UCSD Embedded Linux on line
Week 2

Slide 1

Welcome to Week 2 of Hands-on Embedded Linux. This week we'll install the demo software, hook up the target board, and run our first program.

Slide 2

Here's an outline of what we'll be doing. After installing the tool chain and sample software, we'll take a brief detour to make our Linux host capable of running Blackboard in case you want to view the lessons on it. Then we need to configure a few things on the host. Finally, we'll connect up the target and run a sample program.

Slide 3

Insert the Embedded Linux Learning Kit CD. It should auto mount. cd to your home directory if you're not already there. The first installation step requires you to be root user. Turns out there's a simple command for temporarily becoming root user. It's called su, which many people think means "super user" but actually means "substitute user." If you execute su without an argument it assumes you want to become root user. You are prompted for the root password. Note that the command prompt changes from a "$" to a "#" to clue you in that you're executing as root.

Fedora mounts CDs at /media/<name of the CD>, in this case EmbeddedLinux. If your system mounts the CD somewhere other than /media/EmbeddedLinux, use the alternate version of the command where you enter your mount point as an argument to the script.

When that script finishes, type exit to exit from the root user shell, then execute the install script. Again, if you have a mount point other than /media/cdrom, you'll need to pass that as an argument to the script.

Finally, in order to access the cross tool chain, we have to add the path to those tools to our PATH environment variable. Edit the file .bash_profile in your home directory by double-clicking it in the file manager window. This opens the file in the Kwrite editor. By the way, if you're comfortable with vi or emacs, or any other editor for that matter, feel free to use the editor of your choice. From now on, when I say edit a file, it's your choice.

Down near the bottom of the file you'll see a line that begins with PATH. Edit that line as shown here and save the file.

Slide 4

Let's take a quick look at what we just installed. Under /usr/local is a new directory named arm/ and under it a directory named 4.3.2/. This is version 4.3.2 of the GNU tool chain built for the ARM9 processor on the target board. It includes the tools, like the GCC compiler, along with corresponding libraries and header files that allow us to build executables for the target board.

Also under /usr/local we've added eclipse/, the Open Source IDE that we'll look at next week. Under /usr/src we've added arm/, which has a couple of entries, one of which is simply a generic link to the other. This is a Linux kernel source tree. Several weeks from now we'll configure and build a new kernel for our target board.

Moving on, the /home directory, that is slash home, not <u>your</u> home directory, has a link pointing to target_fs-2451/ under <u>your</u> home directory. The reason for this is something of a historical artifact that I'll explain in the next slide.

Slide 5

In your home directory you'll find some new subdirectories. Busybox is a really neat tool that substantially reduces the memory footprint of the Linux system. We'll look at that about the same time we configure and build the Linux kernel. But the most interesting directory here is target_fs-2451/.

When the kit was based on the Mini2440 board, this was the target board's root file system mounted over NFS. The new boot loader in the Mini2451 somehow gets in the way of mounting the root file system over NFS. Instead, we mount the root file system from NAND flash.

Under target_fs-2451 is home/ and under that are subdirectories include/ and src/. Include/ has a few header files that describe the target board. src/ has subdirectories for each of the projects we'll be working on through the class. Once Linux is running, we can mount home/ over NFS.

Slide 6

Last week I suggested that it's quite feasible to view the Blackboard lessons for the class on your Linux workstation. There are a few necessary steps to make that work right. By now, just about every Linux distro should include a recent version of Firefox. But if your system doesn't include a version 3.x or higher Firefox, I highly recommend installing it. You can download it from mozilla.com. I suggest untarring the download in /usr/local. This creates a new directory /usr/local/firefox.

You may also have difficulty with the default Java runtime environment, the JRE. If so, get the latest JRE from java.com and put it in /usr/local. The download is a shell script. Execute it. You'll be asked to page through the Java end user license and finally accept it.

To get Firefox to recognize Java, create the indicated link in the firefox/plugins directory. To get Eclipse and other applications to recognize the new Java, replace the java link in /usr/bin.

Slide 7

Start your new Firefox. Log into UCSD extension as you have before and navigate to a lesson presentation. Camtasia Studio will probably complain that Flash Player isn't installed and provide you with an appropriate download link. It doesn't matter where you download the tar file.

After untarring the download, move the file libflashplayer.so to the firefox plugins directory.

Restart Firefox, navigate back to your Blackboard page and you should be all set. You should be able to move seamlessly between the lesson presentation and your software development environment.

Slide 8

We'll be using a terminal emulator called minicom to communicate with the target board. Perhaps not surprisingly, there are some parameters we need to adjust to make minicom communicate correctly with the target. In a shell window, as root user, execute minicom –s. You'll get a warning message that there's no config file and the program is reverting to defaults.

Understand that minicom is a text-based application and you won't be able to use the mouse for selection. Use the Down Arrow key and Enter to select Serial Port Setup. Type 'a' and backspace to replace "modem" with "ttyS0" if your host has a true 9-pin serial port, or "ttyUSB0" if you're using a USB to serial converter. Terminate the entry with Enter and then type 'e' to set bps, par, and bits.

Type 'I' for 115 kbaud. Bits, parity, and stop bits should default correctly as 8, N, and 1. If not, type 'q' to set them correctly. Hit Enter to exit the Comm Parameters screen.Type 'F' to toggle hardware flow control off. Software flow control should already be off. Hit Enter to exit serial port setup.

Arrow down to Screen and Keyboard. Type 'b' to toggle the behavior of the Backspace key. Initially, the backspace key sends the backspace code, hex 08, which is not properly recognized by the Linux shell. After toggling, the Backspace key sends the del code, hex 7F, which the shell properly interprets to permit command line editing. Incidentally, the right and left arrow keys work correctly for editing, it's only the Backspace key that needs to change. Hit Enter to exit the screen and keyboard menu.

Slide 9

Arrow back up to Modem and Dialing. Here our intention is simply to eliminate the Init and Reset strings because they are intended for modems and just get in the way when we're talking directly to a serial port. Type 'a' and backspace through the entire string, then type Enter. Do the same thing with 'b'. We don't need to bother with the other parameters on this screen because they will never be invoked given our configuration. Hit Enter to exit from this screen.

Arrow down to Save setup as dfl and then exit minicom.

Slide 10

Very likely, the serial device you have selected, ttyS0 or ttyUSB0, is only readable and writable by the owner, root, and the group, dialout that you're not yet a member of. There are two ways to become a member of any particular group, in this case dialout. Edit the file /etc/group and add your user name to the line that begins dialout. Or use the graphical dialog as described here.

Slide 11

We need to make some changes to the host's network configuration. The easiest way to do this is through the graphical Network Configuration dialog available through KDE. Click the App Launcher > Settings > System Settings > Network and Connectivity > Network Settings. You'll see the dialog screen shown here. Select the Wired tab, which will probably show one connection.

When you installed Linux last week you had the option of getting a network address via DHCP or setting a fixed network address of 192.168.1.2. At that point, you may have chosen to stick with DHCP to be compatible with whatever network you host happens to be connected to.

The issue now is that the target board expects to find a Network File System server at a specific address that happens to be 192.168.1.2. If you selected that in last week's install, you're done. You can close this dialog and move on to slide 13.

Slide 12

Select the Ethernet port and click Edit to bring up the dialog shown here. The gateway address will depend on your network configuration.

If you absolutely must use a different address, we'll make the appropriate adjustment when we boot the target later in this lesson.

Slide 13

You'll want to set up DNS addresses to match what your ISP gave you. The values shown in the previous slide are for my network. When finished, click OK. You'll probably see a dialog with something about the KDE "Wallet Service", whatever the heck that is. Cancel it.

Slide 14

The next step is to make part of the workstation's file system visible over the network so the target can mount it. As root user, open the file /etc/exports in your favorite editor. The install_tools script put this file here. Change <your_user_name> in angle brackets to your actual user name. This has the effect of making your home directory visible on the network.

I won't try to explain what these parameters are. See the exports man page for details. Save the file.

The easiest way to get NFS running is to use the two systemctl commands shown here. It's also possible that the NFS server is not installed on your system. Use the yum command to install it.

If you're running Fedora 17 and need to install additional packages, see the announcement about the status of Fedora 17 for changes to a couple of files in /etc/yum.repos.d.

Slide 15

If you haven't done so already, go ahead and remove the target board from its padded envelope. I haven't had any problems with static with these boards, but it's always prudent to be cautious. Ground yourself before handling the board and, as much as possible, handle it only by the edges.

Slide 16

Here are the basic technical specifications of the board. From the standpoint of computing power, it represents a fairly typical embedded consumer device such as a cell phone, a set top box, or a GPS receiver. Of particular interest, there are three forms of memory on board including RAM, NAND flash, and NOR flash. Each of these memories serves a distinctive purpose as we shall see.

Slide 17

This slide shows a simplified layout of the board to orient you to the various connecters and other features you'll be using during the class. Note that this drawing is turned 90 degrees from the photo in slide 15.

Slide 18

Now it's time to connect and power up the target board. The kit includes an Ethernet crossover cable but not a straight through Ethernet cable. Use the supplied crossover cable if you want to connect directly to your workstation. If you choose to connect it to your network through a hub or switch, get a straight through cable.

Connect the 9-pin serial cable between your workstation and target board.

Before plugging in the power supply, make sure the power switch is in the off position, that is, away from the edge of the board. Also check that the boot select switch is in the NAND position, again away from the edge of the board. Run minicom –w in a shell window on your workstation. The –w option says wrap long lines. For some odd reason that's not a configuration option.

With minicom running, turn on the power switch. After a little while you should see the target boot into Linux and present you with the message "Please press enter to activate this console". You're logged in as root.

Just to prove that Linux really is running, try some simple shell commands like ls, cd, and pwd.

Of course, if you had to choose a different network IP address for your host workstation, the target didn't boot into Linux because it didn't find the NFS server. We'll fix that in the next slide.

Slide 19

There are two fundamental forms of Flash memory, known as NAND and NOR, named for the basic logic structure of the memory cells. These two types have distinctive characteristics that suit them to different functions in the system.

NOR flash is well suited for code storage because of its high reliability, fast read operations, and random access capability. NOR flash resides directly in the processor's address space just like RAM. Because code can be directly executed in place, NOR is ideal for storing firmware, boot code, operating systems, and other data that changes infrequently.

The downside to NOR flash is that write and erase times are relatively slow. That's OK if we're using it to store code because we aren't really writing it that often.

NAND flash, by contrast, is characterized by higher density, lower cost, faster write and erase times, and a longer re-write life expectancy. This makes it useful for storing data. But whereas NOR flash is directly accessible in the processor's memory space just like RAM, NAND flash is accessed over a single 8-bit bus with control signals to differentiate address from data. Consequently, we have to treat NAND flash as a mass storage device with a "file system."

Our board has 256 MB of NAND flash that holds the boot loader, the kernel image and a root file system. At the boot loader prompt, execute the command mtdparts to see how the NAND flash is divided up. The first 256 KB are for the boot loader executable image. The next 128 KB are the boot loader's environment variables. Next comes a 5 MB partition for the kernel image and finally, the remainder of the flash is allocated to the root file system.

Slide 20

Let's pursue this notion of a file system stored in non-volatile memory one step further. In a production embedded environment, the root file system may exist as a compressed image in NOR flash. When the kernel boots, it copies and uncompresses that image into RAM so that the file system can be writable. One consequence of this is that the file system is *volatile*. Any changes made while the system is powered up are lost when power is removed. To make file changes permanent, you have to reflash the image.

But if we store the root file system in NAND flash, we can make changes to files that are reflected back to the flash in real time and so are *non-volatile*.

Linux supports something on the order of 50 different file systems, the most popular of which are ext2 and ext3. ext3 is a journaling version of ext2. The ext file systems generally reside on a mass storage device like a disk and are not well-suited to flash storage devices.

File systems consist of data, the stuff you're interested in, and "metadata," the information that describes how the data is stored on the mass storage medium. If the system is shut down unexpectedly, by a power failure or the user pressing a reset button, the data and metadata may end up in an inconsistent state, usually because disk writes are cached. The result is that the next time the system boots up, it has to check the file system for consistency, which can be time consuming.

Journaling file systems on the other hand, maintain additional metadata that keeps track of the file system state in real-time. If a journaling file system is unexpectedly shut down, it quickly comes back up with the worst case that the data being written when the shutdown occurred may be lost.

File systems on flash devices must deal with issues such as "wear leveling" and bad blocks. Each sector of a flash device has a limited number of writes, usually counted in the hundreds of thousands. A file system for flash memory attempts to maximize the useful life of the device by distributing writes equally over the entire device. Likewise, it detects sectors that have developed flaws and moves data around them.

Ideally, a file system for a flash device will implement these operations transparently so the user or application programmer simply sees the file system as if it were an ext2 or ext3 residing on a hard disk. There are a couple of popular flash file systems in use, JFFS2, the Journaling Flash File System version 2, and YAFFS, Yet Another Flash File System. The target board does have a YAFFS root file system loaded in the NAND flash, but we're not using at the moment because it's not very convenient for software development.

Slide 21

To get a feel for what the target environment is like, list the directories shown in this slide. The /home directory has two subdirectories – include and src. include has a header file describing the target board hardware and src has subdirectories for each of the projects that we'll be working on in this class.

We talked about the /proc file system last week, so just for kicks, execute cat /proc/interrupts just to prove that it does the same thing on the target as it does on your host.

The target's /bin directory is interesting. All but one of the entries is a link to that one entry that isn't a link, namely busybox. The same thing is true for the /sbin directory. The bottom line is that BusyBox is a very clever program that substitutes for a large number of Unix/Linux utilities. We'll look at BusyBox in more detail in a subsequent lesson.

Finally, just for the record, execute the command ifconfig and note that the target's IP address is the one that was set by the command line passed to the kernel by the boot loader.

Slide 22

We're finally ready to do some real programming. We'll start with a variation on the familiar Hello World program that also happens to exercise some hardware I/O. For the moment, we'll just go ahead and build and execute the program without going into a lot of detail. Basically, the led program prints out an introductory message and then sequentially flashes the four LEDs on the board.

Follow the instructions on this slide to build the program on the host workstation and then execute it on the target. It's kind of important to keep track of what's happening where on each of the two computers we're dealing with here. That will probably become more apparent in the next slide.

If you're not familiar with the notation "./" in the last line of this slide, what it means is that the current directory is not part of a normal user's path environment variable and so the shell doesn't search it to find the specified executable. This is a security precaution that is intended to defeat spoofing in a multi-user environment. In effect, "./" evaluates to the current directory.

The bottom line is, you should see the four LEDs on the target board flashing in sequence at a rate of once per second. At this point you have verified the basic communication paths that will serve us for the rest of the class.

Slide 23

So let's be clear on what's really going on here. Last week, in our discussion of how must utilities work, we talked about stdin, stdout, and stderr as being the "standard" file descriptors through which most Linux utilities communicate. Our target environment connects these standard file descriptors to serial port ttySAC0. That port in turn is connected to, probably, either ttyS0 or ttyS1 of your Linux workstation, or maybe ttyUSB0. You then use the minicom terminal emulator to communicate with the target's shell.

The target's Linux kernel was booted from NAND flash and also mounted its root file system from NAND flash. The /home directory on the target is mounted over NFS from the workstation. This means, among other things, that we can execute *on the target* program files that physically reside on the host's file system. This allows us to test target software without having to program it into flash. And of course, programs running on the target can open files on the NFS-mounted host volume.

Slide 24

This slide is a basic memory map of our target board.

The important thing to recognize here is that peripheral components reside in the same address space as memory as far as the processor is concerned. This means that peripherals can be accessed with the same instructions as memory. From the standpoint of C, peripherals can be treated as ordinary integer variables. The program need not know or care whether it's accessing memory or a hardware device. This is in contrast to, for example, the x86 architecture where peripheral devices exist in a separate address space accessed by separate input/output instructions.

Slide 25

But just because peripherals are mapped into the processor's address space doesn't mean you can go off and access them willy-nilly. Linux application programs execute in so-called "virtual" address space that is mapped by the memory management hardware into physical addresses. So before you can access peripheral space, you have to map it into virtual space.

Open led.c with an editor. The first thing to notice, down at line 36, is the variable of type GPIOp. This is defined in include/s3c2410-regs.h as a pointer to a structure that maps the GPIO registers in the address space. Have a look at that file also.

At line 41 we open a file to /dev/mem, which happens to be a character device. The purpose of the mem device is simply to provide a way to map arbitrary sections of physical memory into virtual memory. We then use the file descriptor in a call to the mmap() function that returns a virtual address to the processor's GPIO space. We're asking to map the size of the S3C2410_GPIO structure beginning at 0x56000000. We want both read and write access and we're willing to let other processes share the same space.

The S3C2451 supports nine general purpose I/O ports ranging from 8 to 25 bits wide. These are designated GPA through GPJ (I isn't used). Virtually all of the bits can be assigned to one of three functions: direct digital input or output, or a pre-assigned internal peripheral function. The kit CD includes a user's manual for the S3C2451 that describes all of the peripheral functions in detail.

Referring back to led.c, we see that the four LEDs are split between GPIO port A and GPIO port B. These four bits initialized as general purpose output and are initially set to 1, which is "off" as far as the LEDs are concerned. Then in the loop, one bit is cleared to 0 and then set to 1 after a one second delay. Shift the bit pattern left and do it again.

Slide 26

Linux changes the way we have traditionally done embedded programming, or at least the way I grew up with embedded programming. Embedded systems almost always comprise some combination of RAM and non-volatile memory in the form of ROM, PROM, EPROM or flash. The traditional way to build an embedded system is to create an executable image of your program, including all library functions statically linked, and perhaps a multitasking kernel. You then load or "burn" this image into one or more non-volatile memory chips. When the system boots, the processor begins executing this image directly out of ROM.

In the Linux view, programs are "files" that must be loaded into memory before executing. Therefore, we create a ROM "file system" containing file images of whatever programs the system needs to run. This may include various utilities and daemons for things like networking. These programs are then loaded into RAM by the boot initialization process, or as needed, and they execute from there.

Generally, the C library is not statically linked to these image files, but is dynamically linked so that a single copy of the library can be shared by whatever programs are in memory at a given time.

One of the advantages of the Linux approach is that we're not confined to loading program files from the ROM file system. As slide 24 shows, we can just as easily load programs over a network for testing purposes.

Slide 27

This is probably a good point to step back and take a look at what we've done so far. This slide and the next one summarize the steps we've gone through to get both the workstation and target configured and talking to each other. We started out by installing the class software consisting of a cross tool chain for the ARM processor, the Eclipse IDE, a Linux kernel source tree, and sample code. We configured the minicom terminal emulator on the workstation and extended the PATH environment variable to include the cross tool chain.

Slide 28

We gave the workstation a fixed network IP address so the target can easily find it to mount the /home directory and we exported part of the workstation's file system via NFS.

On the target, the IP address is set in the command line that's passed to the Linux kernel when it starts. The kernel itself is copied from flash to RAM and started. It mounts its root file system from NAND flash then mounts then mounts the /home directory from the workstation using NFS.

That's it for this week.

Next week we'll look at the Eclipse Integrated Development Environment and how to debug our target programs remotely.