

LEARN HOW TO DEVELOP AN ISR AND

ALLOCATE MEMORY.

Get those boards talking under Linux, part 2

PART 1 OF THIS ARTICLE explained how to register a driver with the Linux kernel, how to name a driver, how to call a driver function, and how to initialize a data-acquisition board (Reference 1). Part 2 explains how to develop an ISR (interrupt-service routine) and how to allocate system memory so you can store your data.

Interrupts make an OS (operating system) pause and service the board generating the interrupt. A board can generate an interrupt in response to specified conditions, such as when the input buffer becomes half-full.

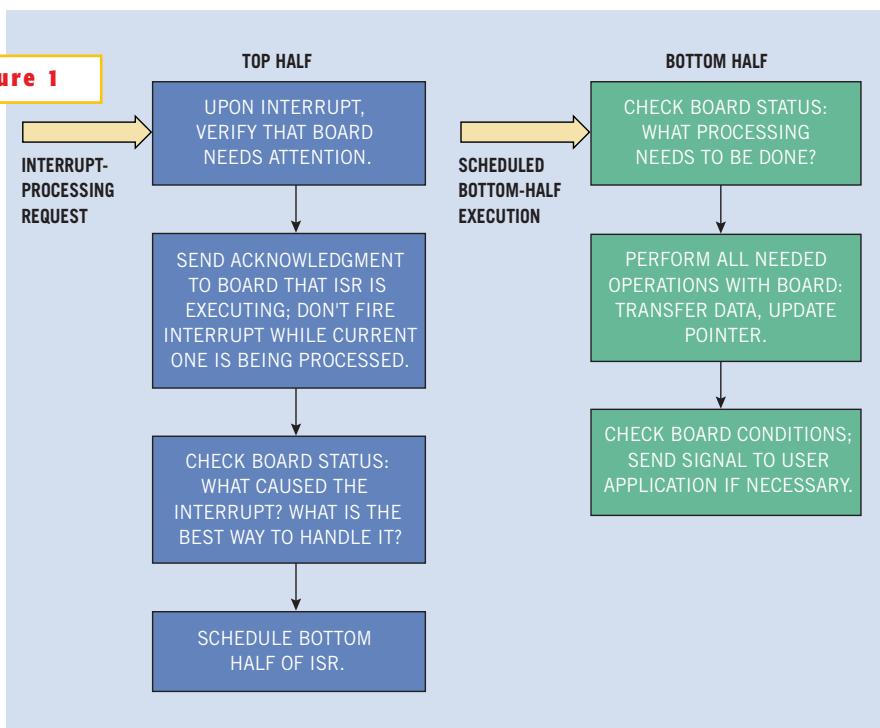
When a board requests service, the OS needs to know which board generated the interrupt and what caused the interrupt. Only then can it know which driver to notify about the interrupt so that the driver can run its appropriate ISR functions. Some interrupts require immediate attention, whereas others can wait until the OS is best able to service them.

To best service interrupts in the Linux OS, you should design your ISR with two parts. In Linux lingo, the two parts are called the top half and the bottom half (Figure 1). Splitting the ISR lets your driver first handle those activities that require immediate attention and then handle other services according to a schedule. Your driver

should immediately verify which board generated the interrupt and what caused the interrupt. Then, the bottom half can service the interrupt when the OS can better devote time to the ISR. Using a two-part ISR design minimizes the time that the OS pauses, which in turn minimizes the chance that your board will lose data.

For your driver to service an interrupt, it needs to know which interrupt number the motherboard BIOS has assigned to the PCI-bus data-acquisition board. To find this information, call the kernel function `pci_find_device()` and examine the structure it returns. You should find information such as which interrupt line and pin the BIOS assigned to the card. The code in Listing 1 uses the returned information

Figure 1



By splitting the duties of an ISR, you can place those tasks that the system must handle immediately into the top half and use the bottom half for ISR tasks that don't require immediate attention.

This article first appeared in our sister publication Test & Measurement World, www.tnmworld.com.

to initialize interrupt-related variables.

The `pci_dev` structure in **Listing 1** also contains address information. A PCI card can have six address regions for I/O or memory. You need these addresses when writing the driver code that performs the I/O operations. To find a card's base address, use the following line of code:

```
BaseAddress0 = dev->
base_address[0]; // base
device address - region 0
```

When a board needs service, it issues an IRQ by placing a signal on one of the interrupt lines. The kernel then invokes every registered ISR associated with this line. You must also inform the Linux kernel which interrupt corresponds to a specific function in your driver. To make the association, you must call the kernel function `request_irq()` from one of two places. In the first approach, you place the call to `request_irq()` inside your `init_module()` routine, which initializes the data-acquisition card. Note, though, that, in this case, that driver becomes the sole owner of the IRQ line, which causes a problem because PC peripherals often have to share an IRQ line.

You can also place the call inside the `open()` function. In this case, the application requests an interrupt when one of its processes first opens the device driver and initializes the hardware. Then, when the hardware issues an interrupt, it uses the interrupt line the device driver has reserved for that purpose. The driver holds ownership of the IRQ line and releases it only when the last process that might need the interrupt closes the driver. Thus, the driver releases the IRQ line when idle so that other drivers can

use that line. The downside of this approach is that it requires you to write extra code to give the driver control over when the application gets ownership of the interrupt line.

LINUX SUPPORTS INTERRUPT SHARING

As mentioned, PC hardware often has to share interrupts. A standard PC defines only a limited number of interrupts. If a driver's ISR takes hold of one IRQ line, that line becomes unavailable to other processes. To help alleviate the problems caused by the limited number of IRQ lines on the PCI bus, you should design the driver so that several devices share one IRQ line. For example, one IRQ line can handle interrupts from multiple PCI boards in the same backplane; even data-acquisition cards from several manufacturers can share an interrupt line. Linux began supporting interrupt sharing with kernel Version 2.0. And, although the kernel authors wrote the sharing scheme with the idea of supporting the PCI spec, it is nevertheless useful when you're working with ISA

boards that share interrupts in hardware. The code in **Listing 2** installs a shared-interrupt handler with the address `pd_isr`. Note that by using the vertical bar, you can OR the flag `SA_SHIRQ` with `SA_INTERRUPT`. This code tells the kernel of your intention to have two devices share an interrupt line.

Because an ISR can service more than one interrupt, each ISR must communicate with the devices it serves to locate the board that issued the interrupt. You can examine a board's status flag to see how to identify which board issued the interrupt (**Listing 3**). (Even though devices can share interrupts, each maintains a separate I/O space.) The code in **Listing 3** uses the driver's `pd_isr()` function. The kernel knows which of the incoming interrupts needs servicing with that function because you previously made the association between that interrupt and the `pd_isr()` function in the kernel with the `request_irq()` function.

The ISR enumerates all the boards installed and determines which one generated the interrupt. Using the code in **Listing 4**, the ISR services the board. The code also schedules the bottom half of the ISR, which performs tasks that are not time critical.

The ISR code in **Listing 4** first checks what caused the interrupt. It sends an acknowledgment to the board to confirm that it is processing

LISTING 1—RETURNED INFORMATION FOR INTERRUPT-RELATED VARIABLE

```
struct pci_dev *dev; // pointer to PCI device structure used by
// pci_find_device()
...
InterruptLine = dev->irq; // device IRQ
pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &u8val);
// device interrupt pin
```

LISTING 2—SHARED INTERRUPT HANDLER

```
if (request_irq(the_board[board].irq, // IRQ
pd_isr, // ISR handler address
SA_SHIRQ | SA_INTERRUPT, // request flags
"PowerDAQ", // device name
(void *)&the_board[boards])) // unique device ID equal to
// address of the_board structure

return -ERR;
```

LISTING 3—STATUS-FLAG EXAMINATION

```
static void pd_isr(int trigger_irq, void *dev_id, struct pt_regs *
regs)
{
int board;
// if any of our boards are registered to this IRQ, service them
for (board = 0; board < num_boards; board++)
if (&the_board[board] == dev_id)
if (pd_isr_serve_board(board)); // OSAL call
```

the interrupt, and then it retrieves the board's status to determine the cause of the interrupt. Once a board receives an ISR acknowledgment, it shouldn't issue a new interrupt until the driver releases the current interrupt and enables the board's interrupt-requesting ability.

Releasing interrupts is part of the rationale behind splitting the ISR, which avoids the problems with one-part, or "atomic," processing. With one-part processing, the ISR proceeds from beginning to end without interruption, and Linux disables all interrupts. If a board tries to trigger the same interrupt line during that processing time, you forever lose that interrupt.

To avoid losing interrupts, your ISR should use as little processor time as possible and should perform no long operations (such as non-DMA transfers). Instead, the ISR's top half should perform immediate operations, such as notifying the kernel about the interrupt and deciding what to do next. In some situations, you should perform data transfers in the top half of the ISR.

Once the top half of the ISR decides it needs to process the bottom half, it must schedule that additional process for execution (**Listing 5**). After the ISR's top half schedules the activities in the bottom half, the bottom half can do its work. Typically, the bottom half gets the board's latest status, which can change after the top half executes and disables the interrupts. Therefore, the board might have been unable to issue a notification that its status had changed.

The bottom half should also take action based on the cause of the interrupt.

LISTING 4—ISR-SERVICING CODE

```
static int pd_isr_serve_board(int board)
{
    ...
    // check if interrupt came from a specific card
    if ( ! pd_check_int_request(board))
        return 0;           // this board not waiting for service
    pd_ack_int_request(board); // acknowledge the interrupt
    // do whatever necessary to service interrupt request, for example
    pd_get_status(board, &status);
    if (status & BRD_DATA_AVAILABLE)
        ...                // proceed according with board status
                                // schedule bottom half of ISR to run
    pd_schedule_dpc(board, board_dpc_proc);
    return 1;
}
```

For instance, if a board supports bus mastering, the interrupt might acknowledge that a portion of the data has been transferred into host memory, or it might request the physical address of the next page of memory if it needs to place more data into system memory.

If your hardware supports no bus mastering, you may need a small "emergency" data transfer. For example, the analog-input subsystem of the data-acquisition board might trigger an interrupt to signal that the card's input FIFO buffer is almost full. For temporary, yet immediate, relief of the situation, the top half might transfer 100 or fewer samples without eating much processor time.

After satisfying any emergency conditions and servicing the interrupt, your driver should:

- Check whether the hardware and ISR satisfy the conditions of which the user application wants to be informed. If they meet these conditions, the driver should employ the SIGIO signal inherent in Linux and send it to the application program. SIGIO is a signal that informs a user application that an asynchronous I/O event has occurred. If the hardware and ISR do not meet the con-

ditions, the driver should not employ SIGIO.

- Re-enable board interrupts. The bottom half is the best place to re-enable interrupts because the driver should complete its processing here, after which it is ready to receive the next IRQ.

MAKE IT ASYNCHRONOUS

Interrupts are asynchronous events, and they can occur at any time. So, you might write a driver that permits communications between it and the calling application program to run asynchronously (also known as overlapped I/O) as well as synchronously (known as blocked I/O).

The differences between the two are important. When the application program makes a function call or sits in a tight loop checking for a status flag, synchronous operation freezes the application until the driver operation is complete. The application program can't execute anything else during this period and wastes processor time. Querying the driver or the hardware for a board's status in a tight loop makes the system appear sluggish. Thus, you might set up your application program to query the driver or hardware less frequently, such as between computations. But this approach can make the application appear to execute sporadically, and it also runs the risk of missing a fast-occurring condition. You have to weigh the advantages and disadvantages of both synchronous and asynchronous communications when designing your driver.

LISTING 5—PROCESSING THE BOTTOM HALF

```
bh_task.data = (void *) board_parameters;
    // pass any parameters bottom half might need
queue_task(&bh_task, &tq_immediate);
    // schedule bottom half to run
mark_bh(IMMEDIATE_BH);
    // activate bottom half execution
```

In contrast, with asynchronous operation, the application program makes a driver call, and the driver later notifies the application program that the operation is complete. The driver can notify the application in several ways, so you must determine how to get that notification.

PROGRAMMING EVENT NOTIFICATION

Having the application know that the driver has completed a function is important. Polling in a loop is inefficient, but, fortunately, Linux provides signals, such as SIGIO, that can inform an application when something happens.

Consider a case in which you want to enable asynchronous notification of activity from the device subsystems, such as the board's analog-input subsystem. From the shared library or application side, the code looks like that in **Listing 6**.

The `fcntl()` function implements a file-control operation that lets a process claim ownership of the file associated with a device. The file descriptor `ain_fd` makes up a parameter you receive from the `open()` function when first using the analog-input subsystem. Next, the program gets and resets a status flag from the device file. Notice that the OR operation with `FASYNC` (the asynchronous flag) enables asynchronous notification. When new data arrives, the input file generates the SIGIO signal. The program must set up a handler to receive and react to that signal (**Listing 7**). The important system call in this code is the `sigaction()` function. It tells the system to invoke a handler function in the user application upon receiving the SIGIO signal.

Before you call the `sigaction()` function, be sure to fill its `io_act` structure with a pointer to your signal handler.

The driver side is also uncomplicated. The application calls `fcntl()` with the `F_SETFL` operation and the `FASYNC` flag as parameters. These parameters tell `fcntl()` which of the many driver functions to call; in this case, the kernel calls the driver's `fasync` method. That method, `pd_fasync`, dedicates all its effort to maintaining the structures needed for the `fasync_helper()` function that resides in the Linux kernel (**Listing 8**).

Now, when the data-acquisition board generates an interrupt and the driver

wants to notify the user application, the driver runs this line of code:

```
kill_fasync(pd_board[board].fasync,
SIGIO);
```

Finally, when closing the driver after use, release all asynchronous readers with this line:

```
pd_fasync(-1, file, 0);
// release any asynchronous readers
```

GET AND KEEP ENOUGH MEMORY

Besides handling interrupts when you develop a driver, you also need to address how you use system memory. In data ac-

LISTING 6—ENABLING ASYNCHRONOUS NOTIFICATION

```
// setting ownership of device file
if (fcntl(ain_fd[board], F_SETOWN, getpid()) != 0) {
    // report error, free allocated memory and resources and return
}
// getting flags of device file
flags = fcntl(ain_fd[board], F_GETFL);
if (flags == -1) {
    // report error, free allocated stuff and return
}
// setting flags of device file
if (fcntl(ain_fd[board], F_SETFL, flags | FASYNC) == -1) {
    // report error, free allocated resources and return
}
```

LISTING 7—SIGNAL HANDLER

```
struct sigaction io_act;

// prototype: void sigio_handler(int sig)
io_act.sa_handler = sigio_handler; // set address of handler procedure
sigemptyset(&io_act.sa_mask);
io_act.sa_flags = 0;
if (sigaction(SIGIO, &io_act, NULL) != 0) {
    // Error: failed to set signal action
    // free allocated memory and resources and return
}
```

LISTING 8—FASYNC_HELPER() FUNCTION

```
static int pd_fasync(
    int fd,
    struct file *file,
    int mode
) {
    int board;
    board = MINOR(file->f_dentry->d_inode->i_rdev) / PD_MINOR_RANGE;
    return fasync_helper(fd, file, mode, &pd_board[board].fasync);
}
```

quisition, the board often streams large amounts of data into the host PC's memory. Often, you must reserve sufficient memory to capture enough data to make your application useful. You also need additional memory because Linux in its standard form isn't a true real-time system, and you should allocate enough memory to run an acquisition if the OS encounters delays in calling the driver code.

Most data-acquisition boards are PCI-bus masters and can move large amounts of data into system memory without host intervention. The simplest way to reserve sufficient memory for such transfers is to allocate a big enough area in the PCI configuration space and map onboard memory to the bus. The beauty of this approach is that it requires no driver support; the hardware takes care of all data transfers. The driver merely maps that piece of physical memory into system virtual memory. Then, when the system-virtual memory fills, the board issues an interrupt to the driver, which moves data to another location. One potential pitfall of this method is that, in most PCI chip sets, the PCI configuration-space address range often limits the buffer size to 16 kbytes.

Another way of performing the bus-mastering process has the driver allocate a big chunk of contiguous memory and then pass information about the start address and size of available memory to the board. This solution improves upon reserving memory ahead of time because it lets the driver allocate larger amounts of memory because the PCI configuration-space address range doesn't limit the allocation.

Unfortunately, you can't be certain that the driver can obtain consecutive memory in blocks of sufficient size. And just how large should you make these blocks? To keep up with real-time acquisition, a driver should be able to buffer 0.33 to 1 sec of samples. For a 1.25M-sample/sec input channel, the buffer should be roughly 1 Mbyte.

It's unlikely that the kernel can always supply a contiguous chunk this large. Thus, you can allocate memory at boot time. Here, though, the driver consumes memory whether it needs it or not. Further, the driver doesn't know how much memory the application needs before

that program actually executes.

The proper way to organize a bus master involves using a full scatter-gather implementation. When the application configures a board for operation, it uses the Linux function `malloc()` to allocate sufficient virtual memory in the user space and passes the address to the driver. Your driver should retrieve virtual addresses of allocated memory and translate them into a list of corresponding physical addresses for the memory pages. The driver should send this list to the hardware. A data-acquisition board fills these pages with digitized data and interrupts the driver when the board fills some or all of the memory pages. Most data-acquisition boards support this process.

Allocation requires only a simple line of code:

```
ain_buffer = (uint16_t *)
malloc(size);
```

Be sure to pass out buffer information in `pd_buffer_info`, which you register with the driver through an `ioctl()` call:

```
ioctl(ain_fd[board], IOCTL_PWR-
DAQ_REGISTER_BUFFER,
&pd_buffer_info)
```

This function informs the driver about the size, address, and parameters of a data buffer in the user space. The driver should translate virtual addresses into physical ones and should make certain that memory pages are physically present in system RAM when the board tries to access the memory. □

REFERENCES

1. Ivchenko, Alex, "Get those boards talking under Linux," *EDN*, June 22, 2000, pg 153, www.ednmag.com/ednmag/reg/2000/06222000/13df2htm.
2. Rubini, A, *Linux Device Drivers*, O'Reilly & Associates, Sebastopol, CA, 1998.

AUTHOR'S BIOGRAPHY

Alex Ivchenko, PhD, is R&D engineering manager at United Electronic Industries (Watertown, MA), where he was one of the major developers of the company's data-acquisition boards. He has most recently spent his time writing Linux drivers for these cards. You can reach him at aivchenko@ueidaq.com.