

**E.L.L.K. III**

**The  
Embedded Linux  
Learning Kit Version 3  
User's Guide**

**Doug Abbott  
Intellimetrix**

For the Raspberry PI 3 single board computer

## **Copyright Notice**

The text and graphics in this manual, its cover, the E.L.L.K. III CD-ROM and its artwork are copyrighted and protected from misuse under local and international laws. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means electronic, mechanical, photocopying, recording, or otherwise without prior written permission of the author.

## **Limitation of Liability, Intended Use**

The Embedded Linux Learning Kit, the kit, is intended as a tool for learning to use Linux in an embedded context. Intellimetrix makes no warranty, representation or guarantee regarding the merchantability, suitability or fitness of the kit, or any of its individual components, for any particular purpose. All software is supplied “as is.”

## **Updates**

In the interest of improving quality and maximizing value to the customer, this manual and the contents of the associated CD-ROM are subject to update and revision at any time. Electronic updates will be posted at [www.intellimetrix.us/downloads.htm](http://www.intellimetrix.us/downloads.htm).

## **Revision History**

2-19-2019: Initial release

## **Contact Information**

The author may be contacted at:

Intellimetrix  
193 Happy Trails N  
Las Cruces, NM 88005  
Phone: +1 575-590-2788  
Email: [doug@intellimetrix.us](mailto:doug@intellimetrix.us)  
Web: [www.intellimetrix.us](http://www.intellimetrix.us)

# Embedded Linux Learning Kit

<b>Preface .....</b>	<b>1</b>
What's in the kit? .....	2
Resources .....	2
<b>1. Getting Started.....</b>	<b>4</b>
Linux installation .....	4
Installation scenarios .....	4
Installing Virtualbox .....	5
Installing Linux.....	7
Getting Help .....	10
Resources.....	11
Linux distributions.....	11
<b>2. The Host Development Environment.....</b>	<b>13</b>
Installing the kit software .....	13
The Terminal Emulator, minicom .....	14
Networking .....	16
Network Interface .....	16
Network File System (NFS) .....	17
The Cross Development Environment.....	18
<b>3. The target .....</b>	<b>19</b>
The boot Partition .....	20
Changing the Target's Network Address.....	20
Booting the Target .....	21
What can go wrong? .....	21
Resources.....	22
<b>4. Eclipse Integrated Development Environment .....</b>	<b>23</b>
Introduction .....	23
Getting Started .....	23
Java Runtime .....	23
Starting Eclipse .....	24
Additional Plug-ins.....	25
The C Development Environment (CDT).....	26
pi-lib .....	27
Creating a Project .....	27
The target execution environment .....	29
Remote debugging with GDB .....	30
Establishing an SSH Connection to Eclipse .....	31

Debugging with CDT .....	32
Resources.....	34
<b>5. Working with Hardware .....</b>	<b>35</b>
User Space vs. Kernel Space .....	35
The ARM I/O Architecture.....	35
I/O in Linux .....	36
The Low Level I/O API.....	36
Driver internal structure.....	38
The /sys File System.....	40
Accessing GPIO Through /sys.....	40
The led Program .....	41
PIGPIO—Another Approach to Hardware Access.....	42
WiringPI .....	43
Resources.....	43
<b>6. A Real-World Application .....</b>	<b>44</b>
The PFC8591 A to D Converter .....	44
<b>7. A Thermostat .....</b>	<b>46</b>
Host workstation as debug environment.....	47
<b>8. Posix Threads.....</b>	<b>50</b>
Creating a Linux Process – the fork() function.....	50
Introducing Threads.....	53
Thread Attributes .....	54
Synchronization—Mutexes .....	55
The thermostat with threads.....	55
Changes required in thermostat.c.....	56
Debugging threaded programs.....	56
Resources.....	57
<b>9. Networking.....</b>	<b>58</b>
Sockets.....	58
The Server Process .....	58
The Client Process .....	59
Socket Attributes .....	59
A Simple Example.....	60
The Server .....	60
The Client .....	61
Try it Out .....	62
A Remote Thermostat.....	62
Multiple monitor threads .....	63

Resources.....	64
<b>10. An Embedded Web Server .....</b>	<b>66</b>
Background on HTTP.....	66
A Simple embedded web server .....	67
cmake.....	67
Configuring and Building Monkey.....	68
Running Monkey .....	68
Resources.....	69
<b>11. The OLED Driver .....</b>	<b>70</b>
The Serial Peripheral Interface (SPI) Bus.....	70
The Video Frame Buffer.....	70
The Frame Buffer Driver .....	71
Exercising the Display .....	72
Thermostat display .....	72
ncurses library.....	73
Qt Extended .....	73
Resources.....	73
<b>12. Configuring and Building the Kernel .....</b>	<b>74</b>
The kernel source tree.....	74
“Upstream” vs. “Downstream” kernels .....	74
Configuring the kernel .....	76
make xconfig .....	77
make menuconfig .....	79
Building the kernel .....	79
make modules_install .....	80
Testing the New Kernel .....	80
Next Step .....	80
Resources.....	81
<b>13. BusyBox .....</b>	<b>82</b>
Resources.....	84
<b>14. The Bootloader.....</b>	<b>85</b>
Raspberry Pi Boot Process.....	85
U-boot.....	85
Background.....	85
Configuring and Building U-boot.....	86
Running U-boot .....	87
Device Trees.....	88
Bootting over the Network .....	90

Setting up the TFTP Server .....	90
Resources.....	91
<b>15. The Final Steps: Linux Initialization and Flash File Systems.....</b>	<b>92</b>
Linux Initialization .....	92
Loading the Application to NAND flash.....	93
Systemd – the Newer Initialization Mechanism .....	94
Resources.....	95
<b>16. Yocto.....</b>	<b>96</b>
The Problem .....	96
One Solution: Yocto .....	96
Getting Started with Yocto .....	97
Metadata .....	99
Testing the New Image – QEMU .....	101
Yocto for the Raspberry Pi .....	101
Alternatives.....	102
Buildroot.....	102
Linaro .....	102
Resources.....	102
<b>17. The Next Steps .....</b>	<b>103</b>
Web Resources .....	103
Commercial sources of embedded Linux .....	104
<b>Appendix A: Target Board Connectors.....</b>	<b>105</b>
Raspberry Pi .....	105
Matrix CK.....	105
<b>Appendix B: Bootloader Commands .....</b>	<b>107</b>
Information Commands .....	107
Memory Commands .....	107
NOR Flash Memory Commands .....	108
NAND Flash Memory Commands .....	109
Execution Control Commands.....	110
Download Commands .....	110
Environment Variable Commands.....	111
Environment Variables .....	111

## Preface

*“ ‘You are in a maze of twisty little passages, all alike’  
Before you looms one of the most complex and utterly intimidating systems ever written.  
Linux, the free UNIX clone for the personal computer, produced by a mishmash team of UNIX  
gurus, hackers, and the occasional loon. The system itself reflects this complex heritage, and  
although the development of Linux may appear to be a disorganized volunteer effort, the sys-  
tem is powerful, fast, and free. It is a true 32-bit operating system solution.”<sup>1</sup>*

No doubt about it. Linux is intimidating to those just encountering it for the first time.

Lots of vendors will sell you an “embedded Linux development kit.” It typically includes a single board computer (SBC), a power supply, necessary cables, and either a CD with a random collection of Linux software or links to the same on the web. If you’re lucky, the CD may include some poorly written documentation attempting to explain how it all fits together. With enough perseverance you may even get it working. Knock yourself out, dude!

In contrast, the Embedded Linux Learning Kit (ELLK) from Intellimetrix aims to teach embedded Linux in a practical, self-paced, hands-on environment. The various software packages on the CD have been configured to work together with the instructions in this manual to lead you through the process of bringing up Linux on a target SBC and getting your embedded application running. It’s the kind of tool I wish I had had when I was climbing that steep Linux learning curve a few years back.

This is the third generation of the ELLK. Each successive generation has been smaller, faster, more capable, and cheaper than the predecessor. That seems to be the way computer science works.

Note: The ELLK is not an introduction to Linux. I assume that you have successfully installed a Linux system and have at least played around with it some. You know how to log in, you’ve experimented with some of the shell commands and have probably fired up a GUI desktop. It goes without saying that a familiarity with C programming is expected. And of course, any credible embedded software engineer needs at least a passing acquaintance with basic digital hardware concepts.

With that in mind, the ELLK course is directed at two different audiences:

- The primary audience is embedded programmers who need an introduction to Linux in the embedded space. This is where I came from and how I got into Linux so it seems like a reasonable way to structure things.
- The other audience is Linux programmers who need an introduction to the concepts of embedded and real-time programming.

---

<sup>1</sup> *Linux Installation and Getting Started*, Matt Welsh, et al

Consequently, each group will likely see some material that is review although it may be presented with a fresh perspective.

## What's in the kit?

This third generation kit is based on the Open Source Raspberry Pi 3 Model B. The board will usually be referred to as the “target” or by a shorthand for its model name “R PI”. Basic features of the target board include:

- Quad core 1.2 GHZ Broadcom processor, 64-bit ARM Cortex A8
- 1 GB RAM
- Micro SD slot
- BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board
- 10/100 Ethernet port
- 4 USB 2.0 host ports, Type A
- Serial debug port, 3.3 volt signal levels
- Micro USB, primarily for power

In addition to the board, the kit contains:

- 8 GB micro SD card with boot loader, Linux kernel, and root file system image
- Micro USB power supply
- Matrix Compact “hat” that plugs into the 40-pin GPIO connector and provides a range of peripheral devices including:
  - Four LEDs
  - Three pushbuttons
  - Buzzer
  - 4-channel, 8-bit A/D converter with pot attached to channel 0
  - 160x80 TFT display
  - Temperature Transducer
  - Infrared receiver
- CD ROM with:
  - Yocto Project build system based on Poky 2.6, “thud”. This in turn provides:
    - GNU cross tool chain for ARM
    - Linux kernel source
    - Target root file system
  - Eclipse integrated development environment (IDE)
  - Sample source code
  - Other miscellaneous goodies
- This users' guide

## Resources

[www.Intellimetrix.us](http://www.Intellimetrix.us) -- The Downloads page of my website will host any updates to the kit software and users' guide.



<https://groups.google.com/forum/#!forum/embedded-linux-book> -- This is a Google Group discussion forum for my book, *Linux for Embedded and Real-time Applications*. It's now also a discussion forum for ELLK users.

# 1. Getting Started

In many cases, the target computer on which an embedded application runs is severely limited in terms of its computing resources. It probably doesn't have a full-sized display<sup>2</sup> or keyboard. It may have as much as a few gigabytes of mass storage in the form of a flash file system, probably not enough to contain a compiler much less a decent integrated development environment (IDE). Thus embedded development usually requires at least two computers—the target on which the embedded program will run and a development workstation on which the embedded program is written and compiled. Before we begin working with our embedded Linux environment, we'll have to set up an appropriate development workstation.

Any modern PC will work just fine as a development host. Minimum requirements are: a Pentium class processor, 4 GB of RAM for graphical operation, and at least 100 GB of disk for a “workstation” class Linux installation. Of course, more RAM and disk are always better. You will need an asynchronous serial port, which can be a USB to serial converter, and a network interface, which can also be a USB converter.

## Linux installation

Since this is not a beginners' guide, we won't say much about installing Linux itself. Pick your favorite Linux distribution—CentOS, Ubuntu, Debian, Suse, whatever. Any fairly recent distribution should work just fine. I'm currently partial to CentOS and run CentoOS 7 as a guest machine under Sun Microsystems' (excuse me, Oracle) VirtualBox<sup>3</sup>. We will be describing some configuration and setup changes later on in terms of CentOS 7. Other distributions present these options differently. In any case, these days Linux installation is a fairly straightforward process.

### *Installation scenarios*

There are basically three installation scenarios for Linux:

- Stand-alone – This is the obvious choice if you can dedicate a machine to Linux. You will let the installation process format the entire disk.
- Dual-boot – In many cases though, you'll probably want to install Linux on a machine that already runs some variant of Windows. Historically dual booting was the way to do this. In this scenario, you select at boot time which operating system to boot. That OS takes full control of the machine. The Linux installation will replace the standard Windows boot loader with GRUB, the GRand Unified Bootloader. GRUB then offers the option of selecting the OS to boot. Before running the Linux installation, you'll need to make enough contiguous free space

---

<sup>2</sup> Actually, the R Pi supports HDMI video and we'll touch on that briefly.

<sup>3</sup> In addition to Oracle's VirtualBox, VMware offers a free version of its virtualization product, VMware. See the Resources section for websites.

available on the disk beyond the space occupied by Windows. The Windows 10 disk manager will help you do this. The installation then formats only the free space.

- **Virtualization** – This is the big buzzword in computing these days. This is the process of running one operating system on top of another. The base, or native, operating system is called the *host*. It runs a *virtual machine manager*, VMM, that in turn runs one or more virtual machines called *guests*. Whereas in the dual boot scenario one or the other operating system is running exclusively, with virtualization the host and the guests are running simultaneously. You can even move files and data between them seamlessly. The “disk” in this case is really just a very large Windows file allocated when you created the virtual machine. Let the Linux installation format the entire “disk”.

## Installing Virtualbox

Unless you plan to dedicate a machine to Linux, I strongly suggest installing Linux as a VM under Virtualbox. With that in mind, your first step then is to install and configure Virtualbox itself. Go to [virtualbox.org](http://virtualbox.org) and download the current version. Double click on the downloaded file to install it. Accept all the default configuration options.

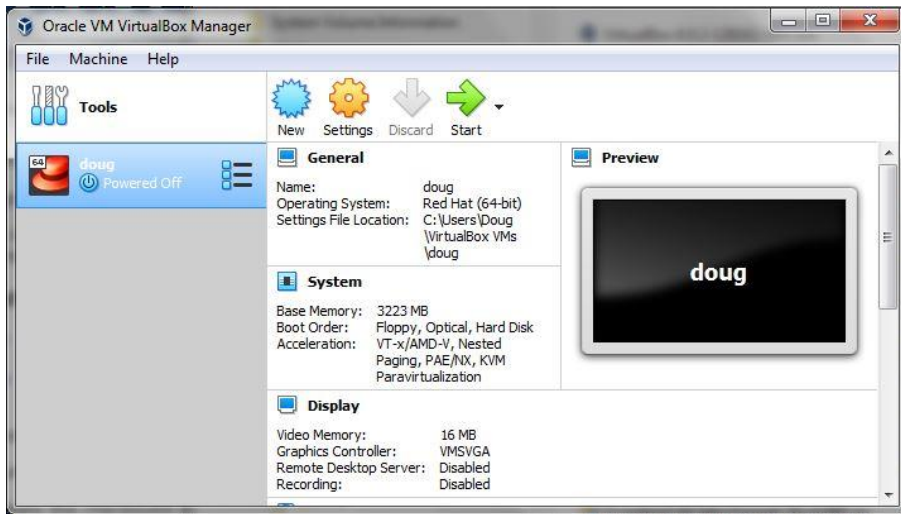


**Figure 1.1: Initial Virtualbox Screen**

Figure 1.1 shows the window when you start up Virtualbox for the first time. Your next task is to create a virtual machine.

1. Click **New**
2. Give the new machine a **Name**. Select **Linux** for the **Type** and select whatever distribution you're installing as the **Version**. Note that there is no version for CentOS. I suggest selecting Red Hat. Click **Next**.

3. Select how much RAM you want for the VM. I suggest half of what's available on the host. Click **Next**.
4. Now you are asked whether or not you want to create a virtual disk drive or use an existing virtual drive. If you've just installed VirtualBox you probably don't have an existing virtual disk, so yes you want to create one now. Click **Create**.
5. Select VDI as the hard disk file type. Click **Next**
6. Now you have a choice of a fixed size virtual disk or a dynamically allocated one. I recommend dynamically allocated. Click **Next**.
7. The default name for the virtual disk file is the same as the VM name. The default size is 12 GB. If you plan to play around with Yocto, you'll need 100 GB. Otherwise 30 GB is probably sufficient. Click **Create**.



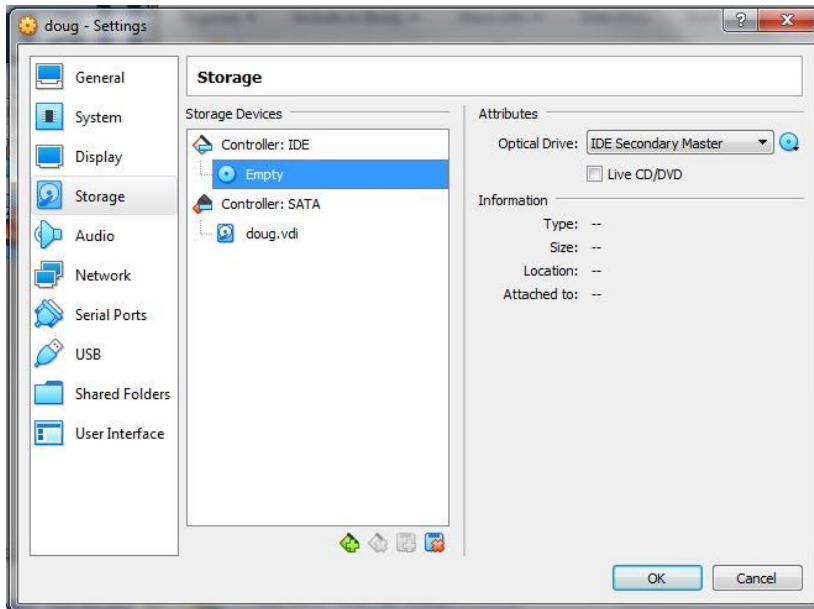
**Figure 1.2: Virtual Machine Screen**

You now have a virtual machine as shown in Figure 1.2. There are several configuration options you should update before starting your new VM.

1. Click **General**. In the **Advanced** tab select **Bidirectional** for both **Shared Clipboard:** and **Drag 'n Drop:**. This allows you to easily move data between the host and the guest.
2. Select **System**. Verify that the **Boot Order:** has the optical drive before the hard disk. If you have a multi-core processor, go to the **Processor** tab and select half the number of available cores for the guest.
3. Select **Storage**. Two controllers should be listed: **IDE** and **SATA**. Your virtual disk file is connected to the **SATA** controller and the **IDE** controller has an icon named **Empty** that looks like an optical disk. Click on **Empty** and you'll see the dialog shown in Figure 1.3. Click on

the disk icon at the far right and you'll have a choice of selecting your physical optical drive or browsing for an ISO image file, which you can "mount" just like a physical disk. Select the device or file with the bootable installation image.

4. Select **Network**. You'll need at least one and maybe two network connections. Even if your host is using a wireless interface, all network interfaces appear to the guest as wired interfaces. Select **Bridged Adapter** for Attached to:. Your primary network interface should appear in the Name: field. Click **Advanced** and be sure **Cable Connected** is checked.
5. If you wish to configure a second network interface, select the **Adapter 2** tab and repeat the above steps.

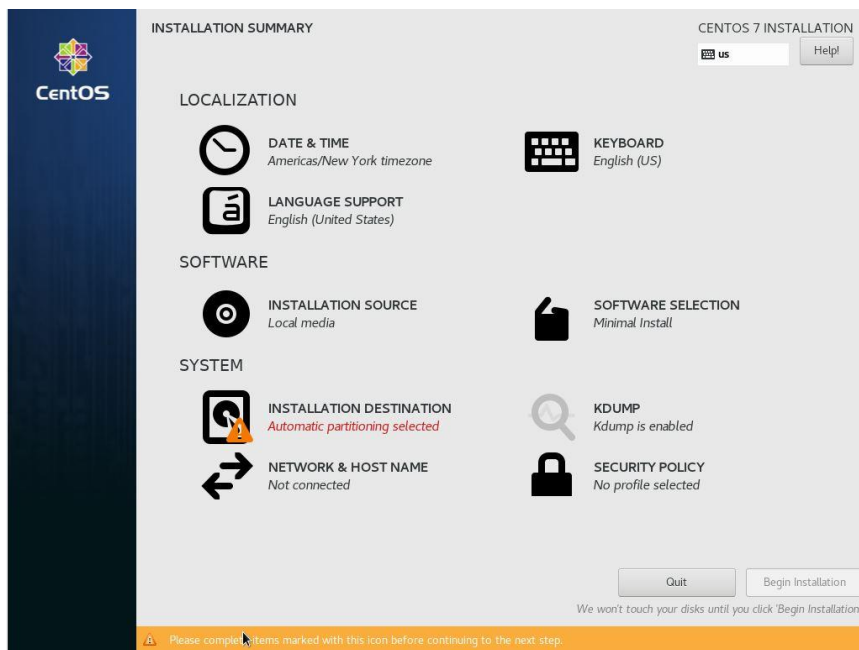


**Figure 1.3: Virtualbox storage dialog**

So go ahead and start your virtual machine.

### *Installing Linux*

I'll briefly describe the process of installing Linux for CentOS, but if you're installing a different distribution, the process will probably be a little different. You'll be asked a few basic questions about things like keyboard and language and then the Installation Summary of Figure 1.4 comes up. There are a few steps you need to complete here before installation actually begins:



**Figure 14: Installation Summary**

**Date and Time:** The time and date themselves are probably already correct. There's a nice graphical interface for selecting your time zone.

**Software Selection:** Let me reveal a personal bias here. Most true Linux hackers do everything from the command shell. Personally, I like GUIs. And while GNOME is perhaps the most popular desktop GUI in the Linux world, I'm partial to KDE. Unless you have a strong preference for GNOME, I recommend selecting KDE Plasma Workspaces as your Base Environment. Alternatively, you could select the GNOME Desktop or the Development and Creative Workstation, which gives you a larger selection of add-ons.

For add-ons you'll need KDE Applications and Development Tools as a minimum. Go ahead and select whatever else you think you might need.

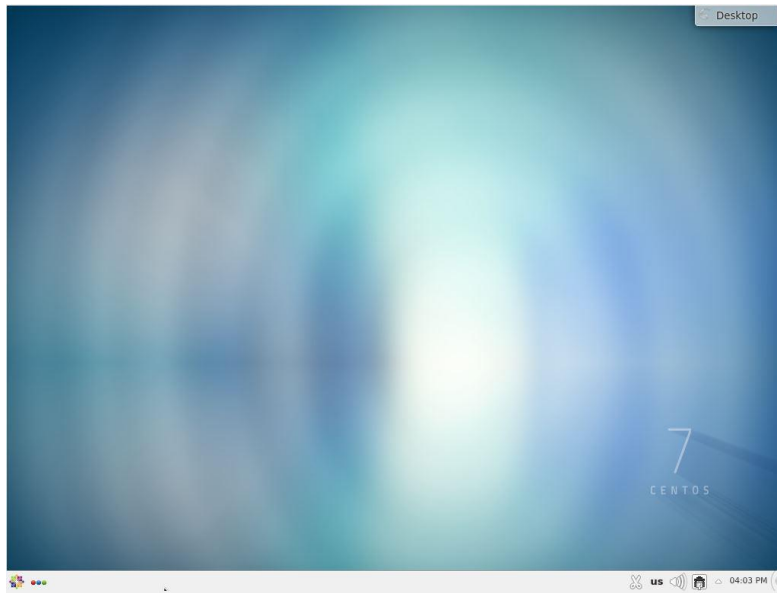
**Disk Partitioning:** The issue here is that automatic partitioning creates a separate /home partition and if you're building a Yocto file system image, you may find that /home fills up even though there's plenty of free space in the root partition. So select I will configure partitioning. Click Done and a "mount point" dialog comes up. Click the "+" icon to add a partition. You'll want to add at least these partitions:

- /boot – 500 MB is more than adequate.
- /swap – The optimal size for a swap partition is subject to some debate. The CentOS folks suggest if you have more than 4 GB of RAM, allocate the size of RAM plus 2 GB. I have 8 GB allocated to my VM, so I allocated 10 GB for the swap on my 100 GB disk.
- / -- The root partition. Everything else.

When finished click **Done** to see a **Summary of Changes**. Click **Accept Changes**.

**Network and Host Name:** This would be a good time to set up your main network connection. As described later on you may want (or need) a second network connection. Select your main network interface, probably a wireless port, if it isn't already selected. Remember that Virtualbox treats all network adapters as "wired". Click on the "slide switch" in the upper right corner to enable the interface. That's it for now. The interface is configured to get its IP address through DHCP. Click **Done**.

Now click **Begin Installation**. This will take a while as there are well over 1,000 packages to install. While you're waiting, you should set a root password and create a normal user. When the installation finishes, click **Reboot**. The installation media will be automatically unmounted and the machine will boot into the new OS. The next step is to accept the license. Then click **Finish Configuration**.



**Figure 1.5: KDE Desktop**

Figure 1.5 shows the basic KDE graphical desktop. The somewhat abstract icon in the lower left corner is the *Application Launcher*, basically the same thing as the Windows Start menu. If you click on that you'll see a collection of cutesie icons. Personally, I don't find this particularly useful. Fortunately, you can change this to what's called the "classic" menu style by right-clicking on the launcher and selecting **Switch to Classic Menu Style**. Now all the various categories of applications are organized into a simple list.

Note that in its present form, the VM will not display full screen, nor will it automatically capture the cursor. Your next step is to install *Virtualbox Guest Additions*. Go to the VM Manager menu at the top of the screen and select **Devices** and then **Insert Guest Additions CD Image**. A pop-up appears in the lower right of the screen showing that an optical disk has been inserted. Click on the icon to the right of the disk name to mount it at `/run/media/<your_user>/VBox_GAs_<version>`.

Open a command shell<sup>4</sup> and `cd` to `/run/media/<your_user>/VBox_GAs_<version>`. Execute the command:

```
sudo sh ./VBoxLinuxAdditions.run
```

`sudo` is one of two commands, the other being `su`, that allow you to become root user. You'll be asked for your user password. `sudo` allows you to temporarily execute privileged commands. `su` by contrast actually makes you root user after asking for the root password.

The script attempts to restart the machine when it's finished. This may not succeed and you may have to reset it from the VMM Machine menu. At this point you'll be able to run the VM as full screen with full cursor integration.

You now have a fully functional Linux machine. If you're not familiar with Linux, this would be a good opportunity to take some time to play around with it. Find the graphical file manager, try out some shell commands. Explore the various configuration options and configure the system to your personal taste.

## Getting Help

Again, if you're relatively new to Linux, you'll probably be looking for some help. The primary help mechanism in Linux is called *man pages* where *man* is short for "manual" as in user's manual. Pretty much every "object" in Linux; shell commands, library functions, system services, and so on, has a man page. From the shell you simply type `man <topic>`. Man pages are not tutorial. They're more reference in nature and tend to be cryptic. Nevertheless they're very useful if you're trying to remember just how a particular command works and what its options are.

There is also a graphical version of man pages, along with other help files, available from the Application Launcher. Finally, and this may come as no surprise, when you're mucking around with Linux, Google is your friend. It is astounding what you can tease out of a Google search. One of my soapbox

---

<sup>4</sup> I'll leave it to you to figure out how to do that.



topics with Linux, and indeed software in general, is error messages. The message rarely tells you what the real problem is, much less a possible solution. But google even a part of an error message and you'll likely come up with a least a dozen different forum posts or how-tos that will shed some light on the problem.

One last tip before we dive into configuring the host workstation. The most useful key on your keyboard is **Tab**. This is the *autocompletion* key for shell commands. Very often you'll encounter very long file names. You can type in the first few letters of the name and then press **Tab**. If the part you typed in is unique, the shell will complete it for you. If not, press **Tab** a second time and it will display the alternatives. Once you become familiar with the **Tab** key you'll be using it constantly.

## Resources

### *Linux distributions*

There are something over 300 Linux distributions, or “distros”, floating around the Internet. Many of them are designed to serve some specific purpose and, in fact, there are a number of embedded distributions. The website <http://iso.linuxquestions.org/> claims to have 1120 versions of 346 “distros” available for download.

In most cases, downloads are available as ISO image files. An ISO file is an exact image of a CD or DVD. Note that burning an ISO image to a CD is not the same as copying a file. Many popular CD burning programs have an option for burning ISO images. If you're running a VM, you can simply “mount” an ISO image as if it were an optical drive.

Some of the more popular distributions include:

Red Hat (redhat.com) – Red Hat is perhaps the best known, and most successful, commercial distribution of Linux. Their focus is on enterprise IT. In fact the product is called Red Hat Enterprise Linux or RHEL for short.

CentOS (centos.org) – an acronym for Community Enterprise Operating System, this is a free, open source clone of RHEL. It is the exact same code base with proprietary branding and artwork removed.

Fedora (fedoraproject.org) -- the Fedora project is the community-supported successor to the free Red Hat Linux distributions that ended with Red Hat 9. It also serves as a “test bed” for Red Hat Enterprise Linux. Fedora is definitely “bleeding edge” with new releases coming out about every six months.

Debian (debian.org) -- the Debian project is known for its adherence to the Unix and free software philosophies, and for its abundance of options — the current release includes over fifteen thousand software packages for eleven computer architectures.

Ubuntu ([ubuntu.com](http://ubuntu.com)) -- Ubuntu was developed in South Africa based on Debian GNU/Linux and has become quite popular in recent years. It emphasizes usability, regular releases, ease of installation, and freedom from legal restrictions. The name comes from the Zulu and Xhosa concept of *ubuntu*, which means "humanity towards others".

Suse ([opensuse.org](http://opensuse.org)) -- the SUSE distribution is currently owned by Novell and is available in both an open source version and a commercial version. The link here is to openSUSE.

[virtualbox.org](http://virtualbox.org) is the website for Oracle's VirtualBox VMM

[vmware.com](http://vmware.com) is the site for VMware's VMM

Sobell, Mark G., *A Practical Guide to Linux*. This book has been my bible and constant companion since I started climbing that steep Linux learning curve. It's an excellent beginner's guide to Linux and Unix-like systems in general, although having been published in 1997 it is getting a bit dated and hard to find. It has been superseded by...

Sobell, Mark G., *A Practical Guide to Linux Commands, Editors, and Shell Programming*, 4<sup>rd</sup> Edition, Addison-Wesley, 2017.

## 2. The Host Development Environment

### Installing the kit software

Once you have your workstation up and running, it's time to install the software that comes with the kit.

1. Insert and mount the CD ROM, normally at `/run/media/<your_user>/EmbeddedLinux` for CentOS distributions.
2. `cd` to your home directory
3. Use the `sudo` or `su` command to become root user. Debian and Ubuntu users don't have `su`. If you are unfamiliar with these commands, see the corresponding `man` page.
4. Execute `/run/media/<your_user>/EmbeddedLinux/install_tools.sh`

The script assumes the CD is mounted at `/run/media/<your_user>/EmbeddedLinux`. If your mount point is different, execute:

```
/<your mount point>/install_tools.sh <your mount point>
```

Some systems, Ubuntu it seems, arbitrarily reset the execute permission bits on files on removable media. The upshot is that the scripts won't execute. You can get around this by executing:

```
bash /run/media/<your_user>/EmbeddedLinux/install_tools.sh.
```

The `install_tools.sh` script installs the ARM cross-compilation tool chain along with a couple of other tools and a root file system for the target board. You'll be asked what directory you want to install the tool in. I suggest `/opt/arm`. The script also changes the permissions on some directories so that you as a normal user can write to them. When `install_tools.sh` finishes, it will prompt you to exit the root user shell and execute `/run/media/<your_user>/EmbeddedLinux/install.sh`.

#### Important Note

Be sure you run the `install_tools.sh` script as root user and you run `install.sh` as your normal user. If you run `install.sh` as root, you will either have to change the ownership of all the files it installs or you will have to be root user to do anything with the kit.

If you should encounter problems with the installation, you can execute the `uninstall` script `/run/media/<your_user>/EmbeddedLinux/uninstall.sh` to remove everything that was installed by the installation scripts.

When the installation is finished, you'll find five new subdirectories under your home directory:

- `busybox-1.29.2` – source distribution for the BusyBox utility that is described in detail in chapter 13
- `linux-rpi-4.19.y` – Linux kernel source tree

- `boot/` -- This is a duplicate of the boot partition on the micro SD card. It may be necessary to re-flash one or more of these if something goes drastically wrong.
- 
- `target_fs/` -- this is the target board's root file system that will be mounted over the network. `target_fs/home/src` contains sample code in several subdirectories:
  - `include/` -- some header files needed for access to the target hardware
  - A couple of libraries and utilities for accessing hardware
  - Project directories – one for each project
- `u-boot` – source tree for the open source bootloader project, u-boot

Eclipse is installed under `/usr/local`.

## The Terminal Emulator, minicom

`minicom` is a fairly simple Linux application that emulates a dumb RS-232 terminal through a serial port. This is what we'll use to communicate with the target board.

Recent Linux distributions tend not to install `minicom` by default. It's easy enough to install. As root user (either `su` or `sudo`) execute:

<code>yum install minicom</code>	in CentOS or Fedora
<code>apt-get install minicom</code>	in Ubuntu or Debian

Both of these commands do pretty much the same thing. They go out on the Internet to find the specified package, resolve any dependencies, download the packages, and install them. Both of these programs are essentially wrappers around RPM, the Red Hat Package Manager.

Once `minicom` is installed, there are a number of configuration options that we need to change to facilitate communication with the target.

In a shell window as root user, enter the command `minicom -s`. If you're running `minicom` for the first time you may see the following warning message:

**WARNING: Configuration file not found. Using defaults**

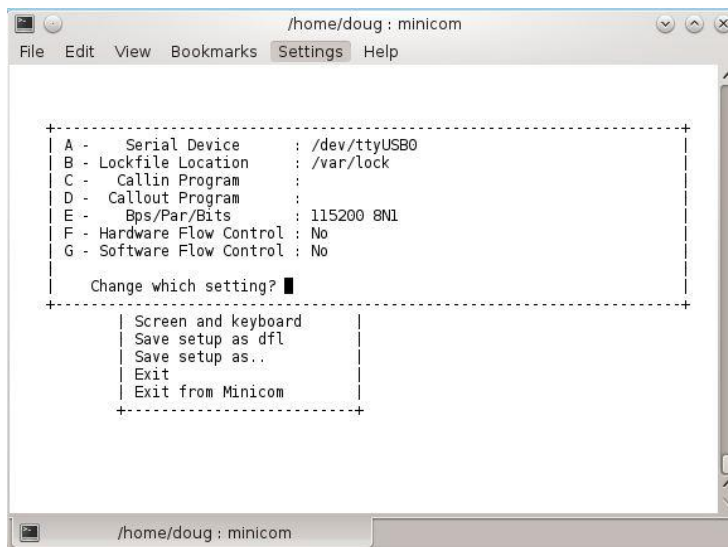
You will be presented with a configuration menu. Select **Serial port setup**. By default, `minicom` communicates through the modem device, `/dev/modem`. We need to change that to talk directly to one of the PC's serial ports. Type "A" and replace the word "modem" with either "ttyS0" or "ttyS1", where `ttyS0` represents serial port COM1 and `ttyS1` represents COM2. However, if your host only has USB ports and you're using a USB to serial converter, the correct device is most likely "ttyUSB0."

Type “E” followed by “I” to select 115200 baud. Make sure both hardware and software flow control are set to “no”. These are toggles. Typing “F” or “G” will toggle the corresponding value between “yes” and “no”. Figure 2.1 shows the final serial port configuration.

Type <Enter> to exit **Serial port setup** and then select **Screen and keyboard**. Type “B” once to change the Backspace key from BS to DEL. Check that the Line Wrap option is set to “yes”. If not, type “R” to toggle it.

Type <Enter> to exit **Screen and keyboard**. Select **Save setup as dfl** (default). Then select **Exit**.

Execute `ls -l /dev/ttyS0` (or `/dev/ttyUSB0` if you’re using a USB-to-serial converter). Note that only the owner and group have access to the device. You can gain access by becoming a member of the dialout group. As root user, edit the file `/etc/group`. Find the line that begins “dialout:” and add “.:<your\_user\_name>” at the end of the line.



**Figure 2.1: minicom serial port setup**

## Networking

### *Network Interface*

Very likely your workstation is connected to the Internet, probably through a wireless interface, and probably gets its network address via DHCP (Dynamic Host Configuration Protocol). You'll need to maintain that Internet connection because you'll be installing additional software packages later on. At the same time, you need to connect the R Pi target board to the network, but in this case, to keep things simple, we'll give it a fixed IP address.

The target board's IP address will need to be on the same network segment as your workstation. So for example, if your workstation has an address such as 192.168.1.46, it is node 46 on network segment 192.168.1. Your target will then need a node number on that same network segment that's outside the range allocated by the DHCP server on your network segment. We'll discuss changing the target's network address in the next chapter.

Sometimes it's inconvenient, impractical, or not even allowed, to attach the target board to the "house" network. In this case a second network interface, probably an Ethernet port, is the solution. The Ethernet port then connects directly to the target and gets a fixed IP address since it doesn't have access to a DHCP server.

If you're running Virtualbox, start by configuring your Ethernet interface as the second network adapter in the VM manager as described in the previous chapter. In a shell window, execute `ifconfig` to see the current state of all three network interfaces. The third interface is the local loopback, `lo`. One of the other two will be configured with an IPv4 network address. This is your Internet interface. The other interface will not have a network address because it hasn't been enabled yet.

From the Application launcher, select **Settings -> System Settings**. Scroll down to **Network and Connectivity** and select **Network Settings**. Select **Network Connections** and click the **Wired** tab. The Internet interface will be listed.

1. Click **Add...** and select **Wired** to bring up the dialog in Figure 2.2.
2. Change the **Connection Name**: to match what `ifconfig` reported. Note that your connection names will probably be different than mine<sup>5</sup>.
3. Change **Method** to **Manual** and enter an IPv4 address. The value shown is compatible with the default setting of the target. The network segment (the first three numbers) must differ from the network segment of your Internet interface. The **Gateway**: doesn't matter because this interface is not connected to the Internet.
4. Check the **Connect automatically** and **System connection** boxes.
5. Select the **Ethernet** tab. **Restrict to interface**: the interface you're configuring.

---

<sup>5</sup> Historically Ethernet interfaces got names like `eth0`, `eth1`, and so on. That has recently changed to a scheme that names interfaces according to how they're connected to the system.

Upon clicking OK, you'll probably see some bizarre message about the KDE Wallet system. Just click Cancel.



**Figure 2.2: Add Network Connection Dialog**

### *Network File System (NFS)*

We'll use NFS (Network File System) to mount the target board's root file system<sup>6</sup>. That means we have to "export" one or more directories on the workstation that the target can mount on its file system. Exported directories are specified in the file `/etc/exports`. The `install_tools` script put a template `exports` file in `/etc` with a line like this:

```
/home/<your_home_name> *(rw,no_root_squash,sync,no_subtree_check)
```

<sup>6</sup> Ubuntu distros tend to not have NFS installed by default. Execute `apt-get install NFS-kernel-server` to install it.

As root user, open `exports` with an editor and replace `<your_home_name>` with the name of your home directory. This makes your home directory visible on the network where other nodes can mount parts of it to their local file system. Note that you must edit this file as root. There's a `man` page for the `exports` file if you want to understand the various options.

Now you'll want to check on the status of NFS and either start it if it's not running, or restart it to force it to reread the `exports` file. Execute:

```
systemctl status nfs-server
```

The output is rather extensive but the only line that counts starts with the word `Active:` followed by either `active (exited)` or `inactive (dead)`. If it's the former, execute:

```
systemctl restart nfs-server
```

If the latter, then:

```
systemctl start nfs-server
```

To have NFS automatically start every time the system boots:

```
systemctl enable nfs-server
```

## The Cross Development Environment

The cross development toolchain installed in `/opt/arm` has all the tools needed to build and test programs for our target board. Fundamentally, it is the GCC toolchain built to generate ARM code.

To use this environment, we need to define a number of environment variables, like for example extending the `PATH` so that the shell can find the cross compiler. A number of variables are involved and some are very long, so a script is provided in `/opt/arm/environment-setup-cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi` to do it for you. Have a look at that script.

Use the `source` command to execute the script, as in:

```
source /opt/arm/environment-setup-cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi
```

But don't forget tab autocompletion. Normally when you execute a shell command, the shell spawns a child shell process to run the command and that process exits when the command finishes. That wouldn't do us much good in this case since the new environment variables would be defined in the child shell that goes away. `source` says execute this command in the current shell.

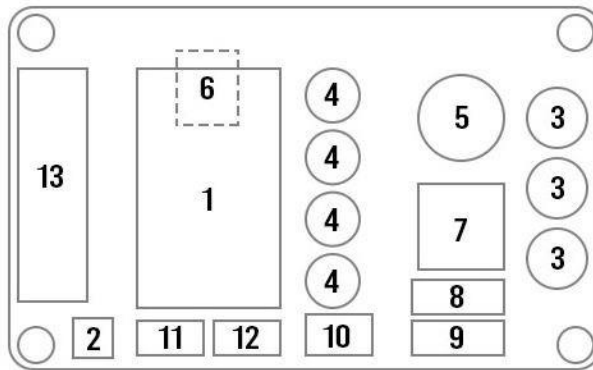
If you're familiar with the file `.bash_profile`, you might ask, why not just put all these variables in there and be done with it? The reason is that there are times when we want to run things in the native environment and so it's best to do this on an individual shell basis.

That's it for configuring the workstation. Let's move on to the target board.



### 3. The target

If you haven't done so already, unpack the target board and power supply. The board is actually two pieces plugged together. The bottom, or base, board is the Raspberry Pi 3 itself. Plugged into the R Pi's 40-pin I/O header is the Matrix CK (Compact Kit), a collection of simple I/O devices that we'll exercise throughout the remainder of the manual. Figure 3.1 identifies the components of the Matrix CK. There's a schematic of the Matrix CK in the docs/ directory of the kit CD.



**Figure 3.1: Matrix CK**

1. 80x160 TFT display
2. Compass
3. Pushbuttons
4. LEDs
5. Buzzer
6. Analog to digital converter
7. Potentiometer connected to channel 0
8. 4-pin header for I2C bus
9. 4-pin header for UART
10. Double row power header: 3 pins 3V3, 3 pins GND
11. Socket for temperature transducer
12. Socket for IR receiver
13. 3x10 pin header
  - 3 analog in
  - 7 digital I/O: 2 can be configured for PWM, 4 for SPI

Pinouts for all the connectors on both the Raspberry Pi and the Matrix CK are in Appendix A.

**Warning:** DO NOT connect the serial port to an RS232 port as this will damage the R Pi. The serial port is 3.3 V.

The serial debug port appears on the 4-pin header labelled 9 in Figure 3.1 above. If you purchased a serial to USB converter from Intellimetrix, the pinout is as follows:

Black	GND	1
White	RX	3
Green	TX	4
Red	5V	NC

It's not necessary to connect the 5V pin.

## The boot Partition

Remove the micro SD card from the target and insert it in a card reader attached to the workstation. The micro SD card has two partitions: **boot** formatted as FAT32 and **rootfs** formatted as ext3. Mount the **boot** partition. In CentOS the default mount point for removable media is `/run/media/<user_name>/<volume_name>`.

Have a look at the **boot** partition with the file manager. There are a number of files here, most of which have to do with the boot process that we'll look at in chapter 14. There are several *device tree blob* (.dtb) files that are hierarchical representations of the hardware. We won't get into device trees in the kit, but there's plenty of stuff online about them.

There are two kernel image files: **kernel.img** and **kernel7.img**. The boot loader finds the kernel image with the highest number, in this case 7, and boots it by default. But our interest at the moment is in two text files; **config.txt** and **cmdline.txt**. Open **config.txt** with an editor. Scroll through the file to get a feel for what kinds of parameters you can set for the R Pi. Documentation at [raspberrypi.org](http://raspberrypi.org) gives a complete description of the parameters.

Note the line at the very bottom of the file:

```
enable_uart=1
```

This enables the serial port so that you can talk to the board from the workstation using the **minicom** terminal emulator without the need of a monitor, keyboard, or mouse.

## Changing the Target's Network Address

If you need to change the target's IP address, open the file **cmdline.txt** in an editor. This file consists of one very long line of text that is passed to the kernel when it boots. Among other things, it tells the kernel where to find its root file system and in this case it specifies the IP address at the very end of the line. Change the value of **ip=** as appropriate.

There are a couple of other entries in this line that deserve mention. `root=/dev/nfs` tells the kernel that it will be mounting its root file system over the Network File System. `nfsroot=` tells the kernel where its root file system is located. If you're changing the target's IP address that probably means you also changed the address of the host workstation. So you'll also need to change the value of `nfsroot=` accordingly.

## Booting the Target

Unmount the SD card from the workstation and insert it back in the target board. Connect the network port using a standard Ethernet cable.

Start up `minicom` in a terminal window. Now plug the power into the mini USB port<sup>7</sup>. The target's bootloader boots the Linux kernel from the micro SD card and mounts the root file system from the host workstation over NFS. The login user name is "root" and there is no password. Try a few simple Linux shell commands to prove that it really is working.

Next, try opening an SSH connection to the target. In another shell window, execute:

```
ssh root@192.168.15.50
```

Of course, if you changed the target's IP address, substitute that value for the one shown here. The first time you `ssh` to the target the program will tell you that the authenticity of the host can't be established and asking if you want to continue connecting, to which you must reply by typing the word "yes". `ssh` then establishes a file with appropriate authentication data so that subsequent invocations will simply proceed.

## What can go wrong?

It's not unusual to encounter difficulties when bringing up an embedded target board such as the ELLK. The most common problems fall into two broad categories. The first is the serial port. Make sure the baud rate is set correctly. This is generally not a problem because 115 kbaud seems to be the default for `minicom`. A more common problem is not turning off hardware flow control.

Common networking problems include having Security Enhanced Linux (SELinux) enabled and/or the firewall turned on. This will prevent the target from NFS mounting its root file system. As a starting point, disable SELinux and turn off the firewall. Later on you can configure either of these features to allow NFS mounting but still provide some level of protection.

SELinux is managed by the file `/etc/selinux/config`. Edit this file as root user. About midway down is a line that reads `SELINUX=enforcing`. Change `enforcing` to `disabled`.

---

<sup>7</sup> Given that you'll probably be powering the board up and down frequently, it would be prudent to insert a switch in the power supply cable to avoid damage to the mini USB connector.

The firewall is managed by `systemctl` commands. Execute the following:

```
sudo systemctl stop firewalld  
sudo systemctl disable firewalld
```

The disable command prevents the firewall from starting on subsequent boots.

Make sure the IP address is correct.

## **Resources**

[raspberrypi.org](http://raspberrypi.org) – This is the official site for Raspberry Pi development and use. There is lots of documentation, projects, and a vigorous user community.

## 4. Eclipse Integrated Development Environment

### Introduction

Integrated development environments (IDE) are a great tool for improving programmer productivity. Desktop developers have been using them for years. Perhaps the most common example is Microsoft's Visual Studio environment. In the past, a number of embedded tool vendors have built their own proprietary IDEs.

In today's Open Source world, the IDE of choice is Eclipse, also known as the Eclipse Platform and sometimes just "the Platform." Several leading embedded Linux vendors such as Monta Vista, TimeSys, LinuxWorks, and Wind River Systems have embraced Eclipse as the basis for their own IDE development. All of these vendors actively contribute to Eclipse development.

"Eclipse is a kind of universal tool platform—an open, extensible IDE for anything and nothing in particular. It provides a feature-rich development environment that allows the developer to efficiently create tools that integrate seamlessly into the Eclipse platform," according to the platform's own on-line overview. Technically, Eclipse itself is not an IDE, but is rather an *open platform* for developing IDEs and *rich client* applications.

### Getting Started

#### Java Runtime

Since Eclipse is written primarily in Java, you need a Java runtime environment (JRE) to run it. Most Linux distributions include a suitable JRE by default. If your system doesn't have one, or the version on your system happens to be incompatible with Eclipse, you can get a free download of the latest version by going to [java.com](http://java.com).

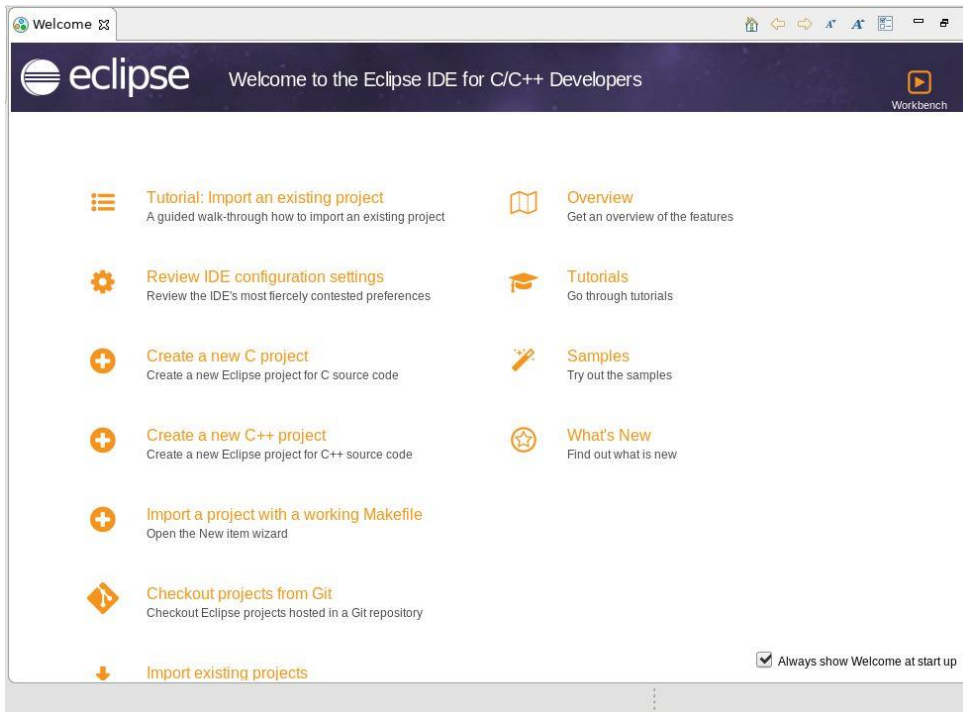
Click **Free Java Download**. The site will recognize that you are running Linux and offer the appropriate Linux downloads. I prefer the self-extracting binary (not the RPM). Follow these instructions to complete the installation:

1. Move the file to the directory in which you want to install it. `/usr/local` is a good choice because you have write access to it. `cd` to that directory.
2. Change the permission of the file to be executable:  
`chmod a+x jre-6u<version>-linux-i586.bin`
3. Run the file:  
`./jre-6u<version>-linux-i586.bin`  
Read and accept the license agreement.
4. Java is installed in `jre1.6.0_<version>` under `/usr/local` (or wherever you chose to put it).
5. Change the link `java` in `/usr/bin` to point to `/usr/local/jre1.6.0_<version>/bin/java`.

For many years the Eclipse Foundation did a major annual release of Eclipse in the June timeframe. This strategy continued through the release of Photon in June of 2018. At that point the Foundation moved to a quarterly release polity where each release is simply numbered with the year and month of release.

## Starting Eclipse

Eclipse is installed in `/usr/local/eclipse`. The executable is `/usr/local/eclipse/eclipse`. To start it up, execute `/usr/local/eclipse/eclipse &` from a shell window where you have set the cross development environment. The `&` means execute the command in the background. To make this a little less tedious, you can create a script named `eclipse` in `/usr/bin`, or any other directory in your path, with just the above command line in it. Now you can start Eclipse just by typing `eclipse`.



**Figure 4.1: Eclipse welcome screen**

When Eclipse first starts up, it asks you to select a workspace. The default is the directory `eclipse-workspace/` under your home directory. Every project you create gets a sub-directory under `eclipse-`

workspace/ unless you choose to put it somewhere else. Following the workspace dialog, you'll see the Welcome screen shown in Figure 4.1. The Welcome screen will appear each time you start Eclipse unless you uncheck the **Always show Welcome at start up** box. You can always access this window from the first item on the **Help** menu, **Welcome**.

The **Overview** icon leads you to a very good, very thorough introduction to using Eclipse. If you're new to the Eclipse environment, I strongly suggest you go through it. There's no point in duplicating it here. Go ahead, I'll wait.

OK, now you have a pretty good understanding of generic Eclipse operation. It's time to get a little more specific and apply what we've learned to C software development for embedded systems.

### *Additional Plug-ins*

There are some additional plug-ins we need primarily to support remote debugging and the Yocto project. Proceed as follows:

1. Click **Help** and then select **Install New Software...**
2. In the **Work with:** dropdown select **CDT** –  
<http://download.eclipse.org/tools/cdt/releases/9.6>
3. Expand **CDT Optional Features** and select:
  - C/C++ Debugger Services Framework (DSF) Examples
  - C/C++ GCC Cross Compiler Support Developer Resources
  - C/C++ Remote Launch Developer Resources
4. Click **Next >**. Review the items to be installed and click **Next >** again.
5. Review and accept the license, then click **Finish**.
6. When the updates are complete, you are asked if you want to restart Eclipse. Not now because there is more software we need from a different repository.
7. Click **Help** and select **Install New Software...** again.
8. This time select **2018-12** – <http://download.eclipse.org/releases/2018-12>
9. Expand **Linux Tools** and select **C/C++ Remote (over TCF/TE) Run Debug Launcher**
10. Expand **Mobile and Device Development** and select:
  - C/C++ Remote (over TCF/TE) Run Debug Launcher
  - Remote System Explorer User Actions
  - TCF C/C++ Debugger
  - TCF Target Explorer
11. Repeat steps 4 to 6 and again we don't want to restart Eclipse yet.
12. Once more click **Help** and select **Install New Software...**
13. This time click **Add...** and enter <http://downloads.yoctoproject.org/releases/eclipse-plugin/2.6/oxygen> for the **Location:** and give it a sensible name. Click **Add**.

14. Select both items in the menu.
15. Repeat steps 4 to 6. This time we do want to restart Eclipse.

## The C Development Environment (CDT)

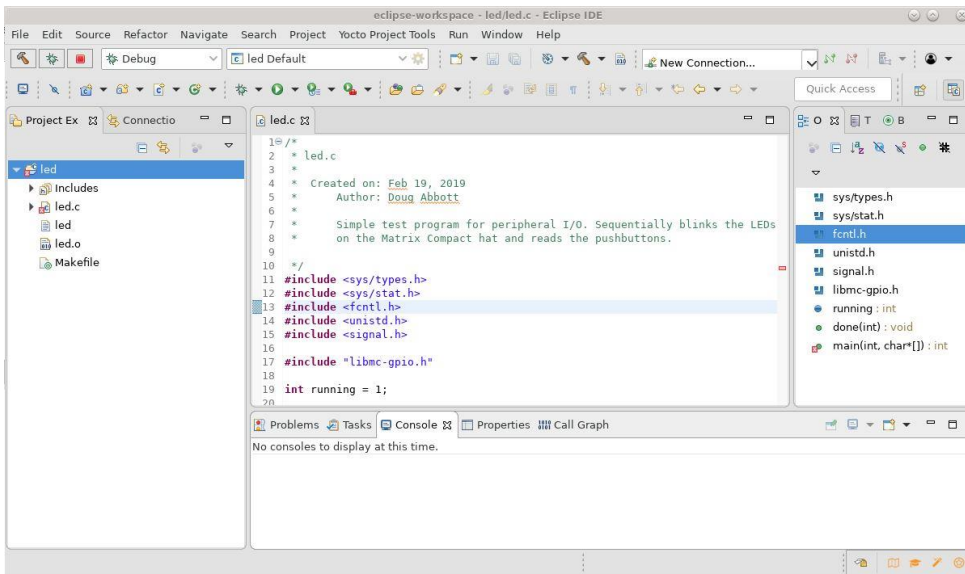
Among the ongoing projects at Eclipse.org is the tools project. A subproject of that is the C/C++ Development Tools (CDT) project whose aim is to build a fully functional C and C++ integrated development environment (IDE) for the Eclipse platform. CDT under Linux is based on GNU tools and includes:

- C/C++ Editor with syntax coloring
- C/C++ Debugger using GDB
- C/C++ Launcher for external applications
- Parser
- Search Engine
- Content Assist Provider
- Makefile generator

Figure 4.2 show the basic Eclipse workbench. It consists of several *views* including:

- Project Explorer--shows the files in the user's workspace
- Text Editor—shows the contents of a file
- Tasks—a list of “to dos”
- Outline—of the file being edited. The contents of the outline view are content-specific





**Figure 4.2: Eclipse workbench**

### *pi-lib*

Before you can run any of the sample programs, you'll need to install a library in the target file system. The library is called `libmchw` and it contains several modules that exercise the hardware on the Matrix CK board. The source code for the library is found in `target_fs/home/src/pi-lib`. `cd` to that directory and execute `make`. This will create two files, `libmchw.a`, a static library, and `libmchw.so`, a shared library.

Copy the two `libmchw` files to `target_fs/lib`. Next we have to update the library cache in the target file system so the run time linker can find the new libraries. The command `ldconfig` does that. On the target, execute `ldconfig`.

### *Creating a Project*

There are two ways to create a project in Eclipse, a *standard* project, also known as a *Makefile* project, or a *managed* project, also known simply as an *Executable*. A managed project automatically creates a project directory, a template C source file, and a makefile in the default workspace.

Typically though, you've already got a number of projects put together in the "conventional" way that you'd like to bring into the Eclipse CDT environment. All of the E.L.L.K. sample projects already

have makefiles. The role of a Makefile project then is to bring an existing makefile into the CDT environment.

For our first project, we'll use the program in `src/led` to illustrate some basic concepts and verify that CDT is functioning properly and that we can execute a newly built program on the target. The `led` program will be covered in more detail in the next chapter.

In Eclipse, select **File -> New -> Makefile Project with Existing Code**. That brings up the dialog shown in Figure 4.3. Enter the project name, "`led`", and browse to `home > target_fs > home > src > led`. Click **Finish**. The project wizard creates the project, which then appears in the Project Explorer window on the left.

Expand the `led` project entry in the Project Explorer. The project itself consists of three files: `Makefile`, `led.c`, and the executable `led`. It depends in turn on some header files in the `includes/` directory and links to the `libmchw.so` library in the `target_fs/lib/` directory. Sources and the Makefile for `libmchw` are in the `src/pi-lib/` directory.

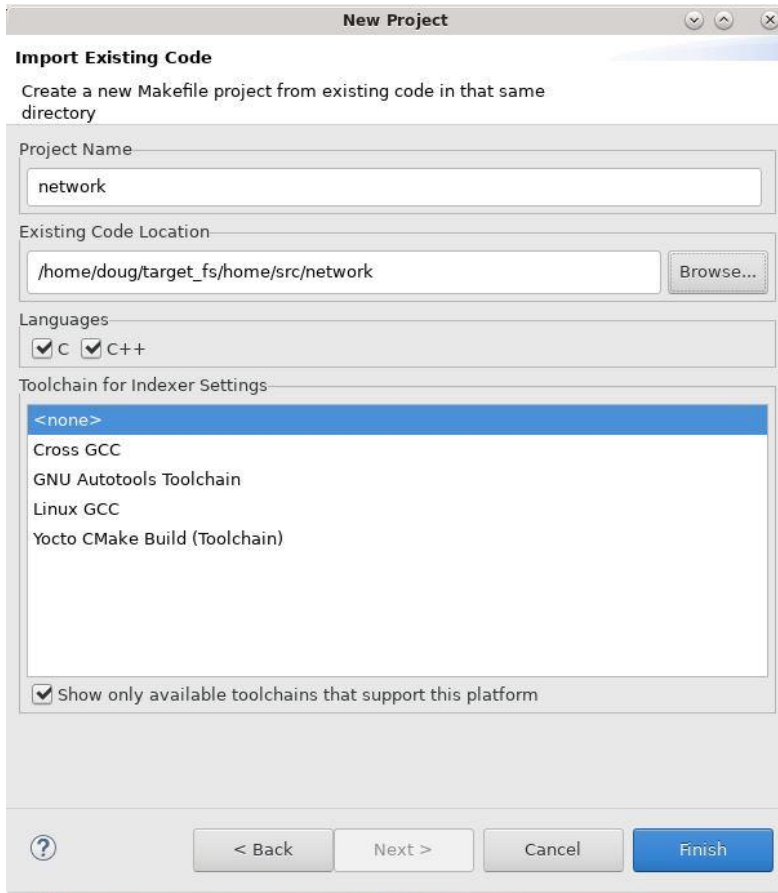
To open a file in the Eclipse editor, just double-click in the Project Explorer. So double-click `led.c`. Note first of all the "?" symbols next to all of the `#include` directives. If you mouse over one of these, it says "Unresolved inclusion". This means that Eclipse couldn't find the header files and thus can't itself resolve the symbols in the header files. The project will in fact build even though Eclipse reports errors for all the symbols defined by the header files. Also, you can't directly open header files from the Outline view.

This appears to be an artifact of makefile projects. Here's how to fix it:

1. Right-click the project entry in the Project Explorer view and select **Properties** down at the bottom of the context menu.
2. Expand the **C/C++ General** entry in the Properties dialog and select **Paths and Symbols**.
3. Make sure the **Includes** tab is selected. Click **Add...** and enter `/opt/arm/sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi/usr/include`. Check **Add to all configurations** and **Add to all languages**.
4. Click **Add...** again and enter `../includes`. This picks up the local header files that primarily define the `libmchw` functions. Check **Add to all configurations** and **Add to all languages**.
5. Click **OK**. Click **Apply**. You'll be asked if you'd like to rebuild the index. Yes, you would.
6. Click **OK** one more time.

The "?" symbols will magically disappear. Note that this is a project-level setting. There does not appear to be a global setting and that probably makes sense. As we'll see later on, when we create a new project, there's a way to *import* the settings we just changed.

The project is also automatically built as part of the creation process. OK, we've built the project, now how do we run it?

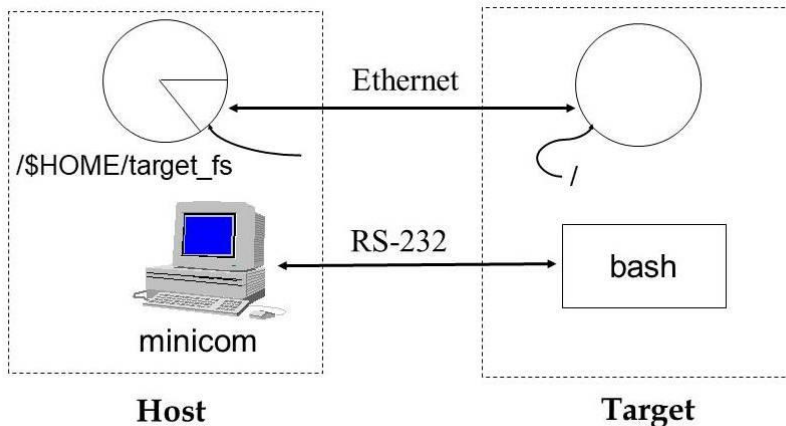


**Figure 4.3: New project wizard**

### *The target execution environment*

Before proceeding, let's step back and review our setup. The target board has a kernel booted from SD card and a root file system mounted over the network from the host workstation. `stdin`, `stdout` and `stderr` on the target are connected to `ttyS0`, which in turn is physically connected to `ttyS0` or `ttyUSB0` on the host. We communicate with the shell running on the target through the terminal emulation program `minicom`. Figure 4.4 illustrates this setup.

One of the directories on the target file system is `/home`. Under `/home/src/` are several subdirectories, each representing a project described later on in this manual.



**Figure 4.4: The host and the target**

Fire up the target and try it out. When the boot completes, execute `ls /home` in the minicom or ssh window. You'll see the same files and subdirectories that are in `$HOME/target_fs/home` on your host. To execute the led program, do:

```
cd /home/src/led
./led
```

The program prints a greeting message and then sequentially flashes the four LEDs. Try pressing the pushbuttons and see what happens.

## Remote debugging with GDB

If you've done any Linux programming at all, you're no doubt familiar with GDB, the Gnu DeBugger, and probably with a graphical front end for GDB such as DDD.

In a typical desktop environment, the target program runs on the same machine as the debugger. But in our embedded environment GDB runs on the host and the program being debugged runs on the target. GDB implements a serial protocol that allows it to work with remote targets either over an RS-232 link or Ethernet.

A program on the target, `gdbserver`, runs the program under test and communicates with GDB running on the workstation. The operation of `gdbserver` is completely transparent to you as the user.

## Establishing an SSH Connection to Eclipse

Before we can debug on the target there are two things we have to do: establish an SSH connection with Eclipse and set up a *debug launch configuration*. If the **Connections** view is not visible next to the **Project Explorer** view, select **Window > Show View > Other...**, expand **Connections** and select **Connections**. Initially the only connection is Local, which isn't particularly interesting. To create a new remote connection, proceed as follows:

1. Click the **New Connection** icon and select **SSH** as the **Connection Type**. Click **Next**.
2. Fill in the **Host Information**. The **Host**: is the target's IP address and the **User**: is "root". Select **Password based authentication**, but of course root has no password. Figure 4.5 shows the completed dialog. Click **Finish**.
3. You may be asked to enter a secure storage password. I just used "raspberry". Then you're asked if you'd like to enter a "password hint". Sure, why not? Click **Finish**.

The new connection now shows up in the **Connections** view. Right click on it and select **Open Command Shell**. The first time you try to open a shell you may get an error about "Local: Command not found". Click **Open Command Shell** again and a **Secure Shell** on the target shows up in the **Console** view.

**New Connection**  
Specify properties of a new connection

Connection name:

Host information

Host:

User:

☐ Public key based authentication      Keys are set at [Network Connections, SSH2](#)

Passphrase:

☒ Password based authentication

Password:

▶ Advanced

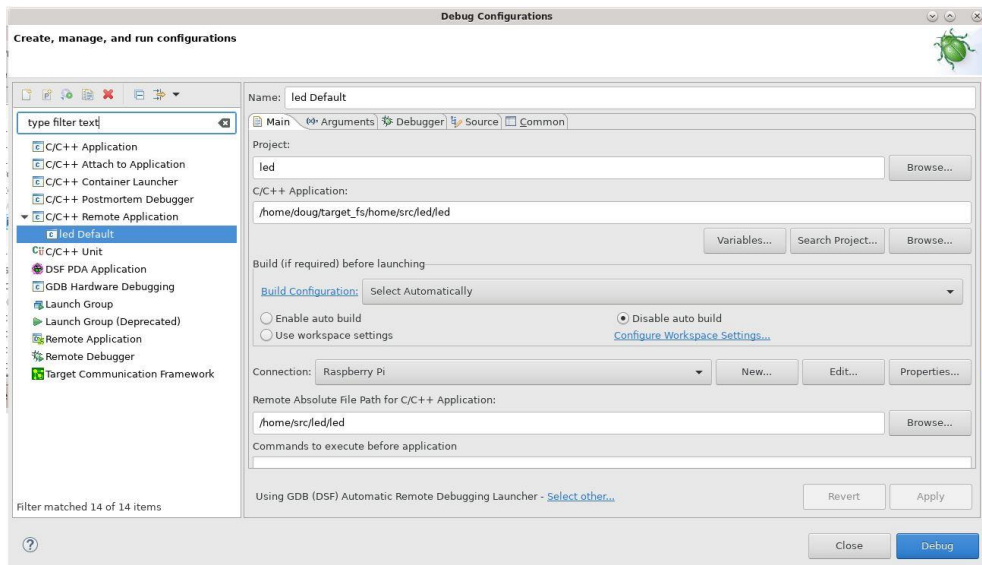
?      < Back      Next >      Cancel      Finish

**Figure 4.5: New Connection Dialog**

## Debugging with CDT

CDT integrates very nicely with GDB. Having established a connection to the target, we can now set up a debug launch configuration.

1. In the Project Explorer view, select the led project.
2. Click Run -> Debug Configurations.
3. In the Debug Configurations dialog box, select C/C++ Remote Application, then click the New icon.
4. In the Main tab the Name: and Project: fields are already filled (I usually delete the “Default” in the name). Click Browse to find the C/C++ Application by navigating to target\_fs/home/src/led and selecting led.
5. In the Connection: drop-down select the connection you just created.
6. In the Build (if required) before launching section select Disable auto build. Auto build can sometimes get in the way.
7. The Remote Absolute File Path for C/C++ Application will initially be set the same as the C/C++ Application. Remove the leading /home/<user\_name>/target\_fs. This then is the absolute path on the target to the executable.
8. Click the Debugger tab
9. Make sure Stop on startup at main is checked.
10. In the GDB debugger: window, enter “arm-pokylinux-gnueabi-gdb”.
11. Click Apply, then Debug.



**Figure 4.6: Setting up a debug configuration**

Eclipse asks if you want to switch to the Debug perspective. Yes, you do. I usually check the box that says don't show this message again.

The first time you click **Debug** after starting Eclipse you may see an error something about “locale: command not found”. This seems to be innocuous. Click **OK**, then click the “bug” icon in the upper left. It should work from then on.

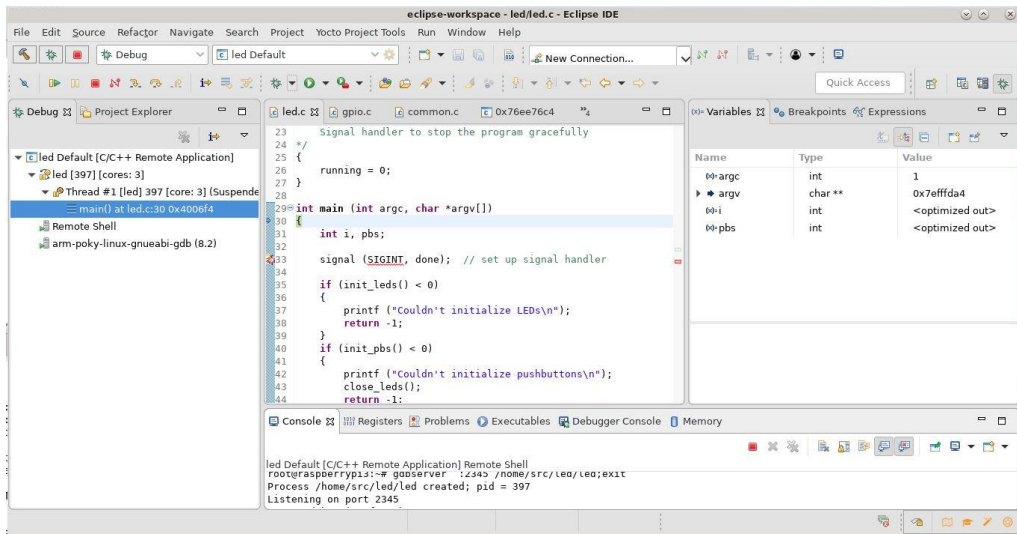
Figure 4.7 shows the initial Debug perspective. Down at the bottom is a Console window where terminal I/O takes place. The Editor window shows `led.c` with the first line of `main()` highlighted. That's because we said **Stop at main()** in our debug configuration setup. The **Outline** tab on the right lists all of the header files included by `led.c` and all of the global and external symbols declared in the file including variables, constants, and functions.

The left window is essentially a stack trace view of your program and is the Navigator view for the Debug perspective. The menu bar has a number of buttons for execution control, such as single step, halt, resume, and so on. The right-hand window has several tabs that provide real-time information on the state of the system, including variables, breakpoints, and registers. And in the center is the Editor with `led.c` open.

Don't worry about the details of the program at this point. We'll cover that in the next chapter.

Let's set a breakpoint on the line that says `printf()`. Right click in the blue column on the left side of the editor window at that line. This is known as the “marker bar”. Select **Toggle breakpoint** from the context menu. An icon appears in the column to indicate that a breakpoint is set and it also shows up in the **Breakpoints** tab in the upper right window.

Click the **Resume** icon, the green arrow, to start the program. When connected to `gdbserver`, the program has effectively already started even though we're at the beginning. So resume is the correct action. After a second or two, the program halts with the breakpoint line highlighted.



**Figure 4.7: Debug perspective**

To see the current value of a variable, just place the cursor over it. In about a second a light-blue box appears with the current value. Try it with `fd`. If the open operation in the previous line succeeded, it should be a small positive number. Put the cursor over `led`, which hasn't been initialized yet so its value is random. It's a local variable so its value is whatever happened to be in that location on the stack when the program started. To execute the current line, click one of the **Step** icons, **Step Into** or **Step Over**, in the menu bar. The arrow advances to the following line.

Play around with the `led` program under the Eclipse debugger to get a feel for what you can do with it.

## Resources

[www.eclipse.org](http://www.eclipse.org) -- The official website of the Eclipse Foundation. There's a lot here and it's worth taking the time to look through it. I particularly recommend downloading the *C/C++ Development Toolkit User Guide* (PDF).

Abbott, Doug, *Embedded Linux Development Using Eclipse*, Elsevier, 2008. Check this one out if you want to dig into the details of using Eclipse as an IDE for embedded development.



## 5. Working with Hardware

Inevitably, embedded systems involve diddling hardware. In this chapter we'll see how to access the peripheral devices on the target board.

In this chapter we'll look at one approach to accessing general purpose I/O (GPIO) on the Raspberry Pi from Linux User Space and briefly describe a couple of other approaches.

### User Space vs. Kernel Space

Linux is a “protected mode” operating system, which conveys a high degree of reliability and security. A Linux system is divided into two basic memory areas: *User Space* and *Kernel Space*. Applications run in User Space at the lowest privilege level where they can do little more than read and write memory and invoke services of the operating system.

Each application consists of one or more *processes* where each process has its own private memory. Any attempt to access memory that hasn't been allocated to a process results in a segmentation fault. User space is *swappable* meaning that parts of it can be swapped out to the disk when memory gets tight.

The operating system kernel runs in *Kernel Space* at the highest privilege level where it can do anything. Inadequately debugged kernel code can bring down the system. Unlike User Space, Kernel Space is not swappable. The whole kernel is always in memory. User space code is not allowed to directly access Kernel space.

In both User and Kernel space the addresses referenced are *virtual*. The on-chip Memory Management Unit (MMU) maps the virtual space into actual physical space. One consequence of this is that every process thinks it has access to the full 4 GB of a 32-bit address space. By convention the lower three GB of virtual space are mapped to User space and the top GB is mapped to Kernel space.

### The ARM I/O Architecture

Fundamentally, ARM maps all peripheral registers into a section of memory address space. Thus, in principle, any peripheral register is accessible as an ordinary C variable provided we know its address. But remember that Linux deals in virtual addresses, not physical, so before we can access a block of peripheral addresses, whether in User or Kernel space, we have to map those physical addresses into virtual addresses. The function that does the mapping in User space is `mmap()`. The corresponding Kernel space function is `ioremap()`.

The ARM I/O architecture is quite extensive and complex. The `docs/` directory on your CD has a 295 page *BCM2835 ARM Peripherals* manual. And besides, we only really have access to 26 GPIO pins brought out to the 40-pin header on the Raspberry Pi. Many of these pins have specific peripheral functions. Rather than wade into the complexity of ARM peripherals, we'll take advantage of GPIO support built in to the kernel.

## I/O in Linux

Before we dive into how we'll access our peripheral devices from User Space, it's worth reviewing how I/O is handled in Linux.

One of the primary contributions that Unix made to operating system theory is the notion that “everything is a file”. 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 standard user space utilities can be used to access devices and 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, are randomly accessible, that is, you can move back and forth within a stream of characters. With character devices the stream generally moves in one direction only. But the primary distinction between block and character devices is that the former must have file systems imposed on them whereas the latter 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. This necessitates a different interface between the kernel and the device driver. Network devices are not nodes in the `/dev` directory.

From here on out we'll only be dealing with character devices.

### *The Low Level I/O API*

The set of User Space system functions closest to the hardware is termed “low level I/O”. For the kind of device we're dealing with in this chapter, low level I/O is probably the most useful because the calls are inherently synchronous. That is, the call doesn't return until the data transfer has completed.

```
int open (const char *path, int oflags);
int open (const char *path, int oflags, mode_t mode);
size_t read (int filedес, void *buf, size_t count);
size_t write (int filedес, void *buf, size_t count);
int close (int filedес);
int ioctl (int filedес, int cmd, ...);
```

### Listing 5.1

Listing 5-1 shows the basic elements of the low level I/O API.

- **OPEN.** Establishes a connection between the calling process and the file or device. `path` is the directory entry to be opened. In the case of a peripheral device, it is usually an entry in the `/dev` directory. `oflags` is a bitwise set of flags specifying access mode and must include one of the following:

<code>O_RDONLY</code>	Open for read-only
<code>O_WRONLY</code>	Open for write-only
<code>O_RDWR</code>	Open for both reading and writing

Additionally `oflags` may include one or more of the following modes:

<code>O_APPEND</code>	Place written data at the end of the file
<code>O_TRUNC</code>	Set the file length to zero, discarding existing contents
<code>O_CREAT</code>	Create the file if necessary. Requires the function call with three arguments where <code>mode</code> is the initial permissions.

If **OPEN** is successful it returns a non-negative integer representing a “file descriptor”. This value is then used in all subsequent I/O operations to identify this specific connection.

- **READ and WRITE.** These functions transfer data between the process and the file or device. `filedes` is the file descriptor returned by **OPEN**. `buf` is a pointer to the data to be transferred and `count` is the size of `buf` in bytes. If the return value is non-negative it is the number of bytes actually transferred, which may be less than `count`.
- **CLOSE.** When a process is finished with a particular device or file, it can close the connection, which invalidates the file descriptor and frees up any resources to be used for another process/file connection. It is good practice to close any unneeded connections because there is typically a limited number of file descriptors available.
- **IOCTL.** This is the “escape hatch” to deal with specific device idiosyncrasies. For example a serial port has a baud rate and may have a modem attached to it. The manner in which these features are controlled is specific to the device. So each device driver can establish its own protocol for the **IOCTL** function.

A characteristic of most Linux system calls is that, in case of an error, the function return value is `-1` and doesn't directly indicate the source of the error. The actual error code is placed in the global variable `errno`. So you should always test the function return for a negative value and then inspect `errno` to find out what really happened. Or better yet call `perror()`, which prints a sensible error message on the console.

There are a few other low level I/O functions but they're not particularly relevant to this discussion.

### *Driver internal structure*

We won't go into a lot of detail on the structure of a device driver, because our goal here is not the write a driver, but simply to use it. Fundamentally, the driver has a set of functions that mirror the User Space low-level API functions. When the application process calls `write()`, for example, the kernel turns around and invokes the driver's corresponding write method. The `open()` call establishes the connection between the application and the driver so that the kernel can invoke the correct write function.

How does the kernel make that connection? That's where the `/dev` directory comes in. Execute `ls -l /dev` on the target board. An excerpt is shown in figure 5.1. Just to the left of the date in each entry is a pair of numbers where the file size would normally be. These represent the "major" and "minor" device numbers for the entry. Each entry begins with a 10-letter field representing the file's "permissions." The first letter actually describes what kind of file this is where "c" means a character device, "b" means a block device, "l" means a logical link, and "d" means directory.

When a device driver registers itself with the kernel, it specifies a major number and one or more minor numbers. When an application opens an entry in the `/dev` directory, the kernel uses the corresponding major number to find the correct device driver. Many of the lower major numbers have been pre-assigned to specific classes of common devices. For example, IDE drives are block device 3 and serial ports, usually identified as `/dev/tty*`, are character devices 3 and 4. Major number 5 is generally assigned as the "console" device.

```

[root@FriendlyARM /]# ls -l /dev
crw-rw---- 1 root  root   10, 59 Aug 1 11:57 adc
crw-rw---- 1 root  root   14,  4 Aug 1 11:57 audio
crw-rw---- 1 root  root   10, 63 Aug 1 11:57 backlight
crw-rw---- 1 root  root   10, 61 Aug 1 11:57 buttons
crw-rw---- 1 root  root   10, 58 Aug 1 11:57 camera
crw-rw---- 1 root  root    5,  1 Aug 1 11:56 console
crw-rw---- 1 root  root  116,  0 Aug 1 11:57 controlC0
crw-rw---- 1 root  root   10, 57 Aug 1 11:57 cpu_dma_latency
brw-rw---- 1 root  root   90,  8 Aug 1 11:56 device
crw-rw---- 1 root  root   14,  3 Aug 1 11:57 dsp
crw-rw-rw- 1 root  root   29,  0 Aug 1 11:57 fb0
crw-rw-rw- 1 root  root    1,  7 Aug 1 11:57 full
drwxr-xr-x 2 root  root    0 Aug 1 11:57 i2c
drwxr-xr-x 2 root  root    0 Aug 1 11:57 input
crw-rw---- 1 root  root    1, 11 Aug 1 11:57 kmsg
crw-rw---- 1 root  root   10, 62 Aug 1 11:57 leds
srw-rw-rw- 1 root  root    0 Aug 1 17:57 log
drwxr-xr-x 2 root  root    0 Aug 1 11:57 loop
crw-rw---- 1 root  root    1,  1 Aug 1 11:57 mem
crw-rw---- 1 root  root   14,  0 Aug 1 11:57 mixer
crw-rw---- 1 root  root   90,  0 Aug 1 11:57 mtd0
crw-rw---- 1 root  root   90,  1 Aug 1 11:57 mtd0ro
crw-rw---- 1 root  root   90,  2 Aug 1 11:57 mtd1
crw-rw---- 1 root  root   90,  3 Aug 1 11:57 mtd1ro
crw-rw---- 1 root  root   90,  4 Aug 1 11:57 mtd2
crw-rw---- 1 root  root   90,  5 Aug 1 11:57 mtd2ro
crw-rw---- 1 root  root   90,  6 Aug 1 11:57 mtd3
crw-rw---- 1 root  root   90,  7 Aug 1 11:57 mtd3ro
crw-rw---- 1 root  root   90,  8 Aug 1 11:57 mtd4
crw-rw---- 1 root  root   90,  9 Aug 1 11:57 mtd4ro
brw-rw---- 1 root  root   31,  0 Aug 1 11:57 mtdblock0
brw-rw---- 1 root  root   31,  1 Aug 1 11:57 mtdblock1
brw-rw---- 1 root  root   31,  2 Aug 1 11:57 mtdblock2
brw-rw---- 1 root  root   31,  3 Aug 1 11:57 mtdblock3
brw-rw---- 1 root  root   31,  4 Aug 1 11:57 mtdblock4

```

**Figure 5.1: /dev directory**

## The /sys File System

The User space interface for the kernel's GPIO support is the `/sys` file system. It's often useful, even necessary, to see what's going on in the kernel. Remember that User space has no way to directly access kernel space. So to see what's happening in the kernel we make use of a couple of "pseudo" file systems: `/proc` and `/sys`. The entries in these two directories are not real files. The data for them is generated in real time when a process asks to read one.

Think of `/proc` and `/sys` as "windows" into the kernel. `/proc` has been around for a long time. Its primary focus is to provide information about running processes. If you `ls /proc`, the first thing you notice is a large number of subdirectories whose names are numbers. Each one of these represents a running process. Within those subdirectories are other subdirectories and files that tell you pretty much everything you'd want to know about a process.

Whereas `/proc` is open ended and, frankly, somewhat "unruly" in terms of file formats, `/sys`, which is newer, is much more structured. The role of `/sys` is to monitor various kernel subsystems with particular emphasis on the Linux driver model.

An `ls` of `/sys` reveals 10 subdirectories:

- **block** – has a subdirectory for each block device in the system.
- **bus** – has a subdirectory for each bus in the system. These may be cross linked reflecting the interconnection of busses. For example a USB controller may be attached to a PCI bus.
- **class** – groups peripheral devices by functionality.
- **dev** – has two subdirectories: `block/` and `char/`. Each of these has links named for a device's major and minor device numbers that point into the `devices/` directory.
- **devices** – contains representations of all the peripheral devices in the system
- **firmware** – holds a representation of the device tree and the flattened device tree blob (FDT).
- **fs** – has a subdirectory for each file system type in the system
- **kernel** – contains various kernel attributes that don't fit anywhere else
- **module** – has a subdirectory for every kernel module that is currently loaded
- **power** – controls the power subsystem

If you do an `ls -l` on any node in `/sys` you'll discover that a great many of the entries are links to other entries in `/sys`. Devices plug into busses and are managed by drivers, and so on. I would estimate that at least a third of all entries in `/sys` are links.

### *Accessing GPIO Through /sys*

Of particular interest to us is `/sys/class/gpio`. `cd` to that directory. It is through this directory that GPIO is made visible to User space. Initially `gpio/` holds two files: `export` and `unexport` and three directories: `gpiochip0/`, `gpiochip100/` and `gpiochip128/`. These directories represent specific GPIO devices and point to entries under `/sys/devices/`.

The `export` and `unexport` files are write-only. Writing a GPIO pin number to `export` “exports” that pin from Kernel space to User space. Referring to Appendix A, we see that LED1 on the Matrix CK board is controlled by GPIO13. If we execute the command:

```
echo 13 > export
```

the directory `gpio13/` shows up under `gpio/`. `gpio13/` has the following read/write attributes (files):

- `direction` – reads as either “in” or “out” indicating whether this is an input pin or output. Initially the pin is input. Writing “out” to this file makes it an output.
- `value` – reads as “0” or “1” indicating respectively the “inactive” and “active” states of the pin. If this is an output pin, this file can be written. Any non-zero value is equivalent to 1.
- `active_low` – reads as either 0 (false) or 1 (true). `active_low` true inverts the `value` attribute so that 1 sets the pin to the low state.
- `edge` – reads as “none”, “rising”, “falling”, or “both”. This file only exists if the pin can be configured to generate interrupts. This selects which edge(s) generates the interrupt.

Try it out on your target board. Execute the following commands starting in the `gpio/` directory:

```
echo 13 > export
cd gpio13
echo out > direction
echo 1 > value
echo 0 > value
```

Congratulations! You can turn the LEDs on and off from the shell. Try the other LEDs. When you’re finished, `echo <led_number> > unexport` for each of the LEDs you exported. Failure to do so will prevent subsequent applications from using those GPIO pins. When you kill a program in the debugger it does not properly unexport the GPIOs that it used. For the LEDs and pushbuttons, there’s an `unexport` script in `src/` that cleans up properly.

## *The led Program*

With that background, let’s see how the `led.c` program works. `led.c` should already to open in the Eclipse editor. It would be helpful to turn on line numbers. Right click in the marker bar and select Show Line Numbers.

The LEDs and push buttons are managed by a set of functions in `libmchw.so`. Around line 35 `init_leds()` is called to set up the GPIOs for the LEDs. That’s followed by a call to `init_pbs()` to set up the GPIOs for the push buttons. Then around line 53 are `ledON()` and `ledOFF()` that each take a number between 0 and 3 representing one of the four LEDs. `read_pbs()` returns a bit mask representing the state of the three push buttons.

To see these functions create a new makefile project for the library, which is located in `src/lib/`. Open the file `gpio.c`. The first thing we have to deal with are the annoying question marks by the `#include` directives. Rather than type that very long path name that we did in the previous chapter, we can *export*

the include paths from the led project and *import* them into the just created library project. Proceed as follows:

1. Select the led project entry in the Project Explorer view. Right click and select Export...
2. In the Export dialog, expand C/C++ and select C/C++ Project Settings. Click Next.
3. Make sure the led project is selected and uncheck # Symbols.
4. Browse to the src/ directory and enter a file name. I used includes. The data will be saved as a XML file.
5. Click Finish.

Next are the steps for importing the project settings into the library project:

1. Right-click the library project entry in Project Explorer and select Import...
2. Expand C/C++ and select C/C++ Project Settings. Click Next.
3. Browse to the src/ directory and select includes.xml, or whatever you chose to name the file. Click OK.
4. Click Finish.

gpio.c has functions to export and unexport a GPIO pin. `exportGPIOPin()` returns an open file descriptor for the corresponding value file. Then there are functions to get and set the `value`, `direction`, and `active_low` files. These are followed by a set of functions to manage the LEDs and the push buttons.

Note in the function `getGPIOValue()` that we must seek to the beginning of the file each time we read. That's because the file only returns one value before reaching the end of file.

## PIGPIO—Another Approach to Hardware Access

If you want to get closer to the hardware, there's a package called PIGPIO (pronounced "PI GPIO", not "Pig Pio"). It's found in a directory of the same name under `home/src/` in the target file system. PIGPIO provides a C library with the following features:

- Hardware timed PWM on any of GPIO 0 to 31
- Hardware timed servo pulses on any of GPIO 0 to 31
- Callbacks or notifications when any of GPIO 0 to 31 change state
- Callbacks at timed intervals
- I2C, SPI, and serial link wrappers

among others. C programs can link directly to this shared library, or you can run a daemon process in the background whereby you can access the library through sockets or pipes. There's also a Python interface.

We won't go into any detail on PIGPIO, I leave that as the proverbial "exercise for the reader". Building PIGPIO is relatively straightforward. You'll need to edit the Makefile. Set `CROSS_PREFIX=arm-poky-linux-gnueabi-` and define `DESTDIR=/home/target_fs`. Then execute:



```
make
sudo make install
```

On the target execute:

```
ldconfig /usr/local/lib
```

This makes the new PIGPIO libraries visible to applications. Once you've built the package, there are a number of tests you can run:

```
./x_pigpio      # tests the basic library operation
```

This program makes extensive use of GPIO 25, which happens to be connected to the Matrix Compact's buzzer, so you'll hear some strange sounds when running it. Other tests:

```
pigpiod        # to start the daemon
x_pigs         # script that tests sockets interface
x_pipe         # script that tests pipe interface
```

## WiringPI

WiringPI is another open source GPIO library specifically for the Raspberry PI. It is written in C but there are wrappers for Python, Perl, Ruby, and possibly others. Interestingly, it is entirely developed on the Raspberry PI running the Raspbian version of Linux. The original author does not support cross-compiling or running under other OSes, although ports do exist that support these features.

WiringPI uses a pin numbering scheme that harks back to the origins of the PI. The intention is to keep the numbering independent of various changes that have occurred over time in the PI's evolution. From the project's web page: WiringPI is "designed to be familiar to people who have used the Arduino '*wiring*' system and is intended for use by experienced C/C++ programmers. It is not a newbie learning tool."

## Resources

<http://abyz.me.uk/rpi/pigpio/> -- This is the web page for PIGPIO.

[wiringpi.com](http://wiringpi.com) – web page for wiringpi

## 6. A Real-World Application

Now that we know how to access GPIO on the target, it's time to move on to something useful. The Matrix CK hat incorporates a multi-channel A/D converter. Channel 0 is connected to a pot. We'll use that as the basis of a simple data acquisition example that will serve throughout the remainder of this tutorial.

### The PFC8591 A to D Converter

The Matrix CK includes a PFC8591 chip described by its manufacturer, Phillips (now NXP), as a “data acquisition device” that incorporates a 4-channel, 8-bit analog to digital converter and a single 8-bit digital to analog converter. The `docs/` directory has a datasheet for the PFC8591.

The PFC8591 connects to the R Pi's processor through the I2C bus. I2C, pronounced “I squared C”, stands for inter-integrated circuit. It is a simple 2-wire multi master, multi slave serial bus intended for connecting lower-speed peripheral ICs to processors and microcontrollers. It was invented by Phillips back in 1982.

The PFC8591 datasheet has a brief description of how I2C works. Needless to say, Linux has extensive driver support for it. `libmchw.so` provides support for the PFC8591 in the file `pcf8591.c`. It contains three functions:

- `init_AD (channel)` – Initializes one channel of the PFC8591 by writing the device name and I2C address to `/sys/bus/i2c/devices/i2c-1/new_device`. This creates a directory `1-0048/` under `/sys/bus/i2c/devices/` containing a number of files including `in0_input` through `in3_input` that are the value files for the four input channels. It returns an open file descriptor for the value file of the specified channel.
- `close_AD (fd)` – simply closes the specified file descriptor.
- `read_AD (fd, &value)` – reads the input channel identified by the file descriptor, converts the resulting string to an integer and returns it in `value`. The input value is divided by 10 since the PCF8591 driver multiplied it by 10. Function return value is 0 on success.

Like GPIO, we access the PCF8591 through the `/sys` file system. Have a look at `/sys/bus/i2c/devices`. Under that is a directory `i2c-1/` that represents the I2C bus controller. In that directory is a write-only file `new_device`. If we write the string “`pcf8591 0x48`” to that file, it will load the PCF8591 driver and instantiate a device at address `0x48` on the bus. It also creates the directory `1-0048/` under `i2c-1/`. In this directory are files for reading each of the four ADC channels as well as writing the DAC channel. The I2C bus has been enabled in `config.txt`, so its driver is loaded at boot time.

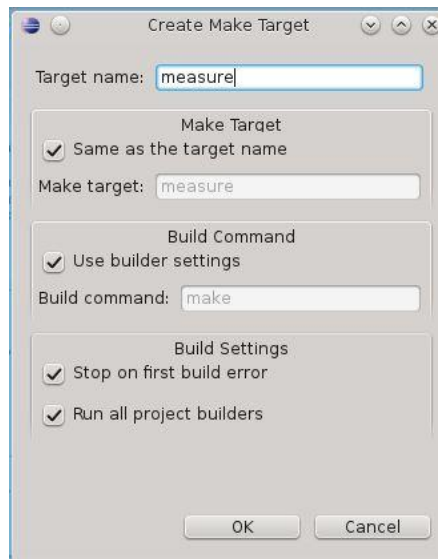
The makefile for the `measure` project has several build targets. By default Eclipse builds the “all” target if it is present. Other build targets require that they be declared to Eclipse. On the right-hand side

of the C/C++ perspective is a view normally labeled **Outline** that shows an outline of whatever is currently displayed in the Editor window. There is another tab in that view labeled **Make Targets** that allows us to specify alternate make targets. Select that tab, right click on the measure project entry and click **New**.

Figure 6.1 shows the dialog brought up by **New**. In this case the **Target Name** is **measure**. The **Make Target** is also **measure**, so we don't have to retype it. Eclipse provides a shortcut for the common case when the target name and make target are the same. Click the **create** button. Now in the **Make Targets** view, under the **measure** project, right click on the **measure** target and select **Build Make Target**. In the **Project Explorer** view on the left-hand side of the perspective two new files show up: **measure.o** and **measure**.

Run the program on the target. The ADC returns 8-bit data in the range of 0 to 255. Turn the pot with a small screwdriver and you should see the reported value change appropriately.

If you want to run the program under the Eclipse debugger, simply open the **Debug Configurations** dialog, change the project name to **measure**, the path to the application and the absolute file path to point to the **measure** program. Click **Apply**. Now click **Debug** in the **Debug Configurations** dialog.



**Figure 6.1: Creating a new make target**

## 7. A Thermostat

In this chapter you'll have a chance to do some "real programming." We'll enhance the measure program by turning it into a "thermostat." We'll also take a look at using the host workstation as a preliminary testing environment.

We'll have our thermostat activate a "cooler" when the temperature rises above a given setpoint and turn the cooler off when the temperature falls below the setpoint. In practice, real thermostats incorporate hysteresis that prevents the cooler from rapidly switching on and off when the temperature is right at the setpoint. This is implemented in the form of a "deadband" such that the cooler turns on when the temperature rises above the setpoint + deadband and doesn't turn off until the temperature drops below setpoint - deadband. Additionally, the program includes an "alarm" that flashes if the temperature exceeds a specified limit.

Two of the LEDs will serve as the cooler and alarm respectively. These are defined in `driver.h` in the `includes/` directory as `COOLER` and `ALARM`. The thermostat is fundamentally a simple state machine with three states based on indicated temperature:

- Normal
- High
- Limit

For each state, the action taken and next state are:

Current state Normal

Temperature above setpoint + deadband

Turn on COOLER

Next state = High

Temperature above limit

Turn on ALARM

Next state = Limit

Current state High

Temperature below setpoint - deadband

Turn off COOLER

Next state = Normal

Temperature above limit

Turn on ALARM

Next state = Limit

Current state Limit

Temperature below limit

```
Turn off ALARM
Next state = High
Temperature below setpoint - deadband
Turn off COOLER
Next state = Normal
```

The state machine itself is just a `switch()` statement on the state variable.

Make a copy of `measure.c` and call it `thermostat.c`. Since `thermostat.c` is already a prerequisite in `measure's` makefile, all you have to do to make it visible to Eclipse is to right click on `measure` in the Project Explorer view and select `Refresh`.

Implement the state machine in `thermostat.c`. Declare the variables `setpoint`, `limit`, and `deadband` as global integers. Pick a suitable `setpoint` and a limit another few “degrees” above that. A `deadband` of plus/minus one count is probably sufficient.

In case you haven't already noticed, Eclipse does not automatically save any modified files when you build a project. You must explicitly save them. There is a configuration option to change that behavior if you choose<sup>8</sup>.

## Host workstation as debug environment

Although remote GDB gives us a pretty good window into the behavior of a program on the target, there are good reasons why it might be useful to do initial debugging on your host development machine. To begin with, the host is available as soon as a project starts, probably well before any real target hardware is available or working. The host has a file system that can be used to create test scripts and document test results.

When you characterize the content of most embedded system software, you will usually find that something like 5% of the code deals directly with the hardware. The rest of the code is independent of the hardware and therefore shouldn't need hardware to test it, provided that the code is properly structured to isolate the hardware-dependent elements.

The idea here is to build a simple simulation that stands in for the hardware I/O devices. You can then exercise the application by providing stimulus through the keyboard and noting the output on the screen. In later stages of testing you may want to substitute a file-based “script driver” for the screen and keyboard to create reproducible test cases.

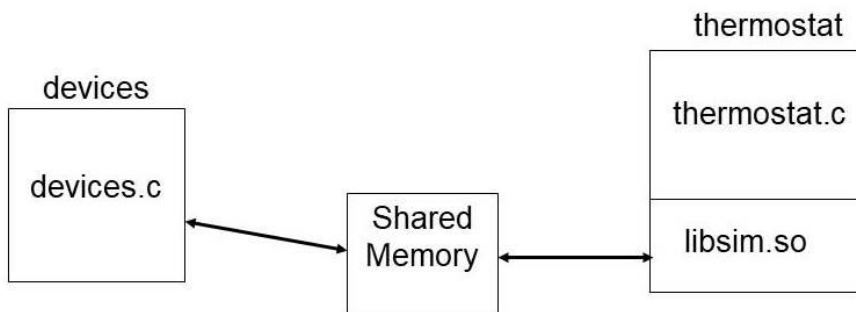
The simulation is implemented by simply substituting a different library for `libmchw.so`. It's called `libsimsim.so` and it's in the directory `sim-lib/`. This library exposes the same APIs as `libmchw.so` but uses a shared memory region to communicate with another process that displays digital outputs on the screen

---

<sup>8</sup> Personally, I think automatic saving should be the default

and accepts analog inputs via the keyboard. This functionality is implemented in `devices.c`. The shared memory region consists of a data structure of type `shmem_t` (defined in `driver.h`) that includes fields for an analog input and a set of digital outputs that are assumed connected to LEDs. It also includes a process ID field (`pid_t`) set to the pid of the `devices` process that allows the `thermostat` process to signal when a digital output has changed.

`devices` creates and initializes the shared memory region. In `libsim`, `exportGPIOpin()` attaches to the previously created shared memory region the first time it is called. `read_AD()` simply returns the current value of the `a2d` field in shared memory. The `LEDOn()` and `LEDOff()` functions turn around and call `setGPIOvalue()`, which in this case sends a signal to the `devices` program to update the output display. Figure 7.1 illustrates the process graphically.



**Figure 7.1: Thermostat simulation**

The executable for the simulation version of the thermostat is called `thermostat_s` (`_s` for simulation) and is the default target in the project makefile. In fact, it was automatically built when you did the Refresh operation to bring `thermostat.c` into Eclipse. To build it again after editing `thermostat.c`, select the Project menu and Build Project.

To build the `devices` program, we need to add another target to the project. Right click on `measure` in Project Explorer and select Make targets -> Create. The Target Name is “`devices`” and the Make Target is the same. Go ahead and build the `devices` target.

Run `devices` in a shell window. To debug `thermostat_s` we’ll have to create another launch configuration that works with the host. From the Run menu select Debug Configurations. Highlight C/C++ Application and click the New button. Name the new configuration “host.” The application is `thermostat_s`. In the Debugger tab verify that the Debugger is `gdb/mi` and the GDB debugger is `gdb`. Click Debug to bring up the Debug perspective with the program halted at the first executable line of `main()`.

Debug your thermostat program by entering different values for the A/D input into the devices program and watch how the thermostat state machine responds.

When you feel the thermostat is running correctly in the simulation, you can rebuild it for the target board by creating another make target. Call this one “ARMthermo.” In this case, the **Make Target** is “all.” Uncheck **Same as the target name** and enter “all” in the **Make target** window. Uncheck **Use default** and specify the **Build command** as “make TARGET=1.” The actual value is not important, it’s only necessary to define the symbol. Before building ARMthermo, delete the file thermostat.o so that it will be recompiled with the ARM compiler. Otherwise the ARM linker will complain about an invalid file format.

To debug thermostat\_t you can use the configuration created earlier for the led project. You might want to rename it something like “remote.” Change the **Project** to “measure” and the **Application** to “thermostat\_t.”

## 8. Posix Threads

As currently configured, our thermostat has a hard coded setpoint and limit. Not terribly useful. In most cases, it would be nice to be able to change these parameters remotely, over a serial line say, or even a network.

The obvious solution is to use `stdin` to enter simple commands to change these parameters. Back in the old days, if we were building this under DOS, we would probably use the function `kbhit()` to poll for console input within the main processing loop in `thermostat`. But Linux is a multi-tasking system. Polling is tacky! The proper solution is to create an independent thread of execution whose sole job is to wait for console input and then interpret it.

There are a couple of different ways we could create that second thread of execution. We could “fork” a second process from within `thermostat.c`, or we could start a second process from the shell, or we could use threads.

### Creating a Linux Process – the `fork()` function

The `fork()` function can seem a little bizarre the first time you encounter it. Linux starts life with one process, the `init` process, created at boot time. Every other process in the system is created by invoking `fork()`. The process calling `fork()` is termed the *parent* and the newly created process is termed the *child*. So every process has ancestors and may have descendants depending on who created who.

Conceptually, `fork()` creates a copy of the parent process—code, data, file descriptors and any other resources the parent may currently hold. This could add up to megabytes of memory space to be copied. To avoid copying a lot of stuff that may be overwritten anyway, Linux employs a *copy-on-write* strategy. So the only things that get copied initially are the task structure and the page tables.

Since both processes are executing the same code, they both continue from the return from `fork()`. In order to distinguish parent from child, `fork()` returns a function value of 0 to the child process but returns the PID of the child to the parent process.

The concept behind `fork()` is that *both processes continue in parallel*. So in the case of our thermostat, we could create a child process that simply monitors `stdin` for commands to alter the operating parameters. The next question of course is, how do we communicate the changed parameter value back to the parent process? In a conventional memory-protected Linux implementation, each process has its own private address space that no other process has direct access to. So the only way for two processes to share data is through shared memory as we've already seen with the simulation.



```
int running = 1;
params_t *p;    //pointer to shared memory

void main (void)
{
    create shared memory space;
    switch (fork()) {
        case -1:
            printf ("fork failed\n");
            break;
        case 0:    // child process
            attach to shared memory space;
            while (running) {
                fgets();
                parse command;
                put result in shared memory;
            }
            break;
        default:    // parent process
            attach to shared memory space;
            while (running) {
                read A/D;
                act on current state;
            }
            break;
    }
    exit (0);
}
```

### Listing 8.1

Listing 8.1 illustrates in pseudo code form how we might implement the thermostat with fork. Fork is usually coded in a switch statement. A return value of -1 indicates a failure. Fork almost never fails.

Of course, what really happens 99% of the time is that the child process invokes a new program by calling `execve()`. In the case of our thermostat, the child process `execve()`s the monitor program. Note that `execve()` doesn't return if it succeeds. Listing 8.2 shows a pseudo code version of using `execve()`.

```
int running = 1;
params_t *p;    //pointer to shared memory

void main (void)
{
    create shared memory space;
    switch (fork()) {
        case -1:
            printf ("fork failed\n");
            break;
        case 0:    // child process
            Put p into argv for execve
            execve ("monitor", argv, environ);
            printf ("execve failed\n");
            break;
        default:
            while (running)    {
                read A/D;
                act on current state;
            }
            break;
    }
    exit (0);
}
```

## Listing 8.2

This in fact is how the shell works. When you enter a command to the shell, **make** for example, it interprets the command name as a file and **execve()**s it. In our case, the parent goes on to execute the basic thermostat algorithm. The shell, on the other hand, normally waits for the child to finish by calling **wait\_pid()**, passing in the PID of the child process. **wait\_pid()** blocks the calling process until the specified process terminates.

But probably the most efficient approach to implementing an independent thread of execution is the concept of "threads".

As we've seen, the basic structural element in Linux is a *process* consisting of executable code and a collection of *resources* like data, file descriptors and so on. These resources are fully protected such that one process can't directly access the resources of another. In order for two processes to communi-

cate with each other, they must use the inter-process communication mechanisms defined by Linux such as shared memory regions as we've already seen.

But suppose we create two *threads* within one process such that one thread carries out the thermostat algorithm and the other waits for input at `stdin`. The operational parameters, setpoint, limit and dead-band, reside in memory that is accessible to both threads. So the `stdin` monitoring thread modifies parameters as required and the thermostat algorithm uses these values without caring that they may have been changed.

## Introducing Threads

We'll use Posix Threads, also known as Pthreads, to create our independent thread of execution. Posix, also written POSIX, is an acronym that means Portable Operating System Interface with an X thrown in for good measure. POSIX represents a collection of standards defining various aspects of a portable operating system based on UNIX. These standards are maintained jointly by the Institute of Electrical and Electronic Engineers (IEEE) and the International Standards Organization (ISO). Recently the various documents have been pulled together into a single standard in a collaborative effort between the IEEE and The Open Group (see the Resources section).

Before diving into a programming example, let's briefly review the threads programming model. The Resources section at the end of the chapter lists an excellent book for further study.

Fundamentally a thread is the same thing that most multitasking OSes call a *task*. It is an independent thread of execution embodied in a function. Each thread has its own stack.

```
int pthread_create (pthread_t *thread, pthread_attr_t *attr, void
    *(*start_routine) (void *), void *arg);
void pthread_exit (void *retval);
int pthread_join (pthread_t thread, void **thread_return);
pthread_t pthread_self (void);
int sched_yield (void);
```

### Listing 8.3

The mechanism for creating and managing a thread is analogous to creating and managing the tasks. As shown in listing 8.3, a thread is created by calling `pthread_create()` with the following arguments:

- `pthread_t` – A *thread object* that represents or identifies the thread. `pthread_create()` initializes this as necessary..
- Pointer to a thread *attribute* object. Often it is `NULL`. More on this later.
- Pointer to the *start routine*. The start routine takes a single pointer to void argument and returns a pointer to void.

- Argument to be passed to the start routine when it is called.

Note incidentally that any number of threads can be created using the same start routine. Each one requires a separate thread object and probably a different argument.

A thread may terminate by calling `pthread_exit()` or simply returning from its start function. The argument to `pthread_exit()` is the start function's return value.

In much the same way that a parent process can wait for a child to complete by calling `waitpid()`, a thread can wait for another thread to complete by calling `pthread_join()`. The arguments to `pthread_join()` are the thread object of the thread to wait on and a place to store the thread's return value. The calling thread is blocked until the target thread terminates.

### *Thread Attributes*

POSIX provides an open-ended mechanism for extending the API through the use of *attribute objects*. For each type of Pthreads object there is a corresponding attribute object that is, in essence, an extended argument list to the related object's create or initialize function. A pointer to an attribute object is always the second argument to a create function. If this argument is NULL the create function uses appropriate default values. This also has the effect of keeping the create functions relatively simple by leaving out a lot of arguments that normally take default values.

An important philosophical point is that all Pthreads objects are considered to be “opaque”. This means that you never directly access members of the object itself. All access is through API functions that get and set the member arguments of the object. This allows new arguments to be added to a Pthreads object type by simply defining a corresponding pair of get and set functions for the API. In simple implementations the get and set functions may be nothing more than a pair of macros that access the corresponding member of the attribute data structure.

Before an attribute object can be used, it must be initialized. Then any of the attributes defined for that object may be set or retrieved with the appropriate functions. This must be done before the attribute object is used in a call to `pthread_create()`. If necessary, an attribute object can also be “destroyed”. Note that a single attribute object can be used in the creation of multiple threads.

The only required attribute for thread objects is the “detach state”. This determines whether or not a thread can be joined when it terminates. The default detach state is `PTHREAD_CREATE_JOINABLE` meaning that the thread can be joined on termination. The alternative is `PTHREAD_CREATE_DETACHED`, which means the thread can't be joined.

Joining is useful if you need the thread's return value or if you need to synchronize several threads shutting down. Otherwise it's better to create the thread detached. The resources of a joinable thread can't be recovered until another thread joins it whereas a detached thread's resources can be recovered as soon as it terminates.

## Synchronization—Mutexes

Pthreads uses the *mutex* (short for “mutual exclusion”) as its primary mechanism for providing exclusive access to shared resources. When a thread needs exclusive access, to a shared memory region for example, it *locks* the mutex protecting that region. When it is finished, the thread *unlocks* the mutex. While a mutex is locked, other threads attempting to lock it are blocked until the mutex is unlocked.

The Pthreads mutex API follows much the same pattern as the thread API. There is a pair of functions to initialize and destroy mutex objects and a set of functions to act on the mutex objects.

Two operations may be performed on a mutex: *lock* and *unlock*. The lock operation causes the calling thread to block if the mutex is not available. There's another function called *trylock* that allows you to test the state of a mutex without blocking. If the mutex is available *trylock* returns success and locks the mutex. If the mutex is not available it returns `EBUSY`.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init (pthread_mutex_t *mutex, const
    pthread_mutexattr_t *mutex_attr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

### Listing 8.4

## The thermostat with threads

We now have enough background to add a thread to our thermostat to monitor the serial port for changes to the parameters. We'll use a very simple protocol to set parameters consisting of a letter followed by a space followed by a number. “s” represents the setpoint “l” the limit, and “d” the deadband. So to change the setpoint you would enter, for example,

```
s 65<Enter>
```

This sets the setpoint to 65 degrees.

Copy `thermostat.c` that currently resides in `src/measure/` to `src/posix/`. Create a new Eclipse project called “posix” located in the `posix/` directory. Open `monitor.c`. The first thing to notice is `#include <pthread.h>`. This header file prototypes the Pthreads API. Note the declarations of `paramMutex` and `monitorT`. The latter will become the handle for our monitor thread.

Because the parameters, `setpoint`, `limit`, `deadband`, are accessed independently by the two threads, the thermostat and monitor, we need a mutex to guarantee exclusive access to one thread or the other. In

reality, this particular application is so simple it probably doesn't matter, but it does serve to illustrate the point.

Note incidentally that `setpoint`, `limit`, and `deadband` are declared `extern` in `thermostat.h`. It is assumed that these variables are allocated in `thermostat.c`. If you happened to use different names for your parameters, you'll need to change them to match.

Take a look at the `monitor()` function. It's just a simple loop that gets and parses a string from `stdin`. Note that before `monitor()` changes any of the parameters, it locks `paramMutex`. At the end of the switch statement it unlocks `paramMutex`. This same approach will be required in your thermostat state machine.

Move down to the function `createThread()`. This will be called from `main()` in `thermostat.c` to initialize `paramMutex` and create the monitor thread. `createThread()` returns a non-zero value if it fails. `terminateThread()` makes sure the monitor thread is properly terminated before terminating `main()`. It should be called from `main()` just before `closeAD()`.

### *Changes required in thermostat.c*

Surprisingly little changes to make the thermostat work with the monitor.

- a. include `pthread.h` and `thermostat.h`.
- b. call `createThread()` and test the return value at the top of `main()`.
- c. lock `paramMutex` just before the switch statement on the state variable and unlock it at the end of the switch statement.
- d. call `terminateThread()` just after the running loop.

That's it! The makefile supports both target and simulation builds as described in the last chapter. Build the simulation and try it out.

### *Debugging threaded programs*

Multi-threading does introduce some complications to the debugging process but, fortunately, GDB has facilities for dealing with those. Debug `thermostat_s` in Eclipse (be sure `devices` is running) using the host configuration and set a breakpoint just after the call to `createThread()`. When it stops at the breakpoint you'll see references to two threads in the **Debug** view:

```
Thread [1] (Suspended: Breakpoint hit.)  
= 1 main() thermostat.c:49 0x08048975  
Thread [2] (Suspended)
```

Thread [1] is the `main()` function in `thermostat.c`, so Thread [2] must be the monitor. It too is suspended, probably in the `fgets()` function. Note that your program counter value may differ.

Open `monitor.c` and set a breakpoint just after the call to `fgets()`. Now enter a line of text in the **Console** view. It doesn't really matter if it's a valid command or not. All we want to do at this point is get

the program to stop at the breakpoint. Let the program continue, and when it reaches the breakpoint note that Thread [2] is now the one suspended at a breakpoint. Thread [1] is most likely suspended inside the `sleep()` function.

Play around with setting breakpoints in various places in both `main()` and `monitor()` to verify that parameters get updated properly and the thermostat responds correctly to the new values.

When you're satisfied with the program running under the simulation, rebuild it for the target and debug it there.

When you start the debugger you're likely to get a warning dialog saying that a required project has errors and do you want to proceed? Yes, you do. The reason for this is that every time you go into the debugger, CDT tries to rebuild the "all" target. Since "all" is the simulation version, it fails because the last thing you built was probably the target version. There's a Preferences option to turn off the automatic build.

The debugger stops at the first line of `main()` as normal. Let the program execute to the breakpoint just after `createThread()`. The Debug view shows two threads just as we saw in the simulation version. From here on out, the target debugging process is essentially identical to the simulation mode.

## Resources

The Open Group has made available free for private use the entire Posix (now called Single Unix) specification. Go to:

[www.unix.org/single\\_unix\\_specification](http://www.unix.org/single_unix_specification)

You'll be asked to register. Don't worry, it's painless and free. Once you register you can read the specifications online or download the entire set of html files for local access.

This chapter has been little more than a brief introduction to the complex world of Posix threads. An excellent, more thorough treatment is found in, Butenhof, David R., *Programming with POSIX Threads*, Addison-Wesley, 1997

## 9. Networking

Everything is connected to the Internet, even refrigerators<sup>9</sup>. So it's time to turn our attention to network programming in the embedded space. Linux, as a Unix derivative, has extensive support for networking.

We'll begin by looking at the fundamental network interface, the socket. With that as a foundation, we'll apply the socket paradigm to our thermostat example and then go on to examine how common application-level network protocols can be used in embedded devices. We'll even build a simple web server.

### Sockets

The “socket” interface forms the basis for most network programming in Unix systems. Sockets are a generalization of the Unix file access mechanism that provides an endpoint for communication either across a network or within a single computer. A socket can also be thought of as an extension of the named pipe concept that explicitly supports a client/server model wherein multiple clients may be attached to a single server.

Both the client and server may exist on the same machine. This simplifies the process of building and testing client/server applications. You can test both ends on the same machine before distributing the application across a network. By convention, network address 127.0.0.1 is a “local loopback” device that allows a client to connect to a server on the same machine. Processes can use this address in exactly the same way they use other network addresses.

### *The Server Process*

Figure 9.1 illustrates the basic steps that the server process goes through to establish communication. We start by creating a socket and then `bind()` it to a name or destination address. For local sockets, the name is a file system entry often in `/tmp` or `/usr/tmp`. For network sockets it is a *service identifier* consisting of a “dotted quad” Internet address (as in 192.168.1.2 for example) and a protocol port number. Clients use this name to access the service.

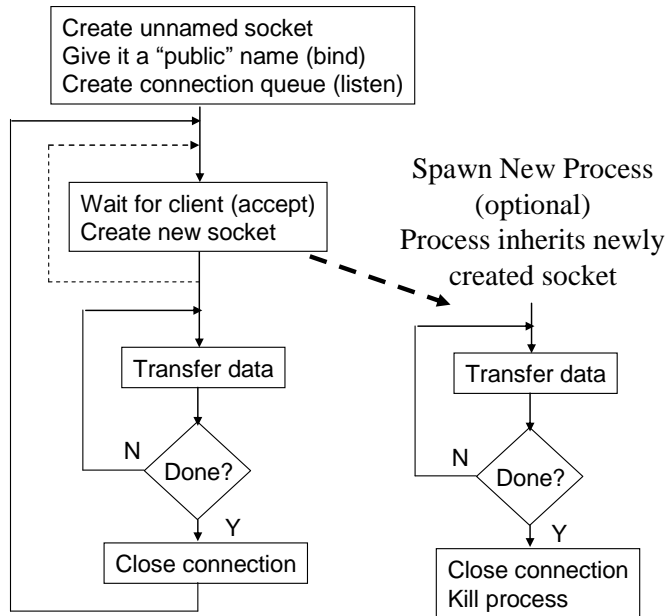
Next, the server creates a connection queue with the `listen()` service and then waits for client connection requests with the `accept()` service. When a connection request is received successfully, `accept()` returns a new socket, which is then used for this connection's data transfer. The server now transfers data

---

<sup>9</sup> <http://www.engadget.com/2011/01/12/samsung-wifi-enabled-rf4289-fridge-cools-eats-and-tweets-we-go/> or just Google “internet refrigerator”. I don't suppose there are any net-connected toaster ovens yet.



using standard `read()` and `write()` calls that use the socket descriptor in the same manner as a file descriptor. When the transaction is complete the newly created socket is closed.



**Figure 9.1: Server Process**

The server may very well spawn a new process or thread to service the connection while it goes back and waits for additional client requests. This allows a server to serve multiple clients simultaneously. Each client request spawns a new process with its own socket.

### *The Client Process*

Figure 9.2 shows the client side of the transaction. The client begins by creating a socket and naming it to match the server's publicly advertised name. Next, it attempts to `connect()` to the server. If the connection request succeeds, the client proceeds to transfer data using `read()` and `write()` calls with the socket descriptor. When the transaction is complete, the client closes the socket.

If the server spawned a new process to serve this client, that process should go away when the client closes the connection.

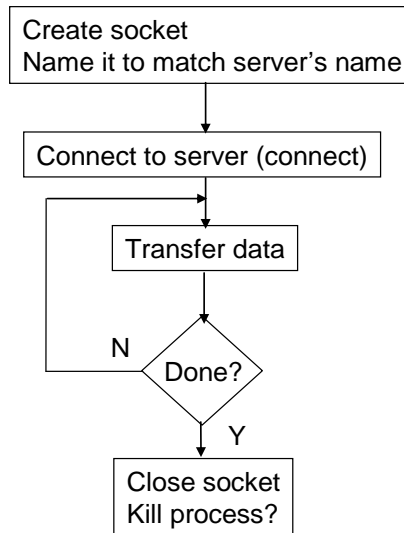
### *Socket Attributes*

The socket system call creates a socket and returns a descriptor for later use in accessing the socket.

```
#include <sys/socket.h>
```

```
int socket (int domain, int type, int protocol);
```

A socket is characterized by three attributes that determine how communication takes place. The *domain* specifies the communication medium. The most commonly used domains are `PF_UNIX` for local file system sockets and `PF_INET` for Internet connections. The “PF” here stands for Protocol Family.



**Figure 9.2: Client Process**

The domain determines the format of the socket name or address. For `PF_INET`, the address is `AF_INET` and is in the form of a dotted quad. Here “AF” stands for Address Family. Generally there is a 1 to 1 correspondence between `AF_` values and `PF_` values and in fact they have the same numeric values. A network computer may support many different network services. A specific service is identified by a “port number”. Established network services like ftp, http, etc. have defined port numbers, usually below 1024. Local services may use port numbers above 1023.

In many cases the protocol is usually determined by the socket domain and type and you don’t have a choice. So the protocol argument is usually zero.

## A Simple Example

### The Server

Create a new Eclipse project called “network” in the `src/network` directory and open the file `net-serve.c`. First we create a `server_socket` that uses streams. Next we need to bind this socket to a specific network address. That requires filling in a `sockaddr_in` structure, `server_addr`. The function

`inet_aton()` takes a string containing a network address as its first argument, converts it to a binary number and stores it in the location specified by the second argument, in this case the appropriate field of `server_addr`. `inet_aton()` returns zero if it succeeds. In this example the network address is passed in through the compile-time symbol `SERVER` so that we can build the server to run either locally through the loopback device or across the network.

The port number is 16 bits and is arbitrarily set to 4201 in the makefile. The function `htons()` is one of a small family of macros that solves the problem of transferring binary data between computer architectures with different byte ordering policies. The Internet has established a standard “network byte order”, which happens to be Big Endian. All binary data is expected to be in network byte order when it reaches the network. `htons()` translates a short (16 bit) integer from “host byte order”, whatever that happens to be, to network byte order. There is a companion macro, `ntohs()` that translates back from network byte order to host order. Then there is a corresponding pair of macros that do the same translations on long (32 bit) integers.<sup>10</sup>

Now we *bind* `server_socket` to `server_addr` with the `bind()` function. Finally, we create a queue for incoming connection requests with the `listen()` function. A queue length of one should be sufficient in this case because there's only one client that will be connecting to this server.

Now we're ready to *accept* connection requests. The arguments to `accept()` are:

- The socket descriptor.
- A pointer to a `sockaddr` structure that `accept()` will fill in.
- A pointer to an integer that currently holds the length of the structure in argument 2. `accept()` will modify this if the length of the client's address structure is shorter.

`accept()` blocks until a connection request arrives. The return value is a socket descriptor to be used for data transfers to/from this client. In this example the server simply echoes back text strings received from the client until the incoming string begins with “q”.

## The Client

Now look at `netclient.c`. `netclient` determines at run time whether it is connecting to a server locally or across the network. We start by creating a socket and an address structure in the same manner as in the server. Then we *connect* to the server by calling `connect()`. The arguments are:

- The socket descriptor.
- A pointer to the `sockaddr` structure containing the address of the server we want to connect to.
- The length of the `sockaddr` structure.

When `connect()` returns we're ready to transfer data. The client prompts for a text string, writes this string to the socket and waits to read a response. The process terminates when the first character of the input string is “q”.

---

<sup>10</sup> Try to guess the names of the long functions.

### *Try it Out*

The network project includes several make targets. See the top of the makefile for a complete list and then create those targets in Eclipse. Build the client and server targets to run on the host.

Open another terminal window. In this window `cd` to the `network/` directory and execute `./netserve`. Go back to the original window and execute `./netclient`. Type in a few strings and watch what happens. To terminate both processes, enter a string that begins with “q” (“quit” for example).

Both the server and client are built with debugging information on so you can run either or both of them under the debugger.

Next we'll want to run `netserve` on the target with `netclient` running on the host. Build the server target with `SERVER=REMOTE`.

In the terminal window connected to the target (the one running `minicom`), `cd` to the `network/` directory and execute `./netserve`. Back in the original window execute `./netclient remote`.

## **A Remote Thermostat**

Moving on to a more practical example, our thermostat may very well end up in a distributed industrial environment where the current temperature must be reported to a remote monitoring station and set-point and limit need to be remotely settable. Naturally we'll want to do that over a network.

The `network/` directory contains a `monitor.c` file that does essentially the same thing over a network that `monitor.c` in the last chapter did with the serial port. Open `monitor.c`. The first thing to notice is that, unlike `netserve.c`, the process of setting up the net server and establishing a connection has been isolated in its own function. The reason for that will become apparent soon.

Two other features that weren't in our previous monitor program are a case for “?” and an “OK” response to the client acknowledging a parameter change. The “?” is a way to send the current temperature over the network. Actually, we might want to query any of the thermostat's parameters. Think about how you might extend the command protocol to accomplish that.

Build the simulation version of the thermostat with the make target `netthermo`.<sup>11</sup>

Our net thermostat works with the net client that we've already built and tested. You'll need *three* shell windows to run the simulated net thermostat: one for `devices` (start this first), another for `netclient` (start this last), and a third for `thermostat_s`.

---

<sup>11</sup> A “feature” of GNU make prevents us from using the obvious target name, “thermostat.”

When you're ready to move the net thermostat to the target, build `netthermo` with the command `SERVER=REMOTE`

Remember to execute `netclient remote` to get the client talking to the server on the target.

### *Multiple monitor threads*

As the net thermostat is currently built, only one client can be connected to it at a time. It's quite likely that in a real application, multiple clients might want to connect to the thermostat simultaneously. Remember from our discussion of sockets that the server process may choose to spawn a new process or thread in response to a connection request so it can immediately go back and listen for more requests.

In our case we can create a new monitor thread. That's the idea behind the `createServer()` function. It can be turned into a server thread whose job is to spawn a new monitor thread when a connection request comes along. So give it a try.

Here, in broad terms, are the steps required to convert the net thermostat to service multiple clients:

1. Copy `monitor.c` to `multimon.c` and work in that file
2. Recast `createServer()` as a thread function.
3. In the `createThread()` function, change the arguments of the `pthread_create()` call to create the server thread.
4. Go down to the last four lines in `createServer()`, following the comment "Accept a connection". Bracket these lines with an infinite while loop.
5. Following the `printf()` statement, add a call to `pthread_create()` to create a monitor thread.
6. Add a `case 'q':` to the `switch` statement in the monitor thread. 'q' says the client is terminating (quitting) the session. This means the monitor thread should exit.

Now, here are the tricky parts. The `client_socket` will have to be passed to the newly created monitor thread. Where does the thread object come from for each invocation of `pthread_create()`? A simple way might be to estimate the maximum number of clients that would ever want to access the thermostat and then create an array of `pthread_ts` that big. The other, more general, approach is to use dynamic memory allocation, the `malloc()` function.

But the biggest problem is, how do you recover a `pthread_t` object when a monitor thread terminates? The monitor thread itself can't return it because the object is still being used. If we ignore the problem the thermostat will eventually become unresponsive either because all elements of the `pthread_t` array have been used, or we run out of memory because nothing is being returned to the dynamic memory pool.

Here are some thoughts. Associate a flag with each `pthread_t` object whether it is allocated on a fixed array or dynamically allocated. The flag, which is part of a data structure that includes the `pthread_t`, indicates whether its associated `pthread_t` is free, in use, or "pending," that is, able to be freed. The server thread sets this flag to the "in use" state when creating the monitor. The structure should proba-

bly also include the socket number. It is this “meta” `pthread_t` object that gets allocated and passed to the monitor thread.

Now the question is, when can the flag be set to the “free” state? Again, the monitor can’t do it because the `pthread_t` object is still in use until the thread actually exits. Here’s where the third state, PENDING, comes in.

The monitor sets the flag to PENDING just before it terminates. Then we create a separate thread, call it “resource” if you will, that does the following:

1. Wakes up periodically and checks for monitor threads that have set their flags to PENDING
2. Joins the thread
3. Marks the meta `pthread` object free
4. Goes back to sleep

Remember that when one thread joins another, the “joiner” (in this case resource) is blocked until the “jinee” (monitor) terminates. So when resource continues from the `pthread_join()` call, the just-terminated monitor’s `pthread_t` object is guaranteed to be free. The semantics of `pthread_join()` are:  
`int pthread_join(pthread_t thread, void **ret_value)`

It’s quite likely that by the time the resource thread wakes up, the monitor thread has already exited, in which case `pthread_join()` returns immediately with the error code ESRCH meaning that *thread* doesn’t exist.

The `createThread()` function now needs to create the server thread and the resource thread. Likewise, `terminateThread()` needs to cancel and join both of those. And of course, any monitor threads that are running also need to be cancelled. If you don’t feel up to writing this yourself, you may have already noticed that there is an implementation of `multimon.c` in the `home/.working/` directory.

There’s a separate makefile called `Makefile.multi` to build the multiple client versions of `netthermo`. The build command for the Eclipse target is:

```
make -f Makefile.multi
```

Add `SERVER=REMOTE` to build the target version.

So how do you test multiple monitor threads? Simple. Just create multiple shell windows and start `netclient` in each one. When you run `thermostat` under the debugger, you’ll note that a new thread is created each time `netclient` is started.

## Resources

*Linux Network Administrators' Guide*, available from the Linux Documentation Project, [www.tldp.org](http://www.tldp.org). Not just for administrators, this is a quite complete and quite readable tutorial on a wide range of net-working issues.

Comer, Douglas, *Internetworking with TCP/IP, Vols. 1, 2 and 3*, Prentice-Hall. This is the classic reference on TCP/IP. Volume 1 is currently up to its fifth edition dated 2005. Volume 2, coauthored with David L. Stevens, is at the third edition, 1998, and volume 3, also coauthored with Stevens, dates from 1997. Highly recommended if you want to understand the inner workings of the Internet.

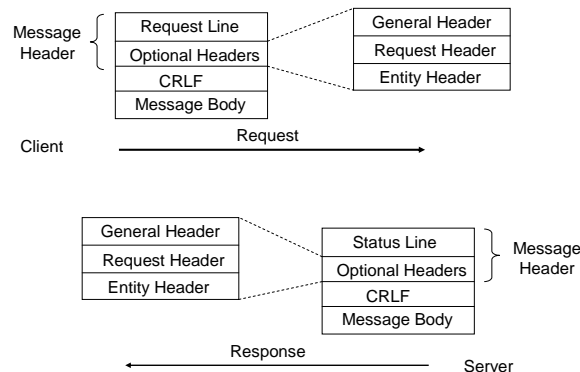
Internet protocols are embodied in a set of documents known as “RFCs”, Request for Comment. The RFCs are now maintained by a small group called the RFC Editor. The entire collection of RFCs, spanning the 30-plus year history of the Internet, is available from [www.rfc-editor.org](http://www.rfc-editor.org). In particular, HTTP is described in RFC 2616.

## 10. An Embedded Web Server

While programming directly at the sockets level is a good introduction to networking, and a good background to have in any case, most real world network communication is done using higher level application protocols. HTTP, the protocol of the World Wide Web is probably the most widely used. No big surprise there. After all, virtually every desktop computer in the world has a web browser. Information rendered in HTTP is accessible to any of these computers with no additional software. So in this chapter, we'll turn our thermostat into a web server.

### Background on HTTP

HTTP is a fairly simple synchronous request/response ASCII protocol over TCP/IP as illustrated in figure 10.1. The client sends a request message consisting of a header and, possibly, a body separated by a blank line. The header includes what the client wants along with some optional information about its capabilities. Each of the protocol elements shown in figure 10.1 is a line of text terminated by CR/LF. The single blank line at the end of the header tells the server to proceed.



**Figure 10.1: HTTP Protocol**

A typical request packet is shown in listing 10.1. The first line starts with a *method token*, in this case GET, telling the server what “method” to use in fulfilling this request. This is followed by the “re-source” that the method acts on, in this case a file name. The server replaces the “/” with the default file index.html. The remainder of the first line says the client supports HTTP version 1.1

The **Host:** header specifies to whom the request is directed, while the **User-Agent:** header identifies who the request is from. Next come several headers specifying what sorts of things this client understands in terms of media types, language, encoding, and character sets. The **Accept:** header line is actually much longer than shown here.



The Keep-Alive: and Connection: headers are artifacts of HTTP version 1.0 and specify whether the connection is “persistent”, or is closed after a single request/response interaction. In version 1.1 persistent connections are the default. Implementing a persistent connection is a non-trivial exercise that’s beyond the scope of our work here. So even though the client wants to keep the connection alive, in our response we’re going to say that the connection is closing.

This example is just a small subset of the headers and parameters available. For our fairly simple embedded server, we can in fact ignore most of them.

```
GET / HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; U; Linux i586; en-US; rv:1.2.1)
Gecko/20030225
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8
Accept-Language: en-us, en;q=0.50
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *,q=0.66
Keep-Alive: 300
Connection: keep-alive
<blank line>
```

### Listing 10.1

## A Simple embedded web server

To make data available via HTTP you need a web server. Creating a web server for an embedded device is not nearly as hard as you might think. That’s because all the rendering, the hard part, is done by the client, the web browser. By and large, all the server has to do is deliver files.

There are a number of small, open source web servers out there. The Resources section lists a few. For our purposes I chose a server called monkey. The primary motivation is that it seemed to be about the simplest and the easiest to build. It is described as a “lightweight and scalable Web Server” that’s been designed “with a strong focus on Embedded devices”.

### *cmake*

The build system for monkey is based on a tool called cmake that is much more powerful and flexible than simple make. It is a compiler-independent set of tools that generates the Makefiles and other components required to build a project. We won’t get into the details of cmake here, the Resources section has the web site location.

Each source directory in the project has a file named `CMakeLists.txt` that provides the data to generate the Makefile for that directory. The monkey project has provided an Autotools-like `configure` script so you can create the configuration without knowing anything about `cmake` itself. `Cmake` puts all of the build files in a single subdirectory of the project, `build/`, making it fairly easy to reconfigure by simply deleting the contents of the `build/` directory.

### *Configuring and Building Monkey*

Monkey is in `target_fs/home/src/monkey-1.6.9`. `cd` to that directory in a shell where the ARM cross tool environment is set. Then execute:

```
./configure --prefix=/home/target_fs --malloc-libc
```

The `--prefix` option is the location of our target file system so `make install` can put the final libraries and executables in the correct location. The `--malloc-lib` option says use the standard C library `malloc()` function rather than monkey's super-duper version, `jemalloc()`. Feel free to try `jemalloc()` if you like. There are a great many other options to `configure`. Execute `./configure -h` to see them all.

Have a look at the Makefile. For each `make` target it simply invokes `make` again changing into the `build/` directory. Execute `make`. There's a minor gotcha' in the `make install` step. `cmake` can't create the directory `/home/target_fs/var/log/monkey` even under `sudo`. That's because the R Pi makes `/var/log` a link to `/var/volatile/log`, which is mounted as a `tmpfs` at boot time and doesn't exist as far as the workstation is concerned. So do the following:

```
sudo mkdir /home/target_fs/var/volatile/log
sudo make install
```

The `install` target puts the executable, `monkey`, in `target_fs/sbin/`, creates and populates `target_fs/etc/monkey/` and puts a set of plug ins in the form of shared libraries into `target_fs/lib/`.

There's another hitch. We had to specify an installation prefix for the build to locate files correctly in the target file system. Unfortunately this got built into monkey and some of its configuration files. As root user edit the following files in `target_fs/etc/monkey/` to remove `/home/target_fs` from various path names:

- `monkey.conf` -- One line beginning with `PidFile`.
- `sites/default` -- Three lines: `DocumentRoot`, `AcessLog`, and `ErrorLog`. Uncomment the last line that begins `Match /cgi-bin/...`
- `plugins.load` -- Everything in this file is commented out. You'll only need to edit it when you want to load the plugins. Later on you'll want to load the CGI plugin.

### *Running Monkey*

Fire up the target and execute:

```
mkdir /var/log/monkey
monkey -c /etc/monkey
```

`var/log` is linked to `var/volatile/log`, which is a temp file system in RAM. That means every time you power up the board it starts empty. Oddly `monkey` doesn't create this directory if it doesn't exist. You only need to make the directory the first time you start `monkey` after rebooting.

The `-c /etc/monkey` option removes `/home/target_fs` from the path to the configuration files. To run `monkey` as a daemon process in the background, add the `-D` option to the command. There are many more options. Execute `monkey -h` to see them all.

On your workstation open a browser and point it at `192.168.15.50:2001`. Of course if you changed the IP address on the board, then use that value. The default port number for `monkey` is 2001 rather than the standard HTTP port 80. If everything went according to plan, you should see the same image as the `monkey-project` home page.

The next step might be to integrate our thermostat into the web browser. That gets into topics like CGI and PHP, which are beyond the scope of this tutorial.

## Resources

Jones, M. Tim, *TCP/IP Application Layer Protocols for Embedded Systems*, Charles River Media, 2002. This covers a wide range of application-level network protocols that can be useful in embedded scenarios.

[https://en.wikipedia.org/wiki/Comparison\\_of\\_web\\_server\\_software](https://en.wikipedia.org/wiki/Comparison_of_web_server_software) -- Short article that summarizes and compares a number of web servers.

<https://cmake.org/> -- The home page for `cmake`

[monkey-project.com/](http://monkey-project.com/) -- The home page for `monkey`.

## 11. The OLED Driver

Our goal in this chapter is to exercise the 160 x 80 OLED display panel on the Matrix CK to create a simple display for our thermostat.

### The Serial Peripheral Interface (SPI) Bus

The OLED panel interfaces to the R Pi through the SPI bus. Much like the I2C bus introduced in chapter 6, the Serial Peripheral Interface (SPI, pronounced “spy”) is a bit-serial interconnect for connecting peripherals to microcontrollers. Whereas I2C is a 2-wire bus, SPI is either three or four wires. The four wires are:

- CLK – the clock, which can be configured for either rising edge or falling edge
- MOSI – Master Out, Slave In for transferring data from the microcontroller to the peripheral
- MISO – Master In, Slave Out for transferring data from the peripheral to the microcontroller
- CS – chip select. The other three wires can be bussed to multiple slave devices. Each device gets a unique chip select.

The 3-wire mode uses MOSI as a bi-directional data line renamed MIMO (Master In, Master Out).

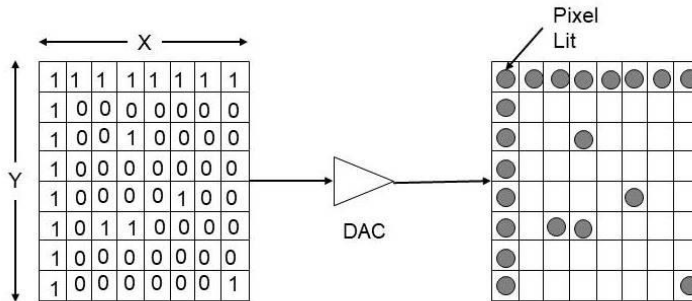
The Raspberry PI has one SPI controller with two chip selects. SPI is enabled in `config.txt` so the driver is loaded at boot time.

The SPI bus is accessed through a conventional Linux character device driver. Everything, even data transfer, is done through the `ioctl()` call. Information about the system's SPI devices is available at `/sys/bus/spi`. The module `spi.c` in the Matrix CK library provides functions to initialize the bus, set the operating mode, and read and write data. We won't need the functions in `spi.c` because the display panel's device driver handles all of the SPI interaction.

### The Video Frame Buffer

Virtually all graphic display devices these days are based on the notion of a *frame buffer* a chunk of RAM that is mapped to individual pixels on the display, typically either two or three bytes per pixel. Figure 11-1 illustrates the idea with a simple one bit per pixel buffer. Wherever there's a 1 in the RAM, the corresponding pixel is lit.

The Matrix CK panel uses two bytes, or 16 bits, per pixel organized as five bits for each color and the remaining bit to represent transparency.



**Figure 11-1: Frame Buffer**

Frame buffer devices are assigned to major device number 29. There's already one on the R Pi for the HDMI port, `/dev/fb0`.

### *The Frame Buffer Driver*

Starting up the frame buffer driver for our OLED panel requires something of a "magic incantation":

```
modprobe fbftt_device name=sainsmart18 cs=0 speed=16000000
gpios=reset:22,dc:7 rotate=270
```

All one line of course. `modprobe` is a utility that loads a kernel module and works out any dependencies to load the dependent modules in the correct order. Before executing this command, execute `lsmod` to see what kernel modules are currently loaded. `fbftt_device` appears to be a "meta driver" that manages a large number of drivers for small frame buffer devices that are typical of embedded systems. I'll be honest, the above command came about through Google searches and a certain amount of trial and error. It turns out that the `sainsmart18` is a device that uses the same frame buffer chip as ours, the ST7735R. We're saying that it's connected to SPI chip select 0 and it uses specific GPIO lines for `reset` and `dc`. By default the panel has a portrait orientation. The `rotate` argument changes it to landscape. You might want to put that command into a script file so you don't have to remember the details later on.

`fbftt_device` loads the driver for the ST7735R and a number of other modules. Do an `lsmod` again after running the above command and you should see seven additional modules. Two of these, `fbftt_device` and `fb_st7735r`, come from the directory `linux/drivers/staging/fbftt`. The staging tree holds "drivers and filesystems that are not ready to be merged into the main portion of the Linux kernel tree at this point in time for various technical reasons" This is done to make these modules available for use, but with a caveat. When you load a module from the staging tree, you get the following warning:

```
<module_name>: module is from the staging directory, the quality is unknown, you
have been warned
```

Furthermore, the kernel's "tainted" flag is set. If your kernel subsequently crashes and you send a core dump to the kernel maintainers, they will see that the tainted flag is set and will promptly put your core dump in the bit bucket.

Getting back to the frame buffer, access to a frame buffer is gained by mapping its memory space into user space using the `mmap()` system call. Have a look at the `OLEDInit()` function in the `oled.c` module in the `pi-lib/` directory. First you open a file to the frame buffer device, then get basic information using `ioctl()` calls. Finally you call `mmap()`, which returns a virtual user space address of the frame buffer's memory.

In our case there's an additional complication. The display panel is 80 pixels by 160 pixels, but the ST7735R controller chip has a short dimension of 128 pixels. Furthermore, the panel appears to be plopped down right in the middle of the memory space, requiring an offset that depends on whether the panel is set for landscape or portrait orientation. The values you see in `oled.c` were determined by experimentation. Note also that when advancing to the next line, you add 128 pixels (256 bytes) to the address rather than 80 pixels.

## Exercising the Display

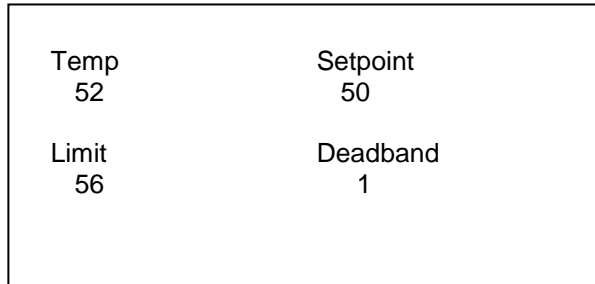
`pi-lib/oled.c` contains a set of simple functions for controlling the display. These include:

- `int OLEDInit (void)` – initialize the LCD screen. Opens a file to the frame buffer device, gets basic information about the display and maps the frame buffer to user space. Returns 0 if successful and -1 if anything fails.
- `void OLEDDelInit (void)` – just unmap the frame buffer.
- `void OLEDDrawChar (int x, int y, char c, int textColor, in bgColor)` – draws a 5x7 character `c` with the upper left corner at coordinates `x, y`. The character is drawn in `textColor` on `bgColor`.
- `int OLEDDrawString (int row, int col, char *string, int textColor)` – writes a string at the indicated row and col. The string is drawn in `textColor` on a black background. Returns the number of characters written.
- `void OLEDWriteNumber (int row, int col, int value, int width)` – writes a decimal value at the indicated row and col with the indicated width.
- `void OLEDFillRect (int x, int y, int w, int h, int color)` – fills a rectangle with `color` starting at `x, y` and extending `w` pixels wide and `h` pixels high.
- `void OLEDClearScreen (void)` – clears the entire screen.

## Thermostat display

With that background, let's add a display to our thermostat that looks something like figure 11.2. Have a go at it using the net version of the thermostat. Then when any of the parameters is changed from the network, you'll see it updated on the display.

And of course feel free to add new functionality to the `oled` module in the library as the need arises.



Temp 52	Setpoint 50
Limit 56	Deadband 1

**Figure 11.2: thermostat LCD screen**

## ncurses library

There's actually a higher level solution to this problem that we encountered briefly back in chapter 6 in the discussion of simulation. The `devices` program makes use of the `ncurses` library to move the cursor around in a window. `ncurses` can also do lots of other things like changing colors and setting fonts.

There are a couple of reasons for not using `ncurses` here. Although it's part of virtually every x86 Linux distribution, it is not by default part of the ARM toolset that we're using. It's a useful tool and I encourage you to check it out, yet nevertheless, understanding the lower level API is still useful.

`ncurses` is part of the GNU library. See Resources below for the web location.

## Qt Extended

If you really want to get fancy, Qt Extended, formerly known as Qtopia, is a popular embedded graphical development environment. It's not part of the minimal file system we're using now, but if you install the Raspbian distribution as mentioned in chapter 15, you'll have Qtopia. Graphical development is beyond the scope of this particular tutorial.

## Resources

[www.gnu.org/software/ncurses/ncurses.html](http://www.gnu.org/software/ncurses/ncurses.html) -- home page of the `ncurses` project at the Free Software Foundation

## 12. Configuring and Building the Kernel

One of the neatest things about Linux is that you have the source code. You're free to do whatever you want with it. Most of us have no intention, or need, to dive in and directly hack the kernel sources. But access to the source code does mean that the kernel is highly configurable. That is, you can build a kernel that precisely matches the requirements, or limitations, of your target system.

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

### The kernel source tree

The source code for Linux kernel version 4.14.94 is located under your home directory at `linux-rpi-4.19.y`<sup>12</sup>.

#### *"Upstream" vs. "Downstream" kernels*

The upstream version of the kernel is maintained by Linus and his associates at [kernel.org](http://kernel.org). A new release of the upstream kernel occurs about every six weeks to two months. [kernel.org](http://kernel.org) maintains every kernel version since Linus' first release in 1991. This is where kernel development occurs and the latest version at [kernel.org](http://kernel.org) could be termed the "bleeding edge".

From time to time, commercial Linux vendors and Open Source projects such as Red Hat, Fedora, and so on, will update their own products. In doing so, they will identify a version of the upstream kernel that seems appropriate at the time. They will then add their own modifications and subject the kernel to intensive testing and integration procedures. The result is a *downstream* kernel. Remember that the kernel is Open Source and so anything that goes into the kernel must be made publically available. Generally, a downstream kernel is made available as a patch set to the corresponding upstream kernel.

In the embedded world there are a few downstream distributions of interest that we won't cover here. The most popular of these of course is Android. Others include OpenEmbedded, Buildroot, and Linaro. The latter is specifically targeted at ARM processors. Our kernel came from the Raspberry Pi layer of the Yocto Project.

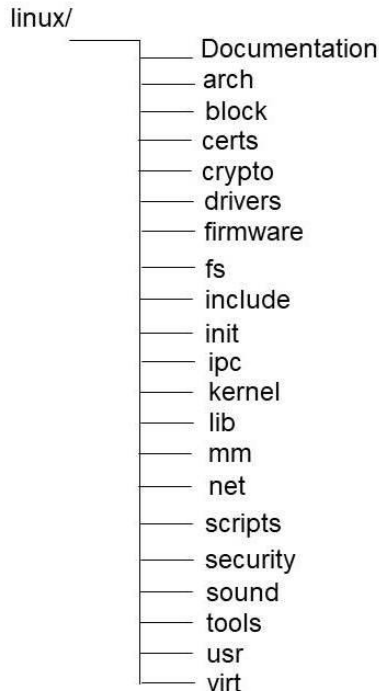
There are over 72,000 files in this kernel source tree taking up a little over 900 MB of disk space. Needless to say, it is critical that these files be organized in some sensible fashion so as to be able to

---

<sup>12</sup> The kernel version may change with subsequent releases and thus this directory name will also change.



find what you're looking for. Figure 12.1 shows the top-level layout of the kernel source. With minor variations, this layout has been the standard for quite some time.



**Figure 12.1: Kernel source tree layout**

Most of these top-level directories have one or more sub-directories under them.

Documentation – just what it says. You'll find a wealth of information here about all kinds of obscure kernel issues. Some of it tends to be dated because it was written when a particular feature was first introduced, but not kept up to date as the feature evolved. Nevertheless, this is a good place to start when trying to puzzle out some kernel problem.

arch – this is where virtually all of the architecture-dependent code is located. Under `arch/` are sub-directories for each of the architectures that the kernel supports. In 3.6 there are 27 of them. Think of this as the “board support package” (BSP), the body of code that ties the kernel to a specific hardware implementation.

block – code for the block subsystem that attempts to optimize access to block structured devices.

certs – code having to do with secure signing of modules

crypto – code related to encryption.

drivers – this is the largest subdirectory, accounting for roughly a quarter of the kernel's source files.

Linux supports literally thousands of the devices and all the drivers are here.

fs – file systems. Linux supports over 60 different file systems. Each of those file systems has a subdirectory here.

include – kernel header files. Kernel API header files are found in the `linux/` subdirectory.

init – initialization code including the kernel's `main()` function. Yes, the kernel has a `main()` function.

ipc – System V inter-process communication mechanisms.

kernel – the core kernel code including such items as the scheduler, timers, interrupt handling, and the kernel's own synchronization mechanisms.

lib – library code.

mm – memory management code.

net – the networking subsystem.

scripts – support scripts for configuring and building the kernel.

security – code related to Security Enhanced (SE) Linux.

sound – the Linux sound subsystem.

tools – a collection of various tools and utilities

usr – an odd collection of files

virt – a virtualization project similar to VirtualBox

## Configuring the kernel

The `linux/` directory contains a standard `Makefile` with a very large number of make targets. The process of building a kernel begins by invoking one of the three make targets that carry out the configuration process.

By default `Makefile` configures and builds the kernel for the architecture on which it's running, which in the vast majority of cases is some variant of x86. In order to configure and build the kernel for another architecture, ARM to be specific, we have to specify a couple of environment variables:

```
ARCH = arm
CROSS_COMPILE = arm-poky-linux-gnueabi-
```

Note that "arm" as used here is the name of a subdirectory under `arch/`.

The environment setting script in `/opt/arm` sets these variables for us. However, something in that script interferes with the configuration target that we're about to invoke and so you need to run the configuration phase in a shell that doesn't have the cross environment set.

If you're operating in an environment that doesn't predefine these variables, you can pass them in on the command line like this:

```
make ARCH=arm CROSS_COMPILE=arm-poky-linux-gnueabi- xconfig
```

These variables also show up in the makefile, so another way to define them is to edit the makefile.

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. There are a couple of ways to create an initial `.config` file. Execute `make ARCH=arm help` in the `linux/` directory. This lists all of the targets the makefile can build. In particular notice the large number of targets with names of the form `<device_name>_defconfig`. These create default configuration files for a range of ARM-based boards.

For our purposes the correct make target is `bcm2835_defconfig`. The Raspberry Pi 3 is based on the Broadcom BCM2835 SoC. So execute `make ARCH=arm bcm2835_defconfig`.

Now execute `make ARCH=arm xconfig`.

### *make xconfig*

After a bit of program and file building, the X Windows-based menu shown in figure 12.2 appears. Of course you must be running an X Windows environment for this to work. You also must have the Qt graphics library and GUI toolkit installed. The package to install is `qt-devel`.

If Qt is not installed and you don't want to bother with it, there's an alternative target called `make gconfig` that uses the GTK graphics libraries that are usually available by default. This tool is very similar to `xconfig`, but in my estimation is not quite as slick. As a last resort, you can use `make menuconfig`, a pseudo-graphic program based on the `ncurses` library. Interestingly, many true kernel hackers prefer `menuconfig`, probably because it doesn't require X Windows.

As shown here, the screen is 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.

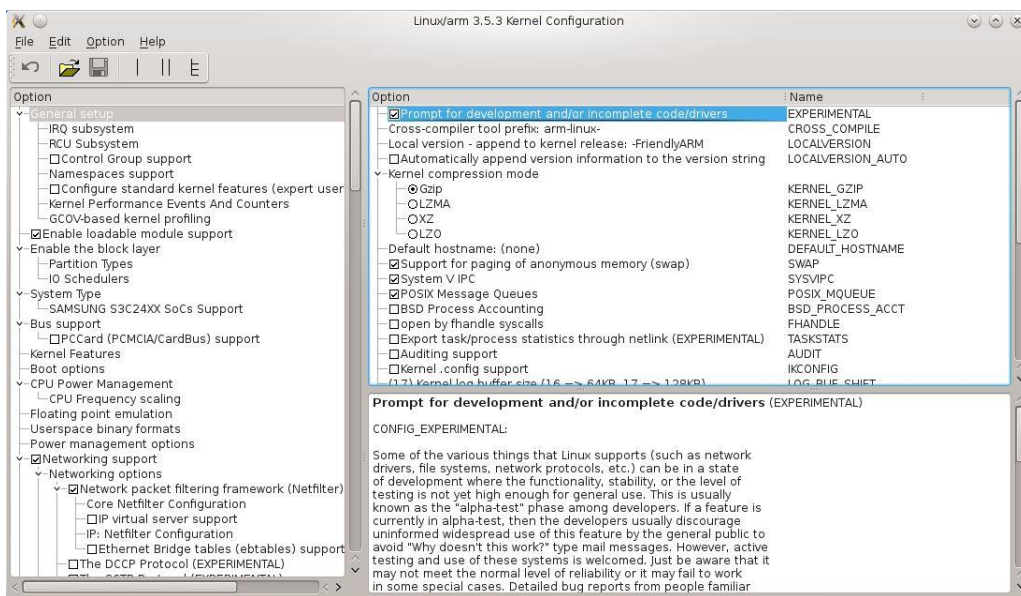


Figure 12.2: xconfig menu

Most of the options are presented as check boxes. Clicking on the box alternates between checked (selected) and unchecked (not selected). Some options have a third value, a dot, to indicate that the feature may be built as a *kernel module*.

Many configuration options are dependent on other options. If the dependencies aren't met, you don't get to see the dependent options. As an example, scroll down to **Bus support**. There's only one item in this category, **PCI support**, and it's not selected. Select it. Suddenly a whole bunch of other options show up.

You shouldn't need to make any changes at this point, but browse around to get a feel for the range of configuration options that are available.

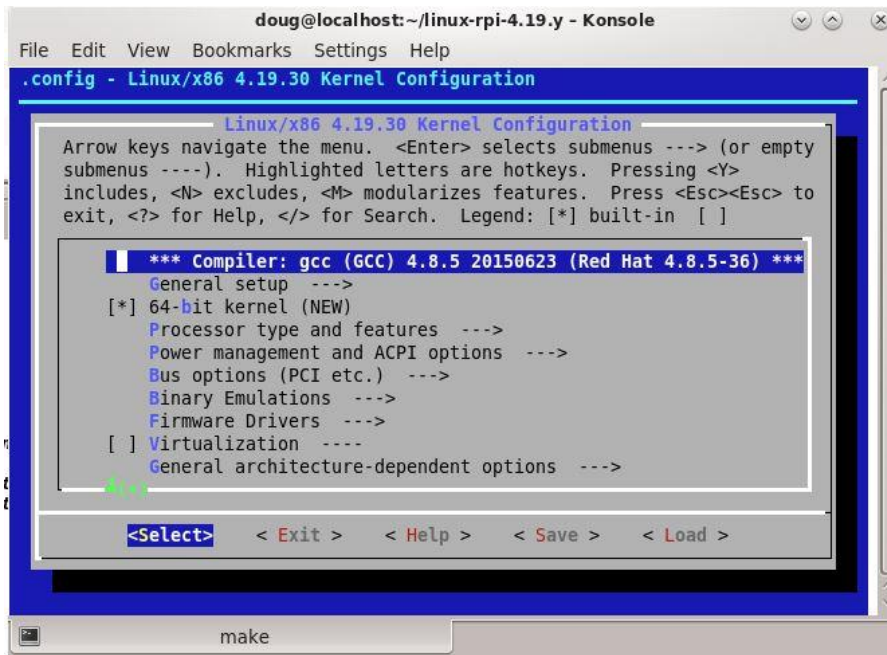
The xconfig menu has a number of interesting options of its own listed under the Option menu. Turning on **Show Name** lists the names of all the configuration variables as they appear in the `.config` file. **Show Range** displays the range of possible values for all variables, most of which are either yes or no. **Show Data** displays the current value, which I find to be somewhat redundant with **Show Range**, although it's useful for text and numeric options. **Show All Options** displays all configuration options including those that would otherwise be invisible because their dependencies haven't been met.

When you're finished, exit from xconfig, replying negatively to the query about saving changes.

## *make menuconfig*

Although `make xconfig` is my favorite configuration tool, I would be remiss if I didn't at least show `make menuconfig`. Figure 12.3 shows the top level menu. There is no mouse support in the curses library, so you use the up and down arrow keys to navigate through the menu and the right and left arrow keys to select among the commands on the bottom.

There's a handy help menu at the top. To change the value of a variable, type space. Entries followed by `---` point to submenus.



**Figure 12.3: make menuconfig**

## Building the kernel

Now move to a shell that has the cross environment set. In the `linux/` directory, execute:

```
make
```

This command builds an executable kernel image. Not surprisingly, this takes a while. The build process for the kernel is recursive. Virtually every subdirectory under `linux/` has its own `Makefile` that deals with the source files in that directory.

This is a good time to take a break and get reacquainted with your family. When the build is finished, you'll find two new files in `linux/arch/arm/boot/`, `Image` and `zImage`. `Image` is the executable and `zImage` is a compressed version with its own decompressor. Note that `zImage` is about half the size of `Image`.

Congratulations! You've just successfully built a Linux kernel. Out of the seven billion people on this planet, probably no more than a few thousand have actually done this<sup>13</sup>. If you've never built a kernel before, go ahead and pat yourself on the back!

### *make modules\_install*

There's one more step in the build process. In addition to the kernel executable, we built a bunch of *kernel loadable modules*. These are run time extensions of the kernel that can be loaded as needed. Most device drivers are implemented as kernel loadable modules.

These modules need to be moved to a specific location in the target file system, specifically `/lib/modules/4.14.94-v7+`. Execute:

```
sudo make modules_install INSTALL_MOD_PATH=/home/target_fs
```

## Testing the New Kernel

Now that we have a new kernel, how do we test it? First of all we have to get the new kernel image into the boot partition of the SD card. The easy way to do that is the `scp` (secure copy) command. In a shell that's either running `minicom` or is `ssh` connected to the target execute:

```
mount /dev/mmcblk0p1 /boot
```

In a workstation shell, `cd` to `linux/` and execute:

```
scp arch/arm/boot/zImage root@a92.168.15.50:/boot
```

Now open `/boot/config.txt` with `vi`. Find the line `kernel=""`. Replace the `""` with `zImage`. Save the file.

**Important!** Any time you make a change to the SD card from the R Pi itself, be sure to execute `sync` before turning off power. Any changes to the SD card file system are maintained in cache until you either unmount the file system or execute `sync`.

Now toggle the power to the board or just enter the command `reboot`. When the board comes back up execute `uname -a` and you should see that it's the kernel version you just built.

### *Next Step*

The default configuration for the R Pi has far too many features enabled IMHO. It builds over 1500 modules. See what all you might be able to remove.

---

<sup>13</sup> Or, to be honest, have had need to. But as an embedded Linux developer, you will no doubt be building lots of kernels.

## Resources

[www.kernel.org](http://www.kernel.org) – This is where you get “official” upstream kernel source code. When applying patches, it’s generally a good idea to start with a “virgin” kernel from kernel.org because that’s generally what the patch creator did.

<http://lxr.linux.no> – This is one of the most fascinating and useful Linux web sites I’ve encountered. It is the entire kernel source tree cross-referenced and hyperlinked. Virtually every version of the kernel is included. It shows where symbols are defined and where they’re used. You can easily trace through a hierarchical stack of function calls. [lxr.linux.no](http://lxr.linux.no) is often off line. Fortunately, lxr itself is an open source project and others have established their own lxr repositories. One of the best is [lxr.missinglinkelectronics.com](http://lxr.missinglinkelectronics.com).

For more details on the process of configuring and building a kernel, look at the files in `linux/Documentation/kbuild`. Additionally, the following HOW-TOs at [www.tldp.org](http://www.tldp.org) may be of interest:

- |              |   |
|--------------|---|
| Config-HOWTO | This HOWTO is primarily concerned with how you configure your system once it’s built. |
| Kernel-HOWTO | Provides additional information on the topics covered in this chapter and a lot more. |

## 13. BusyBox

Very often the biggest problem in an embedded environment is the lack of resources, specifically memory and storage space. As you've no doubt observed, either in the course of reading this guide, or from other experience, Linux is *big*! The kernel itself is usually several megabytes, and then there are all the attendant utility programs that have to go somewhere. A minimum Red Hat or Debian installation comes out to several gigabytes of file storage, unacceptable in many embedded environments.

BusyBox 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 others 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 utilities require the same set of “helper” functionality for things like parsing command strings. Rather than duplicating these functions in dozens of files, BusyBox implements them exactly once.

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

The BusyBox project began in 1996 with the goal of putting a complete Linux system on a single floppy disk that could serve as a *rescue disk* or an installer for the Debian Linux distribution. Subsequently, embedded developers figured out that this was an obvious way to reduce the Linux footprint in resource-constrained embedded environments. So the project grew well beyond its Debian roots and today BusyBox is a part of almost every commercial embedded Linux offering, and is in fact incorporated into E.L.L.K.

BusyBox has been called the “Swiss army knife” of embedded Linux because, like the knife, it's an all-purpose tool. Technically, the developers refer to it as a “multi-call binary”, meaning that the program is invoked in multiple ways to execute different commands. This is done with symbolic links.

Take a look at the `/bin` directory in the target file system. There are relatively few “real” files, one of which is `busybox.nosuid`. Most everything else is a link to the executable `busybox.nosuid`. 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.



There are actually two versions of the BusyBox executable, `busybox.nosuid` and `busybox.suid`. The latter has the SUID permission bit set and allows normal users to run commands as root.

BusyBox is highly modular and configurable. While it is capable of implementing well over 300 shell commands, by no means are you required to have all of them in your system. While many of these are no doubt useful for developing and debugging, they're probably not necessary in a production device. The configuration process lets you select exactly what you need from a wide range of options.

BusyBox is installed in your workstation home directory as `busybox-1.29.2/`. As with the Linux kernel, the default configuration file is called `.config` and doesn't exist initially. Execute `make help` in the BusyBox directory to see what options are available for creating `.config`. Probably the easiest thing to do is execute `make defconfig`, which builds the largest "generic configuration".

BusyBox uses the same basic configuration mechanism as the kernel. So you can execute `make xconfig` to get the configuration menu<sup>14</sup>. There are two major categories of configuration options – BusyBox settings and Applets, of which there are several sub-categories. In particular, check out "Build Options" under "Busybox Settings" (figure 13.1). We have to set a couple of options here because BusyBox build doesn't pay attention to the environment our cross tool chain script set. Execute the following command in a shell with the cross tool environment set:

```
echo $CC
```

There are basically three parts to this string. The first part is the beginning that starts with `arm` and ends with the dash just before `gcc`. Copy this to the **Build Options -> Cross Compiler prefix** option. Remember that you can directly copy strings out of the shell window with `Ctrl+Shift+C`. The next part, up to `-sysroot`, is a set of flags to be passed to the GCC compiler. This substring should be entered as the value for **Additional CFLAGS**. The remainder of the string after `-sysroot=` points to a file system where the tool chain can find header files and libraries. Enter this in the **Path to sysroot**:

Then take a look at **Installation Options > BusyBox Installation prefix**: to see where `make install` puts everything. It's currently set to `__install`, which means everything gets installed under `busybox-1_27_2/` so you can see what it is. In practice you might want to set this to `/home/target_fs` so that everything gets installed directly in the target file system.

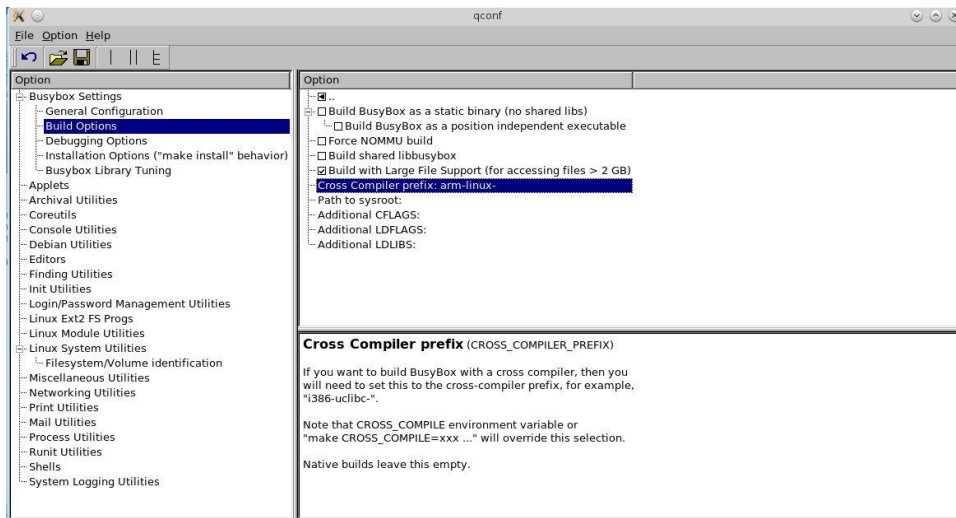
The remaining categories are the individual commands that can be built into the package.

Browse through the various categories to get a better feel for what's there. You'll find that help is generally readable and useful. Decide what you need and what you don't need.

---

<sup>14</sup> When you installed `qt-devel` to run `xconfig` for the kernel, you probably got Qt version 4. Unfortunately, BusyBox only works with version 3, so you'll need to install `qt3-devel`. Or fall back on `make menuconfig`.

When you're finished, exit from `xconfig`, saving the file, and execute `make`. This will take a while. Following the `make` step you'll find two new files in your `busybox-1_27_2 /` directory: the `busybox` executable, and a text file called `busybox.links`. The latter contains the necessary information for creating symbolic links for all the commands you turned on in BusyBox. Execute `make install` and browse the contents of `busybox-1_27_2 /__install. bin/` has the BusyBox executable along with symbolic links for all the commands that are normally found in `/bin`. Likewise `sbin/` has links for all the commands that normally reside in `/sbin`, and `usr/` has subdirectories `bin/` and `sbin/` with links for commands that usually reside there.



**Figure 13.1: BusyBox configuration**

You would then copy these directories to the corresponding directories under `target_fs/`. You can also specify the installation prefix on the command line as follows:

```
make CONFIG_PREFIX=../target_fs install
```

which would create the executable and links directly in the `target_fs/` directory.

## Resources

[www.busybox.net](http://www.busybox.net) – the official site for the BusyBox project

## 14. The Bootloader

In this chapter we'll look at how the R Pi boots up and also explore a popular open source bootloader, u-boot.

### Raspberry Pi Boot Process

The Raspberry Pi bootloader consists of three stages. When power is first turned on, the ARM core is off, and the GPU (Graphics Processing Unit) core is on. At this point the SDRAM is disabled. The GPU starts executing the first stage bootloader, which is stored in ROM on the SoC. The first stage bootloader mounts the SD card, and loads the second stage bootloader (`bootcode.bin`) into the L2 cache, and runs it.

Stage 2 activates the SDRAM. It also understands ELF binaries, and loads Stage 3, `start.elf`, from the SD card. Stage 3 is also known as GPU firmware, and this is what causes the rainbow splash (4 pixels that are blown up to full-screen) to be displayed.

Stage 3 reads the firmware configuration file `config.txt` (if any), and then proceeds to split the RAM between GPU and ARM CPU. Stage 3 also loads a file called `cmdline.txt` if it exists, and will pass its contents as the command line to the kernel. Stage 3 finally enables the ARM CPU and loads the kernel image (`kernel7.img` by default, configurable via `config.txt`), which is executed on the ARM CPU.

`start.elf` is actually a complete proprietary operating system known as VCOS (VideoCore OS).

### U-boot

Desktop Linux systems have a boot loader in addition to the BIOS. These days it's usually GRUB, the GRand Unified Bootloader, or GRUB2. But because the BIOS does all the heavy lifting of initializing the hardware, GRUB itself does little more than load and start the operating system. If you're building an embedded system based on conventional x86 PC hardware, GRUB is probably the way to go.

But ARM-based boards typically don't have a BIOS and therefore require a bootloader. The open source u-boot project has become the de facto bootloader of choice in the embedded Linux world.

### *Background*

U-boot began as a Power PC boot loader named 8xxROM written by Magnus Damm. Wolfgang Denk subsequently moved the project to Source Forge and renamed it PPCBoot. The source code was briefly forked into a product called ARMBoot by Sysgo GmbH. The name was changed to U-boot when the ARMBoot fork was merged back into the PPCBoot tree. Today, U-boot supports roughly a dozen architectures and over 1000 different boards.

The development of U-boot is closely tied to Linux with which it shares some header files. Some of the source code originated in the kernel source tree.

U-boot supports an extensive command set that not only facilitates booting, but also manages flash memory, downloads files over the network, and more. Appendix B details the command set. The command set is augmented by environment variables and a scripting language. Environment variables can contain u-boot shell commands that can then be executed with the `run` command as in:

```
run <variable_name>
```

### *Configuring and Building U-boot*

cd to the `u-boot/` directory in your home directory. There's a very extensive `README` file in the top-level directory. There are additional `README` files in the `doc/` directory that primarily describe features of specific boards.

Current u-boot implementations support the same Kconfig configuration mechanism as the Linux kernel and BusyBox. Again, we have to start with a default configuration. Have a look at the `configs/` directory. There are over 1300 `*_defconfig` files. For our purposes the correct one is `rpi_3_32b_defconfig`. So run:

```
make rpi_3_32b_defconfig
```

Now run `make xconfig`. Note among other things that you can select which shell commands are included in the build. During development you probably want just about everything, but in production you could probably get by with substantially less.

We're going to make a couple specific changes. About 20 lines down the first page of the navigation window is an entry that reads **Enable a default value for bootcmd**. Under that is the actual value of the `bootcmd` variable. `bootcmd` is the variable that is executed if the auto boot sequence is not interrupted. The current value is `run distro_bootcmd`, which simply executes another variable. Change it to the following:

```
run bootkernel
```

Just below that is an entry that reads **Enable preboot** that is currently not selected. Select it and the preboot default value: appears, which is currently empty. Enter the following string for that:

```
usb start; load mmc 0:1 $loadaddr uEnv.txt; env import -t $loadaddr $filesize
```

This is a set of three commands separated by semicolons. Here's what's happening:

- `usb start` – From the current definition of preboot that we'll have to remove
- `load mmc 0:1 $loadaddr uEnv.txt` – Load the file `uEnv.txt` from the SD card into RAM. This file is another set of environment variables. The `$` dereferences a variable causing its name to be replaced by its value.

- `env import -t $loadaddr $filesize` – Import the just-loaded file into the u-boot environment. The `filesize` variable is automatically set each time a file is read.

Exit `xconfig` saving the configuration. Unfortunately, the variable `CONFIG_PREBOOT` is defined in a header file and the compiler complains because the definitions differ. It's not an error, just a warning, but is nevertheless annoying<sup>15</sup>. Edit the file `u-boot/include/configs/rpi.h`. `CONFIG_PREBOOT` is defined at line 76. Comment out that line.

Run `make` in a shell window where the cross-development environment is defined. `make` creates nine new files. Of specific interest to us is `u-boot.bin`. On the target, mount `/dev/mmc0p1` at `/boot` if it isn't still mounted from chapter 12. Use the `scp` command to copy `u-boot.bin` to `/boot` on the target.

Open `/boot/config.txt` on the target with `vi`. Find the line `kernel=zImage` that you modified in chapter 12 and replace `zImage` with `u-boot.bin`. So what's happening is that the proprietary R Pi bootloader is doing its thing as normal and then we're invoking u-boot instead of the kernel. This is to some extent redundant, but the objective is to learn something about u-boot.

### *Running U-boot*

Save the file and be sure to execute `sync`. Now repower the target board. After a short delay while the R Pi bootloader is executing, you'll see a couple of messages from u-boot about USB devices and then a message, Hit any key to stop autoboot: 2. You'll have two seconds to hit a key on the keyboard to stop the autoboot sequence and get the `U-Boot>` prompt.

Type `help` to get a list of the commands in this particular u-boot. To get more details on a command, enter `help <command_name>`. Now type `printenv`. This lists the entire u-boot environment. Over the years u-boot environments have gotten increasing large and complex as many development boards including the R Pi have developed multiple booting mechanisms. The u-boot environment attempts to determine which booting mechanisms are present on the board and picks the first valid one it finds. To list a specific environment variable, enter `printenv <variable_name>`.

The u-boot shell includes the usual array of control structures: `if...then...else`, `for` and `while` loops.

Have a look at `/boot/uEnv.txt`, in particular the environment variable `bootkernel`. This is executed by `bootcmd` as the auto-boot sequence. It in turn runs a variable named `loadkernel` that loads the kernel from the SD card to RAM. The name of the kernel image is defined here, the FDT file is defined in the base environment. It then invokes `bootz` passing in the RAM addresses of the kernel image and the *device tree*. The “-“ in the middle is an optional third parameter that is the address of an initial RAM disk image, which isn't used very often.

---

<sup>15</sup> I consider it “tacky” to define a macro in a header file that also appears in the configuration menu.

Then there's the `bootargs` variable, a string that is passed to the kernel when it starts up. This conveys information such as the kernel's console, the location and type of the root file system, and in this case, the fixed IP address of the board.

Execute `boot`, an alias for `run bootcmd`, and the kernel should boot.

## Device Trees

One of the biggest problems with porting an operating system such as Linux to a new platform is describing the hardware. That's because the hardware description is scattered over perhaps several dozen or so device drivers, the kernel, and the boot loader just to name a few. The ultimate result is that these various pieces of software become unique to each platform, the number of configuration options grows and every board requires a unique kernel image.

There have been a number of approaches to addressing this problem. The notion of a "board support package" or BSP attempts to gather all of the hardware dependent code in a few files in one place. It could be argued that the entire `arch/` subtree of the Linux kernel source tree is a gigantic board support package.

Take a look at the `arch/arm/` subtree of the kernel. In there you'll find a large number of directories of the form `mach-*` and `plat-*`, presumably short for "machine" and "platform" respectively. Most of the files in these directories provide configuration information for a specific implementation of the ARM architecture. And of course, each implementation describes its configuration differently.

Wouldn't it be nice to have a single language that could be used to unambiguously describe the hardware of a computer system? That's the premise, and promise, of device trees.

The peripheral devices in a system can be characterized along a number of dimensions. There are, for example, character vs. block devices. There are memory mapped devices and those that connect through an external bus such as I2C or USB. Then there are *platform* devices and *discoverable* devices.

Discoverable devices are those that live on external busses such as PCI and USB that can tell the system what they are and how they are configured. That is, they can be "discovered" by the kernel. Having identified a device, it's a fairly simple matter to load the corresponding driver, which then interrogates the device to determine its precise configuration.

Platform devices, on the other hand, lack any mechanism to identify themselves. SoC (System on Chip) implementations such as the BCMxxxx series are rife with these platform devices—system clocks, interrupt controllers, GPIO, serial ports, to name a few. The device tree mechanism is particularly useful for managing platform devices.

The device tree concept evolved in the PowerPC branch of the kernel and that's where it seems to be used the most. In fact it is now a requirement that all PowerPC platforms pass a device tree to the ker-

nel at boot time. The text representation of a device tree is a file with the extension `.dts`. These `.dts` files are typically found in the kernel source tree at `arch/$ARCH/boot/dts`.

A device tree is a hierarchical data structure that describes the collection of devices and interconnecting busses of a computer system. It is organized as nodes that begin at a root represented by `/"` just like the root file system. Every node has a name and consists of “properties” that are name-value pairs. It may also contain “child” nodes.

Listing 14-1 is a sample device tree taken from the `devicetree.org` website. It doesn't do anything beyond illustrating the structure. Here we have two nodes named `node1` and `node2`. `node1` has two child nodes and `node2` has one child. Properties are represented by `name=value`. Values can be strings, lists of strings, one or more numeric values enclosed by square brackets, or one or more “cells” enclosed in angle brackets. The value can also be empty if the property conveys a Boolean value by its presence or absence.

```
/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
        };
    };
};
```

### Listing 14-1 Sample Device Tree

The R Pi uses a device tree to describe its hardware. The device tree source files (`*.dts`) are found in `linux/arch/arm/boot/dts`. The one for the R Pi 3 is `bcm2837-rpi-3-b.dts`. This file doesn't define much, but it includes a couple of other files (note that included device tree files get the extension `.dtsi`).

`am33xx.dtsi` defines quite a bit of the system. Note in particular that pretty much every node has a *compatible* property. This is what links a node to the device driver that manages it.

The device tree source code is compiled into a more compact form called a *device tree blob* (\*.dtb), also known as a *flattened device tree* (fdt). The device tree compiler itself is actually part of the kernel source tree. The resulting `bcm2837-rpi-3-b.dtb` is stored in the same boot partition along with everything else related to booting. As the kernel loads device drivers, the drivers interrogate the device tree to determine the specific properties of their respective devices.

The device tree is visible from u-boot using the `fdt` commands. Type `help fdt` to see the commands. Among other things, you can list the fdt, create new nodes and properties, and change property values. Then once the kernel is booted, the device tree is visible at `/proc/device-tree`. This tree is read-only.

## Booting over the Network

With u-boot in our bag of tools, we have another, more convenient way, to test a new kernel. We can load it over the network from the workstation using TFTP (trivial file transfer protocol). TFTP is exactly that: a simple protocol for moving files between a server and a client. Because of its simple design, TFTP can be implemented in a relatively small amount of code and this makes it useful for things like booting.

### Setting up the TFTP Server

The workstation will serve as our TFTP server just as it provides the NFS server function. Some distributions don't install the TFTP server by default and so you may have to install it yourself along with the `xinetd` daemon server. Just as we did earlier with minicom, Fedora and CentOS users can execute:

```
yum install tftp-server xinetd
```

While Ubuntu and Debian users can execute:

```
apt-get install tftp-server xinetd
```

The next step is to edit the file `/etc/xinetd.d/tftp` as root user and change it as follows:

```
disable = no
```

instead of:

```
disable = yes
```

Notice also in this file the line:

```
server_args = -s /var/lib/tftpboot
```

This is the default directory from which files will be downloaded to the target. If this directory doesn't exist on your workstation, create it and make sure that everyone has write access.

Now execute the following commands start and enable the `xinetd` server:

```
sudo systemctl start xinetd
```

```
sudo systemctl enable xinetd
```



Copy the zImage file to /var/lib/tftpboot. Reboot the target board into u-boot. Execute the following:

```
tftpboot $kernel_addr_r zImage  
bootz $kernel_addr_r - $fdt_addr
```

You should see your new kernel boot just as it did earlier. You may want to combine those two commands into a new environment variable so they can be executed in one step.

## Resources

[www.denx.de](http://www.denx.de) – This is the web site for Wolfgang Denk, the principal developer of u-boot. There's a good user's manual here, a wiki, and of course, the source code at:

[git://git.denx.de/u-boot.git](https://git.denx.de/u-boot.git) – This is where to get the latest version of u-boot.

[devicetree.org](http://devicetree.org) – home page for the device tree project. Among other things, this site documents bindings that aren't covered by the Linux kernel or the ePAPR (see next reference). There's also a good tutorial on device tree usage.

*Standard for Embedded Power Architecture™ Platform Requirements ePAPR* – available at [www.power.org/resources/downloads/Power\\_ePAPR\\_APPROVED\\_v1.0.pdf](http://www.power.org/resources/downloads/Power_ePAPR_APPROVED_v1.0.pdf). Probably the most complete definition of the device tree.

## 15. The Final Steps: Linux Initialization and Flash File Systems

The application is working and we have an optimized kernel image, BusyBox, 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 in User space up and running properly.

### Linux Initialization

The `init` executable is typically `/sbin/init` although there are several alternative locations that the kernel will search. In the case of our target board, `/sbin/init`, like virtually every other executable, is simply a link back to BusyBox.

Take a look at `target_fs/etc/inittab`. Each entry has four fields separated by colons:

```
<id>:<runlevel>:<action>:process
```

<code>id</code>	Arbitrary unless this command is invoking a shell through <code>getty</code> in which case it must be the same as the last characters of the device name, after the <code>tty</code> .
<code>runlevel</code>	One or more integers between 0 and 6 indicating what kernel “run levels” this command applies to.
<code>action</code>	one of eight ways <code>init</code> will treat the process.
<code>process</code>	program to run possibly followed by arguments.

The allowable actions are:

<code>once</code>	execute the process once
<code>wait</code>	execute the process once. <code>init</code> waits until the process terminates
<code>askfirst</code>	ask the user if this process should be run
<code>sysinit</code>	these processes are executed once before anything else executes
<code>respawn</code>	restart the process whenever it terminates
<code>restart</code>	like <code>respawn</code>
<code>shutdown</code>	execute these processes when the system is shutting down
<code>ctrlaltdel</code>	execute this when <code>init</code> receives a SIGINT signal, meaning the operator typed CTRL-ALT-DEL.

Note that the `sysinit` action in this case is to execute the `rcS` script.

In this case all of the run levels are simply invoking the same script, `/etc/init.d/rc` with a different argument. This in turn invokes the scripts found in directories `rc0.d/` to `rc6.d/`. Take a look at `rc5.d/` as run level 5 happens to be the default. The scripts are named `Kn*` and `Sn*`. The `S*` scripts are executed

when a run level is entered (start) and the  $K^*$  scripts are executed if you should leave a run level to enter another one. The scripts are executed in numerical order. In fact these scripts are all links to scripts of the same name but with the  $Sn$  and  $Kn$  removed in `/etc/init.d`.

What these scripts primarily do is start *services* in the form of *daemon* processes. A daemon is a server process not attached to a terminal that runs in the background waiting for a client to request its service.

So an easy way to get our thermostat to start up is to create a script. Name it `S99thermostat`. That way it will be the last script executed.

The script itself consists of three executable lines:

- The modprobe command from chapter 11 to load the display driver
- A delay of one second
- The thermostat executable

You should probably create that script in the `src/` directory and copy it over to `/etc/rc5.d`. That way you can restore normal command line operation by just deleting the file in `/etc/rc5.d`.

When you reboot you'll find that you don't get a login prompt and the thermostat doesn't report the temperature on the console. I have to admit I'm a little puzzled by that behavior. It's not clear to me why the console should fail to start. You can still SSH to the board.

Note, by the way, that another approach to starting the application is to simply replace `/sbin/init`. Make it a symbolic link that points to the application executable. This may be fine for simple applications, but there is a lot of functionality in `init` that might be useful in the final product.

## Loading the Application to NAND flash

The next step is to put the application on the SD card and arrange to mount the root file system from the card. There's already a near duplicate of the NFS-mounted file system on the card as the second partition (`/dev/mmcblk0p2` on the target, probably `/dev/sdb2` on your workstation). That file system has a `/home` directory with a `root/` subdirectory, but no `src/` subdirectory. It's also missing the modules directory for the 4.19.30 kernel that we built in chapter 12. Finally, the `S99thermostat` script file needs to be copied over to `/etc/rc5.d`.

Mount the file system partition (`bootfs/`) on your workstation. You can create a `src/network/` subdirectory under `/home` and then copy `thermostat_t` to it, or you could just copy `thermostat_t` to `/bin`. Your choice. Copy of `S99thermostat` to `/etc/rc5.d`. Copy `target_fs/lib/libmchwh*` to `/lib` of the SD filesystem. Finally, copy `target_fs/lib/modules/4.19.30-v7` to `/lib/modules` of the SD filesystem.

Mount the boot partition (`boot/`). We need to change the kernel command line, which shows up in one of two places depending on whether or not you're using u-boot. If you're using u-boot, you edit

uEnv.txt. If you're using the native R Pi loader, it's cmdline.txt. In both cases the command line currently reads:

```
dwc_otg.lpm_enable=0 console=serial0,115200 console=tty1 root=/dev/nfs nfs-  
root=192.168.15.2:/home/target_fs,nfsvers=3 fsck.repair=yes rootwait  
ip=192.168.15.50
```

Note this is actually one line of text. Replace the **bold** part with:

```
root=/dev/mmcblk0p2
```

Unmount both SD partitions, put the card back in the target board and boot. Disconnect the network cable just to verify that the file system is not being mounted over NFS. You should see the thermostat start up.

OK, now you can ship it.

## Systemd – the Newer Initialization Mechanism

The initialization scenario we've just described is called System V init and has been around for a long time. There's a "new improved" initialization mechanism called *systemd* encompassing something like 900 files. By now most major Linux desktop distros have adopted systemd, but it's been a highly contentious issue since its introduction in 2011. Proponents insist that the System V init is too slow because it starts processes serially, one at a time, whereas systemd parallelizes much of the activity. Opponents say "if it ain't broke, don't fix it".

While System V init is still used in many embedded applications, systemd is starting to creep in. For example, if you load the Raspbian distro on your R Pi as suggested in chapter 17, you'll have systemd. Since our file system is based on System V init, we can talk about systemd but can't really do much with it.

Whereas System V init uses the inittab file and a collection of shell scripts, systemd manages and acts on objects called *units*. There are many types of units, but the most common type is a *service*, represented by a file that ends in *.service*. Have a look in */lib/systemd/system* of your workstation file system. You'll see lots of *\*.service* files. To get a feel for what these files are like, open up *console-getty.service*. We won't go into great detail on the syntax of a *.service* file. Note the general structure. There are three sections: Unit, Service and Install. **ExecStart** specifies the program that carries out this service. The Resources section has some useful resources for learning about systemd.

You manage systemd through the **systemctl** command. The most common commands that **systemctl** understands are:

<b>start &lt;unit&gt;</b>	Start the specified unit(s)
<b>stop &lt;unit&gt;</b>	Stop the specified unit(s)
<b>restart &lt;unit&gt;</b>	Restart the specified unit(s). If not running they will be started
<b>reload &lt;unit&gt;</b>	Ask specified unit(s) to reload their configuration files

`status <unit>`     Display the status of the specified unit(s)  
`enable <unit>`    Create symlinks to allow unit(s) to be automatically started at boot  
`disable <unit>`    Remove the symlinks that cause the unit(s) to be started at boot  
`daemon-reload`   Reloads the systemd manager's configuration. Run this any time you make a change to a systemd file.

You had a brief introduction to `systemctl` earlier when configuring your network for the target board. For now, try this simple experiment: just execute `systemctl` by itself with no arguments. You'll get a list of all the services that are available on the workstation. `systemctl` pipes its output to `more` (or `less`, I don't know) so it's presented one page at a time. There are a lot of services available.

## Resources

Here are a few of the many tutorials that are available on systemd. Just google "systemd tutorial" to find more.

<https://www.digitalocean.com/community/tutorials/systemd-essentials-working-with-services-units-and-the-journal>

<https://www.certdepot.net/rhel7-get-started-systemd/>

<https://learn.adafruit.com/running-programs-automatically-on-your-tiny-computer/systemd-writing-and-enabling-a-service> -- this one focuses on the BeagleBone Black

There has been a lot of controversy over systemd vs. System V init.

[http://without-systemd.org/wiki/index.php/List\\_of\\_articles\\_critical\\_of\\_systemd](http://without-systemd.org/wiki/index.php/List_of_articles_critical_of_systemd)

<https://www.zdnet.com/article/linus-torvalds-and-others-on-linuxs-systemd/>

<http://0pointer.de/blog/projects/the-biggest-myths.html>

## 16. Yocto

In this chapter we'll explore a growing trend in embedded Linux development—the use of integrated build environments to build all or part of an embedded Linux distribution.

Yocto is big and complex. This is little more than a brief introduction. See the Resources section for additional material.

### The Problem

There are a number of elements that go into an embedded development environment. These include among others:

- Cross toolchain
  - Editor
  - Compiler
  - Linker
  - Debugger
  - Libraries
- Boot loader
- Linux kernel
- Root file system
- Application program(s)

Getting all of these pieces to “play together” nicely is a non-trivial exercise. Not to mention licensing issues. Open source software is released under a variety of licenses with different provisions. It is important to be sure the various licenses also play well with one another.

Where do all these pieces come from? Ultimately they come from various places around the Internet. That's all well and good, but how do you find them?

The promise of an integrated build environment like Yocto is that it brings everything together in one place. Someone else has done the work of finding all these pieces on the net and verifying that they all work together. Once you finish the configuration process, it goes out on the Internet and downloads the required sources and builds the components you have specified. One consequence of this is that the initial build typically takes a long time. Subsequent modifications typically don't take very long because only a small amount of code is being downloaded and rebuilt. Yocto uses the same configuration tools as the kernel and it has plug-ins for Eclipse.

### One Solution: Yocto

The Yocto Project, or just Yocto for short, grew out of the OpenEmbedded Project, which is where the build system and some of the meta data came from. There is a lot of really good documentation at the

Yocto project website (Resources). I particularly recommend the *Yocto Project Quick Build* as a starting point.

## Getting Started with Yocto

We're going to dive right into building a Yocto file system image. Then while it's building, we'll get into some of the structure of Yocto. The first step to working with Yocto is installing it. On your CD is `poky-<name_version>.tar.bz2`. You can untar it anywhere you like. I suggest `/opt` only because that's where I installed it.<sup>16</sup> You might want to rename the directory `poky-<name_version>/` as just `poky/` because you'll be executing a script file from it from time to time.

Yocto requires a number of packages to be installed on your workstation. The exact list depends on your Linux distribution. Here's the list for CentOS 7:

- `bzip2`
- `chrpath`
- `cpp`
- `diffstat`
- `diffutils`
- `gawk`
- `gcc`
- `gcc-c++`
- `git`
- `glibc-devel`
- `gzip`
- `make`
- `patch`
- `perl`
- `python`
- `SDL-devel`
- `tar`
- `texinfo`
- `unzip`
- `wget`
- `xterm`

The list for other distributions is similar. In reality, most of these were installed during the original CentOS installation. Here is the command for the packages I had to install (run as root of course):

```
yum install texinfo chrpath SDL-devel
```

---

<sup>16</sup> From here on I'll refer to the directory in which Yocto is installed at `<poky_distro>`.

It turns out that the version of python in the CentOS repo is too old for Yocto. The upshot is you have to download and install python3. Go to the python web site, download the latest and consult the README file for instructions on installation.

The next step is to initialize the build environment. Execute the following commands to initialize a build environment starting from your home directory:

```
mkdir yocto
cd yocto
. <poky_distro>/oe-init-build-env build-qemu
```

The dot in that last command is an alternate way to invoke the `source` command to cause the script to be executed in the current shell process. This script creates a couple of configuration files with default values, creates and changes to the `build-qemu/` directory and adds some entries to your `PATH` environment variable. Execute `echo $PATH` and note three new entries at the beginning. If you intend to use Yocto for real, you might want to add these to the `PATH` definition in your `.bash_profile` file. Otherwise you'll have to run the `oe-init-build-env` script every time you want to do something with OE-Core.

One of the files just created is `conf/local.conf`. This provides general configuration for the images you build. Let's have a look. The file is well-commented, describing each variable. The first is the `MACHINE` for which to build images. All of the possibilities are listed as comments. There are several `QEMU` targets as well as some specific hardware targets. The default is `qemux86`, the 32-bit Intel architecture running under `QEMU`, an open source multi-platform emulator supported by Yocto.

Next, some directories are specified. Default values are subdirectories of `TOPDIR`, the `build-qemu/` directory:

- `DL_DIR` – where downloaded source packages are stored
- `SSTATE_DIR` – where *shared state* files are stored. This speeds up subsequent builds
- `TMPDIR` – where build output is stored

You can specify which package management format to use with the `PACKAGE_CLASSES` variable. Unless you're building for a "debian flavor" distro, you should probably select `rpm`.

Save `local.conf` and we're ready to build a cross toolchain and a set of images. Yocto provides a set of predefined targets that we can use to build our own images. A couple of these were listed by `oe-init-build-env`. To see the complete set of image targets execute:

```
ls <poky_dir>/meta*/recipes*/images/*.bb
```

The file system that we've been working with on the R Pi is derived from `core-image-minimal`. For something a little different, I suggest building `core-image-sato`, an X Windowing environment optimized for mobile devices. So to build the `core-image-sato` image, we simply execute:



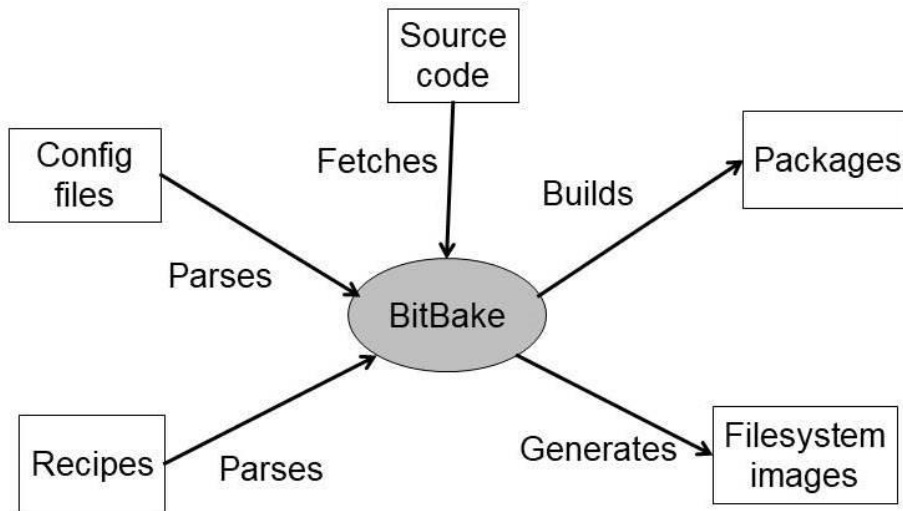
bitbake core-image-sato

Have dinner and watch a movie. This is going to take a while.

### Metadata

While the image is “baking”, it’s an opportunity to fill in some background. Yocto is based on a build tool called BitBake written in Python. BitBake’s operation can be represented graphically as shown in figure 16-1. It interprets, or parses if you prefer, *metadata* in the form of *recipes* and *configuration files* to determine what needs to be built and how. Then it *fetches* the necessary source code over the network. The output is a set of *packages* and file system *images*.

All of the metadata is contained in a set of directories under poky/that start with “meta”.



**Figure 16-1: BitBake**

BitBake is now a sub project of the Yocto Project.

Recipes are the most common form of metadata. A recipe contains a list of settings and tasks (instructions) for building a package that may then be used to build a binary image. A recipe describes where you get source code and what patches to apply. Recipes describe dependencies for libraries or for other recipes, as well as configuration and compilation options. Recipe files are named \*.bb.

Recipes are divided into a hierarchy of four categories:

- Image
- Task
- Package
- Class

The motivation for the hierarchical organization is that recipes in lower levels can be easily reused.

Image is the top level of the hierarchy. An image recipe defines what goes into a specific root file system image. It lists the tasks that make up the image. Have a look at `<poky_dir>/meta/recipes-sato/images/core-image-sato.bb`. It tells us what `IMAGE_FEATURES` it has. It also inherits a class called `core-image`. Class files get the extension `.bbclass` and reside in `meta/classes/`.

Task recipes typically represent aggregations of packages. In fact, many of them are named `packagegroup-*`. For example take a look at `packagegroup-core-boot.bb`. It claims to be “The minimal set of packages required to boot the system”. Among other things, it inherits a class called `packagegroup`.

A package recipe manages the specific needs of an individual package. Package recipes typically have a version number appended to the name. `meta/recipes-core/sysvinit/sysvinit_2.88dsf.bb` is a typical example. It identifies where the package is found on the net, the license terms, the required files, and so on.

Class recipes have a peer-to-peer relationship with the other recipes rather than a hierarchical one. They are used to factor out common recipe elements that can then be reused, through the `inherit` command. The most frequent use of classes is to inherit functions or portions of recipes for commonly used build tools like the GNU autotools. Have a look at `meta/classes/core-image.bbclass`. It in turn inherits `image.bbclass`.

In addition to the hierarchy, recipes are organized into *layers* where a layer contains a collection of related recipes. You use different layers to logically separate information in your build. As an example, you could have BSP, GUI, distro configuration, middleware, or application layers. Putting your entire build into one layer limits and complicates future customization and reuse. Isolating information into layers, on the other hand, helps simplify future customizations and reuse. You might find it tempting to keep everything in one layer when working on a single project. However, the more modular your metadata, the easier it is to cope with future changes.

The layer model even has a mechanism that allows instructions in a recipe to be overridden by instructions in a higher level recipe. This is known as a bitbake append file and is named `*.bbappend`. When bitbake “bakes” a recipe it searches to see if there are any `*.bbappend` files with the same name and adds whatever is in those files to the base recipe. This can also overwrite entries in the base recipe. Have a look at `meta-skeleton/recipes-core-busybox/busybox_%.bbappend`.

Configuration files hold global definitions of variables, user defined variables and hardware configuration information. They tell the build system what to build and put into the image to support a particular platform. Configuration files are named \*.conf.

### *Testing the New Image – QEMU*

How do we test our newly created image? Yocto includes a tool called QEMU, which stands for Quick EMUlator. QEMU is an open source project that emulates several different machine architectures. The image we just built is for the x86 running under QEMU and is named qemux86. So to test it, all we have to say is:

```
runqemu qemux86
```

runqemu is a very large Python script that is very flexible in how it handles parameters. It's in the scripts/ subdirectory of poky/.

## Yocto for the Raspberry Pi

Now that you've got your feet wet with Yocto, you probably want to build an image for the R Pi. There is a meta layer for the R Pi hosted at the Yocto site (see Resources). At the bottom of that page you'll find a link to a Git repository.

For the record, git is a very popular open source distributed version control system. It is widely used in the open source world. The Linux kernel is managed by git. We won't go into any detail on git here as it's relatively easy to use.

Rather than clicking on that repository link, proceed as follows:

- In a shell, cd <poky\_distro>.
- sudo git clone git://git.yoctoproject.org/meta-raspberrypi
- cd meta-raspberrypi
- sudo git checkout warrior

This installs the meta-raspberrypi layer. Git has a concept of *branches* representing different versions of a package. The default branch is **master**, which is the latest and greatest and may not be compatible with other packages. The **checkout** command says get the branch named **warrior** that happens to correspond to the version of poky on your DVD<sup>17</sup>.

Now do the following:

- cd to the directory where you created build-qemu
- source <poky\_distro>/oe-init-build-env build-rpi

---

<sup>17</sup> Note that this is the case as of mid 2019. It could change in future years. Check to see what poky distro your DVD has.

- Edit build-pi/conf/bblayers.conf. Add <poky\_distro>/meta-raspberrypi to the BBLAYERS string.

Have a look at the directory <poky\_distro>/meta-raspberrypi/conf/machine. These are configuration files for the various R Pis. We can specify one of these, specifically raspberrypi3, as the MACHINE variable in local.conf. So edit local.conf and add MACHINE ?= "raspberrypi3" in the section with the other target machines.

bitbake core-image-minimal

Or whatever image you'd like to build. When that process completes the final images show up in build-pi/tmp/deploy/images/raspberrypi3. Among the 100 some odd files generated are:

- Device tree blobs
- Root file system image in several formats: \*.ext3, \*.rpi-sdimg, \*.bz2
- Kernel zimage

## Alternatives

As you've probably surmised by now, Yocto is **big** and **complicated**! If you don't want to deal with that level of complexity, there are a couple other build tools that are worth taking a look at.

### *Buildroot*

Buildroot is a set of Makefiles, scripts and patches that do much the same thing as Yocto, just not as extensively. It grew out of the same project that developed both uClibc and Busybox. uClibc is a smaller footprint version of glibc that was specifically developed for embedded Linux applications.

Buildroot supports the same xconfig configuration mechanism that the kernel, BusyBox and u-boot use.

### *Linaro*

Linaro is an engineering collaboration specifically focused on the ARM architecture. It aims to provide stable, tested tools and code for multiple software distributions to use to reduce low-level fragmentation of embedded Linux software. Linaro works on software that is "close to the silicon" such as kernel, multimedia, power management, graphics and security.

Their primary product appears to be a GCC tool chain.

## Resources

yoctoproject.org – the main site for the Yocto project.

<https://git.yoctoproject.org/cgit/cgit.cgi/meta-raspberrypi/> -- web page for the Yocto meta-raspberrypi layer.

buildroot.org – website for the Buildroot project

linaro.org – website for the Linaro project.

## 17. The Next Steps

We've covered quite a bit of ground in our adventure into hands-on embedded Linux. We've touched on a number of topics, but none of them in any great depth. So there's still a long way to go.

The logical next step would be a visit to the Raspberry Pi website if you haven't been there yet. One thing you may want to try is their Raspbian Linux distribution that is derived from Ubuntu. Since the Pi was specifically developed as a teaching tool, there is a plethora of educational resources on the site.

### Web Resources

A Google search on the phrase “embedded linux” yields, perhaps not surprisingly, 112 million hits. Yes, there's a lot of information about Linux out there on the net. In addition to sites already mentioned, here are a few more that you might find particularly useful

- [www.embedded.com](http://www.embedded.com) – Probably the “go to” site for embedded computing. This site is not specifically oriented to Linux, but is quite useful as a more general information tool for embedded system issues.
- [www.fsf.org](http://www.fsf.org)—The Free Software Foundation, repository for the GNU software collection.
- [www.sourceforge.net](http://www.sourceforge.net)—“World's largest Open Source development website”. Provides free services to open source developers including project hosting and management, version control, bug and issue tracking, backups and archives, and communication and collaboration resources.
- [www.opensource.org](http://www.opensource.org)—The Open Source Initiative (OSI), a non-profit corporation “dedicated to managing and promoting the Open Source Definition for the good of the community.” OSI certifies software licenses that meet its definition of Open Source.
- [www.osdl.org](http://www.osdl.org)—Open Source Development Lab, a non-profit consortium focused on accelerating the growth and adoption of Linux in both the enterprise and, more recently, embedded spaces. In September of 2005, OSDL took over the work of the Embedded Linux Consortium, which had developed a “platform specification” for embedded Linux.
- [www.denx.de](http://www.denx.de)--Wolfgang Denk is the principal originator of U-boot. His site has a lot of interesting stuff including the Embedded Linux Development Kit, ELDK. This was originally built for the Power PC architecture but was subsequently ported to ARM and possibly MIPS. From the home page, click the Software tab to get started.
- [www.openembedded.org](http://www.openembedded.org) – This site offers a “build framework” for embedded Linux. The objective is to build everything for any target architecture from original source—boot loader, kernel, file system. Personally I haven't had much success with it, but others obviously have as it seems to be quite popular.
- [developer.android.com](http://developer.android.com) – Google's implementation of Linux for mobile devices. Android is also making its way into the embedded space. Work has been done to port Android to the Pi: [www.raspberrypi.org/magpi/android-raspberry-pi/](http://www.raspberrypi.org/magpi/android-raspberry-pi/)

### *Commercial sources of embedded Linux*

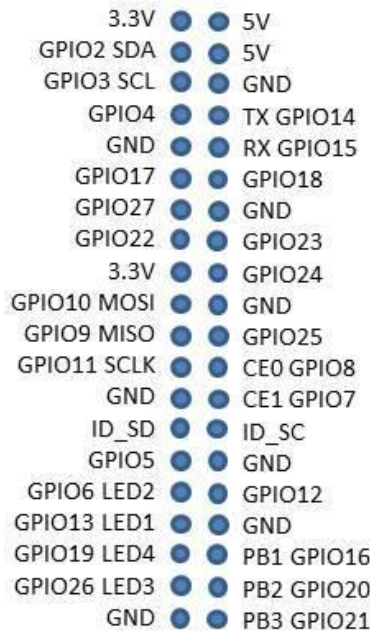
In addition to these major vendors of embedded Linux distributions, numerous vendors of single board computers offer their own customized versions of the Linux kernel.

- [www.mvista.com](http://www.mvista.com) – MontaVista Software offers several commercial Linux-based platforms to address a wide range of embedded application environments
- [www.timesys.com](http://www.timesys.com) – TimeSys offers a subscription-based service that gives developers access to hundreds of cross-compiled packages optimized and tested on reference boards from leading semiconductor manufacturers along with participation in the Developer Exchange
- [www.windriver.com](http://www.windriver.com) – In recent years Wind River (now a division of Intel) has seen the light and jumped on the Linux bandwagon.
- [www.linuxworks.com](http://www.linuxworks.com) – Lynux Works is now Lynx Software Technologies and apparently no longer supports BlueCat Linux. Their principal product is LynxOS, a Linux-compatible hard real-time kernel

# Appendix A: Target Board Connectors

All connectors are presented as seen from above.

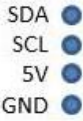
## Raspberry Pi

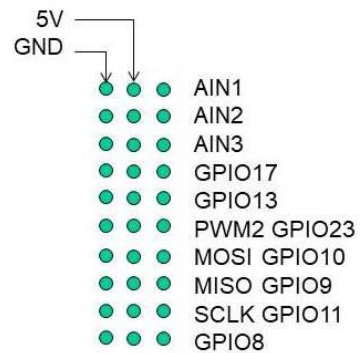
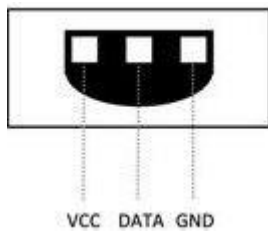
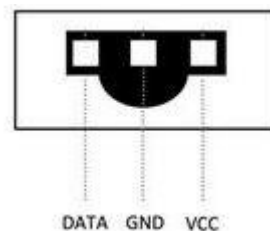


## Matrix CK

Serial Port (UART)

I2C



*Power**I/O Header**Temperature Sensor**IR Receiver*



## Appendix B: Bootloader Commands

Das U-Boot supports an extensive command set. This section describes the most useful of those commands. Because U-Boot is highly configurable, not all commands are necessarily in any given implementation. The behavior of some commands is also configuration-dependent and may depend on environment variable values.

All U-Boot commands expect numbers to be entered in hexadecimal. In the following descriptions, arguments are bracketed by `<>`. Optional arguments are further bracketed by `[]`. Command names can generally be abbreviated to the shortest unique string, shown in parentheses.

A complete user's manual for U-boot can be found at [www.denx.de](http://www.denx.de).

### Information Commands

**bdinfo (bdi)** – Displays information about the target board such as memory sizes and locations, clock frequencies, MAC address, etc. This information is available to the Linux kernel.

**coninfo (conin)** – Displays information about the available console I/O devices. It shows the device name, flags, and the current usage.

**flinfo [<bank #>] (fli)** – Displays information about the available flash memory. Without an argument, flinfo lists all flash memory banks. With a numeric argument, n, it lists just memory bank n.

**iminfo <start addr> (imi)** – Displays header information for images such as Linux kernels or ramdisks. It lists the image name, type and size and verifies the CRC32 checksum. iminfo takes an argument that is the starting address of the image. The behavior of iminfo is influenced by the `verify` environment variable.

**help [<command name>]** – Self-explanatory. Without an argument it prints a short description of all commands. With a command name argument it provides more detailed help.

### Memory Commands

By default, the memory commands operate on 32-bit integers. They can be made to operate on 16-bit words or 8-bit types by appending `“.w”` (for 16 bits) or `“.b”` (for 8 bits) to the command name. There is also a `“.l”` suffix to explicitly specify 32 bits. Example: `cmp.w 1000 2000 20` compares 32 (20 hex) words at addresses 0x1000 and 0x2000.

**base** [**<offset>**] (**ba**) – Get or set a base address to be used as an offset for all other memory commands. With no argument it displays the current base, default is 0. A numeric argument becomes the new base. This is useful for repeated accesses to a specific section of memory.

**crc32** **<start addr>** **<length>** [**<store addr>**] (**crc**) – Computes and displays a 32-bit checksum over **<length>** bytes starting at **<start addr>**. The optional third argument specifies a location to store the checksum.

**cmp** **<addr1>** **<addr2>** **<count>** – Compares two regions of memory. **<count>** is the number of *data items* independent of the specified size, byte, word or long.

**cp** **<source>** **<dest>** **<count>** – Copies a range of memory to another location. **<count>** is the number of items copied.

**md** **<addr>** [**<count>**] – Memory display. Displays a range of memory in both hex and ASCII. The optional **<count>** argument defaults to 64 items. **md** remembers the most recent **<addr>** and **<count>** so that if it is entered without an address argument, it continues from where it left off.

**mm** **<start addr>** -- Modify memory with auto-increment. Displays the address and current contents of an item and prompts for user input. A valid hexadecimal value replaces the current value and the command displays the next item. This continues as long as you enter valid hex values. To terminate the command, enter a non-hex value.

**mtest** **<start addr>** **<end addr>** [**<pattern>**] – Simple read/write RAM test. This test modifies RAM and may crash the system if it touches areas used by U-Boot such as the stack or heap.

**mw** **<addr>** **<value>** [**<count>**] – Writes **<value>** to a range of memory starting at **<addr>**. The optional **<count>** defaults to 1.

**nm** **<addr>** -- Modify memory at a constant address (no auto-increment). Interactively writes valid hex data to the same address. This is useful for accessing I/O device registers. Terminate the command with a non-hex input.

**loop** **<addr>** **<count>** -- Reads memory in a tight loop. This can be useful for scoping. This command **never terminates!** The only way out is to reset the board.

## NOR Flash Memory Commands

U-Boot organizes NOR flash memory into *sectors* and *banks*. A bank is defined as an area of memory consisting of one or more memory chips connected to the same *chip select* signal. A bank is subdivided into sectors where a sector is the smallest unit that can be written or erased in a single operation. Bank numbering begins at 1 while sectors are numbered starting with 0.

**cp <source> <dest> <count>** – The **cp** command is “flash aware.” If **<dest>** is in flash it invokes the appropriate programming algorithm. Note that the copy may fail if the target area has not been erased or if it includes any protected sectors.

**erase <start> <end> (era)** – Erase flash from **<start>** to **<end>**. **<start>** must be the first address of a sector and **<end>** must be the last address of a subsequent sector. Otherwise the command does not execute. A warning is displayed if the range includes any protected sectors. The range to be erased can also be expressed in banks and sectors.

**erase <bank #>:<start sector>[-<end sector>]** – Erase from **<start sector>** to **<end sector>** in flash bank **<bank #>**. If **<end sector>** is not present, only **<start sector>** is erased.

**erase all** – Erases all flash memory except protected sectors.

**protect on | off <start> <end>** -- Sets write protection on or off for the specified range. **<start>** must be the first address of a sector and **<end>** must be the last address of a subsequent sector. Like the **erase** command, the protect range can be expressed in banks and sectors.

**protect on | off <bank #>:<start sector>[-<end sector>]** -- Sets write protection on or off from **<start sector>** to **<end sector>** in flash bank **<bank #>**. If **<end sector>** is not present, only **<start sector>** is affected.

**protect bank <bank #>** -- Set write protection on or off for all sectors in the specified bank.

**protect all** – Set write protection on or off for all flash in the system.

Note that the protection mechanism is software-only that protects against writing by U-Boot. Additional hardware protection depends on the capabilities of the flash chips and the flash device driver.

## NAND Flash Memory Commands

NAND flash is organized into arbitrary sized named *partitions*. All NAND commands are prefixed by the keyword **nand**.

**nand info** – show available NAND devices.

**mtdparts** – list NAND partitions.

**nand erase [clean] [<offset> <size>]** – erase **<size>** bytes starting at **<offset>**. If **<offset>** not specified, erase entire chip.

**nand scrub** – really erase the entire chip, including bad blocks. Considered “unsafe”.

nand createbbt – create bad block table.

nand bad – list bad blocks.

nand read[.jffs2 | .yaffs] <address> <offset> | <partition> <size> -- read <size> bytes from flash either from numerical <offset> or <partition> name into RAM at <address>. The .jffs2 and .yaffs modifiers identify file system images.

nand write[.e | .jffs2 | .yaffs] <address> <offset> | <partition> <size> -- write <size> bytes from <address> in RAM to flash either at numerical <offset> or <partition> name. The .e modifier means write the error correction bytes. The .jffs2 and .yaffs modifiers identify file system images.

## Execution Control Commands

autoscr <addr> -- Executes a “script” of U-Boot commands. The script is written to a text file that is then converted to a U-Boot image with the mkimage utility. <addr> is the start of the image header.

bootm <addr> [<param> ...] – Boots an image such as an operating system from memory. The image header, starting at <addr>, contains the necessary information about the operating system type, compression, if any, load and entry point addresses. <param> is one or more optional parameters passed to the OS. The OS image is copied into RAM and uncompressed if necessary. Then control is transferred to the entry point address. For Linux, one optional parameter is recognized, the address of an initrd RAM disk image.

Note incidentally that images can be booted from RAM, having been downloaded, for example, with TFTP. In this case, be careful that the compressed image does not overlap the memory area used by the uncompressed image.

go <addr> [<param> ...] – Transfers control to a “stand-alone” application starting at <addr> and passing the optional parameters. These are programs that don't require the complex environment of an operating system.

## Download Commands

Three commands are available to boot images over the network using TFTP. Two of them also obtain an IP address before executing the file download.

bootp <load\_addr> <filename> -- Obtain an IP address using the bootp protocol, then download <filename> to <load\_addr>

rarpboot <load\_addr> <filename> (rarp) -- Obtain an IP address using the RARP protocol, then download <filename> to <load\_addr>

tftpboot <load\_addr> <filename> (tftp) -- Just download the file. Assumes client already has an IP address, either statically assigned or obtained through DHCP.

**dhcp** – Get an IP address using DHCP.

It's also possible to download files using a serial line. The recommended terminal emulator for serial image download is **kermit** as some users have reported problems using **minicom** for image download.

**loadb** <offset> -- Accept a binary image download to address <offset> over the serial port. Start this command in U-Boot, then initiate the transmission on the host side.

**loads** <offset> -- Accept an S-record file download to address <offset> over the serial port.

## Environment Variable Commands

**printenv** [<name> ...] – Prints the value of one or more environment variables. With no argument, **printenv** lists all environment variables. Otherwise, it lists the value(s) of the specified variable(s).

**setenv** <name> [<value>] – With one argument, **setenv** removes the specified variable from U-Boot's environment and reclaims the storage. With two arguments it sets variable <name> to <value>. These changes take place in RAM only. **Warning:** use a space between <name> and <value>, not “=”. The latter will be interpreted literally with rather strange results.

Standard shell quoting rules apply when a value contains characters with special meaning to the command line parser such as '\$' for variable substitution and ';' for command separation. These characters are “escaped” with a backslash, '\'. Example:

```
setenv netboot tftp 21000000 ulmage\; bootm
```

**saveenv** – Writes the environment to persistent storage.

**run** <name> [...] -- Treats the value of environment variable <name> as one or more U-Boot commands and executes them. If more than one variable is specified, they are executed in order.

**bootd** (boot) – A synonym for **run bootcmd** to execute the default boot command.

## Environment Variables

The U-Boot environment is kept in persistent storage and copied to RAM when U-Boot starts. It stores environment variables used to configure the system. The environment is protected by a CRC32 checksum. This section lists some of the environment variables that U-Boot recognizes.

The variables shown here serve specific purposes in the context of U-Boot and, for the most part, will not be used explicitly. When needed, U-Boot environment variables are used explicitly in commands much the same way that they are in shell scripts and makefiles, by enclosing them in \$().

**autoload** – If set to “no,” or any string beginning with ‘n’, the **rarp** and **bootp** commands will only get configuration information and not try to download an image using TFTP.

**autostart** -- If set to "yes", an image loaded using the **rarpb**, **bootp**, or **tftp** commands will be automatically started by internally calling the **bootm** command.

**baudrate** – A decimal number that specifies the bit rate for the console serial port. Only a predefined list of baudrate settings is available. Following the **setenv baudrate <n>** command, U-Boot expects to receive a newline character at the new rate before actually committing the new rate. If this fails, the board must be reset and reverts to the old baud rate.

**bootargs** – The value of this variable is passed to the Linux kernel as boot arguments, i.e. the "command line".

**bootcmd** -- Defines a command string that is automatically executed when the initial countdown is *not* interrupted, but only if the variable **bootdelay** is also defined with a non-negative value.

**bootdelay** – Wait this many seconds before executing the contents of the **bootcmd** variable. The delay can be interrupted by pressing any key before the **bootcmd** sequence starts. **Warning;** setting **bootdelay** to 0 executes **bootcmd** immediately and effectively disables any interaction with U-boot. On the other hand, setting this variable to -1 disables auto boot.

**bootfile** – Name of the default image to be loaded by the **tftpboot** command.

**ethaddr** – MAC address for the first or only Ethernet interface on the board, known to Linux as **eth0**. A MAC address is 48 bits represented as six pairs of hex digits separated by dots.

**eth1addr**, **eth2addr** – MAC addresses for the second and third Ethernet interfaces when present.

**ipaddr** – Set an IP address on the target board for TFTP downloading.

**loadaddr** – Default buffer address in RAM for commands like **tftpboot**, **loads**, and **bootm**. Note: it appears, but does not seem to be documented, that there is a default load address, 0x21000000, built into the code.

**serverip** – IP address of the TFTP server used by the **tftpboot** command.

**serial#** -- A string containing hardware identification such as type, serial number, etc. This variable can only be set once, often during manufacturing of the board. U-Boot refuses to delete or overwrite this variable once it has been.

**verify** -- If set to "n" or "no", disables the checksum calculation over the complete image in the **bootm** command to trade speed for safety in the boot process. The header checksum is still verified.

The following environment variables can be automatically updated by the network boot commands (bootp, dhcp, or tftp) depending on the information provided by your boot server:

bootfile -- see above

dnsip -- IP address of your Domain Name Server

gatewayip -- IP address of the Gateway (Router)

hostname -- Target hostname

ipaddr -- see above

netmask -- Subnet mask

rootpath -- Path to the root filesystem on the NFS server

serverip -- see above

filesize -- Size in bytes, as a hex string, of the file downloaded using the last bootp, dhcp, or tftp command.