 By Alex Ivchenko, United Electronic Industries

ROLL UP YOUR SLEEVES AND LEARN HOW TO DESIGN DEVICE DRIVERS FOR DATA-ACQUISITION BOARDS.

Get those boards talking under Linux

LINUX IS NOW AN ATTRACTIVE alternative to Windows, especially among engineers who roll up their sleeves and type at the command line. Linux offers a stability you just can't get with Windows 95/98 or even Windows 2000. As a result, the demand for Linux systems and compatible peripherals is mushrooming.

Because you'll find little measurement hardware with support software for Linux, you may decide to write your own Linux drivers. Writing drivers for Linux is no trivial task, and this article presents some tips that could make your work go more smoothly. This article describes driver registration, naming, and access, as well as hardware initialization.

If you're new to Linux programming, you'll find a dearth of resources for programmers and developers, both in print and online. **Reference 1** gives you an overview of how Linux device drivers work. You'll also find few development and debugging tools for Linux. Nonetheless, you can implement some basic debugging tricks (see **sidebar** "Debugging calls for creativity, too").

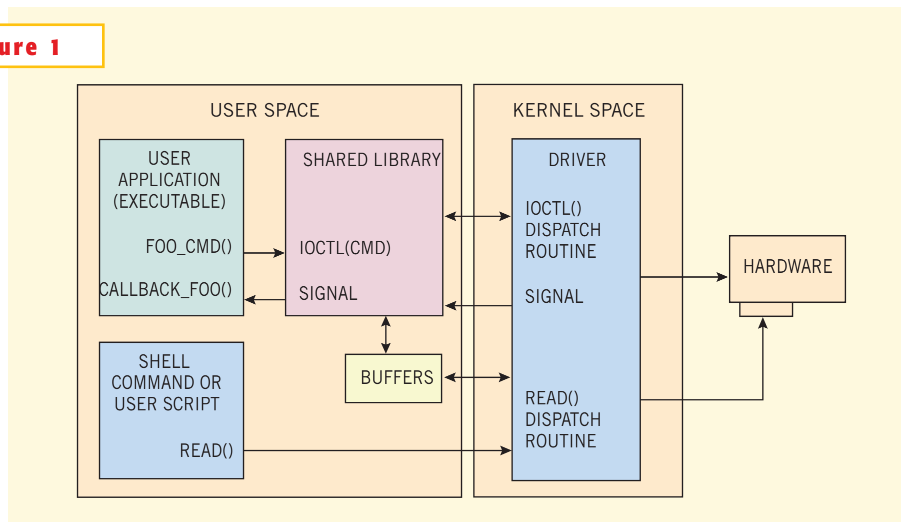
If you've previously worked with Unix, you should feel at home with Linux. But take note of a few key differences. For instance, the Unix and Linux kernels have different function names, implementation details, and levels of sophistication.

GROUND RULES

As you read the **listings** in this article, you'll see function calls, that are preceded with pd__. You need to use different calls, but these give you ideas for how you can construct your own driver. This article assumes that you know how to write C code that writes data to and reads data from an I/O port. The driver architecture described here works as well for multiple cards as it does for a single card, making for easy upgrading and expandability. Any card you use must have a FIFO buffer to store readings, a feature standard on virtually all modern PCI data-acquisition cards.

Regardless of the target hardware, you should write your drivers in two parts. The first consists of low-level code that allows the driver to communicate with the target card at the register level. The second consists of the interface between the driver and the OS. Low-level register code differs for every I/O board. Instead, This article examines the portion of the driver that communicates with the operating system.

Device drivers typically reside in the kernel space of the OS (**Figure 1**) because code in the user space,



Drivers for data-acquisition boards reside in kernel space. You can control a driver either through read() and write() commands or through functions calls using the ioctl() command.

which is memory-protected by Linux, has no direct access to hardware. In this way, the OS protects itself from errant applications. Because a driver resides in the kernel space and can work directly with hardware, it can create havoc—even a system crash—elsewhere if you write data to a wrong address. So, be sure to write your code carefully.

LISTING 1—DRIVER-ARCHITECTURE SPECS

```
$max_cards = 2;
$major = 61;
$minor_range = 5;
system("rm -f $base_dir/daq-*");
for ($card = 0; $card < $max_cards; $card ++)
{
   print "daq card $card\n";
   $minor = $card * $minor_range;
   system("mknod $base_dir/daq-c$card-ain c $major $minor");$minor ++;
   system("mknod $base_dir/daq-c$card-aout c $major $minor");$minor ++;
   system("mknod $base_dir/daq-c$card-dio c $major $minor");$minor ++;
   system("mknod $base_dir/daq-c$card-uct c $major $minor");$minor ++;
   system("mknod $base_dir/daq-c$card-drv c $major $minor");
   system("chmod 0666 $base_dir/daq-*");
}
```

CRUCIAL QUESTIONS

Before you can create a data-acquisition device driver for Linux, you must answer several questions:

- Do you want application programmers to have access to all of the data-acquisition board's hardware options?
- What minimum level of system performance will the driver require?
- Do you want shell access (from the Linux prompt) to the driver or access through function calls only?
- Should you keep the driver simple or add the complexity that makes it portable across operating systems?

When you decide whether all of a data-acquisition card's I/O subsystems—analog input, analog output, digital input, digital output, and counter/timers—should support concurrent access by different processes, carefully consider the costs and benefits. Access from multiple processes is difficult to implement and is often unnecessary.

When you design a Linux driver, you have two choices of driver: block and character. Block drivers can process data in an arbitrary order, but they function similarly to a disk drive. These drivers are best suited for devices that can have a file system.

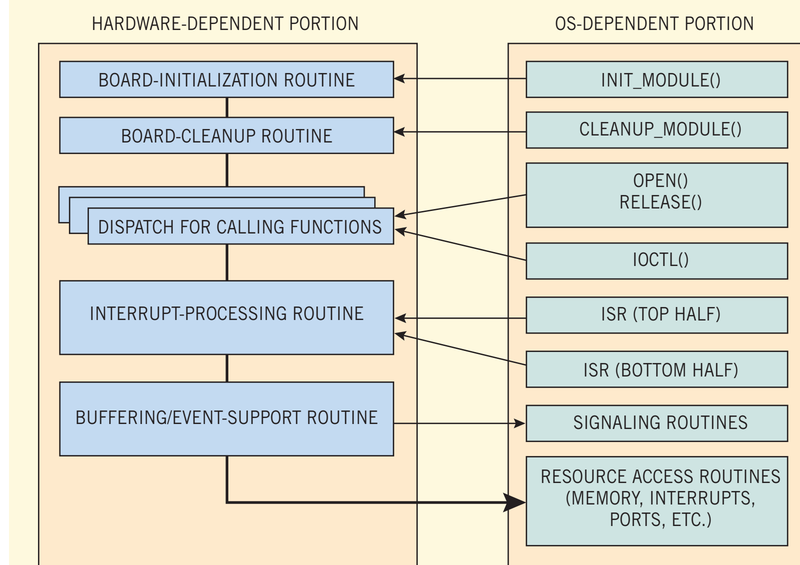
Data-acquisition devices always need a character driver, whose read/write operations have access to data only in sequential order. Don't let the "character" name mislead you: A character driver also can work with blocks of data if one of the arguments you pass to is a pointer to a block of data.

Now, consider how to set up the user and application interfaces for your driver (**Figure 2**). Linux gives you access to device drivers as if they were files. Linux users are accustomed to controlling a driver through shell commands and scripts. Therefore, your driver should include a minimal set of functions accessible using read() and write() operations at the Linux shell command.

Although read() and write() shell commands also let application programmers call the driver directly from their programs, you can provide access to the driver in another way: by creating a library that encapsulates your application-programming-interface (API) calls. Using libraries shields application programmers from making calls directly to the low-level code that controls a card's registers.

When the driver needs to inform the test application about a hardware event, such as a buffer half-full condition, it sends a signal to the library, which in turn calls a call-back function into the appli-

Figure 2



Design a driver in two parts. Keep the hardware- and OS-dependent portions separate, so you can use the driver with another board or another OS without writing an entirely new driver.

cation. A library can also allocate buffers in system memory and connect them to the driver, so this approach hides system-dependent details from the applications and makes drivers portable across operating systems.

Linux device drivers exist as files in a directory called /dev. When you create a driver file, you register the device's name with the OS. As part of the registration process, Linux identifies drivers by integers. Each driver has one major number and can have several minor numbers. It's easy to view installed drivers along with their major numbers by going to the OS shell, moving to the /dev directory, and executing the ls –l command. This command lists all special files, a class into which device drivers fall.

DRIVER-NAMING CONVENTIONS

One driver can serve all devices of similar functionality coming from one manufacturer. So, you typically assign a major number to each group of related drivers. Each minor number refers to a specific subsystem—such as analog input—on a specific board. For example, you could employ the script in **Listing 1** to tell the OS which major and minor numbers to associate with a particular device driver. The person installing a card runs this script, which the hardware manufacturer supplies as part of a device-driver kit, during board installation.

The first line in **Listing 1** indicates how many cards this driver supports: two in this case. A driver architecture that supports any number of boards that contain any or all of the data-acquisition board's subsystems makes it easy to add support for new generations of devices without rewriting major portions of the driver.

The second line in the script assigns to the group of cards a major number: 61. You can arbitrarily select a major number, but be aware that the OS reserves a few of them for specific purposes. Refer to a Linux programming book for details.

The script also tells the OS to define five minor numbers for each card, a number that equals the number of subsystems on the card. The next line is optional and makes a call to the OS to remove all device files with similar names to those the script is about to use. Now, for each of the two data-acquisition cards, the script uses the mknod (make node) command to create five device files with these names: daq-cN-ain, daq-cN-aout, daq-cN-dio, daq-cN-uct, and daq-cN-drv.

Finally, the last line uses a series of octal numbers (here all sixes) along with the chmod command to set the access rights to all device drivers that fall into the designated wild-card name; this line dictates that everybody has read/write permissions for the driver.

Try to use descriptive filenames for device drivers. For instance, the file /dev/daq-c0-ain gives the user access to the analog-input subsystem of card 0. This card is assigned minor numbers 0 to 4, whereas the analog-input subsystem of card 1 uses minor numbers 5 to 9. Drivers with different major numbers, however, can use the same set of minor numbers.

To avoid confusion with this numbering scheme, you can determine which board and subsystem an application program will open with a given driver. Insert the following lines of code in the driver:

```
board = minor /minor_range;
subsystem = minor %
minor_range;
```

The first line finds the integer value. The second uses the remainder to point to the minor number.

The OS cares about major numbers

DEBUGGING CALLS FOR CREATIVITY, TOO

When writing and debugging Linux code, you should anticipate one major hurdle: the lack of any source-code debuggers. You'll spend more time debugging than if you were doing the same job under Windows.

Most Linux developers are familiar with gdb, but that tool is suited for debugging code in the user space and has limited utility for writing drivers, which reside in the kernel space. Similarly, the most recent version of the freeware kdb debugger (which you can download from reality.sgi.com/slurn_engr/) is useful, but it's not a source-level tool. Because of Linux's growing popularity, the developer community can only hope that tools such as NuMega's SoftICE will appear soon.

Meanwhile, several methods exist for using the above-mentioned tools along with other Linux utilities and features that allow you to track what's happened in an errant driver. (Assume that the system didn't crash because the driver sent the board into a sequence that locked up the PCI bus.)

First, you can create log files and examine them after a crash. For instance, you can use printk(KERN_DEBUG "Message \n") to write messages into the /var/log/kern.log file or whichever file you set up for logging kernel messages.

Second, remember that /proc is a virtual file system that provides information about a running process. With it, you can display information from the driver on the fly.

Third, try issuing ioctl() calls from another process and copying the memory-holding driver variables into the user space of that second process.

Fourth, use kdb/kdebug alongside gdb to examine driver code. That step is not mandatory, but you should run the debugging tool from another PC over the serial interface so a crash doesn't bring down your debugger.

As you might surmise, none of these methods give a proper dynamic picture of what's going inside the kernel. For this job, you can try two other methods.

In the first scenario, attach an old Hercules ISA video card to a secondary monochrome monitor. This brand of video card has its own memory accessible from its driver, which can display messages on the fly. You must write only a small debugging windowing procedure to get an instant view of the driver internals.

In the other approach, you could use an ISA-bus digital I/O board. (You could also use the system's primary data-acquisition card if it has digital I/O.) To supply details about processes taking place inside the driver, you can set/clear bits or write some sequence to the digital output port. You can capture and examine these digital outputs with a logic analyzer or a digital I/O board installed in another PC.

only; the driver keeps track of opening and closing minor numbers. You can minimize the driver's complexity if you deny subsystem sharing. Write your driver so that it opens a minor number and stores the process ID (PID) of the process that opened it. If another process tries to gain access to the same minor number (board subsystem), the driver denies it access.

If, instead, the driver code lets several processes have access to the same board subsystem, the driver would have to stop any ongoing operation, store status values, reconfigure the board for another process, run the new operation, and then reset the board to its previous status. This sequence might not present difficulties if two applications with low throughput, such as a voltmeter and a thermocouple monitor, need to share a subsystem. But if one application involves high throughput rates, such as a digital scope, sharing degrades an application program's performance.

As part of your driver design, you should decide whether you want to grant access to your driver from the Linux command line. Providing such access lets application programmers and system integrators confirm that their hardware is working before compiling and running any code. If you're familiar with Linux shell programming, you can use a driver directly from the command line using the Linux shell commands.

You might first want to open a driver by issuing a read() command using a typical name, such as /dev/daq-c0-drv. You receive information about Board 0, such as its serial number and the device's current status. I suggest that you provide limited access to the driver through the Linux shell. Provide just enough access to let board installers test the board before they write application programs.

You can use read() and write() commands for simple devices. When dealing with a complex device that incorporates many functions, however, implementing reads and writes with a command language can become confusing for users, and the driver must take steps to properly parse the command line. So, although these two commands are useful for accessing driver functions from the Linux shell, you should use the ioctl() command when you access the driver using an application program.

LISTING 2—DISPATCH ROUTINE

```
//============ ioctl dispatch routine ===================
static int pd_ioctl(
   struct inode *inode,
   struct file *file,
   unsigned int command,
   unsigned long argument
) {
    int real_minor, board, board_minor, iRes;
    // calculate board# and subsystem
    real_minor = MINOR(file->f_dentry->d_inode->i_rdev);
    board = real_minor / PD_MINOR_RANGE;
    board_minor = real_minor % PD_MINOR_RANGE;
    // check if this process owns the subsystem
    if (!pd_CheckOwnerPID(board, board_minor, current->pid))
       return -EBUSY;
    // dispatch the ioctl() call to board-dependent routine
    iRes = pd_dispatch_ioctl(board,      // adapter number
                            board_minor,  // adapter subsystem
                            command,      // command
                          argument,     // input/output buffer ptr
                            argument      // output buffer ptr
                           );
    // convert board-dependent error code into OS-dependent code
    return  pd_ConvertErrorCode(iRes);
}
```

The ioctl() command presents a different entry point into the same driver code. Programmers working with Windows 95/98/NT are accustomed to using an ioctl() interface to drivers. Further, instead of requiring application programmers to include every I/O and driver parameter in the calling function, programmers can now use a pointer to a buffer that contains that information.

The structure of an ioctl() call is:

ioctl(unsigned int fd,
unsigned int request,
unsigned long argument)

The variable fd refers to a file descriptor that the application receives when it opens the driver. Later, user code opens the device-driver file and then issues ioctl() calls to it. Through the request argument, you can specify the desired action, and the argument is a 32-bit variable that can be a pointer to a buffer containing I/O parameters.

Any call to the driver causes the OS to call a dispatch routine. The dispatch routine selects and executes a driver function based on arguments in the calling function. In **Listing 2**, the dispatch routine decodes the command number from an ioctl() call.

ANALYZE DRIVER ARCHITECTURES

By knowing how the OS registers and calls of a device driver, you can better understand which architectures work best for a driver. You must find the best balance between ease of use and complexity. Using formal interfaces makes the driver easier to write and use, but it adds processor overhead when the driver calls a function.

If you want your driver to be flexible enough to work with new hardware or to be easily ported to another OS, split the driver into OS- and hardware-dependent parts (**Figure 2**). By doing this split, you need to replace only the affected driver portions when adding support for a new card or OS.

After you define the framework and methodology, you can begin to write the working code. Once you write the driver's register-level hardware-interface code (a job that goes beyond the scope of this article), you must tell the kernel which operations the driver supports and where to find its entry points. Write the

LISTING 3—DATA-ACQUISITION-BOARD INITIALIZATION

```
int init_module(void)
{
  struct pci_dev *dev = NULL; // PCI device structure
  // find suitable device on PCI bus and try to work with it
  while (dev = pci_find_device(MOTOROLA_VENDORID, DSP56301_DEVICEID, dev))
  {
     pd_enumerate_devices(dev); // OS-independent function to initialize device
  }
  // register character device in OS
  if(register_chrdev(PD_MAJOR, "powerdaq", &pd_fops)) return -EBUSY;}
```

driver so that it knows where to find various kernel functions. Two Linux commands serve this purpose: insmod loads the driver code into the kernel, and rmmod unloads it.

To execute either command, you must log into the OS as a "super user." While the driver module loads, insmod resolves symbolic names within the driver into entry points, and it modifies addresses inside the driver module so that the driver can gain access to kernel functions and variables. Your driver uses kernel functions and variables that the OS defines in files /usr/include/asm and /usr/include/linux.

Once you link the driver into the kernel, you must initialize the hardware. For this step, Linux automatically calls the driver function init_module(), which lets the driver find devices to serve. The driver registers the major number, character name, and table of supported operations.

The code in **Listing 3** shows how the data-acquisition hardware is initialized. The code sequences through all physical devices on the PCI bus and, using the function pci_find_device(), enumerates each one only if it meets the defined conditions through the vendor and device IDs passed as parameters. For example, the code looks for boards that contain a Motorola DSP56301 DSP, the condition this driver defines. If you want to see the result of this function call, you can use any text editor. You can view a listing of all devices attached to a system's PCI bus by reading the file /proc/pci.

Beyond the operators in the init_module(), Linux introduces other useful PCI-related calls. The header file /include/linux/pci.h contains the declarations of calls to PCI boards. One example, pci_read_config_ XXXX (spci_dev, function, &result), reads the OS's PCI configuration space and returns the value requested as one of the function call's parameters. For example, pci_read_config_ word (dev, PCI_SUBSYSTEM_ID, &subsystem_id) returns the subsystem ID. You need Linux kernel version 2.2 or later to use the PCI functions.

INITIALIZE THE HARDWARE

In **Listing 3**, the function pd_enumerate _devices() initializes the data-acquisition board. The procedure varies for each board, but, in general, you should follow these steps when writing the initialization function:

First, make sure the driver supports the device found. Each manufacturer of PCI cards has a unique ID, as does each family of PCI cards. The driver code reads from the PCI configuration space information about an installed card such as pci_subsystem_ vendor_id and pci_subsystem _id. The driver can then check whether these values match those of the cards it's designed to handle.

When working with legacy ISA boards, this driver-verification process becomes more difficult. To circumvent this difficulty, you have two options. If the ISA card was designed to comply with plug-and-play specs, it should respond to certain port numbers in a specified manner. If not, then you need to know enough about the hardware to be able to develop a sequence that verifies the presence of that board.

Next, you must allocate room for a structure that contains all the device information you need to work with: initialization settings, status, and runtime parameters within the driver's memory space. You should create this structure using an array of pointers (where the_board is a structure that contains board and subsystem-level structures):

```
the_board* pthe_board
[MAX_BOARDS]; // pthe_board
// is a pointer to array
// of pointers to the_board
```

Use the Linux function kmalloc to allocate memory space for structures. Later, in cleanup_module(), you can release the memory space with kfree. Note, however, that kmalloc doesn't fill the allocat-

LISTING 4—DISPATCH ROUTINE

```
switch (e_board_type) // register handlers depending on board type
{   case PD2_MF: // PowerDAQ II multifunction board
        pReadProc[board] = pd_MfReadProc;
  pWriteProc[board] = pd_MfWriteProc;
  pIoctlProc[board] = pd_MfIoctlProc;
    break;
    case PD2_AO: //... PowerDAQ II Analog Output board
    // other board types
    // ...
    default: // default handlers
        pReadProc[board] = DefaultReadProc;
        pWriteProc[board] = DefaultWriteProc;
        pIoctlProc[board] = DefaultIoctlProc;
}
```

ed memory with zeros, so you should make sure the driver does so during the initialization phase. In addition, kmalloc allocates memory by pages (4 kbytes/page on Intel platforms), so you can efficiently allocate memory by creating memory blocks for device structures. Finally, remember that initialization isn't time-critical. Thus, you can allocate memory with the kernel priority level set to GFP_KERNEL, which means that the kernel can wait for sufficient memory to become available as other processes free it.

LISTING 5–IOCTL () ROUTINE

```
static int pd_ioctl( . . . )
{
     int real_minor, board, board_minor;
     // Find out which board/subsystem ioctl is going to access
     real_minor = MINOR(file->f_dentry->d_inode->i_rdev);
     board = real_minor / PD_MINOR_RANGE;
     board_minor = real_minor % PD_MINOR_RANGE;
     // check board subsystem ownership, does it belong to calling process?
     if (!pd_CheckOwnerPID(board, board_minor, current->pid)) return -EBUSY;
     // call registered dispatch routine
     iRes = pIoctlProc[board](board, board_minor, ... );
...
}
```

Some data-acquisition boards require the host computer to download and start executing onboard firmware. If that's how your boards work, perform this task now so that your driver knows exactly what type of device you have and what firmware to load.

If your board contains nonvolatile memory, read any descriptive data from it, such as factory serial number, calibration date, or calibration coefficients. Save this information for future use inside the_board* structure.

Register all required read(), write(), and ioctl() routines with the kernel using the register_chrdev() function as shown in **Listing 3**. One of the parameters in that function, &pd_fops, is a pointer to a file-operation structure that supplies the kernel with the entry points to the functions that serve read(), write(), ioctl(), and other requests to the driver.

You also should write a separate dispatch routine for each type of board your driver supports (**Listing 4**). This approach eliminates the need for the driver to perform an extra checking step, simplifying driver development. Instead of filling the driver with complicated case statements, you write, register, and later call just one routine for each card and its subsystem.

The code in **Listing 4** starts with a switch statement based on the board type (e_board _type), which the driver reads from the PCI configuration space or from each board's nonvolatile memory. For each board type, a case statement stores the address of board-specific read(), write(), and ioctl() routines into a pointer. Thus, in later calls, the driver can employ specific dispatch routines and know that it's calling the proper procedure.

Most modern data-acquisition cards use PCI bus-mastering or DMA, so you must allocate memory pages for these operations. You should use the kernel functions get_free_ page() and _get_ dma_pages() for your memory allocations.

Although critical in a real-time data-acquisition-board driver, PCI-bus mastering and DMA place extra demands on memory, so be realistic about your requests for memory space. The kernel tries to satisfy your allocation request, especially if you set the priority level to GPF_KERNEL by swapping out as many pages as possible. This swapping can dramatically degrade system performance.

Allocate 1 to 2 Mbytes of RAM for any high-speed data-acquisition board. In a system loaded with several boards and 64 Mbytes of memory, you can claim 4 to 16 Mbytes as a DMA buffer.

Also, recognize that a data-acquisition system typically doesn't need all this memory available all the time. Thus, you can try to implement a mechanism to lock pages before the driver needs them and release the pages otherwise.

Next, you should register an interrupt-service routine (ISR) for the board. Assuming that you follow that philosophy, the driver sets the address for the service routine's top half.

You might want to write separate ISRs for different types of boards. This takes extra development time, but it makes execution more efficient because time-critical ISR routines don't have to first check for the board type.

As a final step, run a hardware-initialization routine on each board to restore any settings to the desired start-up state and, if necessary, to load calibration values. You might declare and increment the board_installed counter within the driver so you know how many boards are installed.

As noted earlier, the initialization routine registers entry points to the driver with the kernel. When Linux receives a request from an application, the OS calls the corresponding function. Consider **Listing 5**, which shows the driver's ioctl() routine. The routine first identifies with which board the application program wants to work. Then, it calls the dispatch routine for this board type.□

Part 2 of this article will look at the details of writing Linux driver code.

REFERENCES

1. Marsh, David, "Understand Linux Device Drivers," *Test & Measurement World*, April 15, 2000, pg 6, www.tmworld.com/articles/2000/0415_Linux.htm.

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 data-acquisition boards. He has most recently spent his time writing Linux drivers for these cards. You can reach him at aivchenko@ueidaq.com.