Chapter 7

Using the Frame Scheduler

The REACT/Pro Frame Scheduler makes it easy to structure a real-time program as a family of independent, cooperating processes, running on multiple CPUs, scheduled in sequence at the frame rate of the application. For an overview of the Frame Scheduler, see "REACT/Pro Frame Scheduler" on page 25.

This chapter contains details on the operation and use of the Frame Scheduler, under these main headings:

- "Frame Scheduler Concepts" on page 96 details the operation and methods of the Frame Scheduler.
- "Selecting a Time Base" on page 107 covers the important choice of which source of interrupts should define a frame interval.
- "Using the Scheduling Disciplines" on page 110 explains the options for scheduling activities of different kinds.
- "Preparing the System" on page 113 reviews the system administration steps needed to prepare the CPUs that the Frame Scheduler will use.
- "Implementing a Single Frame Scheduler" on page 114 outlines the structure of an application that uses one CPU.
- "Implementing Synchronized Schedulers" on page 115 outlines the structure of an application that needs the power of multiple CPUs.
- "Handling Frame Scheduler Exceptions" on page 118 describes how overrun and underrun exceptions are dealt with.
- "Using Signals Under the Frame Scheduler" on page 122 discusses the issue of signal latency and the signals the Frame Scheduler generates.
- "Using Timers with the Frame Scheduler" on page 125 covers the use of itimers with the Frame Scheduler.
- "The Frame Scheduler Device Driver Interface" on page 126 documents the way that a kernel-level device driver can generate time-base interrupts for a Frame Scheduler.

Frame Scheduler Concepts

One Frame Scheduler dispatches selected processes at a real-time rate on one CPU. You can also create multiple, synchronized Frame Schedulers so as to dispatch concurrent processes on multiple CPUs.

Frame Scheduler Basics

A Frame Scheduler takes over the scheduling and dispatching of processes on one CPU. It isolates the CPU (see "Isolating a CPU From TLB Interrupts" on page 84), and completely supersedes the operation of the normal IRIX scheduler on that CPU. Only processes enqueued to the Frame Scheduler can use the CPU. IRIX dispatching priorities are not relevant on that CPU. If that CPU is assigned to a processor set (see "Using Processor Sets" on page 76), the set assignment is ignored while the Frame Scheduler is running.

The execution of normal processes, daemons, and pending timeouts are all migrated to other CPUs—typically to CPU 0, which cannot be owned by a Frame Scheduler. All interrupt handling is usually directed away from a Frame Scheduler CPU as well (see "Preparing the System" on page 113). However, a Frame Scheduler CPU can be used to handle interrupts, although doing so runs a risk of causing overruns.

Frame Scheduling

Instead of scheduling processes according to priorities with an attempt at fairness, 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 107.)

The interrupts from the time base define *minor frames*. You tell the Frame Scheduler a fixed number of minor frames that should be considered 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 sub-frames. Major and minor frames are depicted in Figure 7-1.

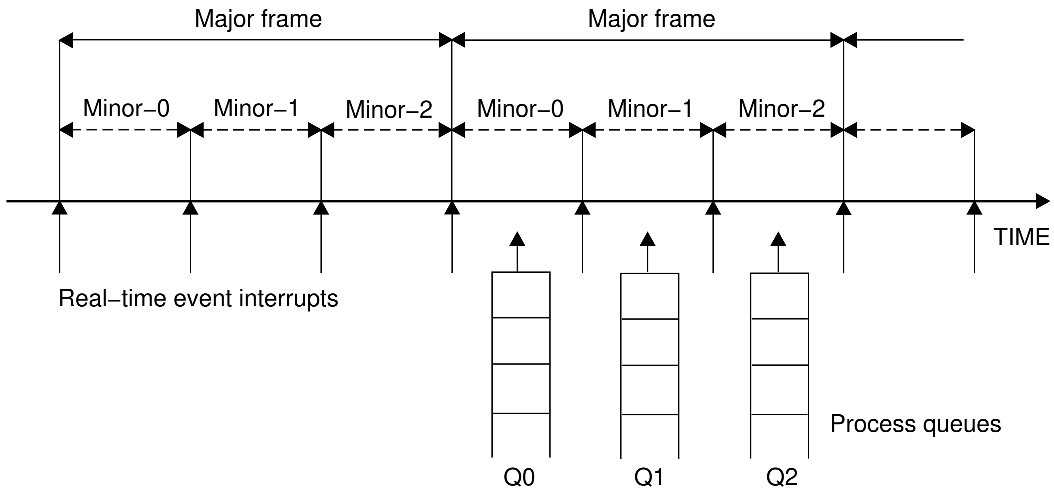

Figure 7-1 Major and Minor Frames

As pictured in Figure 7-1, the Frame Scheduler maintains a queue of processes for each minor frame. You enqueue each activity processes of your program to a specific minor frame. You determine the order of cyclic execution within a minor frame by the order in which you enqueue processes. You can:

- Enqueue multiple processes in one minor frame. They are run in queue sequence within the frame. All must complete their work within the minor frame interval.
- Enqueue the same process to run in more than one minor frame. Say that process *double* is to run twice as often as process *solo*. You could enqueue *double* to Q0 and Q2 in Figure 7-1, and enqueue *solo* to Q1.
- Enqueue a process that takes more than a minor frame to complete its work. If process *sloth* could need more than one minor interval, you could enqueue it to Q0, Q1 and Q2 in Figure 7-1, such that it would be able to continue working in all three minor frames until it completed.
- Enqueue a background process that is allowed to run only when all others have completed, to use up any remaining time within a minor frame.

All these options are controlled by scheduling disciplines you specify for each process as you enqueue it (see “Using the Scheduling Disciplines” on page 110).

The processes that a Frame Scheduler dispatches are typically child processes of the process that creates the Frame Scheduler, but that is not a requirement. Any process can be enqueued, even one that starts execution as a separate command.

The FRS Control Process

The process that creates a Frame Scheduler is called the *frs control* process. It is privileged in three respects:

- Its process ID (PID) is used to identify its Frame Scheduler in various functions.
- It can receive signals when errors are detected by the Frame Scheduler (see "Using Signals Under the Frame Scheduler" on page 122).
- It cannot itself be enqueued to the Frame Scheduler. It continues to be dispatched by IRIX. It executes on some other CPU than the one the Frame Scheduler uses.

The Frame Scheduler API

The details of the Frame Scheduler API can be found in the frs(3) reference page. The API elements are declared in */usr/include/sys/frs.h*. The following are some important types are declared in */usr/include/sys/frs.h*:

typedef frs_fsched_info_t	A structure containing information about one scheduler, including its CPU number, time base, and number of minor frames. Used when creating a Frame Scheduler.
typedef frs_t	A structure containing an frs_fsched_info_t and the process ID of the frs control of the master Frame Scheduler. Used to create or specify any Frame Scheduler.
typedef frs_queue_info_t	A structure containing information about one activity process: the Frame Scheduler and minor frame it uses and its scheduling discipline. Used when enqueuing a process.
typedef frs_recv_info_t	A structure containing error recovery options.

Library Interface for C Programs

The API library functions in */usr/lib/libfrs.so* are summarized in Table 7-1 on “Library Interface for C Programs” on page 99 for convenient reference.

Table 7-1 Frame Scheduler Operations

Operation	Application Interface Options
Create a Frame Scheduler	frs_t* **frs_create(**int *cpu*, int *intr_source*, int *intr_qualifier*, int *n_minors*, pid_t *sync_master_pid*, int *num_slaves***);** frs_t* **frs_create_master(**int *cpu*, int *intr_source*, int *intr_qualifier*, int *n_minors*, int *num_slaves***);** frs_t* **frs_create_slave(**int *cpu*, frs_t* *sync_master_frs***);**
Enqueue an activity process to a Frame Scheduler	int **frs_enqueue(**frs_t* *frs*, pid_t *pid*, int *minor_index*, uint *discipline***);**
Join a Frame Scheduler (activity is ready to start)	int **frs_join(**frs_t* *frs***);**
Start scheduling (all activities enqueued)	int **frs_start(**frs_t* *frs***);**
Yield control after completing activity	int **frs_yield(**void**);**
Pause scheduling at end of minor frame	int **frs_stop(**frs_t* *frs***);**
Resume scheduling at next time-base interrupt	int **frs_resume(**frs_t* *frs***)**:
Destroy a Frame Scheduler and send SIGKILL to its FRS control process	int **frs_destroy(**frs_t* *frs***);**
Interrogate a process queue	int **frs_getqueuelen(**frs_t* *frs*, int *minor_index***);** int **frs_readqueue(**frs_t* *frs*, int *minor_index*, pid_t* *pidlist***);**
Remove a process from a queue	int **frs_premove(**frs_t* *frs*, int *minor_index*, pid_t *remove_pid***);**
Reinsert a process in a queue, possibly changing discipline	int **frs_pinsert(**frs_t* *frs*, int *minor_index*, pid_t *insert_pid*, int *discipline*, pid_t *base_pid***);**
Retrieve error-recovery options	int **frs_getattr(** frs_t* *frs*, int *minor_index*, pid_t *pid*, frs_attr_t *att_index*, frs_recv_info_t* *options***);**
Set error-recovery options	int **frs_setattr(** frs_t* *frs*, int *minor_index*, pid_t *pid*, frs_attr_t *att_index*, frs_recv_info_t* *options***);**

System Call Interface for Fortran and Ada

Each Frame Scheduler function is available in two ways: as a system call to **schedctl()**, or as one or more library calls to functions in the *frs* library, */usr/lib/libfrs.so*. The system call is accessible from FORTRAN and Ada programs because both languages have bindings for **schedctl()** (see the schedctl(2) reference page). The correspondence between the library functions and **schedctl()** calls is shown in Table 7-2.

Table 7-2 Frame Scheduler schedctl() Support

Library Function	Schedctl Syntax
frs_create()	int **schedctl(**MPTS_FRS_CREATE, frs_info_t* *frs_info***)**;
frs_enqueue()	int **schedctl(**MPTS_FRS_ENQUEUE, frs_queue_info_t* *frs_queue_info***)**;
frs_join()	int **schedctl(**MPTS_FRS_JOIN, pid_t *frs_master***)**;
frs_start()	int **schedctl(**MPTS_FRS_START, pid_t *frs_master***)**;
frs_yield()	int **schedctl(**MPTS_FRS_YIELD**)**;
frs_stop()	int **schedctl(**MPTS_FRS_STOP, pid_t *frs_master***)**;
frs_resume()	int **schedctl(**MPTS_FRS_RESUME, pid_t *frs_master***)**;
frs_destroy()	int **schedctl(**MPTS_FRS_DESTROY, pid_t *frs_master***)**;
frs_getqueuelen()	int **schedctl(**MPTS_FRS_GETQUEUELEN, frs_queue_info_t* *frs_queue_info***)**;
frs_readqueue()	int **schedctl(**MPTS_FRS_READQUEUE, frs_queue_info_t* *frs_queue_info*, pid_t* *pidlist***)**;
frs_premove()	int **schedctl(**MPTS_FRS_PREMOVE, frs_queue_info_t* *frs_queue_info***)**;
frs_pinsert()	int **schedctl(**MPTS_FRS_PINSERT, frs_queue_info_t* *frs_queue_info*, pid_t **base_pid***)**;
frs_getattr()	int **schectl(**MPTS_FRS_GETATTR, frs_attr_info_t* *frs_attr_info***)**;
frs_setattr()	int **schectl(**MPTS_FRS_SETATTR, frs_attr_info_t* *frs_attr_info***)**;

Process Execution

An activity process that is enqueued to a Frame Scheduler has the basic structure shown in Example 7-1.

Example 7-1 Skeleton of an Activity Process

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

When the process is ready to start real-time execution, it calls **frs_join()**. This call blocks until all enqueued processes are ready and scheduling begins (see “Starting Multiple Schedulers” on page 105). When **frs_join()** returns, the process is running in its first minor-frame execution.

The process then performs whatever activity it is supposed to complete in each minor frame. When it completes that work, it calls **frs_yield()**. This gives up control of the CPU until the next minor frame in which the process is enqueued.

An activity process is never preempted. As long as it yields before the end of the frame, it can do its assigned work without interruption from other processes (it can be interrupted by hardware interrupts, if any hardware interrupts are allowed in that CPU). The Frame Scheduler preempts the process at the end of the minor frame.

Tip: Because an activity process cannot be preempted, it can often use global data without locks or semaphores. When the process that modifies a global variable is enqueued in a different minor frame from the processes that read the variable, there can be no access conflicts between them.

Conflicts are still possible between two processes that are queued to the same minor frame in different, synchronized Frame Schedulers. However, such processes are guaranteed to be running concurrently. This means they can use spin-locks (see “Locks” on page 33) with high efficiency.

Tip: When a very short minor frame interval is used, it is possible for a process to have an overrun error in its first frame due to cache misses. A simple variation on the basic structure shown in Example 7-1 is to spend the first minor frame touching a set of important data structures in order to “warm up” the cache (see “Reducing Cache Misses” on page 52). This is sketched in Example 7-2.

Example 7-2 Alternate Skeleton of Activity Process

```
/* 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 process is scheduled on more than one minor frame in a major frame, it can be designed to do nothing except warm the cache in the entire first major frame. To do this, the activity process function has to know how many minor frames it is scheduled on, and calls **frs_yield()** that many times in order to pass the first major frame.

Scheduling Within a Minor Frame

Processes in a minor frame queue are dispatched in queue order. Initially, queue order is the order in which processes are named in **frs_enqueue()** calls. (The queues can be reordered dynamically; see “Managing Activity Processes” on page 106.)

Scheduler Flags frs_run and frs_yield

The Frame Scheduler keeps two status flags per queued process, named *frs_run* and *frs_yield*. If a process is ready to run when its turn comes, it is dispatched and its *frs_run* flag is set to indicate that this process has run at least once within this minor frame.

When a process yields, its *frs_yield* flag is set to indicate that the process has released the processor. It will not be activated again within this minor frame.

If a process 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 processes that have not yielded and may have become ready to run. If no ready processes are found, the Frame Scheduler goes into idle mode until a process 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 process. If the *frs_run* flag has not been set, that process 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 process is a candidate for an *overrun* exception.

Whether these exceptions are declared depends on the scheduling discipline assigned to the process. Scheduling disciplines are explained under "Using the Scheduling Disciplines" on page 110).

At the end of a minor frame, the Frame Scheduler resets all *frs_run* flags, except for those of processes that use the Continuable discipline in that minor frame. For those processes, the residual *frs_yield* flags keeps the processes that have yielded from being dispatched in the next minor frame.

Underrun and overrun exceptions are typically communicated via IRIX signals. The rules for sending these signals are covered under "Using Signals Under the Frame Scheduler" on page 122.

Estimating Available Time

It is up to you to make sure that all the processes equeued 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 simplified by the fact that only the enqueued processes can execute in a CPU controlled by the Frame Scheduler. You need to estimate the maximum time each process can consume between one call to **frs_yield()** and the next.

Frame Scheduler processes do compete for CPU cycles with I/O interrupt service in the same CPU. If you direct I/O interrupts away from the CPU (see "Isolating a CPU From Sprayed Interrupts" on page 81 and "Assigning Interrupts to CPUs" on page 81), then the only competition for CPU cycles (other than a very few essential TLB interrupts) is the overhead of the Frame Scheduler itself, and it has been carefully optimized for least 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 (see “Maximum Response Time Guarantee” on page 87) and allow for it.

Using Multiple Synchronized Schedulers

When the activities of one frame cannot be completed by one CPU, you need to 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, which owns the time base and one CPU, and as many synchronized Frame Schedulers as you need, each managing an additional CPU. The synchronized schedulers take their time base from the master, so that all start minor frames at the same instant.

Each Frame Scheduler has its own queues of processes. A given process can be enqueued to only one CPU. (However, you could create multiple processes based on the same code, and enqueue 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 FRS.

A process can have the FRS control relationship to only one Frame Scheduler. In order to create multiple, synchronized Frame Schedulers, you must create a process to be the FRS controller of each one. Typically these will be lightweight processes created with **sproc()**.

Starting a Single Scheduler

A single Frame Scheduler comes into existence when the FRS control process calls **frs_create()**. Then the FRS controller calls **frs_enqueue()** one or more times to tell the new Frame Scheduler the PID values of the processes that it will schedule. The FRS controller calls **frs_start()** when it has enqueued all the processes. Each scheduled process must call **frs_join()** when it has initialized itself and is ready to be scheduled.

The Frame Scheduler requires the **frs_enqueue()** call for a given PID to precede the **frs_join()** call from the same PID. That is, an activity process cannot join the scheduler until the FRS controller has enqueued it—the **frs_join()** returns an error unless the calling process has been enqueued. After the Frame Scheduler receives the **frs_start()** call it waits until all enqueued processes have called **frs_join()**; then it begins the first minor frame.

Tip: A barrier provides a good way to coordinate the startup of a Frame Scheduler (see "Barriers" on page 34). When each activity process starts, it should wait at a barrier. The FRS controller, after performing all the enqueues, comes to the same barrier. The activity processes can then join the scheduler.

Note: In version 1.0, 1.1, and 2.0 of REACT/Pro (the versions used with IRIX prior to version 6.2), the Frame Scheduler allowed a process to join prior to the enqueue. This flexibility was removed in version 3.0 (for IRIX 6.2) in order to simplify the implementation and to improve performance.

Starting Multiple Schedulers

A Frame Scheduler cannot start dispatching activities until

- the FRS controller has enqueued all the activity processes to their minor frames
- all the enqueued processes have done their own initial setup and have joined.

When multiple Frame Schedulers are used, none can start until all are ready.

Each FRS controller tells its Frame Scheduler that it has enqueued all activities by calling **frs_start()**. Each activity process tells 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_enqueue()** calls, an appropriate number of **frs_join()** calls, and an **frs_start()** call. Each synchronized Frame Scheduler tells the master Frame Scheduler when it is ready. When all the schedulers are ready, the master Frame Scheduler gives the downbeat, and the first minor frame begins.

Pausing Frame Schedulers

Any Frame Scheduler can be made to pause and restart. Any process (typically but not necessarily the FRS controller) can call **frs_stop()**, specifying a particular Frame Scheduler. That scheduler continues dispatching processes 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.

Note: If there is a process running Background discipline in the current minor frame, it will continue to execute until it yields or is blocked on a system service.

Since a Frame Scheduler does not stop until the end of a minor frame, you can stop and restart a group of synchronized 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 Processes

The FRS control process creates the initial set of activity processes by calling **frs_enqueue()** prior to starting the Frame Scheduler. All the enqueued processes must call **frs_join()** before scheduling can begin. However, the FRS controller can change the set of activity processes dynamically while the Frame Scheduler is working, using the following functions:

frs_getqueuelen()	Get the number of processes currently in the queue for a specified minor frame.
f**rs_readqueue()**	Return the PID values of all queued processes for a specified minor frame as a vector of integers.
frs_premove()	Remove a process (specified by PID) from a minor frame queue.
frs_pinsert()	Insert a process (specified by PID and discipline) into a given position in a minor frame.

Using these functions, the FRS controller can change the queueing discipline of a process (by removing it and inserting it with a new discipline). The FRS controller can suspend a process by removing it from its queue; or can restart a process by putting it back in its queue.

Note: When an activity process is removed from the last or only queue it was in, it is returned to the normal IRIX scheduler and can begin to execute on some other CPU.When an activity process is removed from a queue, a signal may be sent to the removed process (see "Handling Signals in an Activity Process" on page 124). If a signal is sent to it, it will begin executing in its specified or default signal handler; otherwise, it will simply begin executing following **frs_yield()**. Once returned to the IRIX scheduler, a call to an FRS function such as **frs_yield()** returns an error (this also can be used to indicate the resumption of normal scheduling).

The FRS controller can also enqueue new processes that have not been scheduled before. The Frame Scheduler does not reject an **frs_pinsert()** call for a process that has not yet joined the scheduler. However, a process must call **frs_join()** before it can be scheduled.

If an enqueued process should be terminated for any reason, the Frame Scheduler removes the process from all queues in which it appears.

Selecting a Time Base

Your program specifies an interrupt source to be 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 available.

When you use multiple, synchronized Frame Schedulers, the master Frame Scheduler creates an *interrupt group*, a hardware mechanism that distributes the time-base interrupt to each synchronized CPU. This ensures that minor-frame boundaries are synchronized across all the Frame Schedulers. (For details of the interrupt group mechanism, you can read "Group Interrupts on Challenge and Onyx Systems," a technical paper distributed with the REACT/Pro product.)

On-Chip Timer Interrupt

Each processor chip contains a free-running timer that is used by IRIX for normal process scheduling. This timer is not synchronized between processors, so it cannot be used to drive multiple synchronized schedulers. The on-chip timer can be used as a time base when only one CPU is used and there is a reason to not use the high-precision timer described in the next topic.

To use the on-chip timer, specify FRS_INTRSOURCE_R4KTIMER as the interrupt source, and the minor frame interval in microseconds, to **frs_create()**.

High-Resolution Timer

The high-resolution timer and clock is a timer that is synchronous across all processors, and is ideal to drive synchronous schedulers. On Challenge and Onyx systems this timer is based on the high resolution counter discussed under "Hardware Cycle Counter" on page 40.

To use this timer, specify FRS_INTRSOURCE_CCTIMER, and the minor frame interval in microseconds, to **frs_create()**.

The IRIX kernel uses this timer for managing timer events. When your program creates the master Frame Scheduler, the Frame Scheduler migrates all timeout events to CPU 0, leaving the timer on the scheduled CPU free.

An interrupt group is not required to coordinate multiple Frame Schedulers when this time base is used. The high-resolution timers in all CPUs are synchronized automatically.

Vertical Sync Interrupt

An interrupt is generated for every vertical retrace by the graphics subsystem (see "Understanding the Vertical Sync Interrupt" on page 82). The frame rate will be either 50 Hz or 60 Hz, depending on the installed hardware. This interrupt is especially appropriate for a visual simulator, since it defines a frame rate that matches the graphics subsystem frame rate.

To use the vertical sync interrupt, specify FRS_INTRSOURCE_VSYNC to **frs_create()**. An error is returned if this system lacks a graphics subsystem.

When multiple synchronized schedulers are used, the master Frame Scheduler allocates an interrupt group to distribute the vertical sync interrupt.

External Interrupts

An external interrupt is generated via a signal applied to the external interrupt sockets on a Challenge or Onyx system (see "External Interrupts" on page 173). To use external interrupts as a time base, specify FRS_INTRSOURCE_EXTINTR to **frs_create()**.

When multiple synchronized schedulers are used, the master Frame Scheduler receives the interrupt, and allocates an interrupt group that is used to make the interrupt simultaneously available to the synchronized schedulers.

Note: External output signals can be generated by software using **ioctl()** to the external interrupt driver. An imaginative designer might think of connecting an external output jack to an external interrupt input jack on the same system, thus creating software-controlled external interrupts as an FRS time base. This would work in principle. However, if user process generating the interrupts are generated by a user process that makes any other system calls, there is a possibility of system deadlock.

Device Driver Interrupt

A user-written, kernel-level device driver can supply the time-base interrupt (see "The Frame Scheduler Device Driver Interface" on page 126). The Frame Scheduler allocates an interrupt group. The device driver must direct interrupts to it.

To use a device driver as a time base, specify FRS_INTRSOURCE_DRIVER and the device driver's identifying number, to **frs_create()**.

Software Interrupt

A programmed, software-generated interrupt can be used as the time base. Any user process can send this interrupt to the master Frame Scheduler by calling **frs_userintr()**.

Note: Software interrupts are primarily intended for application debugging. It is not feasible for a user process to generate interrupts with the kind of regularity that a real-time scheduler requires.

To use software interrupts as a time base, specify FRS_INTRSOURCE_USER to **frs_create()**.

Caution: The use of software interrupts has a potential for causing a system deadlock if the interrupt-generating process contends for a resource that is also used by a frame-scheduled activity process. If any activity process calls IRIX system functions, the only way to be absolutely sure of avoiding deadlock is for the interrupt-generating process to avoid using any IRIX system functions. Note that C library functions such as **printf()** invoke system functions, and can lead to deadlocks in this case.

Using the Scheduling Disciplines

When an FRS control process enqueues a process to a minor frame (using **frs_enqueue()**), it must specify a *scheduling discipline* that tells the Frame Scheduler how the process is expected to use its time within that minor frame.

Realtime Discipline

In the simplest case, an activity process should start during the minor frame to which it is queued, and should complete its work and yield within the same minor frame.

If the process is not ready to run (for example, is blocked on I/O) during the entire minor frame, an *underrun* exception is said to occur. If the process fails to complete its work and yield within the minor frame interval, an *overrun* exception is said to occur.

The Frame Scheduler calls this strict discipline the Realtime scheduling discipline.

The simplest case of a Frame Scheduler would consist of

- one minor frame per major frame—the time base is also the frame rate
- one or more activities enqueued to the frame with Realtime discipline

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

However, realistic designs need the flexibility to have processes that

- need not start every frame; for instance, processes 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 Realtime and with each other, to allow these variations.

Background Discipline

The Background discipline is mutually exclusive with the other disciplines. The Frame Scheduler only dispatches a Background process when all other processes queued to that minor frame have run and have yielded. Since the Background process 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 process with the Background discipline must be queued to its frame following all non-Background processes. Do not queue a real-time process after a Background process.

Underrunable Discipline

You specify Underrunable discipline with Realtime discipline to prevent detection of underrun exceptions. You specify Underrunable in two cases:

- When a process needs to run only when some event has occurred such as a lock being released or a semaphore being posted.
- When a process may need more than one minor frame (see "Using Multiple Consecutive Minor Frames" on page 112).

When you specify Realtime+Underrunable, the process 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 Overrunnable discipline with Realtime discipline to prevent detection of overrun exceptions. You specify it in two cases:

- When it truly does not matter if the process fails to complete its work within the minor frame—for example, a calculation of a game strategy which, if it fails to finish, merely makes the computer a less dangerous opponent.
- When a process may need more than one minor frame (see "Using Multiple Consecutive Minor Frames" on page 112).

When you specify Overrunnable+Realtime, the process is not required to call **frs_yield()** before the end of the frame. Even so, the process 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 enqueued. At that time it resumes where it was preempted, with no indication that it was preempted.

Continuable Discipline

You specify Continuable discipline with Realtime 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 102).

The result is that, if the process 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 process execute just once in a block of consecutive minor frames.

Using Multiple Consecutive Minor Frames

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

You must decide the absolute maximum time the process could consume between starting up and calling **frs_yield()**. If this is unpredictable, or if it is predictably longer than the major frame, the process cannot be scheduled by the Frame Scheduler. It should probably run in another CPU under the IRIX scheduler.

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

The discipline settings for each frame should be:

First frame — Realtime + Overrunnable + Continuable—the process must start in this frame (not Underrunnable) but is not required to yield (Overrunnable). If it yields, it is not restarted in the following minor frame (Continuable).

Intermediate — Realtime+Underrunable+Overrunnable+Continuable—the process need not start (it might already have yielded, or might be blocked) but is not required to yield. If it does yield (or if it had yielded in a preceding minor frame), it is not restarted in the following minor frame (Continuable).

Final frame — Realtime+Underrunable—the process need not start (it might already have yielded) but if it starts, it must yield in this frame (not Overrunnable). The process can start a new run in the next minor frame to which it is queued (not Continuable).

A process can be enqueued for one or more of these multi-frame 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 process that should run at a rate of 5 Hz and can use up to 3/60 second at each dispatch. You would enqueue the process to 5 sequences of 3 consecutive frames each. It would start in frames 0, 12, 24, 36, and 48. Frames 1, 13, 25, 37 and 49 would be intermediate frames, and 2, 14, 26, 38 and 50 would be final frames.

Preparing the System

Before a real-time program executes, you must set up the system in the following ways.

1. Choose the CPU or CPUs that the real-time program will use. CPU 0 (at least) must be reserved for IRIX system functions.
2. Decide which CPUs will handle I/O interrupts. By default, IRIX distributes I/O interrupts across all available processors as a means of balancing the load (referred to as *spraying interrupts*). CPUs that are used for real-time programs should be removed from the distribution set (see “Assigning Interrupts to CPUs” on page 81).
3. Make sure that none of the real-time CPUs is managing the clock (see “Assigning the Clock Processor” on page 79). Normally the responsibility of handling 10ms scheduler interrupts is given to CPU 0.
4. Make sure none of the real-time CPUs is handling the fast timer (“Assigning the fasthz Processor” on page 79). This responsibility is typically given to CPU 0 along with all other housekeeping.

Each Frame Scheduler takes care of restricting and isolating its CPU, so that the CPU is used only be processes scheduled by the Frame Scheduler.

Implementing a Single Frame Scheduler

When the activities of your real-time program can be handled within a major frame interval by a single CPU, your program needs to create only one Frame Scheduler.

Typically your program has a top-level process (called the master process here) to handle start-up and termination, and one or more activity processes that are dispatched by the Frame Scheduler. The activity processes are typically lightweight processes created using **sproc()**, but that is not a requirement—the activity processes can be created with **fork()**, and they need not be children of the master process. (See for instance "Example of Scheduling Separate Programs" on page 227.)

In general, these are the steps that the master process follows:

1. Initialize global resources such as memory-mapped segments, memory arenas, files, asynchronous I/O, and other resources.
2. Lock the address space segments shared by activity processes (see "Locking Pages in Memory" on page 49). (When **fork()** is used, each child process must lock its own address space.)
3. Create the Frame Scheduler using **frs_create_master()** (see Table 7-1 on "Library Interface for C Programs" on page 99).
4. Change the Frame Scheduler signals or exception policy, if desired (see "Setting Frame Scheduler Signals" on page 124 and "Setting Exception Policies" on page 119).
5. Create the activity processes using **sproc()** or **fork()** or, if they are independent processes, get them started and obtain their PID values.
6. Use **frs_enqueue()** to queue each activity process to the queue or queues on which it is to run.

   Each activity process independently uses **frs_join()** to let the Frame Scheduler know it is ready to start real-time execution. This call must follow the **frs_enqueue()** call for that process. The call blocks until scheduling begins, then returns to start the first frame dispatch of each activity process.
7. Set up signal handlers for signals from the Frame Scheduler (see "Using Signals Under the Frame Scheduler" on page 122). The handlers are set at this time, after creation of the activity processes, so that the activity processes do not inherit them.
8. Use **frs_start()** (Table 7-1) to enable scheduling.

The Frame Scheduler begins scheduling processes as soon as all the activity processes have called **frs_join()**.

9. Wait for error and termination signals from the Frame Scheduler and for the termination of child processes.
10. Use **frs_destroy()** to terminate the Frame Scheduler.
11. Tidy up the global resources as required.

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.

Syncronized Scheduler Concepts

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

No single process may create more than one Frame Scheduler. This is because every Frame Scheduler must have a unique FRS control process to which it can send signals. As a result, the program will have three types of processes:

- a master process that sets up global data and creates the master Frame Scheduler
- one FRS control process for each sync-slave Frame Scheduler
- activity processes

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

Sync-slave schedulers can be stopped and restarted independently. However, when any scheduler, master or slave, is destroyed, all are immediately destroyed.

Synchronized Schedulers: the Sync-Master Process

A variety of program designs is possible but the simplest is possibly the set of processes described in the following paragraphs.

The master process executes first and performs these steps:

1. Initialize global resources such as memory-mapped segments, memory arenas, files, asynchronous I/O, and other resources. One global resource is the process ID of the master process.
2. Lock the address space shared with lightweight processes.
3. Create the sync-master Frame Scheduler using the call **frs_create_master()**, and store its handle in a global location.
4. Create one FRS control process for each synchronized CPU to be used.
5. Create the activity processes that will be scheduled by the master Frame Scheduler and use **frs_enqueue()** to enqueue them to their assigned minor frames.
6. Set up signal handlers for signals from the Frame Scheduler (see "Using Signals Under the Frame Scheduler" on page 122).
7. Use **frs_start()** (Table 7-1) to tell the master Frame Scheduler that its activity processes are all enqueued.

   The master Frame Scheduler will start scheduling processes as soon as all processes have called **frs_join()** for their respective schedulers.
8. Wait for termination or error signals.
9. Use **frs_destroy()** to terminate the master Frame Scheduler.
10. Tidy up global resources as required.

Synchronized Schedulers: Sync-Slave Processes

Each FRS control process for a synchronized scheduler (a sync-slave) will:

1. Create a synchronized Frame Scheduler using **frs_create_slave()**, specifying information about the master Frame Scheduler stored by the master process. The sync-master must exist. A sync-slave must specify the same time base and number of minor frames as the sync-master.
2. Change the Frame Scheduler signals or exception policy, if desired (see “Setting Frame Scheduler Signals” on page 124 and “Setting Exception Policies” on page 119).
3. Create the activity processes that will be scheduled by this synchronized Frame Scheduler, and use **frs_enqueue()** to enqueue them to their assigned minor frames.
4. Set up signal handlers for signals from the synchronized Frame Scheduler.
5. Use **frs_start()** to tell the synchronized Frame Scheduler that all activity processes have been enqueued.

   The sync-slave notifies the master Frame Scheduler when all processes have called **frs_join()**. When the master Frame Scheduler starts broadcasting interrupts, scheduling will begin.
6. Wait for termination or error signals.
7. Use **frs_destroy()** to terminate the synchronized Frame Scheduler.

For an example of this kind of program structure, refer to “Examples of Multiple Synchronized Schedulers” on page 229.

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

Tip: A program of this type is place where you might use a barrier (see “Barriers” on page 34). For example, you want to be sure that all Frame Schedulers are successfully created before you start to create activity processes. The master and the FRS control processes could rendezvous at a barrier after creating their schedulers, but before starting to create activity processes or enqueue them.

Handling Frame Scheduler Exceptions

The FRS control process for a scheduler controls the handling of the Overrun and Underrun exceptions. It can specify how these exceptions should be handled, and what signals the Frame Scheduler should send. These policies have to be set before the scheduler is started. While the scheduler is running, the FRS controller can query the number of exceptions that have occurred.

Exception Types

The Overrun exception indicates that a process 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 process is queued.

The Underrun exception indicates that a process that should have started in a minor frame did not start. Possibly the process has terminated. More likely it was blocked in some kind of wait because of an unexpected delay in I/O, or a deadlock on a lock or semaphore.

Exception Handling Policies

The FRS control process can establish one of four policies for handling overrun and underrun exceptions. When it detects an exception, the Frame Scheduler can:

- Send a signal to the FRS 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

The default action is to send a signal (the specific signals are listed under “Setting Frame Scheduler Signals” on page 124). The scheduler continues to run. The FRS control process can then take action, for example, terminating the Frame Scheduler.

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 processes 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 process to wake up. Because exactly one frame is inserted, all other processes 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 an on-chip or high-resolution timer (see "Selecting a Time Base" on page 107).

When adding time, the current frame is made longer by a fixed amount of time. Since the minor frame becomes a variable length, it is possible for the Frame Scheduler to drop out of synch 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 processes queued to that following frame can finish their work in a reduced amount of time. If they do not, the Frame Scheduler will steal time from the next frame still.

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_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_setattr()** accepts arguments for minor frame number and process ID; however it currently only allows setting exception policies for all policies and all minor frames. The most significant argument to it is the *frs_recv_info* structure, declared with these 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/sys/frs.h*. The function in Example 7-3 sets the policy of injecting a repeat frame. The caller specifies only the Frame Scheduler and the number of consecutive exceptions allowed.

Example 7-3 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 7-4 sets the policy of stretching the current frame (a function to set the policy of stealing time from the next frame would be nearly identical). The caller specifies the Frame Scheduler, the number of consecutive exceptions, and the stretch time in microseconds.

Example 7-4 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 FRS control process can query for counts of exceptions. This is done with a call to **frs_getattr()**, passing the handle to the Frame Scheduler, the number of the minor frame, and the process ID of the process 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 process in that minor frame. In order to find out the count of all overruns in a given minor frame, you must sum the counts for all processes queued to that frame. If a process is queued to more than one minor frame, separate counts are kept for it in each frame.

The function in Example 7-5 takes a Frame Scheduler handle and a minor frame number. It gets the list of process IDs queued to that that minor frame, and returns the sum of all exceptions for all of them.

Example 7-5 Function to Return a Sum of Exception Counts

```
#define THE_MOST_PIDS 250
int
totalExcepts(frs_t * theFRS, int theMinor)
{
    int numPids = frs_getqueuelen(theFRS, theMinor);
    int j, sum;
    pid_t allPids[THE_MOST_PIDS];
    if ( (numPids <= 0) || (numPids > THE_MOST_PIDS) )
        return 0; /* invalid minor #, or no procs queued? */

    if (!frs_readqueue(theFRS, theMinor, allPids))
        return 0; /* unexpected problem with reading IDs */

    for (sum = j = 0; j<numPids; ++j)
    {
        frs_overrun_info_t work;
        frs_getattr(theFRS,             /* the scheduler */
                    theMinor,          /* the minor frame */
                    allPids[j],        /* the process */
                    FRS_ATTR_OVERRUNS, /* want counts */
                    &work);            /* put them here */
```

```
        sum += (work.overruns + work.underruns);
    }
    return sum;
}
```

Tip: If a function such as the one in Example 7-5 is to be called frequently, it is a good idea to prepare the arrays of process IDs once and save them. The repeated calls to **frs_getqueuelen()** and **frs_readqueue()** can be avoided.

Using Signals Under the Frame Scheduler

The Frame Scheduler itself sends signals to the processes using it. And processes can communicate by sending signals to each other. In brief, an FRS sends signals to indicate that

- The FRS has been terminated
- Overrun or underrun have been detected
- A process has been dequeued

The rest of this topic details how to specify the signal numbers and how to handle the signals.

Signal Delivery and Latency

When a process is scheduled by the IRIX kernel, it receives a pending signal the next time the process exits from the kernel domain. For most signals, this could occur

- when the process is dispatched after a wait or preemption
- upon return from some system call
- upon return from the kernel's usual 10-millisecond tick interrupt

(SIGALRM is delivered as soon as the kernel is ready to return to user processing after the timer interrupt, in order to preserve timer accuracy.) Thus, for a process that is ready to run, in a CPU that has not been made nonpreemptive, normal signal latency is at most 10 milliseconds, and SIGALARM latency is less. However, when the receiving process is not ready to run, or when there are competing processes with superious priorities, the delivery of a signal is delayed until the next time the receiving process is scheduled.

When the CPU is nonpreemptive (see "Making a CPU Nonpreemptive" on page 86), there are no clock tick interrupts, so signals can only be delivered following a system call.

Signal latency can be greater when running under the Frame Scheduler. Like the normal IRIX scheduler, the Frame Scheduler delivers pending signals to a process when it next returns to the process from the kernel domain. This can occur

- when the process is dispatched at the start of a minor frame where it is enqueued
- upon return from some system call

The upper bound on signal latency in this case is the interval between the minor frames to which that process is queued. If the process is scheduled only once in a major frame, it might not receive a signal until a full major frame interval after the signal is sent.

Handling Signals in the FRS Controller

When a Frame Scheduler detects an Overrun or Underrun exception that it cannot recover from, and when it is ready to terminate, it sends a signal to the FRS control process.

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

The FRS control process 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. An FRS controller for a synchronized scheduler should use the default action for SIGHUP (Exit) so that completion of the **frs_destroy()** quietly terminates the FRS controller.

The FRS controller for the master (or only) Frame Scheduler should catch Underrun and Overrun exceptions, report them, and shut down its scheduler.

When an FRS is terminated with **frs_destroy()**, it sends SIGKILL to its FRS control process. This cannot be changed; and SIGKILL cannot be handled. Hence **frs_destroy()** is equivalent to termination for the FRS control process. (In the first release, the FRS sent SIGHUP, but this made deadlocks possible and had to be given up.)

Handling Signals in an Activity Process

A Frame Scheduler can send a signal to an activity process when the process is removed from any queue using **frs_premove()** (see “Managing Activity Processes” on page 106). The scheduler can also send a signal to an activity process when it is removed from the last or only minor frame to which it was enqueued (at which time a process is returned to normal IRIX scheduling).

In order to have these signals sent, the FRS controller must set nonzero signal numbers for them, as discussed in the following topic, “Setting Frame Scheduler Signals.”

Setting Frame Scheduler Signals

The frame scheduler sends signals to the FRS control process.

Note: In earlier versions of REACT/Pro, the Frame Scheduler sent these signals to *all* processes queued to that Frame Scheduler as well as the FRS controller. That is no longer the case. You can remove signal handlers for these signals from activity processes, if they exist.

The signal numbers used for most events can be modified. The signal numbers can be queried using **frs_getattr(**FRS_ATTR_SIGNALS**)** and changed using **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 7-3

Table 7-3 Signal Numbers Passed in frs_signal_info_t

Field Name	Signal Purpose	Default Signal Number
sig_underrun	Notify FRS controller of Underrun.	SIGUSR1
sig_overrun	Notify FRS controller of Overrun.	SIGUSR2
sig_dequeue	Notify an activity process that it has been dequeued with **frs_premove()**.	0 (do not send)
sig_unframesched	Notify an activity process that it has been removed from the last or only queue in which it was enqueued.	SIGRTMIN

Signal numbers must be changed before the Frame Scheduler is started. All the numbers must be specified to **frs_setattr()**, so the proper way to set any number is to first file the *frs_signal_info_t* using **frs_getattr()**. The function in Example 7-6 sets the signal numbers for Overrun and Underrun from its arguments.

Example 7-6 Function to Set Frame Scheduler Signals

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

Using Timers with the Frame Scheduler

In general, interval timers and the Frame Scheduler do not mix. The expiration of an interval is marked by a signal. However, signal delivery to an activity process can be delayed (see “Signal Delivery and Latency” on page 122), so timer latency is unpredictable.

An FRS control process, because it is scheduled by IRIX, not the Frame Scheduler, can use interval timers. However, it has a more reliable time base available in the activity processes it creates. The FRS controller can create a global semaphore on which it waits with **uspsema()** (see “IRIX Semaphores” on page 32). The minimal activity process shown in Example 7-7 can be enqueued to one or more minor frames to provide a repeating interval at any multiple of the major-frame interval.

Example 7-7 Minimal Activity Process as a Timer

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

The Frame Scheduler Device Driver Interface

The Frame Scheduler provides a device driver interface to allow any device with a kernel-level device driver to generate the time-base interrupt. As many as 16 different device drivers can support the Frame Scheduler in any one Challenge/Onyx system. The Frame Scheduler distinguishes device drivers by an ID number in the range 0 through 15 that is coded into each driver.

Note: The structure of an IRIX kernel-level device driver is discussed in the *IRIX Device Driver Programming Guide* (see “Other Useful Books” on page xxiii). The generation of time-base signals can be added as a minor enhancement to a existing device driver.

In order to interact with the Frame Scheduler, a driver provides two routines, one for initialization and one for termination, which it exports during driver initialization. After a master Frame Scheduler has initialized a device driver, the driver calls a Frame Scheduler entry point to signal the occurrence of each interrupt.

Device Driver Overview

The following sequence of actions occurs when a device driver is used as a source of time-base interrupts for the Frame Scheduler.

1. During its initialization in the *pfx***start()** or *pfx***init()** entry point, the driver calls a kernel function to report its *pfx***_frs_func_set()** and *pfx***_frs_func_clear()** functions, and to specify a unique driver identifier between 0 and 15. After this has been done, the Frame Scheduler is aware of the existence of this driver, and will allow programs to request it as the source of interrupts.
2. Later, a real-time program creates a master Frame Scheduler and specifies this driver (by number) as the source of interrupts. The Frame Scheduler verifies that this driver has exported the two functions. Then it calls *pfx***_frs_func_set(***intrgroup***)** for this particular driver. This tells the driver that time signals are needed.
3. The device driver calls **frs_handle_driverintr()** each time its interrupt handling routine is entered. This informs the Frame Scheduler that an interrupt has been received.
4. When the Frame Scheduler is being terminated, it invokes *pfx***_frs_func_clear()** for the associated driver. This tells the driver that time signals are no longer needed, and to cease calling **frs_handle_driverintr()** until it is again initialized by a Frame Scheduler.

The *pfx* in function names is the name of the loadable device driver as specified in its *master.d* file. Device driver names, device driver structure, configuration files, and related topics are covered in the *IRIX Device Driver Programming Guide*.

Exporting the Initialization and Termination Functions

A device driver must export the Frame Scheduler interface functions to make them known to the Frame Scheduler. This call, which occurs during the device driver's own initialization, also makes the driver known as a source of time-base interrupts:

```
frs_driver_export( int frs_driver_id,
                   void (*frs_func_set)(intrgroup_t*),
                   void (*frs_func_clear)(void));
```

A typical call resembles the code in Example 7-8.

The parameter *frs_driver_id* is the driver's identification number. A real-time program specifies the same number to **frs_create_master()** in order to select this driver as the source of interrupts. The identifier is an integer between 0 and 7. Different drivers in the same system must use different identifiers.

Example 7-8 Exporting Device Driver Entry Points

```
/*
** Function called by the example driver to export
** its Frame Scheduler interface functions.
*/
frs_driver_export(3, example_frs_func_set, example_frs_func_clear);
```

Frame Scheduler Initialization Function

The device driver must provide a function with the following prototype:

`void` *pfx*`_frs_func_set ( intrgroup_t* intrgroup ) ;`

A skeleton of an initialization function is shown in Example 7-9. The function is called by a new master Frame Scheduler that is created with an interrupt source parameter of FRS_INTRSOURCE_DRIVER and an interrupt qualifier specifying this device driver's number (see "Device Driver Interrupt" on page 109). A device driver is used by only one Frame Scheduler at a time.

The argument *intrgroup* is passed by the Frame Scheduler to identify the interrupt group it has allocated. A VME device driver must set the hardware devices it manages so that interrupts are directed to this interrupt group (see the paper "Group Interrupts on Challenge and Onyx Systems" distributed with REACT/Pro). The actual group identifier may be obtained using the macro:

```
intrgroup_get_groupid(intrgroup)
```

The effective destination may be obtained using the following macro:

```
EVINTR_GROUPDEST(intrgroup_get_groupid(intrgroup))
```

Example 7-9 Device Driver Initialization Function

```
/*
** Frame Scheduler initialization function
** for the External Interrupts Driver
*/
int FRS_is_active = 0;
int FRS_vme_install = 0;
void
example_frs_func_set(intrgroup_t* intrgroup)
{
   int s;
```

```
   ASSERT(intrgroup != 0);
   /*
   ** Step 1 (VME only):
   ** In a VME device driver, set up the hardware to send
   ** the interrupt to the appropriate destination.
   ** This is done with vme_frs_install() which takes:
   ** * (int) the VME adapter number
   ** * (int) the VME IPL level
   ** * the intrgroup as passed to this function.
   */
   FRS_vme_install = vme_frs_install(
      my_edt.e_adap, /* edt struct from example_edtinit */
      ((vme_intrs_t *)my_edt.e_bus_info)->v_brl,
      intrgroup);
   /*
   ** Step 2: any hardware initialization required.
   */
   /*
   ** Step 3: note that we are now in use.
   */
   FRS_is_active = 1;
}
```

Only VME device drivers need to call **vme_frs_install()**. As suggested by the code in Example 7-9, the arguments to **vme_frs_install()** can be taken from data supplied at boot time to the device driver's *pfx***edtinit()** function:

- the adapter number is in the *edt.e_adap* field
- the configured interrupt priority level is in the *vme_intrs.v_brl* addressed by the *edt.e_bus_info* field

The *pfx***edtinit()** entry point is documented in the *IRIX Device Driver Programming Guide*.

Tip: The **vme_frs_install()** function is a dynamic version of the VECTOR configuration statement. You are not required to use the IPL value from the configuration file.

Frame Scheduler Termination Function

The device driver must provide a function with the following prototype:

```
void prfx_frs_func_clear ( void ) ;
```

A skeleton for this function is shown in Example 7-10. The Frame Scheduler that initialized a device driver calls this function when the Frame Scheduler is terminating. The Frame Scheduler deallocates the interrupt group to which interrupts were directed.

The device driver should clean up data structures and make sure that the device is in a safe state. A VME device driver must call **vme_frs_uninstall()**.

Example 7-10 Device Driver Termination Function

```
/*
** Frame Scheduler termination function
*/
void
example_frs_func_clear(void)
{
   /*
   ** Step 1: any hardware steps to quiesce the device.
   */

   /*
   ** Step 2 (VME only):
   ** Break the link between interrupts and the interrupt
   ** group by calling vme_frs_uninstall() passing:
   ** * (int) the VME adapter number
   ** * (int) the VME IPL level
   ** * the value returned by vme_frs_install()
   */
   vme_frs_uninstall(
      my_edt.e_adap, /* edt struct from example_edtinit */
      ((vme_intrs_t *)my_edt.e_bus_info)->v_brl,
      FRS_vme_install);
   /*
   ** Step 3: note we are no longer in use.
   */
   FRS_is_active = 0;
}
```

Generating Interrupts

A driver has to call the Frame Scheduler interrupt handler from within the driver's interrupt handler using code similar to that shown in Example 7-11. This handler is entered concurrently on each CPU where the master or a synchronized Frame Scheduler is running. It delivers the interrupt to the Frame Scheduler on that CPU. The function to be invoked is

```
void frs_handle_driverintr(void);
```

It is possible for an interrupt handler to be entered at a time when the Frame Scheduler for its processor is not active; that is, after **frs_destroy()** has been called and before the driver termination function has been entered. The **frs_handle_driverintr()** function checks for this and does nothing when nothing is required.

Example 7-11 Generating an Interrupt From a Device Driver

```
void example_intr()
{
   /*
   ** Step 1: anything required by the hardware
   */
   /*
   ** Step 2: if connected to the Frame Scheduler, send
   ** an interrupt to it. Flag FRS_is_active is set in
   ** Example 7-9 and cleared in Example 7-10.
   */
   if (FRS_is_active) frs_handle_driverintr();
   /*
   ** Step 3: any additional processing needed.
   */
   return;
}
```