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

007–4746–015

New Features in this Guide

This revision contains the following:

- Support for changing permissions via the `react`(8) command. See:
  - "`react` Command-Line Summary" on page 107
  - "Specifying Permissions" on page 112

  Note: The `react-permissions.pl` script is deprecated.

- Support for the following new API routines for user capabilities:
  - "`cpu_sysrt_set_allowed_caps`" on page 126, which sets all capabilities allowed by the inherited set
  - "`cpu_sysrt_set_caps`" on page 127, which sets select capabilities allowed by the inherited set

  Prior to using these routines, you must install a software package that will set up the user's inheritable capability set at login time, such as the `pam_capability` package. See "Installing the `pam_capability` Package" on page 128.

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
004	July 2006 Revision to support REACT real-time for Linux 5.0
005	February 2007 Revision to support REACT real-time for Linux 5.1
006	June 2007 Revision to support REACT real-time for Linux 5.2
007	September 2007 Revision to support REACT real-time for Linux 5.3
008	December 2007 Revision to support REACT real-time for Linux 5.4
009	March 2008 Revision to support REACT real-time for Linux 5.5
010	June 2008 Revision to support REACT real-time for Linux 6.0
011	September 2008 Revision to support REACT real-time for Linux 6.1
012	January 2010 Revision to support REACT real-time for Linux 7.0

013	May 2010 Revision to support REACT real-time for Linux 7.1 (part of the SGI ProPack 7.1 release)
014	October 2010 Revision to support SGI REACT 1.0 (a new separate release, and a member of the SGI Performance Suite)
015	January 2011 Revision to support SGI REACT 1.1

Contents

About This Guide . . . . . **xxiii**
Audience xxiii
What This Guide Contains xxiii
Related Publications and Sites xxv
Conventions xxvi
Obtaining Publications xxvi
Reader Comments xxvii

1. Introduction **1**
Real-Time Programs 1
Real-Time Applications 2
Simulators and Stimulators 2
Aircraft Simulators 3
Ground Vehicle Simulators 3
Plant Control Simulators 3
Virtual Reality Simulators 4
Hardware-in-the-Loop Simulators 4
Control Law Processor Stimulator 4
Wave Tank Stimulator 5
Data Collection Systems 5
Process Control Systems 6
REACT™ Features 6
REACT Requirements 7
REACT RPMs 7

2. Linux and REACT Support for Real–Time Programs **9**

Kernel Facilities 9
Special Scheduling Disciplines 9
Virtual Memory Locking 10
Processes Mapping and CPUs 10
Interrupt Distribution Control 11
Frame Scheduler 11
Clocks and Timers (Altix® UV 1000 and Altix UV 100) 12
Clocks 12
Direct RTC Access 14
Interchassis Communication 14
Socket Programming 14
Message-Passing Interface (MPI) 15

3. External Interrupts 17

Abstraction Layer 17
`sysfs` Attribute Files 18
The `/dev/extint#` Device 20
Counting Interrupts 20
Waiting for Interrupts 20
Exclusively Accessing a Device 20
Low-Level Driver Interface 23
Driver Registration 24
Implementation Functions 24
When an External Interrupt Occurs 28
Driver Deregistration 28
Interrupt Notification Interface 28
Callout Mechanism 29
Callout Registration 29

Callout Deregistration 30
Making Use of Unsupported Hardware Device Capabilities 31
Low-level Driver Template 31
Example: SGI IOC4 PCI Device 32
Multiple Independent Drivers 32
External Interrupt Output 34
External Interrupt Ingest 36
Physical Interfaces 36

4. CPU Workload 39
Using Priorities and Scheduling Queues 39
Scheduling Concepts 39
Timer Interrupts 40
Real-Time Priority Band 40
Setting Pthread Priority 41
Controlling Kernel and User Threads 42
Minimizing Overhead Work 43
Avoid the Clock Processor (CPU 0) 43
Redirect Interrupts 43
Restrict, Isolate, and Shield CPUs 44
Restricting a CPU from Scheduled Work and Isolating it from Scheduler Load Balancing 45
Shielding a CPU from Timer Interrupts 45
Avoid Kernel Module Insertion and Removal 46
Avoid Filesystem Mounts 47
Understanding Interrupt Response Time 47
Maximum Response Time Guarantee 48
Components of Interrupt Response Time 48

Hardware Latency 49
Software Latency 49
Kernel Critical Sections 50
Interrupt Threads Dispatch 50
Device Service 51
Interrupt Service Routines 51
User Threads Dispatch 51
Mode Switch 51
Minimizing Interrupt Response Time 51

5. Using the Frame Scheduler 53

Frame Scheduler Concepts 53
Frame Scheduler Basics 54
Thread Programming Model 55
Frame Scheduling 55
Controller Thread 58
Frame Scheduler API 58
Interrupt Information Templates 59
Library Interface for C Programs 60
Thread Execution 62
Scheduling Within a Minor Frame 64
Scheduler Flags `frs_run` and `frs_yield` 64
Detecting Overrun and Underrun 64
Estimating Available Time 65
Synchronizing Multiple Schedulers 66
Starting a Single Scheduler 66
Starting Multiple Schedulers 67
Pausing Frame Schedulers 67

Managing Activity Threads 68
Selecting a Time Base 69
High-Resolution Timer 70
External Interrupts as a Time Base 70
Using the Scheduling Disciplines 71
Real-Time Discipline 71
Underrunable Discipline 72
Overrunnable Discipline 72
Continuable Discipline 73
Background Discipline 73
Using Multiple Consecutive Minor Frames 73
Designing an Application for the Frame Scheduler 75
Preparing the System 76
Implementing a Single Frame Scheduler 77
Implementing Synchronized Schedulers 78
Synchronized Scheduler Concepts 79
Master Controller Thread 79
Slave Controller Thread 80
Handling Frame Scheduler Exceptions 81
Exception Types 81
Exception Handling Policies 82
Injecting a Repeat Frame 82
Extending the Current Frame 82
Dealing With Multiple Exceptions 83
Setting Exception Policies 83
Querying Counts of Exceptions 84
Using Signals Under the Frame Scheduler 86

Handling Signals in the Frame Scheduler Controller . . . 86
Handling Signals in an Activity Thread . . . 87
Setting Frame Scheduler Signals . . . 87
Handling a Sequence Error . . . 88
Using Timers with the Frame Scheduler . . . 89

6. Disk I/O Optimization . . . 91

Memory-Mapped I/O . . . 91
Asynchronous I/O . . . 91
Conventional Synchronous I/O . . . 92
Asynchronous I/O Basics . . . 92

7. PCI Devices . . . 93

8. User-Level Interrupts . . . 97

Overview of ULI . . . 98
Restrictions on the ULI Handler . . . 99
Planning for Concurrency: Declaring Global Variables . . . 101
Using Multiple Devices . . . 101
Setting Up ULI . . . 101
Opening the Device Special File . . . 102
Locking the Program Address Space . . . 102
Registering the Interrupt Handler . . . 103
Registering a Per-IRQ Handler . . . 103
Interacting With the Handler . . . 104
Achieving Mutual Exclusion . . . 105

9. REACT System Configuration . . . 107

`react` Command-Line Summary . . . 107

Initially Configuring REACT 110
Changing the Configuration 111
Disabling REACT 111
Reenabling REACT 112
Specifying Permissions 112
Showing the Configuration 114
Getting Trace Information 115
Running a Process on a Real-Time CPU 117

10. Using the REACT Library 119

REACT Library Routines 119
`cpu_shield` 120
`cpu_sysrt_add` 121
`cpu_sysrt_delete` 122
`cpu_sysrt_info` 122
`cpu_sysrt_irq` 123
`cpu_sysrt_perm` 124
`cpu_sysrt_runon` 126
`cpu_sysrt_set_allowed_caps` 126
`cpu_sysrt_set_caps` 127
Accessing REACT Library Routines 128
Installing the `pam_capability` Package 128
Example Code Using the REACT Library Routines 129

11. SGI Linux Trace 133

Overview of SGI Linux Trace 133
Installing SGI Linux Trace 134
Gathering Trace Data 135

Invoking the `tracevisualizer` GUI . . . 135
Recording Events . . . 136
Trace Files and Event Types . . . 138
Exiting from the `tracedaemon` Command Line . . . 141
Monitoring Trace Events . . . 141
Opening a Trace File . . . 142
Zooming In On An Event . . . 142
Changing the Time Frame . . . 142
Seeing Process Details . . . 143
Seeing All Event Trace Details . . . 143
Filtering Events Based on CPU . . . 143
Exiting from the `tracevisualizer` GUI . . . 143
Removing SGI Linux Trace . . . 144

12. Using the SGI Linux Trace User Library . . . 145

SLT User Library Routines . . . 145
`slt_close_utrace` . . . 145
`slt_open_utrace` . . . 146
`slt_user_trace` . . . 146
Accessing SLT User Library Routines . . . 147
Example Code Using the SLT User Library Routines . . . 147
Generating User and Kernel Data . . . 148
Examining the Data . . . 148
Manually Including User Events in `slt-cpu.all` . . . 149

13. Troubleshooting . . . 151

Diagnostic Tools . . . 151

Problem Removing `/rtcpus` 154

Appendix A. Example Application 155

Setting Up External Interrupts 157

Building and Loading the Kernel Module 158

Building the User-Space Application 159

Running the Sample Application 159

Matrix Multiply Mode Examples 161

Netlink Socket Benchmark Mode Examples 161

`set_affinity` code 162

Appendix B. High-Resolution Timer Example 165

Appendix C. Sample User-Level Interrupt Programs 171

`uli_sample` Sample Program 171

`uli_ei` Sample Program 172

Glossary 173

Index 183

Figures

Figure 3-1 Output and Input Connectors for Interface Circuits of IO9 and PCI-RT-Z Cards 37
Figure 4-1 Components of Interrupt Response Time 49
Figure 5-1 Major and Minor Frames 56
Figure 8-1 ULI Functional Overview 97
Figure 8-2 ULI Handler Functions 100
Figure A-1 Example Work Flow 157

Tables

Table 3-1 Register Format 35
Table 5-1 Frame Scheduler Types 58
Table 5-2 Pthread Types 59
Table 5-3 Frame Scheduler Operations 60
Table 5-4 Activity Thread Functions 68
Table 5-5 Signals Passed in `frs_signal_info_t` 87
Table 8-1 Common Arguments for Registration Functions 98
Table 11-1 Trace Events that are Recorded 139

Examples

Example 3-1 Searching for an Unused External Interrupt Device 21

Example 5-1 Skeleton of an Activity Thread 62

Example 5-2 Alternate Skeleton of an Activity Thread 63

Example 5-3 Function to Set `INJECTFRAME` Exception Policy 84

Example 5-4 Function to Set `STRETCH` Exception Policy 84

Example 5-5 Function to Return a Sum of Exception Counts (pthread Model) 85

Example 5-6 Function to Set Frame Scheduler Signals 88

Example 5-7 Minimal Activity Process as a Timer 89

Example B-1 High-Resolution Timer 165

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 SGI® 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 and applications, summarizes the features that REACT provides, and lists installation requirements
- Chapter 2, "Linux and REACT Support for Real–Time Programs" on page 9, provides an overview of how Linux and REACT support real-time programs
- Chapter 3, "External Interrupts" on page 17, discusses the external interrupts feature and, as an example, the SGI IOC4 PCI device
- Chapter 4, "CPU Workload" on page 39, 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 53, describes how to structure a real-time program as a family of independent, cooperating activities, running on multiple CPUs, scheduled in sequence at the frame rate of the application
- Chapter 6, "Disk I/O Optimization" on page 91, describes how to set up disk I/O to meet real-time constraints, including the use of memory-mapped and asynchronous I/O
- Chapter 7, "PCI Devices" on page 93, discusses the Linux PCI interface
- Chapter 8, "User-Level Interrupts" on page 97, discusses the facility that is intended to simplify the creation of device drivers for unsupported devices
- Chapter 9, "REACT System Configuration" on page 107, explains how to configure real-time CPUs
- Chapter 10, "Using the REACT Library" on page 119, explains how to use the REACT C application programming interface (API) to change the configuration of real-time CPUs from program control without affecting the boot-up configuration for real-time processing
- Chapter 11, "SGI Linux Trace" on page 133, discusses the feature that generates traces for kernel events such as interrupt handling, scheduling, and system calls.
- Chapter 12, "Using the SGI Linux Trace User Library" on page 145, explains how to use the SGI Linux Trace (SLT) user library C API to generate SLT user events
- Chapter 13, "Troubleshooting" on page 151, discusses diagnostic tools that apply to real-time applications and common problems
- Appendix A, "Example Application" on page 155, provides excerpts of application modules to be used with REACT
- Appendix B, "High-Resolution Timer Example " on page 165, demonstrates the use of SGI high-resolution timers
- Appendix C, "Sample User-Level Interrupt Programs" on page 171, contains a sample program that shows how user-level interrupts are used

Related Publications and Sites

The following may be useful:

- Available from the online SGI Technical Publications Library:
  - The user guide for your SGI system
  - *SGI Performance Suite 1.1 Start Here*
  - *Linux Configuration and Operations Guide*
  - *SGI L1 and L2 Controller Software User's Guide*
  - *TP9500 Remote Mirror Premium Feature-Factory*
  - *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 SGI servers, see:

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

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.
ms (or msec)	Millisecond (1 ms is .001 seconds)
ns	Nanosecond (1 ns is .000000001 seconds)
us (or usec)	Microsecond (1 us is .000001 seconds)

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 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

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

- Send mail to the following address:

  SGI
  Technical Publications
  46600 Landing Parkway
  Fremont, CA 94538

SGI values your comments and will respond to them promptly.

Introduction

This chapter discusses the following:

- "Real-Time Programs" on page 1
- "Real-Time Applications" on page 2
- "REACT™ Features" on page 6
- "REACT Requirements" on page 7
- "REACT RPMs" on page 7

Real-Time Programs

A *real-time program* is any program that must maintain a fixed, absolute timing relationship with an external hardware device:

- A *hard real-time program* experiences a catastrophic error if it misses a deadline
- A *firm real-time program* 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* 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 5
- "Process Control Systems" on page 6

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 ms (.01667 seconds). 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™ Features

REACT real-time for Linux® provides the following features:

- SGI Linux Trace debug kernel to provide trace information for analyzing the impact of kernel operations on application performance.
- The `react` command helps you easily generate and configure a real-time system. See Chapter 9, "REACT System Configuration" on page 107.
- 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.

Note: *CPU* refers to cores (not sockets).

REACT Requirements

REACT requires the following:

- One of the following operating systems:
  - SUSE® Linux® Enterprise Server 11 Service Pack 1 (SLES 11 SP1) or later
  - Red Hat® Enterprise Linux® 6 (RHEL 6) or later
- x86-64 Intel® processors with at least 2 cores (4 cores are preferred)
- Sufficient memory so that the system can run the operating system and the real-time applications without swapping

For best performance, run REACT on SGI x86-64 servers.

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

REACT RPMs

The following RPMs are used for REACT:

- **Required** RPMs:
  - Cpuset and bitmask:

    ```
    cpuset-utils
    libbitmask
    libcpuset
    ```

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

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

– REACT configuration (see Chapter 9, "REACT System Configuration" on page 107) and library:

  `react-utils`

– REACT library:

  `libreact`

– REACT licensing (for `react-utils`):

  `lk`

Linux and REACT Support for Real–Time Programs

This chapter provides an overview of how Linux and REACT support real-time programs:

- "Kernel Facilities" on page 9
- "Frame Scheduler" on page 11
- "Clocks and Timers (Altix® UV 1000 and Altix UV 100)" on page 12
- "Interchassis Communication" on page 14

Kernel Facilities

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

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

Special Scheduling Disciplines

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

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

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

- First-in-first-out
- Round-robin

These policies share a real-time priority band consisting of 99 priorities. For more information about scheduling, see "Real-Time Priority Band" on page 40 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 and isolate them from the effects of scheduler load-balancing.
2. 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 43.

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. For example:

```
[root@linux root]# head -1 /proc/interrupts && grep 'SAL console' /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 53.

Clocks and Timers (Altix® UV 1000 and Altix UV 100)

This section discusses the following for Altix UV 1000 and Altix UV 100:

- "Clocks" on page 12
- "Direct RTC Access" on page 14

Clocks

Note: This section does not apply to third-party x86-64 or Altix UV 10 servers .

SGI Altix UV 1000 and Altix UV 100 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 5-ns intervals and using the local APIC timer for timer interrupts (`timer_create()`), which has a 1–ns resolution.

The RTC is 56 bits wide, which ensures that it will not wrap around zero unless the system has been running for more than 11.42 years. 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.

The RTC is the basis for system time, which may be obtained via the `clock_gettime` function call that is implemented in conformance with the POSIX standard. `clock_gettime` takes an argument that describes 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 bypasses the context switch typically necessary for a full system call. SGI recommends that you use `clock_gettime` for all time needs.

`CLOCK_REALTIME` and `CLOCK_MONOTONIC` report the correct resolution. You can use either `CLOCK_REALTIME` or `CLOCK_MONOTONIC` to generate signals via the `timer_create()` function.

To determine the tick frequency, use the `sysconf(_SC_CLK_TCK)` function. The `sysconf(_SC_CLK_TCK)` function will always return the right value on Altix UV 1000 and Altix UV 100 systems.

Direct RTC Access

Note: This section does not apply to third-party x86-64 or Altix UV 10 servers.

In some situations, the overhead of the `clock_gettime` fastcall may be too high. In that case, direct memory-mapped access to the Altix UV 1000 or Altix UV 100 RTC counter is useful. (See the comments in `mmtimer.h`.)

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

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

Interchassis Communication

This section discusses the following:

- "Socket Programming"
- "Message-Passing Interface (MPI)"

The performance of both sockets and MPI depends on the speed of the underlying network. The network that connects nodes (systems) in an array product has a very high bandwidth.

Socket Programming

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

Message-Passing Interface (MPI)

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.

External Interrupts

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

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

This chapter discusses the following:

- "Abstraction Layer" on page 17
- "Making Use of Unsupported Hardware Device Capabilities " on page 31
- "Low-level Driver Template" on page 31
- "Example: SGI IOC4 PCI Device" on page 32

Abstraction Layer

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

This section discusses the following:

- "`sysfs` Attribute Files" on page 18
- "The `/dev/extint#` Device" on page 20
- "Low-Level Driver Interface" on page 23
- "Interrupt Notification Interface" on page 28

sysfs Attribute Files

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

File	Description
dev	Contains the major and minor number of the abstracted external interrupt device. If sysfs, hotplug, and udev are configured appropriately, udev will automatically create a /dev/extint# character special device file with this major and minor number. If you prefer, you may manually invoke mknod(1) to create the character special device file. Once created, this device file provides a counter that can be used by applications in a variety of ways. See "The /dev/extint# Device" on page 20.
mode	Contains the shape of the output signal for interrupt generation. For example, SGI's IOC4 chip can set the output to one of the following: high, low, pulse, strobe, or toggle. For more information, see "External Interrupt Output" on page 34.
modelist	Contains the list of available **valid** output modes, one per line. These strings are the legal valid values that can be written to the mode attribute. For more information, see "External Interrupt Output" on page 34.
	Note: For the SGI IOC4 chip, there are other values that may be read from the mode attribute file that do not appear in modelist; these represent **invalid** hardware states. Only the modes present from the modelist are valid settings to be written to the mode attribute.
period	Contains the repetition interval for periodic output signals (such as repeated strobes, automatic toggling). This period is specified in nanoseconds, and is written as a string. For more information, see "External Interrupt Output" on page 34.
provider	Contains an indication of which low-level hardware driver and device instance are attached to the external interrupt interface. This string is free-form and is determined by the low-level driver. For example, the SGI IOC4 low-level driver will return a string of the form ioc4_intout#.

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

`quantum` Contains the interval to which any writes of the `period` attribute will be rounded. Because external interrupt output hardware may not support nanosecond granularity for output periods, this attribute allows you to determine the supported granularity. The behavior of the interrupt output (when a value that is not a multiple of the quantum is written to the `period` attribute) is determined by the specific low-level external interrupt drive. However, generally the low-level driver should round to the nearest available quantum multiple. For example, suppose the `quantum` value is `7800`. If a value of `75000` was written into the `period` attribute, this would represent 9.6 quantums. The actual period will be rounded to 10 quantums, or 78000 nanoseconds. The actual period will be returned upon subsequent reads from the `period` attribute. For more information, see "External Interrupt Output" on page 34.

`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.

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 /dev/extint# Device

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

- "Counting Interrupts" on page 20
- "Waiting for Interrupts" on page 20
- "Exclusively Accessing a Device" on page 20

Counting Interrupts

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

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

Waiting for Interrupts

The poll(2) and select(2) system calls allow a process to wait for an interrupt to trigger:

- poll() indicates whether an interrupt has occurred since the last open(2) or read() of the file
- select() blocks until the next interrupt is received

Exclusively Accessing a Device

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

Note: You must define the _GNU_SOURCE macro before including the header files in order to use the LOCK_MAND flag on the call to flock(2).

When this lock is obtained, only a process that has access to the corresponding file descriptor will be able to write to the attribute files for that device. Any other process

that attempts a write(2) system call on one of these attribute files will fail with errno set to EAGAIN.

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

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

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

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

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

Example 3-1 Searching for an Unused External Interrupt Device

```
#define _GNU_SOURCE

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

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

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

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

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

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

```
                } else {
                        /* Some other error occurred. */
                        printf("Unexpected error locking %s: %s\n",
                               devfile, strerror(errno));
                }
                /* Try the next device. */
                found = 0;
                close(fd);
                goto try_again;
        }

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

        /* Application code begins... */

        /* ... application code ends. */

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

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

Low-Level Driver Interface

The `extint_properties` and `extint_device` structures provide the low-level driver interface to the abstraction layer driver. The `/usr/local/include/extint.h` file defines the structures and function prototypes.

This section discusses the following:

- "Driver Registration" on page 24
- "Implementation Functions" on page 24
- "When an External Interrupt Occurs" on page 28
- "Driver Deregistration" on page 28

Driver Registration

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

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

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

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

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

The low-level driver uses this value 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 table is as follows:

```
struct extint_properties {
        /* Owner module */
        struct module *owner;

        /* Get/set generation mode */
        ssize_t (*get_mode)(struct extint_device * ed, char *buf);
        ssize_t (*set_mode)(struct extint_device * ed, const char *buf,
                            size_t count);

        /* Get generation mode list */
        ssize_t (*get_modelist)(struct extint_device * ed, char *buf);
```

```
        /* Get/set generation period */
        unsigned long (*get_period)(struct extint_device * ed);
        ssize_t (*set_period)(struct extint_device * ed, unsigned long period);

        /* Get low-level provider name */
        ssize_t (*get_provider)(struct extint_device *ed, char *buf);

        /* Generation period quantum */
        unsigned long (*get_quantum)(struct extint_device * ed);

        /* Get/set ingest source */
        ssize_t (*get_source)(struct extint_device * ed, char *buf);
        ssize_t (*set_source)(struct extint_device * ed, const char *buf,
                              size_t count);

        /* Get ingest source list */
        ssize_t (*get_sourcelist)(struct extint_device * ed, char *buf);

        /* Arm/disarm timer */
        int64_t (*arm_timer)(struct extint_device * ed, int64_t ns, int when);
        void (*disarm_timer)(struct extint_device * ed);
};
```

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

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

```
extint_get_devdata(ed);
```

You can update the value by using the following call:

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

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

Field	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.
`arm_timer`	Sets up the external interrupt device to generate an interrupt at a specified time. The time is specified in nanoseconds via the second argument. The third parameter may be set to the values `EXTINT_TIMER_RELATIVE` or `EXTINT_TIMER_ABSOLUTE`. The third parameter controls whether the time is relative to the moment the function is called or is absolute system time, (as returned by the `getnstimeofday()` system call). Interrupt notifications occur through the standard external interrupt callout mechanism described in

	"Interrupt Notification Interface" on page 28. This field may be set to `NULL` if the low-level driver does not support timer functionality.
`disarm_timer`	Cancels a pending interrupt, if any, scheduled to be delivered due to a prior call to the `arm_timer()` function. If the previously scheduled interrupt has already occurred, it is not necessary to call `disarm_timer()`, and calling `disarm_timer()` when no interrupt is pending should be harmless. This field may be set to `NULL` if the low-level driver does not support timer functionality.

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

Callout Mechanism

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

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

Callout Registration

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

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

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

The second argument is one of the following structures:

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

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

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

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

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

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

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

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

Callout Deregistration

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

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

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

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

Making Use of Unsupported Hardware Device Capabilities

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

- Add a new attribute to the abstraction layer by modifying `struct extint_properties` to add appropriate interface routines and update any existing drivers as necessary.
- Have the low-level driver create its own device class and corresponding attributes and/or character special devices. This method is preferred and is required if the capability is dependent on the hardware in a method that cannot be abstracted.

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

Low-level Driver Template

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

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

Example: SGI IOC4 PCI Device

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

- "Multiple Independent Drivers" on page 32
- "External Interrupt Output" on page 34
- "External Interrupt Ingest" on page 36
- "Physical Interfaces" on page 36

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

Multiple Independent Drivers

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

The IOC4 driver actually consists of the following independent drivers:

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

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

`sgiioc4`	The IDE driver for IOC4. It hooks up to the `ioc4` driver via the appropriate registration, probe, and remove functions.
`ioc4_serial`	The serial driver for IOC4. It hooks up to the `ioc4` driver via the appropriate registration, probe, and remove functions.
`ioc4_extint`	The external interrupts driver for IOC4. IOC4-based I/O controller cards provide an electrical interface to the outside world that can be used to ingest and generate a simple signal for the following purposes: • On the output side, one of the jacks can provide a small selection of output modes (low, high, a single strobe, toggling, and pulses at a specified interval) that create a 0-5V electrical output. • On the input side, one of the jacks will cause the IOC4 to generate a PCI interrupt on the transition edge of an electrical signal. This driver registers with the `extint` abstracted external interrupt driver and lets it take care of the user-facing details.

External Interrupt Output

The output section provides several modes of output:

Mode	Description
`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. The abstraction layer device's `modelist` attribute contains available modes.
`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 I/O controller cards. The `loopback` source is the output of the IOC4's interrupt output section. The source is configurable by the abstraction layer device's `source` attribute. You can find available sources in the abstraction layer device's `sourcelist` attribute.

For example, to set up loopback mode:

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

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

Physical Interfaces

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

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

- Tip: +5V input
- Ring: interrupt input (active low, optoisolated)
- Sleeve: chassis ground/cable shield

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

Figure 3-1 shows the internal driver circuit for the output connector and the internal receiver circuit for the input connector.

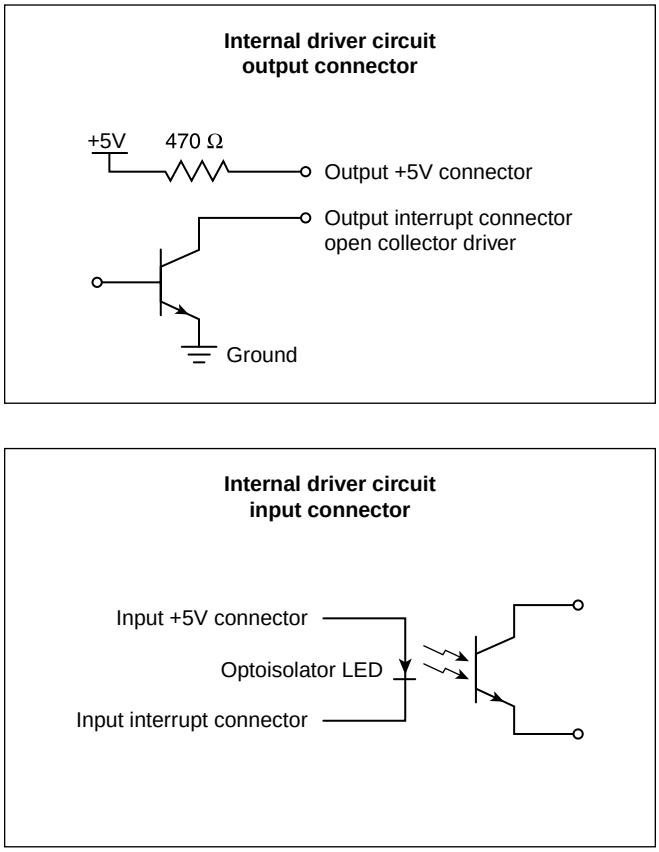

Figure 3-1 Output and Input Connectors for Interface Circuits of IO9 and PCI-RT-Z Cards

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 it must be a +5V source current-limited with series resistor of at least 420 ohms in order to avoid damaging the optoisolator.

Note: The resistor on the output circuit of IO9 and PCI-RT-Z cards is 470 ohms. To protect the input circuit on these cards from damage, a resistor of at least 420 ohms is required.

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 39
- "Minimizing Overhead Work" on page 43
- "Understanding Interrupt Response Time" on page 47
- "Minimizing Interrupt Response Time" on page 51

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 39
- "Setting Pthread Priority" on page 41
- "Controlling Kernel and User Threads" on page 42

Scheduling Concepts

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

- "Timer Interrupts" on page 40
- "Real-Time Priority Band" on page 40

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

Timer Interrupts

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

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

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

Real-Time Priority Band

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

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

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

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

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

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

Setting Pthread Priority

The Linux pthreads library shipped with SLES and RHEL 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 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 `react` command to configure the real-time CPUs; see Chapter 9, "REACT System Configuration" on page 107.

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 43
- "Redirect Interrupts" on page 43
- "Restrict, Isolate, and Shield CPUs" on page 44
- "Avoid Kernel Module Insertion and Removal" on page 46
- "Avoid Filesystem Mounts" on page 47

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. Therefore, you should not to use CPU 0 for running real-time processes.

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*.

You can use the `react` command to redirect interrupts; for more information, see Chapter 9, "REACT System Configuration" on page 107.

Note: SGI recommends that someone with knowledge of the system configuration use `react` to redirect only the interrupts that must be moved.

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

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

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

To view the IRQ/CPU affinity, use the `less` command to view the `smp_affinity` file. For example:

```
[root@linux root]# less /proc/irq/62/smp_affinity
```

Note: To avoid any potential viewing problems, you should use `less`(1) rather than `cat`(1) to view the `smp_affinity` file.

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

Restrict, Isolate, and Shield CPUs

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

- *Restricted and isolated*, which prevents the CPU from running scheduled processes and removes the CPU from load balancing considerations, a time-consuming scheduler operation.
- *Shielded*, which switches off the timer (scheduler) interrupts that would normally be scheduled on the CPU. These are a source of jitter, but only a minor source of interrupt response latency. Shielding should only be done for short periods where basically jitter-free program execution is required.

You should use the `react` command to create a real-time CPU that is restricted and isolated. For more information, see Chapter 9, "REACT System Configuration" on page 107.

You can also use the REACT `C` application programming interface (API) to restrict and isolate a CPU. See Chapter 10, "Using the REACT Library" on page 119.

Restricting a CPU from Scheduled Work and Isolating it from Scheduler Load Balancing

You can restrict one or more CPUs from running scheduled processes and isolate them from scheduler load balancing by designating them as realtime CPUs with the `react` command.

The only processes that can use a restricted CPU are those processes that you assign to it, along with certain per-CPU kernel threads. Isolating a CPU removes one source of unpredictable delays from a real-time program and helps further minimize the latency of interrupt handling.

To restrict one or more CPUs, use the `react -r` command documented in Chapter 9, "REACT System Configuration" on page 107.

After restricting a CPU, you can assign processes to it using the SGI `cpuset` command. See "Running a Process on a Real-Time CPU" on page 117.

Each `rtcpu` is set to be `cpu_exclusive`.

To remove the CPU restriction, allowing the CPU to execute any scheduled process, see "Changing the Configuration" on page 111.

Shielding a CPU from Timer Interrupts

You can shield a CPU from the normally scheduled Linux timer (scheduler) interrupts. For more information on timer interrupts, see "Timer Interrupts" on page 40.

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

When a CPU's timer interrupts are switched off, scheduling on that CPU ceases. A thread must not yield the CPU (sleep) unless it expects to be awoken by an external event such as an I/O interrupt or if timer interrupts will be switched back on before it must be scheduled again.

Note: Be aware of the following:

- Prolonged periods of shielding might eventually result in system resource depletion. System resource depletion usually takes the form of out-of-memory conditions, eventually causing forced shutdown of the application. The kernel ring buffer will indicate this situation by showing a stack trace for the application and a `No available memory in cpuset:` message. To view the kernel ring buffer, run the `dmesg` command.
- You should ensure that all threads are placed in their appropriate cpusets prior to calling `cpu_shield()` anywhere on the system. Movement between cpusets will be held off during periods where any processor's timer interrupts are switched off. After timer interrupts for all processors are switched back on, any pending cpuset thread movement will occur.

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

1. Load the `sgi-shield` kernel module. For example:

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

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

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

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

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

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

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

Avoid Kernel Module Insertion and Removal

The insertion and removal of Linux kernel modules (such as by using `modprobe` or `insmod`/`rmmod`) requires that a kernel thread be started on all active CPUs (including

isolated CPUs) in order to synchronously stop them. This process allows safe lockless-module list manipulation. However, these kernel threads can interfere with thread wakeup and, for brief periods, the ability to receive interrupts.

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

Avoid Filesystem Mounts

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

Prior to running a time-critical application, you should complete all filesystem mounts that may be necessary during application execution. Filesystem unmounts during application execution should be avoided. This includes `autofs` mounts performed by `automount`.

Understanding Interrupt Response Time

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

This section discusses the following:

- "Maximum Response Time Guarantee" on page 48
- "Components of Interrupt Response Time" on page 48

Maximum Response Time Guarantee

In properly configured systems, interrupt response time is guaranteed not to exceed 30 microseconds (usecs) for SGI x86–64 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:

Time	Description
Hardware latency	The time required to make a CPU respond to an interrupt signal. See "Hardware Latency" on page 49.
Software latency	The time required to dispatch an interrupt thread. See "Software Latency" on page 49.
Device service	The time the device driver spends processing the interrupt and dispatching a user thread. See "Device Service" on page 51.
Mode switch	The time it takes for a thread to switch from kernel mode to user mode. See "Mode Switch" on page 51.

Figure 4-1 diagrams the parts discussed in the following sections.

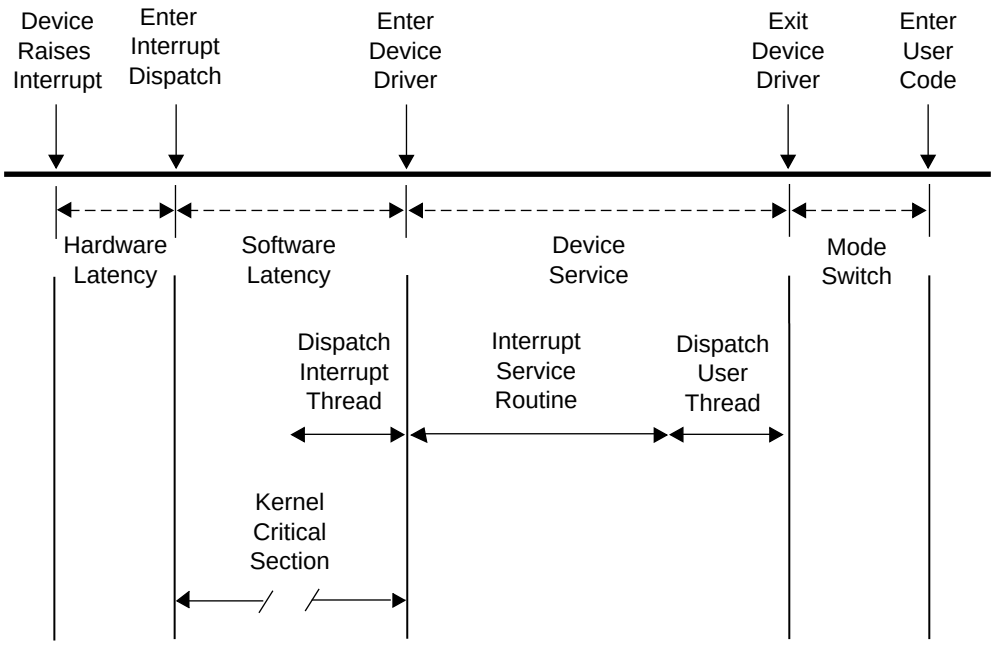

Figure 4-1 Components of Interrupt Response Time

Hardware Latency

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

The time taken for these events is the hardware latency, or *interrupt propagation delay*. For more information, see Chapter 7, "PCI Devices" on page 93.

Software Latency

Software latency is affected by the following:

- "Kernel Critical Sections" on page 50
- "Interrupt Threads Dispatch" on page 50

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 and Isolating it from Scheduler Load Balancing" on page 45.

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 43.

Device Service

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.

Note: With third-party x86-64 and Altix UV 10 architecture, the CC clock source is supplied by the PCI-RT-Z card. HUB hardware timers are not available on third-party x86-64 and Altix UV 10 platforms. On these platforms, you must have one PCI-RT-Z card per asynchronous frame scheduler. Multiple frame schedulers running synchronously can use a single PCI-RT-Z card, however.

This chapter discusses the following:

- "Frame Scheduler Concepts" on page 53
- "Selecting a Time Base" on page 69
- "Using the Scheduling Disciplines" on page 71
- "Using Multiple Consecutive Minor Frames" on page 73
- "Designing an Application for the Frame Scheduler" on page 75
- "Preparing the System" on page 76
- "Implementing a Single Frame Scheduler" on page 77
- "Implementing Synchronized Schedulers" on page 78
- "Handling Frame Scheduler Exceptions" on page 81
- "Using Signals Under the Frame Scheduler" on page 86
- "Using Timers with the Frame Scheduler" on page 89

Frame Scheduler Concepts

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

This section discusses the following:

- "Frame Scheduler Basics" on page 54
- "Thread Programming Model" on page 55
- "Frame Scheduling" on page 55
- "Controller Thread" on page 58
- "Frame Scheduler API" on page 58
- "Interrupt Information Templates" on page 59
- "Library Interface for C Programs" on page 60
- "Thread Execution" on page 62
- "Scheduling Within a Minor Frame" on page 64
- "Synchronizing Multiple Schedulers" on page 66
- "Starting a Single Scheduler" on page 66
- "Starting Multiple Schedulers" on page 67
- "Pausing Frame Schedulers" on page 67
- "Managing Activity Threads" on page 68

Frame Scheduler Basics

When a frame scheduler dispatches threads on one CPU, it does not completely supersede the operation of the normal Linux scheduler. The CPUs chosen for frame scheduling must be restricted and isolated (see "Restrict, Isolate, and Shield CPUs" on page 44). You do not have to set up cpusets for the frame-scheduled CPUs because the frame scheduler will set up cpusets named `rtcpu`*N* (where *N* is the CPU number) if this has not already been done. For more control over cpuset parameters, you can create your own cpusets for the frame scheduler to use (one per CPU, and one CPU per cpuset), by naming them exactly as mentioned above.

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

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

For more information, see "Preparing the System" on page 76.

Thread Programming Model

The frame scheduler supports pthreads.

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

A traditional Linux process has a single active thread that starts after 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 69.) 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 70)
- The Vsync (vertical retrace) interrupt from the graphics subsystem
- A device interrupt from a specially modified device driver
- A system call (normally used for debugging)

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

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

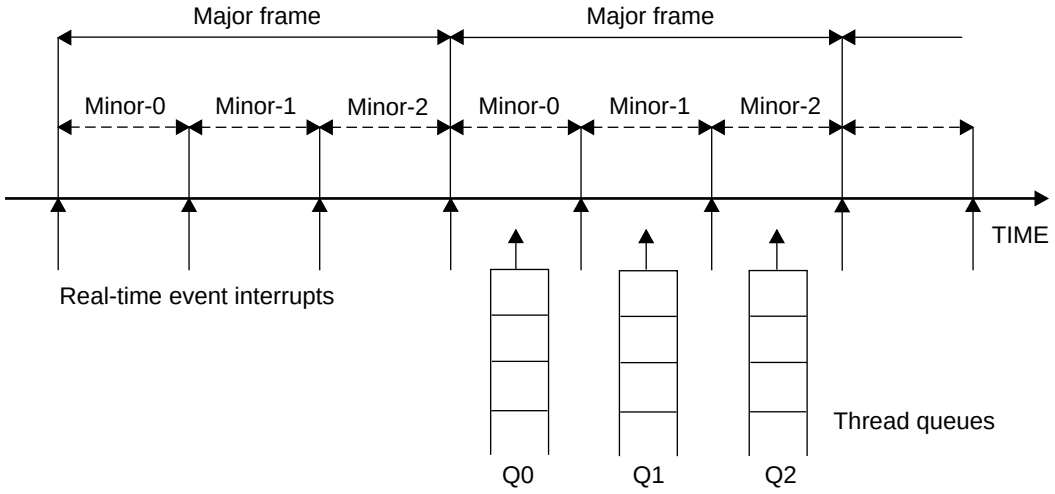

Figure 5-1 Major and Minor Frames

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

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

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

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

- Queue the same thread to run in more than one minor frame. For example, suppose that thread `double` is to run twice as often as thread `solo`. You would queue `double` to Q0 and Q2 in Figure 5-1, and queue `solo` to Q1.
- Queue a thread that takes more than a minor frame to complete its work. If thread `sloth` needs more than one minor interval, you would queue it to Q0, Q1, and Q2, such that it can continue working in all three minor frames until it completes.
- Queue a background thread that is allowed to run only when all others have completed, to use up any remaining time within a minor frame.

All of these options are controlled by scheduling disciplines you specify for each thread as you queue it. For more information, see "Using the Scheduling Disciplines" on page 71.

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

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

- The pthread programming model requires that all threads scheduled by the frame scheduler reside in the same process.
- The `fork()` programming model does not require that the participating threads reside in the same process.

See "Implementing a Single Frame Scheduler" on page 77 for details.

Controller Thread

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

- Its identifier is used to identify its frame scheduler in various functions. The frame scheduler controller thread uses a pthread ID.
- It can receive signals when errors are detected by the frame scheduler (see "Using Signals Under the Frame Scheduler" on page 86).
- It cannot itself be queued to the frame scheduler. It continues to be dispatched by Linux and executes on a CPU other than the one that 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`. Table 5-1 shows some important types that are declared in `/usr/include/frs.h`.

Table 5-1 Frame Scheduler Types

Type	Description
`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.

Type	Description
`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 5-3 on page 60).

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

Table 5-2 Pthread Types

Type	Description
`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 the 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 5-3 summarizes the API library functions in the `/usr/lib/libfrs.a` file.

Table 5-3 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*`);`

Operation	Use	Frame Scheduler API
	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);`
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);`

Operation	Use	Frame Scheduler API
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 67 and the `frs_join`(3) man page.

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

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

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

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

Example 5-2 Alternate Skeleton of an Activity Thread

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

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

Scheduling Within a Minor Frame

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

- Appending a thread at the end of the queue with `frs_pthread_enqueue()` or `frs_enqueue()`
- Inserting a thread after a specific target thread via `frs_pthread_insert()` or `frs_pinsert()`
- Deleting a thread in the queue with `frs_pthread_remove()` or `frs_premove()`

See "Managing Activity Threads" on page 68 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 71.

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

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

Estimating Available Time

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

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

Frame scheduler threads do compete for CPU cycles with I/O interrupts on the same CPU. If you direct I/O interrupts away from the CPU, the only competition for CPU cycles (other than a very few essential interrupts and CPU-specific kernel threads) is the overhead of the frame scheduler itself, and it has been carefully optimized to reduce overhead.

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

Synchronizing Multiple Schedulers

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

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

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

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

Starting a Single Scheduler

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

Each activity thread must be queued to at least one minor frame before it can join the frame scheduler via `frs_join()`. After all threads have called `frs_join()` and the controller has called `frs_start()`, scheduling of worker threads in the first minor frame occurs after the second interrupt arrives.

Note: The first interrupt is used to drive the frame scheduler's internal processing during which time no scheduling occurs.

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 called `frs_join`.

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.

Note: After all threads have called `frs_join()` and the controller has called `frs_start()`, scheduling of worker threads in the first minor frame does not occur until the second interrupt arrives. The first interrupt is used to drive the frame scheduler's internal processing during which time no scheduling occurs.

Pausing Frame Schedulers

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

next minor frame in succession. For more information, see the `frs_stop`(3) and `frs_resume`(3) man pages.

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

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

Managing Activity Threads

The frame scheduler controller 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 functions shown in Table 5-4 on page 68.

Table 5-4 Activity Thread Functions

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

Using these functions, the frame scheduler controller can change the queueing discipline (overrun, underrun, continuable) of a thread by removing it and inserting it

with a new discipline. The frame scheduler controller can suspend a thread by removing it from its queue or can restart a thread by putting it back in its queue.

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

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

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

Selecting a Time Base

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

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

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

This section discusses the following:

- "High-Resolution Timer" on page 70
- "External Interrupts as a Time Base" on page 70

High-Resolution Timer

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

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

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

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

Note: Third-party x86-64 and Altix UV 10 servers do not have a HUB RTC timer. A PCI-RT-Z external interrupt card is supplied by SGI and is required for generation of the Frame Scheduler `cc-timer` interrupts. Each PCI-RT-Z card can generate interrupts at one set frequency, so a PCI-RT-Z card is required for each asynchronous frame scheduler running on a system.

External Interrupts as a Time Base

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

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.

Note: After all threads have called `frs_join()` and the controller has called `frs_start()`, scheduling of worker threads in the first minor frame does not occur until the second interrupt arrives. The first interrupt is used to drive the frame scheduler's internal processing during which time no scheduling occurs.

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

Using the Scheduling Disciplines

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

The disciplines are as follows:

- "Real-Time Discipline" on page 71
- "Underrunable Discipline" on page 72
- "Overrunnable Discipline" on page 72
- "Continuable Discipline" on page 73
- "Background Discipline" on page 73

Real-Time Discipline

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

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

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

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

- Need not start every frame; for instance, threads that sleep on a semaphore until there is work for them to do
- May run longer than one minor frame
- Should run only when time is available, and whose rate of progress is not critical

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

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 73)

To prevent detection of underrun exceptions, specify the underrunable discipline with the real-time discipline. 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 73)

To prevent detection of overrun exceptions, specify an overrunnable discipline with a real-time discipline. 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 64).

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.

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.

Using Multiple Consecutive Minor Frames

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

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

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

The discipline settings should be as follows:

Frame	Description
First	Real-time + overrunnable + continuable The thread must start in this frame (not 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	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 between 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 43.
3. If you are 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 70.
4. Make sure that none of the real-time CPUs is managing the clock. Normally, the responsibility of handling 10–ms scheduler interrupts is given to CPU 0. For more information, see "Avoid the Clock Processor (CPU 0)" on page 43.
5. Restrict and isolate the real-time CPUs, as described in "Restrict, Isolate, and Shield CPUs" on page 44.
6. Load the `frs` kernel module:

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

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

7. If you are using external interrupts as a time base or if you are running the frame scheduler on a third-party x86-64 or Altix UV 10 server, you must load the `ioc4_extint` kernel module:

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

Implementing a Single Frame Scheduler

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

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

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

1. Initialize global resources such as memory-mapped segments, memory arenas, files, asynchronous I/O, semaphores, locks, and other resources.
2. Lock the shared address space segments. (When `fork()` is used, each child process must lock its own address space.)
3. If using pthreads, create a controller thread; otherwise, the initial thread of execution may be used as the controller thread.
   - Create a controller thread using `pthread_create()` and the attribute structure you just set up. See the `pthread_create`(3P) man page for details.
   - Exit the initial thread, because it cannot execute any frame scheduler operations.
4. Create the frame scheduler using `frs_create_master()`, `frs_create_vmaster()`, or `frs_create()`. See the `frs_create`(3) man page.

5. Create the activity threads using one of the following interfaces, depending on the thread model being used:
   - `pthread_create()`
   - `fork()`
6. Queue the activity threads on the target minor frame queues, using `frs_pthread_enqueue()` or `frs_enqueue()`.
7. Optionally, initialize the frame scheduler signal handler to catch frame overrun, underrun, and activity dequeue events (see "Setting Frame Scheduler Signals" on page 87 and "Setting Exception Policies" on page 83). 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 5-3 on page 60.
9. Have the activity threads call `frs_join()`. The frame scheduler begins scheduling processes as soon as all the activity threads have called `frs_join()`.
10. Wait for error signals from the frame scheduler and for the termination of child processes.
11. Use `frs_destroy()` to terminate the frame scheduler.
12. Perform program cleanup as desired.

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

Implementing Synchronized Schedulers

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

This section discusses the following:

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

Synchronized Scheduler Concepts

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

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

- A master controller thread that sets up global data and creates the master frame scheduler
- One slave controller thread for each slave frame scheduler
- Activity threads

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

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

Master Controller Thread

The master controller thread performs these steps:

1. Initializes a global resource. One global resource is the thread ID of the master controller thread.
2. Creates the master frame scheduler using either the `frs_create_master()` or `frs_create_vmaster()` call and stores its handle in a global location.
3. Creates one slave controller thread for each synchronized CPU to be used.

4. Creates the activity threads that will be scheduled by the master frame scheduler and queues them to their assigned minor frames.
5. Sets up signal handlers for signals from the frame scheduler. See "Using Signals Under the Frame Scheduler" on page 86.
6. Uses `frs_start()` to tell the master frame scheduler that its activity threads are all queued and ready to commence scheduling. See Table 5-3 on page 60.

   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 87 and "Setting Exception Policies" on page 83.
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. Uses `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 controller manages 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 81
- "Exception Handling Policies" on page 82
- "Setting Exception Policies" on page 83
- "Querying Counts of Exceptions" on page 84

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 controller 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 controller can then take action, such as terminating the frame scheduler. For more information, see "Setting Frame Scheduler Signals" on page 87.

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 69.

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

When stealing time from the following frame, the frame scheduler returns to the original time base at the end of the following minor frame provided that the threads

queued to that following frame can finish their work in a reduced amount of time. If they do not, the frame scheduler steals time from the next frame.

Dealing With Multiple Exceptions

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

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

Setting Exception Policies

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

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

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

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

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

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

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

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

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

Querying Counts of Exceptions

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

The values returned in a structure of type `frs_overrun_info_t` are the counts of overrun and underrun exceptions incurred by that thread in that minor frame. In order to find the count of all overruns in a given minor frame, you must sum the

counts for all threads queued to that frame. If a thread is queued to more than one minor frame, separate counts are kept for it in each frame.

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

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

```
#define THE_MOST_TIDS 250
int
totalExcepts(frs_t * theFRS, int theMinor)
{
    int numTids = frs_getqueuelen(theFRS, theMinor);
    int j, sum;
    pthread_t allTids[THE_MOST_TIDS];
    if ( (numTids <= 0) || (numTids > THE_MOST_TIDS) )
        return 0; /* invalid minor #, or no threads queued? */

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

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

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

Using Signals Under the Frame Scheduler

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

- The frame scheduler has been terminated
- An overrun or underrun has been detected
- A thread has been dequeued

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

- "Handling Signals in the Frame Scheduler Controller" on page 86
- "Handling Signals in an Activity Thread" on page 87
- "Setting Frame Scheduler Signals" on page 87
- "Handling a Sequence Error" on page 88

Handling Signals in the Frame Scheduler Controller

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

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

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

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

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

Hence `frs_destroy()` is equivalent to termination for the frame scheduler controller.

Handling Signals in an Activity Thread

A frame scheduler can send a signal to an activity thread when the thread is removed from any queue using `frs_pthread_remove()` or `frs_premove()`. The scheduler can also send a signal to an activity thread when it is removed from the last or only minor frame to which it was queued (at which time it is scheduled only by Linux). For more information, see "Managing Activity Threads" on page 68.

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 5-5.

Table 5-5 Signals Passed in `frs_signal_info_t`

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

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

Example 5-6 Function to Set Frame Scheduler Signals

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

Handling a Sequence Error

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

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

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

Using Timers with the Frame Scheduler

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

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

Example 5-7 Minimal Activity Process as a Timer

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

Disk I/O Optimization

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

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

Memory-Mapped I/O

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

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

Asynchronous I/O

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

This section discusses the following:

- "Conventional Synchronous I/O" on page 92
- "Asynchronous I/O Basics" on page 92

Conventional Synchronous I/O

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

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

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

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

Asynchronous I/O Basics

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

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

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

PCI Devices

To perform programmed I/O on PCI devices on an Altix UV system, do the following to determine the resource filename (`resource`*N*) and create an appropriate program to open the file and memory-map it:

1. Examine the output of the `lspci`(8) command to determine which device you want to map:

   a. Record the domain, bus, slot, and function for the device (this information will help you locate the appropriate resource address file).

      For example:

```
# lspci
...
0000:00:1e.0 PCI bridge: Intel Corporation 82801 PCI Bridge (rev 90)
0000:00:1f.0 ISA bridge: Intel Corporation 82801JIR (ICH10R) LPC Interface Controller
0000:00:1f.3 SMBus: Intel Corporation 82801JI (ICH10 Family) SMBus Controller
0000:01:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network Connection (rev 01)
0000:01:00.1 Ethernet controller: Intel Corporation 82576 Gigabit Network Connection (rev 01)
0000:04:00.0 SCSI storage controller: LSI Logic / Symbios Logic SAS1064ET PCI-Express Fusion-MPT SAS (rev 08)
0000:05:00.0 VGA compatible controller: Matrox Graphics, Inc. MGA G200e [Pilot] ServerEngines (SEP1) (rev 02)
...
```

      The first field gives the information that is required to map the PCI registers into memory. The format is:

      Domain:*Bus*:*Slot*.*Function*

      In the above example, the highlighted output of `0000:01:00.1` for the Intel Corporation 82576 Gigabit Network card equates to domain 0, bus 1, slot 0, and function 1.

   b. Determine the `resource`*N* numbers from the `Region` numbers in the `lspci -vv` output. The `Region` value corresponds directly to each `resource`*N* value.

      In the following example, the `Region` *N* output (highlighted) indicates that there are four `resource`*N* values (`resource0`, `resource1`, `resource2` and `resource3`):

```
# lspci -n -s 0000:01:00.1 -vv
0000:01:00.1 0200: 8086:10c9 (rev 01)
        Subsystem: 10a9:8028
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx+
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- SERR- <PERR- INTx-
        Latency: 0, Cache Line Size: 64 bytes
        Interrupt: pin B routed to IRQ 40
        Region 0: Memory at b2140000 (32-bit, non-prefetchable) [size=128K]
        Region 1: Memory at b2120000 (32-bit, non-prefetchable) [size=128K]
        Region 2: I/O ports at 2000 [size=32]
        Region 3: Memory at b2240000 (32-bit, non-prefetchable) [size=16K]
        Expansion ROM at b2100000 [disabled] [size=128K]
        Capabilities: [40] Power Management version 3
                Flags: PMEClk- DSI+ D1- D2- AuxCurrent=0mA PME(D0+,D1-,D2-,D3hot+,D3cold+)
                Status: D0 PME-Enable- DSel=0 DScale=1 PME-
...
...
        Kernel driver in use: igb
        Kernel modules: igb
```

A device can have both 32-bit and 64-bit base address registers (BARs). If a BAR is mapping a 64-bit address space, then two 32-bit BARs are used to map that 64-bit `Region`. As a result, `Region` numbers may not be consecutive. For example, in the following `lspci` output, there are three `Region` values (`Region 0`, `Region 1` and `Region 3`):

```
# lspci -n -s 0000:04:00.0 -vv
0000:04:00.0 0100: 1000:0056 (rev 08)
        Subsystem: 1000:1000
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx-
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Latency: 0
        Interrupt: pin A routed to IRQ 24
        Region 0: I/O ports at 1000 [size=256]
        Region 1: Memory at b2010000 (64-bit, non-prefetchable) [size=16K]
        Region 3: Memory at b2000000 (64-bit, non-prefetchable) [size=64K]
        Expansion ROM at b1c00000 [disabled] [size=4M]
        Capabilities: [50] Power Management version 2
                Flags: PMEClk- DSI- D1+ D2+ AuxCurrent=0mA PME(D0-,D1-,D2-,D3hot-,D3cold-)
                Status: D0 PME-Enable- DSel=0 DScale=0 PME-
...
...
```

```
        Kernel driver in use: mptsas
        Kernel modules: mptsas
```

There is no `Region 2` because the card's second BAR is mapping a 64-bit region and thus uses two 32-bit BARs to do so. In this example, there would be three corresponding resource numbers (`resource0`, `resource1`, and `resource3`) that would be used to memory-map the PCI registers.

Note: Only memory base-address registers (not I/O base-address registers) can be memory mapped. The base address must be page aligned.

---

2. Based on the information in step 1, determine the resource address file that you want to open:

   `/sys/bus/pci/devices/`*domain*`:`*bus*`:`*slot*`.`*function*`/resource`*N*

   For the Intel example above, the resource address files are:

   ```
   /sys/bus/pci/devices/0000:01:00.1/resource0
   /sys/bus/pci/devices/0000:01:00.1/resource1
   /sys/bus/pci/devices/0000:01:00.1/resource2
   /sys/bus/pci/devices/0000:01:00.1/resource3
   ```

   In the case of the LSI Logic® card example showing 64-bit `Region` values:

   ```
   /sys/bus/pci/devices/0000:04:00.0/resource0
   /sys/bus/pci/devices/0000:04:00.0/resource1
   /sys/bus/pci/devices/0000:04:00.0/resource3
   ```

3. Create a program that opens the appropriate resource file for the domain, bus, slot, function, and resource in which you are interested. For example, the C program for the Intel card could include the following lines:

```
sprintf(path, "/sys/bus/pci/devices/%04x:%02x:%02x.%x/%s",
       (unsigned)domain, (unsigned)bus, (unsigned)slot, (unsigned)function,
       "resource0");
if ((fd = open(path, O_RDWR)) == -1) {
  perror("Couldn't open resource file");
  exit(1);
}
```

4. Add a line to the program that will memory-map the opened file from offset 0. For example, in C:

```
ptr = mmap( NULL, getpagesize(), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

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. A programming error in the driver will result in nothing more serious than the termination of a process rather than crashing the entire system, and the developer need not know anything about interfacing a driver into the kernel.

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

A ULI is essentially an *interrupt service routine (ISR)* that resides 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.

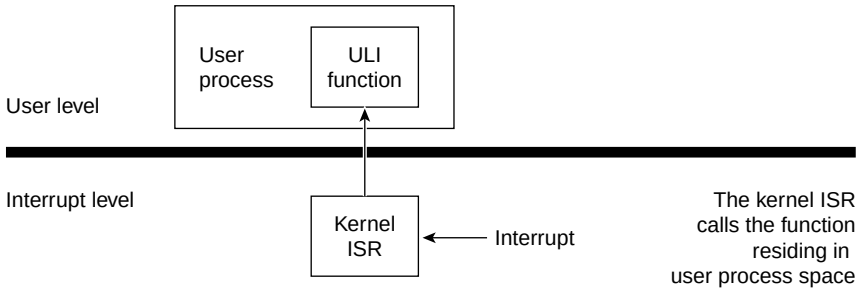

Figure 8-1 ULI Functional Overview

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 98
- "Setting Up ULI" on page 101

Overview of ULI

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

Table 8-1 Common Arguments for Registration Functions

Function	Description
`func`	Points to the function that will handle the interrupt.
`ULI_register_irq`	Requests that an interrupt be handled as a ULI. Once a registration function has been called, the handler function may be called asynchronously any time the associated hardware sees fit to generate an interrupt. Any state needed by the handler function must have been initialized before ULI registration. The process will continue to receive the ULI until it exits or the ULI is destroyed (see `ULI_destroy` below), at which time the system reverts to handling the interrupt in the kernel. The CPU that executes the ULI handler is the CPU that would execute the equivalent kernel-based interrupt handler if the ULI were not registered (that is, the CPU to which the device sends the interrupt).
`ULI_destroy`	Destroys a ULI. When this function returns, the identifier will no longer be valid for use with any ULI function and the handler function used with it will no longer be called.

Function	Description
`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 99
- "Planning for Concurrency: Declaring Global Variables" on page 101
- "Using Multiple Devices" on page 101

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.

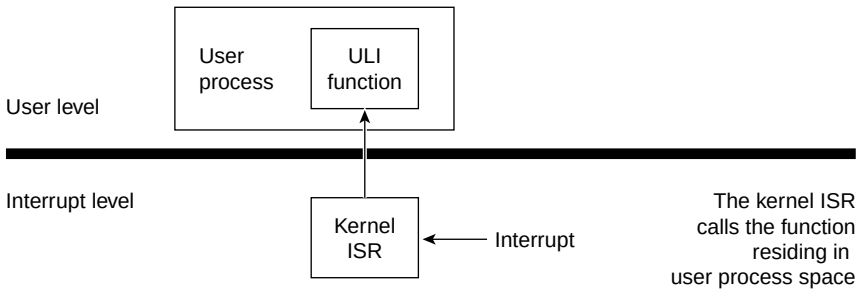

Figure 8-2 ULI Handler Functions

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.

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 104.

Setting Up ULI

A program initializes for ULI in the following major steps:

1. Load the `uli` kernel module:

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

2. For a PCI, map the device addresses into process memory.
3. Lock the program address space in memory.
4. Initialize any data structures used by the interrupt handler.
5. Register the interrupt handler.
6. Interact with the device and the interrupt handler.

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

This section discusses the following:

- "Opening the Device Special File" on page 102
- "Locking the Program Address Space" on page 102
- "Registering the Interrupt Handler" on page 103
- "Registering a Per-IRQ Handler" on page 103
- "Interacting With the Handler" on page 104
- "Achieving Mutual Exclusion" on page 105

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 104. 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, such as 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.

REACT System Configuration

This chapter explains how to configure real-time CPUs that are restricted from running scheduled processes and isolated from load-balancing considerations. It discusses the following:

- "`react` Command-Line Summary" on page 107
- "Initially Configuring REACT" on page 110
- "Changing the Configuration" on page 111
- "Disabling REACT" on page 111
- "Reenabling REACT" on page 112
- "Specifying Permissions" on page 112
- "Showing the Configuration" on page 114
- "Getting Trace Information" on page 115
- "Running a Process on a Real-Time CPU" on page 117

For information about creating an external interrupt character special device file, see "Opening the Device Special File" on page 102. For information about potential problems, see Chapter 13, "Troubleshooting" on page 151.

`react` Command-Line Summary

To configure REACT, you will use the `react`(8) command. Configurable items include:

- The configured real-time CPUs (the `rtcpu` devices)
- The bootcpuset (`/boot`)
- Interrupts, which can be redirected
- Permissions

REACT stores most configuration information supplied via the `react` command in the `/etc/react.conf` file; however, permissions are stored in the `/etc/sysconfig/sgi-react.conf` file.

The `react` command has the following options:

```
/sbin/react -d [-v]
/sbin/react -e [-v]
/sbin/react -h
/sbin/react -i irqlist|RR [-v]
/sbin/react -p group:perm[:action]
/sbin/react -r cpulist [-i irqlist|RR] [-v]
/sbin/react -s
```

`-d` Disables REACT.

`-e` Enables the configuration stored in the `/etc/react.conf` file. For more information, see "Initially Configuring REACT" on page 110.

`-h` Displays the usage statement (the default for `react` without any options).

`-i` *irqlist*`|RR` Specifies the interrupt requests (IRQs) to be redirected. The specification is either:

- A comma-separated list of IRQs and the CPUs to which they should be directed, in the format:

  IRQ`:`*CPU*`,`*IRQ*`:`*CPU*`,`*IRQ*`:`*CPU* ...

- `RR` for round-robin dispersal among CPUs in the bootcpuset (the default).

To minimize latency of real-time interrupts, it is often necessary to direct some IRQs to specific real-time processors and to direct other interrupts away from specific real-time processors. You should only redirect IRQs if you must move them away from CPUs that must be real-time. However, redirected IRQs often have higher latency, so it is preferable to select CPUs for real-time in such a way as to not require interrupt redirection.

By default (if you do not enter `-i`), REACT assumes that the IRQs should be moved off of the real-time CPUs. REACT causes IRQs that can be moved to be evenly dispersed among CPUs in the bootcpuset in a round-robin (`-i RR`) fashion.

`-p` *group*`:`*perm*`[:`*action*`]`

Specifies group ownership, permissions, and write action, where:

- *group* is one of: the numerical group ID, the mask `-1`, or the mask `0`
- *permission* is one of: the octal permission setting (such as `0755`), the mask `-1`, or the mask `0`
- *action* is the mask `1` (optional)

And where the above masks are defined to mean:

- `-1` = Leave the parameter unchanged
- `0` = Read information from `/etc/sysconfig/sgi-react.conf`
- `1` = Write these changes to `/etc/sysconfig/sgi-react.conf`

`-r` *cpulist*

Specifies the real-time CPUs. *cpulist* takes one of the following formats:

- A comma-separated list of CPUs (you cannot specify CPU `0`):

  cpu,cpu,...

- A range of CPUs (you cannot specify CPU `0` or a descending range):

  cpu-cpu

- A mixture of the above:

  cpu,...cpu-cpu,cpu,...

	If you do not specify `-r` *cpulist*, no real-time CPUs are identified.
`-s`	Shows the REACT configuration. See "Showing the Configuration" on page 114.
`-v`	Specifies verbose mode, which sends tracing messages to the console.

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

Initially Configuring REACT

To initially configure REACT, do the following:

1. Specify the real-time CPUs and optionally any interrupt requests (IRQs) to be redirected:

   ```
   [root@linux root]# react -r cpulist [-i irqlist]
   ```

   For example, to restrict CPUs 8-32 and (by default) redirect IRQs away from CPUs 8-32:

   ```
   [root@linux root]# react -r 8-32
   ```

   In another example, to restrict CPUs 2, 3, 4, 5, 6, and 7, to redirect IRQ 59 to CPU 2, and to redirect IRQ 66 to CPU 5:

   ```
   [root@linux root]# react -r 2-7 -i 59:2,66:5
   ```

2. Reboot the system (`react` will add the required kernel command-line options).

   When the system comes back up, REACT is automatically enabled by the `/etc/init.d/sgi_react` script (which runs the `react -e` command).

The enable (`-e`) option does the following:

- Creates a container cpuset named `rtcpus` and cpusets (labeled `rtcpu`*N*) for each CPU that is not part of the bootcpuset (such as `/rtcpus/rtcpu1` for CPU1). 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 `react -r` in step 1 and stored in `/etc/react.conf`.

- Configures the cpuset's memory nodes by setting the values in the following files:
  - `/dev/cpuset/rtcpus/rtcpu`*N*`/mems`
  - `/dev/cpuset/boot/mems`
- Redirects interrupts if specified with the `-i` option in step 1. The proper hexadecimal mask values are echoed to the file `/proc/irq/interrupt/smp_affinity`.

Changing the Configuration

After the system is rebooted with the real-time configuration and REACT is automatically enabled, you can make changes to the real-time and bootcpusets dynamically without additional reboots.

For example, to change the list of real-time CPUs to CPU 2 and CPU 4 and return to the default round-robin handling of IRQs, enter the following:

```
[root@linux root]# react -r 2,4 -i RR
```

To change the IRQ configuration without altering the real-time CPUs, use just the `-i` option. For example, to redirect IRQ 4340 to CPU 3 and to redirect IRQ 66 to CPU 5:

```
[root@linux root]# react -i 4340:3,66:5
```

Note: To temporarily change the running REACT system, you can call `libreact` from a user program to add or remove real-time CPUs. However, these changes will not be stored in `/etc/react.conf`. For more information, see the `libreact`(3) man page.

Disabling REACT

To disable REACT and return the system to normal, do the following:

1. Stop the real-time processes.
2. Enter the disable option:

   ```
   [root@linux root]# react -d
   ```

The disable option does the following:

- Removes the `rtcpu`*N* cpusets and adjusts `/boot` to behave like `/cpuset` on a system without REACT.
- Starts the IRQ balancer, which will move any changed IRQs to CPUs based on the IRQ balancer's policies. For more information, see the `irqbalance`(1) man page.

Reenabling REACT

To reenable a previously configured REACT system that has been disabled and use the configuration that is stored in `/etc/react.conf`, enter the following:

```
[root@linux root]# react -e
```

If you enter `react -e` on a currently enabled REACT system whose configuration has been modified by a user program that calls `libreact`, `react` enables the configuration stored in the `/etc/react.conf` file.

Specifying Permissions

The cpusets, devices, and control files associated with REACT are normally accessible only by the `root` user. The `-p` option lets you specify a group of users that have access to the following REACT features:

- Cpusets created by the `react` command
- User-level interrupts (ULI)
- The frame scheduler
- External interrupts
- User capabilities (the `cpu_sysrt_set_allowed_caps` and `cpu_sysrt_set_caps` routines)

This option generates the `/etc/udev/rules.d/99-sgi-react.rules` file and a new `/etc/sysconfig/sgi-react.conf` configuration file, which initially holds the group ID and permissions. It changes the group ownership and file mode permissions for REACT `/dev`, `/sys/class/extint`, and `/dev/cpuset` files, both immediately and across reboots.

After you use the `-p` option, the specified users can run REACT applications without having the ability to overwrite any file on the system. (That is, the specified users do not have `CAP_DAC_OVERRIDE` authority.)

Note: The specified users will not have access to native system calls that require specific capabilities, such as `sched_setscheduler()`. To directly use those system calls, a user must have the required process capabilities set via the `cpu_sysrt_set_allowed_caps` and `cpu_sysrt_set_caps` routines

---

To use the `-p`option, you specify an explicit group ID, the explicit permissions to set, and (optionally) the action of writing the changes to the `/etc/sysconfig/sgi-react.conf` file (using the mask `1`):

`-p` *group*`:`*permission*`[:1]`

In place of the explicit group ID and permissions, you can also specify a mask that indicates whether to:

- `-1` = Leave the parameter unchanged
- `0` = Read information from `/etc/sysconfig/sgi-react.conf`

For details, see "`react` Command-Line Summary" on page 107.

For example, suppose that the `/etc/group` file has an entry for a group named `usersA` that has a numerical ID of `100` and another group named `usersB` that has a numerical group ID of `222`:

```
[root@linux root]# grep users /etc/group
usersA:x:100:
usersB:x:222:
```

No changes have yet been made to the `/etc/sysconfig/sgi-react.conf` file:

```
[root@linux root]# ls -al /dev/cpuset/rtcpus/tasks
-rwxrwxrwx 1 root root 0 2010-10-06 07:07 /dev/cpuset/rtcpus/tasks
[root@linux root]# cat /etc/sysconfig/sgi-react.conf
[root@linux root]#
```

To allow the group `usersA` to run REACT with execute and write permission (`0755`), and then save those changes to `/etc/sysconfig/sgi-react.conf`, enter the following and show the results:

```
[root@linux root]# react -p 100:0755:1
[root@linux root]# ls -al /dev/cpuset/rtcpus/tasks
-rwxr-xr-x 1 root usersA 0 2010-10-06 07:07 /dev/cpuset/rtcpus/tasks
[root@linux root]# cat /etc/sysconfig/sgi-react.conf
group 100
mode 0755
```

To allow the group `usersB` to run REACT with execute and keep the permission that currently exists in `/etc/sysconfig/sgi-react.conf` (`0755`) for this session (but not save the change), enter the following and show the results:

```
[root@linux root]# react -p 222:0:1
[root@linux root]# ls -al /dev/cpuset/rtcpus/tasks
-rwxr-xr-x 1 root usersB 0 2010-10-06 07:07 /dev/cpuset/rtcpus/tasks
[root@linux root]# cat /etc/sysconfig/sgi-react.conf
group 100
mode 0755
```

To allow the group listed in `/etc/sysconfig/sgi-react.conf` (`usersA` with a group ID of `100`) to read permission (`0644`) for this session (but not save the change), enter the following and show the results:

```
[root@linux root]# react -p 1:0644
[root@linux root]# ls -al /dev/cpuset/rtcpus/tasks
-rwxr--r-- 1 root usersB 0 2010-10-06 07:07 /dev/cpuset/rtcpus/tasks
[root@linux root]# cat /etc/sysconfig/sgi-react.conf
group 222
mode 0755
```

Showing the Configuration

The `-s` option displays the configuration that is running and the configuration that is stored in `/etc/react.conf`.

Note: These may be different if you have called `libreact` from a user program to add or remove real-time CPUs.

For example:

```
[root@linux root]# react -s
++++ REACT is ENABLED ++++
```

```
Live configuration:
===========================
bootcpuset cpus:          0-7,32-39

real-time cpus:           8-31,40-63

Configuration in /etc/react.conf:
=========================================
bootcpuset cpus:          0-7 32-39

real-time cpus:           8-31 40-63

IRQ configuration: 21:0 23:7 54:4 63:45
```

Getting Trace Information

If you add `-v` to the command line with `-d`, `-e`, `-r`, or `-i`, the `react` command prints a trace of its actions to the console. The verbose output will detail the steps taken by `react` and is useful in understanding its behavior and analyzing problems. (The amount of output will vary greatly depending on the number of CPUs and the number of IRQs.)

For example (line breaks shown here for readability:

```
[root@linux root]# react -ve
Default label = ’0’

kernel kernel /boot/vmlinuz-2.6.32-70.el6.x86_64 ro root=LABEL=uv41-sysR12
...
Current Kernel Command line:
ro root=LABEL=uv41-sysR12 rd_NO_LUKS rd_NO_LVM rd_NO_MD rd_NO_DM
...
rtcpus 8-31,40-63
bootcpus 1-7,32-39
Acquiring Lock...
Lock Acquired
cpuset_delete: /rtcpuN does not exist
...
cpuset /rtcpus  cpu 8  mem 2
...
modified cpu list 8-31,40-63
modified mem list 2-7
cpuset: modify /rtcpus
DUP cpu 1
...
cpuset: modify /boot
Releasing Lock
Lock Released
Acquiring Lock...
Lock Acquired
DUP cpu 8
...
cpuset /boot  cpu 0  mem 0
...
cpuset /rtcpus/rtcpu63  cpu 63  mem 7
modified cpu list 63
modified mem list 7
cpuset: modify /rtcpus/rtcpu63
Releasing Lock
Lock Released

++++ REACT is ENABLED ++++
```

```
Live configuration:
===========================
bootcpuset cpus:          0-7,32-39

real-time cpus:           8-31,40-63

Configuration in /etc/react.conf:
=========================================
bootcpuset cpus:          0-7 32-39

real-time cpus:           8-31 40-63

IRQ configuration: 21:0 23:7 54:4 63:45

Total Nodes = 8
CPUs on node 0 - 0-3,32-35
...
IRQ 0 is on node 0, bootcpu on same node == 0
...
IRQ 94 is on node 3, No bootcpu available on node, using bootcpu == 32
...
**** Manually config'd IRQs ****
IRQ 21 cpu 0
IRQ 23 cpu 7
IRQ 54 cpu 4
IRQ 63 cpu 45
```

Running a Process on a Real-Time CPU

To run a process on a real-time CPU, you must invoke or attach it to a real-time cpuset (that is, a cpuset containing a CPU that does not exist in the bootcpuset, such as the `/dev/cpuset/rtcpus/rtcpu`*N* cpusets created above). For example:

```
[root@linux root]# cpuset --invoke /rtcpus/rtcpu4 -I ./foo
```

or:

```
[root@linux root]# echo $$ | cpuset -a /rtcpus
[root@linux root]# dplace -c 1 ./foo
```

Note: The `dplace` command example will attach the process to the second real-time CPU, not the second CPU on the system.

To attach an existing process to a real-time CPU, you can use `cpuset --attach`. For example, to attach your current process to CPU 2:

```
[root@linux root]# echo $$ | cpuset --attach /rtcpus/rtcpu2
```

For more information, see the `cpuset`(1), `dplace`(1), `libreact`(3), and `libcpuset`(3) man pages.

Using the REACT Library

You can use the REACT C application programming interface (API) to change the configuration of real-time CPUs from program control without affecting the boot-up configuration for real-time processing.

The system must have been booted with REACT configured as described in Chapter 9, "REACT System Configuration" on page 107. The real-time CPUs created with the C API have local memory nodes assigned to them by default. The API requires that a `/boot` cpuset is present.

Note: IRQ redirection is not supported through the API.

This chapter discusses the following:

- "REACT Library Routines" on page 119
- "Accessing REACT Library Routines" on page 128
- "Installing the `pam_capability` Package" on page 128
- "Example Code Using the REACT Library Routines" on page 129

REACT Library Routines

This section discusses the following REACT library API routines:

- "`cpu_shield`" on page 120
- "`cpu_sysrt_add`" on page 121
- "`cpu_sysrt_delete`" on page 122
- "`cpu_sysrt_info`" on page 122
- "`cpu_sysrt_irq`" on page 123
- "`cpu_sysrt_perm`" on page 124
- "`cpu_sysrt_runon`" on page 126

- "cpu_sysrt_set_allowed_caps" on page 126
- "cpu_sysrt_set_caps" on page 127

cpu_shield

```
int cpu_shield(int op, int cpu)
```

The cpu_shield routine controls timer interrupts on select CPUs. The cpu_shield routine requires the following arguments:

Argument	Description
op	Starts (SHIELD_START_INTR) or stops (SHIELD_STOP_INTR) timer interrupts
cpu	Specifies the CPU on which to stop or start timer interrupts

Note: Timer interrupts cannot be stopped on CPU 0 because it performs time-keeping tasks.

To avoid system instability, you should only use this routine on isolated CPUs that are not being used by the system in general.

To use cpu_shield, you must install and load the sgi-shield kernel module. To load the module, run the following:

```
# modprobe sgi-shield
```

Return values:

Value	Description
0	Success
-1	Error, setting errno

Note: Because cpu_shield makes use of a device file, errors associated with open(2) also apply. An error of this type likely indicates that the sgi-shield module is not loaded.

cpu_sysrt_add

```
int  cpu_sysrt_add(struct bitmask *cpus, unsigned long rt_flags)
```

The cpu_sysrt_add routine creates real-time CPUs in the given bitmask CPUs. The bitmask can contain one or more CPUs and memory nodes for the given flag. Access to the cpusets must be mutually exclusive during the modification of the real-time CPUs. The cpu_sysrt_add routine can either wait for the lock to become free or can return immediately with errno set to EWOULDBLOCK.

Real-time flags:

Flag	Description
RT_WAIT	Wait until the lock is free
RT_NO_WAIT	Do not wait

Return values:

Value	Description
0	Success
-1	Error, setting errno

cpu_sysrt_delete

```
int  cpu_sysrt_delete(struct bitmask *cpus, unsigned long rt_flags)
```

The cpu_sysrt_delete routine deletes the real-time CPUs in the given bitmask CPUs. The bitmask can contain one or more CPUs and memory nodes for the given flag. Access to the cpusets must be mutually exclusive during the modification of the real-time CPUs. The cpu_sysrt_delete routine can either wait for the lock to become free or can return immediately with errno set to EWOULDBLOCK.

Real-time flags:

Flag	Description
RT_WAIT	Wait until the lock is free
RT_NO_WAIT	Do not wait

Return values:

Value	Description
0	Success
-1	Error, setting errno

cpu_sysrt_info

```
int  cpu_sysrt_info(struct bitmask &b_mask, unsigned long query_flag)
```

The cpu_sysrt_info routine writes the bitmask to b_mask. The bitmask will contain one or more corresponding CPU or memory nodes for the given flag.

As its parameter, cpu_sysrt_info takes an allocated, NULL bitmask structure.

Query flags:

Flag	Description
BOOTCPUS	The CPUs in the /boot cpuset
BOOTMEMS	The memory nodes assigned to the /boot cpuset
RTCPUS	The real-time CPUs currently configured on the system

`RTMEMS`	The real-time memory nodes associated with the real-time CPUs

Return values:

Value	Description
`0`	Success
`-1`	Error, setting `errno`

Note: This routine can fail if an invalid query flag (`EINVAL`) is set. If any of the cpuset query routines fail, an error is printed to `stderr` along with `errno` being set.

cpu_sysrt_irq

```
int cpu_sysrt_irq(char *user_irq_input, unsigned long rt_flags)
```

The `cpu_sysrt_irq` routine changes the CPU affinity of the given IRQs.

Input for `user_irq_input` is in string format, one of the following:

- A comma-separated list of paired IRQs and CPUs:

  IRQ:*CPU*,*IRQ*:*CPU*,*IRQ*:*CPU*, ...

- Round-robin (default):

  `RR`

Note: By default, REACT assumes that the IRQs should be moved off of the real-time CPUs. REACT causes IRQs that can be moved to be evenly dispersed among CPUs in the bootcpuset in a round-robin fashion.

Return values:

Value	Description
`0`	Success
`-1`	Error, setting `errno`

cpu_sysrt_perm

```
cpu_sysrt_perm (gid_t group, mode_t mode, unsigned long
rt_flags)
```

The `cpu_sysrt_perm` routine changes permissions so that REACT can be run by non-`root` users, based on customer-specified group ownership and file-mode permission parameters.

Input:

Value	Description
`gid_t group`	Specifies one of the following: • The group number allowed • `PARAMETER_UNCHANGED`, which leaves the group as is • `READ_FROM_FILE`, which uses the group that was written to the `sgi-react.conf` file
`mode_t mode`	Specifies one of the following: • The file permissions allowed • `PARAMETER_UNCHANGED`, which leaves the mode as is • `READ_FROM_FILE`, which uses the mode that was written to the `sgi-react.conf` file

`unsigned long rt_flag`

Specifies the bitwise-OR of one or more of the following:

- Wait action, one of:

`RT_WAIT`	Waits for the lock to become free
`RT_NO_WAIT`	Returns immediately with `errno` set to `EWOULDBLOCK`

- Write action:

`WRITE_TO_FILE`	Writes the changes to the `sgi-react.conf` file.

Return values:

Value	Description
`0`	Success
`-1`	Error, setting `errno`

Note: The `chmod` and `chown` commands do not exit on error, so the `errno` will not be set on those errors but an error message will be displayed.

cpu_sysrt_runon

```
int cpu_sysrt_runon(int cpu)
```

The `cpu_sysrt_runon` routine assigns a process to run only on the processor number given by `cpu`. `cpu` is assumed to be real-time, configured via `cpu_sysrt_add` or `react`(8), or `errno` will be set to `EINVAL`.

Return values:

Value	Description
`0`	Success
`-1`	Error, setting `errno`

cpu_sysrt_set_allowed_caps

Note: You must run `cpu_sysrt_perm` or `react -p` before using this routine.

```
int cpu_sysrt_set_allowed_caps(int flags)
```

The `cpu_sysrt_set_allowed_caps` routine lets a thread raise all permitted and/or effective capabilities currently allowed for that thread. *Permitted capabilities currently allowed* are all of those in the processes' current inheritable set. *Effective capabilities currently allowed* are all of those in the processes' current permitted set. (The current inheritable set must have been previously set by a security module such as `pam_capability`. See "Installing the `pam_capability` Package" on page 128.)

`int` flags may be the bitwise-OR of one or more of the following:

- `USERCAPS_SET_PERMITTED` sets the permitted capabilities that are currently allowed
- `USERCAPS_SET_EFFECTIVE` sets the effective capabilities that are currently allowed

Caution: SGI assumes no responsibility for any security issues that may result from either the proper use or misuse of this API call.

To use `cpu_sysrt_set_allowed_caps`, you must install and load the `usercaps` kernel module. To load the module, run the following:

```
# modprobe usercaps
```

Return values:

Value	Description
`0`	Success
`-1`	Error, setting `errno`

cpu_sysrt_set_caps

```
int cpu_sysrt_set_caps(unsigned *cap_p, int npcap, unsigned *cap_e, int necap
```

Note: You must run `cpu_sysrt_perm` or `react -p` before using this routine.

The `cpu_sysrt_set_caps` routine lets a thread raise a specified permitted and/or effective capability set. Unlike other Linux commands for raising capabilities, the permitted capability set is validated based exclusively on the inheritable capability set. (The current inheritable set must have been previously set by a security module such as `pam_capability`. See "Installing the `pam_capability` Package" on page 128.)

Flag	Description
`unsigned *cap_p`	An array of capability values to be added to the processes' current permitted set
`int npcap`	Number of capability values in the cap_p array
`unsigned *cap_e`	An array of capability values to be added to the processes' current effective set
`int necap`	Number of capability values in the cap_e array

Caution: SGI assumes no responsibility for any security issues that may result from either the proper use or misuse of this API call.

To use `cpu_sysrt_set_caps`, you must install and load the `usercaps` kernel module. To load the module, run the following:

```
# modprobe usercaps
```

Return values:

Value	Description
`0`	Success
`-1`	Error, setting `errno`

Accessing REACT Library Routines

The following inclusion and linkage provides access to the REACT library from `C` code:

```
#include <bitmask.h>
#include <react.h>
/* link with -lreact */
```

Installing the `pam_capability` Package

Prior to using the `cpu_sysrt_set_allowed_caps` and `cpu_sysrt_set_caps` routines, you must install a software package that will enable the user's inheritable capability set at login time. To accomplish this with the `pam_capability` package, do the following:

1. Install the `pam_capability` package supplied by your Linux distribution vendor. For example, using the `rpm`(8) command:

   ```
   # rpm -ivh pam_capability-0.22-7.9.x86_64.rpm
   ```

2. Edit the `/etc/security/capability.conf` file to configure `pam_capability` with the desired user and group permissions.

For example, to enable `cap_sys_nice` and `cap_ipc_lock` for the `rtgroup` group, the `/etc/security/capability.conf` file should contain the following:

```
role    fbscheduser     cap_sys_nice cap_ipc_lock

group   rtgroup         fbscheduser
```

To enable the same capabilities for an individual user (`rtuser`):

```
role    fbscheduser     cap_sys_nice cap_ipc_lock

user    rtuser          fbscheduser
```

For more information, see the `capability.conf`(5) man page.

3. Enable `pam_capability` by adding a session line to each desired login service.

   For example, to enable `pam_capability` for `ssh` logins, `/etc/pam.d/sshd` should contain the following session line:

   ```
   session  required       /lib64/security/pam_capability.so
   ```

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

Example Code Using the REACT Library Routines

Following is example code using the REACT library.

```
/* Add, Delete and RunOn*/

 int new_rtcpu = 3;

 if ((cpus = bitmask_alloc(cpuset_cpus_nbits())) == NULL) {
        perror("cpuset: bitmask alloc failed:");
        exit (1);
 }

 bitmask_setbit(cpus, new_rtcpu);

 if (cpu_sysrt_add(cpus, RT_WAIT)){
        perror("cpu_sysrt_add failed:");
 }
```

```
 if (cpu_sysrt_runon(new_rtcpu)) {
        perror("cpu_sysrt_runon");
        exit(1);
 }

 ..
 /* RT CODE */
 ..


 if (cpu_sysrt_delete(cpus,RT_WAIT)){
        perror("cpu_sysrt_del failed:");
 }

  bitmask_free(cpus);

=============================================================================
/* IRQ */

  char user_irq_input_buf[45] = "86:2,89:1,87:3,18:4,88:6";

  if (cpu_sysrt_irq(user_irq_input_buf, RT_WAIT)){
        perror("cpu_sysrt_irq failed");
  }


=============================================================================
/* Info */

  struct bitmask *i_cpus = NULL;

  if ((i_cpus = bitmask_alloc(cpuset_cpus_nbits())) == NULL) {
        perror("cpuset: bitmask alloc failed:");
        exit (1);
  }

  if (cpu_sysrt_info(&i_cpus, QRTCPUS)){
        perror("cpu_sysrt_info failed");
  }
```

```
  ..
  /* See libbitmask for use of bitmask structure */
  ..

  bitmask_free(i_cpus);

=========================================================================
/* Permissions */

  gid_t           group_id = 117; /* group id or PARAMETER_UNCHANGED, READ_FROM_FILE */
  mode_t          mode     = 01644; /* permissions or PARAMETER_UNCHANGED, READ_FROM_FILE*/
  unsigned long   mask     = 0;

  mask |= RT_NO_WAIT;    /* or RT_WAIT  */
  mask |= WRITE_TO_FILE;


  if (cpu_sysrt_perm(group_id, mode,  mask) < 0){
        perror("Permissions failed");
  }



=========================================================================
/* Set specific capabilities */

        unsigned caps[3];
        unsigned capse[3];

        caps[0] = CAP_SYS_NICE;
        caps[1] = CAP_CHOWN;
        caps[2] = CAP_DAC_OVERRIDE;
        capse[0] = CAP_SYS_NICE;
        capse[1] = CAP_CHOWN;
        capse[2] = CAP_DAC_OVERRIDE;

        if (cpu_sysrt_set_caps(caps, 3, capse, 3)) {
                perror("cpu_sysrt_set_caps P");
                return -1;
        }
```

NOTE: The following 2 calls are equivalent to the above, but must be done in this order:

```
        if (cpu_sysrt_set_caps(caps, 3, NULL, 0)) {
                perror("cpu_sysrt_set_caps P");
                return -1;
        }

        if (cpu_sysrt_set_caps(NULL, 0, capse, 3)) {
                perror("cpu_sysrt_set_caps E");
                return -1;
        }


======================================================================
/* Set all allowed capabilities */

        if (cpu_sysrt_set_allowed_caps(USERCAPS_SET_PERMITTED|USERCAPS_SET_EFFECTIVE)) {
                perror("cpu_sysrt_set_allowed_caps");
                return -1;
        }
```

The following 2 calls are equivalent to the above, but must be done in this order:

```
        if (cpu_sysrt_set_allowed_caps(USERCAPS_SET_PERMITTED)) {
                perror("cpu_sysrt_set_allowed_caps");
                return -1;
        }
        if (cpu_sysrt_set_allowed_caps(USERCAPS_SET_EFFECTIVE)) {
                perror("cpu_sysrt_set_allowed_caps");
                return -1;
        }
```

SGI Linux Trace

This chapter discusses the following:

- "Overview of SGI Linux Trace" on page 133
- "Installing SGI Linux Trace" on page 134
- "Gathering Trace Data" on page 135
- "Monitoring Trace Events" on page 141
- "Exiting from the `tracevisualizer` GUI" on page 143
- "Removing SGI Linux Trace" on page 144

Overview of SGI Linux Trace

The SGI Linux Trace feature generates traces for kernel events such as interrupt handling, scheduling, and system calls. You can use the SGI Linux Trace tools to record and view trace events and analyze how kernel behavior impacts the execution of applications.

SGI Linux Trace consists of the following:

- A debug kernel with traces inserted
- The `tracevisualizer`(1) graphical user interface (GUI)
- The `tracedaemon`(1) command, which is available from within the GUI or directly from the command line
- Sample platform-specific data files gathered with the frame scheduler enabled and running the `simple_pt` example program:

  ```
  /var/SLT-DataFiles/x86_64/Default-example.proc
  /var/SLT-DataFiles/x86_64/slt-cpu.example-all
  ```

  You can view these files using `tracevisualizer`.

For additional details, see the man pages and the `tracevisualizer` GUI help text in the following file:

`/usr/share/doc/TraceToolkit-0.9.5-1/Help.tracevisualizer`

Note: SGI Linux Trace is based on the open-source Linux Trace Toolkit and has been enhanced.

Installing SGI Linux Trace

To install the `sltdebug` kernel and SGI Linux Trace, do the following:

1. Log in as the superuser.
2. Install the `kernel-sltdebug` RPM:

   ```
   [root@linux root]# rpm -Uvh kernel-sltdebug-*.rpm
   ```

3. Install the `TraceToolkit` RPM:

   ```
   [root@linux root]# rpm -Uvh TraceToolkit-*.rpm
   ```

4. Do one of the following:

   a. To use the `slt` service, set it to start after a reboot and then perform the reboot:

   ```
   [root@linux root]# chkconfig slt on
   [root@linux root]# reboot
   ```

   b. If you do not want to use the `slt` service, you must enter the following the commands manually while running the `slt` kernel. (Without these steps, the `tracedaemon` will not function.)

      i. Create the `/mnt/debug` directory if it does not already exist:

      ```
      [root@linux root]# mkdir /mnt/debug
      ```

      ii. Mount the `debugfs` filesystem:

      ```
      [root@linux root]# mount -t debugfs debugfs /mnt/debug
      ```

iii. Load the `slt` module:

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

Note: This procedure installs the `sltdebug` kernel as the default kernel. When you are done with the `sltdebug` kernel, you should remove the `kernel-sltdebug` RPM or change the symbolic links in `/boot` back to the default kernel. If you reboot into a non-`sltdebug` kernel without removing both the `kernel-sltdebug` and `TraceToolkit` RPMs, you must remove the `slt` module. See "Removing SGI Linux Trace" on page 144.

Gathering Trace Data

The `tracedaemon`(1) command reads buffers of trace data provided by the kernel and writes that data to a file. You can run `tracedaemon` from within the `tracevisualizer` GUI or from the command line.

This section discusses the following:

- "Invoking the `tracevisualizer` GUI" on page 135
- "Recording Events" on page 136
- "Trace Files and Event Types" on page 138
- "Exiting from the `tracedaemon` Command Line" on page 141

Invoking the `tracevisualizer` GUI

To gather data, you must run the `tracevisualizer` GUI or the `tracedaemon` command as `root`. To allow non-`root` users to use the command, you can do one of the following:

- Configure `sudo` to allow execution of `tracedaemon` by specific users.
- Configure the command to set `setuid root`.

To invoke the `tracevisualizer` GUI, enter the following:

```
[user@linux user]# tracevisualizer
```

To write the event data in ASCII format to the specified output file, enter the following:

`tracevisualizer` *trace_input_file proc_file output_file*

For example:

```
[user@linux user]# tracevisualizer slt-cpu.1 Default.proc asciitraceoutput
```

For information about options that filter out the information written to *output_file*, see the `tracevisualizer`(1) man page.

Recording Events

When you want to start recording events, click the stoplight toolbar icon. You can then select options to control the following:

- The time duration for which the trace is to be recorded. You can click **Start** to start recording and **Stop** to stop recording, or you can enter a specific duration in seconds. The default is 120 seconds.
- The CPUs in which threads will be run. Select **Bootcpuset: On** to run threads in every CPU in the bootcpuset (or CPU 0 if no bootcpuset is present) or select **Off** to specify specific CPUs in **CPUs To Run Threads** in one of the following formats:
  - A list of CPUs:

    cpu,cpu,...

  - A range of CPUs (you cannot specify a descending range):

    cpu-cpu

  - A mixture of the above:

    cpu,...cpu-cpu,cpu,...

- The sub-buffer size and number. (A *sub-buffer* is a portion of a CPU buffer. The size of the CPU buffer equals the number of sub-buffers multiplied by the sub-buffer size.) If you experience data being overwritten or dropped, you may need to increase the default values.

- The size of the data files, which can be one of the following:
  - **Fixed size**, sets the size of the sub-buffer and data file. This allows you to continuously collect data within the sub-buffer without filling up all disk space with growing data files.
  - **All data**, which collects data into the sub-buffers and writes those sub-buffers out to the ever-growing data file without restrictions on file size.

Caution: You do not want to collect events indefinitely, because you would end up with huge trace files that would consume all disk space.

You can also use the `tracedaemon` command line to specify the information for recording events:

```
tracedaemon [-h]
tracedaemon [-b] [-c] [-n n_subbufs] [-s subbuf_size] [-t seconds]
tracedaemon [-n n_subbufs] [-r cpulist] [-s subbuf_size] [-t seconds]
```

`-b`	Runs the `tracedaemon` command and threads on all CPUs listed in the bootcpuset. The default is CPU 0.
`-c`	Specifies buffer circular (overwrite) mode, in which data will be written to a fixed size buffer. After the buffer is full, data will be overwritten and lost.
`-h`	Displays the usage statement.
`-n` *n_subbufs*	Specifies the number of sub-buffers. The default is 4.
`-r` *cpulist*	Specifies on which CPUs the `tracedaemon` process and threads can run, where *cpulist* takes one of the following formats described above. This is useful for keeping traffic off of certain CPUs. By default, per-cpu threads run locally on the CPU in which they are collecting data and the `tracedaemon` process runs wherever the scheduler puts it.
`-s` *subbuf_size*	Specifies the sub-buffer size in bytes. The default is 524224.

`-t` *seconds*	Specifies the total run time in seconds. The default is 120 seconds.

For example, to record trace events for 200 seconds and run `tracedaemon` and threads in the bootcpuset (or CPU 0 if there is no bootcpuset), enter the following:

```
[root@linux root]# tracedaemon -t 200 -b
```

Trace Files and Event Types

The event information for each CPU is recorded in a separate file that can be read by `tracevisualizer` and displayed graphically. The files are located in the same directory from which the `tracevisualizer` GUI or the `tracedaemon` command is run.

The trace files are named as follows:

- `slt-cpu.`*N*, with *N* corresponding to the CPU number.
- `slt-cpu.all`, which combines information from all of the individual `slt-cpu.`*N* files. This file is only created when you run `tracedaemon` from inside the GUI.

  Note: If you invoked `tracedaemon` from the command line, the `slt-cpu.all` file is not created.

- `Default.proc` process/IRQ information file.

For example, suppose you have 4 CPUs. If you use the default options in the GUI, the following files would be output:

```
slt-cpu.0
slt-cpu.1
slt-cpu.2
slt-cpu.3
slt-cpu.all
Default.proc
```

Table 11-1 summarizes the types of events that are recorded. For more information, see the `tracedaemon`(8) man page.

Table 11-1 Trace Events that are Recorded

Event Type	Raw Trace Output Name	Event Graph Representation	Description
Advanced programmable interrupt controller (APIC)	`APIC Intr Timer`	`APIC Timer`	The thread has entered a local APIC timer interrupt.
	`APIC Call Func`	`APIC CF`	The thread has entered the APIC `Interrupt for SMP Call` function. The passed function will be run on the CPUs.
	`APIC Intr Exit`	(no representation)	The function execution is finished and the thread is exiting the APIC interrupt processing.
Frame scheduler	`FRS Yield`	`FRSYL`	A frame scheduler application worker thread has called `frs_yield()` to indicate the end of its processing for the current minor frame.
	`FRS Intr Entry`	`FRSINTENT`	The frame scheduler interrupt/event processing has started.
	`FRS Intr Exit`	`FRSINTEX`	The frame scheduler interrupt/event processing has finished.
Interrupt	`Badbreak`	`BBRK`	The thread is entering privileged mode to handle a bad system call.
	`Break`	`BRK`	The thread is entering privileged mode to handle a system call.
	`Fault`	`FAULT`	The thread is entering privileged mode to handle a system fault.

Event Type	Raw Trace Output Name	Event Graph Representation	Description
	`Kernel exit`	(a change will appear in the graph)	The interrupt processing is complete and thread is returning to the previous user-mode processing.
	`Interrupt return`	`INTRT`	The interrupt processing is complete and the thread is returning to the previous kernel-mode processing.
	`IRQ entry`	`IRQ`	The running thread has been preempted to handle the top-half of an interrupt (IRQ) event.
	`Lightweight`	`LTW`	The thread is entering privileged mode via a lightweight mechanism such as a fastpath system call.
	`Opfault`	`OPF`	The thread is entering privileged mode to handle an illegal operation.
	`Pagefault`	`PGF`	The thread is entering privileged mode to handle a fault in the requested page.
	`Soft IRQ`	`SIRQ:`*IRQ number*	Soft-IRQ execution for previous IRQ event.
	`Tasklet action`	`TA:`*function address*	Tasklet execution for previous IRQ event.
	`Tasklet hiaction`	`THA:`*function address*	High-priority tasklet execution for previous IRQ event.
	`Unaligned`	`UNA`	The thread is entering privileged mode to handle an unaligned memory access.
Scheduler	`Sched activate`	`Ac:`*PID*	The thread has been moved onto the CPU run queue. The thread is in the ready-to-run state.
	`Sched deactivate`	`De:`*PID*	The thread has been moved off the CPU run queue. The thread is in the wait/sleeping state.

Event Type	Raw Trace Output Name	Event Graph Representation	Description
	`Sched switch`	`Sw:`*PID*	The CPU has been allocated to a new thread. The new thread's register state, stack, and memory mappings are switched onto the CPU. The new thread is in the running state. The previous thread will likely have been deactivated prior to the switch.
System call	`Syscall entry`	*system call name*	The thread is entering a system call. System calls can be invoked from user-mode and kernel-mode on Linux.
	`Syscall exit`	(no representation)	The thread has exited a system call handler.

Exiting from the `tracedaemon` Command Line

If you run `tracedaemon` from the command line, do one of the following to exit:

- Press `Ctrl-c`
- Enter the following, using the process ID (PID) for the `tracedaemon` process:

```
[root@linux root]# kill -9 tracedaemon_PID
```

Monitoring Trace Events

This section discusses the following:

- "Opening a Trace File" on page 142
- "Zooming In On An Event" on page 142
- "Changing the Time Frame" on page 142
- "Seeing Process Details" on page 143
- "Seeing All Event Trace Details" on page 143
- "Filtering Events Based on CPU" on page 143

For more details, see the GUI help text.

Opening a Trace File

To monitor events, you must open a trace file and the `Default.proc` process/IRQ information file. You must have permission to read the files.

Use the following menu selection to invoke the **Open Trace** window:

File
> Open Trace

Note: You could also click on the left-most icon in the icon bar to open a new trace. For more information about the shortcuts in the icon bar, see the GUI help.

Enter the path to a trace file and process/IRQ information file, or click the **Browse** button to open the **Select Trace File** or **> Select Proc File**, which lets you select a filename.

By default, the trace is shown in the **Event Graph** output, zoomed out to a great distance.

Zooming In On An Event

The graph displays the current start time, the end time, the resulting span of time, and the format of the time ruler in either microseconds (`us`) or nanoseconds (`ns`).

In most cases, the graph will be most useful if you zoom in to a smaller time span. You may also wish to resize the window. To zoom in, select the following from the menu bar or use the + spyglass icon:

Tools
> Zoom In

Changing the Time Frame

You can use several methods to change the time frame:

- Use the scroll bar at the bottom of the graph for slight changes

- Left-click the mouse button to zoom in and right-click to zoom out
- Use the following menu selection to set the start and end time:

  Tools
  > View Time Frame ...

- Display the time difference between the two points on the graph by clicking the middle mouse button at the first point (which will display a vertical line drawn as dashes) and at the second point (which will cause a second vertical line to appear), which then displays the time difference in the left of the bottom status bar.

Seeing Process Details

To see a particular process, click on the **Process Analysis** tab and select the specific process ID number on the left. The right side of the screen will display characteristics of the process and system call accounting. The `Kernel` process (process 0) summarizes the system.

Seeing All Event Trace Details

To see details about all trace entries, click on the **Raw Trace** tab.

Filtering Events Based on CPU

To turn events on or off based on CPU both in the **Event Graph** and **Raw Trace** output, use the **Filter CPU's** menu. Enter the CPUs that you want to display.

Exiting from the `tracevisualizer` GUI

To exit from the `tracevisualizer` GUI, select:

File
> Exit

Removing SGI Linux Trace

To remove the `sltdebug` kernel and SGI Linux Trace, do the following:

1. Log in as the superuser.
2. Remove the `kernel-sltdebug` RPM:

   ```
   [root@linux root]# rpm -ev kernel-sltdebug-*
   ```

3. Remove the `TraceToolkit` RPM:

   ```
   [root@linux root]# rpm -ev TraceToolkit-*
   ```

4. Reboot the system (so that it uses the default kernel).

Note: If you remove the `kernel-sltdebug` RPM but not the `TraceToolkit` RPM, or if you reboot into a non-`sltdebug` kernel without removing either RPM, you must do the following to stop the `slt` service and prevent it from starting after a reboot:

```
[root@linux root]# /etc/init.d/slt stop
[root@linux root]# chkconfig slt off
```

Using the SGI Linux Trace User Library

You can use the SGI Linux Trace (SLT) user library C application programming interface (API) to generate SLT user events, which allow a user program to log data in the same format as the kernel events generated by the SLT debug kernel. No additional SLT kernel events will be generated when logging the user event.

This chapter discusses the following:

- "SLT User Library Routines" on page 145
- "Accessing SLT User Library Routines" on page 147
- "Example Code Using the SLT User Library Routines" on page 147
- "Generating User and Kernel Data" on page 148
- "Examining the Data" on page 148
- "Manually Including User Events in `slt-cpu.all`" on page 149

SLT User Library Routines

This section discusses the following SLT user library routines:

- "`slt_close_utrace`" on page 145
- "`slt_open_utrace`" on page 146
- "`slt_user_trace`" on page 146

`slt_close_utrace`

```
int  slt_close_utrace()
```

The `slt_close_utrace` routine closes the user channel opened by `slt_open_utrace`.

Return values: none

slt_open_utrace

```
int  slt_open_utrace()
```

The `slt_open_utrace` routine opens a user channel that allows user events to be created. You must call this routine before any user events can be created.

Return values: none

slt_user_trace

```
int  slt_user_trace(int EVENT_ID, char* user_string, int cpu)
```

The `slt_user_trace` routine generates up to five user events with the given information supplied by the user.

Input:

Value	Description
`int EVENT_ID`	`SLT_USER_`*N*, where *N* is a numeral in the range `1-5` (such as `SLT_USER_1`).
`char* user_string`	A descriptive string supplied by the user that describes in more detail the type of user event being logged. It can also be used as a search token when examining the user data. The `user_string` can be up to 16 characters in length.
`int cpu`	The CPU on which data was logged (optional). The user can supply the CPU number in order to generate a more accurate representation of the logged data.

Return values:

Value	Description
`0`	Success
`-1`	Error, setting `errno`

Accessing SLT User Library Routines

The following inclusion and linkage provides access to SLT user library the from `C` code:

```
#include <bitmask.h>
#include <react.h>
/* link with -lreact */
```

Example Code Using the SLT User Library Routines

Following is example code using the SLT user library:

```
{
        int user_tt_fd;

        if ((user_tt_fd = slt_open_utrace()) < 0)
          exit(1);

        slt_user_trace(SLT_USER_1, "My Event String",3 /* cpu */);

        slt_close_utrace();

}
```

The following example program that is installed with the TraceToolkit illustrates how to use the SLT user:

`/usr/share/react/SLT/sample-user/u_trace_test.c`

To compile the example:

```
# cc u_trace_test.c -o u_trace_test -lusertrace -lpthread
```

Generating User and Kernel Data

To collect user and kernel data, open the `tracevisualizer`(1) and collect kernel data in the normal manner. While data collection is in progress, execute the user program with the user events:

```
# ./u_trace_test
```

An `slt-cpu.user` data file will be created in the same directory from which the user program was executed. This will contain all of the user events for the last execution of `u_trace_test`.

After the data collection has stopped, the tracevisualizer combines all of the data files in its current directory. For example, suppose you had the following files:

```
slt-cpu.0
slt-cpu.1
...
slt-cpu.N
```

ll of the above files will be combined into one file, `slt-cpu.all`.

Note: To include the user events in `slt-cpu.all`, the `slt-cpu.user` file must be in the same directory as the `slt-cpu.X` files. If this was not the case, see "Manually Including User Events in `slt-cpu.all`" on page 149.

Examining the Data

The `slt-dump` utility program that is installed with the SGI Linux Trace Toolkit will assist in examining the SLT user and kernel data created. To compile it, do the following:

```
# /usr/share/react/SLT/tests/slt-dump.c
# cc slt-dump.c -o slt-dump
```

Due to the amount of data collected, you should redirect the output to a .txt file so that you can examine it with an editor of your choice. For example:

```
# ./slt-dump -cf slt-cpu.all > slt-data-9-20-13:21.txt
```

Manually Including User Events in `slt-cpu.all`

To include the user events in `slt-cpu.all`, the `slt-cpu.user` file must be in the same directory as the `slt-cpu.X` files. If `slt-cpu.all` was created without `slt-cpu.user` in the directory, do the following to manually include the events:

1. Delete the `slt-cpu.all` file.
2. Copy `slt-cpu.user` to the same directory as the `slt-cpu.X` files.
3. Restart the `tracevisualizer`. This will trigger the routine to merge all the data files.

Troubleshooting

This chapter discusses the following:

- "Diagnostic Tools" on page 151
- "Problem Removing `/rtcpus`" on page 154

Diagnostic Tools

You can use the following diagnostic tools:

- Use the `cat`(1) command to view the `/proc/interrupts` file in order to determine where your interrupts are going:

  ```
  [user@linux user]% cat /proc/interrupts
  ```

  For an example, see Appendix A, "Example Application" on page 155.

- 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:

  ```
  [user@linux user]% ps -FC processname
  ```

  For an example, see Appendix A, "Example Application" on page 155.

  To see the scheduling policy, real-time priority, and current processor of all threads on the system, use the following command:

  ```
  [user@linux user]% ps -eLo pid,tid,class,rtprio,psr,cmd
  ```

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

- Use the `top`(1) command to display the largest processes on the system. For more information, see the `top`(1) man page.

- Use the `strace`(1) command to determine where an application is spending most of its time and where there may be large latencies. The `strace` command is a very flexible tool for tracing application activities and can be used for tracking down latencies in an application. Following are several simple examples:

– To see the amount of time being used by system calls in the form of histogram data for a program named `hello_world`, use the following:

```
[root@linux root]# strace -c hello_world
execve("./hello_world", ["hello_world"], [/* 80 vars */]) = 0
Hello World
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 27.69    0.000139          28         5         3 open
 20.92    0.000105          15         7           mmap
 10.76    0.000054          54         1           write
  7.57    0.000038          13         3           fstat
  6.57    0.000033          17         2         1 stat
  5.98    0.000030          15         2           munmap
  4.58    0.000023          12         2           close
  4.38    0.000022          22         1           mprotect
  4.18    0.000021          21         1           madvise
  2.99    0.000015          15         1           read
  2.39    0.000012          12         1           brk
  1.99    0.000010          10         1           uname
------ ----------- ----------- --------- --------- ----------------
100.00    0.000502                    27         4 total
```

– You can record the actual chronological progression through a program with the following command (line breaks added for readability):

```
[root@linux root]# strace -ttT hello_world
14:21:03.974181 execve("./hello_world", ["hello_world"], [/* 80 vars */]) = 0
..
14:21:03.976992 mmap(NULL, 65536, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
   = 0x2000000000040000 <0.000007>
14:21:03.977053 write(1, "Hello World\n", 12Hello World
) = 12 <0.000008>
14:21:03.977109 munmap(0x2000000000040000, 65536) = 0 <0.000009>
14:21:03.977158 exit_group(0)           = ?
```

The time stamps are displayed in the following format:

hour:*minute*:*second*.*microsecond*

The execution time of each system call is displayed in the following format:

<*second*>

Note: You can use the `-p` option to attach to another already running process.

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

- Use the `tracevisualizer` command. See Chapter 11, "SGI Linux Trace" on page 133.
- To find the CPU-to-core numbering scheme, examine the following fields in the `/proc/cpuinfo` file:

  ```
  processor
  physical id
  core id
  ```

  For example, the following output for a third-party x86-64 system shows that logical CPU 0 (`processor 0`) and CPU 2 (`processor 2`) are cores sharing the same socket: (`physical id 0`)

  ```
  processor       : 0
  ...
  physical id     : 0
  siblings        : 2
  core id         : 0
  cpu cores       : 2


  processor       : 2
  ...
  physical id     : 0
  siblings        : 2
  core id         : 1
  cpu cores       : 2
  ```

The following output for an Altix UV 1000 or Altix UV 100 system shows two logical processors CPU 0 (`processor 0`) and CPU 8 (`processor 8`):

```
processor       : 0
..
physical id     : 0
siblings        : 16
core id         : 0
cpu cores       : 8

processor       : 8
..
physical id     : 1
siblings        : 16
core id         : 0
cpu cores       : 8
```

Note the following:

– CPU 0 is housed in the first socket on the system (`physical id 0`). This socket has 8 CPU cores. Each of those cores will have two logical CPUs if hyperthreading is enabled.

– CPU 8 is housed in the second socket (`physical id 1`). This socket has 8 CPU cores. Each of those cores will have two logical CPUs if hyperthreading is enabled.

Each logical CPU is in the first core on its respective socket (`core ID 0`).

Problem Removing `/rtcpus`

You should stop real-time processes before using the `--disable` option. However, the script will attempt to remove the process from the real-time CPUs and display the following failure message if it was unable to move them:

```
"*** Problem removing /rtcpus/rtcpu3. cpuset***
 Try again.  If that doesn't work check /dev/cpuset/rtcpus/rtcpu3/tasks
 for potential problem PIDS;
```

Example Application

This appendix discusses an example of a multithreaded application that demonstrates using external interrupts and other aspects of REACT. It uses netlink sockets to communicate from kernel space to user space. You can use it as a performance benchmark to compare between machines or settings within REACT, such as for external interrupts, cpusets, and CPU isolation.

The application is composed of the following examples:

- A kernel module, which shows examples of the following concepts:
  - Creating and building a driver with a standard miscellaneous device interface
  - Setting up and registering a external interrupt handler
  - Creating and binding a kernel thread
  - Using netlink sockets to communicate with a user application
- A user-space application, which shows examples of the following concepts :
  - Assigning threads to cpusets, thereby changing thread/CPU affinity
  - Changing thread/CPU affinity without cpusets
  - Creating, destroying, and signaling threads
  - Changing a thread's scheduling policies and priorities
  - Locking memory
  - Setting up a netlink socket to communicate with a kernel thread

This example puts the data into a matrix and multiplies two matrices together. The worker thread displays the multiplication and calculates how long it takes to multiply the two matrices together. You can modify the size of the matrix to see how it effects the time to calculate the multiplication. For example, you could use a field-programmable gate array (FPGA) to implement the multiply function in order to show how much faster it is under these circumstances than under normal calculation. You could also run on two different platforms to compare the speed of integer multiplication.

This program runs as a multithreaded process. The main process launches the following threads, sets each thread's scheduling policy and priority, and displays the thread policy and priority information:

- The receiving thread (`netlink_receive`) does the following:
  1. Tells the kernel to start the processing of interrupts (a one-time event).
  2. Locks its current and future memory (if requested).
  3. Uses the example kernel module driver to do the following:
     a. Waits for messages from the kernel netlink socket.
     b. Signals the worker thread with the data from the driver.
- The worker thread (`worker_routine`) does the following:
  1. Waits to be signaled by the receive thread for data.
  2. Fills two matrices with the data and multiplies them together. The output will be printed to the console.
  3. Calculates the time it takes for the matrices to by multiplied together.
- The interrupt handler (`extint_run`) runs when a hardware external interrupt is received. It wakes up the `bench_extintd` thread.
- The kernel thread (`bench_extintd`) gets data, sends messages with the data to the receiving thread (`netlink_receive`), and then sleeps until another interrupt occurs.

`netlink_receive` is set at a higher priority than the time-consuming `worker_routine`.

Figure A-1 describes the example. Step 1 occurs once, but steps 2 through 4 are repeated for each external interrupt.

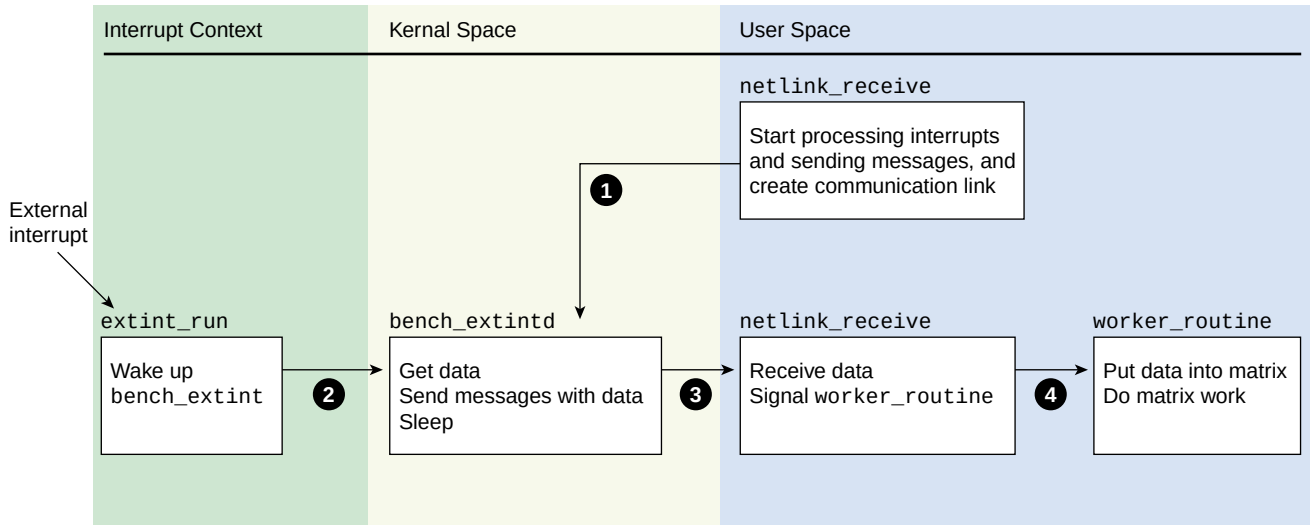

Figure A-1 Example Work Flow

The rest of this section discusses the following:

- "Setting Up External Interrupts" on page 157
- "Building and Loading the Kernel Module" on page 158
- "Building the User-Space Application" on page 159
- "Running the Sample Application" on page 159
- "`set_affinity` code" on page 162

Setting Up External Interrupts

To set up external interrupts, do the following:

1. Log in to the target system as `root`.
2. Load the `ioc4_extint` module:

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

3. Insert the required information into the source, mode, and period files in the /sys/class/extint/extint0/ directory. For example:

```
[root@linux root]# echo loopback >/sys/class/extint/extint0/source
[root@linux root]# echo toggle >/sys/class/extint/extint0/mode
[root@linux root]# echo 1000000 >/sys/class/extint/extint0/period
```

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

Building and Loading the Kernel Module

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

1. Log in to the target system as root.
2. Ensure that the kernel-source-*.rpm RPM is installed.
3. Ensure that the sgi-extint-kmp-modvers RPM is installed.
4. Copy the Module.symvers file from its location in the directory defined by the uname -r output to the kernel directory:

```
[root@linux root]# cp /usr/share/extint/‘uname -r‘/Module.symvers /usr/share/react/examples/bench/kernel/.
```

5. Change to the kernel directory:

   ```
   [root@linux root]# cd /usr/share/react/samples/bench/kernel
   ```

6. Build the bench_extint_mod.ko file:

```
[root@linux kernel]# make -C /lib/modules/‘uname -r‘/build SUBDIRS=$PWD modules
```

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

7. Copy the bench_extint_mod.ko file to the directory defined by the uname -r output:

   ```
   [root@linux kernel]# cp bench_extint_mod.ko /lib/modules/‘uname -r‘
   ```

8. Make a dependency file:

   ```
   [root@linux kernel]# depmod
   ```

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

9. Load the bench_extint_mod module:

```
[root@linux kernel]# modprobe bench_extint_mod
```

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

10. Use the bench_extint_mod kernel module with the bench_example application.

Note: You must load the ioc4_extint module before the bench_extint_mod module.

Building the User-Space Application

To build the user-space module, do the following:

1. Change to the user directory:

```
[root@linux root]# cd /usr/share/react/samples/bench/user
```

2. Build the module:

```
[root@linux root]$ make
```

Running the Sample Application

You can run the bench_example application in the following modes:

- *Matrix multiply mode* receives data from the kernel module and puts that data into a matrix. After two matrices are full, it multiplies them together and calculates the amount of time taken for the calculation. See "Matrix Multiply Mode Examples" on page 161.
- *Netlink socket bench mode* causes the application to send multiple messages from kernel space to user space during one iteration. The number of messages sent per iteration depends upon notification from the user application to start sending messages. See "Netlink Socket Benchmark Mode Examples" on page 161.

Do the following:

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

  For example:

```
[root@linux root]# lsmod
          Module                  Size  Used by
          bench_extint_mod      546232  0
          ioc4_extint            27272  0
          ioc4                   24704  1 ioc4_extint
          extint                 32008  2 bench_extint_mod,ioc4_extint
```

  If the output does not include `bench_extint_mod`, follow the instructions in "Building and Loading the Kernel Module" on page 158.

- Execute the `bench` command as desired.

  The `bench` command has the following options:

`-b` *messages*	Runs the application in benchmark mode with the specified number of messages in each send. *messages* is an integer in the range `1` through `100`. (If you enter an invalid number, the default is `100`.)
`-h`	Prints usage instructions.
`-k` *cpu*	Specifies the CPU where the kthread will run.
`-m`	Locks memory.
`-p` *cpu*	Specifies the CPU where the bench process will run.
`-r` *cpu*	Specifies the CPU where the receive thread will run.
`-s` *size*	Specifies the size of buffers in bytes for network socket bench mode. The default is `1024`. You can vary the size of the buffers to see the impact on performance.
`-t` *sec*	Specifies the total run time in seconds, with a maximum of 30 seconds. The default is `30`.

`-w` *cpu* Specifies the CPU where the worker thread will run.

Matrix Multiply Mode Examples

To run in matrix multiply mode for 30 seconds:

```
[root@linux root]# ./bench -t30
```

To run with memory locked and bench processes running on CPU 2 (real-time or non-real-time):

```
[root@linux root]# ./bench -m -p2 -t30
```

To run the bench process on CPU 3 and the worker and receive threads on CPU 2:

```
[root@linux root]# ./bench -m -p3 -r2 -w2 -t30
```

See also "`set_affinity` code" on page 162.

Netlink Socket Benchmark Mode Examples

The following shows an example in bench mode that runs for 30 seconds with memory locked and a buffer size of 512 bytes. There are 50 messages in each send. The process is running on CPU 1, the receive thread running on CPU 2, the worker thread is running on CPU 3, and the kernel thread is running on CPU 1:

```
[root@linux root]# ./bench -m -t30 -p1 -r2 -w3 -k1 -b50 -s512
```

If you have multiple terminals open, you can run the following `tail`(1) and `ps`(1) commands to see where things are running:

```
[root@linux root]# tail -f /var/log/messages
Feb 16 08:54:05 dewberry kernel: bench_extint init
Feb 16 08:54:40 dewberry kernel: bench_extint ran 14958, thread ran 14958 dropped msgs 0
Feb 16 08:54:40 dewberry kernel: ioctl unregister bench_extint

[root@linux root]# ps -eLF
UID     PID  PPID   LWP  C NLWP    SZ   RSS PSR STIME TTY        TIME CMD
root  10076  6747 10076  0    3  5951 18696   1 11:34 pts/0  00:00:00 ./bench -m -t30 -p1 -r2 -w3 -k1 -b50 -s512
root  10076  6747 10078 11    3  5951 18696   2 11:34 pts/0  00:00:00 ./bench -m -t30 -p1 -r2 -w3 -k1 -b50 -s512
root  10076  6747 10079 99    3  5951 18696   3 11:34 pts/0  00:00:04 ./bench -m -t30 -p1 -r2 -w3 -k1 -b50 -s512
root  10077    15 10077 10    1     0     0   1 11:34 ?      00:00:00 [bench_exintd]
```

`set_affinity` code

You can use the following functions to set process and thread affinity for real-time and non real-time CPUs. You can compile this file as part of another application, but you must link it against the `libcpuset`.

```
#include <sys/syscall.h>
#include <unistd.h>
#include "errors.h"

#define CPUSET_ROOT "/dev/cpuset"
#define BITS_PER_LONG (sizeof(unsigned long) * 8)


pid_t _gettid(){
  return syscall(__NR_gettid);
}

void do_pthread_affinity(int cpu) {

        int nrcpus = cpuset_cpus_nbits();
        int bitmask_size = (nrcpus/BITS_PER_LONG);
        unsigned long cpus[bitmask_size];
        pid_t tid = _gettid();

        cpus[cpu/64] = 1 << (cpu % 64);

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

}
void set_thread_affinity(int cpu) {

        char path[50],fullpath[50];

        sprintf(path, "/rtcpus/rtcpu%d", cpu);
        sprintf (fullpath, CPUSET_ROOT "/rtcpus/rtcpu%d",cpu);

        if (access(fullpath, F_OK) != 0) {
                /* no cpuset, so try moving it without */
```

```
                do_pthread_affinity(cpu);
                return;
        }

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


/* Set the current proc to run on cpu . */
void set_process_affinity(int cpu) {

        int nrcpus = cpuset_cpus_nbits();
        int bitmask_size = (nrcpus/BITS_PER_LONG);
        unsigned long cpus[bitmask_size];
        char path[50],fullpath[50];
        unsigned long mask;

        cpus[cpu/64] = 1 << (cpu % 64);

        sprintf(path, "/rtcpus/rtcpu%d", cpu);
        sprintf (fullpath, CPUSET_ROOT "/rtcpus/rtcpu%d",cpu);

        if (access(fullpath, F_OK) != 0) {
                /* no cpuset, so try moving it without */
        if (sched_setaffinity(getpid(), sizeof(cpus), cpus)) {
                        perror("set_process_affinity");
                }
                return;
        }
        /* Move the process into the cpuset */
        if (cpuset_move(getpid(), path) == -1)
                perror("cpuset_move");


}
```

High-Resolution Timer Example

Example B-1 demonstrates the use of SGI high-resolution timers. It will run high-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.                                                             *
 *                                                                           *
 * A simple way to build this sample program is:                             *
 *   cc -o timer_sample timer_sample.c -lrt                                  *
 *                                                                           *
 * Invocation example (500 usec timer):                                      *
 *   ./timer_sample 500                                                      *
 *                                                                           *
 * Invocation example (500 usec timer on realtime cpu 2):                    *
 *   cpuset --invoke=/rtcpu2 --invokecmd=./timer_sample 500                  *
 *                                                                           *
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <time.h>
#include <errno.h>
#include <asm/unistd.h>
#include <pthread.h>
#include <strings.h>
#include <sys/time.h>
#include <getopt.h>
#include <libgen.h>

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 high resolution timer. */
        printf("\nRunning with CLOCK_REALTIME (normal resolution)..\n");
        if (timer_test(CLOCK_REALTIME, timeout * 1000)) {
                return -1;
        }
}
```

Sample User-Level Interrupt Programs

The following applications demonstrate some of the user-level interrupt (ULI) interface:

- "`uli_sample` Sample Program" on page 171
- "`uli_ei` Sample Program" on page 172

The applications are installed with the ULI RPM and are located in:

`/usr/share/react/uli/examples/`

`uli_sample` Sample Program

The `uli_sample` program registers for notification on CPU 0 for occurrences of a specified interrupt number. To use `uli_sample`, do the following:

1. Load the ULI feature kernel module:

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

2. Change to the directory containing `uli_sample`:

   ```
   [root@linux root]# cd /usr/share/react/uli/examples/
   ```

3. Run `uli_sample`, where *interrupt#* is the interrupt number:

   ```
   [root@linux root]# ./uli_sample interrupt#
   ```

   For example, to register for notification on CPU 0 for occurrences of the interrupt number 34, enter the following:

   ```
   [root@linux root]# ./uli_sample 34
   ```

`uli_ei` Sample Program

The `uli_ei` program requires the external interrupt to run and prints a message every time the external interrupt line is toggled. To use `uli_ei`, do the following:

1. Load the ULI feature kernel module, if not already done:

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

2. Load the external interrupt kernel module:

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

3. Set the external interrupt mode to `toggle`:

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

4. Change to the directory containing `uli_ei`:

   ```
   [root@linux root]# cd /usr/share/react/uli/examples/
   ```

5. Run `uli_ei`:

   ```
   [root@linux root]#  ./uli_ei
   ```

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.

APIC

Advanced programmable interrupt controller.

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*.

BAR

Base address register.

clock tick

A measure of time determined by the resolution of the real-time clock.

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.

CPU

Central Processing Unit refers to cores (not sockets).

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.

IPI

Interprocessor interrupt.

IRQ

Interrupt request.

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*.

LSM

Linux security model.

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*.

microsecond (*us* or *usec*)

1 microsecond is .000001 seconds. Abbreviated as *us* or *usec*.

millisecond (*ms* or *msec*)

1 millisecond is .001 seconds. Abbreviated as *ms* or *msec*.

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.

MPI

Message passing interface.

nanosecond (*ns*)

1 nanosecond is .000000001 seconds. Abbreviated as *ns* or *nsec*.

new pthreads library (NPTL)

The Linux pthreads library shipped with 2.6 Linux.

overrun

When incoming data arrives faster than a data collection system can accept it and therefore data is lost.

overrun exception

When a thread or process scheduled by the frame scheduler should have yielded before the end of the minor frame but did not.

page fault

The hardware event that results when a process attempts to access a page of virtual memory that is not present in physical memory.

pages

The units of real memory managed by the kernel. Memory is always allocated in page units on page-boundary addresses. Virtual memory is read and written from the swap device in page units.

peak data rate

The instantaneous maximum rate of input to a data collection system. The system must be able to accept data at this rate to avoid overrun. See also *average data rate*.

process

The entity that executes instructions in a Linux system. A process has access to an *address space* containing its instructions and data.

pthread

A thread defined by the POSIX standard. Pthreads within a process use the same global address space. Also see *thread*.

rate-monotonic analysis

A technique for analyzing a program based on the periodicities and deadlines of its threads and events.

rate-monotonic scheduling

A technique for choosing scheduling priorities for programs and threads based on the results of *rate-monotonic analysis*.

restrict

To prevent a CPU from running scheduled processes.

scheduling discipline

The rules under which an activity thread or process is dispatched by a frame scheduler, including whether or not the thread or process is allowed to cause *overrun* or *underrun exceptions*.

segment

Any contiguous range of memory addresses. Segments as allocated by Linux always start on a page boundary and contain an integral number of pages.

semaphore

A memory object that represents the availability of a shared resource. A process that needs the resource executes a p operation on the semaphore to reserve the resource,

blocking if necessary until the resource is free. The resource is released by a V operation on the semaphore. See also *locks*.

shield

To switch off the timer (scheduler) interrupts that would normally be scheduled on a CPU.

simulator

An application that maintains an internal model of the world. It receives control inputs, updates the model to reflect them, and outputs the changed model as visual output.

slaves

The other schedulers that take their time base interrupts from the *master scheduler*. See also *sync group*.

soft real-time program

A program that can occasionally miss a deadline with only minor adverse effects. See also *firm real-time program* and *hard real-time program*.

software latency

The time required to dispatch an interrupt thread.

spraying interrupts

The distribution of I/O interrupts across all available processors as a means of balancing the load.

stimulator

An application that maintains an internal model of the world. It receives control inputs, updates the model to reflect them, and outputs the changed model as nonvisual output.

sub-buffer

A portion of a CPU buffer. The size of the CPU buffer equals the number of sub-buffers multiplied by the sub-buffer size.

sync group

The combination of a *master scheduler* and *slaves*.

thread

An independent flow of execution that consists of a set of registers (including a program counter and a stack). Also see *pthread*.

TLB

Translation lookaside buffer, which translates CPU virtual memory addresses to bus physical memory addresses.

transport delay

The time it takes for a simulator to reflect a control input in its output display. Too long a transport delay makes the simulation inaccurate or unpleasant to use.

ULI

User-level interrupt

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.

us (or usec)

Microsecond (1 us is .000001 seconds).

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

Index

A

abstraction layer, 17
access to select REACT features, 112
activity thread management, 68
address space (locking in memory), 102
advanced programmable interrupt controller (APIC), 139
aircraft simulator, 3
allowed capabilities REACT library routine, 126
API
  REACT library, 119
  SLT user library, 145
application example, 155
asynchronous I/O, 91
average data rate, 5

B

/boot, 107
/boot cpuset, 119, 122
BOOTCPUS, 122
bootcpuset, 42, 107
BOOTMEMS, 122

C

C language, 7
cache warming, 63
callout deregistration, 30
callout mechanism, 29
callout registration, 29
CAP_DAC_OVERRIDE authority, 113
cap_ipc_lock, 129
cap_sys_nice, 129
capabilities REACT library routine, 127
cat, 151
clock processor, 43
clock_gettime, 13, 14
CLOCK_MONOTONIC, 13
CLOCK_REALTIME, 13
clock_settime, 13
clocks, 12
close a user trace routine, 145
configuration, 107
console interrupts, 11
control law process stimulator, 4
controller thread, 66, 77
core ID, 154
cores requirement, 7
CPU
  restricting, 10, 45
  workload control, 39
CPU 0, 43
CPU affinity routine, 123
CPU designation routine, 126
CPU-bound, 9
CPU-to-core numbering scheme, 153
cpu_shield, 120
cpu_sysrt_add, 121
cpu_sysrt_delete, 122
cpu_sysrt_info, 122
cpu_sysrt_irq, 123
cpu_sysrt_perm, 124
cpu_sysrt_runon, 126
cpu_sysrt_set_allowed_caps, 126
cpu_sysrt_set_caps, 127
CPUs in the /boot cpuset, 122
cpuset, 42, 117
cpuset-utils, 7
cpusets, 54
create real-time routine, 121

cycles per second, 3

D

data collection system, 5
debug kernel, 6
delete real-time routine, 122
deregistration of callout, 30
dev attribute file, 18
/dev/extint#, 102
device service time, 48, 51
device special file, 102
device-driver time base, 55
diagnostic tools, 151
direct RTC access, 14
disciplines, 9
disk I/O optimization, 91
distributed applications, 15
dplace, 119
driver creation and building, 155
driver deregistration, 28
driver interface, 23
driver registration, 24
driver template, 31

E

earnings-based scheduler, 10
/etc/pam.d/sshd, 129
/etc/security/capability.conf, 129
/etc/sysconfig/sgi-react.conf, 112
/etc/udev/rules.d/99-sgi-react.rules, 112
event recording, 137
events recorded by tracedaemon, 138
examples
  API code
    REACT library , 129
    SLT user library, 147
  matrix multiply mode, 161
  multithreaded application, 155
  Netlink socket benchmark mode, 161
exception types, 81
external interrupt ingest, 36
external interrupt setup and registration, 155
external interrupt with frame scheduler, 70
external interrupts, 17
extint, 7, 19, 102
extint_device, 23
extint_properties, 23

F

fastcall, 13
features, 6
feedback loop, 2
filter tracedaemon events based on CPU, 143
firm real-time program, 1
first-in-first-out, 10
flock system call, 20
fork(), 78
FPGA, 155
frame interval, 3
frame rate, 2
frame scheduler, 6, 53
  advantages, 12
  API, 58
  background discipline, 73
  basics, 54
  concepts, 53
  continuable discipline, 73
  controller thread, 66
  current frame extension, 82
  design process, 75
  exception counts, 84
  exception handling, 81
  exception policies, 83
  exception types, 81
  external interrupt, 70
  frame scheduler controller, 58
  frs_run flag, 64

frs_yield flag, 64
high-resolution timer, 70
interval timers not used with, 89
library interface for C programs, 60
major frame, 56
managing activity threads, 68
minor frame, 56
multiple exceptions, 83
multiple synchronized, 66
overrun exception, 71, 81
overrunnable discipline, 72
overview, 11
pausing, 67
preparing the system, 76
process outline for single, 77
real-time discipline, 71
repeat frame, 82
scheduling disciplines, 71
scheduling rules of, 64
sequence error handling, 88
signal use under, 86
signals in an activity thread, 87
signals produced by, 86, 87
starting up a single scheduler, 66
starting up multiple schedulers, 67
synchronized schedulers, 78
thread programming model, 55
thread structure, 62
time base selection, 55, 69
underrun exception, 71, 81
underrunable discipline, 72
using consecutive minor frames, 73
warming up cache, 63
frame scheduler controller, 58
receives signals, 87
frs
See "frame scheduler", 53
frs_create(), 60, 77
frs_create_master(), 60, 78, 79
frs_create_slave(), 60, 80
frs_create_vmaster(), 60, 78, 79
frs_destroy(), 62, 78, 80, 81
frs_enqueue(), 60, 67, 78
frs_fsched_info_t, 58
frs_getattr(), 62, 84
frs_getqueuelen(), 61, 68
frs_intr_info_t, 59
frs_join, 61
frs_join(), , 62, 67, 78, 80
frs_overrun_info_t(), 84
frs_pinsert(), 61, 68
frs_premove(), 62, 68, 87
frs_pthread_enqueue(), 61, 64, 71, 78, 80
frs_pthread_getattr(), 62, 84
frs_pthread_insert, 61
frs_pthread_insert(), 68
frs_pthread_readqueue(), 61, 68
frs_pthread_register(), 62
frs_pthread_remove(), 62, 68, 87
frs_pthread_setattr(), 61, 83
example code, 84
frs_queue_info_t, 58
frs_readqueue(), 61, 68
frs_recv_info_t, 59
frs_resume(), 61, 67
frs_run, 64
frs_setattr(), 61, 83
frs_start, 61
frs_start(), 67, 78, 80
frs_stop, 61
frs_stop(), 67
frs_t, 58
frs_userintr(), 61
frs_yield, , 61, 62, 64, 73
fsync, 92

G

generate user events routine, 146
generating a REACT system configuration, 107
global variables and ULI, 101
_GNU_SOURCE, 20

ground vehicle simulator, 3

H

hard real-time program, 1
hardware latency, 48, 49
hardware–in–the–loop simulator, 4
high-output modes, 34
high-priority tasklet execution event, 140
high-resolution timer, 70, 165
HUB hardware timers, 53
hyperthreading, 154
Hz (hertz, cycles per second), 3

I

I/O interrupts, 11
I/O-bound, 9
IDE driver, 33
illegal operation event, 140
implementation functions, 24
include files, 128, 147
ingest section for external interrupts, 36
inheritable capability enabling, 128
interchassis communication, 14
internal driver circuit I/O connectors, 37
interrupt
  group. See interrupt group, 69
  See also user-level interrupt (ULI), 97
interrupt control, 11
interrupt events, 140
interrupt group, 69
interrupt information template, 59
interrupt notification interface, 28
interrupt propagation delay, 49
interrupt redirection, 43
interrupt response time
  components, 48
  definition of, 47
  minimizing, 51
interrupt return event, 140
interrupt service routines (ISRs), 51, 97
interval
  See "frame interval", 3
interval timer, 89
introduction, 1
IOC4 chip, 17
IOC4 driver, 32
ioc4_extint, 102
IOC4–specific character special device and class, 31
IRQ redirection, 119

K

kernel critical section, 50
kernel data generation, 148
kernel facilities for real-time, 9
kernel module insertion/removal, 46
kernel scheduling, 39
kernel thread control, 42
kernel thread creating and binding, 155

L

latency, 48, 49
libbitmask, 7
libcpuset, 7, 119
libreact, 8
libuli, 98
linkage, 128, 147
Linux requirement, 7
Linux Trace, 133
Linux Trace Toolkit, 134
lk, 8
LOCK_MAND, 20
locking memory, 102
locking virtual memory, 10
low output modes, 34
low-level driver interface, 23

low-level driver template, 31
lspci, 93

M

major frame, 56
master controller thread, 79
master scheduler, 79
maximum response time guarantee, 48
mechanism for callout, 29
memory locking, 155
memory locking (virtual), 10
memory nodes assigned to the /boot cpuset, 122
memory requirement, 7
memory-mapped I/O, 91
Message-Passing Interface (MPI), 15
minor frame, 56, 64
mlock(), 10, 100
mlockall(), 10, 100
mmap, 20
mode attribute file, 18
mode switch, 48, 51
modelist attribute file, 18
monitoring trace events, 142
MPI, 15
ms (milliseconds), 3
msync, 91, 92
multiple devices and ULI, 101
multiple independent drivers, 32
multiprocessor architecture, 66

N

netlink socket use, 155
new pthreads library (NPTL), 42
nice value, 9
normal-time program, 1
NPTL, 42

O

open a user trace routine, 146
operating system requirements, 7
operator, 2
output modes, 34
overhead work, 43
overrun, 5
overrun exception, 64
overrun in frame scheduler, 71

P

page fault, 10
page fault event, 140
pam_capability, 127, 128
param.h, 40
PCI devices and programmed I/O, 93
PCI-RT-Z, 53
peak data rate, 5
period attribute file, 18
permissions, 112
permissions routine, 124
physical ID, 154
physical interfaces, 36
physical memory requirements, 10
poll, 20
POSIX
  real-time policies, 10
  real-time specification 1003.1-2003, 92
power plant simulator, 3
priorities, 39
priority band, 40
problem removing /rtcpus, 154
/proc manipulation, 11
/proc/cpuinfo, 153
/proc/interrupts, 44, 151
process control, 6
process details in tracevisualizer, 143
process mapping to CPU, 10

process running on a real-time CPU, 117
processor requirement, 7
profile.pl, 151
programmed I/O and PCI devices, 93
programming language for REACT, 7
propagation delay, 49
provider attribute file, 19
ps, 41, 151
pthread priority, 42
pthread_attr_setinheritsched(), 42
pthread_attr_setschedparam(), 42
pthread_attr_setschedpolicy(), 42
pthread_attr_t, 42
pthread_attr_t(), 59
pthread_create(), 42, 78
PTHREAD_EXPLICIT_SCHED, 42
PTHREAD_INHERIT_SCHED, 42
pthread_setschedparam(), 42
pthread_t, 59
pulse output modes, 34

Q

quantum attribute file, 19

R

rate
  See "frame rate", 3
raw trace, 143
react command, 6
react-utils, 8
read system call, 20
real-time applications, 2
real-time clock (RTC), 12
real-time CPU and running a process, 117
real-time CPUs currently configured on the system, 123
real-time memory nodes associated with the real-time CPUs, 123
real-time priority band, 40
real-time program
  and frame scheduler, 11
  terminology, 1
register access, 14
register format, 35
registration of callout, 29
repeat frame, 82
requirements, 7
response time guarantee, 48
restricting a CPU, 45
RHEL requirement, 7
round-robin, 10
RPMs, 7
RT_NO_WAIT, 121, 122
RT_WAIT, 121, 122
RTC, 12
RTC access, 14
rtcpu, 54
rtcpu devices, 107
RTCPUS, 123
RTMEMS, 123

S

sched_setparam(), 40
sched_setscheduler(), 10, 40
scheduler events, 139, 140
scheduling, 39
scheduling disciplines, 9, 71
scheduling policy, 155
select system call, 20
SGI Linux Trace, 6, 133
SGI Linux Trace user library, 145
sgi-extint-kmp-*, 7
sgiioc4 driver, 33
sig_dequeue, 87
sig_overrun, 87
sig_underrun, 87
sig_unframesched, 87

signal, 86
signal handler, 80
SIGRTMIN, 87
SIGUSR1, 87
SIGUSR2, 87
simulator, 2
single frame scheduler start, 66
slave controller thread, 80
slave scheduler, 79
SLES requirement, 7
SLT, 133
  See "SGI Linux Trace", 145
slt-cpu.all, 138, 148
slt-cpu.N, 138
slt-cpu.user, 148
SLT-DataFiles, 133
slt_close_utrace, 145, 146
slt_open_utrace, 146
slt_user_trace, 147
sltdebug kernel, 144
SN hub device interrupts, 51
socket programming, 14
soft real-time program, 1
soft-IRQ execution event, 140
software latency, 48, 49
source attribute file, 19
sourcelist attribute file, 19
special scheduling disciplines, 9
stimulator, 2
strace, 41, 151
strobe output modes, 34
sub-buffer size, 136
swapping requirement, 7
sync group, 79
synchronous I/O, 92
/sys/class/extint/extint#/ , 19
/sys/class/ioc4_intout/intout#/dev, 35
sysconf(_SC_CLK_TCK), 13
sysfs attribute files, 18
system call event, 141
system call exit, 141
system configuration generation, 107
system fault event, 139
system-call time base, 55

T

tasklet action event, 140
tasklet hiaction, 140
thread, 58
thread control, 42
thread creation, destruction, and signals, 155
thread programming model, 55
time base for frame scheduler, 69
time base support, 55
time difference, 143
time estimation, 65
time frame in tracevisualizer, 142
time slices, 40
time-share applications, 10
Timer interrupt control REACT library routine, 120
timer interrupts, 11, 40
timer_create(), 13
toggle output modes, 34
top, 151
Trace, 6
trace data gathering, 135
trace events, 138
trace files and event types, 138
tracedaemon, 133, 137
tracevisualizer, 133, 135, 148
transport delay, 3
troubleshooting, 151

U

u_trace_test, 148
ULI
  See "User-level interrupt (ULI)", 97
uli, 98
ULI_block_intr, 99

ULI_destroy, 98
ULI_register_irq(), 98, 103
ULI_sleep(), 99
ULI_unblock_intr, 99
ULI_wakeup(), 99
unaligned access event, 140
underrun exception, 64
underrun, in frame scheduler, 71
unsupported hardware device capabilities, 31
usecs (microseconds), 48
user access, 112
user application communication, 155
user capabilities, 112
user data generation, 148
user event generation, 145
user thread control, 42
user thread dispatch, 51
user trace routines, 146
user-level interrupt (ULI)
  concurrency, 101
  global variables, 101
  handler interaction, 104
  initializing, 101
  interrupt handler registration, 103
  multiple devices, 101
  mutual exclusion, 105
  overview, 97
  per-IRQ handler, 103
  program address space locking, 102
  restrictions on handler, 99
  ULI_block_intr(), 105
  ULI_sleep (), 104
  ULI_sleep () function, 101
  ULI_wakeup () function, 104
user-level interrupts (ULI), 171
usercaps, 128
USERCAPS_SET_EFFECTIVE, 126
USERCAPS_SET_PERMITTED, 126
/usr/include/asm/param.h, 40
/usr/include/sn/timer.h, 14
/usr/include/sys/pthread.h, 59
/usr/share/src/react/examples, 60

V

virtual memory locking, 10
virtual reality simulator, 4
volatile keyword, 101
Vsync time base, 55

W

wave stimulator, 5
write bitmask routine, 122

Z

zooming in tracevisualizer, 142