EMBEDDING LINUX AND PLAYING WITH OTHERS

by

Kevin J Walchko

I INTRODUCTION

Welcome to the Free Software Foundation and UNIX mindset. Instructions and anything less than the cold deep end of the pool are for feebs. UNIX breeds Gurus.
- Mike McCarty

Linux has currently been installed on a variety of devices such as wrist watches, hand-held devices such as PDA's and cell phones, desktop computers, routers, robots, and large servers such as IBM's S390. What makes Linux so attractive across such a wide spectrum of the technology market? I don't know, but I believe for most it is the flexibility and openness of the OS that attracts so many.

In the embedded market, there are a variety of off-the-shelf operating systems such as VxWorks and Windows CE. However none have the advantage of being able to modify the operating system as needed to suite your needs. This is an important point, since a lot of embedded system utilize custom hardware or operate in specific situations where the off-the-shelf products were not really designed for.

The linux embedded situation is not all roses though. Companies are afraid of licensing issues with the GPL and intellectual property issues.

Also there is no company to go to for help if there is a problem with linux, rather a loose confederation of hackers and programers are your only backup if you suddenly find yourself in over your head and need help. However there are companies such as Red Hat and Lineo that do provide assistance, but this book is discussing how to build your own embedded linux operating system and thus those solutions are not available to us.

Yet another problem with linux is involved with real-time. Other operating systems, such as VxWorks, were designed from the ground up to do real-time. They are able to guarantee answering events with in one microsecond of an event or are able to produce deterministic response latency to events. Unfortunately, real-time means different things to different people, thus there are terms such as hard and soft real-time. However, for most applications linux is capable of being used in situations that call for real-time by the use ??? There are also linux real-time API's such as RTAI and RTLinux. <<more>>

1.1 The Goal We Hope to Achieve

Putting together a linux system that is about 4.5M in size is not difficult. This document will cover my work installing linux on a compact flash (CF) card, and using an IDE adaptor to boot into linux. This configuration is not a bad deal considering a 32M card is $23 and the IDE adaptor is $20, which is far cheaper, more rugged, and less power hungery than a hard drive. Obviously you don't have the room of a hard drive, but then again most _real_ embedded application don't need that much room. However the merits of CF are listed below in no particular order.

1. survive 2,000G's (equivalent 10ft drop)
2. operate at 3.3V or 5V and consumes less %5 of the power of a notebook hard drive.
3. no moving parts
4. 3-5 M/sec burst transfer speed.
5. quiet, no hard drive sound.
6. seems to me to boot faster, but then again I am not booting a full distribution.

Topics to be covered:

1. setting up the development and embedded systems.
2. basic linux overview (e.g. boot process, kernel, programs).
3. slimming down linux to only a couple megabites.
4. get networking going
5. multi-user and sigle user (i.e. boot right into root) configurations.
6. make the root filing system read-only and a ram disk so you can write data, temp files, etc. That way if you turn off the system without powering down you will not corrupt linux. it should almost behave like an appliance.

Conventions Used

TABLE 1-1. Conventions

	Definition
>>	command prompt
filename@	a symbolic link to another file
filename*	a file that is executable
blah ...	information that is typed or displayed on the command line
blah ...	information that is in a configuration file

TOOLS OF THE TRADE II

This chapter will discuss some of the tools you will utilize on both the development system (a desktop or laptop PC) and the embedded system. Much of the explainaitons are very quick and basic but provde you with the necessary information neede to build a linux embedded device.

2.1 Preparing the Development System

All of this work will be done on a desktop computer using linux. You will need to do the following to prepare your system, or modify this to suit your needs.

You will need to install a 2.4 series kernel (which is the current stable series at the time of this writing) and make sure to add support for USB storage. You will also have to add SCSI support to your kernel since usb-storage emulate a SCSI drive.

Once your kernel or modules are recompiled, reboot or load the modules. Attach the CF reader to one of your USB port. Now you should be able to mount your CF on /dev/sda1 (assuming that you do not have any other SCSI storage devices present). Add the following line to your /etc/fstab file.

Listing 2-2. /etc/fstab

```
/dev/sda1 /mnt/compact_flash auto defaults, user,noauto,exec 0 0
```

If you later run into problems running executable, strace or ldd on files on the compact flash card, the problem is you forgot the exec (executable) in the above line.

GCC

Although there is no C or C++ coding necessary to build your own custom linux distribution, we will quickly go other some of the highlights of programming under linux since you will enevitably have to program something at some point.

The gcc compiler is a great cross platform C/C++ compiler that is currently used on just about every existing computing platform. Generally it is a very efficient and well behaved compiler that is easy to work with. However it does have a few quirks.

The basic command to compile a program with gcc is (where the basic syntax is: gcc [option | filename]):

```
>> gcc -o main main.c
```

which will compile the program main.c and produce a binary executable program main. This program will be linked to glibc library (libglibc.so) at run time. A summary of some of the options available for gcc are shown below.

TABLE 2-1. Summary of gcc options.

Option	Description
-ansi	Support all ANSI standard C programs. This turns off certain features of GNU C that are incompatible with ANSI C, such as the asm, inline and typeof keywords.
-c	Compile or assemble the source files, but do not link. The compiler output is an object file corresponding to each source file.
-o *filename*	The output from gcc is given the name filename. If this is not specified, then gcc names the resulting executable a.out.
-L *path*	When linking, add the additional library path *path*.
-l *libname*	When linking, link with the following shared library *libname*.
-O *n*	This option specifies the optimization level that should be used when compiling code. Higher level of optimizations may take short cuts in order to speed execution of the program and result in undesireable effects, thus it is important to understand what this option does. However, it is generally perfectly safe to use -O2 when compiling your program for release.
-D *symbol*	This option defines a symbol in the C preprocessor, which enables options in your code to be turned on and off at compile time.

TABLE 2-1. Summary of gcc options.

Option	Description
-pendantic	This enables strict interpretation of the ANSI C Standar. Also, using -pedantic -W -Wall will warn you about sloppy coding.
-g	This option produces an executable with debugging information, so a debugger such as gdb can be used to step through the code. This option should be turned off when the code is at the release state, since it produces a larger binary due to this debugging information.

Although the syntax for compiling a single program is rather simple, for more complex programs it is common to use Make and Makefiles or even autoconfig to compile projects. These are complex beasts and thus will not be covered here.

C vs C++

Technically, there is no performance difference between C and C++ (provided you do not use some of the more exotic features). These more exotic features add complexity, but also ease the difficulty of producing code by using features such as polymorphism, templates, run-type-type-interface (rtti), and multiple inheritance. When a programmer uses these techniques, it is understood that the code will have to jump through more hoops, and thus lower performance, but these are powerful techniques and exciting things can be done with them. Thus it is recommended when clarity of code is important (because object oriented code has a step learning curve) or fast, low overhead code is desired. But you can still use simple C++ code to full fill these requirements, right?

Unfortunately this is not true in real life. The gcc compiler (currently 2.96 and older) does not produce efficient code when using g++ (which is still gcc but with various different options specifically for C++). However this is supposed to change with the introduction of gcc 3.0 which is on the verge of being distributed right now.

Another factor that will sand bag your C++ code is the linker. There currently are issues with how the linker links with C++ libraries (especially libstdc++) which produces large, slow code. This is particually evident in the KDE desktop which used a program kinit to help speed up the startup time of C++ code. Unfortunately this is a hack and results in 10-15 of these kinit process running at all times, which is not desirable. In addition, it has been shown that the linker will generally increase the size of each running C++ application by as much as .5 MB (which can add up, especially on an embedded device). Hopefully in the future, these problems will be resolved and C++ can be considered as a viable language on embedded systems with scarce resources when working with gcc.

Setting Up and Maintaining CVS

This is an indispensable system used throughout the world. Current Versioning System or CVS for short, is a simple command line system for maintaining code with multiple distributed programers throughout the world. It is the most common versioning system found in the open source or free software world, and has many years of experience under its belt.

Setup and Installation

Most distributions automatically install cvs on your system when you tell it to install the developer tools. However if for some reason they were not, simply go and get the rpm's (Red Hat Package Manager) from your installation CD or the web. Then simply install them using the following command.

```
>> rpm -ihv cvs-1.1-5mdk.i586.rpm
```

Now the name of the rpm file will most likely be different. Here we have cvs version 1.1 made for Mandrake Linux (mdk) and it is the 5th update for Mandrake on a pentium computer (i586).

First create a repository for your code somewhere that all programmers can get to. It is not advisable to set up the repository in the same directory that you are doing work (i.e. /home/bob/programming), but rather a more central location for all developers. Here we will set it up in the /usr/local directory.

```
>> cvs -d /usr/local/cvs init
```

The cvs directory should now exist with some files in it, double check to make sure it is there.

Inorder for developers to work with cvs we need to make a group (or use one that already exists) and add all developers to that group. We will use the group cvs and we need to make the cvs repository accessable to them.

```
>> chgrp -R cvs /usr/local/cvs
>>chmod ug+rwx /usr/local/cvs
```

Also, all developers need to add the following variable to their enviornments (you don't have to, but it will make life easier).

- CVSROOT=/usr/local/cvs
- EDITOR=emacs

The cvsroot variable identifies the location of the cvs repository. This repository can contain hundreds of different project a developer may be working on. Thus it is not necessary to create a cvsroot variable for each project you work on.

The editor variable is used when code is submitted, you must write a little statement as to what you did. Here we set it to emacs, but it can be anything you want. The editor defaults to vi if this variable is not present.

Now that the repository is set up and all developers can work with it, lets import the code. Change into the directory of the code you want in cvs (i.e. /home/bob/my_code where all the c files are located).

```
>> cvs import -m ”initial import of project” my_projet prj start
```

This command tells cvs to import all of the code in the current directory into the cvs. The cvs will file all of this code under the project name my_project and tag it with the comment ”initial import of project” which is what the -m tells it to do. If the m option had not been used, then cvs would have launched the and editor (specified by EDITOR variable) and you would have had to write it in then.

The last two entries on the command line are irrelevant (i.e. the prj and start). These are the vender tag and relase info. They do nothing important in cvs and can be set to anything.

We are finally done setting up cvs. All the code is now in the repository, permissions are setup correctly, and all developers are members of group cvs with the proper environmental variables.

Using CVS

How does a developer use cvs? Well the first thing that has to be done is to check out the code from the cvs (where CVSROOT points to the repository’s location).

```
>>cvs checkout my_project
```

Cvs will download the project named my_project into a folder int the current directory called my_project. All of the code will be there plus some new directories called CVS. These directories contain CVS information on that project and should be left alone.

Now a developer can easily work on the project, modifying source code, compiling and running. Once the code has reached a state where it should be put back in to the repository, you must first update your code (to get any changes anyone else has made) and then do a commit (porvided no conflicts have been found between your code and the code in the repository).

```
>>cvs update
>> cvs commit
```

One thing to remembre to both the commit and update commands, is that they only apply to the current directory and lower. They do not update and commit on code that is in a higher directory.

Also, how does cvs know which project it is commiting to, since you did not state anything on the command line? That is what the directories CVS are for. They track the project and revision histories of all files and directories in that project.

Modifying the CVS

Files or directories can be removed or added to the cvs at a later date. Adding file or directories is easy: cvs add [filename | directory]

Removing a file or directory is a three step process, but is also not that difficult.

1. remove the file or directory from your working directory
2. cvs remove [filename | directory]
3. cvs commit

TABLE 2-2. Shorthand symbols used by cvs to depict the state of a file.

Code	Description
N	new file added to cvs
U	file has been updated to the most current
C	file contains a conflict with what is in the cvs
M	file was modified

Other Commands

The good thing about using cvs is that you never loose anything. If you are working on a project and all of a sudden the program doesn’t work, you can revert to a previous version to try to figure out what is going wrong. This command takes the following form: cvs -j from_version -j to _version filename.

```
>>cvs -j 1.3 -j 1.3 hello.c
```

Here we have grabbed the previous version of a file.

Auto Documentation Tools

As Programs get more and more complex, good documentation is needed. Unfortunately just commenting the code is often not enough. This provides a very narrow view of how all of the parts interact with each other. Often when someone new is first getting to work on a project, it is of greater value to see a higher depiction of everything with out all of that code getting in the way.

There are several tools that accomplish this task, but the one I have used the most is doxygen (document generation). This program pulls comments from source code and creates a complete picture of how everything interacts.

blah...

2.3 Preparing the Embedded System

Hardware

Linux can be run on a variety of embedded systems. Some of these are: x86, ARM, and MIPS. Although we will primarily deal only with x86 processor from Intel and AMD.

I am using a CF IDE adaptor which allows it to be hooked up to any system and treated at the main hard drive. This will simplify the boot of the embedded system, since almost all embedded systems have IDE capabilities.

<< picture IDE adaptor. >>

FIGURE 2-1. IDE adaptor for my compact flash can be obtained from www.pcengines.com/cflash.htm

Compact Flash

This process should work on any type of media that linux can boot from. To prepare the CF, insert the card into your CF reader (make sure to load any modules if not already compiled into the kernel). Verify that linux sees the card

```
>> cat /proc/bus/usb/devices
... bunch of stuff scrolls by ...
S:  Manufacturer=SanDisk Corporation
S:  Product=ImageMate CompactFlash USB
... more stuff ..
```

Linux sees my flash card reader. Now we can put a filing system on it.

```
>> fdisk /dev/sda
```

List all the possible commands (if you are unfamiliar with the program) fdisk has by pressing [m]. Delete the current partition [d] and create a new parition(s) with [n]. next set the partition which will contain the boot files bootable, and write this information to the CF with [w].

Finally we need to format the partition(s) you made. I prefer ext2 (our boot loader, lilo, only works with ext2), but you could use DOS, XFS, MINIX, etc.

```
>> mke2fs /dev/sda1
```

Now mount the CF and change directory in to it. You should see a lost+found directory.

There have been reports of some CF manufactures producing flaky cards, that do not like to be reformatted. Although I have not yet ran across one of these yet, supposedly using the dd command and writing all 0's to the card fixes the problem, and allows you to format the card however you wish.

Other Filing Systems

RAM Filing Systems

CramFS

Journalling Filing Systems

2.4 Text Editors (this needs to be moved)

Vi

blah

Emacs

This is an excellent editor when working on your development system. Its graphical interface is somewhat complex, but can easily be figured out.

However, in the embedding environment emacs from a command prompt is far too powerful and complex to use. I hate to have to remember all kinds of complex key commands in order to get it to do something. Thus, when logging in remotely to your embedded system, save yourself the trouble and use a simpler text editor.

Pico

Unlike vi and emacs, pico is a very simple text editor with all of its commands listed at the bottom of the screen. This program is perfect for modifying simple configuration files when setting up or even fine tuning your linux system. However, pico is not particularly well suited to working on complex programming projects with many C/C++ files. At these times, it is best to learn and use either of the two previous editors.

Nano

My preferred text editor of choice on the command line is pico. Unfortunately pico is part of pine and not redistributable. Nano is a text editor that mimics pico and is released under the GPL. It can be obtained from www.nano-editor.org. Nano uses ncurses so they must also be complied to the embedded system along with some of the files in /usr/share/terminfo. Our environment contains TERM=linux, and under the l directory is a definition for this terminal. Other terminal definitions can be copied too if so needed.

2.5 Visual Information (this needs to be moved)

When you are working on an embedded project, you typically do not have a graphical interface with all of the audio and visual feedback. Typically all you have is the command line, and with the bash shell and linux you can have color.

Colorizing Our Prompt

We are going to use the full version of bash and not the simplified [[terminals]] that come with busybox. Basically all that needs to done is copying the executable from your development system over to the embedded system.

Next we will modify our bash prompt so that we get some useful information and some eye catching color to go along with it. The following listing will produce a prompt with a user's name in green and the current directory in blue. Upon logining in as root however, the root name will appear now in red.

Listing 2-6. Adding color to the prompt: /etc/???

```
# set color names
# normal
CLR="\[\033[0;0m\]"   # normal color scheme
BK="\[\O33[0;30m\]"   # black
BL="\[\033[0;34m\]"   # blue
GR="\[\033[0;32m\]"   # green
CY="\[\033[0;36m\]"   # cyan
RD="\[\033[0;31m\]"   # red
PL="\[\033[0;35m\]"   # purple
BR="\[\033[0;33m\]"   # brown
GY="\[\033[1;30m\]"   # grey
# enhanced
eGY="\[\033[0;37m\]"  # light gray
eBL="\[\033[1;34m\]"  # light blue
eGR="\[\033[1;32m\]"  # light green
eCY="\[\033[1;36m\]"  # light cyan
eRD="\[\033[1;31m\]"  # light red
ePL="\[\033[1;35m\]"  # light purple
eYW="\[\033[1;33m\]"  # yellow
eWT="\[\033[1;37m\]"  # white

if [ $USER = "root" ]; then
	export PS1="[$eRD\u@\h$BL \W$CLR]\\$ \[\e[0m\]"
else
	export PS1="[$GR\u@\h$BL \W$CLR]\\$ \[\e[0m\]"
fi
```

Color for ls

We are also going to colorize the output of the ls command. Unfortunately, the most current version of the ls command uses pthreads, which is almost 600k in size. Since I am currently getting short on space, this is not good. Thus I went to www.fsf.org and found the previous version of the gnu fileutils (version 4.0) and compiled it. This resulted in an ls command that did not require librt or libpthreads.

Note: I had some problems compiling the code. First I did ./configure then make from the top level directory. The make process halted half way through with a bunch of errors on a binary I didn't care about. I then went to src/ and did a make ls. This then compiled the ls command I wanted, and copied to the CF.

Man Pages

Man: - not enough room right now

- Bash: pipes
- less
- more

grep

III How Linux Works

This chapter will cover the basics of the boot up process and discuss the linux kernel. It is not intended to be very technical, but to give you an over all idea of how linux works.

3.1 The Boot Process

When a computer starts, here it is an embedded device, the first thing to be loaded in to memory is the boot loader. This boot loader is found on the first sector (or boot sector) of a disk or hard drive typically. On a hard drive, this sector that is looked at first is also refered to as the master boot record. It is possible to have multiple partitions on a hard drive each with a boot sectors in them.

After the boot loader is put into it will load the kernel from one of these boot sectors. Typically the linux kernel is compressed, since it is rather large now-a-days, thus the kernel needs to be uncompressed into memory.

Next the kernel begins to identify what hardware is installed and prints a lot of information to the screen (if there is one on your embedded device).

Next the root filing system is mounted read-only and is checked, then remounted read-write.

Finally the kernel is designed to start a program init, which configures your system, depending on what it is configured to do. init always starts as the first user process and thus has a process number of 1. It can not be terminate until the system is shutdown. Once init is done its job after boot, your system should be up and running.

3.2 The Linux Kernel and Modules

The linux kernel is a monolithic kernel which means that all device drivers are part of the kernel. The linux kernel is capable of loading and unloading drivers during run-time. This enables linux to load a driver when needed, then remove it later to reduce the size of the running kernel. A smaller kernel is a faster kernel.

These drivers or modules are modular porgrams that assume nothing about the state of the running kernel. This aids in debugging since a module can be compiled, inserted in to the running kernel and tested. The programmer can then easily remove the module from the kernel and make modifications to the code as needed before recompiling and reinstalling the module.

Below is a summary of the module utilities used by linux. Thus

TABLE 3-1. Summery of the module utilities.

Command	Description
insmod	inserts a module into the running kernel
modprobe	inserts a module and any other modules it depends on in to the running kernel
depmod	keeps a record of dependencies for the various modules
lsmod	lists all currently loaded modules
rmmod	removes a module from the runninf kernel

as a simple example, the command to load the parallel port module in to the kernel would be this:

```
>> insmod -v paport parport_pc lp
```

The -v is an option for verbose, the parallel port module lp depends on the two previous modules parport and parport_pc to be loaded prior to it for it to work. This can be alot to type, especially if you are not sure on what the dependencies are. Thus the command modprobe automatically handles the dependences for you. Thus the command would be:

```
>> modprobe -v lp
```

This command is much easier to remember and will have the same results. But when ever your modules change (i.e. added a new module), you must issue the following command to keep the dependencies current:

```
>> depmod -a
```

The -a option specifies that all module dependencies should be updated.

Advantages/Disadvantages
Although previously mentioned, modules are great because they can:

- Reduce the size of the running kernel to only the modules needed, thus producing a faster kernel.
- They also promote modular programming that is easier to maintian, assumes nothing about the running kernel, and is easier to debug.
- They also allow inclusion of binary only drivers which is good since some companies are unwilling to open source their driver code and require Non-Disclosure Agreements (NDA) just to make their hardware compatable with linux.

They unfortunately require the module utilities and their libraries which take up additional space which may alread be scare on embedded systems. Thus for embedded systems it would probably be easier for all needed modules to be compiled in to the kernel. This also reduces the risk of something going wrong and a module not loading when the kernel needs it. Thus modules are one more thing that can go wrong during run-time for your system.

IV Embedding Linux

This chapter is basically going to explain how to design your own customized linux distribution. The mechanics behind how linux works are really very simple, and most people are rather surprised by how easy it is to create a working linux system from scratch.

The first thing we will do is go over the linux kernel. Next we will move on to linked libraries that are needed for running code. Then we will move on to creating a root filing system, with all of the necessary files on it. Finally we will install and set up the boot loader, so that we can properly boot the system when it starts.

4.1 Kernel

First you need to get the source for the linux kernel from www.kernel.org. Decompress and untar the file using either the command:

```
>> tar zxvf linux-2.4.x.tar.gz
>> tar Ixvf linux-2.4.x.tar.bz2
```

where x is the current revision number. Then issue the following commands:

```
>> make mrproper
>> make xconfig
```

Or menuconfig if Xwindows is not running on your system. Now add support for:

- RAM disk support [block devices]
- RAM disk (initrd) [block devices]
- ROM file system [file systems]
- Proper target processor
- TCP/IP networking and proper ethernet card

Does to some attempt try and figure out what should be in the kernel but it doesn't always do a good job, and you typically can remove many drivers that are set up by default. Thus some of the modules you can remove support for (to make the kernel smaller and faster) are:

- SCSI (unless needed)
- DRI (3D hardware acceleration, shouldn't need)
- USB and Fire Wire (unless needed)
- PCMCIA (unless needed)
- modules (we are going to compile everything into the kernel for now)
- NFS and other filing systems (unless needed)
- sound cards
- modems

save changes and exit. We are going to make a compressed kernel, which will save storage space and not effect the performance of linux at run-time. Next type:

```
>> make bzImage
```

Once the kernel is compiled, it will be located in arch/i386/boot/bzImage under your current directory. You will eventually copy this to the boot directory on the CF.

PCMCIA Support / Modules

Eventually I want wireless access and will need to add support for this stuff. I may also need to do modules, if I can't compile the drivers into the kernel. Busybox will do insmod, so that should help.

4.2 Slimming Down Linux

A standard linux distribution uses lots of scripts, libraries, and programs in order to make a users experience using any hardware under any circumstance more enjoyable and productive. In the embedded space, with specific (even custom) hardware and limited space, this is not a good solution. Thus to trim down on some of the fat in linux, but retain the experience, we will use two programs that combine numerous programs into one binary. Busybox and tinylogin give you almost every command you need, with all of the common options retained and the less frequently ones removed.

Glibc

The standard linux libraries are rather big and bulky when dealing in the embedded space. Both of these programs can be compiled with various other libraries (i.e. linux-libc5, uClibc, newlib and diet-libc) designed to reduce the size. However, I will compile both busybox and tinylogin so they can be dynamically linked to glibc at run time.

NSSwitch

I will also use nsswitch stuff which will greatly increase the size and complexity of my embedded system (i.e. Must include all of the ns libraries and /etc/nsswitch.conf). It also increases the security, but this project is target for mobile robotics and security (except maybe military applications) are not high priority. Originally I tried to get around using nsswitch by letting busybox and tinylogin handle permission stuff, but I ran into problems with other programs that were compiled with glibc. They (i.e. inetd, ftpd and telnetd) would not function properly. Thus, a quick recompile of busybox and tinylogin, and things started working.

Busy Box

Busy box is a collection of common unix commands compiled into one executable. The commands contain all commonly used parameters and occupy less space. Get the source from http://busybox.lineo.com and unpack it just as the kernel was done. Modify the Config.h (if needed) to add or remove capabilities of the program. Next, change the following variables in the Makefile.

```
DOSTATIC=false
USE_SYSTEM_PWD_GRP = true
```

The first on will produce a smaller sized binary, but require us to link to glibc. The second one will force us to use. Finally type make install. This will compile the program and create a directory tree in _install with symbolic links for all of the commands.

_install/bin:

```
busybox cp@ echo@ kill@ more@ rm@ sync@ uname@
cat@ date@ false@ ln@ mount@ rmdir@ tar@ zcat@
chgrp@ dd@ grep@ ls@ mv@ sed@ touch@ chmod@ df@
gunzip@ mkdir@ ps@ sh@ true@ chown@ dmesg@
gzip@ mknod@ pwd@ sleep@ umount@
```

where @ represents a symbolic link to busy box. To see this try the following.

```
>> ls -al cp
lrwxrwxrwx 1 root root 7 May 25 11:44 cp -> busybox
```

_install/sbin:

```
halt@ klogd@ mkswap@ reboot@ swapon@
init@ lsmod@ poweroff@ swapoff@ syslogd@
```

_install/usr/bin:

```
basename@ cut@ env@ head@ logger@ tail@ uptime@ whoami@
chvt@ dirname@ find@ id@ reset@ tty@ wc@ xargs@ clear@ du@
free@ killall@ sort@ uniq@ which@ yes@
```

_install/usr:

```
chroot@
```

4.3 Multi-User: Tiny Login

Much like busybox, tinylogin is a simple, reduced size method for the standard set of libraries and tools under linux. Why is something like TinyLogin useful?

```
>> du -ch ‘which addgroup adduser delgroup deluser getty login passwd su sulogin‘
0   /usr/sbin/addgroup
24k /usr/sbin/adduser
16k /sbin/getty
36k /bin/login
28k /usr/bin/passwd
24k /bin/su
12k /sbin/sulogin
140k        total

>> ls -sh ./tinylogin
 40k ./tinylogin*
```

Tinylogin is less than one third the size of the normal gnu utilities it replaces. It can be obtained from http://tinylogin.lineo.com. Unfortunately the documentation for tinylogin is very sparse, but there is enough there to get you started. It creates links for its commands just like busybox.

_install/bin

_install/sbin

_install/usr/bin

_install/usr/sbin

When a user logs into the system from a console, the /etc/profile is read. From here you can set environment variables such as the prompt PS1=”[\u@\h \W]\$ “ which will (with the proper environment) give you the command prompt [bob@embedded bin]$ for example.

To install, just edit the Config.h file to add or remove programs for tinylogin. The modify the Makefile to use nsswitch.

```
USE_SYSTEM_PWD_GRP = true
```

Create a simple pass word and group file so you can log in to the machine. Then use the adduser command to add others to the machine.

Listing 4-4. /etc/passwd

```
root::0:0:root:/root:/bin/bash
```

Listing 4-5. /etc/group

```
root:x:0:
```

This gives root a blank password.

4.6 Creating the Root File System

You want the root filing system to contain the following directories:

- bin - global binaries
- boot - boot files
- dev - hardware stuff
- etc - system configuration stuff
- home - user homes for multi-user systems (optional)
- lib - global libraries
- mnt - mount points for removable media (optional)
- proc - system info
- sbin - system binaries that users don't need access to
- usr - user specific stuff
- var - misc
- tmp - location for temporary files

/bin

Move the bin directory from the busybox directory to the CF.

```
>> mv bin /mnt/compact_flash
```

You will now have the majority of commands you need to run linux and they are all linked to one small binary.

Most everything you need should already be there. I however added a few more. I copied bash over from my desktop system. Unfortunately these are dynamically linked programs so I used ldd to find what they where linked to and copied those libraries over to the CF as well.

Also, to get the su command to work, you must issue the following command to set it suid.

```
>> chmod a+z su
```

/boot

I copied the following files from by desktop into the boot directory.

```
boot-menu.b boot.0340 boot.b@ chain.b message boot-text.b boot.0341
map os2_d.b
```

Also, you need to copy the bzImage (kernel you made previously) here too.

/dev

You will also have to make the following device files using mknod name type major minor (i.e. mknod hda b 3 0)

```
crw-r--r--    1 root  root 5,  1 console
brw-r--r--    1 root  root 2,  0 fd0
brw-rw-r--  1 root  root 3,  0 hda
brw-rw-r--  1 root  root 3,  1 hda1
brw-rw-r--  1 root  root 3, 64 hdb
brw-rw-r--  1 root  root 3, 65 hdb1
brw-rw-r--  1 root  root 3, 66 hdb4
crw-r--r--    1 root  root 1,  3 null
brw-------    1 root  root 1,  0 ram0
brw-rw-r--  1 root  root 8,  0 sda
brw-rw-r--  1 root  root 8,  1 sda1
crw-rw-rw- 1 root  root 5,  0 tty
crw-r--r--    1 root  root 4,  0 tty0
crw-r--r--    1 root  root 4,  1 tty1
```

If you are unfamiliar with this, consult the man page (it really is pretty simple). To get the major and minor numbers for the devices, look at the dev directory on your linux system.

/etc

The following configuration files were also put in this directory. Also copy the services, and protocols from /etc on your development system. When your system boots, remember to run ldconfig which will create /etc/ld.so.conf for you.

Listing 4-7. /etc/fstab, this sets up the file systems on the machine. The last one listed is created in RAM. It is the System V shared memory, minimal virtual filing system.

```
/dev/hda1 / ext2 defaults 1 1
/dev/fd0 /mnt/floppy auto sync,user,noauto,nosuid,nodev 0 0
proc /proc proc defaults 0 0
none /var/shm shm defaults 0 0
```

Listing 4-8. /etc/inittab, this is the script that is run by init which is the first process to be run when the system is booted.

```
::sysinit:/etc/init.d/rcS
::askfirst:/bin/bash
tty2::askfirst:/sbin/login tty2
#tty2::askfirst:/sbin/getty 38400 tty2
tty3::askfirst:/sbin/login tty3
tty4::askfirst:/sbin/login tty4
::ctrlaltdel:/bin/umount -a -r
::shutdown:/bin/umount -a -r
```

Listing 4-9. /etc/nsswitch.conf, handles the permissions.

```
passwd: files nisplus nis
shadow: files nisplus nis
group: files nisplus nis
hosts: files nisplus nis dns
bootparams: nisplus [NOTFOUND=return] files
ethers: files
netmasks: files
networks: files
protocols: files
rpc: files
services: files
netgroup: nisplus
publickey: nisplus
automount: files nisplus
aliases: files nisplus
```

Listing 4-10. /etc/init.d/rcS, this is the main script run by inittab when the system starts up.

```
#! /bin/sh
mount -a                        #mount the default file system in /etc/fstab
e2fsck -v -f -y /dev/hda1       #check root file system before remounting
mount -n / -o remount,rw   #remount / as read/write

/bin/hostname merlin.home.com  # set computer's name

# setup loop back networking
/sbin/ifconfig lo 127.0.0.1 netmask 255.0.0.0 broadcast 127.255.255.255
/sbin/route add -net 127.0.0.0 netmask 255.0.0.0 dev lo

# setup networking to other hosts
. /etc/init.d/network start

# start inetd
/usr/sbin/inetd -i

sleep 10
```

Listing 4-11. /etc/securetty, this lists ttys from which root can log in on.

```
tty1
tty2
tty3
tty4
```

Listing 4-12. /etc/profile, sets up the users environment.

```
export PS1="[• @\h \W]\$ "
export HISTSIZE=200

export USER=‘id -un‘
#export LOGNAME=$USER
export HOSTNAME=‘/bin/hostname‘

alias ls='ls --color '
```

Listing 4-13. /etc/issue, a greeting presented to all that login.

```
Welcome to Kevin's Linux

This is designed for small embedded platforms.
```

/lib

I copied the following files from my desktop and copied them to this directory. Notice that the glibc 2.2 library is 1.2MB in size. That is about 16% of my total storage capacity! This is why other people develop smaller library replacements for glibc.

```
 rwxr-xr-x   root  root    436236  ld-2.2.2.so*
lrwxrwxrwx  root  root           11  ld-linux.so.2 -> ld-2.2.2.so*
-rwxr-xr-x   root  root  1222404  libc-2.2.2.so*
lrwxrwxrwx  root  root           13  libc.so.6 -> libc-2.2.2.so*
-rwxr-xr-x   root  root      22760  libcrypt-2.2.2.so*
lrwxrwxrwx  root  root           17  libcrypt.so.1 -> libcrypt-2.2.2.so*
-rwxr-xr-x   root  root      10764  libdl-2.2.2.so*
lrwxrwxrwx  root  root           14  libdl.so.2 -> libdl-2.2.2.so*
lrwxrwxrwx  root  root           17  libhistory.so.4 -> libhistory.so.4.1*
-rwxr-xr-x   root  root      20596  libhistory.so.4.1*
lrwxrwxrwx  root  root           15  libncurses.so -> libncurses.so.5*
lrwxrwxrwx  root  root           17  libncurses.so.1 -> libncurses.so.5.2*
lrwxrwxrwx  root  root           17  libncurses.so.1.9.9e ->libncurses.so.5.2*
lrwxrwxrwx  root  root           17  libncurses.so.3 -> libncurses.so.5.2*
lrwxrwxrwx  root  root           17  libncurses.so.3.0 -> libncurses.so.5.2*
lrwxrwxrwx  root  root           17  libncurses.so.4 -> libncurses.so.5.2*
lrwxrwxrwx  root  root           17  libncurses.so.4.2 -> libncurses.so.5.2*
lrwxrwxrwx  root  root           17  libncurses.so.5 -> libncurses.so.5.2*
lrwxrwxrwx  root  root           17  libncurses.so.5.0 -> libncurses.so.5.2*
lrwxrwxrwx  root  root           17  libncurses.so.5.1 -> libncurses.so.5.2*
-rwxr-xr-x   root  root    265944  libncurses.so.5.2*
-rwxr-xr-x   root  root      79188  libnsl-2.2.2.so*
lrwxrwxrwx  root  root           15  libnsl.so.1 -> libnsl-2.2.2.so*
-rwxr-xr-x   root  root      39380  libnss_files-2.2.2.so*
lrwxrwxrwx  root  root           21  libnss_files.so.2 -> libnss_files-2.2.2.so*
lrwxrwxrwx  root  root           15  libpwdb.so.0 -> libpwdb.so.0.61*
-rwxr-xr-x   root  root    129548  libpwdb.so.0.61*
lrwxrwxrwx  root  root           18  libreadline.so.4 -> libreadline.so.4.1*
-rwxr-xr-x   root  root    152440  libreadline.so.4.1*
lrwxrwxrwx  root  root           19  libtermcap.so.2 -> libtermcap.so.2.0.8*
-rwxr-xr-x   root  root      11608  libtermcap.so.2.0.8*
-rwxr-xr-x   root  root       9164  libutil-2.2.2.so*
lrwxrwxrwx  root  root           16 libutil.so.1 -> libutil-2.2.2.so*
```

/sbin

This directory is copied from the busybox _install directory. Additionally I added ldconfig (need if you have any dynamically linked programs) and lilo (only necessary if you are going to change something in lilo while the embedded system is running), but these are optional. The e2fsck is for when the root filing system is remounted read/write.

```
-rwxr-xr-x   root    root      484904  e2fsck*
-rwxr-xr-x   root    root      106304  ldconfig*
-rwxr-xr-x   root    root        69116  lilo*
```

/usr/bin

This is copied from busybox.

/usr/lib

I add the following linked libraries.

```
-rw-r--r-- root     18072 libgpm.so.1
-rwxr-xr-x root  259576 libncurses.so.5
```

/usr/sbin

This is copied from busybox. In addition I add the following linked program.

```
-rwxr-xr-x   1 root    root      48006  ckconfig*
-rwxr-xr-x   1 root    root      49870  ftpcount*
-rwxr-xr-x   1 root    root    450082  ftpd*
-rwxr-xr-x   1 root    root      50329  ftprestart*
-rwxr-xr-x   1 root    root      56600  ftpshut*
-rwxr-xr-x   1 root    root      49870  ftpwho*
-rwxr-xr-x   1 root    root      22844  inetd*
-rwxr-xr-x   1 root    root      50131  telnetd*
```

/tmp and /proc

Both of these partitions change during the run-time of the system. Thus since we are planning on using a CF for our media, it would not be wise to mount these on this drive since CF has a finite number of read/write cycles. Neither directory contains any information that is needed from boot to boot. Infact the proc directory is recreated everytime linux boots. Thus we will put these partitions (eventually) in to our ram drive (memory). Since our memory (SDRAM in this case) has no read/write limitation and all information is lost when the system is shut down, this is perfect.

4.14 Boot Loader

There are many ways to boot linux, the most common is using LILO (LInux LOader). The CF has been mounted at /mnt/compact_flash. This is of course an arbitrary mounting point. Change into /mnt/compact_flash/etc and create the file lilo.conf shown below.

Listing 4-15. /etc/lilo.conf

```
boot = /dev/sda     # location on devel system
disk = /dev/sda     # disk geometry
       bios=0x80    # make this disk the first disk
       sectors=32   # got this info from fdisk
       heads=2
       cylinders=246
map = /mnt/compact_flash/boot/map
install = /mnt/compact_flash/boot/boot.b
ramdisk=0
compact
delay = 5          # optional
vga = normal
prompt
timeout=50
default=linux
image = /mnt/compact_flash/boot/bzImage
       label = linux
      root = /dev/hda1  # location on embed system
      read-only
```

Lilo must then be put in the master boot record (mbr) by the following command and a successful attempt should produce a similar result.

```
>> lilo -C ./lilo.conf -v
LILO version 21.5, Copyright (C) 1992-1998 Werner Almesberger
Extensions beyond version 21 Copyright (C) 1999-2000 John Coffman
Released 18-Jul-2000 and compiled at 11:17:43 on Oct  3 2000.

Reading boot sector from /dev/sda
Merging with /mnt/compact_flash/boot/boot.b
Boot image: /mnt/compact_flash/boot/bzImage
Added linux *
/boot/boot.0800 exists - no backup copy made.
Writing boot sector.
```

The C parameter makes lilo modify the mbr using a specified file. If just the command lilo was given, it would automatically use /etc/lilo.conf which is the file that your linux distribution uses, and that is not what you want to use.

It is very important to have the bios=0x80 in the configuration file, many hours were spent trying to figure out why lilo would not boot properly. The system would start up and lilo would hang printing the following.

```
L 01 01 01 01 01 ...
```

The -v is just a verbose option, so that information is printed to the screen.

4.16 Network Connections

We want to add network connectivity. The binaries inetd, ftp and telnet were copied to the compact flash. Then the configuration files services, protocols, hosts, host.allow, host.deny, and inetd.conf were added to the /etc directory.

The driver for our ethernet card (built into the mother board) was compiled into our kernel. A script to start or stop

Listing 4-17. /etc/init.d/network

```
#! /bin/bash
#--------------------------------------------
# this simple script starts or stops the networking based on
# the commands sent to it. if start is sent, then the file
# /etc/network needs to exist.
#--------------------------------------------

# how where we called?
case "$1" in
  start)
    if [ -f /etc/network ] ; then
      . /etc/network
      echo "Configureing network: $NETWORK_IP $NETWORK_MASK "
      /sbin/ifconfig eth0 $NETWORK_IP netmask $NETWORK_MASK
# broadcast $NETWORK_BROADCAST
#      /sbin/route add -net 127.0.0.0 netmask 255.0.0.0 dev lo
      /sbin/route add default gw $NETWORK_GATEWAY
    else
      echo "ERROR: could not start network, /etc/network does not exist"
      exit 1
    fi
    ;;
  stop)
    echo "Shutting down network: $NETWORK_IP $NETWORK_MASK"
    ifconfig eth0 down
    ;;
  *)
    echo "useage: /etc/init.d/network {start|stop}"
    exit 1
esac

exit 0
```

The script relies on the existence of another file that sets up the parameters used for the network. The network file can be located anywhere but i chose to put it in /etc and to adjust the network setting, this is the only file that needs to be changed.

Listing 4-18. /etc/network

```
NETWORK_IP=192.168.2.33
NETWORK_MASK=255.255.255.0
NETWORK_BROADCAST=192.168.2.255
NETWORK_GATEWAY=192.168.2.1
```

Listing 4-19. /etc/hosts

```
127.0.0.1    localhost.localdomain  localhost
192.168.2.33 merlin.home.com        merlin
192.168.2.3  xena.home.com          xena
192.168.2.2  conan.home.com         conan
192.168.2.22 gandolf.home.com       gandolf
```

FTPd

www.wu-ftpd.org

Network Ports

Now we need to add network ports to our system so that our daemons will function properly. Go to /dev and issue the following commands:

```
>> mknod /dev/ptmx c 5 2
>> chmod 666 /dev/ptmx
>> mkdir /dev/pts
```

Then add the following to the /etc/fstab file.

```
none /dev/pts devpts mode=0622 0 0
```

If you don’t want to reboot, then issue the following command, otherwise rc.S will automatically mount it when the system restarts.

```
>> mount /dev/pts
```

If you forget to set this up and you try to telnet in, telnetd will say:

```
telnetd: all network ports in use
```

Web Server: thttpd

This is a small web server with CGI and simple authentication capabilities. I do not think it can be run from inetd, but the binary is about 50k in size. It is available from:

www.acme.com/software/thttpd/

4.20 Security

If for some reason security is an issue, there are several steps you can take to secure your system. Your best source for this information is any linux/unix administration/security guide.

Wrappers

you can include you could try using the “wrapper” daemon tcpd. This will require you to add the configuration files /etc/hosts.allow and /etc/hosts.deny. You will also have to change settings in the /etc/inetd.conf file to look like the following:

```
finger stream  tcp    nowait  root   /usr/sbin/tcpd   in.fingerd
```

Here tcpd answers requests for fingerd, but only from computers listed in /etc/hosts.allow and never from /etc/hosts.deny.

SSH

Of course you can start to really get serious about security by installing things like secure shell (ssh) and secure ftp (sftp). They establish secure, encrypted communications between hosts prior to sending any information (i.e. logins and passwords). This is probably overkill though, unless your embedded client must communicate with other systems that only allow these protocols.

4.21 Read Only Root File System

Once you have linux setup how you need it you want to make the root filing system read-only. This should protect your linux system against power losses, since no important files or binaries can be modified or corrupted. A RAM disk or other partition that is read/write could be used to store user configuration files that need to changed and what not. Some of the things to put into these areas are:

- network settings (/etc/network)
- /tmp
- recorded data

This also increases the life of flash memory systems since they only have a limited number of read/write cycles. Granted that compact flash cards should be good for a million cycles, it is still better to have log files, temp files, log files, etc written to RAM or magnetic media so that you can maximize your system’s life span.

V
CURRENT STATISTICS

```
>> du -h
1.0k   ./lost+found
1.3M   ./bin
638k   ./boot
1.0k   ./dev/pts
2.0k   ./dev
4.0k   ./etc/init.d/old
7.0k   ./etc/init.d
4.0k   ./etc/dont_need
50k    ./etc
1.0k   ./home/kevin
2.0k   ./home/nina
4.0k   ./home
2.4M   ./lib
1.0k   ./mnt/floppy
2.0k   ./mnt
1.0k   ./proc
693k   ./sbin
1.0k   ./tmp
408k   ./usr/bin
928k   ./usr/sbin
23k    ./usr/man/man5
18k    ./usr/man/man8
90k    ./usr/man/man1
132k   ./usr/man
20k    ./usr/lib
3.0k   ./usr/share/terminfo/l
5.0k   ./usr/share/terminfo/x
3.0k   ./usr/share/terminfo/v
12k    ./usr/share/terminfo
13k    ./usr/share
1.5M   ./usr
1.0k   ./var/shm
11k    ./var/run
13k    ./var
3.0k   ./root
6.6M   .
```

5.1 Current Problems

There are currently a couple of problems:

- init is run more than once during boot up.
- init has a problem running inetd.
- the bash prompt for root appears as ‘$’ instead of ‘#’ and I have no idea why.

VI Communicating with Micro controllers

sa11 port

spi - parallel

kermit and minicom - serial

VII

REAL TIME LINUX

Resources VIII

kernel: www.kernel.org

busybox: busybox.lineo.com

tinylogin: tinylogin.lineo.com

telnetd:

ftpd:

IX References

1. Perens, Bruce, “Building Tiny Linux Systems with Busybox - Part 1,”Embedded Linux Journal, Winter 2000, pp. 70-72.
2. Perens, Bruce, “Building Tiny Linux Systems with Busybox - Part 2,” Embedded Linux Journal, January/February 2001, pp. 38-40.
3. Perens, Bruce, “Building Tiny Linux Systems with Busybox - Part 3,” Embedded Linux Journal, March/April 2001, pp. 56-60.
4. Wells, Nicholas D., “TinyLogin: Booting up in Small Places,” Embedded Linux Journal, March/April 2001, pp. 56-60.
5. Sissoms, Jay, “Using Compact Flash Cards in Your Embedded Linux System,” Embedded Linux Journal, May/June 2001, pp. 18-22.
6. Compact Flash Association (www.compactflash.org)
7. Linux System Administrator Guide (www.linuxdoc.org/LDP/sag/index.html)
8. Bootdisk - HOWTO (linuxdocs.org/HOWTOs/HOWTO-INDEX/howtos.html)