled in sequence at the frame rate of the application.

- The kbar(3) kernel barrier facility, which provides for the fast wake-up of many blocked user threads.

- SGI high-resolution POSIX timers. REACT high-resolution timers use the real-time clock (RTC).

- IRIX to Linux REACT compatibility library.

**Note:** Real–time programs using REACT should be written in the C language, which is the most common language for system programming on Linux.

# REACT Installation Requirements

The following RPMs are used for REACT:

- **Required** RPMs:

  – Cpuset and bitmask:

  ```
  cpuset-utils
  libbitmask
  libcpuset
  ```

  – External interrupts (see Chapter 3, "External Interrupts" on page 17):

  ```
  extint
  sgi-extint-kmp-*
  ```

  – REACT configuration (see Chapter 9, "REACT System Configuration" on page 105):

  ```
  react-configuration
  ```

- **Optional** RPMs:

  – Kernel barrier facility (see "kbar Kernel Barrier Facility" on page 39):

  ```
  kbar
  sgi-kbar-kmp-*
  ```

  – IRIX compatibility and sgi-shield (see "Shielding a CPU from Timer Interrupts" on page 48):

  ```
  libsgirt
  sgi-shield-kmp-*
  ```

  – Frame scheduler (see Chapter 5, "Using the Frame Scheduler" on page 55):

  ```
  frs
  sgi-frs-kmp-*
  ```

  – User-level interrupts (see Chapter 8, "User-Level Interrupts" on page 95):

  ```
  uli
  sgi-uli-kmp-*
  ```

– SGI Linux Trace (see Chapter 10, "SGI Linux Trace" on page 119):

```
TraceToolkit
kernel-sltdebug
kernel-sltdebug-source
```

For installation instructions, see the *SGI ProPack 5 for Linux Service Pack 1 Start Here*

# Linux and REACT Support for Real–Time Programs

This chapter provides an overview of the real-time support for programs in Linux:

- "Kernel Facilities" on page 9
- "Frame Scheduler" on page 11
- "Clocks and Timers" on page 12
- "Interchassis Communication" on page 15

## Kernel Facilities

The Linux kernel has a number of features that are valuable when you are designing a real-time program. These are described in the following sections:

- "Special Scheduling Disciplines" on page 9
- "Virtual Memory Locking" on page 10
- "Processes Mapping and CPUs" on page 10
- "Interrupt Distribution Control" on page 11

### Special Scheduling Disciplines

The default Linux scheduling algorithm is designed to ensure fairness among time-shared users. The priorities of time-shared threads are largely determined by the following:

- Their `nice` value
- The degree to which they are CPU-bound versus I/O-bound

While a time-share scheduler is effective at scheduling most standard applications, it is not suitable for real time. For deterministic scheduling, Linux provides the following POSIX real-time policies:

- First-in-first-out

- Round-robin

These policies share a real-time priority band consisting of 99 priorities.For more information about scheduling, see "Real-Time Priority Band" on page 38 and the `sched_setscheduler`(2) man page.

## Virtual Memory Locking

Linux allows a task to lock all or part of its virtual memory into physical memory so that it cannot be paged out and so that a page fault cannot occur while it is running.

Memory locking prevents unpredictable delays caused by paging, but the locked memory is not available for the address spaces of other tasks. The system must have enough physical memory to hold the locked address space and space for a minimum of other activities.

Examples of system calls used to lock memory are `mlock`(2) and `mlockall`(2).

## Processes Mapping and CPUs

Normally, Linux tries to keep all CPUs busy, dispatching the next ready process to the next available CPU. Because the number of ready processes changes continuously, dispatching is a random process. A normal process cannot predict how often or when it will next be able to run. For normal programs, this does not matter as long as each process continues to run at a satisfactory average rate. However, real-time processes cannot tolerate this unpredictability. To reduce it, you can dedicate one or more CPUs to real-time work by using the following steps:

1. Restrict one or more CPUs from normal scheduling so that they can run only the processes that are specifically assigned to them.

2. Isolate one or more CPUs from the effects of scheduler load-balancing.

3. Assign one or more processes to run on the restricted CPUs.

A process on a dedicated CPU runs when it needs to run, delayed only by interrupt service and by kernel scheduling cycles.

## Interrupt Distribution Control

In normal operations, a CPU receives frequent interrupts:

- I/O interrupts from devices attached to, or near, the CPU

- Timer interrupts that occur on every CPU

- Console interrupts that occur on the CPU servicing the system console

These interrupts can make the execution time of a process unpredictable. I/O interrupt control is done by /proc filesystem manipulation. For more information on controlling I/O interrupts, see "Redirect Interrupts" on page 46.

You can minimize console interrupt effects with proper real-time thread placement. You should not run time-critical threads on the CPU that is servicing the system console.

You can see where console interrupts are being serviced by examining the /proc/interrupts file:

```
[root@linux root]# cat /proc/interrupts
          CPU0       CPU1       CPU2       CPU3
..
233:         0      12498          0          0        SN hub  SAL console driver
..
```

The above shows that 12,498 console driver interrupts have been serviced by CPU 1. In this case, CPUs 2 and 3 would be much better choices for running time-critical threads because they are not servicing console interrupts.

Timer processing is always performed on the CPU from which the timer was started, such as by executing a POSIX timer_settime() call. You can avoid the effects of timer processing by not allowing execution of any threads other than time-critical threads on CPUs that have been designated as such. If your time-critical threads start any timers, the timer processing will result in additional latency when the timeout occurs.

# Frame Scheduler

Many real-time programs must sustain a fixed frame rate. In such programs, the central design problem is that the program must complete certain activities during every frame interval.

The *frame scheduler* is a process execution manager that schedules activities on one or more CPUs in a predefined, cyclic order. The scheduling interval is determined by a repetitive time base, usually a hardware interrupt.

The frame scheduler makes it easy to organize a real-time program as a set of independent, cooperating threads. You concentrate on designing the activities and implementing them as threads in a clean, structured way. It is relatively easy to change the number of activities, their sequence, or the number of CPUs, even late in the project. For more information, see Chapter 5, "Using the Frame Scheduler" on page 55.

# Clocks and Timers

**Note:** This section does not apply to Altix XE servers.

This section discusses the following:

- "Clocks" on page 12
- "Direct RTC Access" on page 14
- "ITC Register Access" on page 14
- "SGI High-Resolution POSIX Timers" on page 14

## Clocks

SGI systems provide a systemwide clock called a *real-time clock* (RTC) that is accessible locally on every node. The RTC provides a raw time source that is incremented in 50-ns intervals. The RTC is 55 bits wide, which ensures that it will not wrap around zero unless the system has been running for more than half a century. RTC values are mapped into the local memory of each node. Multiple nodes accessing the RTC value will not reduce the performance of the clock functions. RTCs are synchronized among all of the nodes in an SGI system using a special pin on the NUMAlink cable.

The RTC is the basis for system time, which may be obtained via the the `clock_gettime` function call that is implemented in conformance with the POSIX standard. `clock_gettime` takes an argument describing which clock is wanted.

The following clock values are typically used:

- CLOCK_REALTIME is the actual current time that you would obtain from any ordinary clock. However, CLOCK_REALTIME is set during startup and may be corrected during the operation of the system. This implies that time differences observed by an application using CLOCK_REALTIME may be affected by the initial setting or the later correction of time (via clock_settime) and therefore may not accurately reflect time that has passed for the system.

- CLOCK_MONOTONIC starts at zero during bootup and is continually increasing. CLOCK_MONOTONIC will not be affected by time corrections and the initial time setup during boot. If you require a continually increasing time source that always reflects the real time that has passed for the system, use CLOCK_MONOTONIC.

The clock_gettime function is a *fastcall* version that was optimized in assembler and has been provided in order to bypass the context switch typically necessary for a full system call. It is highly advisable to use clock_gettime for all time needs.

Both CLOCK_REALTIME and CLOCK_MONOTONIC report a resolution via the clock_getres() function call that is in the range 1 ms through 10 ms, corresponding to the timer tick frequency in the range 1000 Hz through 100 Hz. The clock_getres() function call is POSIX compliant.

You can use either CLOCK_REALTIME or CLOCK_MONOTONIC to generate signals via the timer_create function.

**Note:** Linux can only deliver signals based on CLOCK_REALTIME and CLOCK_MONOTONIC in intervals of the timer tick. In order to fulfill the mandates of the POSIX standard, no higher resolution can be reported via clock_getres(). However, the actual resolution of both clocks is the full resolution of the RTC. Time can be measured in 50-ns intervals with both clocks, but it is not possible to generate signals with this accuracy using these clocks. For information on generating signals with greater accuracy than the timer tick allows, see "SGI High-Resolution POSIX Timers" on page 14.

To determine the tick frequency, use the sysconf(_SC_CLK_TCK) function. The sysconf(_SC_CLK_TCK) will always return the right value on SGI Altix systems.

## Direct RTC Access

In some situations, the overhead of the `clock_gettime` fastcall may be too high. In that case, direct memory-mapped access to the RTC counter is useful. (See the comments in `mmtimer.h`..)

**Note:** Measurements have shown that the code generated by a function written to obtain the RTC value and then calculate the nanoseconds that have passed is slower than the fastcall for `clock_gettime`. Direct use of the RTC is only advisable for timestamps.

Like `CLOCK_MONOTONIC`, the RTC counter is monotonically increasing from bootup and is not affected by setting the time.

## ITC Register Access

The Itanium processor provides a 64-bit counter incremented by the processor clock called the *interval time counter* (ITC). ITC register accesses are very fast compared to the RTC (which must retrieve a value from memory) and the ITC typically allows measurements of much smaller time intervals than the RTC.

However, the ITC is a local processor-based counter. The clock frequencies of processors in an SGI system are not synchronized and may be running at different frequencies. Therefore, if you want to measure an interval by using the ITC, you must ensure that the process is not migrating to a different processor. Furthermore, in order to obtain meaningful time information from differences in ITC values, you must know the local ITC frequency.

In contrast to the ITC register, the RTC counter is a global time source and is not subject to these limitations.

## SGI High-Resolution POSIX Timers

You can use POSIX timers to generate signals at higher resolution than the system timer tick on SGI systems by using the SGI `CLOCK_SGI_CYCLE` clock ID. POSIX timers with the `CLOCK_SGI_CYCLE` clock ID use the RTC as a time source. The REACT frame scheduler uses this hardware timer for its high-resolution timer solution. Frame scheduler applications cannot use POSIX high-resolution timers. For more information, see "Clocks" on page 12.

To use SGI high-resolution POSIX timers, do the following:

1. Set up the timer by using `timer_create()` with the kernel value for `CLOCK_SGI_CYCLE` as the clock ID. For more information on `timer_create()`, see the `timer_create`(3p) man page.

2. Use the returned `timer_t` value for the other POSIX timer functions, such as `timer_settime()`.

The `CLOCK_SGI_CYCLE` clock ID is only valid with the `timer_create()` call. It is an invalid argument with other functions that accept clock IDs, such as `clock_gettime()`.

There is a limit of three outstanding high-resolution timers allowed at any given time per hardware node, which means per pair of CPUs on most systems. Attempting to set more than three timers per node will result in `timer_settime()` returning `EBUSY`.

For more information, see Appendix B, "High-Resolution Timer Example" on page 173.

# Interchassis Communication

This section discusses socket programming and the Message-Passing Interface (MPI). The performance of both sockets and MPI depends on the speed of the underlying network. The network that connects nodes (systems) in an array product has a very high bandwidth.

## Socket Programming

One standard, portable way to connect processes in different computers is to use the BSD-compatible socket I/O interface. You can use sockets to communicate within the same machine, between machines on a local area network, or between machines on different continents.

## Message-Passing Interface (MPI)

The Message-Passing Interface (MPI) is a standard architecture and programming interface for designing distributed applications. For the MPI standard, see:

http://www.mcs.anl.gov/mpi

SGI supports MPI in SGI ProPack.

# External Interrupts

Real-time processes often require the ability to react to an external event. *External interrupts* are a way for a real-time process to receive a real-world external signal.

An external interrupt is generated via a signal applied to the external interrupt socket on systems supporting such a hardware feature, such as the IO9 card on an SGI Altix system, which has a 1/8-inch stereo-style jack into which a 0-5V signal can be fed. An exterior piece of hardware can assert this line, causing the card's IOC4 chip to generate an interrupt.

This chapter discusses the external interrupts feature and, as an example, the SGI IOC4 PCI device.

This section discusses the following:

- "Abstraction Layer" on page 17
- "Making Use of Unsupported Hardware Device Capabilities " on page 30
- "Low-level Driver Template" on page 31
- "Example: SGI IOC4 PCI Device" on page 31

## Abstraction Layer

Various external interrupt hardware might implement the external interrupt feature in very different ways. The *external interrupt abstraction layer* provides the ability to determine when an interrupt occurs, count the number of interrupts, and select the source of those interrupts without depending upon specifics of the device being used.

This section discusses the following:

- "`sysfs` Attribute Files" on page 18
- "The `/dev/extint#` Device" on page 20
- "Low-level Driver Interface" on page 24
- "Interrupt Notification Interface" on page 28

## sysfs **Attribute Files**

The external interrupt abstraction layer provides a character device and the following sysfs attribute files to control operation:

```
dev
mode
modelist
period
provider
quantum
source
sourcelist
```

Assuming the usual /sys mount-point for sysfs, the attribute files are located in the following directory:

```
/sys/class/extint/extint#/
```

The extint# component of the path is determined by the extint driver itself. The # character is replaced by a number (possibly multidigit), one per external interrupt device, beginning at 0. For example, if there were three devices, there would be three directories:

```
/sys/class/extint/extint0/
/sys/class/extint/extint1/
/sys/class/extint/extint2/
```

The attribute files are as follows:

| File | Description |
| --- | --- |
| dev | Contains the major and minor number of the abstracted external interrupt device. If sysfs, hotplug, and udev are configured appropriately, udev will automatically create a /dev/extint# character special device file with this major and minor number. If you prefer, you may manually invoke mknod(1) to create the character special device file. Once created, this device file provides a counter that can be used by applications in a variety of ways. See"The /dev/extint# Device" on page 20. |
| mode | Contains the shape of the output signal for interrupt generation. For example, SGI's IOC4 chip can set the output to one of the following: |

high, low, `pulse`, `strobe`, or `toggle`. For more information, see "External Interrupt Output" on page 32.

`modelist`    Contains the list of available **valid** output modes, one per line. These strings are the legal valid values that can be written to the `mode` attribute. For more information, see "External Interrupt Output" on page 32.

---

**Note:** For the SGI IOC4 chip, there are other values that may be read from the `mode` attribute file that do not appear in `modelist`; these represent **invalid** hardware states. Only the modes present from the `modelist` are valid settings to be written to the mode attribute.

---

`period`    Contains the repetition interval for periodic output signals (such as repeated strobes, automatic toggling). This period is specified in nanoseconds, and is written as a string. For more information, see "External Interrupt Output" on page 32.

`provider`    Contains an indication of which low-level hardware driver and device instance are attached to the external interrupt interface. This string is free-form and is determined by the low-level driver. For example, the SGI IOC4 low-level driver will return a string of the form `ioc4_intout#`.

---

**Note:** The # value in `ioc4_intout#` is not necessarily the same number used for `extint#`, particularly if multiple different low-level drivers are in use (for example, IOC3 and IOC4).

---

`quantum`    Contains the interval to which any writes of the `period` attribute will be rounded. Because external interrupt output hardware may not support nanosecond granularity for output periods, this attribute allows you to determine the supported granularity. The behavior of the interrupt output (when a value that is not a multiple of the quantum is written to the `period` attribute) is determined by the specific low-level external interrupt drive. However, generally the low-level driver should round to the nearest available quantum multiple. For example, suppose the `quantum` value is 7800. If a value of 75000 was written into the `period` attribute, this would represent 9.6 quantums. The actual period will be rounded to 10 quantums, or 78000 nanoseconds. The actual period will be returned upon subsequent reads from the `period`

attribute. For more information, see "External Interrupt Output" on page 32.

source      Contains the hardware source of interrupts. For example, SGI's IOC4 chip can trigger either from the external pin or from an internal loopback from its interrupt output section.

sourcelist Contains the list of available interrupt sources, one per line. These strings are the legal values that can be written to the source attribute file.

## The `/dev/extint#` Device

This section discusses the operations that an application can perform with the read-only external interrupt device file /dev/extint#:

* "Counting Interrupts" on page 20

* "Waiting for Interrupts" on page 20

* "Exclusively Accessing a Device" on page 21

### Counting Interrupts

A process may use mmap(2) to memory-map a single memory page from the external interrupt device file into the process' address space. At the beginning of this page, a counter of an unsigned long type is maintained. This counter is incremented with each external interrupt received by the device.

Alternatively, the read(2) system call returns a string representation of the counter's current value.

### Waiting for Interrupts

The poll(2) and select(2) system calls allow a process to wait for an interrupt to trigger. poll() indicates whether an interrupt has occurred since the last open(2) or read() of the file; select() blocks until the next interrupt is received.

**Exclusively Accessing a Device**

The flock(2) system call with the options LOCK_EX|LOCK_MAND ensures exclusive write access to the device attribute files (for example, /sys/class/extint/extint#/mode).

**Note:** The _GNU_SOURCE macro must be defined before including the header files in order to use the LOCK_MAND flag on the call to flock(2).

When this lock is obtained, only a process that has access to the corresponding file descriptor will be able to write to the attribute files for that device. Any other process that attempts a write(2) system call on one of these attribute files will fail with errno set to EAGAIN.

The flock() system call will block until there are no other processes that have the device file open and until no other flock() is active on the device. However, if LOCK_NB is passed to flock(), the call will fail and errno will be set to EWOULDBLOCK.

While a lock is in place, any attempt to call open(2) on the device will block. However, if O_NONBLOCK is passed to open(), the call will fail and errno will be set to EWOULDBLOCK.

To release the lock, call flock() with the LOCK_UN argument. The lock will also be automatically dropped when the last user of the corresponding file descriptor closes the file, including via a process exit. The lock will persist if the file descriptor is inherited across fork(2) or exec(2) system calls.

**Note:** You **must not** pass the LOCK_MAND flag along with the LOCK_UN flag. The flock() system call behavior is unspecified in this case.

Example 3-1 illustrates a method of searching for an unused external interrupt device that can be used exclusively by that program.

**Example 3-1** Searching for an Unused External Interrupt Device

```
#define _GNU_SOURCE

#include <stdio.h>
#include <sys/file.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <string.h>
int main(void) {
        char devfile[PATH_MAX];
        int i = 0;
        int fd;
        int found = 0;
        int status;

try_again:
        /* Search for free /dev/extint# device */
        while (i <= 255) {
                sprintf(devfile, "/dev/extint%d", i);
                i++;

                fd = open(devfile, O_RDONLY|O_NONBLOCK);
                if (fd >= 0) {
                        /* Found a unlocked device. */
                        found = 1;
                        break;
                }

                /* An error occurred.  Check why. */
                if (EWOULDBLOCK == errno) {
                        /* Found a locked device. */
                        printf("Tried %s, but it is locked.\n", devfile);
                }
                /* Some other type of error, just try next device.
                 * But don't complain about non-existent devices.
                 */
                if (ENOENT != errno)
                        printf("Unexpected error opening %s: %s\n",
                                devfile, strerror(errno));
        }

        if (!found) {
                printf("Could not find unlocked extint device to use.\n");
                return 1;
```

```
        }

        /* Try locking this device to gain exclusive access. */
        status = flock(fd, LOCK_EX|LOCK_MAND|LOCK_NB);
        if (status != 0) {
                if (EWOULDBLOCK == errno) {
                        /* The device was available, but another process
                         * has locked it between the time we opened it
                         * and made the flock() call.
                         */
                        printf("Opened %s, but someone else locked it.\n",
                                devfile);
                } else {
                        /* Some other error occurred. */
                        printf("Unexpected error locking %s: %s\n",
                                devfile, strerror(errno));
                }
                /* Try the next device. */
                found = 0;
                close(fd);
                goto try_again;
        }

        /* Successfully gained exclusive use of device */
        printf("Exclusive use of %s established.\n", devfile);

        /* Application code begins... */

        /* ... application code ends. */

        /* Unlock and close external interrupt device */
        flock(fd, LOCK_UN);
        close(fd);

        /* Successful run */
        return 0;
}
```

## Low-level Driver Interface

The low-level driver interface to the abstraction layer driver is provided through the `extint_properties` and `extint_device` structures as defined in `/usr/local/include/extint.h` and the function prototypes contained therein.

This section discusses the following:

- "Driver Registration" on page 24

- "Implementation Functions" on page 25

- "When an External Interrupt Occurs" on page 27

- "Driver Deregistration" on page 28

### Driver Registration

To register the low-level driver with the abstraction layer, use the following call:

```
struct extint_device*
extint_device_register(struct extint_properties *ep,
                       void *devdata);
```

The `ep` argument is a pointer to an `extint_properties` structure that specifies the particular low-level driver functions the abstraction layer should call when reading/writing the attributes described in "`sysfs` Attribute Files" on page 18.

The `devdata` argument is an opaque pointer that is stored by the `extint` code. To retrieve or modify this value, use the following calls:

```
void* extint_get_devdata(const struct extint_device *ed);
void extint_set_devdata(struct extint_device *ed, void* devdata);
```

This value can be used by the low-level driver to determine which of multiple devices it is operating upon.

The return value is one of the following:

- A pointer to a `struct extint_device` (which should be saved for later interrupt notification and driver deregistration).

- A negative error value (in case of registration failure). The driver should be prepared to deal with such failures.

**Implementation Functions**

The struct extint_properties call is as follows:

```
struct extint_properties {
        struct module *owner;
        ssize_t (*get_mode)(struct extint_device *ed, char *buf);
        ssize_t (*set_mode)(struct extint_device *ed, const char *buf,
                        size_t count);
        ssize_t (*get_modelist)(struct extint_device *ed, char *buf);
        unsigned long (*get_period)(struct extint_device *ed);
        ssize_t (*set_period)(struct extint_device *ed, unsigned long period);
        ssize_t (*get_provider)(struct extint_device *ed, char *buf);
        unsigned long (*get_quantum)(struct extint_device *ed);
        ssize_t (*get_source)(struct extint_device *ed, char *buf);
        ssize_t (*set_source)(struct extint_device *ed, const char *buf,
                        size_t count);
        ssize_t (*get_sourcelist)(struct extint_device *ed, char *buf);
};
```

> **Note:** Additional fields not of interest to the low-level external interrupt driver may be present. You should include /usr/local/include/extint.h to acquire these structure definitions.

The owner value should be set to the module that contains the functions pointed to by the remaining structure members. The remaining functions implement low-level aspects of the abstraction layer attributes. They all take a pointer to the struct extint_device as was returned from the registration function. In all of these functions, you can retrieve the value passed as the devdata argument to the registration function by using the following call:

```
extint_get_devdata(ed);
```

You can update the value by using the following call:

```
extint_set_devdata(ed, newvalue);
```

Typically, this value is a pointer to driver-specific data for the individual device being operated upon. It may, for example, contain pointers to mapped PCI regions where control registers reside.

| Value | Description |
|-------|-------------|
| owner | Specifies the module that contains the functions pointed to by the remaining structure members. |
| get_mode | Writes the current mode attribute of the abstraction layer into the single-page-sized buffer passed as the second argument and returns the length of the written string. |
| set_mode | Reads the mode attribute of the abstraction layer as specified in the buffer (passed as the second argument and as sized by the third) and returns the number of characters consumed (or a negative error number in event of failure). It also causes the output mode to be set as requested. |
| get_modelist | Writes strings representing the available interrupt output generation modes into the single-page-sized buffer passed as the second argument, one mode per line. It returns the number of bytes written into this buffer. This implements the modelist attribute of the abstraction layer. |
| get_period | Returns an unsigned long that represents the current repetition period, in nanoseconds. This implements the period attribute of the abstraction layer. |
| set_period | Accepts an unsigned long as the new value for the repetition period, specified in nanoseconds, and returning either 0 or a negative error number indicating a failure. If the requested repetition period is not a value that can be exactly set into the underlying hardware, the driver is free to adjust the value as it sees fit, although typically it should round the value to the nearest available value. This implements the period attribute of the abstraction layer. |
| get_provider | Writes a human-readable string that identifies the low-level driver and a particular instance of a driven hardware device. For example, if the low-level driver provides its own additional device files for extra functionality not present in the abstraction layer, this routine might emit the name of the driver module and the names (or device numbers) of the low-level driver's |

own character special device files. This implements the `provider` attribute of the abstraction layer.

| | |
|---|---|
| get_quantum | Returns an `unsigned long` that represents the granularity to which the interrupt output repetition period can be set, in nanoseconds. This implements the `quantum` attribute of the abstraction layer. |
| get_source | Writes the current interrupt input source into the single-page-sized buffer passed as the second argument and returns the length of the written string. This implements the `source` attribute of the abstraction layer. |
| set_source | Reads the source specified in the buffer (passed as the second argument and as sized by the third) and returns the number of characters consumed or a negative error number in event of failure. It also causes the input source to be selected as requested. This implements the `source` attribute of the abstraction layer. |
| get_sourcelist | Writes strings representing the available interrupt input sources into the single-page-sized buffer passed as the second argument, one source per line. It returns the number of bytes written into this buffer. This implements the `sourcelist` attribute of the abstraction layer. |

**When an External Interrupt Occurs**

When an external interrupt signal triggers an interrupt that is handled by the low-level driver, the driver should make the following call:

```
void
extint_interrupt(struct extint_device *ed);
```

This allows the abstraction layer to perform any appropriate abstracted actions, such as update the interrupt count or trigger `poll/select` actions. The sole argument is the `struct extint_device` that was returned from the registration call.

**Driver Deregistration**

When the driver desires to deregister a particular device previously registered with the abstraction layer, it should make the following call:

```
void
extint_device_unregister(struct extint_device *ed);
```

The sole argument is the `struct extint_device` that was returned from the registration call. There is no error return from this call, but if invalid data is passed to it, the likelihood of a kernel panic is very high.

## Interrupt Notification Interface

In addition to the user-visible aspects of the external interrupt abstraction layer, there is a kernel-only interface available for interrupt notification. This interface provides the ability for other kernel modules to register a callout to be invoked whenever an external interrupt is ingested for a particular device.

This section discusses the following:

- "Callout Mechanism" on page 28
- "Callout Registration" on page 29
- "Callout Deregistration" on page 30

**Callout Mechanism**

For systems (not just applications) that are critically interested in responding as quickly as possible to an externally triggered event, waiting for a poll/select operation, or even busy-waiting on the value of the interrupt counter to change, may have unexpected harmful effects (such as tying up a CPU spinning on a value) or may not provide appropriate response times.

A callout mechanism lets you write your own kernel module in order to gain minimal-latency notification of events and react accordingly. It also provides an extension capability that might be of interest in certain situations. For example, there could be an application that requires an interrupt counter page similar to the one maintained by the abstraction layer, but that starts counting at 0 when the device special file is opened. Or, there could be an application that requires a signal to be generated and delivered to the process when an interrupt is ingested. These examples

are more esoteric than the simple counter page, and are best provided by a separate module rather than cluttering the main external interrupt abstraction code.

**Callout Registration**

To register a callout to be invoked upon interrupt ingest, allocate a `struct extint_callout`, fill it in, and pass it to the following call:

```
int
extint_callout_register(struct extint_device *ed,
                        struct extint_callout *ec);
```

The first argument is the `struct extint_device` corresponding to the particular abstracted external interrupt hardware device of interest. How this structure is found is up to the caller; however, the `file_to_extint_device` function will convert a `struct file` pointer to a `struct extint_device` pointer. This function will return `-EINVAL` if an inappropriate file descriptor is passed to it.

The second argument is one of the following structures:

```
struct extint_callout {
        struct module* owner;
        void (*function)(void *);
        void *data;
};
```

**Note:** Additional fields not of interest to the external interrupt user may be present. You should include `/usr/local/include/extint.h` to acquire these structure definitions.

The `owner` field should be set to the module containing the function and data pointed to by the remaining fields.

The `function` pointer is a callout function that is to be invoked whenever an interrupt is ingested by the abstraction layer for the device of interest. The `data` field is the only argument passed to it; it is used opaquely and is provided solely for use by the caller. That is, the abstraction layer will invoke the following upon each interrupt of the specified device:

```
ec->function(data);
```

You can register multiple callouts for the same abstracted external interrupt device. They will be invoked in no guaranteed order, but will be invoked one at a time.

The interrupt counter will be incremented before the callouts are invoked, but before any signal/poll notifications occur.

The module specified by the `owner` field in the callout structure, as well as the module corresponding to the low-level external interrupt device driver, will have their reference counts increased by one until the callout is deregistered.

### Callout Deregistration

To remove a callout, call the following with the same arguments as provided during callout registration:

```
extern void
extint_callout_unregister(struct extint_device *ed,
                          struct extint_callout *ec);
```

You can remove both active and orphaned callouts in this manner with no distinction between the two.

The callout function must continue to be able to be invoked until the call to `extint_callout_unregister` completes.

## Making Use of Unsupported Hardware Device Capabilities

If your hardware device supports capabilities that are not provided for in the abstraction layer, you can do one of the following:

- Add a new attribute to the abstraction layer by modifying `struct extint_properties` to add appropriate interface routines and update any existing drivers as necessary.

- Have the low-level driver create its own device class and corresponding attributes and/or character special devices. This method is preferred and is required if the capability is dependent on the hardware in a method that cannot be abstracted.

For example, the SGI IOC4 has the ability to map the interrupt output control register directly into a user application to avoid the kernel overhead of reading/writing the abstracted attribute files. Using this capability means that the application must have intimate knowledge of the format of the control register, something that cannot be abstracted away by the kernel and is very specific to this particular I/O controller chip. This capability is provided by the `ioc4_extint` driver, which supplys its own character special device along with an `ioc4_intout` device class.

## Low-level Driver Template

You can use the `ioc4_extint.c` file as a template for a low-level driver. The file is shipped as part of the `extint` source RPM.

**Note:** In addition to providing the abstraction interface, this low-level driver creates an IOC4-specific character special device and an IOC4-specific device class.

## Example: SGI IOC4 PCI Device

This section describes the following for the SGI IOC4 PCI device:

- "Multiple Independent Drivers" on page 31

- "External Interrupt Output" on page 32

- "External Interrupt Ingest" on page 34

- "Physical Interfaces" on page 35

For more information, see the `Documentation/sgi-ioc4.txt` file, which is installed with the Linux source code corresponding to the real-time kernel.

## Multiple Independent Drivers

The IOC4 external interrupt driver is not a typical PCI device driver. Due to certain design features of the IOC4 controller, typical PCI probing and removal functions are not appropriate. Instead, the IOC4 external interrupt driver interfaces with a core IOC4 driver that takes care of the usual PCI-level driver functionality. (An overview is provided below; for more details, see the `Documentation/sgi-ioc4.txt` file in the kernel source code.) However, the IOC4 external interrupt driver does interface very cleanly with the external interrupt abstraction layer, which is within the scope of the following discussion.

The IOC4 driver actually consists of the following independent drivers:

`ioc4`                    The core driver for IOC4. It is responsible for initializing the basic functionality of the chip and allocating the PCI resources that are shared between the IOC4 functions.

This driver also provides registration functions that the other IOC4 drivers can call to make their presence known. Each driver must provide a probe and a remove function, which are invoked by the core driver at appropriate times. The interface for the probe and remove operations is not precisely the same as PCI device probe and remove operations, but is logically the same operation.

sgiioc4
: The IDE driver for IOC4. It hooks up to the `ioc4` driver via the appropriate registration, probe, and remove functions.

ioc4_serial
: The serial driver for IOC4. It hooks up to the `ioc4` driver via the appropriate registration, probe, and remove functions.

ioc4_extint
: The external interrupts driver for IOC4.

IOC4-based I/O controller cards provide an electrical interface to the outside world that can be used to ingest and generate a simple signal for the following purposes:

- On the output side, one of the jacks can provide a small selection of output modes (low, high, a single strobe, toggling, and pulses at a specified interval) that create a 0-5V electrical output.

- On the input side, one of the jacks will cause the IOC4 to generate a PCI interrupt on the transition edge of an electrical signal.

This driver registers with the `extint` abstracted external interrupt driver and lets it take care of the user-facing details.

## External Interrupt Output

The output section provides several modes of output:

high
: Sets the output to logic `high`. The `high` state of the card's electrical output is actually a low voltage (0V).

low            Sets the output to logic `low`. The `low` state of the card's electrical
               output is actually a high voltage (+5V).

pulse          Sets the output to logic `high` for 3 ticks then returns to logic `low` for an
               interval configured by the `period` setting, then repeats. The mode is
               configurable by the abstraction layer device's `mode` attribute. Available
               modes can be found in the abstraction layer device's `modelist`
               attribute.

strobe         Sets the output to logic `high` for 3 ticks, then returns to logic `low`. A
               *tick* is the PCI clock signal divided by 520.

toggle         Alternates the output between logic `low` and logic `high` as configured
               by the `period` setting.

The `period` can be set to a range of values determined by the PCI clock speed of the
IOC4 device. For the `toggle` and `pulse` output modes, this period determines how
often the toggle or pulse occurs. The output period can be set only to a multiple of
this length (rounding will occur automatically in the driver). The `pulse` and `strobe`
output modes have a logic `high` pulse width equal to three ticks. The period should
be configurable by the abstraction layer device's `period` attribute, and the tick length
can be found from the abstraction layer device's `quantum` attribute.

**Note:** For reference, on a 66-MHz PCI bus, the tick length is 7.8 microseconds. On a
33-MHz PCI bus, the tick length is 15.6 microseconds. However, the IOC4 driver
calibrates itself to a more precise value than these somewhat coarse numbers,
depending on actual bus speed, which may vary slightly from bus to bus or even
reboot to reboot. However, IOC4 is only officially supported when running at 66-MHz.

One device file is provided, which can be memory mapped. The first 32-bit quantity
in the mapped area is aliased to the hardware register that controls output. Direct
manipulation of the register, both for reading and writing, may be performed in order
to avoid the kernel overhead that would be necessary if using the abstracted
interfaces. Assuming the typical `sysfs` mount point, the device number files for
these devices can be found at:

```
/sys/class/ioc4_intout/intout#/dev
```

This capability is not abstracted into the external interrupt abstraction layer because it
is critical for an application to know that this is an IOC4 device in order to determine
the format of the mapped register. Table 3-1 shows the register format.

**Table 3-1** Register Format

| Bits | Field | Read/Write Options | Description |
|------|-------|--------------------|-------------|
| 15:0 | COUNT | RW | Reloaded into the counter each time it reaches 0x0. The count period is actually (COUNT+1). |
| 18:16 | MODE | RW | Sets the mode for INT_OUT control:<br><br>• 000 loads a 0 to INT_OUT<br>• 100 loads a 1 to INT_OUT<br>• 101 pulses INT_OUT high for 3 ticks<br>• 110 pulses INT_OUT for 3 ticks every COUNT<br>• 111 toggles INT_OUT for 3 ticks every COUNT<br>• 001, 010, and 011 are undefined |
| 29:19 | (reserved) | RO | Read as 0, writes are ignored. |
| 30 | DIAG | RW | Bypass clock base divider. Operation when DIAG is set to a value of 1 is strictly unsupported. |
| 31 | INT_OUT | RO | Current state of INT_OUT signal. |

**Note:** There are the following considerations:

• The register should always be read and written as a 32-bit word in order to avoid concerns about big-endian and little-endian differences between the CPU and the IOC4 device.
• The /dev/intout# file may be memory-mapped only on kernels with a system page size of 16 KB or smaller. Due to technical constraints, it is not made available on kernels with a system page size larger than 16 KB.

## External Interrupt Ingest

The ingest section provides one control, the source of interrupt signals. The external source is a circuit connected to the external jack provided on IOC4-based I/O controller cards. The loopback source is the output of the IOC4's interrupt output section. The source is configurable by the abstraction layer device's source attribute. You can find available sources in the abstraction layer device's sourcelist attribute.

For example, to set up loopback mode:

```
[root@linux root]# echo loopback >/sys/class/extint/extint0/source
[root@linux root]# echo 100000000 >/sys/class/extint/extint0/period
[root@linux root]# echo toggle >/sys/class/extint/extint0/mode
```

**Note:** The IO10 card does not provide the 1/8–inch stereo connector interface for external interrupts, and thus can only use loopback as its source.

## Physical Interfaces

Use a two-conductor shielded cable to connect external interrupt output and input, with the two cable conductors wired to the +5V and interrupt conductors and the sleeves connected to the cable shield at both ends to maintain EMI integrity.

All IOC4-based external interrupt implementations use female 1/8-inch audio jacks. The wiring for the input jack is as follows:

- Tip: +5V input

- Ring: interrupt input (active low, optoisolated)

- Sleeve: chassis ground/cable shield

The input signal passes through an optoisolator that has a damping effect. The input signal must be of sufficient duration to drive the output of the optoisolator low in order for the interrupt to be recognized by the receiving machine. Current experimentation shows that the threshold is about 2.5 microseconds. To be safe, the driver sets its default outgoing pulse width to 10 microseconds. Any hardware not from SGI that is driving this line should do the same.

Figure 3-1 shows the internal driver circuit for the output connector and the internal receiver circuit for the input connector.
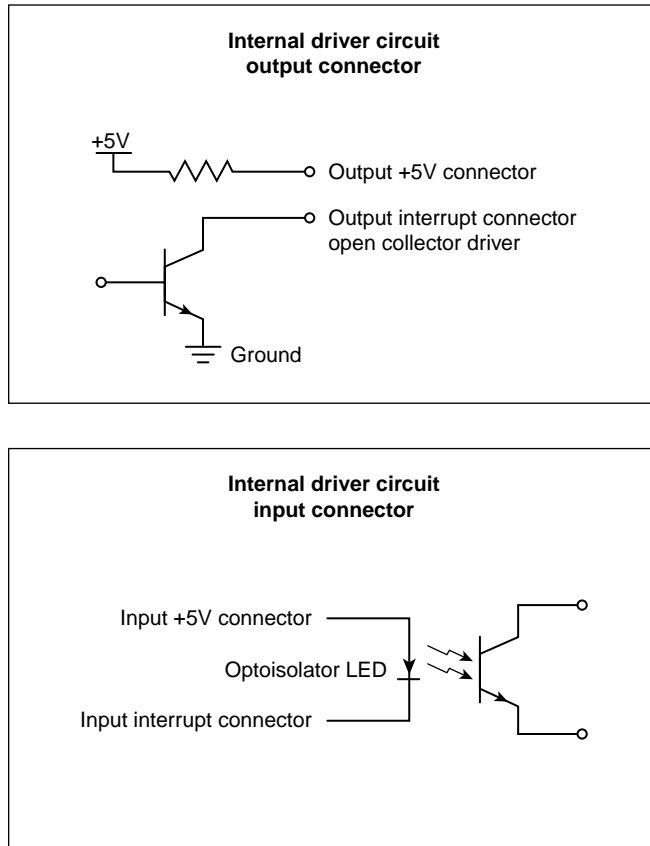
**Figure 3-1** Output and Input Connectors for the Internal Driver Circuit

You can wire an output connector directly to an input connector, taking care to connect the +5V output to the +5V input and the interrupt output to the interrupt input. If some other device is used to drive the input, it must be a +5V-source current limited with a 420–ohm resistor in series in order to avoid damaging the optoisolator.

# CPU Workload

This chapter describes how to use Linux kernel features to make the execution of a real-time program predictable. Each of these features works in some way to dedicate hardware to your program's use, or to reduce the influence of unplanned interrupts on it:

- "Using Priorities and Scheduling Queues" on page 37
- "Minimizing Overhead Work" on page 45
- "Understanding Interrupt Response Time" on page 50
- "Minimizing Interrupt Response Time" on page 53

## Using Priorities and Scheduling Queues

The default Linux scheduling algorithm is designed for a conventional time-sharing system. It also offers additional real-time scheduling disciplines that are better-suited to certain real-time applications.

This section discusses the following:

- "Scheduling Concepts" on page 37
- "kbar Kernel Barrier Facility" on page 39
- "Scheduling Capabilities" on page 42
- "Setting Pthread Priority" on page 43
- "Controlling Kernel and User Threads" on page 44

### Scheduling Concepts

In order to understand the differences between scheduling methods, you must understand the following basic concepts:

- "Timer Interrupts" on page 38
- "Real-Time Priority Band" on page 38

For information about time slices and changing the time-slice duration, see the information about the CPU scheduler in the *Linux Configuration and Operations Guide*.

**Timer Interrupts**

In normal operation, the kernel pauses to make scheduling decisions every several millisecond (ms) in every CPU. You can determine the frequency of this interval with the `sysconf(_SC_CLK_TCK)` function (see "Clocks" on page 12). Every CPU is normally interrupted by a timer every timer interval. (However, the CPUs in a multiprocessor are not necessarily synchronized. Different CPUs may take timer interrupts at different times.)

During the timer interrupt, the kernel updates accounting values, does other housekeeping work, and chooses which process to run next—usually the interrupted process, unless a process of superior priority has become ready to run. The timer interrupt is the mechanism that makes Linux scheduling preemptive; that is, it is the mechanism that allows a high-priority process to take a CPU away from a lower-priority process.

Before the kernel returns to the chosen process, it checks for pending signals and may divert the process into a signal handler.

**Real-Time Priority Band**

A real-time thread can select one of a range of 99 priorities (1-99) in the real-time priority band, using POSIX interfaces `sched_setparam()` or `sched_setscheduler()`. The higher the numeric value of the priority, the more important the thread. For more information, see the `sched_setscheduler`(2) man page.

Many soft real-time applications must execute ahead of time-share applications, so a lower priority range is best suited. Because time-share applications are scheduled at lower priority than real-time applications, a thread running at the lowest real-time priority (1) still executes ahead of all time-share applications.

**Note:** Applications cannot depend on system services if they are running ahead of system threads without observing system responsiveness timing guidelines.

Within a program it is usually best to follow the principles of *rate-monotonic scheduling*. However, you can use the following list as a guideline for selecting scheduling priorities in order to coordinate among different programs:

| Priority | Description |
| --- | --- |
| 99 | Reserved for critical kernel threads and should not be used by applications (99 is the highest real-time priority) |
| 90 - 98 | Hard real-time user threads |
| 60 - 89 | High-priority operating system services |
| 40 - 59 | Firm real-time user threads |
| 31 - 39 | Low-priority operating system services |
| 1 - 30 | Soft real-time user threads |

Real-time users can use tools such as strace(1) and ps(1) to observe the actual priorities and dynamic behaviors.

## `kbar` Kernel Barrier Facility

**Note:** This section does not apply to Altix XE servers.

The kbar(3) kernel barrier facility provides for the fast wake-up of many blocked user threads. When the barrier is signaled, the operating system will use a configurable number of CPUs to quickly wake all blocked threads. A maximum of 64 barriers are supported system-wide.

**Note:** The barriers do not behave precisely as traditional barriers. A specific number of threads do not have to be blocked on the barrier for them to be woken.

To use kbar, you must load the kbar Linux kernel module into the Linux kernel. To do this, enter the following as root:

```
[root@linux root]# modprobe kbar
```

Synopsis:

```
#include <bitmask.h>
#include <sn/kbar.h>
```

```
link with -lkbar

int kbar_open(struct bitmask * mask, int discipline);
int kbar_close(int fd);
int kbar_wait(int fd);
int kbar_signal(int fd);
```

where:

- `kbar_open` creates a kernel-supported barrier. It takes as its arguments a per-CPU bitmask of the CPUs that it should use to wake blocked threads and an assignment discipline. You can generate these bitmasks by using `bitmask_routines` available in `<bitmask.h>`, which ships as part of SGI ProPack for Linux.

  The assignment discipline can be one of the following:

  - `KBAR_LOCAL`, which tries to wake threads with worker CPUs near where they last ran

  - `KBAR_BALANCED`, which tries to balance the waking of threads across worker CPUs

  `kbar_open` returns a file descriptor with which all future interactions with that barrier will be made. If any other process is given access to that file descriptor (by such means as fork or interprocess communication), it can also take part in the barrier.

  On error, `kbar_open` returns `-1` and sets `errno` to one of the following:

  | | |
  |---|---|
  | EFAULT | There was a memory error in accessing the bitmask argument |
  | EINVAL | There was an error with the configuration arguments |
  | ENOENT | The `kbar` Linux kernel module is not loaded |
  | ENOMEM | There was not enough memory to create the barrier |
  | ENOSPC | The maximum system-wide number of barriers have already been created |

- `kbar_close` removes access to a barrier previously allocated by `kbar_open`. If the file descriptor has not been duplicated or is the last copy, then the barrier is deallocated. If user threads are still blocked on the barrier when `kbar_close` is called, the barrier will not be destroyed until they are all interrupted. Further access to the barrier through that file descriptor will no longer be possible.

On error, `kbar_close` returns -1 and sets `errno` to the following:

EBADF        fd is not a valid open file descriptor

- `kbar_wait` causes the calling thread to block on a barrier that was previously allocated by `kbar_open`. The thread will remain blocked until some other thread calls `kbar_signal` on the barrier or until it receives an unblocked signal.

  On error, `kbar_wait` returns -1 and sets `errno` to the following:

  EINTR        The thread was interrupted by a signal

- `kbar_signal` causes the operating system to use the previously designated helper CPUs to wake all the threads blocked on the barrier at the current time.

  On error, `kbar_signal` returns -1 and sets `errno` to the following:

  EBUSY        Another thread is in the process of waking the barrier's threads

Figure 4-1 shows the flow of the `kbar_wait` and `kbar_signal` functions. Threads block on the barrier with `kbar_wait` and are later scheduled again with `kbar_signal`.
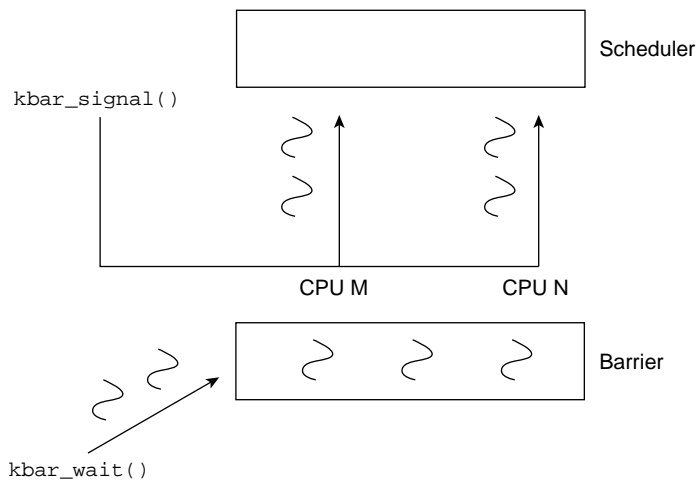


**Figure 4-1** `kbar`(3) Kernel Barrier Facility

For examples, see the `/usr/share/react/kbar/examples/` directory.

## Scheduling Capabilities

The capabilities mechanism provides fine-grained control over the privileges of a process. As a process attribute, a capability allows the process to perform a specific set of restricted operations without granting general override of the system's protection scheme.

A process can possess multiple capabilities. Collectively, all defined capabilities comprise the set of abilities that are traditionally associated with the `root` user. As of the 2.6.14 Linux kernel, there are 31 capabilities implemented, each defined by a bit within the capabilities bitmap. For specifics on implemented capabilities and the individual capability definitions, see the `capabilities`(7) manpage.

Each process has three capability sets containing up to 31 capabilities:

* *Effective set*: the capabilities used by the kernel to perform permission checks for the process.

* *Inheritable set*: the capabilities preserved across an `execve`(2).

* *Permitted set*: the capabilities that the process may assume (that is, a limiting superset for the effective and inheritable sets). If a process drops a capability from its permitted set, it can never reacquire that capability unless it uses `exec`(2) to execute the program that sets the user ID to `root`.

A child created via `fork`(2) inherits copies of its parent's capability sets.

The Linux implementation of capabilities is limited to processes. It does not have the ability to attach capabilities to an executable file, so that a process inherits those capabilities when the file is run via `exec()`. Because the implementation does not support file capability sets, note the following during an `exec()`:

* All three file capability sets are initially assumed to be cleared.

* If a program that sets the user ID to `root` is being executed or if the real user ID of the process is `0` (meaning `root`), then the file inheritable and permitted set capabilities are enabled (that is, each capability is defined to be `1`).

* If a program that sets the user ID to `root` is being executed, then the file effective set is enabled (that is, each capability is defined to be `1`).

Some specific uses of capabilities in the context of real-time applications may involve the following capabilities:

CAP_IPC_LOCK        Permits memory locking. See the mlock(2), mlockall(2), mmap(2), and shmctl(2) man pages.

CAP_SYS_NICE        Allows the following:

- Raising the process nice value. See the nice(2) and setpriority(2) man pages.

- Setting of real-time scheduling policies for calling process and setting scheduling policies and priorities for arbitrary processes. See the sched_setscheduler(2) and sched_setparam(2) man pages.

- Using FIFO and round-robin (real-time) scheduling on processes owned by this process and setting the scheduling algorithm used by another process.

CAP_SYS_RESOURCE        Allows for greater than 64–Hz interrupts from the real-time clock.

CAP_SYS_TIME        Allows modification of the real-time (hardware) clock and the modification of the system clock. See the settimeofday(2), stime(2), and adjtimex(2) man pages.

To access process capabilities and facilitate manipulation of capabilities from within a program, use the capget and capset system calls. To set capabilities, see the For details, see the capget(2), capset(2), and capabilities(7) man pages.

## Setting Pthread Priority

The Linux pthreads library shipped with 2.6 Linux is known as the *new pthreads library (NPTL)*. By default, a newly created pthread receives its priority from the same scheduling policy and scheduling priority as the pthread that created it; new pthreads will ignore the values in the attributes structure.

You can set the priority and scheduling policy of pthreads as follows:

- To change a running pthread, the pthread must call `pthread_setschedparam()`.

- To set the scheduling attributes that a pthread will start with when it is created, use the `pthread_attr_setschedpolicy()` and `pthread_attr_setschedparam()` library calls to configure the attributes structure that will later be passed to `pthread_create()`.

  The `pthread_attr_setinheritsched()` library call acts on the `pthread_attr_t` structure that will later be passed to `pthread_create()`. You can configure it with one of the following settings:

  - `PTHREAD_EXPLICIT_SCHED` causes pthreads to use the scheduling values set in the structure

  - `PTHREAD_INHERIT_SCHED` causes pthreads to inherit the scheduling values from their parent pthread

## Controlling Kernel and User Threads

In some situations, kernel threads and user threads must run on specific processors or with other special behavior. Most user threads and a number of kernel threads do not require any specific CPU or node affinity, and therefore can run on a select set of nodes. The SGI ProPack bootcpuset feature controls the placement of both kernel and user threads that do not require any specific CPU or node affinity. By placing these threads out of the way of your time-critical application threads, you can minimize interference from various external events.

As an example, an application might have two time-critical interrupt servicing threads, one per CPU, running on a four-processor machine. You could set up CPUs 0 and 1 as a bootcpuset and then run the time-critical threads on CPUs 2 and 3.

**Note:** You must have the SGI `cpuset-*.rpm` RPM installed to use bootcpusets. For configuration information, see the `bootcpuset`(8) man page.

You must use the `reactconf.pl` configuration script for this procedure; see Chapter 9, "REACT System Configuration" on page 105.

# Minimizing Overhead Work

A certain amount of CPU time must be spent on general housekeeping. Because this work is done by the kernel and triggered by interrupts, it can interfere with the operation of a real-time process. However, you can remove almost all such work from designated CPUs, leaving them free for real-time work.

First decide how many CPUs are required to run your real-time application. Then apply the following steps to isolate and restrict those CPUs:

- "Avoid the Clock Processor (CPU 0)" on page 45

- "Reduce the System Flush Duration" on page 45

- "Redirect Interrupts" on page 46

- "Select the Console Node for SAL Console Driver Interrupt" on page 46

- "Restrict, Isolate, and Shield CPUs" on page 47

- "Avoid Kernel Module Insertion and Removal" on page 49

- "Avoid Filesystem Mounts" on page 49

**Note:** The steps are independent of each other, but each must be done to completely free a CPU.

## Avoid the Clock Processor (CPU 0)

Every CPU takes a timer interrupt that is the basis of process scheduling. However, CPU 0 does additional housekeeping for the whole system on each of its timer interrupts. Real-time users are therefore advised not to use CPU 0 for running real-time processes.

## Reduce the System Flush Duration

In SGI systems running Linux, the scalable hub (SHub) ASIC is responsible for memory transactions with the processor front-side bus. Periodically, the SHub initiates a system flush, which can impact real-time performance. The system flush duration by default is set to a value appropriate for more general purpose computing, and this default value can interfere with extremely time-sensitive threads that require interrupt response times measured in microseconds. You can set the system flush

duration to a value appropriate for real-time applications by following step 7 in Chapter 9, "REACT System Configuration" on page 105.

## Redirect Interrupts

To minimize latency of real-time interrupts, it is often necessary to direct them to specific real-time processors. It is also necessary to direct other interrupts away from specific real-time processors. This process is called *interrupt redirection*.

The reactconf.pl script will assist you in redirecting interrupts. The process involves writing a bitmask to the /proc/irq/*interruptnumber*/smp_affinity file, which shows a bitmask of the CPUs that are allowed to receive this interrupt. A 1 in the least-significant bit in this mask denotes that CPU 0 is allowed to receive the interrupt. The most-significant bit denotes the highest-possible CPU that the booted kernel could support.

For example, to redirect interrupt 62 to CPU 1, enter the following:

```
[root@linux root]# echo 1 > /proc/irq/62/smp_affinity
```

You can examine the /proc/interrupts file to discover where interrupts are being received on your system.

An example of how to redirect interrupts is demonstrated by the reactconf.pl configuration script. SGI recommends that someone with knowledge of the system configuration use this script to redirect only the interrupts that must be moved. For more information, see Chapter 9, "REACT System Configuration" on page 105.

## Select the Console Node for SAL Console Driver Interrupt

**Note:** This section does not apply to Altix XE servers.

The console node you select for the system abstraction layer (SAL) console driver interrupts depends upon whether your system has an L2 system controller or not:

- If your system has an L2 controller, the SAL console driver interrupt will always appear on a CPU on the first node.

- If your system does not have an L2 controller, the SAL console driver generates interrupts that will be directed toward a single CPU on the node where the console is attached. SGI recommends that you attach the console to a node that

will not be used for time-critical threads. Because the clock processor always runs on CPU 0, SGI recommends that you use node 0 as the console node.

**Note:** You cannot select which CPU on the console node will receive interrupts.

For more information, see the *SGI L1 and L2 Controller Software User's Guide*.

## Restrict, Isolate, and Shield CPUs

In general, the Linux scheduling algorithms run a process that is ready to run on any CPU. For best performance of a real-time process or for minimum interrupt response time, you must use one or more CPUs without competition from other scheduled processes. You can exert the following levels of increasing control:

- *Restricted*, which prevents the CPU from running scheduled processes.

- *Isolated*, which removes the CPU from load balancing considerations, a time-consuming scheduler operation.

- *Shielded*, which switches off the timer (scheduler) interrupts that would normally be scheduled on the CPU. These are a source of jitter, but only a minor source of interrupt response latency. Shielding should only be done for short periods where basically jitter-free program execution is required.

**Note:** Altix XE servers do not support the ability to shield a CPU from timer interrupts.

Generally, you would want to prepare a CPU for time-critical real-time operation by both restricting and isolating it. You should use the reactconf.pl configuration script to perform the steps required to restrict and isolate a CPU. For more information, see Chapter 9, "REACT System Configuration" on page 105.

### Restricting a CPU from Scheduled Work

You can restrict one or more CPUs from running scheduled processes. The only processes that can use a restricted CPU are those processes that you assign to it, along with certain per-CPU kernel threads.

To restrict one or more CPUs, use the reactconf.pl configuration script documented in Chapter 9, "REACT System Configuration" on page 105

After restricting a CPU, you can assign processes to it using the SGI `cpuset` command. See "Running a Process on a Restricted CPU" on page 116.

To remove the CPU restriction, allowing the CPU to execute any scheduled process, rerun the `reactconf.pl` command with a different set of CPUs or use the procedure in "Disabling REACT" on page 117.

### Isolating a CPU from Scheduler Load Balancing

You can isolate a CPU so that it is not subject to the effects of scheduler load balancing. Isolating a CPU removes one source of unpredictable delays from a real-time program and helps further minimize the latency of interrupt handling.

To isolate one or more CPUs, use the `reactconf.pl` configuration script documented in Chapter 9, "REACT System Configuration" on page 105. Each `rtcpu` is set to be `cpu_exclusive`.

Normally, you would also want to restrict the isolated CPUs. See "Restricting a CPU from Scheduled Work" on page 47.

### Shielding a CPU from Timer Interrupts

**Note:** This section does not apply to Altix XE servers.

You can shield a CPU from the normally scheduled Linux timer (scheduler) interrupts. For more information on timer interrupts, see "Timer Interrupts" on page 38.

Timer interrupts are a source of interrupt response latency (usually several usec). Shielding is done dynamically from program control, and should only be done for short periods where essentially jitter-free program execution is required.

**Note:** Prolonged periods of shielding might eventually result in system resource depletion.

System resource depletion usually takes the form of out-of-memory conditions, eventually causing forced shutdown of the application. The kernel ring buffer will indicate this situation by showing a stack trace for the application and a `No available memory in cpuset:` message. To view the kernel ring buffer, run the `dmesg` command.

To shield a CPU from timer interrupts, do the following:

1. Load the `sgi-shield` kernel module.

   For example:

   ```
   [root@linux root]# modprobe sgi-shield
   ```

2. From your application, call the `cpu_shield()` function with the `SHIELD_STOP_INTR` flag and the desired CPU number. Your program must link in the `sgirt` library to access the `cpu_shield()` function. For more information, see the `libsgirt`(3) man page.

   For example, to switch off timer interrupts on CPU 3, perform the following function call from the application:

   ```
   cpu_shield(SHIELD_STOP_INTR, 3)
   ```

To unshield the CPU, call the `cpu_shield()` function with the `SHIELD_START_INTR` flag and the desired CPU number.

For example, when shielding CPU 3 is no longer necessary, perform the following call from the application:

```
cpu_shield(SHIELD_START_INTR, 3)
```

## Avoid Kernel Module Insertion and Removal

The insertion and removal of Linux kernel modules (such as by using `modprobe` or `insmod`/`rmmod`) requires that a kernel thread be started on all active CPUs (including isolated CPUs) in order to synchronously stop them. This process allows safe lockless-module list manipulation. However, these kernel threads can interfere with thread wakeup and, for brief periods, the ability to receive interrupts.

While a time-critical application is running, you must avoid Linux kernel module insertion and removal. All necessary system services should be running prior to starting time-critical applications.

## Avoid Filesystem Mounts

The process of mounting of a filesystem can interfere with response times for a number of CPUs. These delays do not happen after the mount has completed. There is no delay for disk accesses.

Prior to running a time-critical application, you should complete all filesystem mounts that may be necessary during application execution.

# Understanding Interrupt Response Time

*Interrupt response time* is the time that passes between the instant when a hardware device raises an interrupt signal and the instant when (interrupt service completed) the system returns control to a user process. SGI guarantees a maximum interrupt response time on certain systems, but you must configure the system properly in order to realize the guaranteed time.

## Maximum Response Time Guarantee

In properly configured systems, interrupt response time is guaranteed not to exceed 30 microseconds (usecs) for SGI systems running Linux.

This guarantee is important to a real-time program because it puts an upper bound on the overhead of servicing interrupts from real-time devices. You should have some idea of the number of interrupts that will arrive per second. Multiplying this by 30 usecs yields a conservative estimate of the amount of time in any one second devoted to interrupt handling in the CPU that receives the interrupts. The remaining time is available to your real-time application in that CPU.

## Components of Interrupt Response Time

The total interrupt response time includes the following sequential parts:

*Hardware latency*            The time required to make a CPU respond to an interrupt signal.

*Software latency*            The time required to dispatch an interrupt thread.

*Device service time*         The time the device driver spends processing the interrupt and dispatching a user thread.

*Mode switch*                 The time it takes for a thread to switch from kernel mode to user mode.

Figure 4-2 diagrams the parts discussed in the following sections.
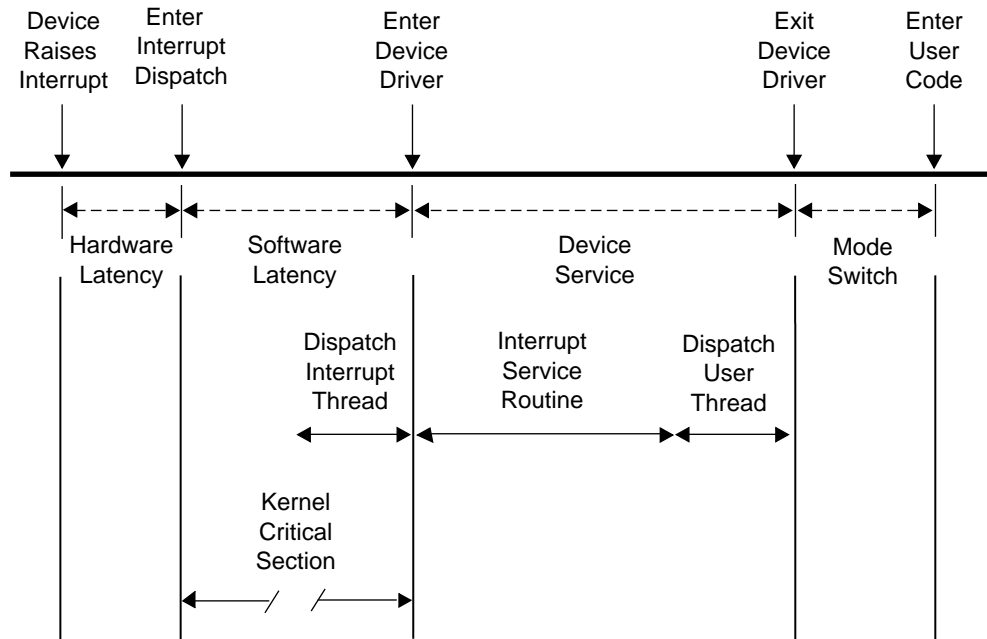
**Figure 4-2** Components of Interrupt Response Time

**Hardware Latency**

When an I/O device requests an interrupt, it activates a line in the PCI bus interface. The bus adapter chip places an interrupt request on the system internal bus and a CPU accepts the interrupt request.

The time taken for these events is the hardware latency, or *interrupt propagation delay*.

For more information, see Chapter 7, "PCI Devices" on page 93.

**Software Latency**

Software latency is affected by the following:

- "Kernel Critical Sections" on page 52
- "Interrupt Threads Dispatch" on page 52

### Kernel Critical Sections

Certain sections of kernel code depend on exclusive access to shared resources. Spin locks are used to control access to these critical sections. Once in a critical section, interrupts are disabled. New interrupts are not serviced until the critical section is complete.

There is no guarantee on the length of kernel critical sections. In order to achieve 30-usec response time, your real-time program must avoid executing system calls on the CPU where interrupts are handled. The way to ensure this is to restrict that CPU from running normal processes. For more information, see "Restricting a CPU from Scheduled Work" on page 47 and "Isolating a CPU from Scheduler Load Balancing" on page 48.

You may need to dedicate a CPU to handling interrupts. However, if the interrupt-handling CPU has power well above that required to service interrupts (and if your real-time process can tolerate interruptions for interrupt service), you can use the restricted CPU to execute real-time processes. If you do this, the processes that use the CPU must avoid system calls that do I/O or allocate resources, such as `fork()`, `brk()`, or `mmap()`. The processes must also avoid generating external interrupts with long pulse widths.

In general, processes in a CPU that services time-critical interrupts should avoid all system calls except those for interprocess communication and for memory allocation within an arena of fixed size.

### Interrupt Threads Dispatch

The primary function of interrupt dispatch is to determine which device triggered the interrupt and dispatch the corresponding interrupt thread. Interrupt threads are responsible for calling the device driver and executing its interrupt service routine.

While the interrupt dispatch is executing, all interrupts at or below the current interrupt's level are masked until it completes. Any pending interrupts are dispatched before interrupt threads execute. Thus, the handling of an interrupt could be delayed by one or more devices.

In order to achieve 30-usec response time on a CPU, you must ensure that the time-critical devices supply the only device interrupts directed to that CPU. For more information, see "Redirect Interrupts" on page 46.

**Device Service Time**

Device service time is affected by the following:

- "Interrupt Service Routines"

- "User Threads Dispatch"

### Interrupt Service Routines

The time spent servicing an interrupt should be negligible. The interrupt handler should do very little processing; it should only wake up a sleeping user process and possibly start another device operation. Time-consuming operations such as allocating buffers or locking down buffer pages should be done in the request entry points for `read()`, `write()`, or `ioctl()`. When this is the case, device service time is minimal.

### User Threads Dispatch

Typically, the result of the interrupt is to make a sleeping thread runnable. The runnable thread is entered in one of the scheduler queues. This work may be done while still within the interrupt handler.

**Mode Switch**

A number of instructions are required to exit kernel mode and resume execution of the user thread. Among other things, this is the time when the kernel looks for software signals addressed to this process and redirects control to the signal handler. If a signal handler is to be entered, the kernel might have to extend the size of the stack segment. (This cannot happen if the stack was extended before it was locked.)

## Minimizing Interrupt Response Time

You can ensure interrupt response time of 30 usecs or less for one specified device interrupt on a given CPU provided that you configure the system as follows:

- The CPU does not receive any other `SN hub` device interrupts

- The interrupt is handled by a device driver from a source that promises negligible processing time

- The CPU is isolated from the effects of load balancing

- The CPU is restricted from executing general Linux processes

- Any process you assign to the CPU avoids system calls other than interprocess communication and allocation within an arena

- Kernel module insertion and removal is avoided

When these things are done, interrupts are serviced in minimal time.

# Using the Frame Scheduler

---

**Note:** This chapter does not yet apply to Altix XE.

---

The frame scheduler makes it easy to structure a real-time program as a family of independent, cooperating activities that are running on multiple CPUs and are scheduled in sequence at the frame rate of the application.

This chapter discusses the following:

- "Frame Scheduler Concepts" on page 55

- "Selecting a Time Base" on page 70

- "Using the Scheduling Disciplines" on page 72

- "Using Multiple Consecutive Minor Frames" on page 74

- "Designing an Application for the Frame Scheduler" on page 76

- "Preparing the System" on page 77

- "Implementing a Single Frame Scheduler" on page 78

- "Implementing Synchronized Schedulers" on page 79

- "Handling Frame Scheduler Exceptions" on page 81

- "Using Signals Under the Frame Scheduler" on page 86

- "Using Timers with the Frame Scheduler" on page 89

## Frame Scheduler Concepts

One frame scheduler dispatches selected threads at a real-time rate on one CPU. You can also create multiple, synchronized frame schedulers that dispatch concurrent threads on multiple CPUs.

This section discusses the following:

- "Frame Scheduler Basics" on page 56

## Frame Scheduler Basics

When a frame scheduler dispatches threads on one CPU, it does not completely supersede the operation of the normal Linux scheduler. The CPUs chosen for frame scheduling must be restricted and isolated (see "Restrict, Isolate, and Shield CPUs" on page 47). You do not have to set up cpusets for the frame scheduled CPUs because the frame scheduler will set up cpusets named `rtcpu`*N* (wherer *N* is the CPU number) if this has not already been done. For more control over cpuset parameters, you can create your own cpusets for the frame scheduler to use (one per CPU, and one CPU per cpuset), by naming them exactly as mentioned above.

If you already have cpusets named `rtcpu`*N*, but they include other than only the CPU number in question, the frame scheduler will return an `EEXIST` error.

**Note:** REACT for Linux does not support Vsync, device-driver, or system-call time bases.

For more information, see "Isolating a CPU from Scheduler Load Balancing" on page 48 and "Preparing the System" on page 77.

## Thread Programming Model

The frame scheduler supports pthreads.

In this guide, a *thread* is defined as an independent flow of execution that consists of a set of registers (including a program counter and a stack). A *pthread* is defined by the POSIX standard. Pthreads within a process use the same global address space.

A traditional Linux process has a single active thread that starts once the program is executed and runs until the program terminates. A multithreaded process may have several threads active at one time. Hence, a process can be viewed as a receptacle that contains the threads of execution and the resources they share (that is, data segments, text segments, file descriptors, synchronizers, and so forth).

## Frame Scheduling

Instead of scheduling threads according to priorities, the frame scheduler dispatches them according to a strict, cyclic rotation governed by a repetitive time base. The time base determines the fundamental frame rate. (See "Selecting a Time Base" on page 70.) Some examples of the time base are as follows:

- A specific clocked interval in microseconds

- An external interrupt (see "External Interrupts as a Time Base" on page 71)

- The Vsync (vertical retrace) interrupt from the graphics subsystem

- A device interrupt from a specially modified device driver

- A system call (normally used for debugging)

**Note:** REACT for Linux does not support Vsync, device-driver, or system-call time bases.

The interrupts from the time base define *minor frames*. Together, a fixed number of minor frames make up a *major frame*. The length of a major frame defines the application's true frame rate. The minor frames allow you to divide a major frame into subframes. Figure 5-1 shows major and minor frames.
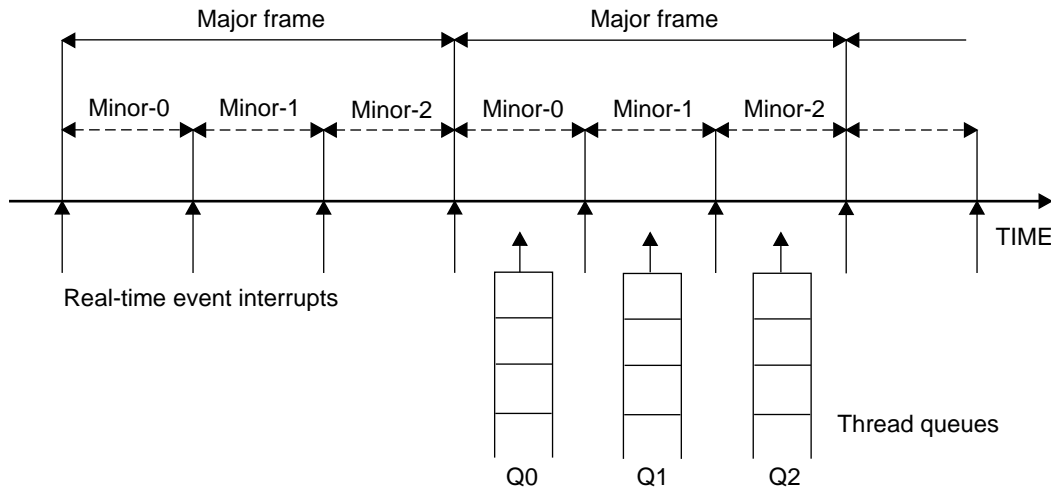
**Figure 5-1** Major and Minor Frames

In the simplest case, there is a single frame rate, such as 60 Hz, and every activity the program performs must be done once per frame. In this case, the major and minor frame rates are the same.

In other cases, there are some activities that must be done in every minor frame, but there are also activities that are done less often, such as in every other minor frame or in every third one. In these cases, you define the major frame so that its rate is the rate of the least-frequent activity. The major frame contains as many minor frames as necessary to schedule activities at their relative rates.

As pictured in Figure 5-1, the frame scheduler maintains a queue of threads for each minor frame. You must queue each activity thread of the program to a specific minor frame. You determine the order of cyclic execution within a minor frame by the order in which you queue threads. You can do the following:

- Queue multiple threads in one minor frame. They are run in the queued sequence within the frame. All must complete their work within the minor frame interval.

- Queue the same thread to run in more than one minor frame. For example, suppose that thread `double` is to run twice as often as thread `solo`. You would queue `double` to Q0 and Q2 in Figure 5-1, and queue `solo` to Q1.

- Queue a thread that takes more than a minor frame to complete its work. If thread `sloth` needs more than one minor interval, you would queue it to Q0, Q1, and Q2, such that it can continue working in all three minor frames until it completes.

- Queue a background thread that is allowed to run only when all others have completed, to use up any remaining time within a minor frame.

All of these options are controlled by scheduling disciplines you specify for each thread as you queue it. For more information, see "Using the Scheduling Disciplines" on page 72.

Typically, a frame scheduler is driven by a single interrupt source and contains minor frames having the same duration, but a variable frame scheduler may be used to implement a frame scheduler having multiple interrupt sources and/or minor frames of variable duration. For more information, see the `frs_create_vmaster()` function.

The relationship between threads and a frame scheduler depends upon the thread model in use:

- The pthread programming model requires that all threads scheduled by the frame scheduler reside in the same process.

- The `fork()` programming model does not require that the participating threads reside in the same process.

See "Implementing a Single Frame Scheduler" on page 78 for details.

## Controller Thread

The thread that creates a frame scheduler is called the *frame scheduler controller thread*. It is privileged in these respects:

- Its identifier is used to identify its frame scheduler in various functions. The frame scheduler controller thread uses a pthread ID.

- It can receive signals when errors are detected by the frame scheduler (see "Using Signals Under the Frame Scheduler" on page 86).

- It cannot itself be queued to the frame scheduler. It continues to be dispatched by Linux and executes on a CPU other than the one the frame scheduler uses.

## Frame Scheduler API

For an overview of the frame scheduler API, see the `frs`(3) man page, which provides a complete listing of all the frame scheduler functions. Separate man pages for each of the frame scheduler functions provide the API details. The API elements are declared in `/usr/include/frs.h`. The following are some important types that are declared in `/usr/include/frs.h`:

| | |
|---|---|
| `typedef frs_fsched_info_t` | A structure containing information about one scheduler (including its CPU number, interrupt source, and time base) and number of minor frames. Used when creating a frame scheduler. |
| `typedef frs_t` | A structure that identifies a frame scheduler. |
| `typedef frs_queue_info_t` | A structure containing information about one activity thread: the frame scheduler and minor frame it uses and its scheduling discipline. Used when enqueuing a thread. |
| `typedef frs_recv_info_t` | A structure containing error recovery options. |
| `typedef frs_intr_info_t` | A structure that `frs_create_vmaster()` uses for defining interrupt information templates (see Table 5-1 on page 62). |

Additionally, the pthreads interface adds the following types, as declared in `/usr/include/sys/pthread.h`:

| | |
|---|---|
| `typedef pthread_t` | An integer identifying the pthread ID. |
| `typedef pthread_attr_t` | A structure containing information about the attributes of the frame scheduler controller thread. |

## Interrupt Information Templates

Variable frame schedulers may drive each minor frame with a different interrupt source, as well as define a different duration for each minor frame. These two

characteristics may be used together or separately, and are defined using an interrupt information template.

An *interrupt information template* consists of an array of `frs_intr_info_t` data structures, where each element in the array represents a minor frame. For example, the first element in the array represents the interrupt information for the first minor frame, and so on for *n* minor frames.

The `frs_intr_info_t` data structure contains two fields for defining the interrupt source and its qualifier: `intr_source` and `intr_qualifier`.

The following example demonstrates how to define an interrupt information template for a frame scheduler having minor frames of different duration. Assume the application requires four minor frames, where each minor frame is triggered by the synchronized clock timer, and the duration of each minor frame is as follows: 100 ms, 150 ms, 200 ms, and 250 ms. The interrupt information template may be defined as follows:

```
frs_intr_info_t intr_info[4];
intr_info[0].intr_source    = FRS_INTRSOURCE_CCTIMER;
intr_info[0].intr_qualifier = 100000;
intr_info[1].intr_source    = FRS_INTRSOURCE_CCTIMER;
intr_info[1].intr_qualifier = 150000;
intr_info[2].intr_source    = FRS_INTRSOURCE_CCTIMER;
intr_info[2].intr_qualifier = 200000;
intr_info[3].intr_source    = FRS_INTRSOURCE_CCTIMER;
intr_info[3].intr_qualifier = 250000;
```

For detailed programming examples, demonstrating use of variable frame schedulers, see the `/usr/share/react/frs/examples` directory and the `frs_create_vmaster`(3) man page.

## Library Interface for C Programs

Table 5-1 summarizes the API library functions in the `/usr/lib/libfrs.a` file.

**Table 5-1** Frame Scheduler Operations

| Operation | Use | Frame Scheduler API |
|---|---|---|
| Create a frame scheduler | Process setup | frs_t* frs_create(*cpu*, (int int *intr_source* int *intr_qualifier*, int, *n_minors*, pid_t *sync_master_pid*, int*num_slaves*); |
| | Process or pthread setup | frs_t* frs_create_master(int *cpu*, int *intr_source*, int *intr_qualifier*, int *n_minors*, int *num_slaves*); |
| | Process or pthread setup | frs_t* frs_create_slave(int *cpu*, frs_t* *sync_master_frs*); |
| | Process or pthread setup | frs_t* frs_create_vmaster(int *cpu*, int *n_minors*, int *n_slaves*, frs_intr_info_t *\*intr_info*); |
| Queue to a frame scheduler minor frame | Process setup | int frs_enqueue(frs_t* *frs*, pid_t *pid*, int *minor_frame*, unsigned int *discipline*); |
| | Pthread setup | int frs_pthread_enqueue(frs_t* *frs*, pthread_t *pthread*, int *minor_frame*, unsigned int *discipline*); |
| Insert into a queue, possibly changing discipline | Process setup | int frs_pinsert(frs_t* *frs*, int *minor_frame*, pid_t *target_pid*, int *discipline*, pid_t *base_pid*); |
| | Pthread setup | int frs_pthread_insert(frs_t* *frs*, int *minor_index*, pthread_t *target_pthread*, int *discipline*, pthread_t *base_pthread*); |
| Set error recovery options | Process setup | int frs_setattr(frs_t* *frs*, int *minor_frame*, pid_t *pid*, frs_attr_t *attribute*, void* *param*); |
| | Pthread setup | int frs_pthread_setattr(frs_t* *frs*, int *minor_frame*, pthread_t *pthread*, frs_attr_t *attribute*, void* *param*); |
| Join a frame scheduler (activity is ready to start) | Process or pthread execution | int frs_join(frs_t* *frs*); |

| Operation | Use | Frame Scheduler API |
|---|---|---|
| Start scheduling (all activities queued) | Process or pthread execution | `int frs_start(frs_t* `*frs*`);` |
| Yield control after completing activity | Process or pthread execution | `int frs_yield(void);` |
| Pause scheduling at end of minor frame | Process or pthread execution | `int frs_stop(frs_t* `*frs*`);` |
| Resume scheduling at next time-base interrupt | Process or pthread execution | `int frs_resume(frs_t* `*frs*`);` |
| Trigger a user-level frame scheduler interrupt | Process or pthread execution | `int frs_userintr(frs_t* `*frs*`);` |
| Interrogate a minor frame queue | Process or pthread query | `int frs_getqueuelen(frs_t* `*frs*`, int `*minor_index*`);` |
|  | Process query | `int frs_readqueue(frs_t* `*frs*`, int `*minor_frame*`, pid_t *`*pidlist*`);` |
|  | Pthread query | `int frs_pthread_readqueue(frs_t* `*frs*`, int `*minor_frame*`, pthread_t *`*pthreadlist*`);` |
| Retrieve error recovery options | Process query | `int frs_getattr(frs_t* `*frs*`, int `*minor_frame*`, pid_t `*pid*`, frs_attr_t `*attribute*`, void* `*param*`);` |
|  | Pthread query | `int frs_pthread_getattr(frs_t* `*frs*`, int `*minor_frame*`, pthread_t `*pthread*`, frs_attr_t `*attribute*`, void* `*param*`);` |
| Destroy frame scheduler and send `SIGKILL` to its frame scheduler controller | Process or pthread teardown | `int frs_destroy(frs_t* `*frs*`);` |
| Remove a process or thread from a queue | Process teardown | `int frs_premove(frs_t* `*frs*`, int `*minor_frame*`, pid_t `*remove_pid*`);` |
|  | Pthread teardown | `int frs_pthread_remove(frs_t* `*frs*`, int `*minor_frame*`, pthread_t `*remove_pthread*`);` |
| Register a thread | Pthread setup | `int frs_pthread_register(void)` |

## Thread Execution

Example 5-1 shows the basic structure of an activity thread that is queued to a frame scheduler.

**Example 5-1** Skeleton of an Activity Thread

```
/* Initialize data structures etc. */
frs_join(scheduler-handle)
do
{
   /* Perform the activity. */
   frs_yield();
} while(1);
_exit();
```

When the thread is ready to start real-time execution, it calls `frs_join()`. This call blocks until all queued threads are ready and scheduling begins. When `frs_join()` returns, the thread is running in its first minor frame. For more information, see "Starting Multiple Schedulers" on page 68 and the `frs_join`(3) man page.

**Note:** Each thread of a pthreaded application (including the controller thread) must first call `frs_pthread_register()` before making any other calls to the frame scheduler. In addition, each activity thread must complete its call to `frs_pthread_register` before the controller thread calls `frs_pthread_enqueue`.

The thread then performs whatever activity is needed to complete the minor frame and calls `frs_yield()`. This gives up control of the CPU until the next minor frame where the thread is queued and executes. For more information, see the `frs_yield`(3) man page.

An activity thread is never preempted by the frame scheduler within a minor frame. As long as it yields before the end of the frame, it can do its assigned work without interruption from other activity threads. (However, it can be interrupted by hardware interrupts, if they are allowed in that CPU.) The frame scheduler preempts the thread at the end of the minor frame.

When a very short minor frame interval is used, it is possible for a thread to have an overrun error in its first frame due to cache misses. A simple variation on the basic structure shown in Example 5-1 is to spend the first minor frame touching a set of important data structures in order to "warm up" the cache. This is sketched in Example 5-2.

**Example 5-2** Alternate Skeleton of an Activity Thread

```
/* Initialize data structures etc. */
frs_join(scheduler-handle); /* Much time could pass here. */
/* First frame: merely touch important data structures. */
do
{
   frs_yield();
   /* Second and later frames: perform the activity. */
} while(1);
_exit();
```

When an activity thread is scheduled on more than one minor frame in a major frame, it can be designed to do nothing except warm up the cache in the entire first major frame. To do this, the activity thread function must know how many minor frames it is scheduled on and call `frs_yield()` a corresponding number of times in order to pass the first major frame.

## Scheduling Within a Minor Frame

Threads in a minor frame queue are dispatched in the order that they appear on the queue (priority is irrelevant). Queue ordering can be modified as follows:

- Appending a thread at the end of the queue with `frs_pthread_enqueue()` or `frs_enqueue()`

- Inserting a thread after a specific target thread via `frs_pthread_insert()` or `frs_pinsert()`

- Deleting a thread in the queue with `frs_pthread_remove()` or `frs_premove()`

See "Managing Activity Threads" on page 69 and the `frs_enqueue(3)`, `frs_pinsert(3)`, and `frs_premove(3)` man pages.

### Scheduler Flags `frs_run` and `frs_yield`

The frame scheduler keeps two status flags per queued thread: `frs_run` and `frs_yield`:

- If a thread is ready to run when its turn comes, it is dispatched and its `frs_run` flag is set to indicate that this thread has run at least once within this minor frame.

- When a thread yields, its `frs_yield` flag is set to indicate that the thread has released the processor. It is not activated again within this minor frame.

If a thread is not ready (usually because it is blocked waiting for I/O, a semaphore, or a lock), it is skipped. Upon reaching the end of the queue, the scheduler goes back to the beginning, in a round-robin fashion, searching for threads that have not yielded and may have become ready to run. If no ready threads are found, the frame scheduler goes into idle mode until a thread becomes available or until an interrupt marks the end of the frame.

### Detecting Overrun and Underrun

When a time base interrupt occurs to indicate the end of the minor frame, the frame scheduler checks the flags for each thread. If the `frs_run` flag has not been set, that thread never ran and therefore is a candidate for an *underrun exception*. If the `frs_run` flag is set but the `frs_yield` flag is not, the thread is a candidate for an *overrun exception*.

Whether these exceptions are declared depends on the scheduling discipline assigned to the thread. For more information, see "Using the Scheduling Disciplines" on page 72.

At the end of a minor frame, the frame scheduler resets all `frs_run` flags, except for those of threads that use the continuable discipline in that minor frame. For those threads, the residual `frs_yield` flags keeps the threads that have yielded from being dispatched in the next minor frame.

Underrun and overrun exceptions are typically communicated via Linux signals. For more information, see "Using Signals Under the Frame Scheduler" on page 86.

### Estimating Available Time

It is up to the application to make sure that all the threads queued to any minor frame can actually complete their work in one minor-frame interval. If there is too much work for the available CPU cycles, overrun errors will occur.

Estimation is somewhat simplified by the fact that a restricted CPU will only execute threads specifically pinned to it, along with a few CPU-specific kernel threads. You must estimate the maximum time each thread can consume between one call to `frs_yield()` and the next.

Frame scheduler threads do compete for CPU cycles with I/O interrupts on the same CPU. If you direct I/O interrupts away from the CPU, the only competition for CPU

cycles (other than a very few essential interrupts and CPU-specific kernel threads) is the overhead of the frame scheduler itself, and it has been carefully optimized to reduce overhead.

Alternatively, you may assign specific I/O interrupts to a CPU used by the frame scheduler. In that case, you must estimate the time that interrupt service will consume and allow for it.

## Synchronizing Multiple Schedulers

When the activities of one frame cannot be completed by one CPU, you must recruit additional CPUs and execute some activities concurrently. However, it is important that each of the CPUs have the same time base, so that each starts and ends frames at the same time.

You can create one master frame scheduler that owns the time base and one CPU, and as many synchronized (slave) frame schedulers as you need, each managing an additional CPU. The slave schedulers take their time base from the master, so that all start minor frames at the same instant.

Each frame scheduler requires its own controller thread. Therefore, to create multiple, synchronized frame schedulers, you must create a controller thread for the master and each slave frame scheduler.

Each frame scheduler has its own queues of threads. A given thread can be queued to only one CPU. (However, you can create multiple threads based on the same code, and queue each to a different CPU.) All synchronized frame schedulers use the same number of minor frames per major frame, which is taken from the definition of the master frame scheduler.

## Starting a Single Scheduler

A single frame scheduler is created when the frame scheduler controller thread calls `frs_create_master()` or `frs_create()`. The frame scheduler controller calls `frs_pthread_enqueue()` or `frs_enqueue()` one or more times to notify the new frame scheduler of the threads to schedule in each of the minor frames. The frame scheduler controller calls `frs_start()` when it has queued all the threads. Each scheduled thread must call `frs_join()` after it has initialized and is ready to be scheduled.

Each activity thread must be queued to at least one minor frame before it can join the frame scheduler via `frs_join()`. After all activity threads have joined and the frame scheduler is started by the controller thread, the first minor frame begins executing. For more information about these functions, see the `frs_enqueue`(3), `frs_join`(3), and `frs_start`(3) man pages.

## Starting Multiple Schedulers

A frame scheduler cannot start dispatching activities until the following have occurred:

- The frame scheduler controller has queued all the activity threads to their minor frames

- All the queued threads have done their own initial setup and have joined

When multiple frame schedulers are used, none can start until all are ready.

Each frame scheduler controller notifies its frame scheduler that it has queued all activities by calling `frs_start()`. Each activity thread signals its frame scheduler that it is ready to begin real-time processing by calling `frs_join()`.

A frame scheduler is ready when it has received one or more `frs_pthread_enqueue()` or `frs_enqueue()` calls, a matching number of `frs_join()` calls, and an `frs_start()` call for each frame scheduler. Each slave frame scheduler notifies the master frame scheduler when it is ready. When all the schedulers are ready, the master frame scheduler gives the downbeat and the first minor frame begins.

## Pausing Frame Schedulers

Any frame scheduler can be made to pause and restart. Any thread (typically but not necessarily the frame scheduler controller) can call `frs_stop()`, specifying a particular frame scheduler. That scheduler continues dispatching threads from the current minor frame until all have yielded. Then it goes into an idle loop until a call to `frs_resume()` tells it to start. It resumes on the next time-base interrupt, with the next minor frame in succession. For more information, see the `frs_stop`(3) and `frs_resume`(3) man pages.

> **Note:** If there is a thread running background discipline in the current minor frame, it continues to execute until it yields or is blocked on a system service. See "Background Discipline" on page 73

Because a frame scheduler does not stop until the end of a minor frame, you can stop and restart a group of synchronized frame schedulers by calling `frs_stop()` for each one before the end of a minor frame. There is no way to restart all of a group of schedulers with the certainty that they start up on the same time-base interrupt.

## Managing Activity Threads

The frame scheduler control thread identifies the initial set of activity threads by calling `frs_pthread_enqueue()` or `frs_enqueue()` prior to starting the frame scheduler. All the queued threads must call `frs_join()` before scheduling can begin. However, the frame scheduler controller can change the set of activity threads dynamically while the frame scheduler is working, using the following functions:

| | |
|---|---|
| `frs_getqueuelen()` | Gets the number of threads currently in the queue for a specified minor frame |
| `frs_pthread_readqueue()` or `frs_readqueue()` | Returns the ID values of all queued threads for a specified minor frame as a vector of integers |
| `frs_pthread_remove()` or `frs_premove()` | Removes a thread (specified by its ID) from a minor frame queue |
| `frs_pthread_insert()` or `frs_pinsert()` | Inserts a thread (specified by its ID and discipline) into a given position in a minor frame queue |

Using these functions, the frame scheduler controller can change the queueing discipline (overrun, underrun, continuable) of a thread by removing it and inserting it with a new discipline. The frame scheduler controller can suspend a thread by removing it from its queue or can restart a thread by putting it back in its queue.

> **Note:** When an activity thread is removed from the last or only queue it was in, it no longer is dispatched by the frame scheduler. When an activity thread is removed from a queue, a signal may be sent to the removed thread (see "Handling Signals in an Activity Thread" on page 87). If a signal is sent to it, it begins executing in its specified or default signal handler; otherwise, it begins executing following `frs_yield()`. After being returned to the Linux scheduler, a call to a frame scheduler function such as `frs_yield()` returns an error (this also can be used to indicate the resumption of normal scheduling).

The frame scheduler controller can also queue new threads that have not been scheduled before. The frame scheduler does not reject an `frs_pthread_insert()` or `frs_pinsert()` call for a thread that has not yet joined the scheduler. However, a thread must call `frs_join()` before it can be scheduled. For more information, see the `frs_pinsert`(3) man page.

If a queued thread is terminated for any reason, the frame scheduler removes the thread from all queues in which it appears.

## Selecting a Time Base

The program specifies an interrupt source for the time base when it creates the master (or only) frame scheduler. The master frame scheduler initializes the necessary hardware resources and redirects the interrupt to the appropriate CPU and handler.

The frame scheduler time base is fundamental because it determines the duration of a minor frame, and hence the frame rate of the program. This section explains the different time bases that are available.

When you use multiple, synchronized frame schedulers, the master frame scheduler distributes the time-base interrupt to each synchronized CPU. This ensures that minor-frame boundaries are synchronized across all the frame schedulers.

This section discusses the following:

- "High-Resolution Timer" on page 71
- "External Interrupts as a Time Base" on page 71

## High-Resolution Timer

The real-time clock (RTC) is synchronous across all processors and is ideal to drive synchronous schedulers. REACT uses the RTC for its frame scheduler high-resolution timer solution.

**Note:** Frame scheduler applications cannot use POSIX high-resolution timers.

To use the RTC, specify `FRS_INTRSOURCE_CCTIMER` and the minor frame interval in microseconds to `frs_create_master()` or `frs_create()`. The maximum frame rate supported by a timer is 2000 Hz.

The high-resolution timers in all CPUs are synchronized automatically.

**Note:** Altix XE servers do not have a built-in high-resolution timer. A `pci-rt` external interrupt card is supplied by SGI and is required for generation of the the high-resolution timer interrupts. Each `pci-rt` can generate interrupts at one set frequency, so a `pci-rt` card is required for each asynchronous frame scheduler running on a system.

## External Interrupts as a Time Base

To use external interrupts as a time base, use the following steps:

1. Load `ioc4_extint` to load the external interrupts modules.

2. Open the appropriate external interrupts device file. For example:

```
if ((fd = open("/dev/extint0", O_RDONLY)) < 0) {
        perror("Open EI control file");
        return 1;
}
```

3. Specify `FRS_INTRSOURCE_EXTINTR` as the `intr_source` and pass the returned file descriptor as the `intr_qualifier` to `frs_create_master` or `frs_create`.

The CPU receiving the interrupt allocates it simultaneously to the synchronized schedulers. If other IOC4 devices are also in use, you should redirect IOC4 interrupts to a non-frame-scheduled CPU in order to avoid jitter and delay.

For more information, see Chapter 3, "External Interrupts" on page 17.

# Using the Scheduling Disciplines

When a frame scheduler controller thread queues an activity thread to a minor frame using `frs_pthread_enqueue()` or `frs_enqueue()`, it must specify a *scheduling discipline* that tells the frame scheduler how the thread is expected to use its time within that minor frame.

The disciplines are as follows:

- "Real-Time Discipline" on page 72
- "Background Discipline" on page 73
- "Underrunable Discipline" on page 73
- "Overrunnable Discipline" on page 74
- "Continuable Discipline" on page 74

## Real-Time Discipline

In the real-time discipline, an activity thread starts during the minor frame in which it is queued, completes its work, and yields within the same minor frame. If the thread is not ready to run (for example, if it is blocked on I/O) during the entire minor frame, an *underrun exception* is said to occur. If the thread fails to complete its work and yield within the minor frame interval, an *overrun exception* is said to occur.

---

**Note:** If an activity thread becomes blocked by other than an `frs_yield()` call (and therefore is not ready to run) and later becomes unblocked outside of its minor frame slot, it will run assuming that no other threads are available to run (similar to "Background Discipline" on page 73) until it yields or a new minor frame begins.

---

This model could describe a simple kind of simulator in which certain activities (such as poll the inputs, calculate the new status, and update the display) must be repeated in the same order during every frame. In this scenario, each activity must start and must finish in every frame. If one fails to start, or fails to finish, the real-time program is broken and must take action.

However, realistic designs need the flexibility to have threads with the following characteristics:

- Need not start every frame; for instance, threads that sleep on a semaphore until there is work for them to do

- May run longer than one minor frame

- Should run only when time is available, and whose rate of progress is not critical

The other disciplines are used, in combination with real-time and with each other, to allow these variations.

## Background Discipline

The background discipline is mutually exclusive with the other disciplines. The frame scheduler dispatches a background thread only when all other threads queued to that minor frame have run and have yielded. Because the background thread cannot be sure it will run and cannot predict how much time it will have, the concepts of underrun and overrun do not apply to it.

**Note:** A thread with the background discipline must be queued to its frame following all non-background threads. Do not queue a real-time thread after a background thread.

## Underrunable Discipline

You specify the underrunable discipline in the following cases:

- When a thread needs to run only when an event has occurred, such as a lock being released or a semaphore being posted

- When a thread may need more than one minor frame (see "Using Multiple Consecutive Minor Frames" on page 74)

You specify the underrunable discipline with the real-time discipline to prevent detection of underrun exceptions. When you specify real time plus underrunable, the thread is not required to start in that minor frame. However, if it starts, it is required to yield before the end of the frame or an overrun exception is raised.

## Overrunnable Discipline

You specify the overrunnable discipline in the following cases:

- When it truly does not matter if the thread fails to complete its work within the minor frame—for example, a calculation of a game strategy that, if it fails to finish, merely makes the computer a less dangerous opponent

- When a thread may need more than one minor frame (see "Using Multiple Consecutive Minor Frames" on page 74)

You specify an overrunnable discipline with a real-time discipline to prevent detection of overrun exceptions. When you specify overrunnable plus real-time, the thread is not required to call `frs_yield()` before the end of the frame. Even so, the thread is preempted at the end of the frame. It does not have a chance to run again until the next minor frame in which it is queued. At that time it resumes where it was preempted, with no indication that it was preempted.

## Continuable Discipline

You specify continuable discipline with real-time discipline to prevent the frame scheduler from clearing the flags at the end of this minor frame (see "Scheduling Within a Minor Frame" on page 65).

The result is that, if the thread yields in this frame, it need not run or yield in the following frame. The residual `frs_yield` flag value, carried forward to the next frame, applies. You specify continuable discipline with other disciplines in order to let a thread execute just once in a block of consecutive minor frames.

# Using Multiple Consecutive Minor Frames

There are cases when a thread sometimes or always requires more than one minor frame to complete its work. Possibly the work is lengthy, or possibly the thread could be delayed by a system call or a lock or semaphore wait.

You must decide the absolute maximum time the thread could consume between starting up and calling `frs_yield()`. If this is unpredictable, or if it is predictably longer than the major frame, the thread cannot be scheduled by the frame scheduler. Hence, it should probably run on another CPU under the Linux real-time scheduler.

However, when the worst-case time is bounded and is less than the major frame, you can queue the thread to enough consecutive minor frames to allow it to finish. A combination of disciplines is used in these frames to ensure that the thread starts when it should, finishes when it must, and does not cause an error if it finishes early.

The discipline settings should be as follows:

| Frame | Description |
|---|---|
| First | Real-time + overrunnable + continuable |
| | The thread must start in this frame (not underrunnable) but is not required to yield (overrunnable). If it yields, it is not restarted in the following minor frame (continuable). |
| Intermediate | Realtime + underrunnable + overrunnable + continuable |
| | The thread need not start (it might already have yielded, or might be blocked) but is not required to yield. If it does yield (or if it had yielded in a preceding minor frame), it is not restarted in the following minor frame (continuable). |
| Final | Realtime + underrunnable |
| | The thread need not start (it might already have yielded) but if it starts, it must yield in this frame (not overrunnable). The thread can start a new run in the next minor frame to which it is queued (not continuable). |

A thread can be queued for one or more of these multiframe sequences in one major frame. For example, suppose that the minor frame rate is 60 Hz and a major frame contains 60 minor frames (1 Hz). You have a thread that should run at a rate of 5 Hz and can use up to 3/60 second at each dispatch. You can queue the thread to 5 sequences of 3 consecutive frames each. It could start in frames 0, 12, 24, 36, and 48. Frames 1, 13, 25, 37, and 49 could be intermediate frames, and 2, 14, 26, 38, and 50 could be final frames.

## Designing an Application for the Frame Scheduler

When using the frame scheduler, consider the following guidelines when designing a real-time application:

1. Determine the programming model for implementing the activities in the program, choosing among POSIX threads or SVR4 `fork()` calls. (You cannot mix pthreads and other disciplines within the program.)

2. Partition the program into activities, where each activity is an independent piece of work that can be done without interruption.

   For example, in a simple vehicle simulator, activities might include the following:

   • Poll the joystick

   • Update the positions of moving objects

   • Cull the set of visible objects

3. Decide the relationships among the activities, as follows:

   • Some must be done once per minor frame, others less frequently

   • Some must be done before or after others

   • Some may be conditional (for example, an activity could poll a semaphore and do nothing unless an event had completed)

4. Estimate the worst-case time required to execute each activity. Some activities may need more than one minor frame interval (the frame scheduler allows for this).

5. Schedule the activities: If all are executed sequentially, will they complete in one major frame? If not, choose activities that can execute concurrently on two or more CPUs, and estimate again. You may have to change the design in order to get greater concurrency.

When the design is complete, implement each activity as an independent thread that communicates with the others using shared memory, semaphores, and locks.

A controller thread creates, stops, and resumes the frame scheduler. The controller thread can also interrogate and receive signals from the frame scheduler.

A frame scheduler seizes its assigned CPU, isolates it, and controls the scheduling on it. It waits for all queued threads to initialize themselves and join the scheduler. The frame scheduler begins dispatching the threads in the specified sequence during each

frame interval. Errors are monitored (such as a thread that fails to complete its work within its frame) and a specified action is taken when an error occurs. Typically, the error action is to send a signal to the controller thread.

## Preparing the System

Before a real-time program executes, you must do the following:

1. Choose the CPUs that the real-time program will use. CPU 0 (at least) must be reserved for Linux system functions.

2. Decide which CPUs will handle I/O interrupts. By default, Linux distributes I/O interrupts across all available processors as a means of balancing the load (referred to as *spraying interrupts*). You should redirect I/O interrupts away from CPUs that are used for real-time programs. For more information, see "Redirect Interrupts" on page 46.

3. If using an external interrupt as a time base, make sure it is redirected to the CPU of the master frame scheduler. For more information, see "External Interrupts as a Time Base" on page 71.

4. Make sure that none of the real-time CPUs is managing the clock. Normally, the responsibility of handling 10–ms scheduler interrupts is given to CPU 0. For more information, see "Avoid the Clock Processor (CPU 0)" on page 45.

5. Restrict and isolate the real-time CPUs, as described in "Restrict, Isolate, and Shield CPUs" on page 47.

6. Load the `frs` kernel module:

   ```
   [root@linux root]# modprobe frs
   ```

   **Note:** You must perform this step after each system boot.

7. If you are using external interrupts as a time base, or you are running the frame scheduler on an Altix XE server, you must load the `ioc4_extint` kernel module:

   ```
   [root@linux root]# modprobe ioc4_extint
   ```

## Implementing a Single Frame Scheduler

When the activities of a real-time program can be handled within a major frame interval by a single CPU, the program requires only one frame scheduler. The programs found in `/usr/share/react/frs/examples` provide examples of implementing a single frame scheduler.

Typically, a program has a top-level controller thread to handle startup and termination, and one or more activity threads that are dispatched by the frame scheduler. The activity threads are typically lightweight pthreads, but that is not a requirement; they can also be created with `fork()`. (They need not be children of the controller thread.) For examples, see `/usr/share/react/frs/examples`.

In general, these are the steps for setting up a single frame scheduler:

1. Initialize global resources such as memory-mapped segments, memory arenas, files, asynchronous I/O, semaphores, locks, and other resources.

2. Lock the shared address space segments. (When `fork()` is used, each child process must lock its own address space.)

3. If using pthreads, create a controller thread; otherwise, the initial thread of execution may be used as the controller thread.

   - Create a controller thread using `pthread_create()` and the attribute structure you just set up. See the `pthread_create`(3P) man page for details.

   - Exit the initial thread, because it cannot execute any frame scheduler operations.

4. Create the frame scheduler using `frs_create_master()`, `frs_create_vmaster()`, or `frs_create()`. See the `frs_create`(3) man page.

5. Create the activity threads using one of the following interfaces, depending on the thread model being used:

   - `pthread_create()`

   - `fork()`

6. Queue the activity threads on the target minor frame queues, using `frs_pthread_enqueue()` or `frs_enqueue()`.

7. Optionally, initialize the frame scheduler signal handler to catch frame overrun, underrun, and activity dequeue events (see "Setting Frame Scheduler Signals" on

page 88 and "Setting Exception Policies" on page 84). The handlers are set at this time, after creation of the activity threads, so that the activity threads do not inherit them.

8.  Use `frs_start()` to enable scheduling. For more information, see Table 5-1.

9.  Have the activity threads call `frs_join()`. The frame scheduler begins scheduling processes as soon as all the activity threads have called `frs_join()`.

10. Wait for error signals from the frame scheduler and for the termination of child processes.

11.  Use `frs_destroy()` to terminate the frame scheduler.

12. Perform program cleanup as desired.

See `/usr/share/react/frs/examples`.

## Implementing Synchronized Schedulers

When the real-time application requires the power of multiple CPUs, you must add one more level to the program design for a single CPU. The program creates multiple frame schedulers, one master and one or more synchronized slaves.

This section discusses the following:

-   "Synchronized Scheduler Concepts" on page 79
-   "Master Controller Thread" on page 80
-   "Slave Controller Thread" on page 81

### Synchronized Scheduler Concepts

The first frame scheduler provides the time base for the others. It is called the *master scheduler*. The other schedulers take their time base interrupts from the master, and so are called *slaves*. The combination is called a *sync group*.

No single thread may create more than one frame scheduler. This is because every frame scheduler must have a unique frame scheduler controller thread to which it can send signals. As a result, the program has the following types of threads:

- A master controller thread that sets up global data and creates the master frame scheduler

- One slave controller thread for each slave frame scheduler

- Activity threads

The master frame scheduler must be created before any slave frame schedulers can be created. Slave frame schedulers must be specified to have the same time base and the same number of minor frames as the master.

Slave frame schedulers can be stopped and restarted independently. However, when any scheduler (master or slave) is destroyed, all are immediately destroyed.

## Master Controller Thread

The master controller thread performs these steps:

1. Initializes a global resource. One global resource is the thread ID of the master controller thread.

2. Creates the master frame scheduler using either the `frs_create_master()` or `frs_create_vmaster()` call and stores its handle in a global location.

3. Creates one slave controller thread for each synchronized CPU to be used.

4. Creates the activity threads that will be scheduled by the master frame scheduler and queues them to their assigned minor frames.

5. Sets up signal handlers for signals from the frame scheduler. See "Using Signals Under the Frame Scheduler" on page 86.

6. Uses `frs_start()` to tell the master frame scheduler that its activity threads are all queued and ready to commence scheduling. See Table 5-1.

   The master frame scheduler starts scheduling threads as soon as all threads have called `frs_join()` for their respective schedulers.

7. Waits for error signals.

8. Uses `frs_destroy()` to terminate the master frame scheduler.

9. Performs any desired program cleanup.

**007–4746–005**

### Slave Controller Thread

Each slave controller thread performs these steps:

1. Creates a synchronized frame scheduler using `frs_create_slave()`, specifying information about the master frame scheduler stored by the master controller thread. The master frame scheduler must exist. A slave frame scheduler must specify the same time base and number of minor frames as the master frame scheduler.

2. Changes the frame scheduler signals or exception policy, if desired. See "Setting Frame Scheduler Signals" on page 88 and "Setting Exception Policies" on page 84.

3. Creates the activity threads that are scheduled by this slave frame scheduler and queues them to their assigned minor frames.

4. Sets up signal handlers for signals from the slave frame scheduler.

5. Use `frs_start()` to tell the slave frame scheduler that all activity threads have been queued.

   The slave frame scheduler notifies the master when all threads have called `frs_join()`. When the master frame scheduler starts broadcasting interrupts, scheduling begins.

6. Waits for error signals.

7. Uses `frs_destroy()` to terminate the slave frame scheduler.

For an example of this kind of program structure, refer to `/usr/share/react/frs/examples`.

---

**Tip:** In this design sketch, the knowledge of which activity threads to create, and on which frames to queue them, is distributed throughout the code, where it might be hard to maintain. However, it is possible to centralize the plan of schedulers, activities, and frames in one or more arrays that are statically initialized. This improves the maintainability of a complex program.

---

## Handling Frame Scheduler Exceptions

The frame scheduler control thread for a scheduler controls the handling of the overrun and underrun exceptions. It can specify how these exceptions should be

handled and what signals the frame scheduler should send. These policies must be set before the scheduler is started. While the scheduler is running, the frame scheduler controller can query the number of exceptions that have occurred.

This section discusses the following:

- "Exception Types" on page 82
- "Exception Handling Policies" on page 82
- "Setting Exception Policies" on page 84
- "Querying Counts of Exceptions" on page 85

## Exception Types

The overrun exception indicates that a thread failed to yield in a minor frame where it was expected to yield and was preempted at the end of the frame. An overrun exception indicates that an unknown amount of work that should have been done was not done, and will not be done until the next frame in which the overrunning thread is queued.

The underrun exception indicates that a thread that should have started in a minor frame did not start. The thread may have terminated or (more likely) it was blocked in a wait because of an unexpected delay in I/O or because of a deadlock on a lock or semaphore.

## Exception Handling Policies

The frame scheduler control thread can establish one of four policies for handling overrun and underrun exceptions. When it detects an exception, the frame scheduler can do the following:

- Send a signal to the controller
- Inject an additional minor frame
- Extend the frame by a specified number of microseconds
- Steal a specified number of microseconds from the following frame

By default, it sends a signal. The scheduler continues to run. The frame scheduler control thread can then take action, such as terminating the frame scheduler. For more information, see "Setting Frame Scheduler Signals" on page 88.

### Injecting a Repeat Frame

The policy of injecting an additional minor frame can be used with any time base. The frame scheduler inserts another complete minor frame, essentially repeating the minor frame in which the exception occurred. In the case of an overrun, the activity threads that did not finish have another frame's worth of time to complete. In the case of an underrun, there is that much more time for the waiting thread to wake up. Because exactly one frame is inserted, all other threads remain synchronized to the time base.

### Extending the Current Frame

The policies of extending the frame, either with more time or by stealing time from the next frame, are allowed only when the time base is a high-resolution timer. For more information, see "Selecting a Time Base" on page 70.

When adding time, the current frame is made longer by a fixed amount of time. Because the minor frame becomes a variable length, it is possible for the frame scheduler to drop out of synchronization with an external device.

When stealing time from the following frame, the frame scheduler returns to the original time base at the end of the following minor frame provided that the threads queued to that following frame can finish their work in a reduced amount of time. If they do not, the frame scheduler steals time from the next frame.

### Dealing With Multiple Exceptions

You decide how many consecutive exceptions are allowed within a single minor frame. After injecting, stretching, or stealing time that many times, the frame scheduler stops trying to recover and sends a signal instead.

The count of exceptions is reset when a minor frame completes with no remaining exceptions.

## Setting Exception Policies

The `frs_pthread_setattr()` or `frs_setattr()` function is used to change exception policies. This function must be called before the frame scheduler is started. After scheduling has begun, an attempt to change the policies or signals is rejected.

In order to allow for future enhancements, `frs_pthread_setattr()` or `frs_setattr()` accepts arguments for minor frame number and thread ID; however it currently allows setting exception policies only for all policies and all minor frames. The most significant argument to it is the `frs_recv_info` structure, declared with the following fields:

```
typedef struct frs_recv_info {
    mfbe_rmode_t  rmode;       /* Basic recovery mode */
    mfbe_tmode_t  tmode;       /* Time expansion mode */
    uint          maxcerr;     /* Max consecutive errors */
    uint          xtime;       /* Recovery extension time */
} frs_recv_info_t;
```

The recovery modes and other constants are declared in `/usr/include/frs.h`. The function in Example 5-3 sets the policy of injecting a repeat frame. The caller specifies only the frame scheduler and the number of consecutive exceptions allowed.

**Example 5-3** Function to Set INJECTFRAME Exception Policy

```
int
setInjectFrameMode(frs_t *frs, int consecErrs)
{
  frs_recv_info_t work;
  bzero((void*)&work,sizeof(work));
  work.rmode = MFBERM_INJECTFRAME;
  work.maxcerr = consecErrs;
  return frs_setattr(frs,0,0,FRS_ATTR_RECOVERY,(void*)&work);
}
```

The function in Example 5-4 sets the policy of stretching the current frame (a function to set the policy of stealing time from the next frame is nearly identical). The caller specifies the frame scheduler, the number of consecutive exceptions, and the stretch time in microseconds.

**Example 5-4** Function to Set STRETCH Exception Policy

```
int
setStretchFrameMode(frs_t *frs,int consecErrs,uint microSecs)
{
  frs_recv_info_t work;
  bzero((void*)&work,sizeof(work));
  work.rmode = MFBERM_EXTENDFRAME_STRETCH;
  work.tmode = EFT_FIXED; /* only choice available */
  work.maxcerr = consecErrs;
  work.xtime = microSecs;
  return frs_setattr(frs,0,0,FRS_ATTR_RECOVERY,(void*)&work);
}
```

## Querying Counts of Exceptions

When you set a policy that permits exceptions, the frame scheduler controller thread can query for counts of exceptions. This is done with a call to frs_pthread_getattr() or frs_getattr(), passing the handle to the frame scheduler, the number of the minor frame and the thread ID of the thread within that frame.

The values returned in a structure of type frs_overrun_info_t are the counts of overrun and underrun exceptions incurred by that thread in that minor frame. In order to find the count of all overruns in a given minor frame, you must sum the counts for all threads queued to that frame. If a thread is queued to more than one minor frame, separate counts are kept for it in each frame.

The function in Example 5-5 takes a frame scheduler handle and a minor frame number. It gets the list of thread IDs queued to that minor frame, and returns the sum of all exceptions for all of them.

**Example 5-5** Function to Return a Sum of Exception Counts (pthread Model)

```
#define THE_MOST_TIDS 250
int
totalExcepts(frs_t * theFRS, int theMinor)
{
    int numTids = frs_getqueuelen(theFRS, theMinor);
    int j, sum;
    pthread_t allTids[THE_MOST_TIDS];
    if ( (numTids <= 0) || (numTids > THE_MOST_TIDS) )
```

```
        return 0; /* invalid minor #, or no threads queued? */

    if (frs_pthread_readqueue(theFRS, theMinor, allTids) == -1)
        return 0; /* unexpected problem with reading IDs */

    for (sum = j = 0; j<numTids; ++j)
    {
        frs_overrun_info_t work;
        frs_pthread_getattr(theFRS     /* the scheduler */
                    theMinor,          /* the minor frame */
                    allTids[j],        /* the threads */
                    FRS_ATTR_OVERRUNS, /* want counts */
                    &work);            /* put them here */
        sum += (work.overruns + work.underruns);
    }
    return sum;
}
```

**Note:** The frame scheduler read queue functions return the number of threads present on the queue at the time of the read. Applications can use this returned value to eliminate calls to `frs_getqueuelen()`.

## Using Signals Under the Frame Scheduler

The frame scheduler itself sends signals to the threads using it. Threads can communicate by sending signals to each other. In brief, a frame scheduler sends signals to indicate the following:

• The frame scheduler has been terminated

• An overrun or underrun has been detected

• A thread has been dequeued

The rest of this section describes how to specify the signal numbers and how to handle the signals:

- "Handling Signals in the Frame Scheduler Controller" on page 87

- "Handling Signals in an Activity Thread" on page 87

- "Setting Frame Scheduler Signals" on page 88

- "Handling a Sequence Error" on page 89

## Handling Signals in the Frame Scheduler Controller

When a frame scheduler detects an overrun or underrun exception from which it cannot recover, and when it is ready to terminate, it sends a signal to the frame scheduler controller.

**Tip:** Child processes inherit signal handlers from the parent, so a parent should not set up handlers prior to `fork()` unless they are meant to be inherited.

The frame scheduler controller for a synchronized frame scheduler should have handlers for underrun and overrun signals. The handler could report the error and issue `frs_destroy()` to shut down its scheduler. A frame scheduler controller for a synchronized scheduler should use the default action for `SIGHUP` (exit) so that completion of the `frs_destroy()` quietly terminates the frame scheduler controller.

The frame scheduler controller for the master (or only) frame scheduler should catch underrun and overrun exceptions, report them, and shut down its scheduler.

When a frame scheduler is terminated with `frs_destroy()`, it sends `SIGKILL` to its frame scheduler controller. This cannot be changed and `SIGKILL` cannot be handled. Hence `frs_destroy()` is equivalent to termination for the frame scheduler controller.

## Handling Signals in an Activity Thread

A frame scheduler can send a signal to an activity thread when the thread is removed from any queue using `frs_pthread_remove()` or `frs_premove()`. The scheduler can also send a signal to an activity thread when it is removed from the last or only minor frame to which it was queued (at which time it is scheduled only by Linux). For more information, see "Managing Activity Threads" on page 69.

In order to have these signals sent, the frame scheduler controller must set nonzero signal numbers for them, as discussed in "Setting Frame Scheduler Signals".

## Setting Frame Scheduler Signals

The frame scheduler sends signals to the frame scheduler controller.

The signal numbers used for most events can be modified. Signal numbers can be queried using `frs_pthread_getattr(FRS_ATTR_SIGNALS)` or `frs_getattr(FRS_ATTR_SIGNALS)` and changed using `frs_pthread_setattr(FRS_ATTR_SIGNALS)` or `frs_setattr(FRS_ATTR_SIGNALS)`, in each case passing an `frs_signal_info` structure. This structure contains room for four signal numbers, as shown in Table 5-2.

**Table 5-2** Signal Numbers Passed in `frs_signal_info_t`

| Field Name | Signal Purpose | Default Signal Number |
| --- | --- | --- |
| sig_underrun | Notify frame scheduler controller of underrun | SIGUSR1 |
| sig_overrun | Notify frame scheduler controller of the overrun | SIGUSR2 |
| sig_dequeue | Notify an activity thread that it has been dequeued with `frs_pthread_remove()` or `frs_premove()` | 0 (do not send) |
| sig_unframesched | Notify an activity thread that it has been removed from the last or only queue in which it was queued | SIGRTMIN |

Signal numbers must be changed before the frame scheduler is started. All the numbers must be specified to `frs_pthread_setattr()` or `frs_setattr()`, so the proper way to set any number is to first file the `frs_signal_info_t` using `frs_pthread_getattr()` or `frs_getattr()`. The function in Example 5-6 sets the signal numbers for overrun and underrun from its arguments.

**Example 5-6** Function to Set Frame Scheduler Signals

```
int
setUnderOverSignals(frs_t *frs, int underSig, int overSig)
{
  int error;
  frs_signal_info_t work;
  error = frs_pthread_getattr(frs,0,0,FRS_ATTR_SIGNALS,(void*)&work);
  if (!error)
  {
    work.sig_underrun = underSig;
    work.sig_overrun = overSig;
    error = frs_pthread_setattr(frs,0,0,FRS_ATTR_SIGNALS,(void*)&work);
  }
  return error;
}
```

## Handling a Sequence Error

When `frs_create_vmaster()` is used to create a frame scheduler triggered by multiple interrupt sources, a sequence error signal is dispatched to the controller thread if the interrupts come in out of order. For example, if the first and second minor frame interrupt sources are different, and the second minor frame's interrupt source is triggered before the first minor frame's interrupt source, then a sequence error has occurred.

This type of error condition is indicative of unrealistic time constraints defined by the interrupt information template.

The signal code that represents the occurrence of a sequence error is `SIGRTMIN+1`. This signal cannot be reset or disabled using the `frs_setattr()` interface.

# Using Timers with the Frame Scheduler

Frame scheduler applications cannot use POSIX high-resolution timers. With other interval timers, signal delivery to an activity thread can be delayed, so timer latency is unpredictable.

If the frame scheduler controller is using timers, it should run on a node outside of those containing CPUs running frame scheduler worker threads.

**Example 5-7** Minimal Activity Process as a Timer

```
frs_join(scheduler-handle)
do {
   usvsema(frs-controller-wait-semaphore);
   frs_yield();
} while(1);
_exit();
```

# Disk I/O Optimization

A real-time program sometimes must perform disk I/O under tight time constraints and without affecting the timing of other activities such as data collection. This chapter covers techniques that can help you meet these performance goals:

- "Memory-Mapped I/O" on page 91

- "Asynchronous I/O" on page 91

## Memory-Mapped I/O

When an input file has a fixed size, the simplest as well as the fastest access method is to map the file into memory. A file that represents a database (such as a file containing a precalculated table of operating parameters for simulated hardware) is best mapped into memory and accessed as a memory array. A mapped file of reasonable size can be locked into memory so that access to it is always fast.

You can also perform output on a memory-mapped file by storing into the memory image. When the mapped segment is also locked in memory, you control when the actual write takes place. Output happens only when the program calls `msync()` or changes the mapping of the file at the time that the modified pages are written. The time-consuming call to `msync()` can be made from an asynchronous process. For more information, see the msync(2) man page.

## Asynchronous I/O

You can use asynchronous I/O to isolate the real-time processes in your program from the unpredictable delays caused by I/O. Asynchronous I/O in Linux strives to conform with the POSIX real-time specification 1003.1-2003.

This section discusses the following:

- "Conventional Synchronous I/O" on page 92

- "Asynchronous I/O Basics" on page 92

## Conventional Synchronous I/O

Conventional I/O in Linux is synchronous; that is, the process that requests the I/O is blocked until the I/O has completed. The effects are different for input and for output.

For disk files, the process that calls `write()` is normally delayed only as long as it takes to copy the output data to a buffer in kernel address space. The device driver schedules the device write and returns. The actual disk output is asynchronous. As a result, most output requests are blocked for only a short time. However, since a number of disk writes could be pending, the true state of a file on disk is unknown until the file is closed.

In order to make sure that all data has been written to disk successfully, a process can call `fsync()` for a conventional file or `msync()` for a memory-mapped file. The process that calls these functions is blocked until all buffered data has been written. For more information, see the `fsync`(2) and `msync`(2) man pages.

Devices other than disks may block the calling process until the output is complete. It is the device driver logic that determines whether a call to `write()` blocks the caller, and for how long.

## Asynchronous I/O Basics

A real-time process must read or write a device, but it cannot tolerate an unpredictable delay. One obvious solution can be summarized as "call `read()` or `write()` from a different process, and run that process in a different CPU." This is the essence of asynchronous I/O. You could implement an asynchronous I/O scheme of your own design, and you may wish to do so in order to integrate the I/O closely with your own configuration of processes and data structures. However, a standard solution is available.

Linux supports asynchronous I/O library calls that strive to conform with the POSIX real-time specification 1003.1-2003. You use relatively simple calls to initiate input or output.

For more information, see the `aio_read`(3) and `aio_write`(3) man pages.

# PCI Devices

The Linux `pci` interface provides a mechanism to access the PCI bus address spaces from user programs.

Performing programmed I/O on `pci` devices on SGI Linux systems involves the following steps:

1. Examine `/proc/bus/pci/devices` and obtain offsets for the base address registers (BARs) of the devices that you want to map. The format of the lines is as follows (with field widths in the number of hexadecimal characters shown, and line breaks added for readability):

*bus:2x  (slotnumber<<3_|_fn):2x  vend:8x  bar0  bar1  bar2  bar3  . . .*

For example:

```
if ((fptr = fopen( "/proc/bus/pci/devices", "r")) == NULL) {
            printf( "Unable to open /proc/bus/pci/devices\n" );
    }

    while(fgets(buf, sizeof(buf) - 1, fptr)) {
            sscanf( buf, "%2x%2x %8x %*x %lx %lx %lx %lx %lx %lx %lx %lx %*lx %*lx %*lx %*lx %*lx %*lx",
            &sbus, &sdevfn, &svend, &sbar[0], &sbar[1], &sbar[2], &sbar[3],
            &sbar[4], &sbar[5], &sbar[6], &sbar[7]);
            if(( sbus == bus ) && (sdevfn == devfn)) {
                    /* This is the bus, slot and function we're looking for,
                     * so save the base address register offset information.  */
                    for(int i=0; i> 16;
                    device = svend & 0xffff;
            }
    }
    fclose( fptr );
```

2. Open the appropriate device file for the bus, slot, and function in which you are interested. The device files are named as follows:

    `/proc/bus/pci/`*bus*`/`*slot*`.`*function*

For example:

```
memfile = (char*) malloc( 32 );
        sprintf( memfile, "/proc/bus/pci/%02d/%02d.%d", bus, slot, function );
        fd = open( memfile, O_RDWR );
```

3. Set the memory map state for the file to MEM space using the PCIIOC_MMAP_IS_MEM request to the ioctl() system call.

   For example:

   ```
   ioctl(fd, PCIIOC_MMAP_IS_MEM);
   ```

4. Map the opened file, using the offset obtained in step 1 as the *offset* parameter.

   For example:

   ```
   tmpPtr = (char *) mmap( NULL, (size_t) len, PROT_READ | PROT_WRITE,
                   MAP_SHARED, fd, (off_t) offset[bar]);
   ```

For a complete example, see Appendix D, "Reading MAC Addresses Sample Program" on page 185.

For details about kernel-level PCI device drivers, see the *Linux Device Driver Programmer's Guide-Porting to SGI Altix Systems*,

# User-Level Interrupts

**Note:** This chapter does not apply to Altix XE servers.

The user-level interrupt (ULI) facility allows a hardware interrupt to be handled by a user process.

A user process may register a function with the kernel, linked into the process in the normal fashion, to be called when a particular interrupt is received. The process, referred to as a *ULI process*, effectively becomes multithreaded, with the main process thread possibly running simultaneously with the interrupt handler thread. The interrupt handler is called asynchronously and has access only to the process's address space.

The ULI facility is intended to simplify the creation of device drivers for unsupported devices. ULIs can be written to respond to interrupts initiated from external interrupt ports. A programming error in the driver will result in nothing more serious than the termination of a process rather than crashing the entire system, and the developer need not know anything about interfacing a driver into the kernel.

The ULI feature may also be used for high-performance I/O applications when combined with memory-mapped device I/O. Applications can make all device accesses in user space. This is useful for high-performance I/O applications such as hardware-in-the-loop simulators.

ULIs are essentially *interrupt service routines (ISRs)* that reside in the address space of a user process. As shown in Figure 8-1, when an interrupt is received that has been registered to a ULI, it triggers the user function. For function prototypes and other details, see the `uli`(3) man page.
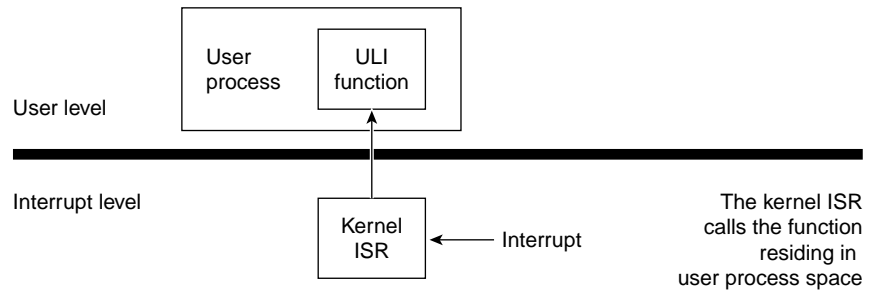
**Figure 8-1** ULI Functional Overview

---

**Note:** The uli(3) man page and the libuli library are installed as part of the REACT package.

---

This chapter discusses the following:

- "Overview of ULI" on page 96
- "Setting Up ULI" on page 100

## Overview of ULI

All registration functions return an opaque identifier for the ULI, which is passed as an argument to various other ULI functions. Table 8-1 lists the arguments that are common to all registration functions.

**Table 8-1** Common Arguments for Registration Functions

| Function | Description |
|----------|-------------|
| func | Points to the function that will handle the interrupt. |
| ULI_register_irq | Requests that an interrupt be handled as a ULI. Once a registration function has been called, the handler function may be called asynchronously any time the associated hardware sees fit to generate an interrupt. Any state needed by the handler function must have been initialized before ULI registration. The process will continue to receive the ULI until it exits or the ULI is destroyed (see ULI_destroy below), at which time the system reverts to handling the interrupt in the kernel. The CPU that executes the ULI handler is the CPU that would execute the equivalent kernel-based interrupt handler if the ULI were not registered (that is, the CPU to which the device sends the interrupt). |
| ULI_destroy | Destroys a ULI. When this function returns, the identifier will no longer be valid for use with any ULI function and the handler function used with it will no longer be called. |
| ULI_block_intr | Blocks a ULI. If the handler is currently running on another CPU in a multiprocessing environment, ULI_block_intr will spin until the handler has completed. |
| ULI_unblock_intr | Unblocks a ULI. Interrupts posted while the ULI was blocked will be handled at this time. If multiple interrupts occur while blocked, the handler function will be called only once when the interrupt is unblocked. |
| ULI_sleep | Blocks the calling thread on a semaphore associated with a particular ULI. The registration function initializes the ULI with a caller-specified number of semaphores. ULI_sleep may return before the event being awaited has occurred, thus it should be called within a while loop. |
| ULI_wakeup | Wakes up the next thread sleeping on a semaphore associated with a particular ULI. If ULI_wakeup is called before the corresponding ULI_sleep, the call to ULI_sleep will return immediately without blocking. |

For more details, see the uli(3) man page.

This section discusses the following:

- "Restrictions on the ULI Handler" on page 98
- "Planning for Concurrency: Declaring Global Variables" on page 99
- "Using Multiple Devices" on page 99

## Restrictions on the ULI Handler

Of the ULI library functions listed above, only ULI_wakeup may be called by the handler function.

Each ULI handler function runs within its own POSIX thread running at a priority in the range 80 through 89. Threads that run at a higher priority should not attempt to block ULI execution with ULI_block() because deadlock may occur.

If a ULI handler function does any of the following, its behavior is undefined:

- Causes a page fault
- Uses the floating point unit (FPU)
- Makes a system call
- Executes an illegal instruction

**Note:** To avoid page faults, use the mlock() or mlockall() function prior to creating the ULI.

You can only use the ULI_sleep and ULI_wakeup functions inside of a share group. These functions cannot wake up arbitrary processes.

In essence, the ULI handler should do only the following things, as shown in Figure 8-2:

- Store data in program variables in locked pages, to record the interrupt event. (For example, a ring buffer is a data structure that is suitable for concurrent access.)
- Program the device as required to clear the interrupt or acknowledge it. The ULI handler has access to the whole program address space, including any mapped-in devices, so it can perform PIO loads and stores.

- Post a semaphore to wake up the main process. This must be done using a ULI function.
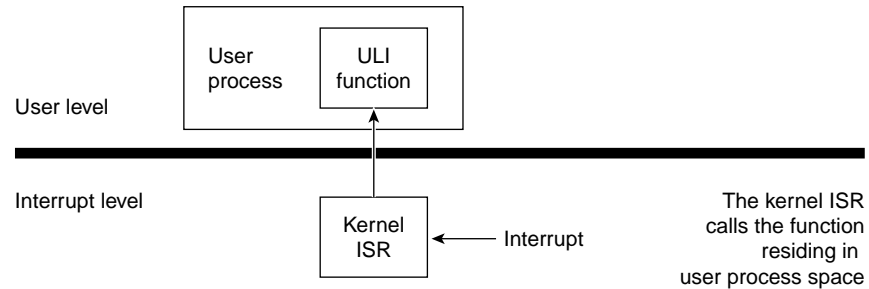


**Figure 8-2** ULI Handler Functions

## Planning for Concurrency: Declaring Global Variables

Because the ULI handler can interrupt the program at any point, or run concurrently with it, the program must be prepared for concurrent execution. This is done by declaring global variables. When variables can be modified by both the main process and the ULI handler, you must take special care to avoid race conditions.

An important step is to specify -D_SGI_REENTRANT_FUNCTIONS to the compiler in order to get the reentrant versions of the C library functions. This ensures that, if the main process and the ULI handler both enter the C library, there is no collision over global variables.

You can declare the global variables that are shared with the ULI handler with the keyword volatile so that the compiler generates code to load the variables from memory on each reference. However, the compiler never holds global values in registers over a function call, and you almost always have a function call such as ULI_block_intr() preceding a test of a shared global variable.

## Using Multiple Devices

The ULI feature allows a program to open more than one interrupting device. You register a handler for each device. However, the program can only wait for a specific interrupt to occur; that is, the ULI_sleep() function specifies the handle of one particular ULI handler. This does not mean that the main program must sleep until

that particular interrupt handler is entered, however. Any ULI handler can waken the main program, as discussed under "Interacting With the Handler" on page 102.

# Setting Up ULI

A program initializes for ULI in the following major steps:

1. Load the `uli` kernel module:

   `[root@linux root]# ` **`modprobe uli`**

2. For a PCI, map the device addresses into process memory.

3. Lock the program address space in memory.

4. Initialize any data structures used by the interrupt handler.

5. Register the interrupt handler.

6. Interact with the device and the interrupt handler.

An interrupt can occur any time after the handler has been registered, causing entry to the ULI handler.

This section discusses the following:

- "Opening the Device Special File" on page 100
- "Locking the Program Address Space" on page 101
- "Registering the Interrupt Handler" on page 101
- "Registering a Per-IRQ Handler" on page 102
- "Interacting With the Handler" on page 102
- "Achieving Mutual Exclusion" on page 103

## Opening the Device Special File

Devices are represented by device special files. In order to gain access to a device, you open the device special file that represents it. If the appropriate loadable kernel modules have been loaded (that is, the `extint` and `ioc4_extint` modules), the device file `/dev/extint#` should be created automatically for you, where # is

replaced by a system-assigned number, one for each of the IOC4 devices present in the system.

## Locking the Program Address Space

The ULI handler must not reference a page of program text or data that is not present in memory. You prevent this by locking the pages of the program address space in memory. The simplest way to do this is to call the `mlockall()` system function:

```
if (mlockall(MCL_CURRENT|MCL_FUTURE)<0) perror ("mlockall");
```

The `mlockall()` function has the following possible difficulties:

- The calling process must have either superuser privilege or `CAP_MEMORY_MGT` capability. This may not pose a problem if the program needs superuser privilege in any case (for example, to open a device special file). For more information, see the `mlockall(3C)` man page.

- The `mlockall()` function locks all text and data pages. In a very large program, this may be so much memory that system performance is harmed.

In order to use `mlock()`, you must specify the exact address ranges to be locked. Provided that the ULI handler refers only to global data and its own code, it is relatively simple to derive address ranges that encompass the needed pages. If the ULI handler calls any library functions, the library DSO must be locked as well. The smaller and simpler the code of the ULI handler, the easier it is to use `mlock()`.

## Registering the Interrupt Handler

When the program is ready to start operations, it registers its ULI handler. The ULI handler is a function that matches the following prototype:

```
void function_name(void *arg);
```

The registration function takes arguments with the following purposes:

- The address of the handler function.

- An argument value to be passed to the handler on each interrupt. This is typically a pointer to a work area that is unique to the interrupting device (supposing the program is using more than one device).

- A count of semaphores to be allocated for use with this interrupt.

The semaphores are allocated and maintained by the ULI support. They are used to coordinate between the program process and the interrupt handler, as discussed in "Interacting With the Handler" on page 102. You should specify one semaphore for each independent process that can wait for interrupts from this handler. Normally, one semaphore is sufficient.

The value returned by the registration function is a handle that is used to identify this interrupt in other functions. Once registered, the ULI handler remains registered until the program terminates or `ULI_destroy()` is called.

## Registering a Per-IRQ Handler

`ULI_register_irq()` takes two additional arguments to those already described:

- The CPU where the interrupt is occurring

- The number of the interrupt line to attach to

## Interacting With the Handler

The program process and the ULI handler synchronize their actions using the following functions:

- `ULI_sleep()`

- `ULI_wakeup()`

When the program cannot proceed without an interrupt, it calls `ULI_sleep()`, specifying the following:

- The handle of the interrupt for which to wait

- The number of the semaphore to use for waiting

Typically, only one process ever calls `ULI_sleep()` and it specifies waiting on semaphore 0. However, it is possible to have two or more processes that wait. For example, if the device can produce two distinct kinds of interrupts (such as normal and high-priority), you could set up an independent process for each interrupt type. One would sleep on semaphore 0, the other on semaphore 1.

When a ULI handler is entered, it wakes up a program process by calling `ULI_wakeup()`, specifying the semaphore number to be posted. The handler must

know which semaphore to post, based on the values it can read from the device or from program variables.

The `ULI_sleep()` call can terminate early, for example if a signal is sent to the process. The process that calls `ULI_sleep()` must test to find the reason the call returned. It is not necessarily because of an interrupt.

The `ULI_wakeup()` function can be called from normal code as well as from a ULI handler function. It could be used within any type of asynchronous callback function to wake up the program process.

The `ULI_wakeup()` call also specifies the handle of the interrupt. When you have multiple interrupting devices, you have the following design choices:

- You can have one child process waiting on the handler for each device. In this case, each ULI handler specifies its own handle to `ULI_wakeup()`.

- You can have a single process that waits on any interrupt. In this case, the main program specifies the handle of one particular interrupt to `ULI_sleep()`, and every ULI handler specifies that same handle to `ULI_wakeup()`.

## Achieving Mutual Exclusion

The program can gain exclusive use of global variables with a call to `ULI_block_intr()`. This function does not block receipt of the hardware interrupt, but does block the call to the ULI handler. Until the program process calls `ULI_unblock_intr()`, it can test and update global variables without danger of a race condition. This period of time should be as short as possible, because it extends the interrupt latency time. If more than one hardware interrupt occurs while the ULI handler is blocked, it is called for only the last-received interrupt.

There are other techniques for safe handling of shared global variables besides blocking interrupts. One important, and little-known, set of tools is the `test_and_set()` group of functions documented in the `test_and_set_bit`(9) and `test_and_clear_bit`(9) man pages and defined in the `/usr/include/asm/bitops.h` file.

# REACT System Configuration

This chapter explains how to configure restricted and isolated CPUs on a system running the REACT real-time for Linux product:

- "Configuration Overview" on page 105

- "Generating a REACT System Configuration" on page 107

- "Example `reactconf.pl` Output" on page 111

- "Running with REACT Enabled" on page 115

- "Running a Process on a Restricted CPU" on page 116

- "Dynamically Changing the Configuration" on page 116

- "Disabling and Enabling REACT" on page 117

For information about creating an external interrupt character special device file, see "Opening the Device Special File" on page 100. For information about potential problems, see Chapter 11, "Troubleshooting" on page 129.

## Configuration Overview

Initial configuration of the REACT real-time system requires that you reboot the system with certain kernel command line options. After the system is rebooted with the real-time configuration, REACT is automatically enabled and you can make changes to the real-time and bootcpusets dynamically, without additional reboots. You can disable, remove, or reenable the cpusets, or enter a new configuration. Configurable items include:

- The configured real-time CPUs (the `rtcpu` devices) and their memory nodes(mems)

  **Note:** Memory is not distributed in Altix XE systems (as opposed to Altix NUMA-based systems), therefore all memory on a single system image is always on node 0.

- The bootcpuset (`/boot`) and its memory nodes

• Interrupts, which can be redirected

**Note:** By default, the system is booted with the most restrictive bootcpuset possible (see /etc/bootcpuset.conf). When the system comes up, all but a few CPU-specific threads will be running within the bootcpuset.

Timers that were created during the boot sequence are also restricted to CPUs in the bootcpuset. If these timers have their base on a potential rtcpu, it could cause significant delays in interrupt response times.

Usually, you cannot change a timer's base CPU. When configuring a system, be careful of creating timers when the system is not configured for real-time. (For example, inserting a module when the bootcpuset is not enabled can create a timer on a CPU that could later be used for an rtcpu.) However, you can move tasks and threads.

The configuration procedure uses the following:

• /usr/sbin/reactconf.pl is a Perl script that generates a REACT configuration. It creates the /etc/bootcpuset.conf and /etc/dynamic_cpuset.conf files for saving information about the cpusets and modifies the file /etc/elilo.conf (Altix) or /boot/grub/menu.lst (Altix XE) appropriately .

**Note:** The script places backup files in /etc/elilo.conf.rtbak or /etc/menu.lst.rtbak and /etc/bootcpuset.conf.rtbak before making any changes to the original files.

REACT is enabled by default when you run the reactconf.pl script and then reboot. When the configuration is complete and REACT is enabled, the system will always come up with the same REACT CPU configuration without operator intervention.

• /etc/init.d/sgi_react is a startup script that enables or disables REACT across reboots. By default, sgi_react is set to off until you first run the reactconf.pl script.

• /sbin/react-init.sh is a bash script that determines if REACT is enabled:

– If sgi_react is set to on, it will boot with init=bootcpuset and set the CPUs allowed for the init process

– If `sgi_react` is set to `off`, it will still boot with `init=bootcpuset` but it will display the following error message to the boot screen:

```
ERROR! ERROR! ERROR! ERROR! ERROR!
Run 'reactconf.pl --enable' or
'reactconf.pl --disable'  and reboot.
Do not manipulate React through chkconfig.
```

## Generating a REACT System Configuration

Do the following:

1. Log in as `root`.

2. Decide which CPUs you want to restrict for real-time use.

   Examining `/proc/interrupts` can aid in determining these CPUs. You should choose CPUs that are not already servicing any regular interrupts beyond the per-CPU interrupts, such as for the timer and the interprocessor interrupts (IPI). You cannot use CPU 0 or the CPU on the node where the console is attached for real time.

   **Note:** The `/usr/sbin/reactconf.pl` script attempts to prevent using incorrect CPUs and redirecting certain interrupts.

3. Use the `/usr/sbin/reactconf.pl` script to configure one or more cpusets as follows:

   reactconf.pl *cpusets*

   where *cpusets* takes one of the following formats:

   • To configure a list of CPUs (you cannot specify CPU 0):

     reactconf.pl *cpu,cpu,...*

   • To configure a range of CPUs (you cannot specify CPU 0 or a descending range):

     reactconf.pl *cpu-cpu*

- To configure a mixture of the above:

  reactconf.pl *cpu,...cpu-cpu,cpu,...*

The `reactconf.pl` script will set up the following files as appropriate, based on the CPUs listed and the number of CPUs on your system:

```
/etc/bootcpuset.conf
/etc/elilo.conf (Altix) or /boot/grub/menu.lst (Altix XE)
/etc/dynamic_cpuset.conf
```

For example, on an 8-processor system, the following command line will configure CPUs 0, 1, 4, 5, and 6 in the bootcpuset, with CPUs 2, 3, and 7 restricted and isolated for real-time use:

```
[root@linux root]# reactconf.pl 2-3,7
```

The `reactconf.pl` script will append `init=/sbin/react-init.sh` to the default section in `/etc/elilo.conf` (Altix) or `/boot/grub/menu.lst` (Altix XE), which will be displayed. (See Example 9-1 on page 112 for Altix output; Altix XE output will be similar). If it is correct, answer `y` and it will be written at the end of the `reactconf.pl` script execution.

4. Configure memory nodes when prompted by the `reactconf.pl` script. (For a complete example of `reactconf.pl` output, see "Example `reactconf.pl` Output" on page 111.)

---

**Note:** Memory is not distributed in Altix XE systems (as opposed to Altix NUMA-based systems), therefore all memory on a single system image is always on node 0.

---

SGI recommends that you use the default setup, because using a different configuration could negatively impact performance. To choose the default setup, enter n:

```
Would you like to configure memory nodes for the boot
and/or real-time cpusets in a manner different then the default setup?
[y/n] n
```

By entering n, nodes will automatically be configured according to the `/var/run/cpunodemap`.

However, if you understand the performance implications and you would like to manually configure memory nodes, enter y and answer the questions that follow. For example:

```
Enter nodes for the /boot cpuset (non-rt), to allocate memory.:

List of nodes that is not in descending order.  0-1

Would you like to configure memory nodes for the rtcpuN cpusets?[y/n] y

Enter node for rtcpu1 cpuset to allocate memory on? 1

Enter node for rtcpu2 cpuset to allocate memory on? 0
   ...
```

5. Choose whether or not to redirect interrupts:

```
Do you want to move any of the IRQs? [y/n]
```

Entering n will not change anything on the system. Entering y will give you the following options:

```
1) Move IRQs round-robined into /boot cpuset.
2) Move IRQs on a case-by-case basis.
3) Do not redirect IRQs.
```

Option 1 causes IRQs that can be moved to be evenly dispersed to CPUs in the /boot cpuset. The reactconf.pl script attempts to prevent redirecting certain interrupts, including the console interrupt.

Option 2 displays a list of individual IRQs. You will be asked to specify if the IRQ is a normal or a real-time interrupt and to select the exact CPU in the respective cpusets to which the interrupt will be redirected.

For example, for a regular IRQ:

```
IRQ list: 28 29 30 31 49 51 56 62 63 65 66 67 232 233 238 239 240 254

Enter the IRQ that you would like to redirect or 'd' when done: 28

Would you like to direct IRQ 28 to a:
                1) CPU in the /boot cpuset
                2) CPU in a /rtcpu cpuset
                Selection [1 | 2]1
```

```
Your cpus configured for the /boot cpuset are: 0
Select one to redirect IRQ 28: 0

Enter the IRQ that you would like to redirect or 'd' when done: d
```

For example, for a real-time IRQ:

```
IRQ list: 28 29 30 31 49 51 56 62 63 65 66 67 232 233 238 239 240 254

Enter the IRQ that you would like to redirect or 'd' when done: 28

Would you like to direct IRQ 28 to a:
                1) CPU in the /boot cpuset
                2) CPU in a /rtcpu cpuset
                Selection [1 | 2]2

Your cpus configured for rtcpus are: 1 2 3
Select one to redirect IRQ 28:1

Enter the IRQ that you would like to redirect or 'd' when done: d
```

If you elect to redirect interrupts, the settings will take place when the reactconf.pl script is automatically run again at startup.

6. Verify that the displayed configuration information is correct and (if so) enter y to confirm its use. For example:

```
/etc/dynamic_cpuset.conf
bootcpus 0
bootmems 0
rtcpus 1,2,3
rtmems 0,1,1
IRQs 28,29,30,31,49,51,62,63,65,66,67,232,233,238,239,240,254
changed_mask 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

Does the above /etc/dynamic_cpuset.conf look OK? (y=yes)y
```

After you enter y, the /etc/dynamic_cpuset.conf file is written.

7. Reduce the Altix system flush duration if appropriate:

> **Note:** This does not apply to Altix XE systems.

> **Note:** System flush creates delays in interrupt response times of approximately 23 usec. Use this procedure only for systems running real-time applications that are sensitive to delays of this length. (There is a slight chance that heavily subscribed systems running with extremely heavy NUMAlink traffic could experience system hangs.) This setting is static across system boots.

    a.  Reboot the machine but interrupt the boot process when the `EFI Boot Manager` menu comes up. Then select the following:

```
EFI Shell [Built-in]
```

    b.  Enter power-on diagnostic (POD) mode by using the `pod` command:

```
Shell> pod
```

    c.  Enter the following (with quotes) to turn the system flush duration down to a single clock tick (the default is 8):

```
 0 000: POD SysCt (RT) Cac> setallenv SysFlushDur "1"
```

    d.  Exit POD mode:

```
0 000: POD SysCt (RT) Cac> exit
```

    e.  Reset the system to allow the new `SysFlushDur` setting to take effect and allow the system to reboot.

## Example `reactconf.pl` Output

Following is an example `reactconf.pl` output. The script modifies the bootcpusets based on the CPUs provided and changes the append value for the default entry in `/etc/elilo.conf` on Altix systems and `/boot/grub/menu.lst` on Altix XE systems. (The script output also contains information currently contained in the `/etc/elilo.conf` file, such as comments.)

> **Note:** POD mode does not apply to Altix XE systems.

**Example 9-1** reactconf.pl Output

```
[root@linux root]# /usr/sbin/reactconf.pl 1-3
cpuset: attached one pid to cpuset </>
/boot cpuset already removed.
SAL Console driver CPU 0
System not booted with bootcpuset configured.


Continuing with realtime configuration...


*********** Warning manual configuration should be done by someone ***********
*********** with a knowledge of the system configuration.       ***********


The preferred method is to have cpus allocate memory on a local node.
By default the script will configure the boot and realtime cpusets in this manner.


Would you like to configure memory nodes for the boot and/or realtime cpusets
in a manner different then the default setup? [y/n] n


/etc/elilo.conf:
# Modified by YaST2. Last modification on Mon Jan 22 09:13:08 CST 2007
timeout = 80
read-only
relocatable
default = Linux
prompt


image = /boot/vmlinuz
###Don't change this comment - YaST2 identifier: Original name: linux###

    label = Linux
    append = "  console=ttySG0 splash=silent thash_entries=2097152 init=/sbin/react-init.sh isolcpus=1,2,3"
    initrd = /boot/initrd
    root = /dev/sda11


image = /boot/vmlinuz
###Don't change this comment - YaST2 identifier: Original name: failsafe###
```

```
    label = Failsafe
    append = "ide=nodma nohalt noresume 3"
    initrd = /boot/initrd
    root = /dev/sda11


image = /boot/vmlinuz.save
###Don't change this comment - YaST2 identifier: Original name: linux###

    label = Linux-sgi-engr
    append = "  console=ttySG0 splash=silent thash_entries=2097152"
    initrd = /boot/initrd.save
    root = /dev/sda11


image = /boot/vmlinuz.save
###Don't change this comment - YaST2 identifier: Original name: failsafe###

    label = Failsafe-sgi-engr
    append = "ide=nodma nohalt noresume 3"
    initrd = /boot/initrd.save
    root = /dev/sda11


cpuset: attached one pid to cpuset
/boot cpuset already removed.
SAL Console driver CPU 0
System not booted with bootcpuset configured.

Does the above elilo.conf look OK? (y=yes overwrites /etc/elilo.conf)y

bootcpuset cpus 0
bootcpuset mems 0
realtime cpus   1 2 3
realtime mems   0 1 1
```

To minimize latency of real-time interrupts, it is often necessary to direct interrupts
to specific real-time processors. It is also necessary to direct other interrupts
away from specific real-time processors. Redirected irq's seem to have higher latency,
so only redirect irqs if you must move them away from CPUs that must be real-time.

It is preferable to select CPUs for real-time in such a way as to not require interrupt
redirection.
Do you want to move any of the IRQs? [y/n] **y**

| | CPU0 | CPU1 | CPU2 | CPU3 | | |
|---|---|---|---|---|---|---|
| 28: | 0 | 0 | 0 | 0 | LSAPIC | cpe_poll |
| 29: | 0 | 0 | 0 | 0 | LSAPIC | cmc_poll |
| 30: | 0 | 0 | 0 | 0 | SN hub | cpe_hndlr |
| 31: | 0 | 0 | 0 | 0 | LSAPIC | cmc_hndlr |
| 49: | 0 | 0 | 0 | 0 | SN hub | SN_hub_error, SN_hub_error |
| 51: | 0 | 0 | 0 | 0 | SN hub | PCIBR error, PCIBR error, PCIBR error, PCIBR error |
| 56: | 0 | 0 | 0 | 0 | SN hub | mmtimer |
| 61: | 10691 | 0 | 0 | 0 | SN hub | ide0 |
| 62: | 0 | 8599 | 0 | 0 | SN hub | libata |
| 63: | 12947 | 0 | 0 | 0 | SN hub | eth0 |
| 232: | 0 | 0 | 0 | 0 | LSAPIC | mca_rdzv |
| 233: | 2374 | 0 | 0 | 0 | SN hub | system controller events, SAL console driver |
| 238: | 0 | 0 | 0 | 0 | LSAPIC | perfmon |
| 239: | 301620 | 301334 | 301373 | 301367 | LSAPIC | timer |
| 240: | 0 | 0 | 0 | 0 | LSAPIC | mca_wkup |
| 254: | 421 | 1351 | 1681 | 1647 | LSAPIC | IPI |
| ERR: | 0 | | | | | |

Here is your current irq configuration.


Options:
     1) Move IRQs round-robined into /boot cpuset.
     2) Move IRQs on a case-by-case basis.
     3) Do not redirect IRQs.
Selection [1 | 2 | 3] 1
Not moving the mmtimer IRQ 56


/etc/dynamic_cpuset.conf
bootcpus 0
bootmems 0
rtcpus 1,2,3
rtmems 0,1,1
IRQs 28,29,30,31,49,51,61,62,63,232,233,238,239,240,254
changed_mask 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

Does the above /etc/dynamic_cpuset.conf look OK? (y=yes)**y**

```
Backup files are /etc/elilo.conf.rtbak and /etc/bootcpuset.conf.rtbak
Remember to change the SysFlushDur during reboot if not done already,
from pod mode, run 'setallenv SysFlushDur "1"',
see the REACT Realtime for Linux documentation for details.
```

```
Please reboot your system to restrict and isolate cpus.
```

## Running with REACT Enabled

When the system comes back up, `reactconf.pl --enable` is automatically run from `/etc/init.d/sgi_react`. The `--enable` option does the following:

- Creates a container `rtcpus` cpuset and cpusets labeled `rtcpuN` for each CPU that is not part of the bootcpuset (`/rtcpus/rtcpuN`). You can use these cpusets to run your real-time threads. You will find these cpusets in `/dev/cpuset`, along with the bootcpuset set up by `reactconf.pl` *cpulist* in the previous steps and stored in `/etc/dynamic_cpuset.conf`.

- Configures the cpuset's memory nodes by setting the values in `/dev/cpuset/rtcpus/rtcpuN/mems` and `/dev/cpuset/boot/mems`.

---

**Note:** Memory is not distributed in Altix XE systems (as opposed to Altix NUMA-based systems), therefore all memory on a single system image is always on node 0.

---

In step 4 above, you either specified that the memory nodes would be set up in a default manner that is optimal for performance or selected specific nodes and cpusets yourself.

- Redirects interrupts from user specification. The proper mask values are echoed in to `/proc/irq/interrupt/smp_affinity`. For more information, see "Redirect Interrupts" on page 46.

## Running a Process on a Restricted CPU

To run a process on a restricted CPU, you must invoke or attach it to a real-time cpuset (that is, a cpuset containing a CPU that does not exist in the bootcpuset, such as the /dev/cpuset/rtcpus/rtcpu*N* cpusets created above). For example:

```
[root@linux root]# cpuset --invoke /rtcpus/rtcpu4 -I ./foo
```

or:

```
[root@linux root]# echo $$ | cpuset -a /rtcpus
[root@linux root]# dplace -c 1 ./foo
```

**Note:** The dplace command example will attach the process to the second real-time CPU, not the second CPU on the system.

To attach an existing process to a restricted CPU, you can use cpuset --attach. For example, to attach your current process to CPU 2:

```
[root@linux root]# echo $$ | cpuset --attach /rtcpus/rtcpu2
```

You can also assign a process by using the following libsgirt function:

```
sysmp(MP_MUSTRUN_PID, cpu, pid)
```

For more information, see the cpuset(1), dplace(1), libsgirt(3), and libcpuset(3) man pages.

## Dynamically Changing the Configuration

After the system has been initially booted with a real-time configuration, that configuration is the configuration that will be in place across reboots and saved into /etc/dynamic_cpuset.conf. You can thereafter make changes dynamically without rebooting the system. Use the following command to form and implement a new configuration (that is, rewrite the contents of /etc/dynamic_cpuset.conf; no reboot is required):

```
reactconf.pl cpulist
```

where *cpulist* is one of the varieties explained in "Generating a REACT System Configuration" on page 107.

**Note:** If CPUs that previously were not reserved for real-time are later added to the list of CPUs reserved for real-time, holdoff situations could be encountered until the next reboot due to timers having been previously set on those CPUs.

# Disabling and Enabling REACT

You can disable and enable REACT by using the `reactconf.pl` script:

- "Disabling REACT" on page 117
- "Enabling REACT" on page 117

## Disabling REACT

To return the system to normal, do the following:

1. Stop real-time processes.

2. Enter the following:

   ```
   [root@linux root]# reactconf.pl --disable
   ```

The `--disable` option does the following:

- Removes the created `/boot` and `rtcpuN` cpusets
- Moves any changed IRQs back to their original CPUs

**Note:** If timers are created while the system is not configured for real-time, they could run on any CPU, potentially creating significant holdoffs when the system is reconfigured for real-time (unless a reboot is done first).

## Enabling REACT

If you disable REACT using the `reactconf.pl --disable` option ("Disabling REACT" on page 117), you can reenable it by using the following command:

```
[root@linux root]# reactconf.pl --enable
```

No reboot is necessary. The system will use the configuration saved in
`/etc/bootcpuset.conf` and `/etc/dynamic_cpuset.conf`.

On Altix XE systems, if the previous IRQ configuration was manual, the configuration
will default to `round-robin` after `--enable` is run.

# SGI Linux Trace

The SGI Linux Trace feature generates traces for kernel events such as interrupt handling, scheduling, and system calls. The recorded trace events can be viewed by the SGI Linux Trace tools, allowing users to analyze how kernel behavior impacts the execution of applications. SGI Linux Trace is based on the open-source Linux Trace Toolkit and has been enhanced for use on SGI ProPack systems.

SGI Linux Trace consists of the following:

- A debug kernel with traces inserted

- The tracevisualizer(1) graphical user interface

- The tracedaemon(1) command

For details, see the man pages and the tracevisualizer GUI help text in the following file:

/usr/share/doc/TraceToolkit-0.9.5-1/Help.tracevisualizer

This chapter discusses the following:

- "Installing SGI Linux Trace" on page 119

- "Using tracedaemon" on page 120

- "Using tracevisualizer" on page 124

- "Removing SGI Linux Trace" on page 127

## Installing SGI Linux Trace

To install the sltdebug kernel and SGI Linux Trace, do the following:

1. Log in as the superuser.

2. Install the kernel-sltdebug RPM:

   [root@linux root]# **rpm -Uvh kernel-sltdebug-\*.rpm**

3. Install the `TraceToolkit` RPM:

   `[root@linux root]#` **`rpm -Uvh TraceToolkit-*.rpm`**

4. Reboot the system (so that it will use the `sltdebug` kernel).

5. Mount the `relayfs` filesystem:

   `[root@linux root]#` **`mount -t relayfs relayfs /mnt/relay`**

6. Load the `slt` module:

   `[root@linux root]#` **`modprobe slt`**

**Note:** This procedure installs the `sltdebug` kernel as the default kernel. When you are done with the `sltdebug` kernel, you should remove the `kernel-sltdebug` RPM or change the symbolic links in `/boot` back to the default kernel. If you reboot into a non-`sltdebug` kernel without removing both the `kernel-sltdebug` and `TraceToolkit` RPMs, you must remove the `slt` module. See "Removing SGI Linux Trace" on page 127.

## Using `tracedaemon`

The `tracedaemon`(1) command reads data buffers of trace data provided by the kernel and writes that data to a file. This section discusses the following:

- "Invoking `tracedaemon`" on page 120

- "Recording Events with `tracedaemon`" on page 121

- "Exiting from `tracedaemon`" on page 124

### Invoking `tracedaemon`

The `tracedaemon`(1) command records each CPU's event information in a separate file that can be read by the `tracevisualizer` command. The trace files are named `slt-cpu.`*N* by default, with *N* corresponding to the CPU number, plus a `Default.proc` process/IRQ information file. The files are located in the same directory from which `tracedaemon` is run.

When you want to start recording events, you must run tracedaemon for the duration of time desired to obtain trace data. (You do not want to run tracedaemon indefinitely because you would end up with huge trace files that would consume all disk space.)

You must run the tracedaemon program as root. To allow non-root users to use the command, you can configure sudo to allow execution of tracedaemon by specific users or configure the command to set setuid root.

To start recording events, enter the following:

```
[root@linux root]# tracedaemon
```

The output files for tracedaemon will be written to the current working directory for the user of tracedaemon.

For example, suppose you have 4 CPUs. To run tracedaemon using the default options, enter the following:

```
[root@linux root]# tracedaemon
```

The following files would be output:

```
slt-cpu.0
slt-cpu.1
slt-cpu.2
slt-cpu.3
Default.proc
```

## Recording Events with tracedaemon

Table 10-1 summarizes the types of events that you can record with the tracedaemon command. For more information, see the tracedaemon(1) man page.

**Table 10-1** Trace Events Recorded by `tracedaemon`

| Event Type | Raw Trace Output Name | Event Graph Representation | Description |
|---|---|---|---|
| Interrupt | Badbreak | BBRK | The thread is entering privileged mode to handle a bad system call. |
| | Break | BRK | The thread is entering privileged mode to handle a system call. |
| | Fault | FAULT | The thread is entering privileged mode to handle a system fault. |
| | Kernel exit | (a change will appear in the graph) | The interrupt processing is complete and thread is returning to the previous user-mode processing |
| | Interrupt return | INTRT | The interrupt processing is complete and the thread is returning to the previous kernel-mode processing |
| | IRQ entry | IRQ | The running thread has been preempted to handle the top-half of an interrupt (IRQ) event. |
| | Lightweight | LTW | The thread is entering privileged mode via a lightweight mechanism such as a fastpath system call. |
| | Opfault | OPF | The thread is entering privileged mode to handle an illegal operation. |
| | Pagefault | PGF | The thread is entering privileged mode to handle a fault in the requested page. |

| Event Type | Raw Trace Output Name | Event Graph Representation | Description |
|---|---|---|---|
| | Soft IRQ | SIRQ:*IRQ number* | Soft-IRQ execution for previous IRQ event. |
| | Tasklet action | TA:*function address* | Tasklet execution for previous IRQ event. |
| | Tasklet hiaction | THA:*function address* | High-priority tasklet execution for previous IRQ event. |
| | Unaligned | UNA | The thread is entering privileged mode to handle an unaligned memory access. |
| Scheduler | Sched activate | Ac:*PID* | The thread has been moved onto the CPU run queue. The thread is in the ready-to-run state. |
| | Sched deactivate | De:*PID* | The thread has been moved off the CPU run queue. The thread is in the wait/sleeping state. |
| | Sched switch | Sw:*PID* | The CPU has been allocated to a new thread. The new thread's register state, stack, and memory mappings are switched onto the CPU. The new thread is in the running state. The previous thread will likely have been deactivated prior to the switch. |
| System call | Syscall entry | *system call name* | The thread is entering a system call. System calls can be invoked from user-mode and kernel-mode on Linux. |
| | Syscall exit | (no representation) | The thread has exited a system call handler. |

## Exiting from `tracedaemon`

To exit `tracedaemon`, do one of the following:

- Press `Ctrl-c`

- Enter the following, using the process ID (PID) for the `tracedaemon` process:

  `[root@linux root]# `**`kill -9 `***`tracedaemon_PID`*

# Using `tracevisualizer`

The `tracevisualizer` command reads the data files created by `tracedaemon`. This section discusses the following:

- "Invoking `tracevisualizer`" on page 124

- "Monitoring Trace Events" on page 125

- "Exiting `tracevisualizer`" on page 126

## Invoking `tracevisualizer`

To display the results graphically on the screen, enter the following:

`[root@linux root]# `**`tracevisualizer`**

To write the event data in ASCII format to the specified output file, enter the following:

`% `**`tracevisualizer`** *`trace_input_file`* *`proc_file`* *`output_file`*

For example:

`[root@linux root]# `**`tracevisualizer slt-cpu.1 Default.proc asciitraceoutput`**

For information about options that filter out the information written to *output_file*, see the `tracevisualizer`(1) man page.

## Monitoring Trace Events

This section discusses the following:

- "Opening a Trace File" on page 125
- "Zooming In On An Event" on page 126
- "Changing the Time Frame" on page 126
- "Seeing Process Details" on page 126
- "Seeing All Event Trace Details" on page 126

For more details, see the GUI help text.

### Opening a Trace File

To monitor events, you must open a trace file the `Default.proc` process/IRQ information file. You must have permission to read the files.

Use the following menu selection to invoke the **Open Trace** window:

> **File**
> > **> Open Trace**

**Note:** You could also click on the left-most icon in the icon bar to open a new trace. For more information about the shortcuts in the icon bar, see the GUI help.

Enter the path to a trace file and process/IRQ information file, or click the **Browse** button to open the **Select Trace File** or **> Select Proc File**, which lets you select a filename.

By default, the trace is shown in the **Event Graph** output, zoomed out to a great distance.

**Zooming In On An Event**

In most cases, the graph will be most useful if you zoom in to a smaller time span. You may also wish to resize the window. To zoom in, select the following from the menu bar or use the + spyglass icon:

> **Tools**
> > **> Zoom In**

**Changing the Time Frame**

To move to an earlier point in time, use the scroll bar at the bottom of the graph for slight changes or use the following menu selection to set the start and end time:

> **Tools**
> > **> View Time Frame ...**

**Seeing Process Details**

To see a particular process, click on the **Process Analysis** tab and select the specific process ID number on the left. The right side of the screen will display characteristics of the process and system call accounting. The `Kernel` process (process 0) summarizes the system.

**Seeing All Event Trace Details**

To see details about all trace entries, click on the **Raw Trace** tab.

## Exiting `tracevisualizer`

To exit from the `tracevisualizer` program, select:

> **File**
> > **> Exit**

## Removing SGI Linux Trace

To remove the sltdebug kernel and SGI Linux Trace, do the following:

1. Log in as the superuser.

2. Remove the kernel-sltdebug RPM:

   [root@linux root]# **rpm -ev kernel-sltdebug-\***

3. Remove the TraceToolkit RPM:

   [root@linux root]# **rpm -ev TraceToolkit-\***

4. Reboot the system (so that it uses the default kernel).

**Note:** If you remove the kernel-sltdebug RPM but not the TraceToolkit RPM, or if you reboot into a non-sltdebug kernel without removing either RPM, you must do the following:

1. Remove the slt module:

   [root@linux root]# **modprobe -r slt**

2. Unmount the relayfs filesystem at its mounting point:

   [root@linux root]# **umount /mnt/relay**

# Troubleshooting

This chapter discusses the following:

- "Diagnostic Tools" on page 129
- "Problem Removing /rtcpus" on page 131
- "Problem Removing /boot Cpuset" on page 131

## Diagnostic Tools

You can use the following diagnostic tools:

- Use the cat(1) command to view the /proc/interrupts file in order to determine where your interrupts are going:

  ```
  cat /proc/interrupts
  ```

  For an example, see Appendix A, "Example Application for SGI Altix and SGI Altix XE" on page 133.

- Use the profile.pl(1) Perl script to do procedure-level profiling of a program and discover latencies. For more information, see the profile.pl(1) man page.

- Use the following ps(1) command to see where your threads are running:

  ```
  ps -FC processname
  ```

  For an example, see Appendix A, "Example Application for SGI Altix and SGI Altix XE" on page 133.

  To see the scheduling policy, real-time priority, and current processor of all threads on the system, use the following command:

  ```
  ps -eLo pid,tid,class,rtprio,psr,cmd
  ```

  For more information, see the ps(1) man page.

- Use the top(1) command to display the largest processes on the system. For more information, see the top(1) man page.

- Use the strace(1) command to determine where an application is spending most of its time and where there may be large latencies. The strace command is a very flexible tool for tracing application activities and can be used for tracking down latencies in an application. Following are several simple examples:

  – To see the amount of time being used by system calls in the form of histogram data for a program named hello_world, use the following:

```
[root@linux root]# strace -c hello_world
execve("./hello_world", ["hello_world"], [/* 80 vars */]) = 0
Hello World
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 27.69    0.000139          28         5         3 open
 20.92    0.000105          15         7           mmap
 10.76    0.000054          54         1           write
  7.57    0.000038          13         3           fstat
  6.57    0.000033          17         2         1 stat
  5.98    0.000030          15         2           munmap
  4.58    0.000023          12         2           close
  4.38    0.000022          22         1           mprotect
  4.18    0.000021          21         1           madvise
  2.99    0.000015          15         1           read
  2.39    0.000012          12         1           brk
  1.99    0.000010          10         1           uname
------ ----------- ----------- --------- --------- ----------------
100.00    0.000502                    27         4 total
```

  – You can record the actual chronological progression through a program with the following command (line breaks added for readability):

```
[root@linux root]# strace -ttT hello_world
14:21:03.974181 execve("./hello_world", ["hello_world"], [/* 80 vars */]) = 0
..
14:21:03.976992 mmap(NULL, 65536, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
   = 0x2000000000040000 <0.000007>
14:21:03.977053 write(1, "Hello World\n", 12Hello World
) = 12 <0.000008>
14:21:03.977109 munmap(0x2000000000040000, 65536) = 0 <0.000009>
14:21:03.977158 exit_group(0)            = ?
```

The time stamps are displayed in the following format:

*hour*:*minute*:*second*.*microsecond*

The execution time of each system call is displayed in the following format:

*<second>*

---

**Note:** You can use the -p option to attach to another already running process.

---

For more information, see the strace(1) man page.

- Use the tracevisualizer command. See Chapter 10, "SGI Linux Trace" on page 119.

## Problem Removing /rtcpus

You should stop real-time processes before using the --remove option. However, the script will attempt to remove the process from the real-time cpus and display the following failure message if it was unable to move them:

```
"*** Problem removing /rtcpus/rtcpu3. cpuset***
 Try again.  If that doesn't work check /dev/cpuset/rtcpus/rtcpu3/tasks
 for potential problem PIDS;
```

## Problem Removing /boot Cpuset

Removing the REACT configuration by using reactconf.pl --remove can sometimes fail due to a large number of sleeping processes in the /boot cpuset. The following error message indicates this problem:

```
*** Problem removing /boot cpuset. ***
Try again, or check /dev/cpuset/boot/tasks for potential problem PIDS
and manually move them to root cpuset.
# echo  | cpuset --attach=/
```

To resolve the problem, run the reactconf.pl --remove command again.

# Example Application for SGI Altix and SGI Altix XE

This appendix contains an example of a multithreaded application that demonstrates using external interrupts and other aspects of REACT. It uses netlink sockets to communicate from kernel space to user space. You can use it as a performance benchmark to compare between machines or settings within REACT, such as for external interrupts, cpusets, and CPU isolation.

The application is composed of the following:

- Example kernel module, which shows examples of the following concepts:
  - Creating and building a driver with a standard miscellaneous device interface
  - Setting up and registering a external interrupt handler
  - Creating and binding a kernel thread
  - Using netlink sockets to communicate with a user application
- Example user-space application, which shows examples of the following concepts :
  - Creating, destroying, and signaling threads
  - Changing a thread's scheduling policies and priorities
  - Locking memory
  - Setting up a netlink socket to communicate with a kernel thread

This example puts the data into a matrix and multiplies two matrices together. The worker thread displays the multiplication and calculates how long it takes to multiply the two matrices together. You can modify the size of the matrix to see how it effects the time to calculate the multiplication. For example, you could use a field-programmable gate array (FPGA) to implement the multiply function in order to show how much faster it is under these circumstances than under normal calculation. You could also run on two different platforms to compare the speed of integer multiplication.

This program runs as a multithreaded process. The main process launches the following threads, sets each thread's scheduling policy and priority, and displays the thread policy and priority information:

- The receiving thread (`netlink_receive`) does the following:

    1. Tells the kernel to start the processing of interrupts (a one-time event)

    2. Locks its current and future memory (if requested).

    3. Uses the example kernel module driver to do the following:

        a. Waits for messages from the kernel netlink socket.

        b. Signals the worker thread with the data from the driver.

- The worker thread (`worker_routine`) does the following:

    1. Waits to be signaled by the receive thread for data.

    2. Fills two matrices with the data and multiplies them together. The output will be printed to the console.

    3. Calculates the time it takes for the matrices to by multiplied together.

- The interrupt handler (`extint_run`) runs when a hardware external interrupt is received. It wakes up the `bench_extintd` thread.

- The kernel thread (`bench_extintd`) gets data, sends messages with the data to the receiving thread (`netlink_receive`), and then sleeps until another interrupt occurs.

`netlink_receive` is set at a higher priority than the time-consuming `worker_routine`.

Figure A-1 describes the example. Step 1 occurs once, but steps 2 through 4 are repeated for each external interrupt.
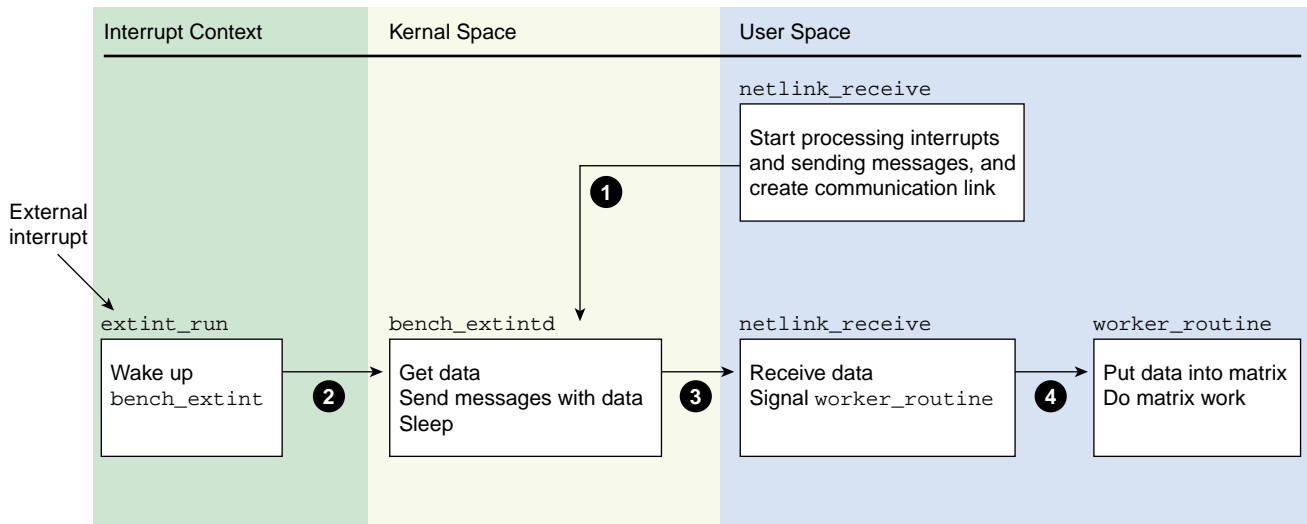
**Figure A-1** Example Work Flow

The rest of this section discusses the following:

- "Setting Up External Interrupts" on page 135
- "Building and Loading the Kernel Module" on page 136
- "Building the User-Space Application" on page 145
- "Running the Sample Application" on page 164
- "Common Files" on page 167

## Setting Up External Interrupts

To set up external interrupts, do the following:

1. Log in to the target system as root.

2. Load the ioc4_extint module:

   [root@linux root]# **modprobe ioc4_extint**

3. Insert the required information into the source, mode, and period files in the /sys/class/extint/extint0/ directory. For example:

```
[root@linux root]# echo loopback >/sys/class/extint/extint0/source
[root@linux root]# echo toggle >/sys/class/extint/extint0/mode
[root@linux root]# echo 1000000 >/sys/class/extint/extint0/period
```

For more information about external interrupts see Chapter 3, "External Interrupts" on page 17..

## Building and Loading the Kernel Module

To build the bench_extint_mod application kernel module, do the following on the target system:

1. Log in to the target system as root.

2. Ensure that the kernel-source-*.rpm RPM is installed.

3. Make the bench_example kernel module directory:

   ```
   [root@linux root]# mkdir -p /root/bench_example/kernel
   ```

4. Change to the kernel directory:

   ```
   [root@linux root]# cd /root/bench_example/kernel
   ```

5. Copy the bench_example source files to the target system. For example:

   ```
   [root@linux root]# scp bob@server:~/bench_example/kernel/* .
   ```

6. Build the bench_extint_mod.ko file:

```
[root@linux root]# make -C /lib/modules/`uname -r`/build SUBDIRS=$PWD modules
```

For more information, see the uname(1) man page.

7. Copy the bench_extint_mod.ko file to the directory defined by the uname -r output:

   ```
   [root@linux root]# cp bench_example_mod.ko /lib/modules/`uname -r`
   ```

8. Make a dependency file:

   ```
   [root@linux root]# depmod
   ```

For more information, see the depmod(8) man page.

9. Load the bench_extint_mod module:

[root@linux root]# **modprobe bench_extint_mod**

For more information, see the modprobe(8) man page.

10. Use the bench_extint_mod kernel module with the bench_example application.

---

**Note:** You must load the ioc4_extint module before the bench_extint_mod module.

---

## bench_extint.c

**Example A-1** bench_extint.c file

```
#include <linux/module.h>
#include <linux/types.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/interrupt.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/file.h>
#include <linux/cdev.h>
#include <linux/cpumask.h>
#include <linux/miscdevice.h>
#include <asm/atomic.h>
#include <asm/semaphore.h>
#include <asm/delay.h>
#include <asm/uaccess.h>

#include "../include/extint.h"
#include "../include/bench_example.h"
#include <linux/skbuff.h>
#include <linux/capability.h>
#include <linux/socket.h>
#include <linux/ip.h>
```

```
#include <net/scm.h>
#include <linux/netlink.h>
#include <net/sock.h>
#include <linux/random.h>

MODULE_LICENSE("GPL");

unsigned char buf[MAX_SIZE];
int extint_run_count;
int thread_run_count;
int sending_errors;

unsigned int int_buf;

static struct nlmsghdr *nlh;
static  struct benchmsg msg;
static int pid;

/* We might store more information here if we were measuring response
 * times at different points.
 */
static struct sock *benchsk;
static DECLARE_MUTEX(benchmark_sem);
static struct task_struct *ex_task;

int ex_ioctl(struct inode *, struct file *, unsigned int , unsigned long);
void extint_run(void *);
static int ex_sock_thread(void *nothing);
static void benchmark_rcv_sk(struct sock *sk, int len);

static struct file_operations ex_fops = {
        owner: THIS_MODULE,
        .ioctl = ex_ioctl,
};

/* The register structure for /dev/ex_misc */
static struct miscdevice ex_miscdev = {
        MISC_DYNAMIC_MINOR,
        EX_NAME,
        &ex_fops,
};
```

```
struct extint_device * store_ed;
struct extint_callout * store_ec;

int
extint_register(parm_t * ex_args, int fd) {

        struct file * filp;
        struct extint_device * ed;
        struct extint_callout * ec;

        if (!(ec = kmalloc(sizeof(struct extint_callout), GFP_KERNEL))) {
                return ENOMEM;
        }

        /* Get a reference to the EI file */
        filp = fget(fd);
        if (!filp) {
                kfree(ec);
                return EINVAL;
        }

        /* Get a reference to its EI */
        ed = file_to_extint_device(filp);
        fput(filp);
        if (ed==(void *)-EINVAL) {
                kfree(ec);
                return EINVAL;
        }

        /* Request the EI */
        ec->owner = THIS_MODULE;
        ec->function = extint_run;
        ec->data = ex_args;
        if (extint_callout_register(ed, ec)) {
                kfree(ec);
                return EINVAL;
        }
        /* Store a link to the EI for later deregistration */
        store_ed = ed;
        store_ec = ec;
```

```
        return 0;
}


int
ex_ioctl(struct inode *inode, struct file *file,
                 unsigned int ioctl_num, unsigned long ioctl_param)
{
        int error = 0;
        parm_t *ex_args = NULL;

        switch (ioctl_num) {
        case REG_EI:
                extint_run_count=0;
                thread_run_count=0;
                sending_errors=0;

                if (ex_task) {
                  /*
                   * Application died before it had a chance to clean up.
                   * Handle clean-up, then it can be run again
                   *
                   */
                        printk(KERN_ERR "bench_exintd exited abnormally\n");
                        error = -EAGAIN;
                        goto out;
                }

                ex_task = kthread_create(ex_sock_thread, NULL, "bench_exintd");
                if (IS_ERR(ex_task)) {
                        printk(KERN_ERR "bench_exint: Failed to start exintd\n");
                        return PTR_ERR(ex_task);
                }

                benchsk = (struct sock *)
                  netlink_kernel_create(NETLINK_BENCHMARK ,0,
                                        benchmark_rcv_sk,THIS_MODULE);

                if (!benchsk) {
                        printk("netlinkbech: Cannot create netlink socket\n");
                        error = -ENOMEM;
                        goto out;
```

```
            }

            ex_args = kmalloc(sizeof(parm_t ), GFP_KERNEL);
            if (ex_args == NULL) {
                    error = -ENOMEM;
                    goto out;
            }
            ex_args->cmd = 0;
            ex_args->arg1 = (unsigned long)file;
            if ((error = extint_register(ex_args, ioctl_param))) {
                    printk(KERN_INFO "error registering bench_extint\n");
                    return error;
            }
            break;
    case UNREG_EI:
out:
            printk(KERN_INFO "bench_extint ran %d, thread ran %d dropped msgs %d\n",
                    extint_run_count, thread_run_count, sending_errors);

            if (ex_task)
                    kthread_stop(ex_task);

            ex_task = NULL;

            if (benchsk)
                    sock_release(benchsk->sk_socket);

            if ((store_ed != NULL) && (store_ed != NULL)){
                    printk(KERN_INFO "ioctl unregister bench_extint\n");
                    extint_callout_unregister(store_ed, store_ec);
            }
            if (ex_args)
                    kfree(ex_args);

            break;
    case KTHREAD_BIND:
            kthread_bind(ex_task, ioctl_param);
            break;
    }
    return error;
}
```

```
static int
send_netlink_unicast(int size, void *payload, int pid)
{
        struct sk_buff *skb;
        struct nlmsghdr *nlh;
        char *data;
        unsigned char *old_tail;
        int i;

        skb = alloc_skb(NLMSG_SPACE(size), GFP_ATOMIC);

        /* possible cache effect: use skb as random number */
        i = (int) skb;

        old_tail = skb->tail;

        nlh = NLMSG_PUT(skb, 0, 0, BENCHMARK_NL_EVENT, size - sizeof(*nlh));

        data = NLMSG_DATA(nlh);

        if (payload != NULL)
                memcpy(data, payload, size);
        else
                memset(data, i, size);

        nlh->nlmsg_len = skb->tail - old_tail;

        return(netlink_unicast(benchsk, skb, pid, 0));

nlmsg_failure:
        if (skb)
                kfree(skb);
        printk("Error sending netlink packet: too exhausted to deal?\n");
        return(-1);
}

static inline void
benchmark_rcv_skb(struct sk_buff *skb)
{
        nlh = (struct nlmsghdr *) skb->data;
```

```
        pid = nlh->nlmsg_pid;
        memcpy(&msg, NLMSG_DATA(nlh), sizeof(struct benchmsg));

}

static void
benchmark_rcv_sk(struct sock *sk, int len)
{

        do {

                struct sk_buff *skb;

                down_trylock(&benchmark_sem);

                while ((skb = skb_dequeue(&sk->sk_receive_queue)) != NULL) {
                        benchmark_rcv_skb(skb);
                        kfree_skb(skb);
                }

                up(&benchmark_sem);

        } while (benchsk && benchsk->sk_receive_queue.qlen);

}

static int
ex_sock_thread(void *nothing)
{
        int i;

        while (!kthread_should_stop()){
                __set_current_state(TASK_RUNNING);
                thread_run_count++;
                for (i=0;i< msg.number;i++) {
                        get_random_bytes(&int_buf, sizeof(int_buf));
                        int_buf %= 10;
                        sprintf(buf, "%u", int_buf);
                        if (send_netlink_unicast(msg.size, buf, pid) < 0) {
                                sending_errors++;
                        }
```

```
                }
                __set_current_state(TASK_INTERRUPTIBLE);
                schedule();
        }

        return 0;
}
int
ex_init(void)
{
        int res;

        /* Create the /dev/ex-misc entry */
        if ((res = misc_register(&ex_miscdev)) < 0) {
                printk(KERN_ERR "%s: failed to register device, %d\n",
                        EX_NAME, res);
                return res;
        }

        printk(KERN_INFO "bench_extint init\n");
        return 0;
}
void
ex_exit(void)
{
        misc_deregister(&ex_miscdev);
        printk(KERN_INFO "exit\n");
}
void
extint_run(void * arg)
{

        /*
         * External interrupt handler.  Wakes the sending thread
         * and keeps a running total of how many times its been called.
         * A count of how many times the thread actually runs is kept
         * as well.  If the extint runs more then the thread, consider
         * slowing down the period of the extint.
         */

        extint_run_count++;
```

```
        if (ex_task)
                wake_up_process(ex_task);

}
module_init(ex_init);
module_exit(ex_exit);
```

## Kernel-Space `Makefile` Extract

**Example A-2** Kernel-Space Makefile Extract

```
FLAGS += -D__KERNEL__ -DMODULE -g

obj-m += bench_extint_mod.o
bench_extint_mod-objs := bench_extint.o

clean:
        rm -f *.mod.* *.o *.ko *.symvers .bench_extint*
        rm -rf *.tmp_versions
```

# Building the User-Space Application

To build the user-space module, enter the following:

```
[user@linux user]$ make
```

## User-Space `Makefile` Extract

**Example A-3** User-Space Makefile Extract

```
CC=cc
CFLAGS=-g
LDFLAGS=-lpthread -lrt
SRCS = bench_example.c libnl.c
OBJS = $(SRCS:.c=.o)
PGMS = bench

all:
        $(CC) $(CFLAGS) $(SRCS)  $(LDFLAGS) -o $(PGMS)
```

```
                          clean:
                                   rm -f $(OBJS) $(PGMS)
```

## bench_example.c

**Example A-4** bench_example.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stropts.h>
#include <time.h>
#include <errno.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <asm/types.h>
#include <sys/socket.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <linux/netlink.h>
#include <asm/unistd.h>
#include "errors.h"
#include "../include/bench_example.h"

/* name of the device, usually in /dev */
#define MOD_NAME        "extint0"

/* Safety measure for experimenting to prevent runaway threads. */
#define MAX_LOOPS        500000
#define MAX_RUN_TIME     30         /* Run for 30 seconds default */
#define MAX_INDEX        6          /* Change value to increase or decrease matrix size */
#define MAX_NUM_MSG      100        /* Can causes problems if this is too large and extint period
                                     * is too small
                                     */
float max_calctime = 0;
float min_calctime = 0x7fffffffffffff;
unsigned long total_time = MAX_RUN_TIME; /* sec */
unsigned long start;
```

```
unsigned long run_time;
int         receiving_data = 1, number_msg = 1;
int         data_size = 1024;
int         bench_run = 0;
cpuid_t     cpu = 0;
int         memlock = 0;
unsigned int  rcvbufs = 300000;
unsigned int  global_count = 0;
unsigned char buf[MAX_SIZE];
int         ex_misc_fd;

struct timespec tv_start_time;

typedef struct cond_mutex {
  pthread_mutex_t    mutex;      /* Protects access to data */
  pthread_cond_t     cond;       /* Signals change to data  */
  int                matrix_data; /* Access protected by mutex */
}cond_mutex_t;

cond_mutex_t data = {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0};

typedef struct  lmatx_struct {
    long lmatx [MAX_INDEX][MAX_INDEX];
    int index;
} lmatx_t;

char * usage =
"usage: bench [-hm]  [-b # of msgs] [-t seconds] [-s buffer MB] \n"
"       -h  print usage instructions\n"
"       -m  lock memory\n"
"       -b# socket benchmark mode (# of msgs)\n"
"       -s# size of buffers (for benchmark mode)\n"
"       -r# cpu where the kthread will run\n"
"       -t# total run time (secs)\n";

static void
usage_exit(void)
{
        fprintf(stderr, "%s", usage);
        exit(0);
}
```

```
void
print_matrix(lmatx_t print_matx)
{
  int out_index = 0;
  int in_index  = 0;

  for (out_index=0; out_index < MAX_INDEX; out_index++) {
        for (in_index=0; in_index < MAX_INDEX; in_index++) {
              printf(" %d,",print_matx.lmatx[out_index][in_index]);
        }
        printf("\n");
  }

}

lmatx_t
multiply (lmatx_t multi_a, lmatx_t multi_b )
{
  int out_index = 0;
  int in_index  = 0;
  int k =0;
  lmatx_t result;

  /* initialize result matrix */
  for (out_index=0; out_index < MAX_INDEX; out_index++) {
       for (in_index=0; in_index < MAX_INDEX; in_index++) {
         result.lmatx[out_index][in_index]= 0,0;
       }
  }

  for(out_index=0; out_index < MAX_INDEX; out_index++) {
      for(in_index=0; in_index < MAX_INDEX; in_index++) {
          for(k=0; k < MAX_INDEX; k++) {
                  result.lmatx[out_index][in_index] += (multi_a.lmatx[out_index][k] * multi_b.lmatx[k][in_index]);
          }
      }
  }
   return result;
}
```

```
float
calc_time(struct timespec *tv_start, struct timespec *tv_end)
{
        float time;
        long sec, nsec;

        sec = tv_end->tv_sec - tv_start->tv_sec;
        if (sec == 0) {
                nsec = tv_end->tv_nsec - tv_start->tv_nsec;
                goto out;
        }

        if (tv_end->tv_nsec < tv_start->tv_nsec) {
                sec--;
                nsec = (tv_end->tv_nsec+1000000000) - (tv_start->tv_nsec);
                if (nsec < 0) printf("BUG: calc_rate\n");
        } else {
                nsec = tv_end->tv_nsec - tv_start->tv_nsec;
        }

out:    time = (float) (sec + (nsec * 0.000000001));

        return time;
}


void
output_rate_mbits(int size, int numbytes, int drop, int nmsg, float time, char *type)
{
        if ((global_count % 500) == 1) {
          printf("%s: %d pkts of %d bytes (drop %d) in %f seconds (%.2f Mbits/s) (%.2f pkts/s)\n",
                type, nmsg, size, drop, time, (numbytes*8/time) / 1024000, nmsg/time);
        }
}

/* routine to do work of receive thread */
void *
worker_routine( void *arg )
{
  int status;
```

```
int inside = 0;
int outside = 0;
int matrix_full;
float  matx_calctime = 0;
struct timespec tv_matx_stime, tv_matx_etime;

tv_matx_stime.tv_sec  = 0;
tv_matx_stime.tv_nsec = 0;

tv_matx_etime.tv_sec  = 0;
tv_matx_etime.tv_nsec = 0;


/*
 * matx_multiplier[0] ==  1st matrix multiplier (A)
 * matx_multiplier[1] ==  2nd matrix multiplier (B)
 * matx_multiplier[2] ==  result (A) * (B)
 *
 */
lmatx_t  matx_multiplier[3] = {{0},0};
int stx = 0; /* 0-2 to control struct index */

/*receive not done and matrix not full */
KEEP_LOOPING:

do {
    if(!bench_run) {
        matrix_full=0;
        status = pthread_mutex_lock(&data.mutex);
        if (status !=0)
              err_abort (status, "Lock mutex");


        status = pthread_cond_wait(&data.cond, &data.mutex);
        if (status !=0)
              err_abort (status, "Cond Wait");

        matx_multiplier[stx].lmatx[outside][inside] = data.matrix_data;
        inside++;

        status = pthread_cond_signal (&data.cond);
```

```
if (status !=0)
     err_abort (status, "Signal condition");



status = pthread_mutex_unlock(&data.mutex);
if (status !=0)
     err_abort (status, "Unlock mutex");


/*
 * Time how long it takes to do the multiplication of the
 * the matrices.  Change MAX_INDEX to increase the size
 * of the matrices.  The multiply function could be done
 * via a FPGA then see how much faster it does the multiplication.
 */
 if (inside == MAX_INDEX ) {
   inside = 0;
   outside++;
   if (outside == MAX_INDEX) {
     matrix_full = 1;
     if (matx_multiplier[stx].index ==  0) {
             /*
              * Don't print first matrix if time has expired.
              * There is no matrix to multiply it to
              */
             if (run_time < total_time ) {
               print_matrix(matx_multiplier[stx]);
               printf("     X \n");
             }
             stx++;
             matx_multiplier[stx].index = 1;
     } else {
             print_matrix(matx_multiplier[stx]);
             /* Use Matrix Data */
             printf("==================================== \n");

             clock_gettime(CLOCK_REALTIME,&tv_matx_stime);

             matx_multiplier[2] =  multiply(matx_multiplier[0],matx_multiplier[1]);
             matx_multiplier[stx].index = 0;
             stx--;
```

```
                      clock_gettime(CLOCK_REALTIME,&tv_matx_etime);

                      /* calculation time is in nsec */
                      matx_calctime = calc_time(&tv_matx_stime, &tv_matx_etime);

                      if (matx_calctime > max_calctime) {
                        max_calctime = matx_calctime;
                      }
                      if (matx_calctime < min_calctime) {
                        min_calctime = matx_calctime;
                      }
                      /* print result matrix */
                      print_matrix(matx_multiplier[2]);
                      printf("Calculation time for matrix multiplication %f seconds.\n",matx_calctime );
                      printf("//////////////////////////////////////////////////////////\n");
                  }
                  outside = 0;
              }
            }
      } /* bench_run */
          if (global_count > MAX_LOOPS)
                goto done;

  }  while ( run_time < total_time);

  if(!bench_run) {
    if (!matrix_full)
        goto KEEP_LOOPING;
  }

 done:
  printf("sends = %d, with %d msgs per send in %ld seconds.\n",global_count, number_msg, total_time);
  printf("Worker routine..done receiving \n");
  receiving_data = 0;
  pthread_exit(NULL);
}



/*
 * This routine receives the data from the driver
```

```
 * and signals the worker thread w/ the data
 */
void *
netlink_receive(void *arg)
{
        int sockfd, addrlen;
        struct sockaddr_nl local, peer;
        struct benchmsg msg;

        /*
         * Give worker a chance to get started.
         * If extint period is set too fast,
         * this thread will be receiving and worker
         * doesn't get a chance to start
         */
        printf("Waiting...\n");
        sleep(1);

        sockfd = libnetlink_create_socket(NETLINK_BENCHMARK);
        if (sockfd < 0) {
                perror("socket:");
                exit(0);
        }

        if (libnetlink_bind(sockfd, &local, &peer) < 0) {
                printf("cannot bind\n");
                exit(-1);
        }

        addrlen = sizeof(peer);

        if (setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, (void *) &rcvbufs, sizeof(rcvbufs)) < 0) {
                perror("setsockopt");
                exit(-1);
        }



        msg.type = 0;
        msg.size = data_size;
        msg.number = number_msg;
```

```
        msg.timestamp_sec = 0;
        msg.timestamp_usec = 0;

        do{
           if (global_count > MAX_LOOPS)
                 /* This will allow worker to exit safely */
                 run_time = MAX_RUN_TIME;

           global_count++;
           netlink_kernel_to_user(sockfd, &peer, &msg);
        } while(receiving_data);

        printf("netlink_receive thread exit\n");
        pthread_exit(NULL);

}


/*
 * kernel_netlink_send - receive netlink messages from kernel space
 * @fd: socket descriptor
 * @peer: socket structure which contains information about the peer
 * @msg: benchmark message
 * @debug_level: print debug if needed
 */

int
netlink_kernel_to_user(int fd, struct sockaddr_nl *peer, struct benchmsg *msg)
{
        struct timeval tv;
        struct timespec tv_bench1, tv_bench2, tv_current;
        int addrlen = sizeof(*peer);
        int status, bytes, numbytes = 0, drop = 0, i;
        fd_set read_fds,netlink;
        float time = 0;
        int nmsg = msg->number;
        int size = msg->size;
        struct nlmsghdr *nlh;

        FD_ZERO(&netlink);
        FD_SET(fd, &netlink);
```

```
/* hey kernel, start sending me packets! */
if (libnetlink_send_netlink_message(fd, msg, sizeof(*msg)) < 0) {
        perror("send_netlink:");
        exit(-1);
}

if(bench_run) {

  /* start the timer... */
  clock_gettime(CLOCK_REALTIME,&tv_bench1);
}

for (i=0; i< nmsg; i++) {

        tv.tv_sec = 1;
        tv.tv_usec = 0;
        read_fds = netlink;

        /* we have a winner */
        if (select(fd+1, &read_fds, NULL, NULL, &tv) < 0) {
                printf("error en select\n");
                return -1;
        }

        if (!FD_ISSET(fd, &read_fds)) {
                /* 1 packet drop means 1 second */
                printf("packet drop! (%d)\n", i);
                drop++;
                continue;
        }

        /* receiving packets! */
        /* we receive the packet + nlmsghdr which is 16 bytes */
        if ((bytes = recvfrom(fd, buf, MAX_SIZE-1, 0,(struct sockaddr *)&peer, &addrlen)) <0) {
                perror("recvfrom");
                return -1;
        }
        nlh = (struct nlmsghdr *) buf;
        bytes -= sizeof(struct nlmsghdr);
        numbytes += bytes;
```

```
              /*
               * Calculate condition to signal worker that shared data is ready
               * And signal worker pthread
               */

              clock_gettime(CLOCK_REALTIME, &tv_current);

              /* elapsed time so far is in sec */
              run_time =  calc_time(&tv_start_time, &tv_current);
              if (!bench_run) {
                status = pthread_mutex_lock(&data.mutex);
                if (status !=0)
                        err_abort (status, "Lock mutex");

                /*
                 * Get data from driver and signal
                 * woker thread to manipulate it
                 */
                data.matrix_data =atoi(NLMSG_DATA(nlh));

                status = pthread_cond_signal (&data.cond);
                if (status !=0)
                      err_abort (status, "Signal condition");


                status = pthread_mutex_unlock(&data.mutex);
                if (status !=0)
                      err_abort (status, "Unlock mutex");
              }
      }

      if(bench_run) {
        /* stopping timer */
        clock_gettime(CLOCK_REALTIME,&tv_bench2);
        time = calc_time(&tv_bench1, &tv_bench2);
        /* Only print out every 500 times */
        output_rate_mbits(size, numbytes, drop, nmsg, time, "kernel -> user");
      }
      return 0;
}
```

```
int
main(int argc, char **argv)
{
        int extint_fd, status, thread_policy, max_priority, min_priority, opt;
        int bind_thread = 0;
        pthread_t pt_receive, pt_worker;
        pthread_attr_t thread_attr;
        struct sched_param thread_param;

        static char* opt_string = "mb:t:s:r:h";
        while ((opt = getopt(argc, argv, opt_string)) >= 0) {
                switch (opt){
                case 'm':
                        memlock = 1;
                        break;
                case 'b':
                        bench_run = 1;
                        number_msg = atoi(optarg);
                        if (number_msg > MAX_NUM_MSG)
                                number_msg = MAX_NUM_MSG;
                        break;
                case 't':
                        total_time = atoi(optarg);
                        if (total_time > MAX_RUN_TIME)
                                total_time = MAX_RUN_TIME;
                        break;
                case 's':
                        data_size = atoi(optarg);
                        break;
                case 'r':
                        bind_thread = 1;
                        cpu = atoi(optarg);
                        break;
                case 'h':
                default:
                        usage_exit();
                }
        }

        /* Get the file descriptor for extint, pass to driver */
        if ((extint_fd = open("/dev/"MOD_NAME,O_RDONLY)) == -1) {
```

```
  perror("open /dev/extint0");
  exit(1);
}

/* Open the module device */
if ((ex_misc_fd = open("/dev/"EX_NAME,O_RDWR)) == -1) {
  perror("open /dev/ex_misc");
  exit(1);
}
/* Do not allow paging if selected */
if (memlock &&
    (mlockall(MCL_CURRENT | MCL_FUTURE) < 0)) {
        perror("memlock");
        exit(1);
}
/* Register the external interrupt */
if (ioctl(ex_misc_fd, REG_EI, extint_fd) < 0) {
        close(extint_fd);
        close(ex_misc_fd);
        perror("ioctl /dev/ex_misc");
        exit(1);
}
/* Bind kernel thread to a cpu if passed in */
if (bind_thread) {
        if (ioctl(ex_misc_fd, KTHREAD_BIND, cpu) < 0)
                perror("ioctl /dev/ex_misc");
}

clock_gettime(CLOCK_REALTIME,&tv_start_time);

/* Get info about policies and create Receive thread */
status = pthread_attr_init(&thread_attr);
status = pthread_attr_getschedpolicy (&thread_attr, &thread_policy);
if(status != 0)
        err_abort (status, "Get policy");

status = pthread_attr_getschedparam (&thread_attr, &thread_param);
if(status != 0)
        err_abort (status, "Get sched param");

printf ("Default policy is %s, priority is %d\n",
```

```
            (thread_policy == SCHED_FIFO ? "FIFO":
             (thread_policy == SCHED_RR ? "RR" :
              (thread_policy == SCHED_OTHER ? "OTHER":
             "unknown"))), thread_param.sched_priority);

    status = pthread_attr_setschedpolicy (&thread_attr, SCHED_FIFO);
    if(status != 0)
            err_abort (status, "Unable to set SCHED_FIFO policy.\n");


    min_priority = sched_get_priority_min (SCHED_FIFO);
    if (min_priority == -1)
            errno_abort ("Get SCHED_FIFO min priority");


    max_priority = sched_get_priority_max (SCHED_FIFO);
    if (max_priority == -1)
            errno_abort ("Get SCHED_FIFO max priority");

    thread_param.sched_priority = (min_priority + max_priority)/3;

    printf("SCHED_FIFO priority range is %d to %d: using %d\n",
          min_priority,
          max_priority,
          thread_param.sched_priority);

    status = pthread_attr_setschedparam (&thread_attr, &thread_param);
    if(status != 0)
             err_abort (status, "Set params");

    printf ("Creating thread at FIFO/%d\n", thread_param.sched_priority);

    status = pthread_attr_setinheritsched (&thread_attr, PTHREAD_EXPLICIT_SCHED);
    if(status != 0)
            err_abort (status, "Set inherit");

    status = pthread_create(&pt_receive, &thread_attr, netlink_receive, NULL);
    if (status != 0)
            err_abort(status,"Create thread");

    /* Create worker thread */
```

```
        status = pthread_attr_init(&thread_attr);
        status = pthread_attr_setschedpolicy (&thread_attr, SCHED_RR);
        if(status != 0)
                err_abort (status, "Unable to set SCHED_RR policy.\n");


        thread_param.sched_priority = min_priority;


        status = pthread_create(&pt_worker, &thread_attr, worker_routine, NULL);
        if (status != 0)
                err_abort(status,"Create thread");


        status = pthread_join(pt_worker,NULL);
        if (status != 0)
                err_abort(status,"Join thread send");


        pthread_cancel(pt_receive);
        if (status != 0)
                err_abort(status,"Cancel thread");


        status = pthread_join(pt_receive,0);
        if (status != 0)
                err_abort(status,"Join thread receive");


        /* Unregister the external interrupt */
        if (ioctl(ex_misc_fd, UNREG_EI, 0) < 0)
                perror("ioctl /dev/ex_misc");


        close(extint_fd);
        close(ex_misc_fd);


        return 0;
}
```

## errors.h

**Example A-5** errors.h

```
#ifndef __errors_h
#define __errors_h

#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifdef DEBUG
# define DPRINTF(arg) printf arg
#else
# define DPRINTF(arg)
#endif

#define err_abort(code,text) do { \
    fprintf (stderr, "%s at \"%s\":%d: %s\n", \
             text, __FILE__,__LINE__, strerror (code)); \
    abort (); \
  } while (0)

#define errno_abort(text) do  {                 \
    fprintf (stderr, "%s at \"%s\":%d: %s\n", \
             text, __FILE__,__LINE__, strerror (errno)); \
     abort;\
   } while (0)

#endif
```

**libnl.c**

**Example A-6** libnl.c

```
#include <asm/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>
#include <string.h>

/**
 * libnetlink_create_socket - Create a netlink socket
 * @id: identifier of the new netlink socket
 *
 * Up to 32 netlink sockets could be register, this parameter is set in
 * compilation time and it could be increased. Some id's are already in
 * use, see netlink.h in kernel sources.
 */
int
libnetlink_create_socket(int id)
{
        int fd;

        fd = socket(PF_NETLINK, SOCK_RAW, id);
        if (fd < 0) {
                perror("libnetlink_create_socket:");
                return(-1);
        }

        return(fd);
}

/**
 * libnetlink_bind - Bind a descriptor to their sockets structures
 * @fd: descriptor associated to the socket
 * @local: structure which contains the information of the local peer
 * @peer: structure which contains the information of the remote peer
 *
 * FIXME: since we are working with broadcast netlink sockets, group
 * should be passed as parameter!!
 */

int
```

```
libnetlink_bind(int fd, struct sockaddr_nl *local, struct sockaddr_nl *peer)
{
        int status;

        memset(local, 0, sizeof(struct sockaddr_nl));
        local->nl_family = AF_NETLINK;
        local->nl_pid = getpid();
        local->nl_groups = 0;
        local->nl_pid = getpid();

        memset(peer, 0, sizeof(struct sockaddr_nl));
        peer->nl_family = AF_NETLINK;
        peer->nl_pid = 0;
        peer->nl_groups = 0;
        peer->nl_pid = 0;

        status = bind(fd, (struct sockaddr *) local, sizeof(*local));
        if (status == -1) {
                perror("bind:");
                close(fd);
                return(-1);
        }

        return(1);
}

/**
 * libnetlink_send_netlink_message - send a netlink message to kernel
 * @fd: descriptor associated to the socket
 * @data: payload of the message
 * @size: size of the payload
 */
int
libnetlink_send_netlink_message(int fd, void *data, int size)
{
        struct nlmsghdr nlh;
        struct msghdr msg;
        struct iovec iov[2];
        size_t tlen;
        int ret;
```

```
        memset(&nlh, 0, sizeof(nlh));
        nlh.nlmsg_flags = NLM_F_REQUEST;
        nlh.nlmsg_type = 0x20;
        nlh.nlmsg_pid = getpid();

        iov[0].iov_base = &nlh
        iov[0].iov_len = sizeof(nlh);
        iov[1].iov_base = data;
        iov[1].iov_len = size;

        tlen = sizeof(nlh) + size;

        //msg.msg_name = (void *) &peer
        //msg.msg_namelen = sizeof(peer);
        msg.msg_name = NULL;
        msg.msg_namelen = 0;
        msg.msg_iov = iov;
        msg.msg_iovlen = 2;
        msg.msg_control = NULL;
        msg.msg_controllen = 0;
        msg.msg_flags = 0;
        nlh.nlmsg_len = tlen;

        if ((ret = sendmsg(fd, &msg, 0)) < 0)
                perror("sendmsg:");

        return(ret);
}
```

## Running the Sample Application

You can run the bench_example application in the following modes:

- *Matrix multiply mode* receives data from the kernel module and puts that data into a matrix. After two matrices are full, it multiplies them together and calculates the amount of time taken for the calculation.

- *Netlink socket bench mode* causes the application to send multiple messages from kernel space to user space during one iteration. The number of messages sent per

iteration depends upon notification from the user application to start sending messages.

Do the following:

- Ensure that you have the bench_extint_mod module loaded by using the lsmod(1) command, which should show it in the module list. For example:

```
[root@linux root]# lsmod
           Module                 Size  Used by
           bench_extint_mod       546232  0
           ioc4_extint            27272  0
           ioc4                   24704  1 ioc4_extint
           extint                 32008  2 bench_extint_mod,ioc4_extint
```

  If the output does not include bench_extint_mod, follow the instructions in "Building and Loading the Kernel Module" on page 136.

- Execute the bench command as desired.

  The bench command has the following options:

  | | |
  |---|---|
  | -h | Prints usage instructions |
  | -m | Locks memory |
  | -s*size* | Specifies the size of buffers in bytes (for network socket bench mode). The default is 1024. You can vary the size of the buffers to see the impact on performance. |
  | -r*cpu* | Specifies the CPU where the kthread will run. |
  | -t*seconds* | Specifies the total run time in seconds, with a maximum of 30 seconds. The default is 30. |

## Matrix Multiply Mode Examples

To run in Matrix Multiply for 30 seconds:

```
[root@linux root]# ./bench -t30
```

To run with memory locked and kthread running on CPU 2:

```
[root@linux root]# ./bench -m -r2 -t30
```

To run on a real-time CPU 3 and attach a shell process to a real-time CPU:

```
[root@linux root]# echo $$ | cpuset --attach=/rtcpus/rtcpu3
[root@linux root]# ./bench -m -r2 -t30
```

**Note:** You can also implement this by using the cpuset API in the application.

## Netlink Socket Benchmark Mode Examples

To run with 50 messages being sent each iteration and with memory locked:

```
[root@linux root]# ./bench -m -t30 -b50
```

To run on real-time CPU 3, with the kthread running on CPU 2 and abuffer size of 512 bytes:

```
[root@linux root]# echo $$ | cpuset --attach=/rtcpus/rtcpu3
[root@linux root]# ./bench -m -t30 -b50 -r2 -s512
```

If you have multiple terminals open, you can run the following tail(1) and ps(1) commands to see where things are running:

```
tail -f /var/log/messages
Feb 16 08:54:05 dewberry kernel: bench_extint init
Feb 16 08:54:40 dewberry kernel: bench_extint ran 14958, thread ran 14958 dropped msgs 0
Feb 16 08:54:40 dewberry kernel: ioctl unregister bench_extint
```

```
ps -eLF
UID        PID  PPID   LWP  C NLWP   SZ   RSS PSR STIME TTY          TIME CMD
root     28470    15 28470  3    1    0     0   2 08:54 ?        00:00:07 [bench_exintd]
root     28515  6279 28515  0    3 5421 18620   3 08:56 pts/1    00:00:00 ./bench -m -t30 -b50 -r2
root     28515  6279 28516 12    3 5421 18620   3 08:56 pts/1    00:00:02 ./bench -m -t30 -b50 -r2
root     28515  6279 28517  0    3 5421 18620   3 08:56 pts/1    00:00:00 ./bench -m -t30 -b50 -r2
```

# Common Files

This section contains files that are used by the user application and the kernel module.

## bench_example.h

**Example A-7** bench_example.h

```
#define EX_NAME "ex_misc"
#define REG_EI            1
#define UNREG_EI          2
#define KTHREAD_BIND      3
#define NETLINK_TEST      17
#define BENCH_GROUP       1
#define NETLINK_BENCHMARK 21
#define BENCHMARK_NL_EVENT 141
#define MAX_SIZE          65335*8


struct benchmsg {
        int type;
        int size;
        int number;
        long timestamp_sec;
        long timestamp_usec;
};


typedef struct parm_s {
        unsigned long cmd;
        unsigned long arg1;
        unsigned long random_dat;
        unsigned long ed;
        unsigned long ec;
} parm_t;

typedef signed short    cpuid_t;        /* cpuid */
```

**extint.h**

**Example A-8** extint.h

```
/*
 * This file is subject to the terms and conditions of the GNU General Public
 * License.  See the file "COPYING" in the main directory of this archive
 * for more details.
 *
 * Copyright (c) 2005,2007 Silicon Graphics, Inc.  All Rights Reserved.
 */

/* External interrupt control abstraction */

#ifndef _LINUX_EXTINT_H
#define _LINUX_EXTINT_H

#include <linux/device.h>
#include <linux/types.h>
#include <linux/wait.h>

struct extint_device;

struct extint_properties {
        /* Owner module */
        struct module *owner;

        /* Get/set generation mode */
        ssize_t (*get_mode)(struct extint_device * ed, char *buf);
        ssize_t (*set_mode)(struct extint_device * ed, const char *buf,
                            size_t count);

        /* Get generation mode list */
        ssize_t (*get_modelist)(struct extint_device * ed, char *buf);

        /* Get/set generation period */
        unsigned long (*get_period)(struct extint_device * ed);
        ssize_t (*set_period)(struct extint_device * ed, unsigned long period);

        /* Get low-level provider name */
        ssize_t (*get_provider)(struct extint_device *ed, char *buf);
```

```
        /* Generation period quantum */
        unsigned long (*get_quantum)(struct extint_device * ed);

        /* Get/set ingest source */
        ssize_t (*get_source)(struct extint_device * ed, char *buf);
        ssize_t (*set_source)(struct extint_device * ed, const char *buf,
                              size_t count);

        /* Get ingest source list */
        ssize_t (*get_sourcelist)(struct extint_device * ed, char *buf);
};

struct extint_device {
        /* Current status of device */
        int state;

        /* Semaphore protects 'props' field.  If 'props' is NULL, the
         * driver that registered this device has been unloaded, and
         * if class_get_devdata() points to something in the body of
         * that driver, it is also invalid.
         */
        struct semaphore sem;
        struct extint_properties *props;        /* Downcalls */


        /* A list of callouts to invoke whenever this device ingests
         * an interrupt.
         */
        struct list_head callouts;
        spinlock_t callouts_lock;

        /* A list of all the open files for this device */
        struct list_head files;
        struct semaphore files_lock;

        /* File/process allowed to update device attributes */
        atomic_t serial;        /* Next unique ID to assign to a file */
        atomic_t updater;       /* ID which holds flock */
        pid_t updater_tgid;     /* Process which can update attributes */
        wait_queue_head_t updater_wait; /* Tasks waiting for updater clear */
        wait_queue_head_t release_wait; /* Tasks waiting for file release */
```

```
        /* Mappable counter page support */
        struct cdev counter_cdev;              /* Character dev */
        struct page *counter_page;             /* Mappable page... */
        unsigned long *counter_addr;           /* ...and mapped addr */
        wait_queue_head_t counter_queue;       /* Poll/select queue */

        /* The class device structure */
        struct class_device class_dev;

        /* Device number of abstracted counter */
        dev_t devt;

        /* Private device data for device-specific drivers */
        void* devdata;
};

static inline void* extint_get_devdata(const struct extint_device *ed)
{
        return ed->devdata;
}

static inline void extint_set_devdata(struct extint_device *ed, void* devdata)
{
        ed->devdata = devdata;
}

struct extint_callout {
        struct list_head list;
        struct module *owner;          /* Callout function and data owner */
        void (*function) (void *);     /* Callout to invoke */
        void *data;                    /* Passed to callout */
};

extern struct extint_device *extint_device_register(struct extint_properties
                                                *ep, void *devdata);
extern void extint_device_unregister(struct extint_device *ed);

/* Used by other kernel modules to request a function be invoked each
 * time an interrupt is ingested
 */
```

```
extern struct extint_device *file_to_extint_device(const struct file *filp);
extern int extint_callout_register(struct extint_device *ed,
                                   struct extint_callout *ec);
extern void extint_callout_unregister(struct extint_device *ed,
                                      struct extint_callout *ec);

/* Used by external interrupt low-level drivers to trigger invocation
 * of per-interrupt actions.
 */
extern void extint_interrupt(struct extint_device *ed);

#endif
```

# High-Resolution Timer Example

**Note:** This appendix does not apply to Altix XE servers.

Example B-1 demonstrates the use of SGI high-resolution timers. It will run both high-resolution and normal-resolution POSIX timers in both relative mode and absolute mode.

**Example B-1** High-Resolution Timer

```
/***************************************************************************
 *                                                                         *
 * This sample program demonstrates the use of SGI high resolution timers  *
 * in SGI REACT.  It will run both high resolution and normal resolution   *
 * POSIX timers in both relative mode and absolute mode.                   *
 *                                                                         *
 * This sample program requires the REACT rtgfx kernel, which supports high *
 * resolution timers via clock id CLOCK_SGI_CYCLE.                         *
 *                                                                         *
 * Note that the clock id CLOCK_REALTIME is RTC (high) resolution with     *
 * clock_gettime() on SGI systems, but is jiffies resolution with          *
 * timer_create(), clock_getres() and other POSIX time calls.             *
 *                                                                         *
 *                                                                         *
 * A simple way to build this sample program is:                           *
 *   cc -o timer_sample timer_sample.c -lrt                                *
 *                                                                         *
 * Invocation example (500 usec timer):                                    *
 *   ./timer_sample 500                                                    *
 *                                                                         *
 * Invocation example (500 usec timer on realtime cpu 2):                  *
 *   cpuset --invoke=/rtcpu2 --invokecmd=./timer_sample 500                *
 *                                                                         *
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
```

```
#include <time.h>
#include <errno.h>
#include <asm/unistd.h>
#include <pthread.h>
#include <strings.h>
#include <sys/time.h>
#include <getopt.h>
#include <libgen.h>

#define CLOCK_SGI_CYCLE 10      /* value of CLOCK_SGI_CYCLE from kernel */

struct timespec time1;
int flag;

/* Timer has triggered, get current time and indicate completion */
void sigalarm(int signo)
{
        clock_gettime(CLOCK_REALTIME,&time1);
        flag = 1;
}


int timer_test(int clock_id, long nanosec) {
        struct itimerspec ts;
        struct sigevent se;
        struct sigaction act;
        sigset_t sigmask;
        struct timespec sleeptime, time0;
        timer_t timer_id;
        long i;
        int signum = SIGRTMAX;
        int status;

        /* Set up sleep time for loops: */
        sleeptime.tv_sec = 1;
        sleeptime.tv_nsec = 0;

        /* Set up signal handler: */
        sigfillset(&act.sa_mask);
        act.sa_flags = 0;
        act.sa_handler = sigalarm;
```

```
sigaction(signum, &act, NULL);

/* Set up timer: */
memset(&se, 0, sizeof(se));
se.sigev_notify = SIGEV_SIGNAL;
se.sigev_signo = signum;
se.sigev_value.sival_int = 0;
status = timer_create(clock_id, &se, &timer_id);
if (status < 0) {
        perror("timer_create");
        return -1;
}


/* Start relative timer: */
ts.it_value.tv_sec = nanosec / 1000000000;
ts.it_value.tv_nsec = (nanosec % 1000000000);
ts.it_interval.tv_sec = 0;
ts.it_interval.tv_nsec = 0;

printf("Waiting for timeout of relative timer: ");
fflush(stdout);
flag = 0;
/* Get current time for reference */
clock_gettime(CLOCK_REALTIME,&time0);
/*
 * There will be some latency between getting the start time above,
 * and setting the relative time in timer_settime.
 */
status = timer_settime(timer_id, 0, &ts, NULL);
if (status < 0) {
        perror("timer_settime");
        return -1;
}

/* Loop waiting for timer to go off */
while (!flag) nanosleep(&sleeptime, NULL);
if (time1.tv_nsec < time0.tv_nsec)
        printf("Total time=%luns\n",
                1000000000LL - (time0.tv_nsec - time1.tv_nsec) +
                ((time1.tv_sec - time0.tv_sec -1)*1000000000LL));
```

```
        else
                printf("Total time=%luns\n",
                          time1.tv_nsec - time0.tv_nsec +
                          ((time1.tv_sec - time0.tv_sec)*1000000000LL));


        /* Start absolute timer: */
        printf("Waiting for timeout of absolute timer: ");
        fflush(stdout);
        flag = 0;
        /* Get current time and add timeout to that for absolute time */
        clock_gettime(CLOCK_REALTIME,&time0);
        i = time0.tv_nsec + (nanosec % 1000000000);
        ts.it_value.tv_nsec = i % 1000000000;
        ts.it_value.tv_sec = (time0.tv_sec + (nanosec / 1000000000)) +
                          (i / 1000000000);
        /* There should be less latency than what we saw above */
        status = timer_settime(timer_id, TIMER_ABSTIME, &ts, NULL);
        if (status < 0) {
                perror("timer_settime");
                return -1;
        }

        /* Loop waiting for timer to go off */
        while (!flag) nanosleep(&sleeptime, NULL);
        if (time1.tv_nsec < time0.tv_nsec)
                printf("Total time=%luns\n",
                          1000000000LL - (time0.tv_nsec - time1.tv_nsec) +
                          ((time1.tv_sec - time0.tv_sec -1)*1000000000LL));
        else
                printf("Total time=%luns\n",
                          time1.tv_nsec - time0.tv_nsec +
                          ((time1.tv_sec - time0.tv_sec)*1000000000LL));


        /* Cleanup */
        timer_delete(timer_id);

        return 0;
}
```

```
int main(int argc, char *argv[])
{
        long timeout;

        if (argc < 2) {
                printf("usage: %s <timeout usec>\n", basename(argv[0]));
                return -1;
        }

        timeout = atol(argv[1]);
        if (timeout <= 0) {
                printf("Timeout negative or 0 specified\n");
                printf("usage: %s <timeout usec>\n", basename(argv[0]));
                return -1;
        }

        /* Run timer_test with normal (jiffies) resolution timer. */
        printf("\nRunning with CLOCK_REALTIME (normal resolution)..\n");
        if (timer_test(CLOCK_REALTIME, timeout * 1000)) {
                return -1;
        }


        /* Now run timer_test with high resolution timer. */
        printf("\nRunning with CLOCK_SGI_CYCLE (high resolution)..\n");
        return timer_test(CLOCK_SGI_CYCLE, timeout*1000);
}
```

# Sample User-Level Interrupt Program

---

**Note:** This appendix does not apply to Altix XE servers.

---

Example C-1 shows how user-level interrupts (ULIs) are used.

**Example C-1** ULI Program

```
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/uli.h>

/*
 * This is a simple demonstration application which uses the User
 * Level Interrupt (ULI) feature.
 */

#define ULI_CPU 0       /* CPU to use */
#define THREAD_SEMA 0   /* Sema thread waits on */
#define NUM_SEMAS 1             /* Number of semaphores to use */

static int uli_data = 0;
static void * intr;
pthread_attr_t attrs;
pthread_t pt;

/*
 * This is the handler function for the ULI.  It modifies a global
 * variable then wakes up a waiter.  It demonstrates a simple
 * handler plus the use of ULI_wakeup.
 */
static void
uli_handler(void * arg_data)
{
```

```
        int * data = (int*) arg_data;

        /* Increment some global counter */
        (*data)++;

        /* Wake up the waiting thread */
        ULI_wakeup(intr, THREAD_SEMA);

        /* We're done, complete */
        return;
}

/*
 * This simple thread waits for the ULI to fire and then exits.  It
 * demonstrates the use of the ULI_sleep.
 */
static void *
thread_func(void * arg)
{
        printf("Thread blocking on ULI\n");

        if (ULI_sleep(arg, THREAD_SEMA) < 0) {
                perror("ULI_sleep");
                exit(1);
        }

        printf("Thread woken by ULI and exiting\n");

        return NULL;
}

int
main(int argc, char * argv[])
{
        int line;
        int before;

        /* Check for the needed input argument */
        if (argc <= 1) {
                fprintf(stderr, "Need to specify an IRQ line to use\n");
                return 1;
```

```
        }

        /* Get the IRQ line to use */
        line = atoi(argv[1]);

        /* Lock down memory */
        if (mlockall(MCL_CURRENT | MCL_FUTURE) < 0) {
                perror("mlockall");
                return 1;
        }

        /*
         * Create the ULI.  We create NUM_SEMAS semaphores and have the
         * library create a default size stack.
         */
        if ((intr = ULI_register_irq(ULI_CPU, uli_handler, &uli_data, NUM_SEMAS,
                                        line)) == NULL) {
                perror("ULI_register_irq");
                return -1;
        }

        /* Create a child thread */
        pthread_attr_init(&attrs);
        if (pthread_create(&pt, &attrs, thread_func, intr)) {
                perror("pthread_create");
                return 1;
        }

        /* Wait for it to get woken by the ULI and exit */
        pthread_join(pt, NULL);

        /* Let the user know we're checking blocking */
        printf("Child completed\n");
        printf("Checking interrupt blocking\n");

        /* Block the ULI handler from running */
        ULI_block_intr(intr);

        /* Check to make sure that the ULI actually is blocked */
        before = uli_data;
        sleep(5);
```

```
        printf("ULIs handled before blocking: %d\n",
                before);
        printf("ULIs handled while blocking: %d\n",
                uli_data - before);

        /* Unblock the ULI handler */
        ULI_unblock_intr(intr);

        /* Destroy the ULI */
        if (ULI_destroy(intr) < 0) {
                perror("ULI_destroy");
                return -1;
        }

        /* Let's see how many times the ULI got handled */
        printf("ULI handler ran %d times\n",
                uli_data);

        return 0;
}
```

Example C-2 shows a sample ULI program with external interrupts.

**Example C-2** ULI Program with External Interrupts

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sn/uli.h>

/*
 * This is a simple demonstration application which uses the User
 * Level Interrupt (ULI) feature with the External Interrupt (EI) feature.
 */

#define EI_FILE "/dev/extint0"  /* Device control file */
#define LOOPS   10      /* Numbers of EI toggles to receive */
#define NUM_SEMAS 1             /* Number of semaphores to use */
```

```
static int uli_data = 0;
static void * intr;

/*
 * This is the handler function for the ULI.  It modifies a global
 * variable then wakes up a waiter.  It demonstrates a simple
 * handler plus the use of ULI_wakeup.
 */
static void
uli_handler(void * arg_data)
{
        int * data = (int*) arg_data;

        /* Increment some global counter */
        (*data)++;

        /* Wake up the waiting thread */
        ULI_wakeup(intr, 0);

        /* We're done, complete */
        return;
}



int
main(int argc, char * argv[])
{
        int fd, i;

        /* Open the EI control file */
        if ((fd = open(EI_FILE, O_RDONLY)) < 0) {
                perror("Open control file");
                return 1;
        }

        /* Lock down memory */
        if (mlockall(MCL_CURRENT | MCL_FUTURE) < 0) {
                perror("mlockall");
                return 1;
        }
```

```
       /*
        * Create the ULI.  We create NUM_SEMAS semaphores and have the
        * library create a default size stack.
        */
       if ((intr = ULI_register_ei(uli_handler, &uls_data, NUM_SEMAS,
                                       fd)) == NULL) {
               perror("ULI_register_irq");
               return -1;
       }

       /* Receive a few EI toggles */
       for (i = 0; i < LOOPS; i++) {
               if (ULI_sleep(intr, 0) < 0) {
                       perror("ULI_sleep");
                       return 1;
               }
               printf("EI toggled\n");
       }

       /* Destroy the ULI */
       if (ULI_destroy(intr) < 0) {
               perror("ULI_destroy");
               return -1;
       }

       /* Let's see how many times the ULI got handled */
       printf("ULI handler ran %d times\n",
               uli_data);

       return 0;
}
```

# Reading MAC Addresses Sample Program

Example D-1 reads the MAC address from an ethernet card on an SGI system for Linux (line breaks added for readability). It demonstrates how to memory map and interact with hardware devices from user space.

**Note:** The following example is for an Altix ia64 architecture server. It will not correctly read the MAC address on Altix XE servers because the offset to the MAC address is different (that is, it is not 0x414 on Altix XE servers).

**Example D-1** Reading MAC Addresses on an Altix Server

```
/* Sample code to map in PCI memory for a specified device and display
 the contents of a (hard coded) register. */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/pci.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <errno.h>


int main( int argc, char **argv )
{
  char path[128];
  char buf[1024];
  int fd;
  FILE *fptr;
  unsigned int bus, device, function, devfn;
  unsigned int sbus=0, sdevfn=0, svend;
  unsigned long bar, sbar =0;
  char *ptr;
  unsigned int *data;
```

```
if( argc != 4 )
{
  printf( "Must supply bus slot and function\n" );
  exit( 1 );
}

bus = atoi( argv[1] );
device = atoi( argv[2] );
function = atoi( argv[3] );
devfn = PCI_DEVFN( device, function );

errno = 0;
if( (fptr = fopen( "/proc/bus/pci/devices", "r" )) == NULL )
{
  perror( "Unable to open /proc/bus/pci/devices" );
  exit( 1 );
}

while( fgets( buf, sizeof(buf) - 1, fptr ) )
{
 sscanf( buf, "%2x%2x %8x %*x %lx %*lx %*lx %*lx %*lx %*lx %*lx %*lx
        %*lx %*lx %*lx %*lx %*lx %*lx",
        &sbus, &sdevfn, &svend, &sbar);
  if( ( sbus == bus ) && (sdevfn == devfn) )
  {
    bar = sbar & ~0xf;
    printf( "Found matching bar %lx\n", bar );
  }
  sbar = 0;
}
fclose( fptr );
if( !bar ) exit(1); /* did not find */

sprintf( path, "/proc/bus/pci/%02d/%02d.%d", bus, device, function );
printf( "path is :%s\n", path );

fd = open( path, O_RDWR );
if( fd == -1 ) perror( "Couldn't open device file" );

ioctl( fd, PCIIOC_MMAP_IS_MEM );
ptr = mmap( NULL, getpagesize(), PROT_READ | PROT_WRITE, MAP_SHARED, fd, (off_t) bar);
```

```
  if( ptr == MAP_FAILED )
  {
    perror( "mmap failed!" );
    exit( 1 );
  }

  data = (unsigned int*) (ptr + 0x414);
  printf( "ptr is %p, data is %p\n", ptr, data );
  printf( "MAC is %08x\n", *data );
}
```

# IRIX Differences

This appendix summarizes some differences between REACT for IRIX and REACT for Linux:

- `sysfs` attribute files are not compatible with the IRIX `ei`(7) driver. The IRIX driver uses `ioctl`(2) to interact with user space, which is generally not preferred in Linux and does not perform well due to kernel locking issues. An advantage of the attribute file interface is that it is easy to use from the command line, rather than requiring specially written and compiled applications.

- IRIX has the ability to attach capabilities to an executable file, so that a process inherits those capabilities when the file is run via `exec()`. The Linux implementation of capabilities is limited to processes.

- The Linux `capget`(2) and `capset`(2) system calls are used for getting and setting capabilities and are essentially equivalent to the IRIX `cap_get_proc`(3C) and `cap_set_proc`(3C) system calls. For details, see the man pages for the system calls and the Linux `capabilities`(7) and IRIX `capabilities`(4) man pages.

# Glossary

**activity**

When using the frame scheduler, the basic design unit: a piece of work that can be done by one thread or process without interruption. You partition the real-time program into activities and use the frame scheduler to invoke them in sequence within each frame interval.

**address space**

The set of memory addresses that a process may legally access. The potential address space in Linux is $2^{64}$; however, only addresses that have been mapped by the kernel are legally accessible.

**arena**

A segment of memory used as a pool for allocation of objects of a particular type.

**asynchronous I/O**

I/O performed in a separate process so that the process requesting the I/O is not blocked waiting for the I/O to complete.

**average data rate**

The rate at which data arrives at a data collection system, averaged over a given period of time (seconds or minutes, depending on the application). The system must be able to write data at the average rate, and it must have enough memory to buffer bursts at the *peak data rate*.

**control law processor**

A type of stimulator provides the effects of laws of physics to a machine.

**controller thread**

A top-level process that handles startup and termination.

**device driver**

Code that operates a specific hardware device and handles interrupts from that device.

**device service time**

The time the device driver spends processing the interrupt and dispatching a user thread.

**device special file**

The symbolic name of a device that appears as a filename in the /dev directory hierarchy. The file entry contains the *device numbers* that associate the name with a *device driver*.

**external interrupt**

A hardware signal from an I/O device, such as the SGI IOC4 chip, that is generated in response to a voltage change on an externally accessible hardware port.

**fastcall**

A version of a function call that has been optimized in assembler in order to bypass the context switch typically necessary for a full system call.

**file descriptor**

A number returned by open() and other system functions to represent the state of an open file. The number is used with system calls such as read() to access the opened file or device.

**firm real-time program**

A program that experiences a significant error if it misses a deadline but can recover from the error and can continue to execute. See also *hard real-time program* and *soft real-time program*.

**frame interval**

The amount of time that a program has to prepare the next display frame. A frame rate of 60 Hz equals a frame interval of 16.67 milliseconds.

**frame rate**

The frequency with which a simulator updates its display, in cycles per second (Hz). Typical frame rates range from 15 to 60 Hz.

**frame scheduler**

A process execution manager that schedules activities on one or more CPUs in a predefined, cyclic order.

**frame scheduler controller**

The thread or process that creates a frame scheduler. Its thread or process ID is used to identify the frame scheduler internally, so a thread or process can only be identified with one scheduler.

**frame scheduler controller thread**

The thread that creates a frame scheduler.

**guaranteed rate**

A rate of data transfer, in bytes per second, that definitely is available through a particular file descriptor.

**hard real-time program**

A program that experiences a catastrophic error if it misses a deadline. See also *firm real-time program* and *soft real-time program*.

**hardware latency**

The time required to make a CPU respond to an interrupt signal.

**hardware-in-the-loop (HWIL) simulator**

A simulator in which the role of operator is played by another computer.

**interrupt**

A hardware signal from an I/O device that causes the computer to divert execution to a device driver.

**interrupt information template**

An array of `frs_intr_info_t` data structures, where each element in the array represents a minor frame.

**interrupt propagation delay**

See *hardware latency*.

**interrupt redirection**

The process of directing certain interrupts to specific real-time processors and directing other interrupts away from specific real-time processors in order to minimize the latency of those interrupts.

**interrupt response time**

The total time from the arrival of an interrupt until the user process is executing again. Its main components are *hardware latency*, *software latency*, *device service time*, and *mode switch*.

**interrupt service routine (ISR)**

A routine that is called each time an interrupt occurs to handle the event.

**interval time counter (ITC)**

A 64–bit counter that is scaled from the CPU frequency and is intended to allow an accounting for CPU cycles.

**interval timer match (ITM) register**

A register that allows the generation of an interval timer when a certain ITC value has been reached.

**isolate**

To remove the Linux CPU from load balancing considerations, a time-consuming scheduler operation.

**IPI**

Interprocessor interrupt

**jitter**

Numerous short interruptions in process execution.

**locks**

Memory objects that represent the exclusive right to use a shared resource. A process that wants to use the resource requests the lock that (by agreement) stands for that resource. The process releases the lock when it is finished using the resource. See *semaphore*.

**major frame**

The basic frame rate of a program running under the frame scheduler.

**master scheduler**

The first frame scheduler, which provides the time base for the others. See also *slaves* and *sync group*.

**minor frame**

The scheduling unit of the frame scheduler, the period of time in which any scheduled thread or process must do its work.

**mode switch**

The time it takes for a thread to switch from kernel mode to user mode.

**new pthreads library (NPTL)**

The Linux pthreads library shipped with 2.6 Linux.

**overrun**

When incoming data arrives faster than a data collection system can accept it and therefore data is lost.

**overrun exception**

When a thread or process scheduled by the frame scheduler should have yielded before the end of the minor frame but did not.

**page fault**

The hardware event that results when a process attempts to access a page of virtual memory that is not present in physical memory.

**pages**

The units of real memory managed by the kernel. Memory is always allocated in page units on page-boundary addresses. Virtual memory is read and written from the swap device in page units.

**peak data rate**

The instantaneous maximum rate of input to a data collection system. The system must be able to accept data at this rate to avoid overrun. See also *average data rate*.

**process**

The entity that executes instructions in a Linux system. A process has access to an *address space* containing its instructions and data.

**pthread**

A thread defined by the POSIX standard. Pthreads within a process use the same global address space. Also see *thread*.

**rate-monotonic analysis**

A technique for analyzing a program based on the periodicities and deadlines of its threads and events.

**rate-monotonic scheduling**

A technique for choosing scheduling priorities for programs and threads based on the results of *rate-monotonic analysis*.

**restrict**

To prevent a CPU from running scheduled processes

**scheduling discipline**

The rules under which an activity thread or process is dispatched by a frame scheduler, including whether or not the thread or process is allowed to cause *overrun* or *underrun exceptions*.

**segment**

Any contiguous range of memory addresses. Segments as allocated by Linux always start on a page boundary and contain an integral number of pages.

**semaphore**

A memory object that represents the availability of a shared resource. A process that needs the resource executes a p operation on the semaphore to reserve the resource, blocking if necessary until the resource is free. The resource is released by a v operation on the semaphore. See also *locks*.

**shield**

To switch off the timer (scheduler) interrupts that would normally be scheduled on a CPU.

**simulator**

An application that maintains an internal model of the world. It receives control inputs, updates the model to reflect them, and outputs the changed model as visual output.

**slaves**

The other schedulers that take their time base interrupts from the *master scheduler*. See also *sync group*.

**soft real-time program**

A program that can occasionally miss a deadline with only minor adverse effects. See also *firm real-time program* and *hard real-time program*.

**software latency**

The time required to dispatch an interrupt thread.

**spraying interrupts**

The distribution of I/O interrupts across all available processors as a means of balancing the load.

**stimulator**

An application that maintains an internal model of the world. It receives control inputs, updates the model to reflect them, and outputs the changed model as nonvisual output.

**sync group**

The combination of a *master scheduler* and *slaves*.

**thread**

An independent flow of execution that consists of a set of registers (including a program counter and a stack). Also see *pthread*.

**TLB**

Translation lookaside buffer, which translates CPU virtual memory addresses to bus physical memory addresses.

**transport delay**

The time it takes for a simulator to reflect a control input in its output display. Too long a transport delay makes the simulation inaccurate or unpleasant to use.

**ULI process**

A user process that has registered a function with the kernel, linked into the process in the normal fashion, to be called when a particular interrupt is received.

**underrun exception**

When a thread or process scheduled by the frame scheduler should have started in a given minor frame but did not (owing to being blocked), an underrun exception is signaled. See *overrun exception*.

**unsynchronized drifty ITCs**

Systems with processors that run at the same speed but do not have the same clock source and therefore their ITC values may experience drift relative to one another.

**user-level interrupt (ULI)**

A facility that allows a hardware interrupt to be handled by a user process.

# Index

**A**

abstraction layer, 17
activity thread management, 69
address space (locking in memory), 101
aircraft simulator, 3
application example, 133
asynchronous I/O, 91
average data rate, 5

**B**

backup configuration files, 106
barrier facility, 39
base address registers (BARs), 93
/boot, 106
/boot/grub/menu.lst, 106
bootcpuset, 44, 106

**C**

C language, 6
cache warming, 64
callout deregistration, 30
callout mechanism, 28
callout registration, 29
cap_get_proc, 191
CAP_IPC_LOCK, 43
cap_set_proc, 191
CAP_SYS_NICE, 43
CAP_SYS_RESOURCE, 43
CAP_SYS_TIME, 43
capabilities and executable file, 189
capabilities scheduling, 42
capability sets, 42

cat, 129
chkconfig, 106
clock processor, 45
clock_getres(), 13
clock_gettime, 13, 14
CLOCK_MONOTONIC, 13
CLOCK_REALTIME, 13
clock_settime, 13
CLOCK_SGI_CYCLE, 15
clocks, 12
configuration, 105
configuration scripts, 105
console driver, 46
console interrupts, 11
console node, 46, 47
control law process stimulator, 4
controller thread, 78
CPU
    isolating, 48
    restricting, 10, 47
    workload control, 37
CPU 0, 45, 47
CPU-bound, 9
cpuset, 44, 116
cpusets, 56
cycles per second, 2

**D**

data collection system, 4
deregistration of callout, 30
dev attribute file, 18
/dev/cpuset/boot/mems, 115
/dev/cpuset/rtcpus/rtcpuN/mems, 115
/dev/extint#, 100
device service time, 50, 53

system flush duration, 45
system-call time base, 56

**T**

tasklet action event, 123
tasklet hiaction, 123
test_and_clear_bit, 105
test_and_set_bit, 105
thread, 59
thread control, 44
thread creation, destruction, and signals, 133
time base for frame scheduler, 70
time base support, 56
time estimation, 66
time frame in tracevisualizer, 126
time slices, 38
time-share applications, 10
timer interrupts, 11, 38
timer_create, 13
timer_create(), 15
timer_settime(), 15
timer_t, 15
timers, 14
toggle output modes, 33
top, 130
trace events, 121
tracedaemon, 119
tracevisualizer, 119
transport delay, 2
troubleshooting, 129

**U**

ULI
    See "User-level interrupt (ULI) facility", 95
uli, 96
ULI_block_intr, 97
ULI_destroy, 97
ULI_register_irq(), 97, 102

ULI_sleep(), 97
ULI_unblock_intr, 97
ULI_wakeup(), 97
unaligned access event, 123
underrun exception, 66
underrun, in frame scheduler, 72
unsupported hardware device capabilities, 30
usecs (microseconds), 50
user application communication, 133
user thread control, 44
user thread dispatch, 53
user-level interrupt (ULI), 95
    concurrency, 99
    global variables, 99
    handler interaction, 102
    initializing, 100
    interrupt handler registration, 101
    multiple devices, 99
    mutual exclusion, 103
    overview, 95
    per-IRQ handler, 102
    program address space locking, 101
    restrictions on handler, 98
    ULI_block_intr(), 103
    ULI_sleep (), 102
    ULI_sleep () function, 100
    ULI_wakeup () function, 102
user-level interrupts (ULI)
    sample program, 179
/usr/include/asm/bitops.h, 105
/usr/include/asm/param.h, 38
/usr/include/sn/timer.h, 14
/usr/include/sys/pthread.h, 60
/usr/sbin/reactconf.pl, 105
/usr/share/src/react/examples, 61

**V**

virtual memory locking, 10
virtual reality simulator, 3

volatile keyword,  99
Vsync time base,  56

**W**

wave stimulator,  4

**Z**

zooming in tracevisualizer,  126