UCSD Embedded Linux online
Week 6

Slide 1

This week we're going to shift gears somewhat. Up to this point we've been looking at Linux from the application perspective, that is, how to build and run applications under Linux. This week we're going to start looking at the Linux kernel itself.

The neat thing about Linux as an embedded operating system is that you get the source code. While most of us will have no need to directly hack the kernel sources, we often have good reason to be able to configure the kernel to exactly match the specific requirements of a given hardware environment or application scenario. Fortunately, the process of configuring and building the kernel has improved substantially over the years to the point where it is now a good confidence-building exercise for developers new to Linux

Slide 2

So why would you want to build a custom kernel? Well, first of all because you can. But practically speaking, if you have a standard desktop Linux distribution, you might want to rebuild the kernel simply to reduce the bloat caused by all the hardware support that your particular system doesn't use. The default kernel that comes with most standard distributions is fairly large because it must, of necessity, encompass a very wide range of hardware configurations. You can strip out the unnecessary hardware support, reducing the required disk space and, to a lesser extent, the runtime memory footprint.

You might also find that the standard kernel doesn't support some peculiar feature of your machine. Historically, this seems to have been particularly true of laptops, which I found had problems with display resolution, CD-ROM drives, and mouse input.

Another common omission in standard Linux kernels is support for DOS and Windows file systems. This is useful if you want to build a "dual-boot" system and share files between the DOS/Windows environment and Linux.

But, as an embedded developer, probably the main reason you need to build a custom kernel is not for your workstation, but for your embedded target environment that may not even be the same architecture as your workstation. Chances are you will need to configure a kernel to meet the very specific resource constraints of your target.

Slide 3

Before we dive into the details of configuring the kernel, let's look at a couple background issues first.

Perhaps not surprisingly, there's a very specific policy for numbering kernel versions as shown here. The first number is the "major revision", in this case 3. This number supposedly increments only when truly major architectural changes are made to the kernel and its API. But in fact, Linus decided in May of 2011 to roll the major revision from 2.6. to 3 in honor of the 20th anniversary of Linux. The actual changes were minimal. Prior to that we had been at 2 for something like 12 years. The major revision rolled to 4 in late April of 2015.

The second number, 6 in this example, is called the "minor version" and represents changes where the kernel API may differ but the differences aren't enough to justify a major version change. This number rolls about every two to three months as the kernel evolves.

In many cases there's a third number represents bug fixes and security patches only. No API changes are allowed at this level.

The text following the dash serves a couple of different purposes. Development versions of the kernel are identified with a string like "rc4" representing "release candidate 4." When that minor version is deemed stable the "rc" string is dropped and that version becomes the new latest stable release. Then the minor version is incremented and the process starts over.

The other use for this string is that when you build your own kernel with your specific feature set, you should set it to represent what you did. In this case the "FA" says that this kernel is configured for the FriendlyARM Mini2451 board. Established Linux distributions do the same thing because they make their own modifications to the kernel. This source tree is in fact a "fork" of the kernel created by FriendlyARM.

It's not at all unusual to have several versions of the Linux kernel and corresponding source code on your system. Generally, Linux sources are installed as subdirectories of /usr/src, although in our case we've put it in /usr/src/arm to make it clear that we're building it for the ARM architecture. The subdirectories usually get names that reflect the version number. Then you create a link called linux in /usr/src/arm that points to the version you're currently working on.

The shell command uname –a will give you lots of information about the running kernel, including the version string. Try it out and notice what the additional features string is.

Slide 4

The Linux kernel is divided into "upstream" and "downstream" products. The upstream kernel is maintained by kernel.org and, as I mentioned in the previous slide, a new release of that kernel appears every couple of months. Vendors and distributors create their own downstream kernels from specific releases of the upstream kernel. For example, Ubuntu 10.04 is based on the 2.6.32 kernel and Fedora 17 is based on 3.3.4. Downstream kernels are typically released as a collection of patches to be applied to the corresponding upstream kernel.

Slide 5

So which version should you use? At the risk of stating the obvious, pick the one that meets your needs. I like to stay a few minor revisions behind the latest release on the theory that it's likely to be a little more stable. In our case, the upstream kernel does not support the Mini2451. We're using a fork of 3.6 created by FriendlyARM.

In any case, resist the temptation to "chase the kernel." It changes daily and , unless you're a kernel developer, that's an exercise in frustration.

Slide 6

Needless to say, the kernel source tree is pretty large. The 3.6 tree that we're using has over 40,000 files in close to 3000 directories adding up to 450 MB. Perhaps not surprisingly, there's a standard directory tree to organize these files in a manageable fashion as shown here. Follow along on your workstation as I describe the major subdirectories of the source tree.

Documentation is pretty much self-explanatory. This is a collection of text files describing various aspects of the kernel, problems, "gotchas", and so on. There are several subdirectories under here for topics that require more extensive explanations. This is a good place to look if you're puzzling over some bizarre behavior of the kernel.

arch – All architecture-dependent code is contained in subdirectories of arch. Each architecture has a directory under arch with its own subdirectory structure. You might think of the whole arch sub-tree as the "board support package" for Linux.

arch turns out to be the largest sub-tree with over 12,000 files in 886 subdirectories.

crypto – Code dealing with the cryptographic aspects of system security. I have to admit I've never really looked at any of this.

drivers – Device driver code. Under drivers is a set of subdirectories for various devices and classes of device. This is the second largest sub-tree with 9500 files in 474 subdirectories.

fs – Filesystems. Under fs is a set of directories for each type of file system that Linux supports.

include – Header files. The most important subdirectory of include is linux, which contains pretty much all of the header files you need for the kernel.

init – As its name implies, this holds code for initializing the kernel when it boots up.

ipc has code to support Unix System 5 Inter-Process Communication mechanisms such as semaphores, message passing and shared memory.

kernel – This is the heart of the matter. Most of the basic architecture-independent kernel code that doesn't fit in any other category is here. Things like the scheduler, for example.

lib has several utility functions that are collected into a library.

mm is memory management functions.

net is network support. Subdirectories under net contain code to support various networking protocols.

scripts has text files and shell scripts that support the configuration and build process.

security offers alternative security models for the kernel

sound provides support for sound cards and the Advanced Linux Sound Architecture

Take a little time to browse through some of these subdirectories, particularly arch and drivers, to get a feel for what's there.

Slide 7

The process of building a kernel begins by invoking one of the many make targets that carry out the configuration process. But first we have to edit Makefile to specify that we're building the kernel for an ARM processor. By default Makefile builds the kernel for the architecture on which it's running, which in the vast majority of cases is some variant of x86.

Open Makefile with an editor and go to line 195. Edit the line to read as shown here. Note that arm as used here is the name of a subdirectory under arch. The next line, CROSS_COMPILE also needs to change, but that can be done from the configuration menu. Save the edited file.

Alternatively, we could specify the ARCH variable on the make command line. Personally I find it easier to edit the Makefile.

The Makefile has a very large number of make targets, one of which is help that will list all the targets the Makefile builds. Give that a try.

The kernel configuration file is called .config and it resides at the top level of the kernel source tree. But it doesn't exist initially. We'll create it by copying the file mini2451_linux_config to .config. Note that if the Mini2451 were properly supported in the upstream kernel, there would be a defconfig target for it.

Slide 8

After some program and script building, the xconfig menu comes up. As shown here, it's divided into three panels. On the left is a navigation panel that lists option categories. Select a category and its options show up in the upper right panel. Select an option and help for that option shows up in the lower left panel. And in most cases it's actually helpful. It really does explain what the option is.

As an example, let's select the second category, "General setup," and the first option, "Prompt for development and/or incomplete code/drivers". In any given kernel release, there are some features that are considered to be "experimental," meaning they haven't had sufficient testing, or reached a level of maturity where they could be considered stable. If you turn this option off, by clicking the checkbox, then you won't even see the experimental options and won't have the choice to turn them on.

To illustrate, leave the prompt option turned on, then select Loadable module support in the navigation panel. Note that there's an option called Forced module unloading. Go back to General setup options and deselect the Prompt option. Now go back to Loadable module support and note that Forced module unloading doesn't appear.

Even though this particular feature has been around for quite some time and is pretty mature, it's included in the experimental category because it's really only needed if you're developing and debugging kernel code.

Similar to the experimental option, there are many features that are dependent on the state of other features. If the dependencies aren't satisfied, those features won't be visible.

Slide 9

Most kernel options fall into one of the first two types shown here. Boolean options, as the name implies, have just two values – one or off. And, as you saw in the last slide, the state is toggled by clicking the check box. Tri-state options have three values. Click an empty box once and a dot appears. Click it again and the check appears. These are options that can either be built into the kernel executable image or built as kernel loadable modules. We'll look at kernel modules in more detail in a couple weeks.

An example of a tri-state option is in the Power management options category. Click Power Management support, then click Advanced Power Management Emulation. The dot appears meaning this feature would be built as a module. Click it again and the check appears, so now this feature would be built into the kernel. Click it one more time to turn it off.

Some options are numbers. An example of this is in the boot options category. Double-click the Compressed ROM boot loader base address option. A dialog box opens at the bottom of the option pane allowing you to change this value.

There's probably only one text string option in the kernel and that's under General setup. The second item, Local version, is a text string and is edited the same as number options by double-clicking. Go ahead and enter "-mini2440" here. You have to terminate the string with the enter key to get the new value to "stick."

Finally, there are some "radio button" options where you select one value from a list of possible values. An example of that is in the System type category. ARM system type lets you select a specific chip from among some two dozen different ARM implementations. Currently the Samsung S3C2410, etc, etc, is selected. That selection in turn enables a sub menu with a number of specific implementations of the S3C architecture in the form of "machines". Our machine is the FriendlyARM Mini2440 development board that shows up under S3C2440 Machines

Needless to say, there are a <u>lot</u> of configuration options. Most of the options have to do with supporting various hardware devices. That's the section called Device drivers. For the kernel that comes with a standard distribution like Fedora, just about everything here is turned on and built as kernel modules. In our case, not much is enabled and the features that are, are built into the kernel image. This configuration doesn't build any kernel modules even though we've got module support turned on.

Check the Graphics support section under Device Drivers to verify that support for frame buffer devices is selected and under that, S3C2410LCD framebuffer support is enabled.

Another interesting section is file systems. Currently, MSDOS and VFAT file systems are enabled. I suspect that's because the board has an SD/MMC socket and you might plug in an MSDOS formatted card. Otherwise, it hardly seems necessary.

Take a look at Kernel hacking. This section offers lots of interesting features for testing and debugging the kernel itself.

Go ahead and browse around the other sections to get a feel for what all is here.

Slide 10

xconfig has an Options menu that turns on the display of additional information. Try it out as I describe what each of the options does.

Show Name adds a column to the Option pane that displays the names of the configuration environment variables as they are called out in the .config file. Independent of this option, the configuration variable name always appears in the Help pane.

Show Range indicates the range of possible values for variables. This only seems to be relevant for Boolean and tri-state variables. That is, you can easily identify from the three columns, N, M, and Y, which variables are Boolean and which are tri-state.

Show Data displays the current value of all variables. This is somewhat redundant for Boolean and tri-state variables as the Show Range columns also display the current value. It is useful for numeric and text values. When this column is turned on, numeric and text values are entered here instead of at the bottom of the Option pane.

Then there's a set of radio button options that controls what options are displayed.

Show Normal Options displays the options that are valid based on the current configuration. This is the "normal" setting.

Show All Options displays all options that are otherwise not visible because dependencies are not satisfied. For example, there's some option that you just know should be there but it's not showing up. Turn on Show All Options.

Show Prompt Options is somewhat new and I'm not really clear on what it does.
OK, we're not really going to change anything at this point. My objective was to introduce you to xconfig and all its features. So when you're finished browsing around, click the Close button and then click Discard changes.

Slide 11

Open the file .config with an editor. An excerpt is shown in this slide. The first thing to notice is the comment at the top. Don't try to manually edit this file. It is considered bad form to manually edit a file that was created by a tool.

.config is really just a large collection of environment variables whose names all start with "CONFIG_". The remainder of the name matches the name displayed by the Show Name option in xconfig. Variables that have not been enabled are commented out. Enabled Boolean variables get a value of "y". Tri-state variables can be either "y" or "m" for module. Numeric and text string variables get the values you set.

.config is included by the Makefile, which uses many of the environment variables to determine which source files to build and how. .config is also converted into a header file, autoconf.h, so the configuration variables are also visible to GCC to be used to control conditional compilation.

Slide 12

There are a couple of other configuration targets that are worth mentioning.

make menuconfig was the original menu-driven configuration interface. It uses the ncurses library to create a pseudo graphical interface on a text screen. Some hackers still prefer menuconfig. We'll see it when we talk about BusyBox.

make oldconfig is useful when you're upgrading to a later kernel version. Inevitably, there will be new configuration options. oldconfig adds these new options to your .config file, prompting you for a value for each of them. I find that the default values are fine and so I just hit enter until it finishes.

make defconfig creates a default configuration. How useful is that? Don't know, I've never tried it.

Slide 13

OK, let's build a kernel! Strictly speaking you should always do a make clean before building the kernel to make sure that everything gets rebuilt. But the first time you do a build it's not really necessary because there's nothing to clean. The build command is simple enough, it's just make. The final product of make is two files that show up in arch/arm/boot. Image is the actual executable kernel and zImage is a compressed version that carries with it its own decompressor. Note that zImage is about half the size of Image.

The make process is recursive. Virtually every subdirectory in the source tree has its own Makefile that's responsible for building the files in that directory. So the top-level Makefile works its way down each branch of the source tree in turn until it has traversed the entire tree. Needless to say, building a kernel takes a while.

Slide 14

So where does the information come from to build the xconfig menus? It comes from a set of files named Kconfig that are scattered around the source tree much like the Makefiles. These Kconfig files are effectively scripts written in what's called "configuration language." The top level Kconfig file is in the subdirectory of arch that represents the architecture we're building the kernel for, in our case, arch/arm.

Fire up make xconfig again and open arch/arm/Kconfig with an editor. Turn on Show all options. Comparing the xconfig menu with the Kconfig file should give you a pretty good idea of what's going on. I won't go into any detail on configuration language other than to point out that the keyword source is the equivalent of #include in C and is the mechanism for extending the menu with other Kconfig files.

The documentation file listed here is a pretty thorough description of configuration language.

It's useful to know about Kconfig files because at some point you might want to add something to the kernel configuration, a new device driver for example. You do that by editing the appropriate Kconfig file to add your new configuration option.

Slide 15

In order to test our new kernel, we'll have to load it into NAND flash. There is an alternate boot mechanism called "Superboot" on the Mini2451 for the purpose of loading files into NAND flash. It boots from an SD card. The boot selector switch is next to the green audio connector. Move the switch toward the edge of the board and insert a properly formatted SD card. Then turn on power or push the reset button. The alternate bootloader follows instructions in a script file to burn various files into NAND flash

The first step is formatting an SD card, which should be at least a GB. The kit CD includes a Windows utility called SD-Flasher that formats an SD card for use as a bootloader. Copy the file SD-Flasher.zip on the class CD to a Windows machine and unzip it. The only file it contains is SD-Flasher.exe. SD-Flasher has been tested on Windows 7 and is not guaranteed to work on XP. Also, there are reports that the integrated card readers in some notebooks and laptops don't work well, so it's best to use an external USB card reader.

SD-Flasher erases all data and creates two partitions on a FAT32-formatted SD card: a 130 MB partition for the bootloader itself and the remainder of the card for the files to be burned into NAND flash.

Slide 16

Right-click on SD-Flasher.exe and select Run as administrator. Select the Machine type as Mini2451/Tiny2451. With a card inserted in the reader, click Scan. The result is shown Here. Available says No, meaning the card has not been formatted by SD-Flasher. Now click Relayout. It asks you to confirm that all data on the "disk" will be lost. When it finishes, click Scan again. The label has changed to FRIENDLYARM and Available says Yes.

With the kit CD inserted, click on the … next to the file dialog and browse to /images/Superboot2451.bin. Now click Fuse to load Superboot onto the card. In the Reports window you should see "1 Total, 1 Succeeded, 0 Failed, 0 Skipped". Quit SD-Flasher. Finally, copy the entire /images directory to the SD card.

Slide 17

Have a look at the file FriendlyARM.ini in the /images directory. It's written in a script language invented specifically for Superboot. The file is currently set up to install Linux, the bootloader, kernel, and root file system. It also specifies the command line to be passed to the kernel when it boots.

For the moment, our interest is in testing out the new kernel we built in the last chapter, so you can comment out the lines for Linux-BootLoader and Linux-RootFs-InstallImage.

Copy linux/arch/arm/boot/zImage to images/linux on the SD Card.

Slide 18

Follow the instructions here to:

- Invoke the SD bootloader
- Load the new kernel into NAND flash
- Reboot

To prove that you booted the new kernel, execute uname –a after the system boots. One piece of information uname puts out is the date the kernel image was built. It should be today's date.

More details on Superboot can be found in Chapter 13 of ELLKUsersGuide.pdf found in the Documentation/ directory of the class CD.

Slide 19

Let's turn our attention to another tool in our embedded bag of tricks, BusyBox. This is a strategy for substantially reducing the storage footprint required by Linux utilities. Even if your embedded device is "headless", that is it has no screen and/or keyboard in the usual sense for user interaction, you still need a minimal set of command line utilities. You'll no doubt need mount, ifconfig, and probably several other utilities to get the system up and running. Remember that every shell command line utility is a separate program with its own executable file.

The idea behind BusyBox is brilliantly simple. Rather than have every utility be a separate program with its attendant overhead, why not simply write one program that implements *all* the utilities? Well, perhaps not all, but a very large number of the most common utilities. Most of these programs require the same set of "helper" functionality, like parsing command line options. Rather than duplicating these functions in dozens of files, BusyBox implements them exactly once.

In many cases, the BusyBox version of a utility omits some of the more obscure options and switches. Think of them as "lite" versions. Nevertheless, the most useful options are preserved.

Take a look at the bin/ directory in the target filesystem. The only real file in bin/ is busybox. Everything else is a link to that file, which happens to be an executble. The same for /sbin. Remember that the first argument passed to any program under most operating systems is the name of the command that invoked the executable. That's how BusyBox figures out what it's really supposed to do.

Slide 20

As with the Linux kernel, it's useful to have a known good configuration to start with when building a new version of BusyBox. And like the kernel, the default configuration file is called .config. There is a file in the factory_images/ directory called bb_config that serves as a starting point for our board.

BusyBox supports make xconfig just like the kernel. There are basically two categories of configuration options – BusyBox settings and Applets, of which there are several sub-categories. Busybox settings contains options that control the behavior of Busybox as a whole and how it is configured and built. Of particular interest is the build option cross compiler prefix that allows us to specify a cross compiler. Take a look at that one. The compiler prefix should already be set correctly to arm-linux-.

Another interesting feature is under Installation options and that's the Installation prefix. That one is set to a directory under busybox..

Now browse through the various categories of applets. There's actually quite a bit of stuff turned on that we probably don't even need in our development target environment, much less a production device. The Archival utilities are a good example, and maybe editors.

Select the basename command under Coreutils and take a look at the Help text. This provides a brief description of the command, the configuration variable name and where it's defined. Busybox, like many configurable open source projects, uses the same configuration language that the kernel uses.

Scroll down to ls in Coreutils to see an example of a utility with several optional features. When you've had a good look through the various applet categories, decide what you think you don't really need and disable those features.

Slide 21

Exit from xconfig, saving the new configuration file, and execute make. When that's finished, execute make install.

When make install completes, you'll find a new directory in the BusyBox tree named _install. Under that you'll find a partial Linux root file system consisting of the bin, sbin, and usr directories. bin has the BusyBox executable plus links for those utilities that normally reside in bin. sbin has links for the utilities it normally contains. usr is basically more of the same in its subdirectories bin and sbin. You would then copy these directories to your target's root file system.

Alternatively, you can execute the make command on the last line of this slide. This will install Busybox and the links directly into the target's root file system. You can also set the install location as a configuration option.

Slide 22

So that's it for this week. Time to review. We started out with a look at the kernel source tree and related issues such as how version numbers are generated and the distinction between upstream and downstream kernels.

Then we got into the process of configuring and building the kernel. There are several make targets for configuring the kernel, but make xconfig is the most user friendly. You should always to a make clean to be sure everything is built according to the current configuration. make by itself then builds the kernel and any associated modules.

Next we saw how to load the new kernel into NAND flash on the Mini2451 by using an alternate bootloader mechanism called "Superboot". Finally we looked at BusyBox, a tool for substantially reducing the storage footprint of an embedded system.

Next week we'll go back to working with the target and make use of the LCD screen. We'll also go through the process of installing final firmware on the board.