Chapter 6

Control of External Interrupts

Some Silicon Graphics computer systems can generate and receive external interrupt signals. These are simple, two-state signal lines that cause an interrupt in the receiving system.

The external interrupt hardware is managed by a kernel-level device driver that is distributed with IRIX and automatically configured when the system supports external interrupts. The driver provides two abilities to user-level processes:

- the ability to change the state of an outgoing interrupt line, so as to interrupt the system to which the line is connected)
- the ability to capture an incoming interrupt signal with low latency

Note: Some software for external interrupt support is closely tied to the hardware of the system. The features described in this chapter are hardware-dependent and in many cases cannot be ported from one system type to another without making software changes.

Beginning in IRIX 6.2, there is a hardware-independent way to capture incoming external interrupts: the user-level interrupt facility (ULI). To learn about ULI, see Chapter 7, “User-Level Interrupts,” especially “Registering an External Interrupt Handler” on page 127.

External Interrupts in Challenge and Onyx Systems

The hardware architecture of the Challenge/Onyx series supports external interrupt signals as follows:

- Four jacks for outgoing signals are available on the master IO4 board. A user-level program can change the level of these lines individually.
- Two jacks for incoming interrupt signals are also provided. The input lines are combined with logical OR and presented as a single interrupt; a program cannot distinguish one input line from another.

The electrical interface to the external interrupt lines is documented at the end of the ei(7) reference page.

A program controls the outgoing signals by interacting with the external interrupt device driver. This driver is associated with the device special file */dev/ei*, and is documented in the ei(7) reference page.

Generating Outgoing Signals

A program can generate an outgoing signal on any one of the four external interrupt lines. To do so, it first opens */dev/ei*. Then it can apply **ioctl()** on the file descriptor to switch the outgoing lines. The principal functions are summarized in Table 6-1.

Table 6-1 Functions for Outgoing External Signals

Operation	Typical ioctl() Call
Set pulse width to *N* microseconds.	ioctl(*eifd*, EIIOCSETOPW, *N*)
Return current output pulse width.	ioctl(*eifd*,EIIOCGETOPW,&*var*)
Send a pulse on some lines *M*.[a]	ioctl(*eifd*, EIIOCSTROBE, *M*)
Set a high (active, asserted) level on lines *M*.	ioctl(*eifd*, EIIOCSETHI, *M*)
Set a low (inactive, deasserted) level on lines *M*.	ioctl(*eifd*, EIIOCSETLO, *M*)

a. *M* is an unsigned integer whose bits 0, 1, 2, and 3 correspond to the external interrupt lines 0, 1, 2, and 3. At least one bit must be set.

In the Challenge and Onyx series, the level on an outgoing external interrupt line is set directly from a CPU. The device driver generates a pulse (function EIIOCSTROBE) by asserting the line, then spinning in a disabled loop until the specified pulse time has elapsed, and finally deasserting the line. Clearly, if the pulse width is set to much more than the default of 5 microseconds, pulse generation could interfere with the handling of other interrupts.

The calls to assert and deassert the outgoing lines (functions EIIOCSETHI and EIIOCSETLO) do not tie up the kernel. However, direct assertion of the outgoing signal should be used only when the desired signal frequency and pulse duration are measured in milliseconds or seconds. No user-level program can hope to generate pulse durations measured in microseconds by calling these functions. For one thing, the minimum guaranteed interrupt service time is 200 microseconds. An interrupt occurring between the call to assert the signal and the call to deassert it will stretch the intended pulse width by at least 200 microseconds.

Receiving Incoming External Interrupts

An important feature of the Challenge and Onyx external input line is that interrupts are triggered by the level of the signal, not by the transition from deasserted to asserted. This means that, whenever external interrupts are enabled and any of the input lines are in the asserted state, an external interrupt occurs. The interface between your program and the external interrupt device driver is affected by this hardware design. The functions for incoming signals are summarized in Table 6-2.

Table 6-2 Functions for Incoming External Interrupts

Operation	Typical ioctl() Call
Enable receipt of external interrupts.	ioctl(*eifd*, EIIOCENABLE) eicinit();
Disable receipt of external interrupts.	ioctl(*eifd*, EIIOCDISABLE)
Block in the driver until an interrupt occurs.	ioctl(*eifd*,EIIOCRECV,&*eiargs*)
Request a signal when an interrupt occurs.	ioctl(*eifd*, EIIOCSTSIG, *signumber*)
Wait in an enabled loop for an interrupt.	eicbusywait(1)
Set expected pulse width of incoming signal.	ioctl(*eifd*, EIIOCSETIPW,*microsec*)

Table 6-2 (continued) Functions for Incoming External Interrupts

Operation	Typical ioctl() Call
Set expected time between incoming signals.	ioctl(*eifd*, EIIOCSETSPW,*microsec*)
Return current expected time values.	ioctl(*eifd*, EIIOCGETIPW,&*var*) ioctl(*eifd*, EIIOCGETSPW,&*var*)

Detecting Invalid External Interrupts

The external interrupt handler maintains two important numbers:

- the expected input pulse duration in microseconds
- the minimum pulse-to-pulse interval, called the "stuck" pulse width because it is used to detect when an input line is "stuck" in the asserted state

When the external interrupt device driver is entered to handle an interrupt, it waits with interrupts disabled until time equal to the expected input pulse duration has passed since the interrupt occurred. The default pulse duration is 5 microseconds, and it typically takes longer than this to recognize and process the interrupt, so no time is wasted in the usual case. However, if a long expected pulse duration is set, the interrupt handler might have to waste some cycles waiting for the end of the pulse.

At the end of the expected pulse duration, the interrupt handler counts one external interrupt and returns to the kernel, which enables interrupts and returns to the interrupted process.

Normally the input line is deasserted within the expected duration. However, if the input line is still asserted when the time expires, another external interrupt occurs immediately. The external interrupt handler notes that it has been reentered within the "stuck" pulse time since the last interrupt. It assumes that this is still the same input pulse as before. In order to prevent the stuck pulse from saturating the CPU with interrupts, the interrupt handler disables the external interrupt signal.

External interrupts remain disabled for one timer tick (10 milliseconds). Then the device driver reenables external interrupts. If an interrupt occurs immediately, the input line is still asserted. The handler disables external interrupts for another, longer delay. It continues to delay and to test the input signal in this manner until it finds the signal deasserted.

Setting the Expected Pulse Width

You can set the expected input pulse width and the minimum pulse-to-pulse time using **ioctl()**. For example, you could set the expected pulse width using a function like the one shown in Example 6-1.

Example 6-1 Function to Test and Set External Interrupt Pulse Width

```
int setEIPulseWidth(int eifd, int newWidth)
{
   int oldWidth;
   if ( (0==ioctl(eifd, EIIOCGETIPW, &oldWidth))
   &&   (0==ioctl(eifd, EIIOCSETIPW, newWidth)) )
      return oldWidth;
   perror("setEIPulseWidth");
   return 0;
}
```

The function retrieves the original pulse width and returns it. If either **ioctl()** call fails, it returns 0.

The default pulse width is 5 microseconds. Pulse widths shorter than 4 microseconds are not recommended.

Since the interrupt handler keeps interrupts disabled for the duration of the expected width, you want to specify as short an expected width as possible. However, it is also important that all legitimate input pulses terminate within the expected time. When a pulse persists past the expected time, the interrupt handler is likely to detect a “stuck” pulse, and disable external interrupts for several milliseconds.

Set the expected pulse width to the duration of the longest valid pulse. It is not necessary to set the expected width longer than the longest valid pulse. A few microseconds are spent just reaching the external interrupt handler, which provides a small margin for error.

Setting the Stuck Pulse Width

You can set the minimum pulse-to-pulse width using code like that in Example 6-1, using constants EIIOCGETSPW and EIIOCSETSPW.

The default stuck-pulse time is 500 microseconds. Set this time to the nominal pulse-to-pulse interval, minus the largest amount of “jitter” that you anticipate in the signal. In the event that external signals are not produced by a regular oscillator, set this value to the expected pulse width plus the duration of the shortest expected “off” time, with a minimum of twice the expected pulse width.

For example, suppose you expect the input signal to be a 10 microsecond pulse at 1000 Hz, both numbers plus or minus 10%. Set the expected pulse width to 10 microseconds to ensure that all pulses are seen to complete. Set the stuck pulse width to 900 microseconds, so as to permit a legitimate pulse to arrive 10% early.

Receiving Interrupts

The external interrupt device driver offers you four different methods of receiving notification of an interrupt. You can

- have a signal of your choice delivered to your process
- test for interrupt-received using either an **ioctl()** call or a library function
- sleep until an interrupt arrives or a specified time expires
- spin-loop until an interrupt arrives

You would use a signal when interrupts are infrequent and irregular, and when it is not important to know the precise arrival time. Signal latency can be milliseconds long in some extreme cases. For this reason, it would not be wise to use signals to handle a high rate of interrupts, nor to expect to time interrupts closely. Use a signal when, for example, the external interrupt represents a human-operated switch or some kind of out-of-range alarm condition.

The **ioctl(**EIIOCRECV**)** call tests for an interrupt, or suspends the caller until an interrupt arrives or a timeout expires (see the ei(7) reference page for details). Use this method when interrupts arrive frequently enough that it is worthwhile devoting a process to handling them.

The **ioctl()** call is a fairly costly method of polling, since it entails entry to and exit from the kernel. This is not significant if the polling is infrequent—for example, if one poll call is made every 60th of a second.

When the **ioctl()** call is used to wait for an interrupt, an unknown amount of time can pass between the moment when the interrupt handler unblocks the process and the moment when the kernel dispatches the process. This makes it impossible to timestamp the interrupt at the microsecond level.

In order to detect an incoming interrupt with minimum latency, use the library function **eicbusywait()** (see the ei(7) reference page). This function does not switch into kernel mode, so it is a very fast method of polling for an interrupt. However, if you ask it to wait until an interrupt occurs, it waits by spinning on a repeated test for an interrupt. This monopolizes the CPU, so this form of waiting can be used reliably only by a process running in an isolated CPU. (If there are other processes to run, or interrupts to handle, the polling loop in **eicbusywait()** shares the CPU and can be preempted for long periods.)

The benefit of **eicbusywait()** is that, in an isolated, nonpreemptive CPU, control returns to the calling process in negligible time after the interrupt handler detects the interrupt, so the interrupt can be handled quickly and timed precisely.