User-Level Interrupts

The user-level interrupt (ULI) facility complements the ability to perform programmed I/O (PIO) from user space. You can use PIO to initiate a device action that leads to a device interrupt, and you can intercept and handle the interrupt in your program. For function prototypes and other details, see the uli(3) reference page.

Overview of ULI

In the past, PIO could only be synchronous: the program wrote to a device register, then polled the device until the operation was complete. With ULI, the program can manage a device that causes interrupts on the VME bus. You set up a handler function within your program. The handler is called whenever the device causes an interrupt.

In IRIX 6.2, user-level interrupts are supported for VME-bus devices, and for external interrupts on the Challenge and Onyx systems.

When using ULI with a VME device, you use VME PIO to initiate device actions and to transfer data to and from device registers (see "VME Programmed I/O" on page 62). When using ULI to trap external interrupts, you enable the interrupts with **ioctl()** calls to the external interrupt handler (see Chapter 6, "Control of External Interrupts").

The User Level Interrupt Handler

The ULI handler is a function within your program. It is entered asynchronously from the IRIX kernel's interrupt-handling code. The kernel transfers from the kernel address space into the user process address space, and makes the call in user (not privileged kernel) execution mode. Despite this more complicated linkage, you can think of the ULI handler as a subroutine of the kernel's interrupt handler. As such, the performance of the ULI handler has a direct bearing on the system's interrupt response time.

Like the kernel's interrupt handler, the ULI handler can be entered at almost any time, regardless of what code is being executed by the CPU—a process of your program or a process of another program, executing in user space or in a system function. In fact, the ULI handler can be entered from one CPU while the your program executes concurrently in another CPU. Your normal code and your ULI function can execute in true concurrency, accessing the same global variables.

Restrictions on the ULI Handler

Because the ULI handler is called in a special context of the kernel's interrupt handler, it is severely restricted in the system facilities it can use. The list of features the ULI handler may not use includes the following:

- Any use of floating-point calculations. The kernel does not take time to save floating-point registers during an interrupt trap. The floating-point coprocessor is turned off and an attempt to use it in the ULI handler causes a SIGILL (illegal instruction) exception.
- Any use of IRIX system functions. Because most of the IRIX kernel runs with interrupts enabled, the ULI handler could be entered while a system function was already in progress. System functions do not support reentrant calls. In addition, many system functions can sleep, which an interrupt handler may not do.
- Any storage reference that causes a page fault. The kernel cannot suspend the ULI handler for page I/O. Reference to an unmapped page causes a SIGSEGV (memory fault) exception.
- Any calls to C library functions that might violate the preceding restrictions.

There are very few library functions that you can be sure will use no floating point and make no system calls. Unfortunately, library functions such as **sprintf()**, often used in debugging, must be avoided.

In essence, the ULI handler should only do such things as

- store data in program variables to record the interrupt event

  A ring buffer is a data structure that is suitable for concurrent access.

- program the device as required to clear the interrupt or acknowledge it

  The ULI handler has access to the whole program address space, including any mapped-in devices, so it can perform PIO loads and stores.

- post a semaphore to wake up the main process

  This must be done using a ULI function.

Planning for Concurrency

Since the ULI handler can interrupt the program at any point, or run concurrently with it, the program must be prepared for concurrent execution. There are two areas to consider: global variables, and library routines.

Debugging With Interrupts

The asynchronous, possibly concurrent entry to the ULI handler can confuse a debugging monitor such as *dbx*. Some strategies for dealing with this are covered in the uli(3) reference page.

Declaring Global Variables

When variables can be modified by both the main process and the ULI handler, you must take special care to avoid race conditions.

An important step is to specify -D_SGI_REENTRANT_FUNCTIONS to the compiler, so as to get the reentrant versions of the C library functions. This ensures that, if the main process and the ULI handler both enter the C library, there will be no collision over global variables.

You can declare the global variables that are shared with the ULI handler with the keyword "volatile," so that the compiler will generate code to load the variables from memory on each reference. However, the compiler never holds global values in registers over a function call, and you almost always have a function call (such as **ULI_block_intr()**) preceding a test of a shared global variable.

Using Multiple Devices

The ULI feature allows a program to open more than one interrupting device. You register a handler for each device. However, the program can only wait for a specific interrupt to occur; that is, the **ULI_sleep()** function specifies the handle of one particular ULI handler. This does not mean that the main program must sleep until that particular interrupt handler is entered, however. Any ULI handler can waken the main program, as discussed under “Interacting With the Handler” on page 128.

Setting Up

A program initializes for ULI in the following major steps:

1. Open the device special file for the device.
2. For a VME device, map the device addresses into process memory (see “Mapping a VME Device Into Process Address Space” on page 62.
3. Lock the program address space in memory.
4. Initialize any data structures used by the interrupt handler.
5. Register the interrupt handler.
6. Interact with the device and the interrupt handler.

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

Opening the Device Special File

Devices are represented by device special files (see “Device Special Files” on page 32). In order to gain access to a device, you open the device special file that represents it.

The file that represents the external interrupt lines on a Challenge or Onyx system is */dev/ei*. It can be opened by more than one process at a time, under rules that are spelled out in the ei(7) reference page. The methods of opening */dev/ei*, configuring pulse widths, and generating output pulses are covered in Chapter 6, “Control of External Interrupts,” and remain the same.

The files that represent VME control units are */dev/vme/vme**. The rules for opening one of these files and mapping a device into memory are covered under "VME Programmed I/O" on page 62, and remain the same. The difference is that, with ULI, you can map in a device that can cause interrupts.

The program should open the device and (for a VME device) verify that the device exists and is active before proceeding.

Locking the Program Address Space

The ULI handler must not reference a page of program text or data that is not present in memory. You prevent this by locking the pages of the program address space in memory. The simplest way to do this is to call the **plock()** system function:

```
if (plock(PROCLOCK))
{ perror("plock"); exit();}
```

The **plock()** function has two possible difficulties. One is that the calling process must have superuser privilege (see the plock(2) reference page). This may not pose a problem if the program needs superuser privilege in any case, for example in order to open a device special file. The second is that it locks all text and data pages. In a very large program this may be so much memory that system performance is harmed.

The **mpin()** function can be used by unprivileged programs to lock a limited number of pages. The limit is set by the tunable system parameter *maxlkmem*. (Check its value—typically 2000—in */var/sysgen/mtune/kernel*. See the systune(1) reference page for instructions on changing a tunable parameter.)

In order to use **mpin()**, you must specify the exact address ranges to be locked. Provided that the ULI handler refers only to global data and its own code, it is relatively simple to derive address ranges that encompass the needed pages. If the ULI handler calls any library functions, the library DSO needs to be locked as well. The smaller the scope of the ULI handler, the easier it is to use **mpin()**.

Registering the Interrupt Handler

When the program is ready to start operations, it registers its ULI handler. The ULI handler is a function that matches the prototype

```
void function_name(void *arg);
```

The registration function takes arguments with the following purposes:

- the file descriptor of the device special file
- four arguments related to the ULI handler:
  - the address of the handler function
  - an argument value to be passed to the handler on each interrupt—typically a pointer to a work area that is unique to the interrupting device, supposing the program is using more than one device
  - the size, and an optional address, of memory to be used as stack space when calling the handler
- a count of semaphores to be allocated for use with this interrupt

You can ask the ULI support to allocate a stack space by passing a null pointer for the stack argument. When the ULI handler is as simple a function as it normally is, the default stack size of 1024 bytes is ample.

The semaphores are allocated and maintained by the ULI support. They are used to coordinate between the program process and the interrupt handler, as discussed under "Interacting With the Handler" on page 128. You should specify one semaphore for each independent process that can wait for interrupts from this handler. Normally one semaphore is sufficient.

The returned value is a handle that is used to identify this interrupt in other functions. Once registered, the ULI handler remains registered until the program terminates (there is no function for un-registration).

Registering an External Interrupt Handler

The **ULI_register_ei()** function takes the arguments described in the preceding topic. Once it has successfully registered your handler, all external interrupts are directed to that handler.

It is important to realize that, so long as a ULI handler is registered, none of the other interrupt-reporting features supported by the external interrupt device driver (see Chapter 6, "Control of External Interrupts" and the ei(7) reference page) operate any more. These restrictions include the facts that:

- The per-process external interrupt queues are not updated.
- Signals requested by **ioctl(**EIIOCSETSIG**)** are not sent.
- Calls to **ioctl(**EIIOCRECV**)** sleep until they are interrupted by a timeout, a signal, or because the program using ULI terminated and an interrupt arrived.
- Calls to the library function **eicbusywait()** do not terminate.

Clearly you should not use ULI for external interrupts when there are other programs running that also use them.

Registering a VME Interrupt Handler

The **ULI_register_vme()** function takes two additional arguments:

- the interrupt level that the device uses.
- a word that contains, or receives, an interrupt vector number

The interrupt level used by a device is normally set by hardware and documented in the VECTOR line that defines the device (see "Learning VME Device Addresses" on page 62).

Some VME devices have a fixed interrupt vector number; others are programmable. You pass a fixed vector number to the function. If the number is programmable, you pass 0, and the function allocates a number. You must then use PIO to program the vector number into the device.

Interacting With the Handler

The program process and the ULI handler synchronize their actions using two functions.

When the program cannot proceed without an interrupt, it calls **ULI_sleep()**, specifying

- the handle of the interrupt for which to wait
- the number of the semaphore to use for waiting

Typically only one process ever calls **ULI_sleep()** and it specifies waiting on semaphore 0. However, it is possible to have two or more processes that wait. For example, if the device can produce two distinct kinds of interrupts—normal and high-priority, perhaps—you could set up an independent process for each interrupt type. One would sleep on semaphore 0, the other on semaphore 1.

When an ULI handler is entered, it wakes up a program process by calling **ULI_wakeup()**, specifying the semaphore number to be posted. The handler must know which semaphore to post, based on the values it can read from the device or from program variables.

The **ULI_sleep()** call can terminate early, for example if a signal is sent to the process. The process that calls **ULI_sleep()** must test to find the reason the call returned—it is not necessarily because of an interrupt.

The **ULI_wakeup()** function can be called from normal code as well as from a ULI handler function. It could be used within any type of asynchronous callback function to wake up the program process.

The ULI_wakeup() call also specifies the handle of the interrupt. When you have multiple interrupting devices, you have the following design choices:

- You can have one child process waiting on the handler for each device. In this case, each ULI handler specifies its own handle to **ULI_wakeup()**.
- You can have a single process that waits on any interrupt. In this case, the main program specifies the handle of one particular interrupt to **ULI_sleep()**, and every ULI handler specifies that same handle to **ULI_wakeup()**.

Achieving Mutual Exclusion

The program can gain exclusive use of global variables with a call to **ULI_block_intr()**. This function does not block receipt of the hardware interrupt, but does block the call to the ULI handler. Until the program process calls **ULI_unblock_intr()**, it can test and update global variables without danger of a race condition. This period of time should be as short as possible, because it extends the interrupt latency time. If more than one hardware interrupt occurs while the ULI handler is blocked, it will be called for only the last-received interrupt.

There are other techniques for safe handling of shared global variables besides blocking interrupts. One important, and little-known, set of tools is the **test_and_set()** group of functions documented in the test_and_set(3) reference page. These instructions use the Load Linked and Store Conditional instructions of the MIPS instruction set to safely update global variables in various ways.

Sample Program

The program listed in Example 7-1 is a hypothetical example of how user-level interrupts can be used to handle external interrupts in a Challenge and Onyx system.

Example 7-1 Hypothetical ULI Program

```
/* This program demonstrates use of the External Interrupt source
 * to drive a User Level Interrupt.
 *
 * The program requires the presence of an external interrupt cable looped
 * back between output number 0 and one of the inputs on the machine on
 * which the program is run.
 */
#include <sys/ei.h>
#include <sys/uli.h>
#include <sys/lock.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
/* The external interrupt device file is used to access the EI hardware */
#define EIDEV "/dev/ei"
static int eifd;
/* The user level interrupt id. This is returned by the ULI registration
 * routine and is used thereafter to refer to that instance of ULI
```

```
 */
static void *ULIid;
/* Variables which are shared between the main process thread and the ULI
 * thread may have to be declared as volatile in some situations. For
 * example, if this program were modified to wait for an interrupt with
 * an empty while() statement, e.g.
 *      while(!intr);
 * the value of intr would be loaded on the first pass and if intr is
 * false, the while loop will continue forever since only the register
 * value, which never changes, is being examined. Declaring the variable
 * intr as volatile causes it to be reloaded from memory on each iteration.
 * In this code however, the volatile declaration is not necessary since
 * the while() loop contains a function call, e.g.
 *   while(!intr)
 *      ULI_sleep(ULIid, 0);
 * The function call forces the variable intr to be reloaded from memory
 * since the compiler cannot determine if the function modified the value
 * of intr. Thus the volatile declaration is not necessary in this case.
 * When in doubt, declare your globals as volatile.
 */
static int intr;
/* This is the actual interrupt service routine. It runs
 * asynchronously with respect to the remainder of this program, possibly
 * simultaneously, on an MP machine. This function must obey the ULI mode
 * restrictions, meaning that it may not use floating point or make
 * any system calls. (Try doing so and see what happens.) Also, this
 * function should be written to execute as quickly as possible, since it
 * runs at interrupt level with lower priority interrupts masked.
 * The system imposes a 1-second time limit on this function to prevent
 * the cpu from freezing if an infinite loop is inadvertently programmed
 * in. Try inserting an infinite loop to see what happens.
 */
static void
intrfunc(void *arg)
{
   /* Set the global flag indicating to the main thread that an
    * interrupt has occurred, and wake it up
    */
   intr = 1;
   ULI_wakeup(ULIid, 0);
}
/* This function creates a new process and from it, generates a
 * periodic external interrupt.
 */
static void
```

```
signaler(void)
{
   int pid;
   if ((pid = fork()) < 0) {
   perror("fork");
   exit(1);
   }
   if (pid == 0) {
      while(1) {
         if (ioctl(eifd, EIIOCSTROBE, 1) < 0) {
            perror("EIIOCSTROBE");
            exit(1);
         }
         sleep(1);
      }
   }
}
/* The main routine sets everything up, then sleeps waiting for the
 * interrupt to wake it up.
 */
int
main()
{
   /* open the external interrupt device */
   if ((eifd = open(EIDEV, O_RDONLY)) < 0) {
      perror(EIDEV);
      exit(1);
   }
   /* Set the target cpu to which the external interrupt will be
    * directed. This is the cpu on which the ULI handler function above
    * will be called. Note that this is entirely optional, but if
    * you do set the interrupt cpu, it must be done before the
    * registration call below. Once a ULI is registered, it is illegal
    * to modify the target cpu for the external interrupt.
    */
   if (ioctl(eifd, EIIOCSETINTRCPU, 1) < 0) {
      perror("EIIOCSETINTRCPU");
      exit(1);
   }
   /* Lock the process image into memory. Any text or data accessed
    * by the ULI handler function must be pinned into memory since
    * the ULI handler cannot sleep waiting for paging from secondary
    * storage. This must be done before the first time the ULI handler
    * is called. In the case of this program, that means before the
    * first EIIOCSTROBE is done to generate the interrupt, but in
```

```
     * general it is a good idea to do this before ULI registration
     * since with some devices an interrupt may occur at any time
     * once registration is complete
     */
    if (plock(PROCLOCK) < 0) {
       perror("plock");
       exit(1);
    }
    /* Register the external interrupt as a ULI source. */
    ULIid = ULI_register_ei( eifd,   /* the external interrupt device */
                        intrfunc,   /* the handler function pointer */
                        0,      /* the argument to the handler */
                        1,      /* the number of semaphores needed */
                        NULL,   /* the stack to use (supply one) */
                        0);     /* the stack size to use (default) */
    if (ULIid == 0) {
       perror("register ei");
       exit(1);
    }
    /* Enable the external interrupt. */
    if (ioctl(eifd, EIIOCENABLE) < 0) {
       perror("EIIOCENABLE");
       exit(1);
    }
    /* Start creating incoming interrupts. */
    signaler();
    /* Wait for the incoming interrupts and report them. Continue
     * until the program is terminated by ^C or kill.
     */
    while (1) {
    intr = 0;
    while(!intr) {
       if (ULI_sleep(ULIid, 0) < 0) {
          perror("ULI_sleep");
          exit(1);
       }
       printf("sleeper woke up\n");
    }
}
```