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

007–4746–003

New Features in this Guide

This revision contains the following:

- Information about using the `sysconf(_SC_CLK_TCK)` function to determine the tick frequency. For more information, see "Clocks and Timers" on page 10.
- Chapter 3, "External Interrupts" on page 15 and Example C-2, page 142
- "`kbar` Kernel Barrier Facility" on page 35
- "SGI High-Resolution POSIX Timers" on page 13, "High-Resolution Timer" on page 65, and Appendix B, "High-Resolution Timer Example" on page 133
- "Avoid Kernel Module Insertion and Removal" on page 43

Record of Revision

Version	Description
001	February 2005 Original publication to support REACT real-time for Linux 4.0
002	July 2005 Revision to support REACT real-time for Linux 4.2
003	December 2005 Revision to support REACT real-time for Linux 4.3

Contents

About This Guide . . . . **xxi**
Audience xxi
What This Guide Contains xxi
Related Publications and Sites xxii
Conventions xxiii
Obtaining Publications xxiv
Reader Comments xxiv

1. Introduction **1**
Real-Time Programs 1
Real-Time Applications 1
Simulators and Stimulators 2
Aircraft Simulators 3
Ground Vehicle Simulators 3
Plant Control Simulators 3
Virtual Reality Simulators 3
Hardware-in-the-Loop Simulators 4
Control Law Processor Stimulator 4
Wave Tank Stimulator 4
Data Collection Systems 4
Process Control Systems 5
REACT Real-Time for Linux 6

2. Linux and REACT Support for Real–Time Programs **7**
Kernel Facilities 7

Special Scheduling Disciplines 7
Virtual Memory Locking 8
Processes Mapping and CPUs 8
Interrupt Distribution Control 9
Frame Scheduler 10
Clocks and Timers 10
Clocks 10
Direct RTC Access 12
ITC Register Access 12
SGI High-Resolution POSIX Timers 13
Interchassis Communication 13
Socket Programming 14
Message-Passing Interface (MPI) 14
3. External Interrupts 15
Abstraction Layer 15
`sysfs` Attribute Files 16
Low-level Driver Interface 19
Driver Registration 19
Implementation Functions 20
When an External Interrupt Occurs 22
Driver Deregistration 23
Interrupt Notification Interface 23
Callout Mechanism 23
Callout Registration 24
Callout Deregistration 25
Making Use of Unsupported Hardware Device Capabilities 25
Low-level Driver Template 26

Example: SGI IOC4 PCI Device 26
Multiple Independent Drivers 26
External Interrupt Output 27
External Interrupt Ingest 29
Physical Interfaces 30

4. CPU Workload 33
Using Priorities and Scheduling Queues 33
Scheduling Concepts 33
Timer Interrupts 34
Real-Time Priority Band 34
`kbar` Kernel Barrier Facility 35
Scheduling Capabilities 38
Setting Pthread Priority 38
Controlling Kernel and User Threads 39
Minimizing Overhead Work 39
Avoid the Clock Processor (CPU 0) 40
Reduce the System Flush Duration 40
Redirect Interrupts 40
Select the Console Node for SAL Console Driver Interrupt 41
Restrict and Isolate CPUs 41
Restricting a CPU from Scheduled Work 42
Isolating a CPU from Scheduler Load Balancing 43
Avoid Kernel Module Insertion and Removal 43
Understanding Interrupt Response Time 44
Maximum Response Time Guarantee 44
Components of Interrupt Response Time 44

Hardware Latency 45
Software Latency 45
Kernel Critical Sections 46
Interrupt Threads Dispatch 46
Device Service Time 47
Interrupt Service Routines 47
User Threads Dispatch 47
Mode Switch 47
Minimizing Interrupt Response Time 47

5. Using the Frame Scheduler 49

Frame Scheduler Concepts 49
Frame Scheduler Basics 50
Thread Programming Model 51
Frame Scheduling 51
Controller Thread 53
Frame Scheduler API 54
Interrupt Information Templates 54
Library Interface for C Programs 56
Thread Execution 58
Scheduling Within a Minor Frame 59
Scheduler Flags `frs_run` and `frs_yield` 59
Detecting Overrun and Underrun 60
Estimating Available Time 60
Synchronizing Multiple Schedulers 61
Starting a Single Scheduler 61
Starting Multiple Schedulers 62
Pausing Frame Schedulers 62

Managing Activity Threads 63
Selecting a Time Base 64
High-Resolution Timer 65
External Interrupts as a Time Base 65
Using the Scheduling Disciplines 66
Real-Time Discipline 66
Background Discipline 67
Underrunable Discipline 67
Overrunnable Discipline 68
Continuable Discipline 68
Using Multiple Consecutive Minor Frames 68
Designing an Application for the Frame Scheduler 70
Preparing the System 71
Implementing a Single Frame Scheduler 71
Implementing Synchronized Schedulers 73
Synchronized Scheduler Concepts 73
Master Controller Thread 74
Slave Controller Thread 74
Handling Frame Scheduler Exceptions 75
Exception Types 76
Exception Handling Policies 76
Injecting a Repeat Frame 76
Extending the Current Frame 77
Dealing With Multiple Exceptions 77
Setting Exception Policies 77
Querying Counts of Exceptions 78
Using Signals Under the Frame Scheduler 80

Handling Signals in the Frame Scheduler Controller 80
Handling Signals in an Activity Thread 81
Setting Frame Scheduler Signals 81
Handling a Sequence Error 82
Using Timers with the Frame Scheduler 83

6. Disk I/O Optimization 85
Memory-Mapped I/O 85
Asynchronous I/O 85
Conventional Synchronous I/O 86
Asynchronous I/O Basics 86

7. PCI Devices 87

8. User-Level Interrupts 89
Overview of ULI 90
Restrictions on the ULI Handler 91
Planning for Concurrency: Declaring Global Variables 93
Using Multiple Devices 93
Setting Up ULI 93
Opening the Device Special File 94
Locking the Program Address Space 94
Registering the Interrupt Handler 95
Registering a Per-IRQ Handler 96
Interacting With the Handler 96
Achieving Mutual Exclusion 97

9. Installation Overview 99

10. REACT System Configuration 101
Generating a REACT System Configuration 102
Example `reactcfg.pl` Output 104
Example `reactboot.pl` Output 106

11. Troubleshooting 107

Appendix A. Example Application 109
Building and Loading a Kernel Module 110
Kernel-space `Makefile` 112
`rt_samplemod.h` 112
`rt_samplemod.c` 113
`ipi.c` 117
Building and Loading a User-Space Application 119
User-space `Makefile` 119
`rt_sample.c` 119
`common.h` 125
`common.c` 125
`mmtimer.c` 128
Running the Sample Application 130

Appendix B. High-Resolution Timer Example 133

Appendix C. Sample ULI Program 139

Appendix D. Reading MAC Addresses Sample Program 145

Glossary 149

Index 157

Figures

Figure 3-1 Output and Input Connectors for the Internal Driver Circuit 31
Figure 4-1 `kbar`(3) Kernel Barrier Facility 37
Figure 4-2 Components of Interrupt Response Time 45
Figure 5-1 Major and Minor Frames 52
Figure 8-1 ULI Functional Overview 89
Figure 8-2 ULI Handler Functions 92

Tables

Table 3-1 Register Format 29
Table 5-1 Frame Scheduler Operations 56
Table 5-2 Signal Numbers Passed in `frs_signal_info_t` 81
Table 8-1 Common Arguments for Registration Functions 90

Examples

Example 5-1 Skeleton of an Activity Thread 58
Example 5-2 Alternate Skeleton of Activity Thread 59
Example 5-3 Function to Set `INJECTFRAME` Exception Policy 78
Example 5-4 Function to Set `STRETCH` Exception Policy 78
Example 5-5 Function to Return a Sum of Exception Counts (pthread Model) 79
Example 5-6 Function to Set Frame Scheduler Signals 82
Example 5-7 Minimal Activity Process as a Timer 83
Example 10-1 Example `reactcfg.pl` Output 105
Example 10-2 Example `reactboot.pl` Output 106
Example A-1 Kernel-space `Makefile` Extract 112
Example A-2 `rt_samplemod.h` Extract 112
Example A-3 `rt_samplemod.c` Extract 113
Example A-4 `ipi.c` Extract 117
Example A-5 User-space `Makefile` Extract 119
Example A-6 `rt_sample.c` Extract 119
Example A-7 `common.h` Extract 125
Example A-8 `common.c` Extract 125
Example A-9 `mmtimer.c` Extract 128
Example B-1 High-Resolution Timer 133
Example C-1 ULI Program 139
Example C-2 ULI Program with External Interrupts 142
Example D-1 Reading MAC Addresses 145

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 7 provides an overview of the real-time support for programs in Linux and REACT
- Chapter 4, "CPU Workload" on page 33 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 49 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 85 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 87 discusses the Linux PCI interface
- Chapter 8, "User-Level Interrupts" on page 89
- Chapter 9, "Installation Overview" on page 99 lists the RPMs required to run REACT
- Chapter 10, "REACT System Configuration" on page 101 discusses the `reactcfg.pl` and `reactboot.pl` configuration scripts that are provided to assist you in setting up REACT
- Chapter 11, "Troubleshooting" on page 107 discusses diagnostic tools that apply to real-time applications
- Appendix A, "Example Application" on page 109 provides excerpts of application modules to be used with REACT
- Appendix B, "High-Resolution Timer Example" on page 133 demonstrates the use of SGI high-resolution timers.
- Appendix C, "Sample ULI Program" on page 139 contains a sample program that shows how user-level interrupts are used.
- Appendix D, "Reading MAC Addresses Sample Program" on page 145 provides a sample program for reading the MAC address from an ethernet card

Related Publications and Sites

The following books may be useful:

- Available from the online SGI Technical Publications Library:
  - *Linux Configuration and Operations Guide*
  - *Linux Device Driver Programmer's Guide-Porting to SGI Altix Systems*
  - *SGI Altix 3000 User's Guide*
  - *SGI Altix 350 System User's Guide*
  - *SGI Altix 330 System User's Guide*
  - *SGI L1 and L2 Controller Software User's Guide*
  - *SGI ProPack 4 for Linux 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:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, California 94043–1351

SGI values your comments and will respond to them promptly.

Chapter 1

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

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:

- A SUSE Linux Enterprise Server 9 (SLES9) kernel built by SGI for the appropriate service pack (see the REACT for Linux release notes), which includes recent community-accepted enhancements that enable low-latency interrupt response
- System configuration scripts `reactcfg.pl` and `reactboot.pl` 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.

Chapter 2

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 7
- "Frame Scheduler" on page 10
- "Clocks and Timers" on page 10
- "Interchassis Communication" on page 13

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 7
- "Virtual Memory Locking" on page 8
- "Processes Mapping and CPUs" on page 8
- "Interrupt Distribution Control" on page 9

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 the earnings-based scheduler is effective at scheduling time-share 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. Tasks scheduled using the POSIX real-time policies are not subject to “earnings” controls. For more information about scheduling, see "Real-Time Priority Band" on page 34 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 40.

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:

```
# 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 49.

Clocks and Timers

This section discusses the following:

- "Clocks" on page 10
- "Direct RTC Access" on page 12
- "ITC Register Access" on page 12
- "SGI High-Resolution POSIX Timers" on page 13

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 13.

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 "`rt_sample.c`" on page 119 for an example or read the comments in `/usr/include/sn/timer.h`, which is installed by the `mmtimer-devel` RPM.

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.

Some versions of `glibc` allow access to the time since the start of the process or thread based on the ITC via the `CLOCK_PROCESS_CPUTIME_ID` and `CLOCK_THREAD_CPUTIME_ID` clocks. This avoids a system call and provides a fast way to access time information. However, the POSIX standard states that these two clocks must return the processor time used by a process or thread, not the real time that has passed since the start of the process. Moreover, the values returned from these two clocks are not reliable on many systems because a process may be switched to a different processor while executing. ITC values on the other processor may not relate to the ITC values obtained on the prior processor. As a result, an application may exhibit strange phenomena if it uses these two clocks; for example, time may seem to skip forward or backward. This is a known problem in all versions of `glibc` included with SLES9.

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 10.

Note: With SGI ProPack, only the REACT kernel provides SLES9 POSIX timer support for `CLOCK_SGI_CYCLE`. It is also supported in recent 2.6 Linux community kernels.

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 133.

Interchassis Communication

This section discusses the following:

- "Socket Programming" on page 14

- "Message-Passing Interface (MPI)" on page 14

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.

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.

Chapter 3

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. Examples are the IO9 and IO10 cards on SGI Altix systems, which have a 1/8-in stereo-style jack into which a 0-5V signal can be fed. An exterior piece of hardware can move 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. For more information about these topics, see the following files, which are installed with the Linux source code corresponding to the real-time kernel:

```
Documentation/extint.txt
Documentation/sgi-ioc4.txt
```

This section discusses the following:

- "Abstraction Layer" on page 15
- "Making Use of Unsupported Hardware Device Capabilities " on page 25
- "Low-level Driver Template" on page 26
- "Callout Mechanism" on page 23
- "Example: SGI IOC4 PCI Device" on page 26

Abstraction Layer

Different chips 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 16
- "Low-level Driver Interface" on page 19

- "Interrupt Notification Interface" on page 23

`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:

- It can be memory-mapped read-only as a single page, in which case the first unsigned long word of the page contains a counter that is incremented each time an interrupt occurs.
- You can use `poll`(2) and `select`(2) for reading on a read-only file descriptor opened on the device, in which case `poll` will indicate whether an interrupt has occurred since the last `read`(2) or `open`(2) of the file, and `select` will return when the next interrupt is received.
- The file can also be the subject of `read`, in which case it returns a string representation of the counter's value.

`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
toggle
```

For more information, see "External Interrupt Output" on page 27.

`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.

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.

For more information, see "External Interrupt Output" on page 27.

`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 27.

`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#` 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 27.
`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.

Note: This is 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.

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 `<linux/extint.h>` and the function prototypes contained therein.

This section discusses the following:

- "Driver Registration" on page 19
- "Implementation Functions" on page 20
- "When an External Interrupt Occurs" on page 22
- "Driver Deregistration" on page 23

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 16.

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 `<linux/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 23
- "Callout Registration" on page 24
- "Callout Deregistration" on page 25

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 `<linux/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 IO 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 following example low-level driver as a template:

`linux/drivers/char/ioc4_extint.c`

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 26
- "External Interrupt Output" on page 27
- "External Interrupt Ingest" on page 29
- "Physical Interfaces" on page 30

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 IO 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.

For the most part, this driver simply 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: • `000` loads a `0` to `INT_OUT` • `100` loads a `1` to `INT_OUT` • `101` pulses `INT_OUT` high for 3 ticks • `110` pulses `INT_OUT` for 3 ticks every `COUNT` • `111` toggles `INT_OUT` for 3 ticks every `COUNT` • `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 IO 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. Available sources can be found in the abstraction layer device's `sourcelist` attribute.

Physical Interfaces

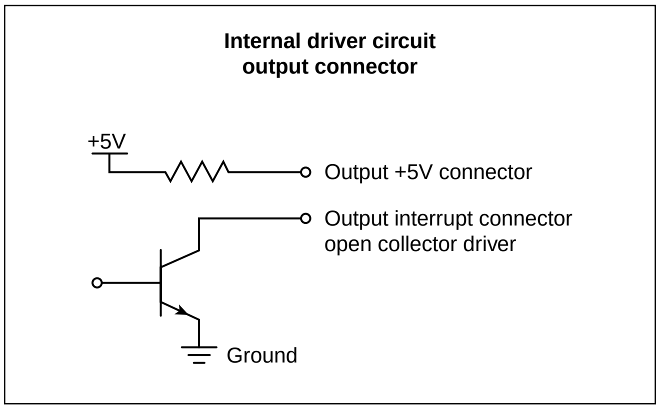
Use a two-conductor shielded cable to connect external interrupt output and input, with the two cable conductors wired to the +5V and interrupt conductors and the sleeves connected to the cable shield at both ends to maintain EMI integrity.

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.



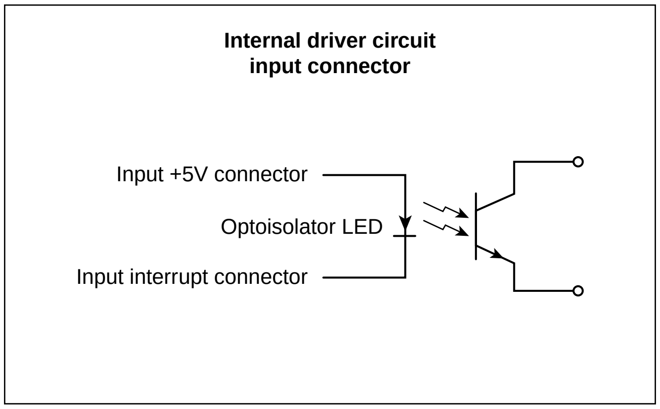


Figure 3-1 Output and Input Connectors for the Internal Driver Circuit

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.

Chapter 4

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 33
- "Minimizing Overhead Work" on page 39
- "Understanding Interrupt Response Time" on page 44
- "Minimizing Interrupt Response Time" on page 47

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 33
- "`kbar` Kernel Barrier Facility" on page 35
- "Scheduling Capabilities" on page 38
- "Setting Pthread Priority" on page 38
- "Controlling Kernel and User Threads" on page 39

Scheduling Concepts

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

- "Timer Interrupts" on page 34
- "Real-Time Priority Band" on page 34

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 1 millisecond (ms) in every CPU. You can determine the frequency of this interval with the `sysconf(_SC_CLK_TCK)` function (see "Clocks" on page 10). 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 simply 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

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`:

```
# 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`.

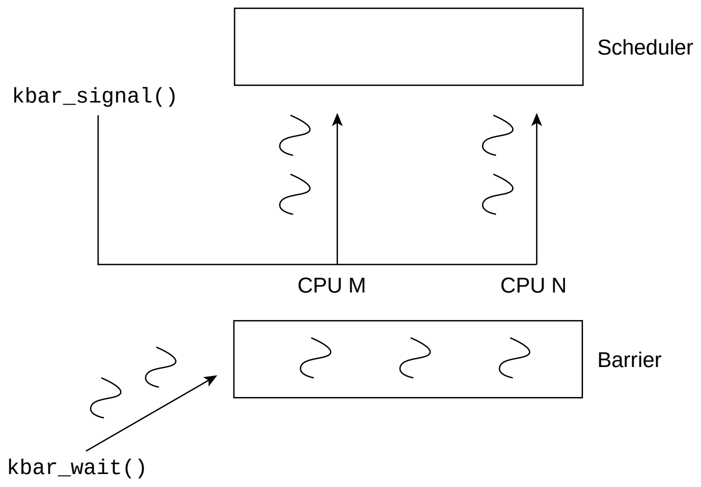


Figure 4-1 `kbar`(3) Kernel Barrier Facility

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

Scheduling Capabilities

Linux supports capability sets. These per-process attributes allow a process to perform actions normally restricted to the `root` user without requiring the process owner to run as `root`. Of particular interest to real-time applications is the ability to change the scheduling priority of a process and to lock memory without needing to run as `root`.

You can manipulate capabilities within a program with the `cap_set_proc`(3) and `cap_get_proc`(3) functions.

Note: Linux does not support filesystem capability attributes; that is, you cannot modify a binary to run each time with a certain set of capabilities.

For more information, see the `capabilities`(7) man page.

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 can use the `reactcfg.pl` configuration script to simplify this procedure; see Chapter 10, "REACT System Configuration" on page 101.

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 40
- "Reduce the System Flush Duration" on page 40
- "Redirect Interrupts" on page 40
- "Select the Console Node for SAL Console Driver Interrupt" on page 41
- "Restrict and Isolate CPUs" on page 41
- "Avoid Kernel Module Insertion and Removal" on page 43

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 4 in Chapter 10, "REACT System Configuration" on page 101.

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 `/proc/irq/`*interrupt_number*`/smp_affinity` file shows a bitmask of the CPUs that are allowed to receive a given interrupt. By writing a bitmask to this file, you can indicate which CPU is allowed to receive that 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:

```
# echo 2 > /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 `reactboot.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 10, "REACT System Configuration" on page 101.

Select the Console Node for SAL Console Driver Interrupt

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 and Isolate 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
- Isolated

Note: The term *isolate*, when referring to a CPU in Linux, means to remove the CPU from load balancing considerations, a time-consuming scheduler operation.

You can use the `reactcfg.pl` configuration script to perform the step required to restrict or isolate a CPU. For more information, see Chapter 10, "REACT System Configuration" on page 101.

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, do the following (or use the `reactcfg.pl` configuration script documented in Chapter 10, "REACT System Configuration" on page 101):

1. Configure a bootcpuset as described in "Controlling Kernel and User Threads" on page 39 and reboot the system.
2. Do one of the following:
   - Make a cpuset that encompasses just the CPU that you want to restrict. See the `cpuset`(1) man page for more information.
   - Run the SGI-provided `sysmp(MP_RESTRICT,cpu)` call from program control found in `libsgirt`. For more information, see the `libsgirt`(3) man page.

To remove the restriction, allowing the CPUs to execute any scheduled process, you must reboot the system without the configured bootcpuset.

After restricting a CPU, you can assign processes to it using the SGI `cpuset` command.

For example, to run a program on the `cpuset` named `rtcpu3` (which was set up to include only CPU 3), do the following:

```
# cpuset --invoke=/rtcpu3 -I ~rt/bin/rtapp
```

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) and `libsgirt`(3) man pages.

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, you must specify them at system boot time. Do the following (or use the `reactcfg.pl` configuration script documented in Chapter 10, "REACT System Configuration" on page 101):

1. Include the following string in the `append` argument for the kernel you are booting, or in the `/etc/elilo.conf` file:

   ```
   isolcpus=cpu,..
   ```

   For example, suppose you have the following existing `append` argument:

```
append = "selinux=0 console=ttyS0,115200n8 init=/sbin/bootcpuset"
```

   To isolate CPUs 2 and 3, change the `append` argument to the following:

```
append = "selinux=0 console=ttyS0,115200n8 init=/sbin/bootcpuset isolcpus=2,3"
```

2. If you edited the `/etc/elilo.conf` file in step 1, run the `elilo` command to place a copy of the updated `elilo.conf` file in the appropriate directory:

   ```
   # elilo
   ```

3. Reboot the system. After the reboot completes, CPUs 2 and 3 will be isolated.

Normally, you would also want to restrict the isolated CPUs. See "Restricting a CPU from Scheduled Work" on page 42.

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.

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. Linux guarantees a maximum interrupt response time on certain systems, but you must configure the system properly 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.

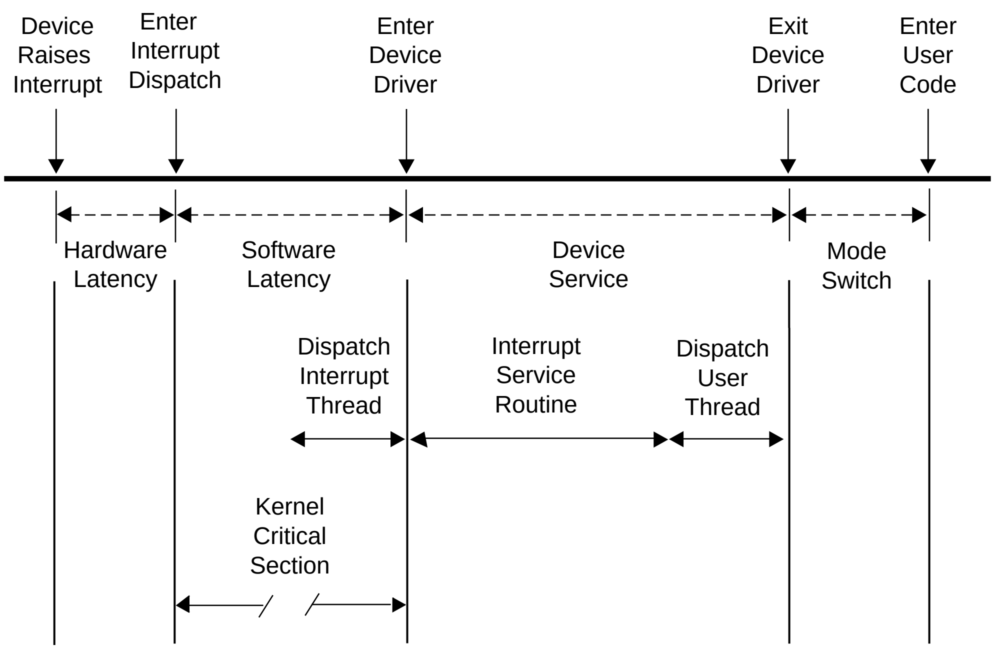


Figure 4-2 Components of Interrupt Response Time

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 87.

Software Latency

Software latency is affected by the following:

- "Kernel Critical Sections" on page 46
- "Interrupt Threads Dispatch" on page 46

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 42 and "Isolating a CPU from Scheduler Load Balancing" on page 43.

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 40.

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

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 49
- "Selecting a Time Base" on page 64
- "Using the Scheduling Disciplines" on page 66
- "Using Multiple Consecutive Minor Frames" on page 68
- "Designing an Application for the Frame Scheduler" on page 70
- "Preparing the System" on page 71
- "Implementing a Single Frame Scheduler" on page 71
- "Implementing Synchronized Schedulers" on page 73
- "Handling Frame Scheduler Exceptions" on page 75
- "Using Signals Under the Frame Scheduler" on page 80
- "Using Timers with the Frame Scheduler" on page 83

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 50
- "Thread Programming Model" on page 51
- "Frame Scheduling" on page 51

- "Controller Thread" on page 53
- "Frame Scheduler API" on page 54
- "Interrupt Information Templates" on page 54
- "Library Interface for C Programs" on page 56
- "Thread Execution" on page 58
- "Scheduling Within a Minor Frame" on page 59
- "Synchronizing Multiple Schedulers" on page 61
- "Starting a Single Scheduler" on page 61
- "Starting Multiple Schedulers" on page 62
- "Pausing Frame Schedulers" on page 62
- "Managing Activity Threads" on page 63

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 and Isolate CPUs" on page 41). You do not have to set up cpusets for the frame scheduled CPUs because the frame scheduler will set up cpusets named `rtcpu` *cpu#* 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`*cpu#*, 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 43 and "Preparing the System" on page 71.

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 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 64.) 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 65)
- 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.

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.

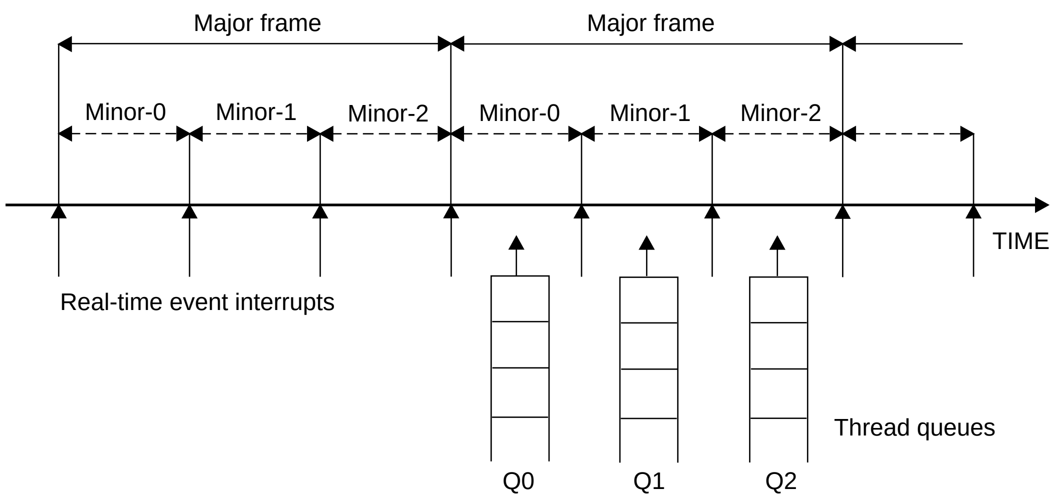


Figure 5-1 Major and Minor Frames

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 66.

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, as follows:

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

See "Implementing a Single Frame Scheduler" on page 71 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 80).
- 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 56).

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 62 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 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 63 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 66.

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 80.

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 67

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 81). If a signal is sent to it, it begins executing in its specified or default signal handler; otherwise, it simply 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 65
- "External Interrupts as a Time Base" on page 65

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 high-resolution timers in all CPUs are synchronized automatically.

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 15.

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 66
- "Background Discipline" on page 67
- "Underrunable Discipline" on page 67
- "Overrunnable Discipline" on page 68
- "Continuable Discipline" on page 68

Real-Time Discipline

In the real-time scheduling 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 67) 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 in some way and must take some 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 68)

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 68)

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 59).

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 for each frame should be as follows:

First frame	Real-time + overrunnable + continuable The thread must start in this frame (not underrunable) but is not required to yield (overrunnable). If it yields, it is not restarted in the following minor frame (continuable).
Intermediate	Realtime + underrunable + 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 frame	Realtime + underrunable 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 40.
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 65.
4. Make sure that none of the real-time CPUs is managing the clock. Normally, the responsibility of handling 10ms scheduler interrupts is given to CPU 0. For more information, see "Avoid the Clock Processor (CPU 0)" on page 40.
5. Restrict and isolate the real-time CPUs, as described in "Restrict and Isolate CPUs" on page 41.
6. Load the `frs` kernel module:

   ```
   # modprobe frs
   ```

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

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, since 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 81 and "Setting Exception Policies" on page 77). 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 73
- "Master Controller Thread" on page 74
- "Slave Controller Thread" on page 74

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 80.
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.

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 81 and "Setting Exception Policies" on page 77.
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 76
- "Exception Handling Policies" on page 76
- "Setting Exception Policies" on page 77

- "Querying Counts of Exceptions" on page 78

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 81.

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 64.

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.

This section discusses the following:

- "Handling Signals in the Frame Scheduler Controller" on page 80
- "Handling Signals in an Activity Thread" on page 81
- "Setting Frame Scheduler Signals" on page 81
- "Handling a Sequence Error" on page 82

Handling Signals in the Frame Scheduler Controller

When a frame scheduler detects an overrun or underrun exception that it cannot recover from, 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 63.

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`

Field Name	Signal Purpose	Default Signal Number
`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();
```

Chapter 6

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 85
- "Asynchronous I/O" on page 85

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 simply 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 86
- "Asynchronous I/O Basics" on page 86

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 145.

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

User-Level Interrupts

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. An error in programming 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.

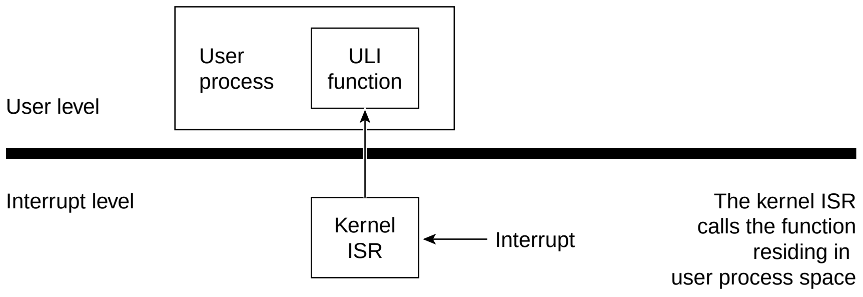


Figure 8-1 ULI Functional Overview

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 90
- "Setting Up ULI" on page 93

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).

Function	Description
`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 91
- "Planning for Concurrency: Declaring Global Variables" on page 93
- "Using Multiple Devices" on page 93

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.

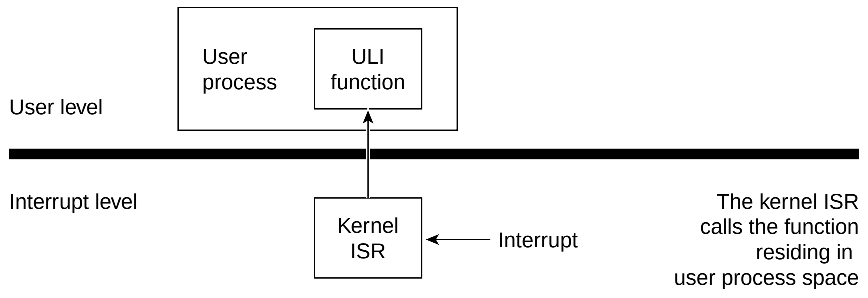


Figure 8-2 ULI Handler Functions

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 96.

Setting Up ULI

A program initializes for ULI in the following major steps:

1. Load the `uli` kernel module:

   ```
   # 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 94
- "Locking the Program Address Space" on page 94
- "Registering the Interrupt Handler" on page 95
- "Registering a Per-IRQ Handler" on page 96
- "Interacting With the Handler" on page 96
- "Achieving Mutual Exclusion" on page 97

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 96. 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.

Installation Overview

To install REACT, do the following:

1. Install SUSE Linux Enterprise Server 9, (SLES9) with the appropriate service pack as outlined in the REACT for Linux release notes and *SGI ProPack 4 for Linux Start Here*.

   Note: Do not install a kernel other than the default Linux kernel.

2. Install the appropriate SGI ProPack 4 for Linux RPMs for your site, including at least the following:

   `cpuset-*.rpm`

   The following RPM is optional but recommended:

   `mmtimer-devel`

   Note: Do not install a kernel and kernel module RPMs other than the defaults. In step 3 below, the REACT kernel and modules will be installed over the defaults.

   For more information, see *SGI ProPack 4 for Linux Start Here*.

3. Install the following RPMs from the REACT CD:

   - REACT kernel (required):

     `kernel-rtgfx-*.rpm`

   - REACT sample system configuration scripts (required to run `reactcfg.pl` and `reactboot.pl`):

     `react-configuration-*.rpm`

   - REACT kernel source (required to build `rt_sample_mod` discussed in Appendix A, "Example Application" on page 109):

     `kernel-rtgfx-source-*.rpm`

- IRIX to Linux REACT compatibility library:

  `libsgirt-*.rpm`

- REACT documentation:

  `react-docs-*.rpm`

- SLES9 modules built for the appropriate service pack (see the REACT for Linux release notes) and the REACT kernel:

  `km-rtgfx-*.rpm`

For more information, see the following file on the REACT CD:

`/usr/share/doc/sgi-react-4.2/README.relnotes`

REACT System Configuration

This chapter explains how to configure restricted and isolated CPUs on a system running the REACT real-time for Linux product. For information about creating an external interrupt character special device file, see "Opening the Device Special File" on page 94. The procedure uses the following configuration scripts to assist in generating a simple REACT configuration:

- `reactcfg.pl` is a Perl script that edits the `/etc/bootcpuset.conf` and `/etc/elilo.conf` scripts. "Example `reactcfg.pl` Output" on page 104 shows the script workflow.

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

- `reactboot.pl` is a Perl script that sets the cpusets and redirects interrupts. See "Example `reactboot.pl` Output" on page 106.

These scripts as released make assumptions about how your system is configured. You should use them in their default state as an aid to learning the configuration procedure. No configuration changes are made to your system until you confirm the action to overwrite the `/etc/elilo.conf` file, at which point both `/etc/bootcpuset.conf` and `/etc/elilo.conf` will be overwritten.

After you understand the overall procedure and the REACT system configuration, you can modify the scripts to meet your specific needs and generate your final production configuration. If you only have a single system to configure, you may prefer to edit the `/etc/elilo.conf` and `/etc/bootcpuset.conf` files directly rather than modify the `reactcfg.pl` script. However, if you have multiple systems to modify, it may be more efficient to modify the script and then reuse it on the other systems. The `reactboot.pl` script is meant to be run after every reboot; therefore, you should modify the script as necessary or write a new script.

Note: This procedure assumes that, at a minimum, you have installed the following RPMs from the SGI ProPack for Linux and REACT CDs:

```
cpuset-*.rpm
kernel-rt-*.rpm
react-configuration-*.rpm
```

You must have at least these RPMs installed before beginning this procedure.

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 `timer` and `IPI`). You cannot use CPU 0 for real time. Although the `reactboot.pl` script attempts to redirect all interrupts to CPUs that are not used for real time, certain interrupts cannot be moved, including the console interrupt.

   Note: Interrupt affinity changes are temporary and are not retained across system reboots. SGI recommends that you run a set-up script equivalent to `reactboot.pl` after every reboot to set up your real-time cpusets and set the affinity of device interrupts.

3. Run the `reactcfg.pl` script, supplying a comma-separated list of individual real-time CPUs, a range of real-time CPUs, or a mixture of both:

   `reactcfg.pl` *real-time-CPU-list*

   The script will set up the appropriate `/etc/bootcpuset.conf` and `/etc/elilo.conf` files based on the CPUs listed and the number of CPUs on your system. As an example, on an 8-processor system, the following command line will set up CPUs 0, 1, 4, 5, and 6 in the bootcpuset, with CPUs 2, 3 and 7 restricted and isolated for real-time use:

```
# reactcfg.pl 2-3,7
```

It will also create a new section labeled `rt` in your `/etc/elilo.conf` file and make this the default boot section for your system.

4. Reduce the Altix system flush duration:

   a. Reboot the machine but interrupt the boot process when the **EFI Boot Manager** menu comes up and 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"
      ```

      Note: Use this procedure only for systems running time-critical real-time applications. (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.

   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. All but a few CPU-specific threads will be running within the bootcpuset. The real-time CPUs will be isolated from the effects of load balancing.

5. When the system comes back up, run the `reactboot.pl` script to determine how to do the run-time set up of your system:

   ```
   # reactboot.pl
   ```

The `reactboot.pl` script does the following:

- Creates cpusets labeled `rtcpu`*N* for each CPU that is not part of the bootcpuset. 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 `reactcfg.pl`.
- Redirects interrupts to the CPUs contained in the bootcpuset. This is just a working approximation; someone familiar with your site’s hardware configuration and system requirements should set up the interrupt redirection. To set up the interrupt redirection, echo the correct values to the `/proc/irq/`*interrupt*`/smp_affinity` cpumasks. For more information, see "Redirect Interrupts" on page 40.

SGI recommends that you run a set-up script equivalent to `reactboot.pl` after every reboot to set up your real-time cpusets and set the affinity of device interrupts. You can modify `reactboot.pl` to configure interrupt affinity and cpusets according to your own needs.

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/rtcpu`*N* cpusets created above).

For more information, see the `cpuset`(1) and `libcpuset`(3) man pages and the `set_affinity_cpuset()` function in the `rt_sample` application in Appendix A, "Example Application" on page 109.

Example `reactcfg.pl` Output

Example 10-1 is an example of the output generated by `reactcfg.pl` (line breaks added here for readability). The script modifies the bootcpusets based on the CPUs you provide, adds the `rt` label, and changes the `append` values. (The script output also contains information currently contained in the `/etc/elilo.conf` file, such as comments.)

Example 10-1 Example `reactcfg.pl` Output

```
# reactcfg.pl 2-3
realtime cpus   2 3
bootcpuset cpus 0 1
bootcpuset mems 0


/etc/bootcpuset.conf
cpus 0,1
mems 0

Does the above /etc/bootcpuset.conf look OK? (y=yes)y


/etc/elilo.conf:
# This file has been transformed by /sbin/elilo.
# Please do NOT edit here -- edit /etc/elilo.conf instead!
# Otherwise your changes will be lost e.g. during kernel-update.
#
# Modified by YaST2. Last modification on Fri Nov  5 16:32:33 2004


prompt
timeout = 80
read-only
relocatable
default = rt
append = "selinux=0 console=ttyS0,115200n8 kdb=on splash=silent elevator=cfq
  thash_entries=2097152"

image = vmlinuz
    ###Don't change this comment - YaST2 identifier: Original name: linux###
    label = Linux
    initrd = initrd
    root = /dev/sdb6

image = vmlinuz
    ###Don't change this comment - YaST2 identifier: Original name: failsafe###
    label = Failsafe
    initrd = initrd
    root = /dev/sdb6
    append = "ide=nodma nohalt noresume selinux=0 barrier=off"

image = vmlinuz
    label = rt
    initrd = initrd
    root = /dev/sdb6
append = "selinux=0 console=ttyS0,115200n8 kdb=on splash=silent elevator=cfq
  thash_entries=2097152 init=/sbin/bootcpuset isolcpus=2,3"


Does the above elilo.conf look OK? (y=yes overwrites /etc/elilo.conf)y


Backup files are /etc/elilo.conf.rtbak and /etc/bootcpuset.conf.rtbak
Please reboot your system to restrict and isolate cpus
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.
```

Example `reactboot.pl` Output

Example 10-2 is an example of the output generated by `reactboot.pl`.

Example 10-2 Example `reactboot.pl` Output

```
# reactboot.pl
CPUSET /dev/cpuset/rtcpu2 created
CPUSET /dev/cpuset/rtcpu3 created
```

Troubleshooting

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 "Running the Sample Application" on page 130.

- 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 "Running the Sample Application" on page 130. 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:

```
$ 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):

```
$ 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.

Example Application

This appendix provides excerpts from a sample real-time application to be used with REACT. The application is composed of two different parts:

- *Example kernel module*, which shows a simple example of creating and building a driver with a standard miscellaneous device interface. It is provided primarily to service the example user-space application.

  The example kernel module provides two modes of operation:

  – A thread can wait for an interrupt to be sent to the CPU on which it last ran

  – A thread can send an interrupt to a specific CPU

  Upon receiving an interrupt, the waiting process will simply read the system real-time clock (RTC) and return that value to user-space.

- *Example user-space application*, which shows examples of the following concepts:

  – Creating cpusets and assigning threads to them, thereby changing thread/CPU affinity

  – Changing thread/CPU affinity without cpusets

  – Changing scheduling policies and priorities

  – Reading the system RTC via the memory-mapped `mmtimer` driver

  – Locking memory

  This program runs as a dual process:

  – A *sending process* (`IPI_send`) that does the following:

    1. Sets its CPU affinity either via cpusets or via the `sched_setaffinity()` call. For more information, see the `sched_setaffinity`(2) man page.
    2. Sets its scheduling policy and priority.
    3. Uses the example kernel module driver to repeatedly send interrupts to a given destination CPU.

– A *receiving process* (`main`) that does the following:
  1. Locks its current and future memory (if requested).
  2. Sets its CPU affinity via cpusets.
  3. Sets its scheduling policy and priority.
  4. Uses the example kernel module driver to do the following:
     - Wait for interrupts
     - Retrieve kernel RTC readings
     - Read the memory-mapped RTC for comparison readings

Note: Header information has been truncated from the examples for readability.

This example application requires that the SGI ProPack for Linux `mmtimer-devel` RPM is installed on the system.

Building and Loading a Kernel Module

To build the `rt_sample_mod` application kernel module, do the following on the target system:

1. Log in to the target system as `root`.
2. Set up a kernel module development environment:
   a. Ensure that the `kernel-rt-source-*.rpm` RPM is installed.

   b. Ensure that the `kernel-rt` kernel is in use.

   c. Change to the `source` directory:
   ```
   # cd /lib/modules/‘uname -r‘/source
   ```
   d. Create the configuration file:
   ```
   # make cloneconfig
   ```
   e. Build the kernel:
   ```
   # make compressed
   ```

3. Make the `rt_sample_mod` directory:

   ```
   # mkdir /root/rt_sample_mod
   ```

4. Change to the `rt_sample_mod` directory:

   ```
   # cd /root/rt_sample_mod
   ```

5. Copy the `rt_mod_sample` source files to the target system. For example:

   ```
   # scp bob@server:~/rt_sample/rt_sample_mod/* .
   ```

6. Build the `rt_sample_mod.ko` file:

   ```
   # make -C /lib/modules/‘uname -r‘/source SUBDIRS=$PWD modules
   ```

7. Copy the `rt_sample_mod.ko` file to the `‘uname -r‘` directory:

   ```
   # cp rt_sample_mod.ko /lib/modules/‘uname -r‘
   ```

   For more information, see the `uname`(1) man page.

8. Make a dependency file:

   ```
   # depmod
   ```

   For more information, see the `depmod`(8) man page.

9. Load the `rt_sample_mod` module:

   ```
   # modprobe rt_sample_mod
   ```

   For more information, see the `modprobe`(8) man page.

You can then use the `rt_sample_mod` kernel module with the `rt_sample` application.

Kernel-space Makefile

Example A-1 is a portion of the kernel-space Makefile.

Example A-1 Kernel-space Makefile Extract

```
CFLAGS += -D__KERNEL__ -DMODULE -g

obj-m += rt_sample_mod.o
rt_sample_mod-objs := rt_samplemod.o ipi.o
```

rt_samplemod.h

Example A-2 is a portion of the rt_samplemod.h file.

Example A-2 rt_samplemod.h Extract

```
#define MAXCPUS 64

/* We might store more information here if we were measuring response
 * times at different points.
 */
typedef struct rttask_struct {
        unsigned long long rtctime;     /* Time waiting while thread is woken */
        struct semaphore sysrt_sema;    /* Mutex to wait on */
        int ready;                      /* Waiting for interprocessor interrupt (IPI)*/
} rttask_t;

extern int rt_register_percpu_irq(void);
extern void rt_send_IPI_single (int);
extern void rt_free_percpu_irq(void);

extern rttask_t * rttasks[MAXCPUS];

#define IA64_IPI_RT     0xfc    /* Interprocessor interrupt for real-time test */
```

`rt_samplemod.c`

Example A-3 is a portion of the `rt_samplemod.c` file.

Example A-3 `rt_samplemod.c` Extract

```
#include <linux/miscdevice.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/sn/addrs.h>
#include <asm/sn/shub_mmr.h>
#include <asm/sn/clksupport.h>

#include "rt_samplemod.h"

MODULE_LICENSE("GPL");

#define RTMOD_NAME      "rt_sample_mod"

static ssize_t rt_samplemod_run(struct file * file, const char * buf, size_t count,
                loff_t *f_pos);
/*
 * Avoid the big kernel lock (BKL) by making rt_samplemod_run() a 'write' call rather
 *  than an ioctl(), to avoid introducing latency.
 */
static struct file_operations rt_samplemod_fops = {
        owner:  THIS_MODULE,
        write:  rt_samplemod_run,
};
static struct miscdevice rt_samplemod_miscdev = {
        152,
        RTMOD_NAME,
        &rt_samplemod_fops
};

#define RT_INTR_WAIT    1
#define RT_SEND_IPI     2

rttask_t * rttasks[MAXCPUS];

int rt_samplemod_init(void) {
        memset(rttasks, 0, sizeof(rttasks));

        /* Register this device */
        strcpy(rt_samplemod_miscdev.devfs_name, RTMOD_NAME);
        if (misc_register(&rt_samplemod_miscdev)) {
                printk(KERN_ERR "%s: failed to register device\n", RTMOD_NAME);
                return -1;
        }
        /* Set up the interrupt request */
        rt_register_percpu_irq();

        printk(KERN_INFO "%s has been initialized\n",RTMOD_NAME);

        return 0;
}

void rt_samplemod_exit(void) {
        int i;

        rt_free_percpu_irq();
        for (i=0; i<MAXCPUS; i++) {
                if (rttasks[i] != NULL) kfree(rttasks[i]);
        }
        misc_deregister(&rt_samplemod_miscdev);

        printk(KERN_INFO "%s exit\n",RTMOD_NAME);
}

/*
 * Avoid the BKL by making rt_samplemod_run() a
 * 'write' call rather than an ioctl(), to avoid introducing latency.
 * Pass everything in as 'buf', ignoring everything else.
 */
static ssize_t
rt_samplemod_run(struct file * file, const char * buf, size_t count,
                loff_t *f_pos)
{
        int error = 0;
        unsigned long long cmd;
        unsigned long long arg1;
        unsigned long long * argp = (unsigned long long *)(buf+sizeof(cmd));
```

```
if (copy_from_user(&cmd, (void *)buf, sizeof(cmd))) {
        return -EFAULT;
}
if (__get_user(arg1, argp)) {
        return -EFAULT;
}

switch (cmd) {
        case RT_INTR_WAIT:
                {
                rttask_t * curr;
                unsigned long cpu = smp_processor_id();


                /* First time here on this CPU */
                if (rttasks[cpu] == NULL) {
                        /* Allocate and initialize the per-cpu data area */
                        curr = kmalloc(sizeof(rttask_t),GFP_KERNEL);
                        if (curr == NULL) {
                                error = -ENOMEM;
                                break;
                        }
                        curr->ready = 0;
                        init_MUTEX_LOCKED(&curr->sysrt_sema);
                        rttasks[cpu] = curr;
                } else {
                        curr = rttasks[cpu];
                }
                /* The send will not happen until 'ready' is set and
                 * the semaphore is down, so there is no race.
                 */
                curr->ready = 1;
                down_interruptible(&curr->sysrt_sema);

                /* This is how we read the RTC time in the kernel.
                 * Note that we are only measuring the time to go from
                 * kernel space to user space.  An actual interrupt response
                 * measurement would take RTC time readings at other
                 * points within this driver.
                 */
```

```
                        curr->rtctime = rtc_time();
                        argp = (unsigned long long *)(buf+sizeof(cmd));
                        /* Copy RTC reading to user space */
                        error = __put_user(curr->rtctime, argp);
                        }
                        break;
                case RT_SEND_IPI:
                        {
                        int cpuid = arg1; /* Destination cpu */
                        rttask_t * tsk = rttasks[cpuid];

                        if ((tsk == NULL) || (!tsk->ready) ||
                                (tsk->sysrt_sema.count.counter != -1)) {
                                error = -ENOMEM;
                                break;
                        }
                        /*
                         * Only single threaded here per CPU, so no race with
                         * above.
                         */
                        tsk->ready = 0;
                        rt_send_IPI_single(cpuid);
                        }
                        break;
                default:
                        error = -EINVAL;
                        break;
        }
        return error;
}

module_init(rt_samplemod_init);
module_exit(rt_samplemod_exit);
```

ipi.c

Example A-4 is a portion of the `ipi.c` file.

Example A-4 `ipi.c` Extract

```
#include <linux/irq.h>
#include <asm/sn/clksupport.h>
#include <asm/sn/intr.h>
#include <asm/hw_irq.h>
#include "rt_samplemod.h"


/*
 * This is the actual interrupt handler.  We could get an RTC reading
 * here if we were measuring response times
 */
static irqreturn_t
rt_handle_IPI (int irq, void *dev_id, struct pt_regs *regs)
{
        int this_cpu = get_cpu(); /* Disable preemption and get CPU number */
        rttask_t * tsk = rttasks[this_cpu];

        mb();

        up(&tsk->sysrt_sema); /* Raise the semaphore */

        put_cpu(); /* Re-enable preemption */

        return IRQ_HANDLED;
}

static inline void
rt_send_IPI(int cpuid, int vector, int delivery_mode)
{
        int nasid = cpuid_to_nasid(cpuid);
        long physid = cpu_physical_id(cpuid);

        sn_send_IPI_phys(nasid, physid, vector, delivery_mode);
}
/*
 * Replaces send_IPI_single, but no per-CPU operation bit is set.
 * It also disables preemption and interrupt requests itself, and calls
 * rt_send_IPI rather than the nonexported platform_send_ipi
 * (which becomes sn2_send_IPI).
 * Caller must ensure the existence of the rttasks element.
 */
void
rt_send_IPI_single (int dest_cpu)
{
        unsigned long s;

        get_cpu(); /* Prevent preemption. */
        /* For performance, make sure we are not interrupted until the IPI is sent */
        local_irq_save(s);
        rt_send_IPI(dest_cpu, IA64_IPI_RT, IA64_IPI_DM_INT);
        local_irq_restore(s);
        put_cpu();
}

/* Register our interrupt request */
int
rt_register_percpu_irq(void)
{
        /* With SA_PERCPU_IRQ, this becomes similar to register_percpu_irq */
        return request_irq(IA64_IPI_RT, rt_handle_IPI, SA_PERCPU_IRQ,
                                "Realtime IPI", NULL);
}

/* Free our interrupt request */
void
rt_free_percpu_irq(void)
{
        free_irq(IA64_IPI_RT,NULL);
}
```

Building and Loading a User-Space Application

To build and load the user-space module, enter the following:

```
[user@linux user]$ make
```

User-space `Makefile`

Example A-5 is a portion of the user-space `Makefile`.

Example A-5 User-space `Makefile` Extract

```
TARGETS=rt_sample
COMFILES=common.c mmtimer.c
COMOBJECTS=$(COMFILES)
SFILES=rt_sample.c
SOBJECTS=$(SFILES)

default: $(TARGETS)

rt_sample: $(SOBJECTS) $(COMOBJECTS) -lcpuset
```

`rt_sample.c`

Example A-6 is a portion of the `rt_sample.c` file.

Example A-6 `rt_sample.c` Extract

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <signal.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sn/mmtimer.h>
```

```
#include "common.h"

#define SLEEPTIME       3000            /* usecs to sleep between interrupts */

extern unsigned long * open_mmtimer(void);


/* name of the device, usually in /dev */
#define RTMOD_NAME "rt_sample_mod"
#define DEFAULT_RUN_TIME        60              /* Run for 60 seconds default */
#define RT_INTR_WAIT     1
#define RT_SEND_IPI      2

typedef signed short    cpuid_t;        /* cpuid */
typedef struct parm_s {
        unsigned long cmd;
        unsigned long arg1;
} parm_t;


cpuid_t cpu = 2;
cpuid_t ipi_cpu = 3;
int memlock = 0;

/* These externs get set in mmtimer.c */
extern unsigned long period;
extern unsigned long mask;
int rt_fd;


char * usage =
"usage: rt_sample [-chm] [-p rcv_processor] [-o snd_processor] [-t seconds]\n"
"       -c  IPI src processor is in cpuset\n"
"       -h  print usage instructions\n"
"       -m  lock memory\n"
"       -p# receiving processor\n"
"       -o# IPI src processor\n"
"       -t# total run time (secs)\n";


/*
```

```
 * Print out the command line options and quit.
 */
static void
usage_exit(void)
{
        fprintf(stderr, "%s", usage);
        exit(0);
}

/* This routine sends IPIs to the receiving CPU. */
void
IPI_send(int cpuset)
{
        parm_t rti;
        struct timespec req;

        if (cpuset)
                set_affinity_cpuset(ipi_cpu);
        else
                set_affinity(ipi_cpu);

        set_scheduling(10, SCHED_RR);
        sleep(5);  /* Give other IPI_send processes a chance to start if
                        they are on the same CPU */

        req.tv_sec = 0; req.tv_nsec = SLEEPTIME * 1000;
        rti.cmd = RT_SEND_IPI;
        rti.arg1 = cpu;
        while (1) {
                /* Send interrupt */
                if (write(rt_fd, &rti, 0) != 0) {
                        /* printf("Send not successful\n"); */
                        continue;
                }
                nanosleep(&req, NULL);
        }
}

int
main(int argc, char **argv)
{
        unsigned long max_delta = 0;
        unsigned long min_delta = 0x7fffffffffffff;
        unsigned long total_time = DEFAULT_RUN_TIME; /* sec */
        unsigned long start;
        unsigned long run_time;
        unsigned long * ptimer;
        int cid = 0;    /* default to no cpusets for IPI source CPU */
        pid_t child;
        int opt;

        static char* opt_string = "cmp:o:t:h";
        while ((opt = getopt(argc, argv, opt_string)) >= 0) {
                switch (opt){
                case 'c':
                        cid = 1;
                        break;
                case 'm':
                        memlock = 1;
                        break;
                case 'p':
                        cpu = atoi(optarg);
                        break;
                case 'o':
                        ipi_cpu = atoi(optarg);
                        break;
                case 't':
                        total_time = atoi(optarg);
                        break;
                case 'h':
                default:
                        usage_exit();
                }
        }
        /* Open rt_sample_mod sample driver */
        if ((rt_fd = open("/dev/"RTMOD_NAME,O_RDWR)) == -1) {
                printf("Failed to open /dev/%s\n",RTMOD_NAME);
                exit(1);
        }
        /* Open mmtimer driver and get memory mapped RTC address */
        if (!(ptimer = open_mmtimer())) {
                exit(1);
        }

        /* Fork off the IPI sender */
        child = fork();
        if (child==0) {
                IPI_send(cid);
                exit(0);
        }

        /* Do not allow paging if selected */
        if (memlock &&
            (mlockall(MCL_CURRENT | MCL_FUTURE) < 0)) {
                perror("memlock");
                exit(1);
        }

        /* Create and attach to a single CPU cpuset for this CPU. */
        set_affinity_cpuset(cpu);

        /* Change the scheduling to priority and policy. */
        set_scheduling(30, SCHED_FIFO);

        /* Read the RTC via mmtimer */
        start = *ptimer;

        /* Loop for total_time receiving IPIs from IPI_send */
        do {
                time_t ct;
                struct timespec thrwake_ts, current_ts;
                unsigned long current;
                parm_t rti;

                unsigned long delta;

                rti.cmd = RT_INTR_WAIT;
                rti.arg1 = 0;

                /* Wait to be signaled that an interrupt has arrived */
                if (write(rt_fd, &rti, 0) != 0) {
                        perror("write to rt_fd failed");
                        continue;
                }

                /* The rest of this loop represents whatever processing
                 * must be done on the data from the driver.
                 *
                 * This simple example just takes the RTC time difference
                 * between the RTC value measured in the kernel and the
                 * current RTC time.
                 */

                /* Read the RTC via mmtimer */
                current = *ptimer;

                /* delta is in nsec */
                delta = ((current & mask) - (rti.arg1 & mask)) * period;
                if (delta > max_delta) {
                        max_delta = delta;
                }
                if (delta < min_delta) {
                        min_delta = delta;
                }
                /* elapsed time so far is in sec */
                run_time = (((current & mask) - (start & mask)) * period) / 1e9;
        } while (run_time < total_time);

        if (kill(child, SIGTERM)) {
                perror("Unable to kill child process\n");
        }
        wait(child, NULL, 0);
        printf("Minimum delta %ld nsec\n",min_delta);
        printf("Maximum delta %ld nsec\n",max_delta);
        return 0;
}
```

common.h

Example A-7 is a portion of the `common.h` file.

Example A-7 `common.h` Extract

```
extern void set_affinity(int);
extern void set_affinity_cpuset(int);
extern void set_scheduling(int, int);
```

common.c

Example A-8 is a portion of the `common.c` file.

Example A-8 `common.c` Extract

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sched.h>
#include <bitmask.h>
#include <cpuset.h>
/* NR_CPUS is size kernel is configured for */
#define NR_CPUS 512
#define BITMASK_SIZE NR_CPUS/64

/* Set the current process to run on CPU <cpu>.
 * Note that this CPU must not be part of a cpuset other than the
 * one the current process is in.
 */
void set_affinity(int cpu) {
        unsigned long cpus[BITMASK_SIZE];

        if (cpu > (NR_CPUS-1)) {
                printf("set_affinity: Invalid cpu %d\n",cpu);
                exit(1);
        }
        cpus[cpu/64] = 1 << (cpu % 64);

        if (sched_setaffinity(0, sizeof(cpus), cpus)) {
                perror("set_affinity");
                exit(1);
        }
}

/*
 * Set up a cpuset with cpus==<cpu> and mems==<cpu>/2.
 * Then set the current process to run on CPU <cpu> in cpuset 'rtcpu<cpu>'
 * For more information about mems, see the cpuset(1) man page.
 */
void set_affinity_cpuset(int cpu) {
        char path[50];
        struct cpuset *cp;
        struct bitmask *cpus;
        struct bitmask *mems;

        sprintf(path, "/rtcpu%d", cpu);
        cp = cpuset_alloc();
        if (cp == NULL) {
                printf("cpuset_alloc failed\n");
                exit(1);
        }
        cpus = bitmask_alloc(cpuset_cpus_nbits());
        mems = bitmask_alloc(cpuset_mems_nbits());

        /* Set up bitmasks for cpus and mems */
        bitmask_setbit(cpus, cpu);
        bitmask_setbit(mems, cpu/2);

        /* Set the 'cpus' value in the cpuset structure */
        if (cpuset_setcpus(cp, cpus) == -1) {
                perror("cpuset_setcpus");
                exit(1);
        }
        /* Set the 'mems' value in the cpuset structure */
        if (cpuset_setmems(cp, mems) == -1) {
                perror("cpuset_setmems");
                exit(1);
        }
        /* Create the actual cpuset in /dev/cpuset */
        if (cpuset_create(path, cp) == -1) {
                if (errno != EEXIST) {
                        perror("cpuset_create");
                        exit(1);
                }
        }

        cpuset_free(cp);
        bitmask_free(cpus);
        bitmask_free(mems);

        /* Move the process into the cpuset */
        if (cpuset_move(getpid(), path) == -1) {
                perror("cpuset_move");
                exit(1);
        }

}

/* Change the scheduling policy and priority <pri>
   A priority of -1 passed in selects the maximum priority.
 */
void set_scheduling(int pri, int policy) {
        struct sched_param param;
        int maxpri;

        if (policy != SCHED_FIFO && policy != SCHED_RR) {
                printf("Bad policy %d set!\n", policy);
                exit(1);
        }
        /* Get the maximum priority value for this policy */
        maxpri = sched_get_priority_max(policy);

        if (pri==-1)
                param.sched_priority = maxpri;
        else
                param.sched_priority = pri > maxpri ? maxpri : pri;

        /* Change the priority and policy if pri != 0 */
        if ((pri==0) || (sched_setscheduler(0, policy, &param)==-1)) {
                perror("Unable to set sched policy");
        } else {
                printf("Sched policy set to %d, pri=%d\n",
                        policy, param.sched_priority);
        }
}
```

mmtimer.c

Example A-9 is a portion of the `mmtimer.c` file:

Example A-9 `mmtimer.c` Extract

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <cntl.h>
#include <n/mmtimer.h>
#include "common.h"


unsigned long period;
unsigned long mask = 0xffffffffffffffff;

/*
 * Open RTC interface and get parameters.
 */
unsigned long * open_mmtimer(void) {
        int fd, result, offset;
        unsigned long * iotimer_addr64;
        unsigned long val = 0;

        if((fd = open("/dev/"MMTIMER_NAME, O_RDONLY)) == -1) {
                printf("failed to open /dev/%s", MMTIMER_NAME);
                return NULL;
        }

        /*
```

```
         * Can we mmap in the counter?
         */
        if ((result = ioctl(fd, MMTIMER_MMAPAVAIL, 0)) != 1) {
                printf("mmap unavailable\n");
                return NULL;
        }

        /*
         * Get the offset for each clock
         */
        if ((offset = ioctl(fd, MMTIMER_GETOFFSET, 0)) == -ENOSYS) {
                printf("offset unavailable for clock\n");
                return NULL;
        }

        /*
         * Get the frequency in Hz and calculate the period in nsec
         */
        if((result = ioctl(fd, MMTIMER_GETFREQ, &val)) != -ENOSYS)
                if(val < 10000000) /* less than 10 MHz? */
                        printf("ERROR: frequency only %ld MHz, should be >= 10 MHz\n", val/1000000);
                else {
                        printf("frequency: %ld MHz\n", val/1000000);
                }
        else
                printf("ERROR: failed to get frequency\n");

        /*
         * Get the resolution in femtoseconds (1e-15) and save period in nsec.
         */
        if((result = ioctl(fd, MMTIMER_GETRES, &val)) == -ENOSYS) {
                printf("ERROR: failed to get resolution\n");
                return NULL;
        }
        period = (val/1e+6);
        printf("period: %lld nsec\n",period);

        /*
         * Get the number of valid bits and compute the mask.
         */
        if((result = ioctl(fd, MMTIMER_GETBITS, 0)) == -ENOSYS) {
                printf("ERROR: can’t get number of bits in counter\n");
                return NULL;
        }
        mask = ~(0xffffffffffffffff << result);

        /*
         * Map the clock.
         */
        iotimer_addr64 = (unsigned long *)mmap(0, 0x4000, PROT_READ,
                                                        MAP_PRIVATE, fd, 0);
        if (iotimer_addr64 <= 0) {
                printf("failed to mmap /dev/%s",MMTIMER_NAME);
                return NULL;
        }
        iotimer_addr64 += offset;
        close(fd);

        return iotimer_addr64;
}
```

Running the Sample Application

To run the `rt_sample` user-space application, do the following:

1. Ensure that you have the `rt_sample_mod` module loaded by using the `lsmod`(1) command, which should show it in the module list:

   ```
   # lsmod
           Module                  Size  Used by
           rt_sample_mod          72784  0
           ..
   ```

   If the output does not include `rt_sample_mod`, follow the instructions in "Building and Loading a Kernel Module" on page 110.

2. Execute the `rt_sample` command as desired.

The `rt_sample` command has the following options:

`-c`	Specifies that the CPU sending the interrupt must be in the cpuset. By default, the CPU does not have to be in a cpuset.

`-h`	Prints the usage instructions.
`-m`	Locks memory. By default, memory is not locked.
`-p` *receiving_CPU*	Specifies the number of the CPU that is receiving; this CPU must be in a cpuset. The default is CPU 2.
`-o` *sending_CPU*	Specifies the CPU sending the interrupt. The default is CPU 3.
`-t` *seconds*	Specifies the total run time in seconds. The default is 60 seconds.

Therefore, to run `rt_sample` for 60 seconds (the default) with CPU 3 as the sender in a cpuset and CPU 2 as the receiver (which always must be in a cpuset), enter the following:

```
rt_sample -c -p 2 -o 3 -m
```

You can monitor the IPI interrupts produced and received by examining the `/proc/interrupts` file. Doing so for the above example should result in output similar to the following:

```
# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
 28:         10         10         10         10          LSAPIC  cpe_poll
..
252:          0          0       6839          0          LSAPIC  Realtime IPI
..
```

To see where the threads are running, use the `-F` option to the `ps` command (the seventh column, `PSR`, shows the last processor on which the thread ran). For example:

```
# ps -FC rt_sample
UID        PID  PPID  C    SZ  RSS PSR STIME TTY          TIME CMD
root      6407  6151  0   185 2880   2 15:04 pts/0    00:00:00 rt_sample -c -p 2 -o 3 -m
root      6408  6407  0   185 1248   3 15:04 pts/0    00:00:00 rt_sample -c -p 2 -o 3 -m
                                     ^
                                     ^
                                     displays the last CPU on which the thread ran
```

High-Resolution Timer Example

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 <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 <ibgen.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 ULI Program

Example C-1 shows how user-level interrupts 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.

Example D-1 Reading MAC Addresses

```
/* 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 );
}
```

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.

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 and did not, an overrun exception is signalled.

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.

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*.

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*.

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).

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, 15
activity thread management, 63
address space (locking in memory), 94
aircraft simulator, 3
append argument, 43
application example, 109
asynchronous I/O, 85
average data rate, 5

B

barrier facility, 35
base address registers (BARs), 87
bootcpuset, 39, 103
bootcpuset.conf, 103

C

C language, 6
cache warming, 58
callout deregistration, 25
callout mechanism, 23
callout registration, 24
cap_set_proc, 38
capabilities scheduling, 38
cat, 107
clock processor, 40
clock_getres(), 11
clock_gettime, 11, 12
CLOCK_MONOTONIC, 11
CLOCK_PROCESS_CPUTIME_ID, 12
CLOCK_REALTIME, 11
clock_settime, 11
CLOCK_SGI_CYCLE, 13
clocks, 10
compatibility library (IRIX to Linux) RPM, 100
configuration, 101
configuration script RPMs, 99
configuration scripts, 101
console driver, 41
console interrupts, 9
console node, 41
control law process stimulator, 4
controller thread, 72
CPU
  isolating, 43
  restricting, 8, 42
  workload control, 33
CPU 0, 40, 41
CPU-bound, 7
cpumasks, 104
cpuset, 39, 104
cpusets, 50
cycles per second, 2

D

data collection system, 4
deregistration of callout, 25
dev attribute file, 16
/dev/extint#, 94
device service time, 44, 47
device special file, 94
device-driver time base, 50
direct RTC access, 12
disciplines, 7
disk I/O optimization, 85
distributed applications, 14
documentation RPM, 100

driver deregistration, 23
driver interface, 19
driver registration, 19
driver template, 26

E

earnings-based scheduler, 8
ei driver, 18
elilo, 43
/etc/elilo.conf, 43, 103
exception types, 76
external interrupt ingest, 29
external interrupt with frame scheduler, 65
external interrupts, 15
extint, 16, 94
extint_device, 19
extint_properties, 19

F

fastcall, 11
feedback loop, 2
first-in-first-out, 8
flush duration, 40
fork(), 72
frame interval, 2
frame rate, 2
frame scheduler, 10, 49, 51
  advantages, 10
  API, 54
  background discipline, 67
  basics, 50
  continuable discipline, 68
  current frame extension, 77
  design process, 70
  exception counts, 78
  exception handling, 75
  exception policies, 77
  exception types, 76
  external interrupt, 65
  frame scheduler controller, 53, 61
  frs_run flag, 59
  frs_yield flag, 59
  hhigh-resolution timer, 65
  interval timers not used with, 83
  library interface for C programs, 56
  major frame, 51
  managing activity threads, 63
  minor frame, 51
  multiple exceptions, 77
  multiple synchronized, 61
  overrun exception, 66, 76
  overrunnable discipline, 68
  pausing, 62
  preparing the system, 71
  process outline for single, 71
  real-time discipline, 66
  repeat frame, 76
  scheduling disciplines, 66
  scheduling rules of, 59
  sequence error handling, 82
  signal use under, 80
  signals in an activity thread, 81
  signals produced by, 80, 81
  starting up a single scheduler, 61
  starting up multiple schedulers, 62
  synchronized schedulers, 73
  thread structure, 58
  time base selection, 51, 64
  underrun exception, 66, 76
  underrunable discipline, 67
  using consecutive minor frames, 68
  warming up cache, 58
frame scheduler controller, 53, 61
  receives signals, 81
frs
  See "frame scheduler", 49
frs_create(), 56, 72
frs_create_master(), 56, 72, 74
frs_create_slave(), 56, 74

frs_create_vmaster(), 56, 72, 74
frs_destroy(), 57, 73–75
frs_enqueue(), 56, 62, 72
frs_fsched_info_t, 54
frs_getattr(), 57, 78
frs_getqueuelen(), 57, 63
frs_intr_info_t, 54, 55
frs_join, 56
frs_join(), , 58, 62, 73–75
frs_overrun_info_t(), 79
frs_pinsert(), 56, 63
frs_premove(), 57, 63, 81
frs_pthread_enqueue(), 56, 59, 66, 72, 74, 75
frs_pthread_getattr(), 57, 78
frs_pthread_insert, 56
frs_pthread_insert(), 63
frs_pthread_readqueue(), 57, 63
frs_pthread_register(), 57
frs_pthread_remove(), 57, 63, 81
frs_pthread_setattr(), 56, 77
  example code, 78
frs_queue_info_t, 54
frs_readqueue(), 57, 63
frs_recv_info_t, 54
frs_resume(), 57, 62
frs_run, 60
frs_setattr(), 56, 77
frs_start, 57
frs_start(), 62, 72, 74, 75
frs_stop, 57
frs_stop(), 62
frs_t, 54
frs_userintr(), 57
frs_yield, 57, 60
frs_yield(), , 58
  with overrunable discipline, 68
fsync, 86

G

generating a REACT system configuration, 101
glibc, 12
global variables and ULI, 93
ground vehicle simulator, 3

H

hard real-time program, 1
hardware latency, 44, 45
hardware–in–the–loop simulator, 4
high output modes, 27
high-resolution POSIX timers, 13
high-resolution timer, 65, 133
Hz (hertz, cycles per second), 2

I

I/O interrupts, 9
I/O-bound, 7
IDE driver, 27
implementation functions, 20
ingest section for external interrupts, 29
installation, 99
interchassis communication, 13
internal driver circuit I/O connectors, 31
interrupt
  group. See interrupt group, 64
  See also user-level interrupt (ULI), 89
interrupt control, 9
interrupt group, 64
interrupt information template, 55
interrupt notification interface, 23
interrupt propagation delay, 45
interrupt redirection, 40
interrupt response time
  components, 44
  definition of, 44
  minimizing, 47
interrupt service routines (ISRs), 47, 89
interval

See "frame interval", 2
interval time counter, 12
interval timer, 83
introduction, 1
IOC4 chip, 15
IOC4 driver, 26
ioc4_extint, 94
IOC4–specific character special device and class, 26
ioctl, 18
IRIX compatibility library, 100
IRIX functions
ULI_sleep(), 93
isolating a CPU, 43
isolcpus, 43
ITC, 12
ITC register access, 12

K

kbar, 35
kernel barrier facility, 35
kernel critical section, 46
kernel facilities for real-time, 7
kernel module building/loading, 110
kernel module insertion/removal, 43
kernel RPM, 99
kernel scheduling, 33
kernel thread control, 39

L

L2 system controller, 41
latency, 44, 45
libc reentrant version, 93
libcpuset, 104
libsgirt, 42
libuli, 90
linux/drivers/char/ioc4_extint.c, 26
load balancing, 43
locking memory, 94
locking virtual memory, 8
low output modes, 28
low-level driver interface, 19
low-level driver template, 26

M

MAC address sample program, 145
major frame, 51
master scheduler, 73
maximum response time guarantee, 44
mechanism for callout, 23
memory locking (virtual), 8
memory-mapped I/O, 85
Message-Passing Interface (MPI), 14
minor frame, 51, 59
mlock(), 8, 92
mlockall(), 8, 92
mode attribute file, 17
mode switch, 44, 47
modelist attribute file, 17
MPI, 14
ms (milliseconds), 2
msync, 85, 86
multiple devices and ULI, 93
multiple independent drivers, 26
multiprocessor architecture
and frame scheduler, 61

N

new pthreads library (NPTL), 38
nice value, 7
normal-time program, 1
NPTL, 38

O

operator, 2
output modes, 27
overhead work, 39
overrun, 5
overrun exception, 60
overrun in frame scheduler, 66

P

page fault, 8
param.h, 34
PCI devices, 87
peak data rate, 5
period attribute file, 17
physical interfaces, 30
physical memory requirements, 8
POSIX real-time policies, 8
POSIX real-time specification 1003.1-2003, 86
POSIX timers, 13
power plant simulator, 3
priorities, 33
priority band, 34
/proc manipulation, 9
/proc/bus/pci/devices, 87
/proc/interrupts, 41, 102, 107
process control, 5
process mapping to CPU, 8
profile.pl, 107
programming language for REACT, 6
propagation delay, 45
provider attribute file, 18
ps, 35, 107
pthread priority, 38
pthread_attr_setinheritsched(), 38
pthread_attr_setschedparam(), 38
pthread_attr_setschedpolicy(), 38
pthread_attr_t, 38
pthread_attr_t(), 54
pthread_create(), 38, 72
PTHREAD_EXPLICIT_SCHED, 38
PTHREAD_INHERIT_SCHED, 39
pthread_setschedparam(), 38
pthread_t, 54
pulse output modes, 28

Q

quantum attribute file, 18

R

rate
  See "frame rate", 2
reactboot.pl, 41
reactcfg.pl, 39, 42, 43, 101
README.relnotes, 100
real-time applications, 1
real-time clock (RTC), 10
real-time priority band, 34
real-time program, 1
  and frame scheduler, 10
reentrant C library, 93
register access, 12
register format, 28
registration of callout, 24
release notes, 100
repeat frame, 76
response time, 44
response time guarantee, 44
restricting a CPU, 42
round-robin, 8
RPMs, 99
RTC, 10
RTC access, 12
rtcpu, 50

S

SAL console driver, 41
sample system configuration script RPMs, 99
sample system configuration scripts, 101
sched_setparam(), 34
sched_setscheduler(), 8, 34
scheduling, 33
scheduling capabilities, 38
scheduling discipline
  See also frame scheduler scheduling disciplines, 66
scheduling disciplines, 7
script RPMs, 99
scripts, 101
SGI high-resolution POSIX timers, 13
sgiioc4 driver, 27
sig_dequeue, 82
sig_overrun, 82
sig_underrun, 81
sig_unframesched, 82
signal, 80
signal handler, 74, 75
SIGRTMIN, 82
SIGUSR1, 81
SIGUSR2, 82
simulator, 2
single frame scheduler start, 61
slave scheduler, 73
SLES9, 99
SLES9 modules, 100
SN hub device interrupts, 47
socket programming, 14
soft real-time program, 1
software latency, 44, 45
source attribute file, 18
sourcelist attribute file, 18
special scheduling disciplines, 7
stimulator, 2
strace, 35, 107
strobe output modes, 28
SUSE Linux, 99
sync group, 73
synchronous I/O, 86
/sys/class/extint/extint#/ , 16
/sys/class/ioc4_intout/intout*/dev, 28
sysconf(_SC_CLK_TCK), 12
sysfs attribute files, 16
system abstraction layer (SAL) console driver, 41
system configuration generation, 101
system configuration script RPMs, 99
system configuration scripts, 101
system controllers, 41
system flush duration, 40
system-call time base, 50

T

test_and_clear_bit, 99
test_and_set_bit, 99
thread, 53
thread control, 39
time base for frame scheduler, 64
time base support, 50
time estimation, 60
time slices, 34
time-share applications, 8
timer interrupts, 9, 34
timer_create, 11
timer_create(), 13
timer_settime(), 13
timer_t, 13
timers, 13
toggle output modes, 28
top, 107
transport delay, 2
troubleshooting, 107

U

ULI

  See "User-level interrupt (ULI) facility", 89
uli, 90
ULI_block_intr, 91
ULI_destroy, 91
ULI_register_irq(), 90, 96
ULI_sleep(), 91
ULI_unblock_intr, 91
ULI_wakeup(), 91
underrun exception, 60
underrun, in frame scheduler, 66
unsupported hardware device capabilities, 25
usecs (microseconds), 44
user thread control, 39
user thread dispatch, 47
user-level interrupt (ULI), 89
  concurrency, 93
  global variables, 93
  handler interaction, 96
  initializing, 93
  interrupt handler registration, 95
  multiple devices, 93
  mutual exclusion, 97
  overview, 89
  per-IRQ handler, 96
  program address space locking, 94
  restrictions on handler, 91
  ULI_block_intr(), 97
  ULI_sleep (), 96
  ULI_sleep () function, 93
  ULI_wakeup () function, 96
user-level interrupts (ULI)
  sample program, 139
user-space application building/loading, 119
/usr/include/asm/bitops.h, 99
/usr/include/asm/param.h, 34
/usr/include/sn/timer.h, 12
/usr/include/sys/pthread.h, 54
/usr/share/src/react/examples, 55

V

virtual memory locking, 8
virtual reality simulator, 3
volatile keyword, 93
Vsync time base, 50

W

wave stimulator, 4