Linux interrupted

Thomas Besemer - August 01, 2000

EmbeddedSystems PROGRAMMING

To read original PDF of the print article, click here.

Internet Appliance Design

Linux, Interrupted

Thomas Besemer

Every operating system has different mechanisms and approaches to handling interrupts. This article lays out the specifics of handling interrupts in Linux. It presents an overview of the Linux runtime environment, discusses how to communicate with hardware peripherals, and provides a working example of servicing an interrupt from application code.

A significant difference between the Linux execution environment and typical real-time operating systems is the memory model. Linux, natively, executes with protected memory space: processes are isolated from other processes through the kernel and underlying hardware memory management unit (MMU). Processes are also isolated from the underlying hardware-application code can't directly read and write peripheral registers. Figure 1 provides a visual representation of the Linux execution environment.

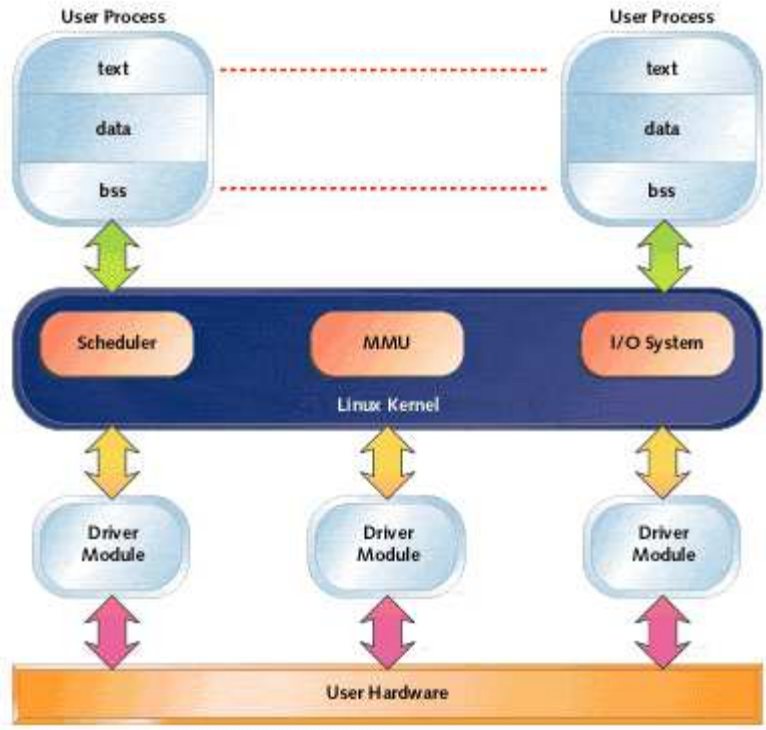

Figure 1: Linux execution environment

Almost all embedded operating systems can be depicted as in Figure 1; the primary difference with Linux is that each "layer" in this figure is truly isolated from the others. With just a handful of exceptions, embedded operating systems typically use a "flat" memory model, in which all software components in the system (OS, application tasks, device drivers, and so on) are able to read and write from and to any location in memory. Device drivers may or may not be employed in typical environments, because the developer has the choice of either using a device driver or communicating directly with hardware devices from within the application code itself.

In a Linux environment, application code runs as a set of processes and/or group of POSIX threads within one process. In order for the processes and threads to communicate with underlying hardware, the developer must install one or more device drivers to support the various hardware peripherals. There are two basic schemes for communication with hardware in a Linux environment. The first is to place the peripherals on a PCI bus. The second approach is to memory- or I/O-map the peripherals. Both approaches require a device driver.

With PCI-based peripherals, the application-level processes or threads may share an underlying PCI driver to communicate with one or more peripherals, supporting and interfacing with them through this common driver. In this configuration, the PCI driver provides support for interrupts from the peripherals. Application processes or threads read or write the peripheral, and may be blocked within the kernel space until the driver services an interrupt.

With memory- or I/O-mapped peripherals, the developer must design and implement a specific driver to support each peripheral. Interrupts from peripherals are handled at the driver level, and it is important that the application developer understand how to approach servicing the interrupt from the application (process) level.

This article discusses how to work with memory- or I/O-mapped peripherals. It concentrates on the basic structure of a device driver that supports interrupts.

Hardware interface

When PCI is not employed, the system designers must map the hardware's registers into a space that is accessible to device driver code. The registers may be mapped into memory space, with the peripheral appearing as a chunk of "memory" in the system. Or, if the processor architecture supports it, the registers may be mapped into a dedicated I/O space. For example, x86 processors provide an I/O space that can be accessed through the assembly language instructions in and out. A basic understanding of both memory and I/O-mapped peripherals is extremely helpful.

The PC platform provides a convenient foundation for experimentation with Linux. All driver and code examples in this article were tested on a PC running Red Hat Linux.

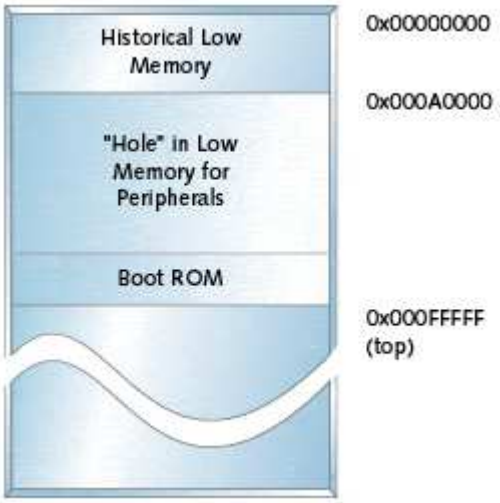

Figure 2: PC memory layout

Figure 2 provides a block diagram of the basic PC platform memory layout. The most notable aspect of this block diagram is the memory region below 1MB. This region has ties back to the original IBM PC and 8086 architecture, which had only 20 bits of address space. A "hole" sits in memory, between 640K and the start of the boot ROM (also known as the BIOS). This "hole" is where VGA cards and other memory-mapped PC peripherals placed their shared buffers in the pre-PCI era.

This is important because designers planning to use a standard PC-style platform for their product must keep the following in mind:

- The ISA/EISA bus has only 20 address bits, so memory-mapped peripherals must reside below 1MB
- Memory-mapped I/O must fit into the location between 640K and the bottom of the boot ROM, and must not overlap in

memory space with any other hardware present in the system

The Linux kernel ignores this region of low memory in that it does not try to use it. However, device drivers are able to read and write from and to this space, allowing communication with any potential hardware devices that decode in this region of memory. When working with custom hardware, or processors such as the PowerPC, hardware designers will be required to implement specialized decode and mapping circuitry to map peripherals into memory space. Additionally, there may be requirements to change the Linux kernel MMU page tables and mappings to allow addressing of these memory regions.

For example, if the hardware designer designs a PC-compliant motherboard, yet places specialized peripherals at 0xA0000000, the MMU tables will need to be updated to allow access to that physical location. Since Linux is distributed with source code, this is a relatively minor issue. It's just important that the developer be aware of it.

Another approach, when using a PC-style platform, is to map peripherals into I/O space. I/O space is for peripherals that respond to special bus cycles from the processor. These cycles are generated through special-purpose assembly language instructions. This article is based on a hardware environment that resides in I/O space.

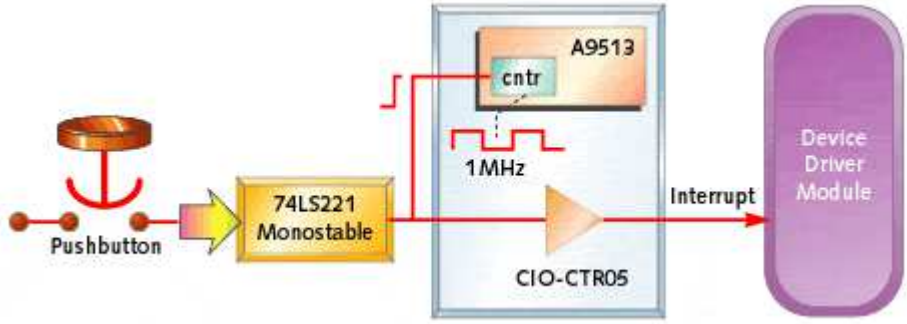

Figure 3: Test fixture configuration

Figure 3 provides a block diagram of the hardware and configuration used for the examples in this article. This hardware test bed consists of a pushbutton that is debounced through a 74LS221 Monostable Multivibrator on a breadboard, and an off-the-shelf counter/timer board (CIO-CTR05) from Computer Boards (www.computerboards.com). The CIO-CTR05 contains an AMD 9513 counter/timer device, which has five 16-bit counter/timers. Additionally, the CIO-CTR05 provides eight digital inputs and eight digital outputs, as well as the ability to assert an interrupt to the system.

In my experiments, I used one of the 9513's 16-bit counters and the logic on the CIO-CTR05 to generate an interrupt. When the pushbutton is depressed, the Monostable generates a 300ms pulse. This pulse, on its active edge, asserts an interrupt to the processor and starts the timer counting. The counter is clocked at 1MHz.

A Linux device driver services the interrupt. When its interrupt handler is called-from the Linux kernel's "low-level interrupt handler"-it notes the value in the counter and then potentially

alerts application code that the interrupt has been asserted. If the application code has provided an "interrupt" handler, that handler will execute in process space and perform a read of the device driver to stop the counter and read it's value. Through this scheme, the following is accomplished:

- A driver is implemented to support the interrupt and counter, and provides a means to alert application level code
- Instrumentation is put in place to characterize and provide timing analysis of interrupt operation in Linux

Driver basics

Linux device driver constructs and implementation details are beyond the scope of this article. This information may be found in books such as *Linux Device Drivers* by Alessandro Rubini (O'Reilly & Associates, 1998).

As a foundation for this article, a driver for the CIO-CTR05 was located at a site at North Carolina State University (ftp://lx10.tx.ncsu.edu/pub/Linux/drivers). This driver provided me with a head start for putting the test fixture in place. (One of the great things about working with Linux is the tremendous amount of source code that's already out there to supplement it.)

The driver from NCSU is fairly robust, supporting many of the functions of the CIO-CTR05 board. In order to present the concepts within the bounds of this article, the basic driver was scaled down to support only the topics discussed within. As each aspect of the driver is discussed, working source code is presented to support the discussed concepts. It is also important to note that several types of drivers exist in the Linux environment character, block, and network. The driver discussed in this article is a character device driver.

Linux device drivers are considered "modules," and these modules may be loaded dynamically at runtime. This means that the developer is able to implement, compile, load, test, and then unload the driver. This cycle may be repeated over and over without rebooting the Linux machine, providing a suitable environment for developing a driver.

Listing 1: Output from lsmod

The list of installed modules can be obtained with the **lsmod** command. This command displays what modules are loaded, and what components in the system are using each module. Listing 1 provides the output from **lsmod** in the system on which the driver in this article was developed.

The module tulip is the device driver for a PCI network card, while the module ctr05 is the driver for the CIO-CTR05. The module ctr05 was installed with the following command:

/sbin/insmod -f ctr05.o

This module can be removed during runtime with the following command:

/sbin/rmod ctr05

Every device driver module has two key functions:

- **init_module()** -Responsible for installing the driver in the kernel; called by the kernel when the module is loaded
- **cleanup_module()** -Responsible for doing any hardware shutdown or internal cleanup when a module is removed from the kernel

Listing 2: CIO-CTR05 module initialization

The procedure **init_module()** registers the driver with the kernel, telling the kernel about other internal functions such as **read()** , **write()** , or **ioctl()** . Additionally, if the driver supports interrupts, it requests that the kernel call an internal function when the specified interrupt is asserted. Finally, this procedure is responsible for configuring and initializing the hardware. Listing 2 shows the initialization function for the CIO-CTR05. Note that to keep the listing short for this article, most of the error checking has been removed. However, this listing does present the basics of the initialization of the module. The following key steps are performed:

1. Registering the module as a character device driver. After this, application code running in user space may perform **open()** , **close()** , **read()** , **write()** , or **ioctl()** operations on the module
2. Checking for and requesting a "region." This allows the driver to communicate with the physical hardware device (which is, in this case, the CIO-CTR05 board)
3. Registering an interrupt handler with the kernel. This operation associates a physical interrupt level with a module-based handler
4. Configuring the hardware

Removing a module from the kernel requires that the reverse be done: release the region, release the interrupt, and un-register the device.

Listing 3: CIO-CTR05 module removal

Listing 3 shows the procedure **cleanup_module()** used in the CIO-CTR05 driver. When reviewing this listing, note the function **printk()** . This function is identical to **printf()** , with the exception that it operates at the kernel level. Diagnostic messages generated with **printk()** are sent to the Linux console. For all practical purposes, this is the most useful debug tool for implementing Linux drivers and interrupt handlers.

Application code

To this point, we have discussed the concept and basics of a Linux device driver. The important points to note are as follows:

- Interrupts must be serviced by a driver
- Hardware may only be communicated with through a driver

Specific operation on an interrupt will be discussed in a bit. First, we need to understand how application-level (process) code running in user-mode interacts with the kernel-mode driver. Device drivers look like files to application code. Their primary interface is through special files called device files, which are generally located in the /dev directory in the Linux file system. Device drivers have associated with them major and minor device numbers. The major device number is used to associate a device file with a device driver

(module). Listing 2 shows that when the module is registered with the kernel, it passes in a major device number.

Listing 4: CIO-CTR05 device files

Device files in the **/dev** directory are created by the module developer using the Linux command mknod. Listing 4 shows the device files for the CIO-CTR5 module. This listing was made by issuing the command **ls -rtl /dev/ctr*.**

The first field indicates that the module is a character device **(c)** that can be both read and written **(rw-)** by users or processes running as root, but only read **(r--)** by other users. The fifth field, 50 in this example, is the major device number, while the sixth field, 0 and 1 for these device files, is the minor device number. Minor numbers may be selected in any way that makes sense to the driver developer. In the CIO-CTR05 driver, they represent and specify one of the five counters in the AMD 9513 contained on the I/O board.

Finally, **/dev/ctr05_DIO** and **/dev/ctr05_CTR1** are device file names. These names generally represent something meaningful to the developer. In this case, ctr05 identifies the module, while DIO states that this file is for operating on the digital I/O portion of the driver, and **CTR1** is for counter 1 control. These device files were created by executing the following commands:

/bin/mknod /dev/ctr05_DIO c 50 0
/bin/mknod /dev/ctr05_CTR1 c 50 1

Application code operates on these by first performing either an **open()** or **fopen()** on the device file. After that, the returned file descriptor or stream pointer may be used to **read()** , **write()** , **ioctl()** , or **close()** the specified device (module). Internal to the module are procedures to support each of these functions. These procedures behave in a consistent way through standardized calling conventions, but internally perform operations specific to the peripheral.

Interrupt delivery

When an interrupt is asserted, the developer has two choices for processing it:

- In the driver, at kernel level
- In user space, in the application process or processes

Most applications will perform processing of an interrupt in both places. The driver must contain some logic for handling the interrupt, as the context of the driver is the only location that the interrupt handling code may execute. The driver services the interrupt, possibly doing some minor operations on the peripheral, and then alerts the application.

Getting the message to the application must be done in a unique fashion under Linux, as the driver cannot call interrupt handlers running in user (process) space. The driver may only execute in kernel space. It's important to note that the application must be expecting and waiting for the interrupt to occur, in order to respond to it.

The best approach is often to use an *interrupt dispatch* process or thread that waits for all interrupts serviced by the driver module

(as the module may service hardware devices capable of generating several different interrupt types through the same IRQ line).

When an interrupt occurs and the dispatch process is placed into execution, it sends a message to the appropriate process or thread, based on the interrupt type serviced by the driver ISR. The process or thread then handles the interrupt, while the dispatch process waits for the next interrupt. In practical operation, this process opens the appropriate module device file, then makes (typically) an **ioctl()** or **read()** call that "blocks," suspending the caller and allowing other processes and threads to run while the "dispatch" process waits for the interrupt.

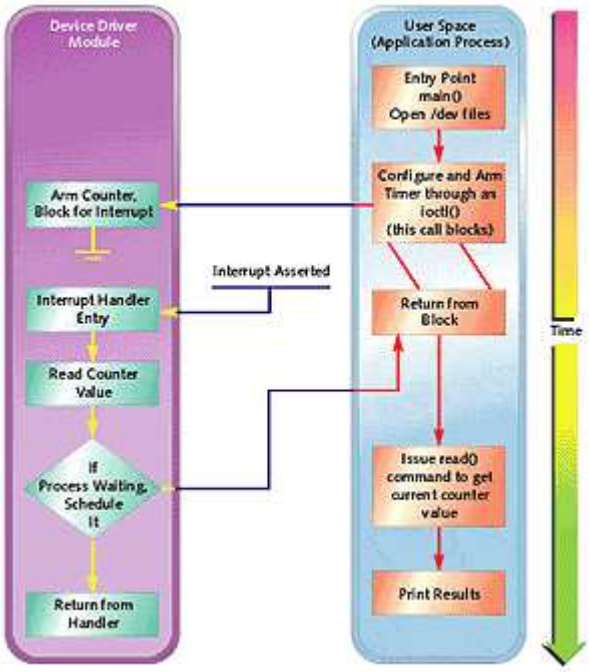

Figure 4: Sequence of events

Figure 4 shows the sequence of events for the following discussions. Each block in this figure is described in detail in subsequent paragraphs.

The interrupt handler in the driver module executes when an interrupt is asserted. It does minor processing of the interrupt, then checks to see if an application process is waiting for that interrupt. If one is waiting, the interrupt handler alerts the process, and the Linux scheduler places it into execution based on its priority within the system.

Listing 5: CIO-CTR05 module interrupt handler

Listing 5 shows the interrupt handler used in the CIO-CTR05 device driver module. It is extremely important to note that this interrupt handler executes in the context of the kernel. This interrupt handler performs the following functions:

- Reads the counter value (the counter started counting at the moment the interrupt was asserted to the processor)
- Makes a call to kernel function **waitqueue_active()** . This function checks to see if an application process has blocked on a user-specified Wait Queue called **int_wq**

- If a process is blocked on the Wait Queue, waiting for the interrupt, a call is made to **wake_up_interruptible()** . This function unblocks the process, and the kernel will place it into execution based on its specific scheduling rules (regular Linux process vs. real-time Linux process)

Listing 6: CIO-CTR05 module ioctl() function

Listing 6 shows the CIO-CTR05 module logic (which allows user processes to block) waiting for the interrupt. This logic is implemented as an **ioctl()** operation in the driver. Like the interrupt handler, this procedure executes in the context of the Linux kernel.

The core logic in Listing 6 follows the **if** statement to see if the **ioctl ()** operation specified by cmd is **WAIT_FOR_INTERRUPT** . In this case, the following occurs:

1. The counter is armed and its value is reset to zero, and it is configured to wait for the edge transition from the monostable, which occurs when the pushbutton is depressed. When this edge transition occurs, the counter begins counting at a 1MHz rate
2. The counter makes a call to **interruptible_sleep_on()** , a kernel function that blocks the calling process until **wake_up_interruptible()** is called

Listing 7: Application process for interrupt

After returning from **wake_up_interruptible()** , the **ioctl()** call returns to the user's process, resuming execution in the user's process at the location following the **ioctl()** call. Listing 7 shows the application process that makes the **ioctl()** call. This code fragment executes in the context of Linux user space.

In the function **waitForInt()** , the **ioctl()** function is called, which blocks the caller until the interrupt occurs. Upon return (interrupt asserted), a **read()** is done on the module to get the current value of the counter. This integer data contains two fields; the counter's value at the time the interrupt handler executed, and the value at the time the **read()** function was called.

Listing 8: open() on /dev devices

Listing 9: CIO-CTR05 read() function

The file descriptors used for the **ioctl()** and **read()** calls were acquired from a standard **open()** call. Listing 8 shows the application code that opened these descriptors. Listing 9 shows the **read()** function as implemented in the CIO-CTR05 module. Like the **ioctl()** and interrupt handler fragments, this fragment executes in the context of the Linux kernel.

The function **ctr05_read()** performs the following functions:

- Reads the current value of the counter through the function read_counter(minor). The minor number is 1 in this case from the device file **/dev/ctr05_CTR1** , indicating Counter 1
- Combines the data with the counter value saved when the interrupt handler executed
- Copies the data into user process space (isolated from kernel and other process space by the MMU) and returns

It is important to note that this call does not block, and returns immediately to the calling application process after transferring the data into the user buffer identified as **buf** .

Table 1: Timing data, no load

Table 2: Timing data, system loaded

Interrupt characterization

The test fixture presented in this article provides for basic characterization of response times within Linux. The examples were tested on a 100MHz Pentium processor with 80MB of RAM. Table 1 shows the response times for five back-to-back interrupts (initiated through the pushbutton) with no load on the system. The numbers represent the number of ticks from the time the interrupt was asserted to the processor. The ticks increment at 1MHz, representing time in microseconds.

Listing 10: Processor loading script

Listing 10 shows a simple shell script that was used to place a load on the processor.

The command:

ping -f tbcorp1 &

causes the local machine to ping the machine tbcorp1 as quickly as the network interface is able to send and receive packets. In the test environment, this causes heavy PCI activity, heavy interrupt activity, and mild CPU loading.

The "while" loop **tar**s up **/usr** into a file, then removes it, over and over. The **tar** command operates verbose, which causes a considerable amount of text information to be sent to the console. Additionally, the tar command itself causes lots of disk activity, which causes interrupts and heavy traffic on the bus.

Table 2 shows the results of five back-to-back interrupts through the test fixture while this load executed. The lack of consistency is notable. With some interrupts, such as Assertion 5, the numbers are almost as low as when the tests were run without load. In other cases, such as Assertion 1, interrupt latency was not terrible, but the amount of time before the application process executed was lengthy. The worst case measurement is Assertion 3, which shows long interrupt latency and long latency to process execution.

These numbers provide some starting points for analysis. These were taken using standard Red Hat Linux, with no real-time extensions in place. The application process was a regular Linux process vs. a real-time Linux process. (Real-time Linux processes execute at a higher priority than any other process. Because of this, interrupt latency is an issue, whereas latency time in process execution is not.) Additionally, a real-time Linux project is under way which solves interrupt off-time problems, and has improved scheduling. Regardless of what version or flavor of Linux you use, if you have hardware devices that your application code must communicate with, and these devices have interrupts, you will need to design, implement, and install a driver module as discussed in this article.

Thomas Besemer has been developing embedded software since 1980, using a variety of tools, processors, environments, and approaches. He is the president of a consulting firm, a supporter of public radio, and a photographer, who spends as much time as he can enjoying life with his 4-year-old son. Contact him at tbesemer@tbcorp.com.