# UCSD Embedded Linux online
# Week 7

## Slide 1

We'll start out this week taking a look at Linux device drivers with particular emphasis on the drivers that manage the target board's LCD screen. Then we'll consider the process of going from development code hosted on the workstation to production code on the target device.

## Slide 2

First a little background on the Linux device driver model. While other OSes treat devices as files, "sort of", Linux goes one step further in actually creating a directory for devices. Typically this is /dev. One consequence of this is that the redirection mechanism can be applied to devices just as easily as to files.

Devices come in four "flavors": Character, Block, Pipe, and Network. The principal distinction between character and block is that the latter, such as disks, transfer data in fixed sized block and are randomly accessible, that is, you can move back and forth within a stream of characters. With character devices the stream moves in one direction only. Block devices generally have a file structure associated with them whereas character devices don't. In both cases, I/O data is viewed as a "stream" of bytes.

Pipes are pseudo-devices that establish uni-directional links between processes. One process writes into one end of the pipe and another process reads from the other end.

Network devices are different in that they handle "packets" of data for multiple protocol clients rather than a "stream" of data for a single client. Network devices also have to respond asynchronously to packets arriving from the outside world. This necessitates a different interface between the kernel and the device driver. Network devices are not nodes in the /dev directory.

Most devices are in fact character devices and from here on out that's what we'll be dealing with.

## Slide 3

Take a look at the /dev directory, an excerpt of which is shown here. In this case it's better to use the ls command in a shell window than to view it with a file manager window.

The first character of the attributes field identifies the device entry as either character, 'c', or block, 'b'. The file name is, in this case, the device name. In the place of file size we find two numbers: the Major number and Minor number. The Major number identifies a device driver and, in fact, serves as the index into a table of driver entries.

The driver itself uses the Minor number to distinguish among multiple devices that it's responsible for. As you scroll through the /dev directory, you'll see, first of all, that there are a <u>lot</u> of devices. In many cases, a device name has multiple entries differentiated by a number suffix. For block devices, the number often represents a partition on a mass storage device. For character devices, the numbers usually represent different ways of handling a particular device.

Incidentally, the reason I suggested using the ls command here is that the KDE file manager window doesn't show the major and minor numbers.

Note the use of a link here to attach a "logical" mouse device to a specific port, in this case a PS/2 port.

Device nodes, as these entries are called, are created with the mknod command. And you must be root to use mknod.

## Slide 4

Many of the major device numbers have been pre-assigned to specific devices as indicated by this excerpt. There's an extensive document in the kernel Documentation directory called devices.txt that lists all of the assigned device numbers and describes the minor numbers for many of these.

## Slide 5

User Space programs can access I/O at two different levels. We'll be using the low-level I/O APIs that include the kernel services shown in this slide.

The read() and write() calls are fairly obvious. read transfers data from the device (or file) to memory while write transfers in the opposite direction. The purpose of the open() function is to establish a connection, or "path" between the device or file and the application. The value returned by open() is an integer that serves as a *file descriptor* or *handle* if you will, that's used as an argument to all other I/O calls to identify the connection. When a connection is no longer needed, the close function frees up the resources used by that connection.

ioctl() may be thought of as a general-purpose "escape" mechanism. Every device has some peculiar characteristics that don't fit neatly into the stream-of-bytes model. A printer has an out-of-paper status condition. A serial port or modem has a data rate parameter. ioctl provides an open-ended mechanism for handling all of these device-specific conditions. Each device driver is allowed to define its own set of ioctl commands and any parameters those commands might need.

## Slide 6

Many years ago when I started in this business, the typical "console" device for a Unix system was a dumb terminal connected to the computer via a serial line. This is a classic case of a character device. Then video terminals like DEC's VT100 developed some smarts, providing means to move the cursor around, change colors, and selectively erase parts of the screen. Later, the PC introduced the notion of bit-mapped "framebuffer" devices built into the computer itself.

These developments inevitably introduced additional layers of complexity in the Linux driver model. During the era of serial line video terminals, every manufacturer initially invented its own protocol for things like cursor control. So for the sake of portability, these various, usually incompatible, protocols had to be "abstracted out" into something uniform.

The role of a "console device driver" then is to provide a uniform interface to applications that need to do more than just output a stream of characters. The VT100 protocol has emerged as the de facto standard for terminal control and this is what a typical console driver exposes as its API. This protocol, which is now an ANSI standard, uses the Escape character, hex 1b, to introduce a control string. Many programmers refer to this as "ANSI escape sequences."

I have yet to find a truly readable version of the ANSI escape sequence standard. The best I've found is from the European Computer Manufacturers' Association and is in the Documentation directory of your class CD as ECMA-048.pdf. A quick perusal of that document will give you a feel for the scope of the ANSI escape sequence standard.

## Slide 7

Bitmapped framebuffer devices come in various shapes and sizes with a range of color depths and various ways of mapping bits and bytes to pixels on the screen. The role of a frambuffer driver is to expose a uniform API that allows applications (and indeed other drivers like the console) to manage these differences in a consistent manner. Among other things, the framebuffer driver can report on the characteristics of the device it manages such as screen size and organization, pixel depth, and so on.

Our kernel includes a framebuffer driver for the S3C2410 LCD controller. Framebuffers are character devices with major number 29. But we're not going to use it directly. Instead, we're going to use a console driver that's attached to device number 4 0. This driver properly attaches itself to the framebuffer driver such that anything written to it ends up displayed on the LCD.

If you feel like digging further into the details of framebuffer drivers, there's a whole subdirectory of the kernel Documentation/ directory called fb/ that has lots of information. Start with the file framebuffer.txt.

## Slide 8

To illustrate how a typical console driver works, go to the directory target_fs/home/src/lcd. By now you know how to create makefile projects in Eclipse and how to import resources into the project. So create a project called lcd and have a look at the makefile to see what targets it builds.

Now open the file lcdutils.c. This defines several simple utility functions that use ANSI escape sequences to write text to random locations on the 240 by 320 pixel LCD screen. Location is specified in terms of row and column based on the 8x16 font that's built into the console driver.

These functions build on one another so that, for example, write_string uses set_cursor and write_number uses write_string.

## Slide 9

The lcd/ directory also has a simple test program to exercise the utilities. Open lcdtest.c to see how it works. Shown here is the command format for lcdtest. s writes a string. n writes a number with a specified width. e erases the screen. b turns the backlight on and off, and finally, q quits the program.

Like most of our sample programs, there's virtually no sanity checking on the input. So if you enter something wrong, it's quite likely it will crash the program.

lcdtest is the all target in the makefile, so it was probably built when you created the project. Now, before you run it, you need to connect the LCD panel to the board. That's illustrated by the photos on the next slide. It's not a particularly difficult process, but it's not exactly obvious either.

## Slide 10

Of course, the whole point of this exercise is to add a display to our thermostat. Here's an example of what the screen might look like, displaying the current temperature and the three operational parameters.

## Slide 11

Start with the thermostat.c file from the network/ directory. Copy that over to the lcd/ directory. I'm suggesting here that the display functionality can be accomplished with the addition of two functions. init_screen() calls LCD_init() and then writes the fixed labels. init_screen() should return the return value from LCD_init(). write_screen() writes the variable values and is called at the end of the main loop.

Make the appropriate additions, build the program and try it out. The build target is netthermo_t. There's no simulation version of this one because we don't have a simulation of the LCD panel. But there's no reason you couldn't build one using the ncurses library. The devices program in the measure directory is an example of how that might be accomplished.

## Slide 12

OK, let's say that the LCD networked thermostat is the final version of our product. The application is working and we have an optimized kernel image and root file system. It's time to load everything into flash and ship it! Well, almost.

First, we have to arrange for our thermostat application code to start up automatically without any console login. The key to that is the init process and the inittab file that it interprets. Init is always the first process that the kernel starts and its job is to get everything up and running properly.

The init executable is typically /sbin/init although there are several alternative locations that the kernel will search if /sbin/init doesn't exist. In the case of our target board, /sbin/init, like virtually every other executable, is simply a link back to BusyBox.

## Slide 13

The inittab file has an interesting format. Take a look at target_fs/etc/inittab. The comments at the top give a pretty good idea of what's going on. Each entry has at least four fields separated by colons as shown in this slide. ID is a unique character string that identifies this entry. By convention, if the associated process is connected to the console, the ID is the terminal name.

Run level is a set of one or more integers ranging from 1 to 6 that specify the run levels for which this process is to be started. Turns out that BusyBox's init process ignores run levels so this field is left blank in our case.

Action describes the circumstances under which the process executes and is taken from the list shown here. The sysinit action in this case is to execute the script rcS. Take a look at rcS just to see what it does.

For more details on the inittab format, check its man page.

## Slide 14

So, one way to get our product up and running is simply to change etc/inittab in the target file system. The Eclipse editor does not automatically make a backup of files you change. If you choose to edit inittab with Eclipse, I suggest you make a backup just in case. Comment out the line that begins ttySAC0 and add the new line shown here. Save the file.

Remember that we're mounting the home/ directory over NFS. We need to change that so home/ becomes just part of the root file system in flash. Edit the boot up script etc/init.d/rcS. Comment the mount command near the bottom that mounts the home/ directory. Save the file.

There are a couple of alternate approaches to starting the application at boot time. We could simply replace /sbin/init by, for example, making it a link to the application executable. In most cases though, the functionality of the standard init process is useful to have.

Another approach is to add the application executable as the last line in rcS.

## Slide 15

We've already loaded our new kernel into the target's NAND flash. The next step is to move the new root file system to flash.

We create an image file of the file system using the mkyaffsimage-128M utility. Note in the slide that the mkyaffsimage-128M command is a single line. Like the kernel, the image file is put in the images/linux/ directory of the Superboot SD card.

Edit images/FriendlyARM.ini. This time we're not going to load the kernel, but we will load the root file system. Copy FrientlyARM.ini to images/ on the SD card.

## Slide 16

The process of moving the file system to NAND is much like moving the kernel. Insert the SD card. Change the boot selector switch and boot the board. The YAFFS file system image is loaded into flash. Move the selector switch back and reboot. The thermostat should start executing.

## Slide 17

Having changed the inittab file to start up the thermostat, the flash root file system is no longer able to run a normal console shell. Having proved our point, you might want to change it back. Just follow these simple steps to get the flash file system back to normal operation.

You need to remove the home/ directory from target_fs-2451 because we mount it from NFS.

## Slide 18

That's it for this week. Let's review. We started out with a review of device drivers from the viewpoint of application programs. We then used that to look at the role of console and framebuffer devices in managing memory mapped video hardware. That led us to build a display for our thermostat.

With the display thermostat working, it was time to put everything in NAND flash and ship it. We looked at how Linux initializes itself and how to get our application started without a shell.

Next week we'll take a look at Linux device drivers in more detail.