UCSD Embedded Linux online
Week 9

Slide 1

Our primary goal this week is to create a Linux device driver that replaces the user space "pseudo drivers" if you will that we invented earlier for our thermostat. First we'll review and elaborate on the APIs that connect a driver to real hardware.

Slide 2

There are several issues related to accessing peripherals that we kind of skipped over in our earlier thermostat examples. The first issue is that different processors treat I/O differently. Some, like the x86, put I/O registers in a totally different address space accessed by unique instructions. Others, such as the ARM9 that we've been using, simply reserve a part of the memory space for I/O ports. In this case, as we've seen, I/O access is no different from memory access. In the interest of portability, Linux defines a set of architecture-dependent I/O port macros that map to the appropriate address space and instruction set.

Unlike memory, I/O access usually has "side effects." That is to say that the act of reading or writing an I/O port often does something other than simply transfer data. For example, reading a status register may cause an interrupt flag to be reset. It may be important that certain registers be read or written in a particular order.

Memory access has no side effects. No matter how many times you read a memory location, you get the same answer, unless of course it's shared memory and some other process writes to it. The compiler takes advantage of this lack of side effects to implement optimizations that may change the order in which events occur, or even eliminate them altogether. We must be very careful about compiler optimizations when it comes to I/O port access.

Finally, it should be noted that I/O ports are allocated exclusively to one driver at a time. Before your driver can use a range of I/O ports, it must ask the kernel's permission.

Slide 3

Here is the Linux API for accessing I/O ports in an architecture-independent manner. I/O ports are often "length-specific" and so we need functions to access them according to size. There is a set of functions (actually macros) for accessing a single datum from an 8-, 16-, or 32-bit port and a corresponding set of "string" functions (macros) for accessing multiple items of the same sizes.

Note that where the term unsigned is used without further type specification, it refers to an architecture-dependent definition whose exact nature is not relevant. The functions are almost always portable because the compiler automatically casts the values during assignment and making them unsigned helps prevent compile-time warnings. No information is lost with these casts as long as you assign sensible values that avoid overflow.

Slide 4

Here is the API for accessing I/O memory space. Remember that the kernel deals in virtual addresses. So, before we can use I/O memory, we have to map it into virtual space. That's what ioremap() does. And of course, when you're done using the space, you should unmap it.

For some reason, the kernel developers chose to use "8", "16" and "32" to represent data size rather than "b", "w" or "l". The _rep forms are the equivalent of the "string" functions for I/O ports. In this case, count is the number of items of the specified size. So read32_rep() returns count 32-bit items.

You're probably asking, Why do we need the I/O read and write functions? Why not just use the pointer returned by I/O remap directly just like we did in user space? In most cases you can. But there are apparently some architectures that require some additional processing. That's what's encapsulated in the I/O read and write functions. So

even if your architecture allows you to use the memory pointer directly, if you're concerned about portability, you should use these functions.

Slide 5

Just like any other resource in the system, before your driver is allowed to access an I/O port, it must ask the kernel's permission. There are two sets of functions for doing this depending on whether the I/O region is mapped to ports or to memory space. request_region() and request_mem_region() respectively are the functions that do that. In both cases you give the beginning port number, the number of ports, and the name of your device. The latter is used by /proc/ioports to identify the driver that owns the I/O region. The functions return a non-null value if the request is granted.

Of course, when your driver exits, it should release the I/O ports back to the kernel with release_region() or release_mem_region().

check_region() and check_mem_region() simply test whether the requested region is available. They return a negative error if the region is not available. These functions are "deprecated" because the process of checking, and then subsequently requesting, a region is not atomic. So the return value from check region functions may or may not be useful. They're still in the API because a number of older drivers use it.

Slide 6

Our example for experimenting with hardware access is called simple_hw and it's found in the simple_hw/ directory under target_fs/home/src. Make an Eclipse project if you choose, or just open simple_hw.c with an editor. Although this is intended to be a very generic driver, the default values for the operational parameters are intended to expose the LED bits on the target board. Actually, for the Mini2451, it only exposes the two LED bits in GPIO port B.

The driver creates 8 devices, each of which represents one 4-byte register. The default value for pbase maps these devices to the eight registers starting with General Purpose port B, GPB. pbase stands for physical base, by the way. We can specify that transfers are 1, 2, or 4 bytes at a time and we can specify an offset to say, for example, which byte of a 4-byte register the value will be written to.

The module init function does some simple sanity checking on the module parameters to make sure that offset is sensible for the selected size. It then requests and remaps the memory region and registers the device. I couldn't think of anything for open and release to do and they could probably be eliminated.

I think the read and write methods are fairly straightforward. The count is adjusted to be an integral multiple of the transfer size. A buffer is allocated for the adjusted count and we go into a loop to transfer data by the specified size. The write method is optimized for the LEDs connected to GPB. The data passed in is inverted so a 1 means the LED is on and it's shifted left 5 bits so that the least significant bits represent the 4 LEDs. Write also includes a commented out example of direct access using the virtual pointer. This works on the ARM 9 and probably for all ARM variants.

A quick note about the iminor macro that shows up in lines 75 and 130. This is a portable way to extract a minor device number from an inode structure. We didn't discuss the inode structure and I'm not sure it's worth it at this point. Just take my word for it for now. There's also an imajor macro to extract the major device number.

Look at the hw_load script and note that the devices are named to match the PIO register descriptions in the S3C2440 data sheet and in the s3c2410-regs.h header file. Also, take a look at the Makefile to see that we're pointing to our ARM kernel.

Slide 7

Follow the steps in this slide to build and load the driver. We'll use the dd command to get the current state of the Port B control and data registers. We're telling dd to use a block size of four bytes and read one record from /dev/pio_GPBCON and then a record from /dev/pio_GPBDAT.

The return value from the data register is, according to the ASCII chart, printable, but not by a standard Linux console driver. I happen to know it's hex 61a because I ran devmem on it.

Slide 8

So what do these values represent? Two of the four LEDs are connected to bits 5 and 6 of GPB as shown in the GPBDAT diagram here. Writing a 0 to a bit turns the LED on and writing a 1 turns it off. GPBCON, the control register, reserves two bits for each bit in GPBDAT. This allows each data bit to be configured as either digital in, digital out, or a dedicated peripheral function.

Bits 10 through 13 of GPBCON then represent our two LED data bits. The S3C2451 data sheet tells us that 01 in the control register configures the corresponding data bit for output. So the pattern in bits 10 through 13 configures data bits 5 and 6 as output. Those bits are set to 0 in the data register and indeed two of the LEDs are on.

Slide 9

Because of the "hacks" in the write method it's fairly easy to control the LEDs by outputting a single character to GPBDAT. Note that the echo command normally appends a newline character to the end of any string it writes. So the last byte written is always 0a hex. The –n option tells echo not to add the newline.

OK, whoop-de-do you might say. Hey, we did that several weeks ago. Granted, this is a somewhat contrived example, but what it does illustrate is that, now that we have a real device driver, we have the potential to access the board's hardware using ordinary Linux utilities.

Play around with it some more. If you're familiar with shell script programming, try writing a script that will blink the LEDs in turn just like we did with the very first target board program.

Slide 10

You may have noticed an odd looking function call at the end of the switch statements in both the read and write methods. There's rmb() in the read method and wmb() in write. These are "barriers" and are intended to make sure the hardware completes the data transfer operation before proceeding.

GCC generally does a very good job of optimizing. Consider the not very useful code fragment shown in the slide. Without the barrier call, the compiler might very well store a in a register and not move it back to memory until the loop is complete. This is fine if a is a memory location. The value that's finally stored back in memory location a is correct. But if a is an I/O port, maybe we have a good reason for writing b to it ten times. The barrier() function tells the compiler to store back in memory all values that are currently modified and held in CPU registers before continuing.

The remaining functions shown in this slide perform the same operation at the hardware level and are architecture-specific. rmb() guarantees that all reads appearing before the barrier are completed before any subsequent read. wmb() does the same thing for writes and mb() guarantees both.

Slide 11

From the driver's viewpoint, the data coming in to the write method is pure binary. We're forced to express that data in ASCII just because of the nature of Linux utilities.

But if you think about it, it really is a little clumsy trying to manipulate bits using ASCII characters. What would probably be more useful is to be able to write a hex string to the register as shown here. We could create another module parameter called hex that determines whether the driver works on pure binary or interprets incoming data as a hex string.

We could create ioctl commands to get and set this hex parameter. Heck, we could create ioctl commands to get and set any of the operational parameters. Want to give it a try?

Slide 12

What we've been leading up to here is a device driver that implements the hardware access for our thermostat. You probably wouldn't normally incorporate all of this functionality in a single driver. It would probably make sense to do it as two drivers. But for our purposes it's a good way to go.

Your assignment then, is to take the functionality expressed in trgdrive.c in the measure/ directory and translate that to a device driver. The simple_hw/ directory has a template starting point called therm_driver.c. This is actually an optional, extra credit assignment worth up to 15 points.

therm_driver.c registers three minor device numbers; 0 is for reading the ADC, 1 is for setting LEDs and 2 is for clearing LEDs. It turns out we can use the same header file to declare the register offsets that we used with the user space driver.

Slide 13

It's useful to define two modes of data transfer for the analog to digital converter. Initially, you'll want the driver to return temperature as ASCII so you can use cat, for example, to test it. Later, when you modify the thermostat application to use the driver, you'll want it to return binary. So the driver has a mode parameter to select the data type.

For the first pass, don't worry about implementing ioctl. Add therm_driver.o to the first line of the Makefile. Take a look at the therm_load script and note that it creates three device nodes.

As usual, probably the easiest way to test the read function of a driver is with the cat command. Try it out. If it succeeds, you'll notice that data comes out very fast, certainly a lot faster than you can read it. So in this particular case, it's probably useful to introduce a delay into the read method. That's the purpose of the other module parameter delay.

I suggest implementing the delay with the function wait_event_interruptible_timeout() with a condition of zero, that is a condition that's always false. wait_event_interruptible_timeout() expresses time in "jiffies."

Slide 14

Like most systems, Linux keeps track of time with an interrupt that fires off at regular intervals. The HZ macro, defined in asm/param.h, specifies the number of timer tick interrupts per second, normally in the range of 400 to 1000 on PCs running 2.6 or later kernels. At each interrupt the kernel increments a global variable, called jiffies, declared in linux/jiffies.h. In effect then, jiffies tracks the elapsed time since the system was booted.

The 2.6 kernel added a 64-bit tick counter, jiffies_64, of which the lower 32 bits are the original jiffies counter.

The need arises quite often in driver writing to compare the jiffy count against some other value. At 1000 Hz, the 32-bit jiffies counter wraps around in about 50 days. There are lots of Linux systems that run for more than 50 days. So prudence dictates that you use one of these "wrap safe" macros when comparing jiffies to a number.

Go ahead and implement the delay and try that out.

Slide 15

The next step then is to modify the thermostat application to work with our new driver.

You'll need to open file descriptors to all three of the therm devices. There are two alternate approaches you could take here:

- Replace all of the driver API calls in thermostat.c with corresponding read and write calls the driver. In this case we completely eliminate trgdrive.c.

- Modify trgdrive.c by replacing the physical I/O access with the corresponding read and write calls. In this case thermostat.c doesn't change.

I chose the first approach.

You'll want to run the driver in binary data mode with a delay of zero. Alternatively you could take the delay out of the application and rely on the driver's delay. This would also be a good time to implement the ioctl function.

Slide 16

Having gone through the agony of writing a device driver, you can pat yourself on the back. Consider yourself a Linux hacker.

Slide 17

Let's step back and review where we've gone on this 9-week journey through the joys of embedded Linux. We began of course with installing and getting familiar with Linux. Some of you may have already had a Linux installation. We touched on the philosophy of open source and its implications.

Next we set up the target board and installed the class software, which included a cross tool chain that allows us to build programs for the target on the host. We configured the simple terminal emulator minicom to talk to the target board's serial port. The target's Linux kernel and root file system are loaded from NAND flash but we configured host networking so the target could mount part of its file system, the home/ directory, over NFS.

Then we learned about Eclipse, the open source integrated development environment that forms the basis for most commercial vendors' development platforms. That gave us the tools to start building and testing applications on the target board. We saw how Eclipse seamlessly supports debugging on the target. We also looked at the idea of building simple simulation environments that support preliminary testing on the host.

Slide 18

Posix threads gave our application the ability to do two things at once: execute a thermostat algorithm and look for changes in operating parameters. Then we looked at network programming and saw how we can test both the server and client on the same computer before distributing them across the network.

Next we looked at configuring and building the Linux kernel, using the xconfig tool to conveniently manage the large number of configuration options in the kernel. How do we test a new kernel? In the case of the Mini2451, we load it into NAND flash.

BusyBox is a powerful tool for substantially reducing the memory footprint of a target Linux installation. It combines more than 300 common Linux utilities into one executable. It's also highly configurable so you only have to build in what you really need.

Slide 19

Next we played around with the LCD display panel using a pair of existing device drivers. With our thermostat application fully tested and ready to ship, we looked at the Linux initialization process to see how to get the thermostat to automatically start up at boot time. Then we loaded a new root file system into the target's flash and saw that our thermostat application starts directly.

Changing gears once again, we explored the whacky world of kernel programming with a particular emphasis on device drivers. We saw how loadable kernel modules are used to extend the functionality of the kernel without rebuilding it.

With an understanding of basic character driver concepts, this week we created a device driver for the elements of the target hardware used by the thermostat application. And finally, we modified the thermostat to work with that new driver.

Slide 20

All that's left now is to take the final and fill out the course evaluation next week.

Now then, to sum it all up, I hope this exploration of Linux in the embedded space has been useful, helpful, hey perhaps even fun. Linux is fun because it's open. You can see how it works, play around with it, change it to meet your needs. But as I said at the very beginning, the Linux learning curve is very steep.

In this course we've covered a fairly broad range of topics, none of them very deeply. My intent was to give you a good, solid starting point from which to pursue your particular topics of interest in more detail. It's all out there on the Internet and there are shelves full of books about Linux. So, best of luck and have fun with Linux.