REACT™ In IRIX™ 5.3 Technical Report

Silicon Graphics, Inc.

A description of the real-time capabilities of IRIX Version 5.3 running on Onyx™ and CHALLENGE™ multiprocessor systems.

Table Of Contents

List Of Figures

List Of Tables

1.0 Introduction

The REACT extensions to IRIX enable a multiprocessor system to be configured to provide deterministic performance, including the response to external interrupts and signals. REACT also provides features that simplify the implementation of real-time applications.

The approach used in IRIX with REACT to achieving determinism is to provide the user with full control over the assignment of software activity to processors. One processor (or more, if desired) is designated as the system processor, and all non-deterministic system activity takes place on that processor. For example, system activity typically includes the UNIX^{TM} scheduler and general-purpose disk and network I/O. The remaining processors are designated as real-time processors, and no system activity takes place on those processors unless it is explicitly requested by a real-time process.

1.1 Scope Of This Document

This document covers the subset of IRIX operating system functionality and of Onyx and CHALLENGE multiprocessor hardware that is of particular interest to developers of real-time applications. It provides a description of system functionality, identifies the system calls and user interfaces that provide access to the functionality, and provides background information.

1.2 Configuration Assumptions

The descriptions included in this document are accurate for all Onyx systems and all multiprocessor CHALLENGE systems running IRIX 5.3. Within this document, these configurations are referred to generically as “the system”.

1.3 Related Documentation

Refer also to the following documents:

Advanced Site and Server Administration Guide (M4-ADMIN-3.0, or available on-line through IRIS Insight^{TM})

CHALLENGE and Onyx Performance Report

IRIX Device Driver Programming Guide (007-0911-030)

IRIX Device Driver Reference Pages (007-2183-001)

IRIX Man Pages (M4-IRXMP-4.0, or available on-line through IRIS Insight)

IRIX System Programming Guide (available on-line through IRIS Insight)

Symmetric Multiprocessing Systems Technical Report (EVER-IND-TR (01/93))

2.0 System Interrupt Response

This section describes the events that occur in response to an external interrupt.

2.1 Total Interrupt Response Time

The REACT extensions included in IRIX provide guaranteed deterministic interrupt response on a properly configured system. Performance is specified in terms of total interrupt response, which is defined as the interval between the occurrence of an external interrupt and the start of execution of a user process that was enabled by that interrupt. The worst-case total interrupt response time for a properly configured system running IRIX V.5 is guaranteed not to exceed 200 µs.

Total Interrupt Response time can be divided into two component intervals (refer to Figure 1):

1. **Interrupt latency.** The time between the occurrence of a hardware interrupt and the instant when the operating system begins responding to that interrupt.
2. **Software interrupt response time.** The system time spent responding to the interrupt, ending when a user process begins executing.

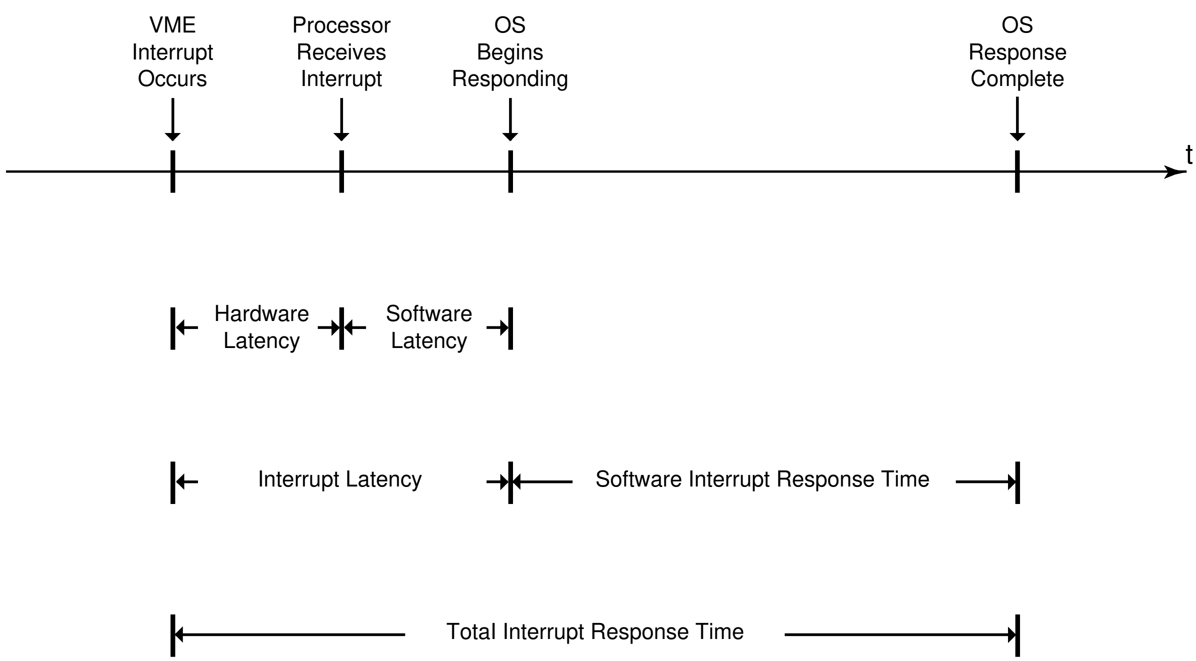

Figure 1: Components of Total Interrupt Response

Figure 1: Components of Total Interrupt Response

Interrupt latency, as defined in the previous section, can be subdivided into two component intervals:

1. **Hardware interrupt latency.** The time required for the interrupt to propagate through the hardware from its source to the R4400™ interrupt pin.
2. **Software interrupt latency.** The interval between the instant when the R4400 receives the interrupt, and the instant when the operating system begins responding to the hardware interrupt.

2.1.1 Hardware Interrupt Latency

The system implements 128 interrupt levels with priority control hardware on the CPU boards. In addition to being divided by levels, interrupts may be directed to specific processors or to groups of processors.

The path followed by VME interrupt signals is shown in Figure 2.

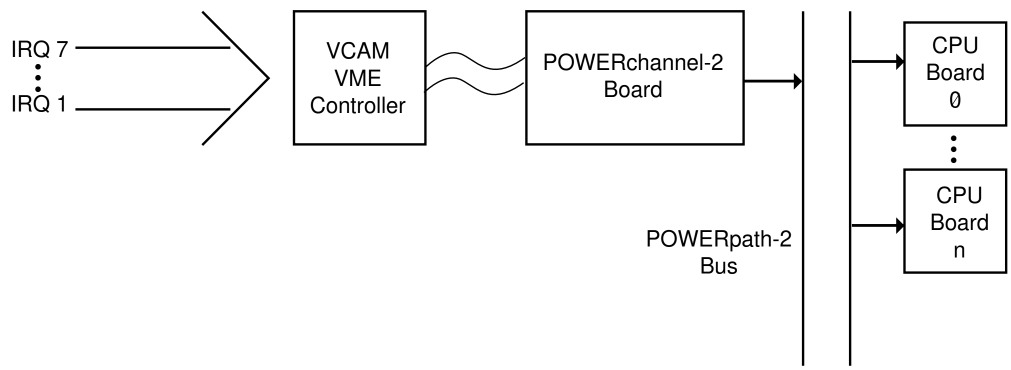

Figure 2: Hardware Interrupt Path

Figure 2: Hardware Interrupt Path

On the VCAM VME controller, there are eight Interrupt Level / Destination registers which can be programmed with arbitrary interrupt levels and destination processors. (Direct access to these registers is limited to the OS; see section 3.1.) Seven of these registers are directly associated with the seven VME interrupt lines; the eighth is associated with error detection circuitry within the VCAM. When one of the interrupt lines is active or an error is detected by the VCAM, the interrupt level and destination contained in the associated register is used to transmit an interrupt over the POWERpath-2™ bus, where it is received and acted on by the appropriate CPU.

The typical time required for the interrupt to propagate from the VME bus to the appropriate CPU chip is approximately 2µs. The theoretical worst-case propagation delay is 8 µs. Note that the theoretical worst case requires a very large configuration. For typical configurations, 4µs is a more appropriate worst-case maximum propagation delay.

Worst case interrupt propagation time can be reduced significantly by not supporting graphics or HIPPI interfaces with the same POWERchannel-2 interface used for VME.

Upon receiving a VME interrupt at a CPU, the OS's VME Kernel Driver will read status registers to identify the interrupting device(s) and clear the interrupt. This information is used to select which device-specific interrupt handler will execute. Interrupts are not cleared until all devices requesting service at that level have been read.

In order to minimize the time spent handling interrupts on the VME bus, it is preferable to connect only a single device to each VME interrupt line.

2.1.2 Software Interrupt Latency

Software interrupt latency is the interval between the instant when the hardware interrupt signal arrives at the R4400, and the instant when the OS begins responding to the interrupt.

Anytime interrupts are not masked in software, then software interrupt latency is less than one instruction time. The approach taken in IRIX with REACT is to enable real-time processors to always have interrupts enabled when a real-time interrupt can occur, so that software latency is less than one instruction time. The implications of this are covered in the following sections, which describe the situations in which interrupts are masked.

2.1.3 Potential Sources Of Software Interrupt Latency

The two situations in which interrupts are masked are:

1. When a higher priority interrupt handler is executing.
2. When critical regions of kernel code are executing.

IRIX with REACT enable users to prevent higher priority interrupts from introducing latency by directing them away from real-time processors. Critical regions of kernel code are avoided or carefully managed. The following sections examine these two situations in detail.

2.1.3.1 Interrupt Handlers

While an interrupt handler is executing, it masks interrupts at an equal or lower priority from being serviced. Furthermore, all pending interrupts that are equal to or higher than the priority of a new interrupt must complete execution before the new interrupt is serviced. Interrupt handlers run only on the processor to which the interrupt is directed. Once non-real-time interrupts are directed away from real-time processors as described in Section 3, they will never introduce latency.

2.1.3.2 Critical Kernel Regions

IRIX is a fully symmetric multiprocessor OS that allows multiple processors to execute in the kernel simultaneously. Certain critical sections of kernel code require exclusive access to shared resources. Spin locks are used to arbitrate which processor has exclusive access to the shared resources. Once a processor acquires a spin lock and enters a critical section of kernel code, it raises its interrupt level. New interrupts that are below the processor's interrupt level will not be serviced until the critical section of kernel code is complete and the processor's interrupt level is lowered.

Where possible, real-time processes should avoid making system calls during time-critical regions. Instead, forking processes, allocating memory, etc. should be done as part of an initialization routine. Typically, the only types of kernel activity that must be done during a time-critical region are process synchronization and context switches.

2.2 Software Interrupt Response Time

The software interrupt response time is the interval between the time the OS begins responding to an interrupt and the time that a user level process begins executing. Figure 3 shows the sequence of operations that are included in the software interrupt response time. These are described below:

- **Mode switch.** A mode switch occurs whenever a processor enters or leaves the kernel. It involves saving / restoring basic resources such as the integer registers and graphics registers (if any). When entering the kernel, it also includes steps to determine why the kernel was entered.
- **Interrupt handler.** This is device-specific code that always runs in kernel mode on the processor to which an interrupt is directed. Users must provide an interrupt handler for any device they add to the system that generates interrupts. Typically, an interrupt handler will unblock a process or send a signal to alert a process that the interrupt has occurred.
- **Dispatch cycle.** During the dispatch cycle, the scheduler determines which user process should run next. A dispatch cycle will be followed either by a context switch and a mode switch, or by only a mode switch, depending on the circumstances. If the next user process to be run is not the same user process that was previously running, the scheduler initiates a context switch.

  If the processor was in the kernel idle loop at the time the interrupt occurred, a context switch will not be required if the next process to be run was also running just before the processor entered the idle loop.
- **Context switch.** During a context switch, the kernel saves the context of one user process and restores the context of another user process.

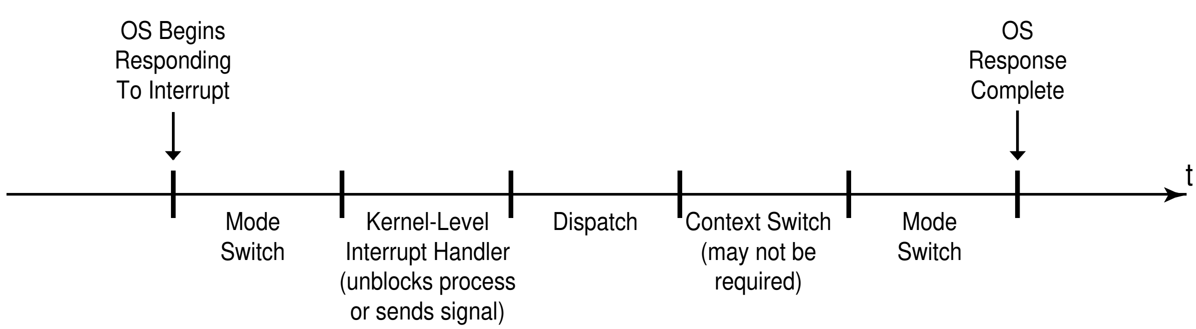

Figure 3: Components of Software Interrupt Response Time

3.0 Configuring For Real-Time Operation

The steps required to establish a set of real-time processors with deterministic response to interrupts are outlined below. A detailed description of each step is provided in the sections that follow.

1. Direct interrupts not related to real-time processes away from real-time processors. Direct real-time interrupts to the real-time processors.
2. Restrict each real-time processor so that all processes not explicitly assigned to it will be run on a non-real-time processor.
3. Lock real-time processes onto real-time processors.
4. Allocate and lock physical memory to all virtual addresses used by real-time processes.
5. Isolate the real-time processors from interprocessor interrupts used in a multiprocessor system.
6. Exempt real-time processors from system clock interrupts and UNIX timesharing scheduler activity.

Once these steps are complete, the total response time to an interrupt directed to a real-time processor will be less than 200 µs, from the time the hardware interrupt occurs until a user process begins executing.

This timing assumes that the interrupt is initiated by a VME device, and that no other VME interrupts of the same level are outstanding at the time the interrupt occurs. An interrupt handler specific to that interrupt will unblock a user process that is sleeping awaiting the interrupt. The guaranteed timing assumes that another user process is executing at the time the interrupt occurs, so a context switch is necessary.

An additional step is required to ensure that the processes assigned to real-time processors execute in the desired order. The user must assign a priority in the real-time band to each process. All real-time priorities (i.e. priorities in the real-time band) are non-degrading.

When the REACT/pro frame scheduler is used to schedule processes, step 2 - 6 listed above are invoked automatically. In this case, the only other explicit user action required to configure for real-time operation is to direct interrupts properly.

3.1 Redirecting Interrupts

Mapping of hardware interrupts to processors is controlled by system software. The user has full control over the assignment of VME interrupts to processors. The user cannot specifically control the direction of other types of interrupts to processors, but he can prevent the system from assigning any of these interrupts to real-time processors.

Controlling the mapping of interrupts to processors is done by editing the file */var/sysgen/system*, then running *autoconfig(1M)* or *lboot(1M)*. Non-real-time interrupts should be directed away from the real-time processors (using NOINTR directive), and real-time interrupts should be directed to real-time processor (using IPL directive). VME interrupts are controllable by VME interrupt level (IRQn). That is, all interrupts at a particular IRQ level on the VME bus will be directed to the same processor. In systems with multiple VME buses, the mapping of IRQ levels to processors cannot be specified by bus. This means, for example, that IRQ7 for all VME buses will be directed to the same processor.

3.2 Assigning Processes To Processors

The POWERpath-2 hardware architecture is fully symmetric, meaning that there are no software activities which can only run on a specific processor. If desired, processes can be allowed to migrate freely among processors. In its default (non-real-time) state, IRIX optimizes utilization by assigning the highest priority ready-to-run process to the first available processor (subject to some modification due to cache affinity). Other scheduling modes are available, however, enabling the user to explicitly control assignment of processes to processors.

In a real-time environment, the user should designate one or more processors for running real-time processes, lock real-time processes onto these processors, and restrict these processors from running any other processes. This can be done using shell commands, or more typically, using system calls.

The *sysmp(2)* command MP_MUSTRUN specifies that a process must run on a specific processor. Its shell command equivalent is *mpadmin(1)*. The locked process will always run on the specified processor, even when kernel code is executing on behalf of the process. An exception to this can occur with drivers which are not semaphored. If the process calls a non-semaphored driver that includes an interrupt handler for an interrupt directed to another processor, however, the interrupt handler will execute on the other processor. This temporary override of MP_MUSTRUN is transparent to the user.

Threads created using *sproc(2)* will inherit their parent's processor assignment. In the case where threads are created prior to calling *sysmp(2)*, each thread must be assigned to a processor individually.

The *sysmp(2)* command MP_RESTRICT restricts a processor to running only processes assigned to it using *sysmp(2)* MP_MUSTRUN, or its shell command equivalent, *runon(1)*. An exception to this restriction are interrupt handlers of interrupts directed to that processor. To avoid having an interrupt handler run on a processor, the interrupt should be directed to a different processor using NOINTR. MP_RESTRICT requires superuser privileges.

3.3 Locking Processes Into Memory

In a virtual memory system, any memory reference can potentially cause a page fault. The time required to bring the data being referenced from disk into physical memory will destroy real-time behavior. IRIX with REACT enables users to lock processes into memory, ensuring that real-time processes will never incur a page fault.

IRIX with REACT provides the ability to lock / unlock a specified range of addresses into physical memory using *mpin(2)* and *munpin(2)*. When called, *mpin(2)* causes the kernel to allocate physical memory to the specified address range and locks those pages down before returning to the user program. The call *munpin(2)* undoes any previous locking using *mpin(2)*. Shared memory segments (see *shmop(2)*), shared arenas (see *usinit(3P)*), and *mmap(2)* files must still be individually locked using *mpin(2)*.

UNIX SVR3 provided this capability in a rudimentary fashion, using *plock(2)*, which allows text, data, or stack segments to be locked in their entirety. IRIX also supports *plock(2)*, and has extended its functioning to automatically lock shared library sections. In addition, when *plock(2)* is used, any growth of a locked section (via *sbrk(2)*) will cause the pages to be immediately faulted in and locked down.

3.4 Processor Isolation

3.4.1 Overview

The processor isolation feature of IRIX with REACT enables the user to prevent the kernel from sending inter-processor interrupts a real-time processor.

In order to maintain the integrity of the shared memory, symmetric multiprocessing programming environment, IRIX must carry on two system activities that are not visible to user processes. These are instruction cache flushes, and Translation Look-Aside Buffer (TLB) flushes. Nominally, at irregular intervals IRIX will generate inter-processor interrupts to all processors to signal them to flush their TLB or instruction cache. For isolated processors, IRIX will instead set a status bit to indicate that a flush is pending. When an isolated processor enters kernel mode, the status bits are tested and any pending flushes are carried out. (A processor enters kernel mode whenever the running process makes a system call, or when an external interrupt occurs that is directed to that processor.) This ensures that the user's real-time process will never be preempted by an unsolicited interrupt from the kernel, and eliminates the overhead of fielding an inter-processor interrupt.

Processor isolation is established using either the *sysmp(2)* command MP_ISOLATE, or the *mpadmin(1)* shell command.

3.4.2 Activities That Override Processor Isolation

All IRIX kernel services are available to a process running on an isolated processor, but certain system calls will generate inter-processor interrupts that are not blocked by processor isolation. Fielding such an interrupt introduces latency which can cause non-deterministic behavior in processes running on the processor.

The following system calls will generate interrupts if they are executed by a process running on an isolated processor, or by an *sproc* of such a process running on any processor. These system calls can be used in a real-time application without introducing non-determinism provided they are executed in an initialization routine.

- *cachectl(2)* system call to mark pages cacheable or uncacheable
- *fork(2)* system call to create a new process
- sproc(2) system call to create a new share group process
- *sbrk(2)* system call that releases memory or grows memory past a 4 MB boundary
- *mprotect(2)* system call to set protection on a portion of memory that is shared (MAP_LOCAL is immune)
- *prctl(2)* system call to acquire information on the current process

The following system calls will generate inter-processor interrupts to an isolated processor if they are called from any processor and passed the *pid* of a process running on the isolated processor.

- *prctl(2)* system call to acquire information on a process running on an isolated processor
- a write using *proc(4)* to the address space of a process running on an isolated processor

3.4.3 Minimizing Memory Management Overhead

A processor on which no process frees any memory page will never be required to flush its TLB, even without processor isolation. Accordingly, real-time processes and device drivers should be written so that memory resources are allocated once and reused, rather than repeatedly allocated and freed.

3.5 Controlling Process Scheduling

UNIX typically implements priority aging for processes. The priority of a process that is CPU-bound is lowered gradually as the process runs. This ensures that lower priority processes can eventually run, and is desirable behavior in the environments for which UNIX originally was designed; i.e. interactive users typing on ASCII terminals. In a real-time environment, the user typically wishes to ensure that a process will run immediately when an event occurs (e.g. delivery of a timer signal). Specifying a high priority with the UNIX *nice(2)* system call is only a partial solution; even “niced” jobs age. IRIX with REACT provides fixed-priority scheduler services which meet the requirements of a real-time environment.

3.5.1 Setting Process Priority

Under IRIX with REACT, the user can specify a fixed priority for a process that does not decrease over time. Fixed priorities are available in three bands: above normal UNIX priorities (real-time band), within the same range as normal UNIX priorities, and below normal UNIX priorities. These bands are appropriate for real-time processes, general purpose processes, and large background processes, respectively. Any process in the highest priority (real-time) band will always take precedence over any process in either of the other two bands. No process in the lowest priority band will ever run if there are any processes ready to run in either of the two higher priority bands.

Within the real-time priority band, there are 10 priority levels (30 - 39). If two or more processes are ready-to-run at the time of a scheduling event, the highest priority process (lowest priority level) will always run. If a high priority process becomes ready to run while a lower priority process is running, the lower priority process will be preempted immediately.

Fixed priorities in the real-time band can be dangerous if misused. If a process with the highest fixed priority enters an infinite loop, then all other processes will be unable to run on that processor. On a uniprocessor, a system reset is the only way to regain control. On a multiprocessor system, it is possible to kill the offending process from a shell executing on another processor.

Fixed process priorities are established using the *schedctl(2)* system call. The real-time priorities range between 30 and 39, inclusive. The calling process must have superuser privileges to set a fixed priority above normal UNIX priorities.

If a real-time process is waiting for a lock that is held by a lower priority process, the priority of the process that holds the lock will be temporarily raised to that of the real-time process. This avoids a form of priority inversion that would otherwise occur if the low priority process had to wait for a resource held by a process whose priority was above its own, but below that of the real-time process.

3.5.2 Disabling The UNIX Scheduler (Shutting Off Clock Interrupts)

When configured as a timesharing system, the UNIX scheduler daemon wakes up on each processor at the beginning of each time slice and determines which process will run on that processor during that time slice. The scheduler updates the accounting statistics for the running process, decrements its time slice, and performs some other time-sharing accounting.

The UNIX scheduler daemon is awakened by a system clock interrupt sourced by a hardware timer. Each processor has its own hardware timer. The system clock timers are clocked by the processor's clock signal, and generate an interrupt every 10 ms. The default time slice is 30 ms, which means that the scheduler is run every third clock interrupt. The time slice is a tunable parameter.

In a real-time environment, this periodic system activity may be unnecessary, and represent undesirable overhead. IRIX with REACT enables the user to shut off periodic scheduler activity on an isolated processor using *sysmp(2)*. (See the previous section for a description of processor isolation.) The clock must remain enabled on at least one processor in a multiprocessor system. By default, the clock on processor 0 must remain enabled. This can be modified using the MP_FASTCLOCK command in the *sysmp(2)* system call.

Shutting off the clock on a processor suspends the action of the scheduler daemon. Under these circumstances, a running user process will continue to run without interruption until it chooses to yield. The process accounting information usually collected by the scheduler will be lost while the clock inteprocessor interrupt is shut off.

Scheduler services remain available on an as-needed basis on a processor that does not receive clock interrupts. That is, if a process blocks or sleeps waiting for completion of an I/O system call, the scheduler will be invoked to run the next-highest priority process that is assigned to that processor (if any).

If the clock interrupt is not shut off on a real-time processor, that processor will incur a clock interrupt once per time slice. The clock interrupt handler typically requires 200µs to execute, and user interrupts to the processor are disabled during this time.

Unless the REACT/pro frame scheduler is being used (see next section), processors on which the clock has been disabled will continue to receive an occasional clock interrupt. This is because it is not possible to completely disable clock interrupts in hardware. Instead, when a processor's clock interrupt is disabled, the timer is programmed to a large value. It continues to generate an interrupt every F0000000 (Hex) ticks. The duration of this period varies with the speed of the processor being used, and can be calculated by multiplying F0000000 (Hex) times the clock period of the processor. The overhead of processing these occasional interrupts is minimized by the clock interrupt handler. Upon being entered, the interrupt handler will determine that clock interrupts are disabled, and return immediately. Handling an interrupt in this way requires the processor to execute in kernel mode for approximately 20 microseconds.

When in use, the REACT/pro frame scheduler periodically resets the clock interrupt timer and thus prevents a clock interrupt from ever occuring.

3.5.3 Deadline Scheduling

IRIX with REACT includes a deadline scheduling mode which guarantees that a process will be given an opportunity to execute for a specific amount of time within a specific, recurring period. This capability is useful for applications such as data acquisition, which require a fixed amount of processing on successive frames of data. Media servers which must source frames of data at fixed intervals also require this capability.

Deadline scheduling is enabled using the system call *schedctl(2)*, which permits the user to set the length of the recurring period, as well as the run time to be allocated to the process in each period. The process is guaranteed to receive its allotment during the period, though exactly when is not specified.

3.5.3.1 Frame Scheduler

REACT/pro, available as a layered software product for IRIX 5.3 and subsequent releases, includes a frame scheduler that is useful in real-time simulation applications. The frame scheduler is a kernel module that cyclically schedules processes at intervals defined by a regularly-occurring interrupt. When enabled on a processor, the frame scheduler replaces all other IRIX scheduling policies on that processor. Frame schedulers can be enabled on all but one processor in a multiprocessor system; i.e. all but the system processor. Each frame scheduler manages execution of processes only on its own processor, but multiple frame schedulers can be synchronized to enable frames on separate processors in a system to be synchronized.

The frame scheduler uses an incoming interrupt to partition time into a sequence of minor frames. The interrupt source can be the internal timer (see Section 4.1), an external interrupt (see Section 4.3), an interrupt from one of the VME buses, the vertical retrace interrupt in an Onyx graphics system, or an interrupt sent from another user process. A user-defined number of minor frames comprise a larger recurring periodic time slice, called a major frame. This arrangement is shown in Figure 4.

Each minor frame is associated with a queue of processes to be executed within that interval of time. The list of processes in the queue is maintained in priority order. At the beginning of each minor frame, control is passed to the first process in the queue. The remaining processes in the queue are executed when the previous process yields the CPU. Processes are typically enqueued as part of the set-up of the frame scheduler, but they also can be added or deleted from a queue dynamically after the frame scheduler has been started.

The minor / major frame construct enables multiple, related frame rates to be created in the following way. Specify a timer interrupt to start minor frames at the highest desired rate, and specify the number of minor frames per major frame equal to the ratio of the lowest and highest desired rates. Enqueue processes to be run at the highest rate in every minor frame, and enqueue processes to be run at the lowest rate in only one minor frame per major frame. Enqueue processes to be run at the middle rate in every other minor frame, and so on.

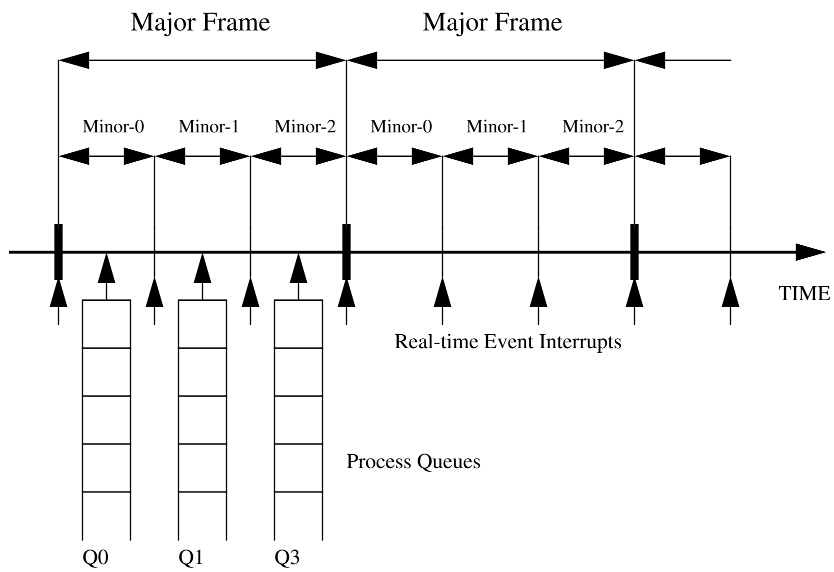

Figure 4: Asynchronous I/O operation

Each enqueued process is assigned a *discipline*, which specifies the process' preemption behavior within the queue. Nominally, a process that has not been found ready to run throughout the duration of a minor frame will generate an underrun error, and a process that has not yielded the CPU by the end of its minor frame will generate an overrun error. Errors result in a signal being sent to the group of processes participating in the frame scheduler setup. These errors can be masked by specifying the *real-time underrunnable* or *real-time overrunable* discipline, respectively, for the process when it is created. A process that is overrunable can also be assigned the *continue* discipline, which specifies that if the process that has not yielded the CPU by the end of the minor frame, it will be continued from that point in a later minor frame (on which it is queued). One or more processes per queue can be assigned the *background* discipline. These processes must be enqueued last in the queues. Background processes run only when there are spare cycles, and do not generate overrun or underrun errors.

Processes controlled by the frame scheduler can perform operations that result in their execution being suspended, such as awaiting completion of an I/O operation or availability of a system resource. When a process suspends, the frame scheduler will start execution of the next highest priority process. As soon as the event occurs on which the higher process is waiting, the frame scheduler will suspend execution of the lower priority process and resume execution of the higher priority process.

A frame scheduler is created by a user-mode program using the *frs_create(3)* call. This call specifies the CPU that will run the frame scheduler process group, isolates this CPU (see Section 3.4), specifies the number of minor frames that will form a major frame, and specifies the interrupt source that will drive the scheduler.

The process issuing the *frs_create* call becomes the *master process* for this frame scheduler. The master process creates all the other processes that are members of the frame scheduler process group, using *fork(2)* or *sproc(2).* The master process assigns each process to a minor frame, using *frs_enqueue(3)*. (The master process is not queued, since it must run asynchronously in order to respond to signals.) A single process may be enqueued on several queues simultaneously; that is, the same process may execute on multiple minor frames within the same major frame. After a process is enqueued, it connects itself to the frame scheduler using the function *frs_join(3)*.

After all processes have been enqueued, the master process signals the frame scheduler to start activating processes, using *frs_start(3*). If all enqueued processes have been connected to the frame scheduler using *frs_join(3)*, the scheduler enables the event interrupt, waits for its next occurrence, and starts scheduling the member processes in real-time.

The master process or any process in the frame scheduler process group may initiate the termination of a frame scheduler using *frs_destroy(3)*. This call disconnects all processes from the frame scheduler and un-isolates the CPU. After disconnection, processes continue execution in normal mode and may not issue any frame scheduler calls.

Each frame scheduler executes on an isolated processor. Multiple frame schedulers executing on different CPUs can be synchronized, such that one frame scheduler is defined to be a synchronization master and the other frame schedulers are synchronization slaves. In all cases, frame schedulers executing on different processors will be synchronized if they are using the same interrupt source. For the case of the internal timer interrupt, synchronization is maintained as a result of hardware synchronization of all timers in the system. For other interrupt sources, synchronization is maintained using the interrupt multi-cast feature of the POWERpath-2 system bus. When the selected interrupt occurs, it is multicast simultaneously to all processors.

4.0 Other Real-Time Programming Features

This section discusses additional features included in IRIX with REACT that are often useful to real-time users.

4.1 Timers

IRIX with REACT includes support for Berkeley interval timers (*itimers*). These timers generate an interrupt at the end of the specified interval. Hardware support for *itimers* included in the system enables short intervals to be timed with high accuracy and no increase in system overhead.

IRIX with REACT also supports event timers, which are useful for measuring the real time between two points in the user's code. Three types of event timer facilities are provided in IRIX with REACT: UNIX System V timers, BSD4.2 timers, and direct access to hardware timers from user code.

4.1.1 Interval Timers

BSD4.2 UNIX introduced the *itimer* facility. An *itimer* allows the user to specify both an offset from the current time (the delay) and the recurrence time (the interval). The timer will wait until the delay has passed, then begin timing the interval. At the end of the interval, it will fire, interrupting the processor that set up the *itimer*. The kernel's *itimer* interrupt handler delivers a signal to the process that set it up. As described below, three types of *itimers* are provided, each of which delivers a different signal to the process. Each user process can utilize up to one *itimer* of each type.

The first type is the real-time *itimer*, which delivers the signal SIGALRM. This timer measures wall clock time. It can time intervals from 400 µs to 200 hours with resolution of greater than 1 µs, subject to the restrictions on the *fasthz* parameter that are described in the following section. The accuracy of the hardware that underlies the *itimers* is one hundred parts per million. (NOTE: The timer resolution specified in this paragraph assumes IRIX Version 5.2 or later.)

The second *itimer* type is a process-virtual-time timer, which delivers the signal SIGVTALRM. It runs only when the process is running in user mode.

The third *itimer* type is the system-virtual-time timer, which delivers the signal SIGPROF. It runs both when the process is in user mode and when the kernel is operating on behalf of the user. During all system calls and I/O driver execution, the kernel is operating on behalf of a user process. Time spent executing interrupt handlers is not counted as part of system virtual time, since it is not always possible to determine to which process the time should be assigned.

The resolution of the process-virtual-time and system-virtual-time *itimers* is 10 ms. Timers other than *itimers* (such as *stimer* or *utimer*) do not measure real-time, and should not be used for real-time applications.

4.1.2 Event Timers

Event timers are typically used to measure the elapsed time between events. By getting the time before and after an operation and then subtracting, the application can calculate elapsed time. For timing events, IRIX with REACT supports direct user code access to a free-running hardware timer, and two syntaxes of UNIX system calls. The free-running hardware timer provides the highest resolution and accuracy, and is recommended for use in real-time applications.

Each processor has a free-running timer which consists of a 52-bit counter clocked continuously by the POWERpath-2 bus clock. The SGI_QUERY_CYCLECNTR command of *syssgi(2)* returns the address of the counter and the period of its clock signal. (In the current generation of multiprocessor systems, the clock period is 21 nS, though this hardware dependency should not be programmed into applications.) After querying for the counter's address, a user process can use *mmap(2)* to map the counter into its virtual address space. The process can subsequently read the hardware counter directly, without the overhead of a system call. The time required to read this counter is approximately 100 nS. All timers in a system are synchronized in hardware; i.e. all timers start at zero on the same clock tick and so always contain the same value.

In IRIX V.5, there are two options for reading the timers, depending upon the requirements of the application. The first option is to read the counter as a 64-bit integer. However, the underlying assembler code will perform two 32-bit loads. This gives rise to the possibility of a carry occuring between the two 32-bit loads. (A carry will occur every 90.2 seconds.) To fully account for this possibility, the user must perform three 32-bit reads: the high order section, followed by the low order section, followed by the high order section. If the two reads of the high order section are not equal, then a carry occurred.

In IRIX V.6, the 52-bit timer can be read as a 64-bit integer.

The *gettimeofday(3B)* call provides a system call interface to the timer described above. The system initializes a time base on power-up using the battery backed up time-of-day clock and associates a counter value with that time. Subsequent *gettimeofday(3B)* calls will return the original time base plus the difference between the current counter value and the original power-up counter value. The resolution of this counter is determined by the system parameter *fasthz* (see the following section for an explanation of *fasthz*). Note that if the *timed(1M)* daemon is enabled, it may modify the *gettimeofday(3B)* timer, resulting in erroneous event timing.

Process execution time accounting information is traditionally measured under UNIX System V using the *times(2)* system call. Berkeley added the *getrusage(3)* system call. The *times(2)* and *getrusage(3B)* return reports of accumulated real, user, and system times. Under IRIX V.5, the kernel time-stamps each process state transitions between user and kernel mode using the free-running timer and accumulates the elapsed time between state transitions.

Shutting off the system clock using the *sysmp(2)* command MP_NONPREEMPTIVE does not affect the gathering of process state transition data.

4.1.3 Details Of Timer Resolution

Each processor in a multiprocessor system includes two hardware clock interrupt signals.

The first clock signal, called the CPU scheduler clock, has a fixed period of 10 ms, and is used for scheduling and statistics gathering. The system clock is not accessible to the user.

The second clock signal, referred to as the “fast clock”, has a variable frequency set by the system parameter *fasthz*. The fast clock is distributed to all processors, and is used as the basis of the real-time timers. It is also used for event timer system calls such as *gettimeofday(3B)*.

The allowable frequency range for *fasthz* is from 500 Hz to 2500 Hz (10 ms to 400 µs period). The resolution of the real-time *itimer* is determined by the *fasthz* period. When *setitimer(2)* is called, it divides the *fasthz* period into the requested *itimer* interval to determine how many clock ticks in duration the interval should be. This results in an interval with a precision of one *fasthz* period.

By choosing an appropriate value of *fasthz*, it is possible to time any interval within the range of 400 µs to 200 hours (accurate to the level of the underlying crystal oscillator which generates the clock signal). An appropriate value of *fasthz* is one whose period is an even divisor of the desired interval. For example, if an interval of 5 ms is desired, 2000 Hz (whose period is 500 µs) is an appropriate value.

The default value of *fasthz* is 1000 Hz (1 ms). It can be modified using the *systune(1M)* utility.

In the current generation of hardware, the frequency of the hardware clock signal that is used to generate *fasthz* is 47 MHz (21 ns period). (Always use the SGI_QUERY_CYCLECNTR command of *syssgi(2)* to read the frequency in any application program that uses it.) The accuracy of the crystal oscillator that sources this signal is 100 parts per million. This translates to a maximum timing error of 100 microseconds per second. While a timer may drift this much relative to an external time source, the skew among timers for different processors in the same system will be less than one clock tick, since all timers are clocked by the same hardware clock signal.

4.2 Asynchronous Disk I/O

Typically when a user process makes a system call to perform disk I/O, the kernel initiates the I/O operation, and the user process is blocked until the I/O operation completes. To meet the needs of real-time applications, IRIX with REACT supports asynchronous I/O. When an asynchronous I/O system call is made, the kernel initiates the I/O request on behalf of the user process and returns control to the user process.The user process can either wait for the I/O operation to complete, or it can continue executing until receipt of a signal. This sequence of events is shown in Figure 5.

.

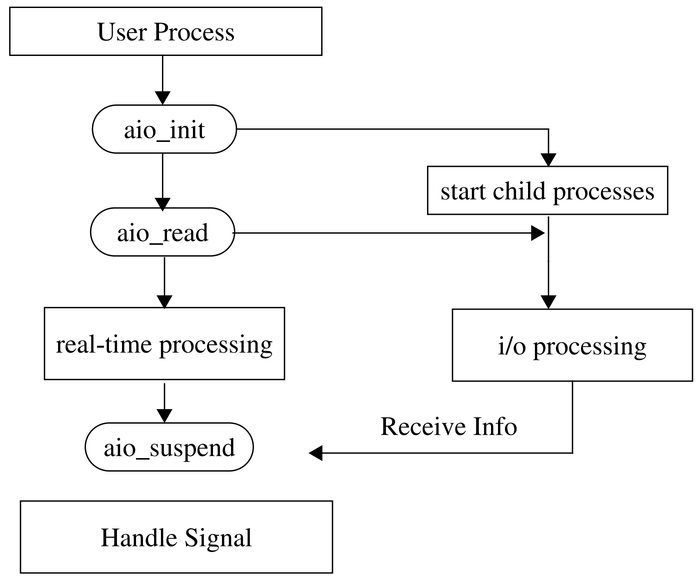

Figure 5: Asynchronous I/O operation

The asynchronous I/O interface is implemented using four child processes created by *sproc(2)* that perform the actual I/O operations, and a control block in memory (*aiocb*) containing user- and system-defined status and control information for the transaction, such as the file pointer, the number of bytes, and the priority of the request.

The child processes can be created by the *aio_init* system call, or by the call that initiates the first asynchronous I/O transaction. For real-time application, it is preferable to create the child processes using the system call *aio_init*, which allows the calling process to assign execution of the child processes to another processor. (This is because child processes created using *sproc* inherit their parent's processor assignment, so that child processes created in a real-time process' address space will run on that real-time processor.) The *aio_init(3)* call should be made prior to isolating the real-time processor. It is available in IRIX 5.2 and later versions.

The asynchronous I/O interface is initialized when a process issues its first asynchronous I/O read (*aio_read*), asynchronous I/O write (*aio_write)*, or multiple asynchronous I/O reads/writes (*lio_listio*) request. Initialization includes creating the child processes (if they have not already been created by *aio_init*) and enqueueing the *aiocb* for the subsequent I/O transactions. The child processes busy wait on a semaphore that is incremented each time an I/O request is made. When the semaphore is non-zero, one of the child processes wakes up and handles the first aio request from the linked list of aio control blocks.

By default, aio requests are queued in the order of the priority of the processes initiating the requests. The user can optionally reduce the priority of a particular request by specifying a value other than 0 in the *aio_reqprio* field of the *aiocb*. This value will be added to the process priority to determine the request's order in the queue.

The *lio_listio(3)* allows multiple I/O requests to be made in a single function call. The user process can simultaneously enqueue a number of aio requests to a device and optionally receive a queued signal when the request completes. The operation to be performed (read or write) is specified in the *aio_lio_opcode* field of each *iocb*.

The user process can choose to synchronously wait a specified amount of time for aio completion, using the *aio_suspend(3)* call. This gives a program the capability of queuing a number of aio requests, and then waiting until at least one of them has completed, or the program is interrupted by a signal, or the timeout specified in the call expires. When *aio_suspend* is used with *aio_error(3)* and *aio_return(3)*, the user process incurs the least amount of overhead using asynchronous I/O: upon return from *aio_suspend*, the functions *aio_error* and *aio_return* can be applied to the individual *aiocb* for completion status.

Alternatively, the user process can continue executing after the aio request has been queued, and be notified of I/O completion by a signal. The signal type is specified either in the *aio_sigevent.sevt_signo* field of the *aiocb*, or as the *sig.sevt_signo* argument to *lio_listio*. Because of the overhead of asynchronous signal delivery, this method is most appropriate when the number of outstanding asynchronous I/O requests is small, or when the process cannot afford to block because it has other time-sensitive tasks to complete.

The *aio_error* and *aio_return* calls can be used to determine the error status and return status, respectively, of an aio operation while it is proceeding.

Pending aio requests can be cancelled using *aio_cancel(3).* In compliance with the POSIX standard, any requested signal will be delivered to the process that initiated the request.

4.3 External Interrupts

CHALLENGE and Onyx systems provide six I/O lines (two inputs and four outputs) designed to be connected to external equipment. The interface to these lines is provided by the special device file */dev/ei* (see *ei(7)*). This interface allows separate machines to send and receive interrupts over a dedicated wire for purposes of inter-machine synchronization.

The eternal interrupt device driver maintains per-process state information, allowing any number of processes to open this device and use it without interfering with each other. In order to distribute the overhead associated with the *ioctl* calls, a user-process can specify which CPU executes the device driver system calls (EIIOCSETSYSCPU) and which CPU executes the interrupt handlers (EIIOCSETINTRCPU).

The device driver maintains queues of incoming interrupts for each process that has the device file open. There are two interrupt queues for each process: one for use by the signal handler and the other for use by the busy wait function, as explained below. The interrupt queues allow software to determine how many interrupts have arrived but have not yet been retrieved by the process. The queue can be flushed by the user program.

A user process can enable or disable interrupts using the EIIOCENABLE and EIIOCDISABLE commands. Interrupts are automatically disabled when the device is closed by the last process. A process can assert (EIIOCSETHI) or de-assert (EIIOCSETLO) an interrupt on any of the four output pins and generate an outgoing interrupt pulse (EIIOCSTROBE).

Incoming interrupts can be handled in a number of ways. Using the EIIOCSETSIG command, a process can instruct the driver to send a signal when each interrupt arrives. The interrupt queue permits the signal handler to know exactly how many interrupts have arrived, even if a signal was discarded. Or a user process may request to block in an *ioctl()* until an interrupt is received. Finally, in situations where the overhead of a system call is unacceptable (for example, when interrupts occur frequently), a process can busy wait for an interrupt to arrive, using the *eicbusywait* library function. The interrupt queue maintained for this function insures that an interrupt arriving before the library call is made will still be available to the calling process.

A process can specify the value in microseconds of the outgoing pulse width (EIIOCSETOPW), the expected incoming pulse width (EIIOCSETIPW), and the threshold beyond which an incoming pulse is considered to be "stuck" (EIIOCSETSPW).

The outgoing pulse width determines how long the output lines are asserted when the driver generates an outgoing interrupt using EIIOCSTROBE. This value should not be set too high, because the processor busy waits with all interrupts blocked during this time. On the other hand, too short a pulse may not be reliably received by the remote machine. The default is 5µs and should not normally be changed.

The expected incoming pulse width determines how long the interrupt handler will wait before returning. The interrupt handler must not return while the input line is still asserted; otherwise, multiple interrupts are received from the same input pulse, indicating to the driver that the line is "stuck". The value of the expected incoming pulse width should match the outgoing pulse width of the machine producing the pulse. The default is 5µs.

The "stuck" pulse width defines the minimum allowable time between distinct input pulses: any two pulses that arrive within this time are considered to be the same pulse. Setting this value too low will cause a single pulse to be processed as more than one interrupt; on the other hand, setting this value too high will limit the maximum rate at which interrupts can be received. The default value is 500 microseconds.

4.4 Signals

A signal is a synchronous or asynchronous notification of an event that is sent to a process when the event associated with that signal occurs. Examples of such events include hardware exceptions, timer expiration, terminal activity, as well as calls to *kill(2)*, *sigqueue(3)*, *sigsend(2)*, or *raise(3c)*. In some cases, a single event generates signals for multiple processes. A process may request a detailed notification of the source of the signal and the reason why it was generated (see *siginfo(5)).*

IRIX with REACT supports the signal functions in BSD4.3 and System V, as well as the POSIX P1003.1b-1993 real-time signals extension. The basic differences between these signal interfaces are summarized in Table 1. Because only the POSIX convention provides reliable and deterministic signal notification, the remaining discussion in this section will be confined to just those signals.

TABLE 1. Signal Functions Summary

System V	BSD4.3	Posix 1003.1	Posix 1003.1b-1993
32 signals	32 signals	32 signals	64 signals
Not reliable	Reliable	Reliable	Reliable
Error if signal occurs during system call	Restarts system call on signal	Restarts system call on signal	Restarts system call on signal
Signals not queued	Signals not queued	Signals not queued	Signals queued

IRIX with REACT supports signal numbers between 0 and 64. The signals between 0 and 32 have predefined names (see */var/include/sys/signal.h*). The POSIX standard reserves all signals between 33 (SIGRTMIN) and 64 (SIGRTMAX) for real-time applications.The signals between 1 and 32 are of equal priority, but have a higher priority than real-time signals. The real-time signals are prioritized such that the lower the signal number, the higher the signal's priority.

Each process may specify a system action to be taken in response to each signal type sent to it, called the signal's *disposition*.The set of system signal actions for a process is initialized from that of its parent. Once a disposition has been installed for a specific signal, it usually remains installed until another disposition is explicitly requested by a call to either *sigaction(2),* or until the process execs. When a process execs, all signals whose disposition have been set to catch the signal will be set to the default disposition, SIG_DFL. Alternatively, a process may request that the system automatically reset the disposition of a signal to SIG_DFL after it has been caught (see *sigaction(2)*).

A signal is said to be delivered to a process when the appropriate action for the process and signal is taken. During the time between the generation of a signal and its delivery, the signal is said to be pending (see *sigpending(2)*). A process can determine the signals that are currently pending using *sigpending*.When a pending signal is delivered, the signal will remain pending if there are additional signals queued to that signal number. Otherwise the pending indication is reset.

Because a signal‘s disposition is determined at the time it is delivered, rather than when it is caught, a signal‘s disposition can change while it is pending. When multiple unblocked signals are pending, the highest priority signal will be delivered first. On the other hand, a lower priority signal cannot preempt a higher priority signal handler.

Each process has a signal mask that defines the set of signals currently blocked from delivery to it (see *sigprocmask(2)*). The signal mask for a process is initialized from that of its parent. A signal that is blocked by a process will not be lost, but will be queued and left pending, so that if it is later unblocked (*sigsuspend(2)*) the signal will be delivered. A process can also block a signal by setting its disposition to SIG_IGN.

If the disposition of a signal is the address of a function, and the signal's SA_SIGINFO flag is set (see *sigaction(2)*), the handler will be passed a pointer to the *siginfo_t* structure, containing the signal's cause, as well as a pointer to the structure *ucontext_t,* which contains the receiving process' context when the signal was delivered. When the signal handler returns, the receiving process resumes execution at the point it was interrupted, unless the signal handler makes other arrangements.

A process can wait for the occurrence of a signal in a number of ways. It can wait for a number of signals to occur with a specified timeout (*sigtimedwait*) or without timeout (*sigwaitrt*). It can unblock a signal and then wait for that signal in a single atomic operation (*sigsuspend(3)*). Or it can simply return the value for a queued signal (*sigwaitinfo(3)*).

5.0 VME Bus Capabilities and Use

The system VME interface is a high performance implementation which provides full support for all features of Revision C.2 of the VME Specification plus the A64 and D64 modes as defined in Revision D. The VME interface is designed to allow both direct access of addresses on the POWERpath-2 bus by devices on the VME bus and direct access of addresses on the VME bus by devices on the POWERpath-2. Address mapping is provided which allows VME devices to perform DMA access to user-process virtual addresses.

5.1 Configurations

All CHALLENGE and Onyx systems contain a 9U VME bus in their main card cage as part of the standard I/O complement. Rack configurations may optionally include an auxiliary 9U VME card cage. This cage may be configured as one, two, or four VME busses. Table 2 illustrates the number of VME slots available in various CHALLENGE and Onyx configurations:

TABLE 2. CHALLENGE / Onyx VME Slots

	Main Cage VME Slots	Aux Cage VME Slots - 1 Bus	Aux Cage VME slots - 2 Busses	Aux Cage VME Slots - 4 Busses
CHALLENGE L	5	none	none	none
Onyx Deskside	3	none	none	none
Challenge XL	5	20	10 / 9	5 / 4 / 4 / 4
Onyx Rack	4	20	10 / 9	5 / 4 / 4 / 4

Determining whether to split the auxiliary VME cage into multiple busses should be done by examining the bandwidth required. Each additional VME bus which is configured requires an F cable output from an F-HIO card installed in a POWERchannel-2 board and a Remote VCAM board installed in the auxiliary VME cage.

Up to a total of three VME busses (two in auxiliary cage) may be supported using the first POWERchannel-2 board in a system; four or more busses requires the addition of a second POWERchannel-2. Table 3 illustrates the configuration requirements for various numbers of VME busses.

TABLE 3. POWERchannel-2 (Pc-2) Configurations vs. Number of VME Busses

# VME Busses	Pc-2 #1 Slot 1	Pc-2 #1 Slot 2	Pc-2 #2 Slot 1	Pc-2 #2 Slot 2
1	open	open	not req.	not req.
2	F-HIO Short	open	not req.	not req.
3 (1 Pc-2)	F-HIO Short	F-HIO Short	not req.	not req.
3 (2 Pc-2)	open	open	F-HIO	open
4	open	open	F-HIO	F-HIO
5	open	open	F-HIO	F-HIO

Note that F-HIO Short modules, which are intended for use only on POWERchannel-2 #1, have only a single F Cable output, while regular F-HIO modules provide two F Cable outputs. This explains why a second POWERchannel-2 board is required for 4 or more VME busses, but the HIO slots on POWERchannel-2 #1 is not used in this configuration.

5.2 Memory Mapping

The system supports mapping of VME addresses into the POWERpath-2 address space for programmed I/O (PIO) control of VME devices and supports mapping of POWERpath-2 addresses into the address space of the VME bus for direct memory accesses (DMA) by devices on the VME bus. The implementation and operation of these two mappings differ, as described below.

5.2.1 PIO Mapping

PIO mapping is based on opening a 128 megabyte *window* from the POWERpath-2 bus address map into the VME address spaces. Each VME bus in the system has its own window. This window is divided into sixteen 8 Mb segments. The first of these segments contains VCAM control registers and the A16 VME address space.

The other 15 segments are used to access the VME bus through a 15 entry Map Ram in the VMECC. Each Map Ram entry contains a six bit Address Modifier and nine high-order address bits. When a reference is made to one of the segments, these bits are read from the Map Ram and combined with the offset into the 8 Mb segment to form an A64, A32, or A24 VME address. One of the VCAM control registers stores the additional 32 high order address bits which are used to form A64 addresses.

PIO Map Ram entries are set up by making a system call. Parameters of the call are the VME address range to be mapped and the Address Modifier code to use in accessing the VME bus. The service returns a pointer to the virtual address at which the VME bus will be accessed. The mapping remains established for the life of the calling process unless another call is made to close the mapping. See *usrvme(7).*

5.2.2 DMA Mapping

DMA mapping is based on the use of page tables which are stored in the system main memory. This scheme allows VME DMA devices to reference a stream of contiguous virtual addresses which correspond to virtual addresses in the user process' space.

These virtual addresses may refer to scattered physical pages in main memory. Thus, the VME device is able to view the addresses it references in the same way as the user process, and DMA transfers which span multiple pages can be performed as a single operation.

Each DMA stream from the VME bus is assigned a VME Virtual Base address which corresponds to a pointer into a translation ram on the POWERchannel-2. The contents of this ram point to the beginning of a page table in main memory. Bits 20:12 of the VME Virtual Address are used as the offset into the page table. The page table entry is used to translate the VME Virtual Address into a physical address on the POWERchannel-2.

Each VME I/O adapter on the POWERchannel-2 caches sixteen translations internally. Whenever a VME device performs a memory reference, the translation cache is checked. If a hit occurs, the cache entry is used to translate the VME Virtual Address to a POWERpath-2 bus physical address. If a miss occurs, the POWERchannel-2 hardware automatically fetches a new translation from the page tables in main memory and loads the new translation into the cache, replacing the least recently used entry. Because each VME bus has its own translation cache with eight pairs of entries, up to eight DMA streams can be simultaneously active on a single VME bus without incurring a loss of performance due to thrashing.

A kernel-resident device driver is required to set up DMA mapping. See *IRIX Device Driver Programming Guide* for more information.

5.3 Optimizing Bandwidth and Latency

The system architecture is designed to maximize the total available I/O connectivity and bandwidth. Most high performance I/O devices operate by performing block-mode DMA, and this is the mode of operation which yields the best results. For devices which are not capable of DMA operation, the system provides the option of accessing these boards via programmed I/O. In addition, the system provides a DMA engine in the VME interface which can perform DMA with non-DMA VME devices to optimize their performance.

Table 4 summarizes the performance of the system VME bus using a VME DMA device to access system memory in various transfer modes. Actual performance may vary somewhat depending on the timing of the DMA controller used. The system has enough bandwidth that performance is nearly independent of whether other I/O streams (from other VME busses or other controllers on the POWERchannel-2) are active at the same time.

5.3.1 PIO Operations

Some VME devices are only capable of operating in a slave mode. For these boards, there are two alternative ways to access them: using PIO read operations from a CPU or using the DMA engine included in the POWERpath-2 architecture. PIO is the most straightforward method and will generally require little or no rewriting to port application software from other systems. However, the bandwidth available through this method is limited, especially for reads (see Table 5.). This is because PIO operations cannot be pipelined - the entire path from the POWERchannel-2 to the VME bus and

TABLE 4. VME Bus DMA Performance (using DMA board on VME)

	READ	WRITE	Block Size
D8	0.4 Mb / S	0.6 Mb / S	n/a
D16	0.8 Mb / S	1.3 Mb / S	n/a
D32	1.6 Mb / S	2.6 Mb / S	n/a
D32 BLOCK	22 Mb / S	24 Mb / S	256 Byte
D64 BLOCK	55 Mb / S	58 Mb / S	2048 Byte

the CPU itself remains occupied from the time a PIO read starts to its completion. Each PIO read operation requires two transfers over the POWERpath-2 bus; one to send the VME address to be read, and another to retrieve the data. Write operations to the VME bus are somewhat faster, requiring data to be passed in only one direction. (In contrast, DMA transfers can be extensively pipelined through the use of data prefetching, which is performed by the hardware.) Latency for single PIO reads from the VME bus is approximately 4 microseconds.

If multiple VME busses are in use, PIO bandwidth may be increased by using a different CPU to perform PIO accesses to each VME bus. Using more than one CPU to do PIO accesses to devices on the same VME bus will result in little or no increase in overall bandwidth, due to the fact that the POWERpath-2 bus uses split transactions.

TABLE 5. VME PIO Bandwidth (CPU accessing Slave on VME Bus)

	READ	WRITE
D8	0.2 Mb / S	0.75 Mb / S
D16	0.5 Mb / S	1.5 Mb / S
D32	1 Mb / S	3 Mb / S

5.3.2 DMA Engine

Because of the modest PIO transfer rates achievable, a DMA engine is included as part of each VME bus in a POWERpath-2 system. The DMA engine enables efficient block-mode DMA transfers of data between system memory and VME boards which support only slave mode (PIO) operations.

Provided that blocks of data of at least 32 contiguous bytes are used, this DMA engine can transfer data at a higher rate than that achieved using PIO. The DMA engine is capable of performing D8, D16, D32, D32 Block, and D64 Block data transfers in A16, A24, and A32 address spaces.

Table 6 illustrates the performance of the DMA engine versus the size of the transfer for a typical VME slave device using D32 accesses. The performance increases with increasing block size because the start-up cost of using the DMA engine is amortized over a larger number of bytes. In systems with multiple VME busses, each bus includes an independent DMA engine, with the result that simultaneous performance on each bus will be as shown in Table 6. Performance using D64 transfers will be approximately (though somewhat less than) twice the performance shown in Table 6.

A set of library functions for controlling the DMA engine from a user process is provided in IRIX with REACT. These allow the user to set up buffers, map the DMA engine into a process' virtual address space, and initiate DMA operations. (See *usrdma(7)* and *udmalib(3)*). The initialization routines use system calls, but the actual

TABLE 6. VME DMA Engine Performance vs. Block Size (MB / sec, D32 transfers)

Size (bytes)	READ	WRITE	BLOCK READ	BLOCK WRITE
32	2.8	2.6	2.7	2.7
64	3.8	3.8	4.0	3.9
128	5.0	5.3	5.6	5.8
256	6.0	6.7	6.4	7.3
512	6.4	7.7	7.0	8.0
1024	6.8	8.0	7.5	8.8
2048	7.0	8.4	7.8	9.2
4096	7.1	8.7	7.9	9.4

transfers are completed in user mode, eliminating the overhead of making a system call. This keeps very low the overhead involved in using the DMA engine, resulting in a net performance increase compared with PIOs for transfers of 32 bytes and larger.

5.3.3 Intelligent I/O Controllers

In addition to the techniques described above, it is possible to use a VME CPU board (non-SGI) to act as an intelligent DMA engine to copy data between a slave VME board and system memory. This may offer advantages if non-contiguous blocks of data must be transferred, or if it is desirable to do some preprocessing of the data before writing it to system memory.

6.0 SCSI Capabilities and Use

The system includes two SCSI-2 controllers on each POWERchannel-2 board, and has the capability to add additional controllers on HIO modules in groups of three. Since these controllers are capable of differential output, SCSI devices can be cabled long distances from the system cabinet. The SCSI controllers are inexpensive and efficient DMA devices which make near-optimum use of the POWERpath-2 bus bandwidth.

Using 16-bit SCSI, useful bandwidths of up to 14 megabytes per second may be achieved on each SCSI channel. Using 8-bit SCSI, approximately 7 megabytes per second is available. These bandwidths may be achieved by configuring the system to perform DMA directly into the user process' address space without buffering.

Although SCSI represents a non-traditional approach to controlling real-time external hardware, some of our real-time customers are finding that it can be used to provide a more cost-effective interface than VME for controlling remote devices. The ability to cable it long distances and the high density (up to eight SCSI channels per POWERchannel-2 Board) of connections make SCSI a particularly attractive option in the system.

7.0 Summary of REACT System Functions

TABLE 7. REACT System Functions

System Call Name	Description
realtime(5)	Introduction to IRIX real-time facilities
sproc(2)	Creates share group process
ei(7)	External interrupt interface specification
ftimer(1)	Reports status of high-resolution interval timer
timers(5)	Description of BSD4.3 interval timers
systune(1M)	Displays or sets system tuning parameters
sleep(2)	Suspends process execution for specified time
time(2)	Returns time in seconds since 1/1/70
times(2)	Returns time since start of process
gettimeofday(2)	Returns current time in seconds and microseconds
syssgi(2)	Returns system-dependent information
mmap(2)	Maps range of I/O addresses to user's address space
frs_create(3)	Creates frame scheduler process group
frs_enqueue(3)	Enqueues a process on a minor frame
frs_join(3)	Connects an enqueued process to the frame scheduler
frs_start(3)	Starts frame scheduler
frs_yield(3)	Causes an enqueued process to yield the processor to another process
frs_destroy(3)	Terminates a frame scheduler
sigaction(2)	Specifies and reports on handling of individual POSIX signals
sigaltstack(2)	Sets or gets signal on alternate stack
sigblock(3B)	Blocks signals from delivery to process (BSD4.3)
siginfo(5)	Returns information about signal generation
signal(2)	Interface to (unreliable) System V UNIX signals
signal(3B)	Interface to (reliable) BSD4.3 signals
signal(5)	Description of POSIX signal mechanism
sigpending(2)	Returns set of pending signals (POSIX)
sigprocmask(2)	Manipulates signals blocked from delivery to process (POSIX)
sigqueue(3)	Queues a signal to process or group of processes
sigsend(2)	Sends signal to process or group of processes
sigsuspend(2)	Releases blocked signals and waits for interrupt (POSIX)
sigsetops(3)	Manipulates and examines POSIX signal dispositions
sigset(2)	Manages signal disposition (System V)
sigstack(2)	Sets or gets signal stack context

TABLE 7. REACT System Functions

System Call Name	Description
sigtimedwait(3)	Waits on set of signals with timeout
sigvec(3B)	Specifies and reports on disposition of individual BSD4.3 signals
sigwait(3)	Blocks process and waits for signal
sigwaitinfo(3)	Returns value of queued signal
sigwaitrt(3)	Waits for queued signals (IRIX 5.2 only)
mpin(2)	Locks specified range of addresses in memory
munpin(2)	Unlocks specified range of addresses in memory
plock(2)	Locks entire virtual space in memory
punlock(2)	Unlocks entire virtual space
aio_read(3)	Issues aio read request
aio_write(3)	Issues aio write request
aio_cancel(3)	Cancels one or more aio requests
aio_error(3)	Returns error status of an aio request
aio_init(3)	Initializes POSIX asynchronous I/O interface
aio_return(3)	Returns error status of an aio request
aio_suspend(3)	Suspends calling process until aio request completes
lio_listio(3)	Issues multiple aio requests
schedctl(2)	Sets non-degrading priorities
pset(1M)	Displays and manages processor set information
usinit(3P)	Initializes shared arena
usmalloc(3P)	Allocates shared memory from shared arena
usnewlock(3P)	Allocates and initializes lock from shared arena
usnewsema(3P)	Allocates and initializes semaphore from shared arena
poll(2)	Waits for completion of multiple file operations
usnewpollsema(3P)	Allocates and initializes pollable semaphore
blockproc(2)	Blocks a process
runon(1)	Locks process to a specified processor
mpa*sysmp(2)*dmin(1)	Controls and reports processor status
sysmp(2)	Sets multiprocessing system parameters
1boot(1M)	Implements updates of */var/sysgen/system*
udmalib(3X)	Library of routines for using the DMA engine