The Real-time Frame Scheduler in IRIX Programmer's Guide

CONTRIBUTORS

Written by Luis Stevens
Edited by David Cortesi
Illustrated by Luis Stevens

The Real-time Frame Scheduler in IRIX Programmer’s Guide

Contents

What is Frame Scheduling

A frame scheduler is a process execution manager that schedules processes in a predefined, cyclic sequence. The scheduling period, or *frame*, is determined by a repetitive time base, usually a processor interrupt.

The REACT/Pro Frame Scheduler (FRS) provides this basic scheduling service, but extends it with additional features to create a powerful development framework for real-time applications, allowing real-time application developers to more easily exploit the PowerPath-2 architecture.

IRIX Process Scheduling

The normal IRIX scheduler is flexible and tunable. Real-time executives can use IRIX system calls to set real-time priorities, to isolate and reserve processors, and to achieve low-overhead synchronization of multiple processes. (These standard kernel features are described in the REACT Technical Report.) However, the designer of a real-time application is required to understand and integrate a variety of privileged system calls while creating a multiprocessor scheduling mechanism—work that is at best a distraction from the real task of designing a high-performance real-time program.

The REACT/Pro Frame Scheduler integrates the IRIX scheduler features and extends them, providing an efficient and easy-to-use framework for real-time application development.

Operation of the FRS

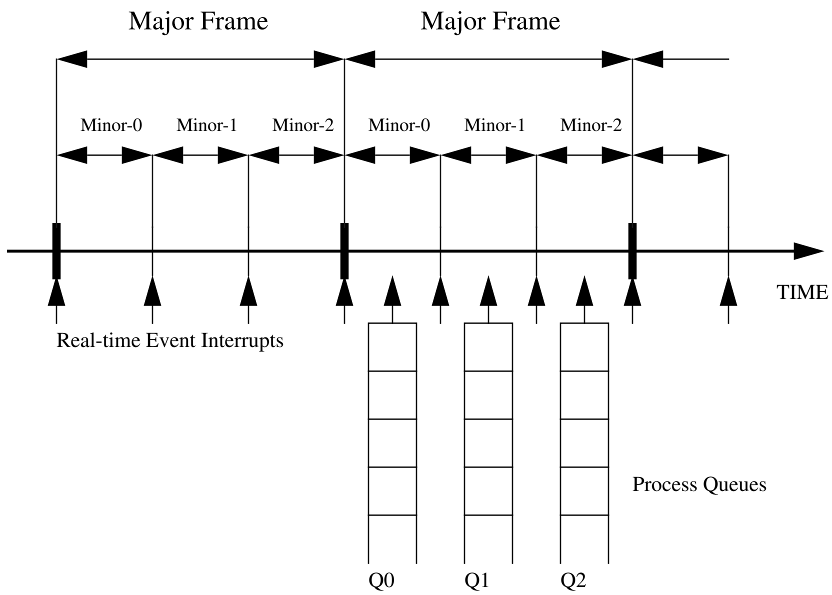
A single FRS is a kernel object that cyclicly schedules a set of processes, according to program-defined rules, on a specific processor. The FRS supercedes the IRIX scheduler, so that only FRS-scheduled processes run on that processor.

The application designer determines how many processors are required to meet the performance requirements, and initiates one FRS for each processor to be used.

Major and Minor Frames

Within one FRS, each scheduling cycle is initiated by a real-time event. These repetitive events divide time into intervals called *minor frames*. A fixed number of minor frames is defined as a *major frame*. The major frame is the basic repetitive unit.

Each minor frame has a private process queue listing all the processes to be run within the minor frame interval, as shown in the following figure.



Within a minor frame, processes are dispatched in the order that they appear in the queue. (Normal Irix process priorities are ignored by the FRS.)

Using Minor Frames

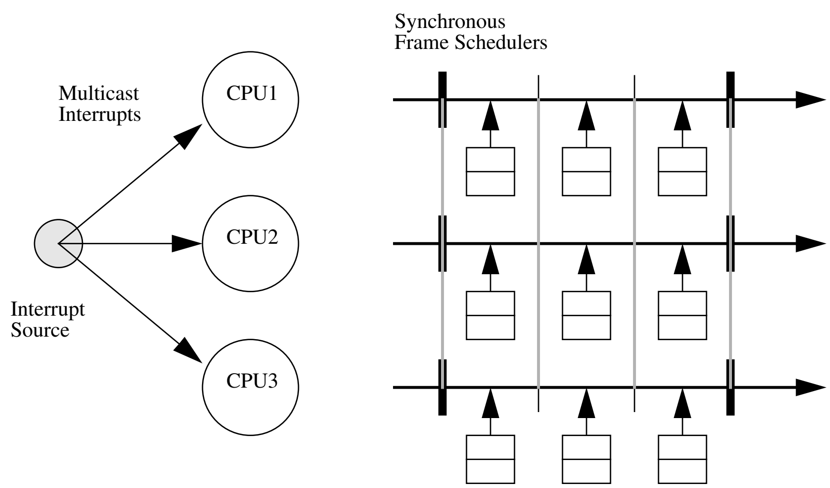
Processes are enqueued to one or more minor frames. The queues they appear on, and their order in those queues, determines their relative execution sequence within the major frame. Processes can be queued to more than one queue, so as to be scheduled more than once in a major frame.

A process is enqueued under one of a variety of scheduling disciplines (described later), allowing even more flexibility.

Using Multiple Processors

Creating an FRS establishes its control over a single processor. An FRS may be invoked on all but one of the processors in a Challenge or Onyx system. (One processor, usually processor 0 by convention, must be left available to run daemons and other normal UNIX housekeeping processes.)

Once invoked, an FRS has full control over its processor. For multiprocessor applications, the programmer creates one FRS per processor, and defines a master-slave relationship among the FRSs. The interrupts that define the scheduling time base are broadcast to the FRSs by an *interrupt group*, implemented using the interrupt multicasting capability available on the Challenge (see the paper "Group Interrupts on Challenge and Onyx Systems," distributed with REACT/Pro) to keep them in synchrony.



Defining the Time Base

Synchronization to any interrupt source on a Challenge or Onyx system is possible, including synchronization to:

- A specified frame interval measured by a high-precision clock that is synchronized in all processors
- An external interrupt (refer to the ei(7) man page for details on external interrupt support)
- The vertical retrace interrupt from the graphics subsystem
- A call generated by a kernel-level device driver
- Program-supplied interrupts from a program-defined time base (most often used for debugging)

The master FRS sets up an interrupt group, to synchronize the slave FRSs.

Programming with the Frame Scheduler

This section summarizes the process of designing and implementing a real-time application based on the REACT/Pro Frame Scheduler.

Designing with the Frame Scheduler

Use of the Frame Scheduler permits the designer to plan in terms of a set of abstract activities that must be completed in each major frame. A typical program would consist of a parent process that initializes resources; creates the needed FRSs; then creates each activity in the form of a lightweight process started with sproc(2) or with fork(2).

The first step of each child process would be to join its assigned Frame Scheduler. Thereafter each child process is dispatched by the FRS at the appropriate time within a major frame, when it runs until it yields control.

To take advantage of this model, the designer would:

- Specify the real-time program in terms of logically-independent abstract activities, noting which must run serially and which could run concurrently.
- Estimate the execution time of each activity.
- Allocate the activities to as few processors as possible, based on the execution times and the possibilities for concurrent execution.
- Plan the frame schedule in terms of the interrupt source to use, the number of minor frames per major frame, and the queue that each process should join.

Frame Scheduling Disciplines

In the simplest case, the process that represents an activity 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* error is said to occur. If the process fails to complete its work and yield within the minor frame interval, an *overrun* error is said to occur.

The Frame Scheduler calls this strict discipline the Realtime scheduling discipline. However, a number of other disciplines are supported for flexibility of design.

Underrunnable
: Underrun errors are ignored but overrun errors are detected. Underrunable processes are allowed to skip frames; however, if they do run they must yield before the end of the frame, or they cause an overrrun error.

Overrunnable
: Overrun errors are ignored but underrrun errors are detected. Overrunable processes must execute on every frame to which they are queued, but they are not required to yield before the end of the frame.

Underrunnable + Overrunnable
: Neither error is detected. These processes can skip frames and overrun them.

Overrunnable + Continuable
: This discipline is used to allow a process to extend its execution beyond a single minor frame.

Background
: A process with Background discipline runs only when all Realtime processes in its queue have yielded, that is, only when the processor would otherwise be idle. Underrun and overrun errors are not relevant to this discipline.

Using the Background Discipline

The background discipline is mutually exclusive with the other discipline types. 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.

Multiple Executions In A Major Frame

A process can be scheduled to run and yield more than once in a major frame, by queueing it to more than one minor frame.

Note: A process cannot be queued for dispatch by more than one FRS. A process can be dispatched in multiple minor frames, but on only one processor. To run the same process on multiple FRSs at once, start multiple instances of the process.

Using Multiple Consecutive Minor Frames

The Continuable discipline is used for a process that may require more than one minor frame to complete its work. To allow a process to use multiple consecutive minor frames, it must be enqueued in each of the minor frames in which it can run. The discipline settings for each frame should be:

First frame — Realtime + Overrunnable + Continuable—the process must start in this frame (not Underrunable) but is not required to yield (Overrunable). If it yields, it will not be 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).

In general terms, the Continuable flag allows the programmer to alter the effective length of a minor frame for one particular process. 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, starting in frames 0, 12, 24, 36, and 48. The disciplines in each of the 3-frame sequences would be those shown in the preceding list.

In implementation dependent terms, the Continuable flag allows the programer to control the place where the *frs_run* and the *frs_yield* flags are reset (see “Scheduling Within A Minor Frame” on page 17).

Preparing the System

The Challenge or Onyx processors that will be used by FRSs must be prepared for use by a real-time program. The designer must:

- Select the group of processors on the machine that will be used for real-time execution. Processor 0 (at least) must be reserved for IRIX system functions.

- Specify which processors will handle I/O interrupts. By default, IRIX distributes I/O interrupts (referred to as *spraying*) across all available processors as a means of balancing the load. Real-time processors should be removed from the distribution set. This is done using the directive NOINTR: *cpu*# [*cpu*#]* in the /var/sysgen/system/irix.sm file. After modifying irix.sm, rebuild the kernel using the command /etc/autoconfig -vf.
- Make sure that none of the real-time processors is acting as the *clock processor.* See mpadmin(1). This is the processor that fields 10ms scheduler interrupts, and the responsibility would typically be given to processor 0.
- Make sure none of the real-time processors is acting as the *fasthz processor*. See mpadmin(1). This is the processor that is interrupted at the rate set by the FASTHZ kernel parameter when high-precision itimers are in use. If fast itimers are used by the real-time program, it would be best to give the fasthz interrupt to a different processor than 0, provided there is another processor not commited to real-time use.

Implementing a Single FRS

The general steps that a real-time program needs to carry out are as follows. This model assumes a single FRS.

1. The parent process initializes any global resources such as memory-mapped segments, memory arenas, files and other objects.
2. The parent process locks down the address space it will share with lightweight processes—see plock(2) and mpin(2).
3. The parent process creates the Frame Scheduler using the call **frs_create_master()** (see page 20).
4. The parent process creates the activity processes using either **sproc()**—preferred—or **fork()**. (If **fork()** is used, each child process must lock its own address space.)
5. The parent uses **frs_enqueue()** to queue each child to the FRS queue or queues on which it is to run (see page 22).

6. Each process uses **frs_join()** (page 24) to let the FRS know the process is ready to start real-time execution. This call blocks until frame scheduling begins, then returns during the first frame dispatch of the process.
7. The parent process uses **frs_start()** (page 25) to enable scheduling. The Frame Scheduler will start scheduling processes as soon as all processes have called **frs_join()**.
8. After completing its work for a minor frame, a process calls **frs_yield()** (page 26) to yield the processor until its next scheduled dispatch.
9. The parent process waits for error and termination signals from the Frame Scheduler and other processes.
10. To terminate frame scheduling, any process calls **frs_destroy()** (page 32).

In the typical real-time program the real-time processes are lightweight processes created with sproc(). However, this is not a restriction of the Frame Scheduler. The Frame Scheduler can manage processes created with fork(). Moreover, the real-time processes do not have to be child processes of the frs-master process. For an example, see “Example of Scheduling Separate Programs” on page 39.

Implementing Multiple Synchronized Schedulers

When the real-time application requires the power of multiple CPUs, the program design is only slightly different from that shown above. The program will create multiple FRSs, one master and one or more slaves. There is a restriction that each FRS must have a unique frs-master process. That is, one process can create only one FRS, master or slave.

A variety of designs is possible but the simplest is probably to replicate the model in the preceding section, as follows:

1. The parent process initializes any global resources such as memory-mapped segments, memory arenas, files and other objects. One of these global resources is the process ID of the parent process.
2. The parent process locks down the address space it will share with lightweight processes—see plock(2) and mpin(2).

3. The parent process creates the Frame Scheduler using the call **frs_create_master()**, and stores its handle in a global location.
4. The parent process creates one sub-parent process for each processor to be used.
5. Each sub-parent creates a Frame Scheduler using **frs_create_slave()**, specifying information about the master FRS.

Now the parent and each sub-parent proceed to create and enqueue the real-time processes that will be dispatched. When all processes are enqueued, the parent and sub-parents each start their respective FRS. Each real-time process begins with and **frs_join()** call to its assigned FRS.

For an example of this kind of program structure, refer to “Example of Multiple Synchronized Schedulers” on page 41.

Frame Scheduler Methods

This section describes the details of the concepts and abstractions that the Frame Scheduler uses.

Interrupt Sources

The program specifies an interrupt source to be the time base when it creates the master FRS. The Frame Scheduler takes care of initializing hardware resources and redirecting the interrupt to the appropriate processor and handler.

On-Chip Timer Interrupt

Each processor chip contains a USYNC timer which is used by IRIX for normal process scheduling. This timer is not synchronized between processors, so it cannot be used to drive multiple FRSs in a master-slave relationship. This timer can be used as an FRS time-base when the application creates only one FRS and there is a reason to not use the SYNC timer.

High-Resolution Timer

The high-resolution timer and clock is a timer that is synchronous across all processors, and is ideal to drive synchronous FRSs. On Challenge and Onyx systems this timer is based on a high resolution counter found on the CC chip.

The IRIX uses this timer for managing timeout events. At creation time, the master FRS migrates all timeout events dependent on this timer to processor 0, leaving the timer on the FRS processor free.

External Interrupts

An external interrupt is generated via a signal applied to the external interrupt sockets on a Challenge or Onyx system (refer to ei(7) for details). At FRS creation time, the system allocates an interrupt group which is used to make the interrupt simultaneously available to all processors running related FRSs.

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

Vertical Sync Interrupt

An interrupt is generated for every vertical retrace by the graphics subsystem (see gsync(3g) for details). Similar to what is done for the external interrupts, the Frame Scheduler allocates an interrupt group to distribute the VSYNC interrupt.

Device Driver Interrupt

A user-written, kernel-level device driver can supply the FRS time-base interrupt. See “The Frame Scheduler DDI” on page 33. The Frame Scheduler allocates an interrupt group to which the device driver must direct interrupts.

Software Interrupt

A programmed, software-generated interrupt can be used as the time base. Any user process can send this interrupt to the group of synchronous FRSs using the call **frs_userintr()** (see "Delivering Time-Base Interrupts to a Frame Scheduler" on page 28).

When a user creates an FRS with this interrupt source, the system allocates an interrupt group that is used to send the interrupt simultaneously to all processors involved in the execution of the real-time application.

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.

Note: User interrupts are normally triggered by a process running on cpu 0. This can lead to system deadlock if the user process generating the interrupts issues system calls other than **frs_userintr()**.

Interrupt Isolation

When an FRS is created on a processor and the real-time event interrupt is set, all other interrupts are either shut off or redirected, except for some I/O interrupts if the processor belongs to the set of processors that are sprayable (see "Preparing the System" on page 11).

Interrupt Synchronization

Frame Schedulers running on different processors can be synchronized. To make this possible, Frame Schedulers are categorized as synchronization masters or synchronization slaves. The synchronization type of an FRS is specified at creation time via the *sync_master* parameter of **frs_create()**. This takes the value FRS_SYNC_MASTER for the master FRS, and the process ID of the process that created the master FRS when creating a slave FRS.

The master FRS acts as a representative for the whole group of FRSs. It allocates the interrupt group they will use. It also synchronizes the startup procedure: it waits for all its slaves to be ready to start real-time execution, which happens only when every slave has done an **frs_start()** and all queued processes have called **frs_join()**.

After the startup procedure, the frames are kept synchronous via one of two mechanisms, depending on the interrupt source. For the SYNC Timer, all FRS's use the same CC Counter Base and clock. For the user, external, driver, and vsync interrupts, the frame schedulers use interrupt groups to broadcast one interrupt to several processors simultaneously.

Startup Synchronization

A Frame Scheduler cannot start real-time execution until all its managed processes have done their own initial setup and are ready to run their real-time loop. When **frs_start()** is invoked in order to trigger the startup procedure, the FRS waits for all processes in its queues to invoke **frs_join()**. When this state is reached the FRS waits for the next real-time interrupt, then proceeds to schedule the processes as specified in the queues.

The process that creates a frame scheduler becomes the *frs-master process*. This process may not be frame scheduled (that is, enqueued on any of the minor frames queues). The frs-master process becomes the representative for an FRS, and is sent all the error and termination signals that are sent to the processes in the FRS queues.

Scheduling Within A Minor Frame

Processes in a minor frame queue are activated in queue order. (Initially queue order is the order in which processes were named in **frs_enqueue()** calls. The queues can be reordered dynamically; see “Dynamic Management of Scheduler Queues” on page 30.)

The FRS keeps two status flags per queued process. If a process is ready to run when its turn comes, it is activated 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 already run and 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 IO), 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 to run processes are found, the FRS goes into idle mode until a process becomes available or until the end of the frame.

When a real-time interrupt is received, indicating end of minor frame, the FRS checks the flags for each processe. If the *frs_run* flag has not been set, the process never ran and therefore is a candidate for an *underrun* error. If the *frs_run* flag is set and the *frs_yield* flag is not, the process is a candidate for an *overrun* error. Whether these errors are declared depends on the scheduling discipline assigned to a process (see "Frame Scheduling Disciplines" on page 9).

At the end of a minor frame, the FRS resets all *frs_run* flags. It also resets all *frs_yield* flags except for those of processes that are Continuable in that minor frame. For those processes, the residual *frs_yield* flags will keep processes that have yielded from being dispatched in the next minor frame. All frs_yield flags are reset at the end of a major frame.

Underrun and overrun errors are communicated via IRIX signals(5). These signals are sent to *all* processes in the FRS queues (see "Frame Scheduler Signals" on page 18).

Frame Scheduler Signals

The frame scheduler uses 3 signals. By default they are set as follows:

- Overrun Signal is SIGUSR2
- Underrun Signal is SIGUSR1
- Termination Signal is SIGHUP

Signals are sent to all queued processes and to the frs-master process. The overrun and underrun signals are sent following detection of those errors. The termination signal is sent after invocation of **frs_destroy()**.

Signals are delivered to processes during kernel-to-user transitions. In timeshared mode these transitions happen at least once every scheduler clock interrupt (10 ms for Irix). However, while in frame scheduler mode, these transitions are more sporadic; they only happen at minor frame boundaries and processes context switches due to system calls or yielding.

The Frame Scheduler API

The authorized programming interface (API) to the Frame Scheduler is composed of system calls, which are available to either C or FORTRAN programs, and library calls, which provide a C program with a more convenient interface to the system calls. Calls are provided for the following operations:

- Creating a master or slave Frame Scheduler
- Enqueuing processes to a Frame Scheduler
- Joining a Frame Scheduler
- Yielding control at the end of a frame
- Delivering user-defined time-base interrupts
- Stopping and resuming Frame Schedulers
- Dynamic Management of scheduler queues
- Destroying a Frame Scheduler

Creating a Master or Slave Frame Scheduler

Data Structures

```
typedef struct frs_fsched_info {
        int     cpu;
        int     intr_source;
        int     intr_qualifier;
        int     n_minors;
        int     sync_master;
        int*    sync_slaves;
        int     num_slaves;
} frs_fsched_info_t;
typedef struct frs {
        frs_fsched_info_tfrs_info;
        int             sync_master_pid;
} frs_t;
```

System Call

```
int     schedctl ( MPTS_FRS_CREATE, frs_fsched_info_t* frs_fsched_info ) ;
```

Library Calls

```
frs_t*  frs_create ( int cpu,int intr_source, int intr_qualifier, int n_minors,
                int sync_master_pid, int* sync_slaves, int num_slaves ) ;
frs_t*  frs_create_master ( int cpu, int intr_source, int intr_qualifier, int n_minors,
                int* sync_slaves, int num_slaves ) ;

frs_t*  frs_create_slave ( int cpu, frs_t* sync_master_frs ) ;
```

A Frame Scheduler can be created using any of the calls above. Both **schedctl()** and **frs_create()** allow for the creation of either kind of FRS.

The process that creates an FRS becomes the frs-master process for that FRS. The process ID of the frs-master serves to identify the master FRS. As a result, the frs-master process is restricted in several ways:

- That process cannot create another FRS. You must create a separate process (usually with **sproc()**) for each FRS, master or slave.
- The frs-master process cannot be enqueued on any FRS.
- The frs-master receives the signals generated by its FRS.

The call **frs_create_master()** is a wrapper that uses **frs_create()** to create an FRS with *sync_master_pid* set to FRS_SYNC_MASTER. Similarly, **frs_create_slave()** is a wrapper that calls **frs_create()** using the parameters found in the specified *sync_master_frs*.

Parameters

cpu	The processor that will be managed by the created frame scheduler. Processors are numbered from 0, but processor 0 cannot be used by an FRS.
intr_source	The interrupt source, which must be one of: • FRS_INTRSOURCE_R4KTIMER for the R4K timer (not allowed for a slave FRS) • FRS_INTRSOURCE_CCTIMER for the synchronous timer • FRS_INTRSOURCE_EXTINTR for external interrupts • FRS_INTRSOURCE_VSYNC for vsync interrupts • FRS_INTRSOURCE_DRIVER for device driver interrupts • FRS_INTRSOURCE_USER for user generated interrupts
intr_qualifier	Additional information about the *Intr_source*: • The minor frame interval in microseconds when the interrupt source is either timer • The graphic pipe when the interrupt source is vsync • The device driver identifier number when the interrupt source is a device driver (see “Exporting the Initialization and Termination Functions” on page 35)
n_minors	The number of minor frames per major frame.
sync_master_pid	Specify whether the created FRS is a master or a slave: • FRS_SYNC_MASTER specifies a master • The process id of the frs-master process of the master FRS specifies that the new FRS is a slave
sync_slaves	When creating a master FRS, the address of an array of length *num_slaves*. Each array element contains the CPU number of a processor that will run a slave FRS. This information lets the master FRS allocate an interrupt group

and determine who it should wait for when starting real-time execution. When creating a slave FRS, *sync_slaves* is NULL.

num_slaves The number of slave FRSs listed in *sync_slaves*, or 0.

Return Values

frs_create(), **frs_create_master()**, and **frs_create_slave()** return a pointer to an frs_t structure when the creation was succesful, or NULL in case of error. The **schedctl()** call returns 0 to indicate success, and -1 to indicate error. In case of error, *errno* is set to

- EINVAL for invalid parameters
- EFAULT if the frs_fsched_info_t* parameter is NULL
- ENOSPC if the system could not allocate an interrupt group
- ENOMEM if the system could not allocate memory for the FRS Object
- EEXIST if an FRS Object already exists for the specified processor
- EBUSY is the specified processor is the clock processor
- EPERM for lack of permission
- ENODEV to indicate the FRS subsystem has not been installed.

Enqueuing Processes to a Frame Scheduler

Data Structure

```
typedef struct frs_queue_info {
        int     cpu;
        int     pid;
        int     minor_index;
        int     discipline;
} frs_queue_info_t;
```

System Call

```
int     schedctl ( MPTS_FRS_ENQUEUE, frs_queue_info_t* frs_queue_info ) ;
```

Library Call

```
int     frs_enqueue ( frs_t* frs, int pid, int minor_index, uint discipline ) ;
```

Processes may be created using **fork()** or **sproc()** . Using the child process ID returned by these system calls, a process may be enqueued on a frame scheduler minor frame queue using either of the calls above. **frs_enqueue()** is just a wrapper for the direct **schedctl()** call.

The frs-master process is responsible for enqueueing the processes that will be scheduled by an FRS, on the queues associated with the minor frames where they are to run.

A process may be enqueued on several queues simultaneously. Hence the same process may be activated on different minor frames within the same major frame. A process should not be enqueued to more than one FRS.

Enqueueing a process on one or more frame queues does not mean that it will immediately start being scheduled by the frame scheduler. The queued process itself has to use **frs_join()** to join the frame scheduler, and the frs-master process has to order the scheduler to start its operation. Further, the scheduler will not start scheduling processes, even if it has been signalled to start, until all processes in its queues have joined (see "Joining a Frame Scheduler" on page 24). And in the case of multiple, synchronized frame schedulers, execution will not start until all processes on all frame schedulers have joined.

Parameters

frs	The frame scheduler descriptor returned by **frs_create()**
frs_queue_info	For **schedctl()**, a structure containing the following values.
pid	The child process ID to be enqueued
minor-index	The minor frame where the process should run. Minor frame 0 is the first minor frame in the major frame.
cpu	This field of the *frs_queue_info* parameter in the **schedctl()** call identifies the processor being managed by the destination FRS.
discipline	The scheduling discipline for this child process on this queue. See "Frame Scheduling Disciplines" on page 9. The parameter must be an arithmetic combination of: • FRS_DISC_RT for the Realtime discipline, used with every process that is not FRS_DISC_BACKGROUND. • FRS_DISC_UNDERRUNNABLE, added to FRS_DISC_RT to prevent detection of underrun errors.

- FRS_DISC_OVERRUNNABLE, added to FRS_DISC_RT to prevent detection of overrun errors.
- FRS_DISC_CONT, added to FRS_DISC_RT and FRS_DISC_OVERRUNNABLE to allow the process to extend its execution over consecutive minor frames.
- FRS_DISC_BACKGROUND for a process that has no real-time constraints, and runs only in idle time.

FRS_DISC_BACKGROUND and FRS_DISC_RT are mutually exclusive.

Note: Processes using the FRS_DISC_BACKGROUND discipline must be enqueued last, after all FRS_DISC_RT processes have been enqueued.

Return Values

frs_enqueue() and the corresponding **schedctl()** call return 0 to indicate success and a negative integer to flag a failure. In case of error, *errno* will contain one of the following codes:

- EINVAL for invalid parameters
- EFAULT if the frs_queue_info_t* parameter is NULL
- EPERM for lack of permission
- ENODEV to indicate the FRS subsystem has not been installed.

Joining a Frame Scheduler

System Call

```
int     schedctl ( MPTS_FRS_JOIN, int cpu ) ;
```

Library Call

```
int     frs_join ( frs_t* frs )
```

Real-time processes need to allocate resources and set up before they start real-time execution. This includes pinning down stacks and dynamic memory (memory allocated before an **sproc()** can be pinned by the parent process, but memory allocated by the child, and the address space created by **fork()**, need to be pinned by the child), opening I/O resources, initializing data structures, locks, and semaphores, etc.

A process uses either of these calls to notify the Frame Scheduler to which it is queued that it is initialized and ready to start real-time execution.

The frs-master process cannot call **frs_enqueue()** until it has created the child process; however the child process typically calls **frs_join()** soon after it is created. Hence there's a race between the action of enqueueing a process and the action of joining the frame scheduler. If the enqueueing happens first, there is a potential problem if the scheduler starts running before the process joins. This is avoided by having the scheduler wait for all its enqueued processes to join before it starts real-time execution. This synchronization is done by the frame scheduler, so the programmer does not need to be concerned with this race. There's no problem if **frs_join()** happens first, unless, of course, the frs-master forgets to ever enqueue the process, which will have the expected effect of keeping the unenqueued process waiting forever.

Parameters

frs	An FRS descriptor as returned by **frs_create()**.
cpu	Specifies the processor being managed by the destination FRS.

Return Values

In case of success both calls return a value greater or equal to zero; and in case of error, -1 is returned and *errno* is set to one of these values:

- EINVAL for invalid parameters
- EPERM for lack of permission
- ENODEV to indicate the FRS subsystem has not been installed.

Starting the Frame Scheduler

System Call

```
int     schedctl ( MPTS_FRS_START, int cpu ) ;
```

Library Call

```
int     frs_start ( frs_t* frs ) ;
```

When all processes to be scheduled have been enqueued , the master process commands the frame scheduler to start activating processes. This is done using either of the calls above.

The scheduler waits for all enqueued processes to call **frs_join()** before it starts real-time operation. Once all processes have joined, the scheduler enables the event interrupt (if necessary), waits for the next interrupt, and starts scheduling the member processes in real-time.

When multiple synchronized FRSs are used, each FRS must be started individually. Slave FRSs do not start until the master FRS sends an enabling signal. The master FRS does not send the signal until all slave FRSs report that all processes have joined.

Parameters

cpu	Specifies the processor managed by the frame scheduler that is being started
frs	Specifies an FRS descriptor as returned by **frs_create()**

Return Values

Both calls return 0 for sucess and -1 for error. In case of error, *errno* is set to one of the following values:

- EINVAL for invalid parameters
- EPERM for lack of permission
- ENODEV to indicate the FRS subsystem has not been installed.

Yielding Control at the End of a Frame

System Call

```
int     schedctl ( MPTS_FRS_YIELD ) ;
```

Library Call

```
int     frs_yield ( void ) ;
```

A process under control of a frame scheduler is never preempted within its minor frame time period. After it finishes its work for that frame, it has to explicitly yield the processor using either of the calls above.

The concept of yielding is different than the concept of blocking. When a process blocks (by requesting some system service) it becomes unavailable to run until it is unblocked. On the other hand, when a process yields by using **frs_yield()**, it remains available for execution (ready to run), so that the frame scheduler can readily activate it when its turn to run comes again. The frame scheduler does not control framed real-time execution by blocking and unblocking processes, but rather by activating them only when their turn comes.

Processes within the same frame queue are activated in a non-preemptable round robin fashion until they yield. This means that a process may request system services, block, unblock, and continue running within a single minor frame. It is only the yield operation that causes a process to be prevented from running again within its minor frame.

At the end of a minor frame the scheduler preempts whatever is running and verifies that all processes in the frame queue were activated at least once. If a process was never activated because it was always blocked and the process is not Underrunnable, the scheduler raises an underrun exception. Similarly, the scheduler verifies that all processes in the frame queue have yielded. If it finds one that has not yielded but is not Overrunnable , it raises an overrun exception.

The scheduler resets the yield state of a process at the end of each minor frame, unless the process has been defined to be Continuable in which case the yield state carries over to the next minor frame. This is the mechanism that prevents a Continuable process from being redispatched once it has yielded. (See “Scheduling Within A Minor Frame” on page 17.)

Parameters

This call takes no parameters.

Return Values

When the operation is succesful, this call returns the number of the last minor frame where this process yielded or was preempted. (Minor frames are numbered from 0 within each major frame.) In case of error, -1 is returned with *errno* set to one of the following values:

- EINVAL for invalid parameters
- EPERM for lack of permission
- ENODEV to indicate the FRS subsystem has not been installed.

Delivering Time-Base Interrupts to a Frame Scheduler

System Call

```
int     schedctl ( MPTS_FRS_INTR, int cpu ) ;
```

Library Call

```
int     frs_userintr ( frs_t* frs ) ;
```

These calls allow the frs-master process to send real-time interrupts to one or several synchronous frame schedulers. When multiple, synchronized FRSs are in use, the specified FRS must be the master FRS.

This capability is intended to be used as a debugging aid by giving you full control of when each minor frame begins. Typically you would have a debug version of your application in which, by conditional compilation, the frs-master process generates the interrupt signal on some user input such as a mouse click.

Note: It would not be practical to drive a production real-time program from software-supplied interrupts, owing to the unpredictable and varying latency in the scheduling of the interrupt-generating process. To drive a Frame Scheduler from a custom interrupt source, use the device driver interface (see "The Frame Scheduler DDI" on page 33).

Parameters

cpu	Specifies the processor being managed by the destination frame scheduler
frs	Specifies an FRS descriptor as returned by **frs_create**.

Return Values

Both calls return 0 for sucess and -1 for error. In case of error, *errno* is set to one of the following values:

- EINVAL for invalid parameters
- EPERM for lack of permission
- ENODEV to indicate the FRS subsystem has not been installed.

Stopping and Resuming Frame Schedulers

System Calls

```
int     schedctl ( MPTS_FRS_STOP, int cpu ) ;
int     schedctl ( MPTS_FRS_RESUME, int cpu ) ;
```

Library Calls

```
int     frs_stop ( frs_t* frs ) ;
int     frs_resume ( frs_t* frs ) :
```

These calls provide a way to stop and resume a Frame Scheduler. When the stop call is issued, the scheduler state is changed such that the time-base interrupts are ignored. The scheduler continues executing all processes in the current minor frame until they have all yielded. Then the scheduler falls into an idle loop.

The resume call enables the reception of time-base interrupts. Upon the next interrupt, the scheduler starts scheduling the processes on the queue associated with the minor frame immediately after the minor frame that was active when the stop call was issued.

If the stop and resume calls are issued so close together that no event interrupt gets ignored, the resulting effect is null. The stop and resume calls always change the length of a major frame by a whole number of minor frame periods (possibly zero).

When multiple synchronized FRSs are in use, all FRSs start and stop independently. Stopping one FRS while others continue can result in their major frames being out of synchronization (presuming all major frames were the same length to begin with). You can keep all FRSs in a group in major-frame synchrony by stopping and starting all of them at the same effective time. This can be done because a scheduler does not actually stop or start until a time-base event happens. Thus if you stop all FRSs within one minor frame interval, and if you resume all FRSs in the interval after one time-base event and before the next, the group of FRSs will remain synchronized to the same minor frame.

Parameters

cpu	Specifies the processor being managed by the destination frame scheduler
frs	Specifies an FRS descriptor as returned by **frs_create()**.

Return Values

All calls return 0 for sucess and -1 for error. In case of error, *errno* is set to one of the following values:

- EINVAL for invalid parameters
- EPERM for lack of permission
- ENODEV to indicate the FRS subsystem has not been installed.

Dynamic Management of Scheduler Queues

System Calls

```
int     schedctl ( MPTS_FRS_GETQUEUELEN, frs_queue_info_t* frs_queue_info ) ;
int     schedctl ( MPTS_FRS_READQUEUE, frs_queue_info_t* frs_queue_info,
                                                    int* pidlist ) ;
int     schedctl ( MPTS_FRS_PREMOVE, frs_queue_info_t* frs_queue_info ) ;
int     schedctl ( MPTS_FRS_PINSERT, frs_queue_info_t* frs_queue_info, int base_pid ) ;
```

Library Calls

```
int     frs_getqueuelen ( frs_t* frs, int minor_index ) ;
int     frs_readqueue ( frs_t* frs, int minor_index, int* pidlist ) ;
int     frs_premove ( frs_t* frs, int minor_index, int remove_pid ) ;
int     frs_pinsert ( frs_t* frs, int minor_index, int insert_pid, int discipline, int base_pid ) ;
```

These functions are used to dynamically control the minor frame process queues while the Frame Scheduler is running.

Note: Queue manipulation entails considerable overhead, taking up valuable time from a frame and possibly leading to overrun errors.

The length of a queue is obtained using **frs_getqueuelen()**. The length of the queue specified by minor_index is returned as the function result.

The list of processes in a queue can be read using **frs_readqueue()**. All process IDs are returned in queue order to the specified array. The array must be long enough to receive them all.

The call **frs_premove()** removes a specific process from a queue. The process stops being scheduled for execution and is temporarily suspended. It can be re-inserted in the queue.

frs_pinsert() inserts a process in a queue following *base_pid* in the queue. To insert a process at the head of the queue, specify 0 for *base_pid*. The process is dispatched again in its correct turn.

Note: Only processes that have been queued with **frs_enqueue()** can be inserted. The **frs_premove()** and **frs_pinsert()** calls are designed for the purpose of re-ordering the queues of existing processes, or for changing the discipline of existing processes. They are not intended for the purpose of adding new processes to the dispatch queue.

Note: Do not rearrange the queue so as to place a Background process ahead of a real-time process. A Background process must always be the last in its queue.

Parameters

frs_queue_info	Specifies a structure in which the *cpu* field identifies the frame scheduler, and the other fields specify information about a queued process.
frs	Specifies an FRS descriptor as returned by **frs_create()**.
minor_index	As a parameter or a field in *frs_queue_info*, specifies the minor frame. Minor frames are numbered from 0.
pidlist	Address of a vector of integers long enough to receive the process IDs of all processes in one queue.
remove_pid	Process ID of a queued process to be removed from the queue.
base_pid	Process_ID of a process in the queue, or 0.
insert_pid	Process ID of a process to be inserted in the queue following *base_pid*.
discipline	The scheduling discipline for the inserted process (see "Frame Scheduling Disciplines" on page 9).

Return Values

All calls return 0 for sucess and -1 for error. In case of error, *errno* is set to one of the following values:

- EINVAL for invalid parameters
- EPERM for lack of permission

- ENODEV to indicate the FRS subsystem has not been installed
- EFAULT to indicate that pidlist cannot be accessed.

Destroying a Frame Scheduler

System Call

```
int     schedctl ( MPTS_FRS_DESTROY, int cpu ) ;
```

Library Call

```
int     frs_destroy ( frs_t* frs ) ;
```

Any user process may initiate the termination of a frame scheduler using either of the calls above.

This call causes termination signals to be sent to all member processes and to the frs-master process (see “Frame Scheduler Signals” on page 18).

The FRS dequeues all processes from the frame scheduler and unisolates the cpu it was attached to. The IRIX scheduler resumes control of the CPU. Processes continue execution in normal mode. Frame scheduler calls such as **frs_yield()** return errors. The best way to avoid problems is to have the termination signal handler wrap up all processing and then cause the process to exit.

Parameters

cpu	Specifies the processor being managed by the destination frame scheduler
frs	Specifies an FRS descriptor as returned by **frs_create()**.

Return Values

Both calls return 0 for sucess and -1 for error. In case of error, *errno* is set to one of the following values:

- EINVAL for invalid parameters
- EPERM for lack of permission
- ENODEV to indicate the FRS subsystem has not been installed.

The Frame Scheduler DDI

The Frame Scheduler provides a device driver interface to allow any device with a kernel-level device driver to generate the frame scheduler time-base interrupt. The system can contain up to 16 different drivers that support the Frame Scheduler. The Frame Scheduler distinguishes them by an *frs_driver_id* in the range 0 through 15.

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 an FRS initializes a device driver, the driver calls an FRS entry point to signal the event of an interrupt.

Note: The structure of an IRIX kernel-level device driver is discussed in the *IRIX Device Driver Programming Guide*. The latest version, for IRIX 5.3, has part number 007-0911-044.

Frame Scheduler Initialization Function

The device driver must provide a function with the following prototype (the actual function name is not significant):

```
void prefix_frs_func_set ( intrgroup_t* intrgroup ) ;
```

This function is called by a new FRS when the FRS is created with an *inter_source* of FRS_INTRSOURCE_DRIVER and *inter_qualifier* specifying this device driver's number (see "Creating a Master or Slave Frame Scheduler" on page 20).

The argument *intrgroup* identifies the interrupt group allocated for the newly created instance of the frame scheduler. The 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))
```

Note: When the device driver has directed the hardware interrupt to the interrupt group specified, the hardware interrupt will be presented at every CPU in the group. That is, the interrupt will be taken concurrently on each CPU occupied by the master or a slave FRS. The device driver interrupt handler code is entered concurrently on each CPU on each interrupt.

Example of Initialization Function

```
/*
** Frame Scheduler initialization function
** for the External Interrupts Driver
*/
void
eintr_frs_func_set(intrgroup_t* intrgroup)
{
    int s;
    ASSERT(intrgroup != 0);
    /*
    ** locking and spl level management
    ** specific to this (ei) driver
    */
    s = EILOCK();
    DISABLE_INTR();
    /*
    ** setting up the hardware to send intr to
    ** the appropriate destination.
    */
    evintr_connect(
              (evreg_t *)(epcbase + EPC_IIDINTR4),
              EVINTR_LEVEL_EPC_INTR4,
              SPLDEV,
        EVINTR_GROUPDEST(intrgroup_get_groupid(intrgroup)),
              (EvIntrFunc*)eiintr,
              (void *)0);
    /*
    ** Epilog specific to this driver
    */
    ENABLE_INTR();
    setsyscpu(-1);
    intr_enabled = 1;
    EIUNLOCK(s);
}
```

Frame Scheduler Termination Function

The device driver must provide a function with the following prototype (the actual function name is not significant):

```
void prefix_frs_func_clear ( void ) ;
```

A Frame Scheduler that initialized a device driver calls this function when the Frame Scheduler is terminating (see “Destroying a Frame Scheduler” on page 32). 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.

Example of Termination Function

```
/*
** Frame Scheduler termination function
** for the External Interrupts Driver
*/
void
eintr_frs_func_clear(void)
{
    /*
    ** Epilog specific to this
    ** particular (ei) driver
    */
    disable_intr();
    setintrcpu(config_intrcpu);
    setsyscpu(config_syscallcpu);
}
```

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 is made 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));
```

The parameter *frs_driver_id* is the driver identification number. It is specified in the *intr_qualifier* parameter to **frs_create()** in order to select this driver as the source of interrupts. The identifier is an integer between 0 and 15. Different drivers must use different identifiers.

Example of Exporting Entry Points

```
/*
** Function called by the External Interrupts
** driver to export its Frame Scheduler interface
** functions.
*/
frs_driver_export(12,
              eintr_frs_func_set,
              eintr_frs_func_clear);
```

Generating Interrupts

A driver has to call the frame scheduler interrupt handler from within its interrupt handler. This handler will be entered concurrently on each processor where the master or a slave FRS is running. It delivers the interrupt to the FRS on that processor. The function to be invoked is:

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

It is possible for a handler to be entered at a time when the FRS for its processor is not active. However, the frs_handle_driverintr() function checks for this and does nothing when nothing is required.

Example of Generating Interrupt

```
void eiintr(struct eframe_s *ep, unsigned int arg)
{
   /*
   ** Do anything required by the hardware...
   ** ...then call the Frame Scheduler:
   */
   frs_handle_driverintr();
   /*
   ** Do any additional processing needed.
   */
   return;
}
```

Using a Device Driver

This is the sequence of actions that occur when a device driver is used as a source of time-base interrupts for the Frame Scheduler.

During its initialization at system boot time, the driver exports the **prefix_frs_func_set()** and **prefix_frs_func_clear()** functions, specifying a unique driver identifier between 0 and 15. After these functions have been exported the frame scheduler is aware of the existence of this driver and will allow programs to request it as the source of interrupts.

Later, a program creates a frame scheduler specifying this driver as the source of interrupts. The scheduler verifies that this driver has exported the initialization and termination functions and invokes **prefix_frs_func_set(***intrgroup***)** for this particular driver.

As long as the FRS exists, the driver invokes **frs_handle_driverintr()** each time its interrupt handling routine is entered. This informs the FRS that an interrupt has been received.

When the FRS is finally destroyed, it invokes **prefix_frs_func_clear()** for the associated driver. The driver ceases calling **frs_handle_driverintr()** until it is again initialized by an FRS.

Examples

The following example programs are provided:

simple	A frame scheduler running several processes on a single processor
mprogs	A frame scheduler setup with independent processes
multi	Three synchronous frame schedulers running lightweight processes on three processors
driver	A pseudo-driver that demonstrates the frame scheduler device driver interface.
sixtyhz	A very simple example running one process at 60Hz
memlock	An example that takes care of locking memory down and which skips the first minor frame to account for initial cold caches.

The code for each example is in a subdirectory named for the example.

Basic Example

This example shows how to map a real-time application specification into a Frame Scheduler specification and also how to write the corresponding program using the Frame Scheduler API.

Real-time Application Specification

The application consists of 2 processes that have to periodically execute a specific sequence of code. The period for the first process, process A, is 600 milliseconds (ms). The period for the other process, process B, is 2400 ms.

Note: Such long periods are unrealistic for real-time applications. However, they allow the use of printf statements within the “real-time” loops in this sample program.

Frame Scheduler Design

The two periods and their ratio determine the selection of the minor frame period—600 ms—and the number of minor frames per major frame—4, for a total of 2400 ms.

The discipline for process A is strict real-time (FRS_DISC_RT). Underrun and overrrun errors should cause signals.

Process B should run only once in 2400 ms, so it operates as Continuable over as many as 4 minor frames. For the first 3 frames, its discipline is Overrunnable and Continuable. For the last frame it is strict real-time. The Overrunnable discipline allows process B to run without yielding past the end of each frame. The Continuable discipline ensures that once process B does yield, it is not resumed until the fourth minor frames has passed. The combination allows process B to extend its execution to the allowable period of 2400 ms, and the strict real-time discipline at the end makes certain that it yields by the end of the major frame.

There is a single FRS so a single processor is used by both processes. Process A runs within a minor frame until yielding or until the expiration of the minor frame period. In the latter case the frame scheduler generates an overrun error signaling that process A is misbehaving.

When process A yields, the frame scheduler immediately activates process B. It runs until yielding, or until the end of the minor frame where it is preempted. This is not an error since process B is Overrunable.

Starting the next minor frame, the FRS allows process A to execute again. After it yields, process B is allowed to resume running, if it has not yet yielded. Again in the third and fourth minor frame, A is started, followed by B if it has not yet yielded. At the interrupt that signals the end of the fourth frame (and the end of the major frame), process B must have yielded, or an overrun error is signalled.

Example of Scheduling Separate Programs

The code in directory *mprogs* is based on the code for the basic example (see “Basic Example” on page 38). However, the scheduled processes A and B are physically loaded as separate commands. The main program establishes the single FRS. The real-time processes are started as separate programs. They communicate with the main program using SVR4-compatible interprocess communication messages (see intro(2) and msgget(2)).

There are three separate executables in the mprogs example. The master program, in master.c, is a command that has the following syntax:

`master [-p` *cpu-number*`] [-s` *slave-count*`]`

The *cpu-number* specifies which processor to use for the one FRS this program creates. The default is processor 1. The *slave-count* tells the master how many subordinate programs will be enqueued to the FRS. The default is 2 programs.

The problems that need to be solved in this example are as follows:

- The frs-master program must enqueue the subordinate processes but, since they are started as separate programs, the master has to be informed of their process IDs after they start.
- The subordinates need to specify upon which minor frames they should be enqueued, and with what discipline.
- The master needs to enqueue the subordinates in the proper order on their minor frames, so they will be dispatched in the proper sequence. Therefore the master has to distinguish the subordinates in some way.
- The subordinate programs must join the FRS, so they need the handle of the FRS to use as an argument to **frs_join()**. However, this information is in the master's address space.
- If an error occurs when enqueueing, the master needs to tell the subordinate so it can terminate in an orderly way.

There are many ways in which these objectives could be met. In this example, the master and subordinates communicate using a simple protocol of messages exchanged using msgget(2) and msgput(2). The sequence of operations is as follows:

1. The master program creates a Frame Scheduler.
2. The master sends a message inviting the most important subordinate to reply. (All the message queue handling is in module ipc.c.)
3. The subordinate compiled from the file processA.c replies to this message, sending its process ID and requesting the FRS handle.

4. The subordinate processA sends a series of messages, one for each minor queue on which it should enqueue. The master enqueues it as requested.
5. The subordinate processA sends a “ready” message.
6. The master sends a message inviting the next most important process to reply.
7. The program compiled from processB.c will reply to this request, and the sequence 3-6 is repeated for as many slaves as the *slave-count* parameter to the master program. (Only two slaves are provided. However, you can easily create more using processB.c as a pattern.)
8. The master issues **frs_start()**, and waits for the termination signal.
9. The subordinates independently issue **frs_join()** and the real-time dispatching begins.

Example of Multiple Synchronized Schedulers

The example *multi* demonstrates the creation of three synchronized FRSs. The three FRSs use the SYNC timer to establish a minor frame interval of 50 ms. All three have 20 minor frames per major frame, for a major frame rate of 1 Hz.

The following processes are scheduled in this example:

- Processes A and D require a frequency of 20 Hz
- Process B requires a frequency of 10 Hz and can consume up to 100 ms of execution time each time
- Process C requires a frequence of 5 Hz and can consume up to 200 ms of execution time each time
- Process E requires a frequency of 4 Hz and can consume up to 250 ms of execution time each time
- Process F requires a frequency of 2 Hz and can consume up to 500 ms of execution time each time
- Processes K1, K2 and K3 are background processes that should run as often as possible, when time is available.

The processes are assigned to processors as follows:

- FRS 1 runs processes A (20 Hz) and K1 (background)
- FRS 2 runs processes B (10 Hz), C (5 Hz) and K2 (background)
- FRS 3 runs processes D (20Hz), E (4 Hz) and F (2 Hz) and K3.

In order to simplify the coding of the example, all real-time processes use the same function body, **process_skeleton()**, which is parameterized with the process name, the number of the FRS it is to join, and the "real-time" function it is to execute. In the sample code, all real-time functions are empty function bodies.

Example of Device Driver

The code in directory *driver* contains a skeletal test-bed for a kernel-level device driver that interacts with the Frame Scheduler. Most of the driver functions consist of minimal or empty stubs. However, the **ioctl()** entry point to the driver (see ioctl(2)) simulates a hardware interrupt and calls the Frame Scheduler entry point, **frs_handle_driverintr()** (see "Generating Interrupts" on page 36). This allows you to test the driver. Calling its **ioctl()** entry is equivalent to using **frs_usrintr()** (see "Delivering Time-Base Interrupts to a Frame Scheduler" on page 28).

Example of a 60 Hz Frame Rate

The example in directory *sixtyhz* demonstrates the ability to schedule a process at a frame rate of 60 Hz, a common target rate in visual simulators. A single FRS is created. It uses the high-precision timer with an interval of 16,666 microseconds (16.66 ms, approximately 60 Hz). There is one minor frame per major frame.

One real-time process is enqueued to the FRS. By changing the compiler constant LOGLOOPS you can change the amount of work it attempts to do in each frame.

Example of Memory and Cache Management

The example in directory *memlock* is almost identical to that in directory *sixtyhz*. (Use diff(1) to display the altered code.)

The functional difference between the two examples is that the real-time process in memlock.c locks its own virtual address space so as to avoid page faults. Also, it executes one major frame's worth of **frs_yield()** calls immediately after return from **frs_join()**. The purpose of this is to "warm up" the processor cache with copies of the process code and data. (An actual application process could access its major data structures prior to this yield in order to speed up the caching process.)