A Simple Approach to CHARACTER DRIVERS in USER SPACE

The BaseBoard4 from Demand Peripherals can contain different combinations of 25 different character devices, all multiplexed on to a single USB-serial link. Its drivers, described here, show how writing drivers in user space can get a complex device up and running quickly.

BOB SMITH

Demand Peripherals, Inc., makes an FPGA-based robot controller that gives a robot or other industrial control systems the high I/O pin count and precise timing that a Linux laptop or single-board computer alone cannot offer. The company has built more than 25 different FPGA-defined peripherals for the controller, and it wanted to offer Linux device drivers for all of them.

Doing 25 drivers in the kernel, although possible, would have required time and effort far beyond what the company could afford. The process of building kernel device drivers would have been even more complicated because the FPGA card connects to the Linux host over a USB-serial link. The solution, illustrated in Figure 1, is to have a dæmon manage the USB-serial port and demultiplex the various FPGA-based peripherals out to their own device nodes. The device nodes are little more than shims that let the high-level application deal with separate device entries for each peripheral.

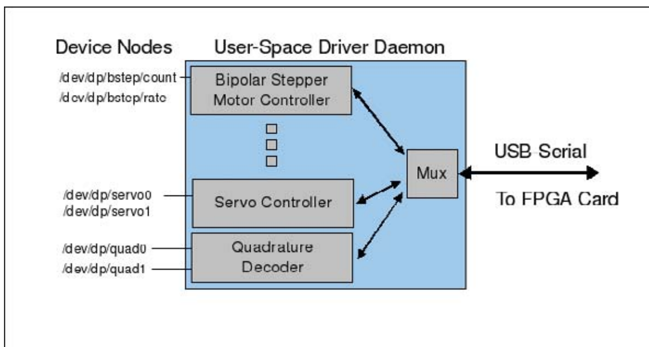

Figure 1. Example of a User-Space Device Driver

The customer selects the mix of peripherals to be loaded into the FPGA. Figure 2 shows a BaseBoard4 with some cards that demonstrate what might be a fairly common peripheral mix. The system pictured has eight peripherals, including a four-channel servo controller, a dual H-bridge controller, a quad interface for the Parallax Ping))) range sensor, a RAM-based pattern generator (driving the data and clock lines going to a 48-bit shift register that connects directly to the LCD), a unipolar stepper motor controller, a bipolar stepper motor controller, a quad event or frequency counter (connected to a single Parallax light-to-frequency sensor), and a dual quadrature decoder. Schematics for all of these demo cards

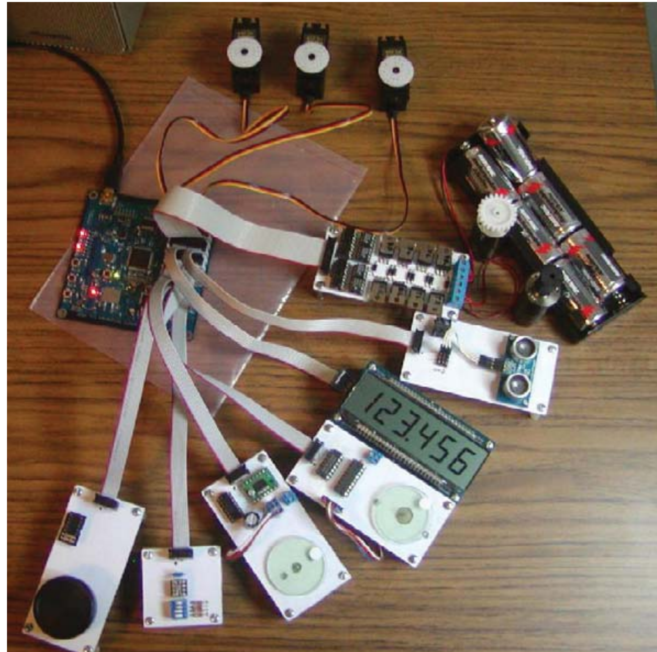

Figure 2. Robot Peripherals, All with Linux Drivers

are on the Demand Peripherals Web site.

All of the peripherals shown in Figure 2 can be configured and controlled using device nodes in the /dev directory. The following Bash commands, for example, might be part of the higher-level control software for the system pictured:

```
# Feed wheel quadrature counts to a motor control program
cat /dev/dp/quad0 | my_motor_pgm &
# Feed the same quadrature counts to a navigation program
cat /dev/dp/quad0 | my_navi_pgm &
# Set a stepper motor step rate to 1000
echo "1000" > /dev/dp/bstep1/rate
# Now step 300 steps
echo "300" > /dev/dp/bstep1/count
# Monitor distance reported by a Parallax Ping)))
cat /dev/dp/ping0/dist &
# Set a servo pulse width to 1.5 ms (1500000 ns)
echo "1500000" > /dev/servo/servo4
```

USE CASES

The above commands illustrate two of three important use cases for the user-space drivers: sensor broadcast and driver configuration. The third use case is bidirectional transfer.

The first use case is sensor broadcast, and in the example above, it's actually multicast of sensor data. Did you know that the /dev/input drivers implement a multicast mechanism? Multiple readers get identical copies of the events that come from the input devices. There is a simple experiment you can do to demonstrate this. Press Ctrl-Alt-F2 (to go to a different console), log in, and run the command `sudo cat /dev/input/mice | od -b`. Do the same for another console (for example, Ctrl-Alt-F3). Now, move the mouse a little and switch between the F2 and F3 consoles. They both display the same thing, don't they? What a shame that Linux does not

FOR ROBOTICS, THE ABILITY TO FAN A SENSOR READING OUT TO SEVERAL PROCESSES IS PARTICULARLY IMPORTANT.

have some generic way to do multicast like that of the /dev/input subsystem.

For robotics, the ability to fan a sensor reading out to several processes is particularly important. For example, a quadrature encoder attached to a wheel needs to be seen by both the motor controller software and by the navigation software. The motor controller might need to know if the wheel is turning to know whether the motor is stalled, and the navigation software might count the wheel revolutions to compute the robot's current location.

The second use case is peripheral or driver configuration. DC motor controllers need to know the frequency of the PWM pulses. Stepper motors need to know the step rate, and the SPI (Serial Peripheral Interface) ports need to be told the clock frequency and the mode of operation. Either an ioctl() call or a sysfs-style interface can be used for driver configuration.

Configuration interfaces can be a little tricky, in that the information is often not a simple stream of bytes—it may encompass several different pieces of information. An ioctl() interface typically passes a data structure for complex configurations, while a sysfs interface might use a space-separated list of ASCII-encoded values. Demand Peripherals uses the ASCII-encoded numbers approach, because the overhead of decoding and parsing a line of text is not too onerous given the relative infrequency of driver configuration. Also, being able to `cat` a sysfs type file to see the driver configuration is kind of handy.

The third use case, bidirectional transfer, is really the most common use case. You probably are already familiar with serial ports, the most common example of bidirectional I/O. Although none are included in the examples above, the FPGA-based robot controller needs bidirectional I/O for peripherals that transparently pass data from one end to the other. These include both FPGA-defined serial ports and SPI ports. You may prefer, as we did, to be able to do block reads and writes until both sides of the interface are open.

REQUIREMENTS FOR USER-SPACE DRIVERS

Our number one requirement for this project was to spend as little programmer time as possible on it. This meant minimizing the number of lines of code to be written and avoiding modifying someone else's poorly or completely undocumented code. This requirement also implied that we not try to hide our interfaces in an application library. Because a library is part of the higher-level control application, you still would need a dæmon, still need some common IPC mechanism, and still need to document the internal and the external interfaces. The other problem with a library approach is that it is usually not just one library; you may need to write a library, or binding, for every programming language you want to support. Using a real character device instead of a library means your customers can program in any language they want, not just the ones for which you've written a binding.

The second requirement was that the driver security model be based on file permissions. This implied that all of the device data and configuration interfaces should be visible in the filesystem. That is, you should to be able to do a `chmod 644` on something like /dev/dp/bstep1/rate. Using named pipes and FUSE (Filesystem in USErspace) could have fulfilled this requirement. Doing this using pseudo-terminals would have been tricky.

Another requirement is that select() works both in the higher-level control application and in the user-space driver itself. This requirement comes about because select() is so much faster than threads in most applications. Embedded systems, such as robotic or other industrial control systems, often run on the cheapest, lowest cost hardware possible, and, in the case of robots, often on battery power. These constraints lead embedded Linux programmers to prefer select()-based systems.

FUSE often is suggested as a way to implement character drivers, but I was unable to get select() to work on both sides of a FUSE interface. I like FUSE; it can solve a lot of user-space driver problems, but it seems unfair to me to ask FUSE, a filesystem, to double as a character driver. After all, who would expect ext3 or other kernel filesystems to have built-in character drivers?

The last requirement was that writers block until a reader is present. Both named pipes and pseudo-ttys allow the writer to write 4KB before blocking. It was important to us that the driver not fill a buffer with stale data that a higher-level robotic application must discard to get to the current data.

A SIMPLE APPROACH TO USER-SPACE DRIVERS

In the end, we didn't find any existing Linux facilities that satisfied all of our requirements and use cases. However, we were able to find or create two relatively simple device drivers that could. Figure 3 illustrates the basic idea.

The idea is to have two very thin drivers that sit between the higher-level applications and the user-space driver. These are real drivers and appear as such to the higher-level software. The data exchanged between the application and the user-space driver passes as transparently as possible through the kernel. Even flow control passes transparently between the application and the user-space driver.

The first use case, that of multicasting sensor data, is solved by the "fanout" driver described in detail at **www.linuxtoys.org**. Demand Peripherals uses fanout devices for quadrature decoders,

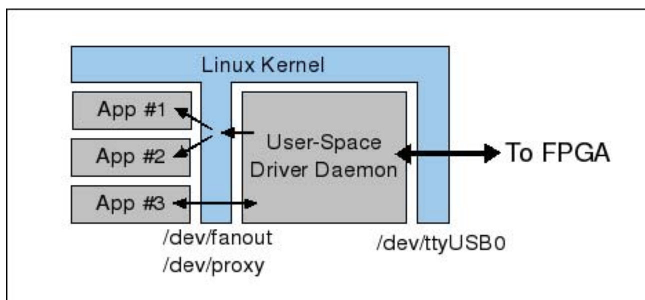

Figure 3. Two New Drivers Link Applications to Driver Dæmons

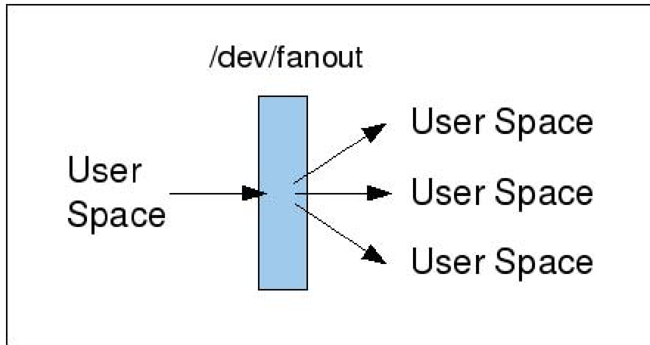

Figure 4. A Simple Multicast Device

IR receivers, ultrasonic range sensors, PlayStation controller interfaces, event counters and all other continuously sampled sensors. Figure 4 shows the basic data flow in a fanout device.

You can skip further down in this article to get and install fanout, or you can continue reading and come back to try the examples. Once you've installed fanout and created the device nodes for it, you can test it with a few simple commands:

```
cat /dev/fanout &
cat /dev/fanout &
cat /dev/fanout &
echo "Hello World" > /dev/fanout
```

The message appears three times, as you'd expect. Fanout is like /dev/input in that it protects the writer, not the reader. If a reader does not keep up, the reader gets the error, allowing the writer and other readers to continue unimpeded.

For data flowing in the opposite direction, you need something like a "fan-in" device—that is, something that protects the reader. A named pipe works reasonably well for this.

The low-speed nature of driver configuration, the second use case, makes possible several approaches. The approach we took was to write a driver, called proxy, that solved both the configuration use case as well as the bidirectional transfer use case. The two defining features of proxy are that one side cannot write until the other side is open for reading, and that a write of zero bytes is passed through the driver and seen as a read of zero bytes at the other end. The usefulness of the second feature is best shown by an example. Consider the case of a user reading the current value of a configuration parameter:

```
cat /dev/dp/bstep/rate
```

/dev/dp/bstep/rate is a proxy device, and the user-space driver dæmon on the other side of it would see that a write is possible when `cat` opens the device. The dæmon writes the

visit us at www.siliconmechanics.com
or call us toll free at 866-352-1173

Powerful.
Intelligent.

Dominic and Maddison are logistics and shipping Experts for Silicon Mechanics. That's especially important recently. They have geared up to deliver the newest Silicon Mechanics rackmount servers and storage products with next-generation Intel CPU technology: the Intel® Xeon® Processor 5600 Series.

They are both excited to be shipping products that take advantage of the increased performance and decreased energy consumption made possible by features like 6-core CPUs with up to 12 threads and 12 MB of cache, and Intel® Turbo Boost Technology. No, we can't put the Experts themselves into the boxes we deliver, but you can be sure that every one of our products contains our Experts' commitment to quality and service, and that includes packaging and shipping.

When you partner with Silicon Mechanics, you get more than increased performance and improved energy efficiency — you get Experts like Maddison and Dominic.

For more information about products featuring the Intel Xeon Processor 5600 Series, visit www.siliconmechanics.com/5600

Expert included.

Silicon Mechanics and the Silicon Mechanics logo are registered trademarks of Silicon Mechanics, Inc. Intel, the Intel logo, Xeon, and Xeon Inside, are trademarks or registered trademarks of Intel Corporation in the US and other countries.