Instructions and Notes working with a KuroBox

Trevor Woerner

Instructions and Notes working with a KuroBox
by Trevor Woerner

Revision History

Revision 0.0.1 15 January 2005 Revised by: TW
Document start.
Revision 1.0.0 20 Febuary 2005 Revised by: TW
First release.

Table of Contents

Chapter 1. Overview

The KuroBox is a small, embedded computer based on a PowerPC MPC8241 CPU. It comes from a company in Japan and is resold in North America by a company in Texas called Buffalo Technology[1]. The main english website for the KuroBox is kurobox.com[2].

This document was created to outline information about my experiences with this box. I don't have a PC running any form of Windows at home, therefore I am not able to install an image and work with my KuroBox as described in the instructions that came with the device. I also don't have a Mac running Linux (or similar) from which to generate a base filesystem. Following the steps in the forums for updating the box without the use of a Windows PC did not leave me with a system that was as up-to-date as I had wanted.

I started by creating my own cross-development toolchain (from x86 to PowerPC 603e). Using that toolchain I prepared a completely new base installation (glibc) and used Busybox for the userspace tools. Once that was done I did some testing to make sure everything would work, then installed my new image and rebooted.

This document describes all the steps necessary to take a brand-new, just assembled KuroBox from nothing to having a fully up-to-date user-space environment with nothing more than an x86 Linux PC and the sources to a few code packages. The resulting environment on the KuroBox is running the latest shared libraries, ready to execute programs cross-compiled on the x86 development machine in any of C, C++, Objective-C, or Fortran languages. The x86 Linux cross-development PC is ready to cross-compile programs for the KuroBox. Programs compiled on the x86 machine can be transfered to the KuroBox and executed.

I hope you find this information useful but please see the disclaimer. It worked for me but in no way do I guarantee it will work perfectly for you. If you'd like to get in touch with me you can send a message to (the word ordering is reversed but the letters in the words are in the right order) ca dot vtnet at kuro456.

Notes

1. http://www.buffalotech.com/buffalo-home.php
2. http://kurobox.com

Chapter 2. Building the Cross Tools

Introduction

For our purposes a compiler is a black box; a program which accepts ASCII text as input and generates binary data as output. In most cases a compiler running on a given platform will generate code for that same platform. The purpose of this section is to create a program which will generate code for an entirely different platform than the one on which it is running -- this is called a cross compiler. We want to build a compiler on one architecture that will create code for another architecture, from sources.

> **Note:** It is also possible to create a compiler on architecture A, that will generate code for architecture B, which runs, not on the architecture it was created on, but on an entirely different architecture C! In other words: compile a program on this machine that will run on another machine which will generate code for an entirely different machine. None of the three of which are the same architecture.
>
> This is called a *Canadian Cross*. In this document we will only be creating a cross-compiler, not doing a Canadian Cross. In other words, we will create a compiler on this machine, that runs on this machine, and generates code for another machine.

Creating a cross-compiler involves a chicken-and-egg problem: the compiler needs to know things about the code it is generating as output, but that information comes from the C library. But if we don't have a compiler how can we compile the C library before the compiler? The GCC people have solved this problem by cleverly using stages: in our first pass we don't try to compile a complete and entire cross-compiler, we only generate a small, static cross-compiler. We then use this small cross-compiler to generate a simple C library for the new architecture. Now that we have just enough of a C library for the new architecture, we can now go ahead and create a full and complete cross-compiler. Using this full compiler we are now able to create a full C library for the new architecture.

Compiling a cross compiler is a bit of a touchy procedure; the exact sequence of steps varies depending on which version of the source files you are using. If you aren't starting from the exact same source version numbers that are listed below, the instructions have less chance of succeeding for you. Another variable to consider which affects your success rate is the base system from which you are performing you cross-compile. In general it is more difficult for older compilers to generate newer compilers. All of the following procedures were carried out on a machine running Fedora Core 2.

My first step, before compiling any of the following, was to start by updating my build tools (**binutils** and **gcc**) to their latest versions, the versions I used below to create my cross-development environment. Make sure the installation directory of your new tools precedes that of your system's installed tools in your `${PATH}` variable.

Nomenclature

Build system

: the machine we are using to *create* the cross-compiler programme.

Target

: the architecture for which the cross-compiler will be generating output.

When configuring the software prior to building it, we will need to pass a large number of flags to some of the configure routines. In order to make it easy to read through the options lists without going cross-eyed I will generally type each single option on a line by itself. You must realise that when actually typing in the command that all the options have to be typed in on one line (or on multiple lines with the use of the backslash continuation character). I believe this notation helps with readability.

Obtaining the Sources

For this procedure I used the following sources:

- binutils 2.15[1] (11 MB)
- gcc[2] 3.4.3[3] (26.2 MB)
- glibc 2.3.3[4] (12.4 MB)
- glibc linux threads 2.3.3[5] (0.2 MB)
- linux kernel[6] (here, just about any recent 2.4.x or 2.6.x kernel will do, but I ended up using) 2.6.10[7] (35 MB)
- busybox[8] 1.00[9] (1.1 MB)

File Layout

You're free, of course, to use any file arrangement and layout you want, but personally I like the following layout (and will use it for the remainder of these instructions):

```
${BASE}
 |- src
 |  |- binutils
 |  |  |- binutils-2.15
 |  |  '- build-powerpc-603e
 |  |- gcc
 |  |  |- gcc-3.4.3
 |  |  |- build-powerpc-603e-static
 |  |  '- build-powerpc-603e-full
 |  |- glibc
 |  |  |- glibc-2.3.3
 |  |  |  '- linuxthreads
 |  |  |- build-powerpc-603e-toolchain
 |  |  '- build-powerpc-603e-target
 |  |- linux
 |  |  '- linux-2.6.10
 |  '- busybox
 |     '- busybox-1.00
 |- toolchain
 |  '- powerpc-603e-linux-gnu
 |- target
 |  '- powerpc-603e-linux-gnu
 '- kuro
```

In the above layout `src` is the directory where the compiling of the sources takes place, `toolchain` will be the location where the cross tools are installed, `target` is where the files which are ready to be run on the target will be installed, and `kuro` is where we're going to assemble our files together for transfer to the KuroBox (a.k.a.

`$IMAGE` in the following discussions). `$BASE` is any location where you have permission to create files and directories.

The GNU tools (**binutils**, **gcc**, and **glibc**) generally suggest that you don't build the programs in the same directory as the sources. That is why each of those tools has their own `build-powerpc-603e` directories. This makes it easy to try different things with the same source while only having to unpack the source tarball once.

Be sure to unpack the glibc-linuxthreads tarball within the `glibc` source directory.

The `powerpc-603e-linux-gnu` subdirectories under the `target` and `toolchain` directories leave open the possibility that you will want, in the future, to support different target architectures. In that case each supported architecture would end up in its own directory.

When I was done with all the configuration, compiling, and installation steps, the above directories and their files consumed 2.1 GB of disk space. Of course you don't have to keep all the sources and build directories around.

Compile Procedure

binutils-2.15

- Move to the build directory...

```
$ cd ${BASE}/src/binutils/build-powerpc-603e
```

- ...configure...

```
$ ../binutils-2.15/configure
    --prefix=${BASE}/toolchain/powerpc-603e-linux-gnu
    --target=powerpc-603e-linux-gnu
    --disable-nls
    2>&1 | tee LOG.configure
```

- ...make...

```
$ make 2>&1 | tee LOG.make
```

- ...and install.

```
$ make install 2>&1 | tee LOG.install
```

In the `configure` step we specified that the target is a 603e and not an 8241. The reason for this is that the tools we're about to build have no idea about the PowerPC 8241; they do, however, know about the 603e. As it turns out, the PowerPC 8241 is just a bunch of extra stuff and peripherals built around a PowerPC 603e core. So the best thing to do is to tell the tools that we're targeting a 603e, this keeps the tools from targeting any generic PowerPC CPU.

Procedure Glue

Now we need to perform a couple of steps to prepare for the next phases of compilation.

- First of all we need to add to our `$PATH` the location where the cross-**binutils** tools were just installed so that we can start using them. We don't have to add their location to the beginning of our `$PATH` because the tools are all prefixed with `powerpc-603e-linux-gnu-` and future compilation stages will know to look for tools with that prefix (thanks to the `--target=` configure option in future configurations). The following assumes you're using **bash** or something bash-like as your shell.

  ```
  $ export PATH=${PATH}:${BASE}/toolchain/powerpc-603e-linux-gnu/bin
  ```

- Now we need to prepare a set of `include` files culled from both the system `include` directory and the kernel `include` directory.

  We start with the system `include` files:

  ```
  $ cd ${BASE}/toolchain/powerpc-603e-linux-gnu/powerpc-603e-linux-gnu
  $ cp -a /usr/include .
  ```

  This copies a lot of files, most of which aren't needed, but it doesn't hurt that they're there. You could go through and prune if you like. The next compilation, **gcc**, will also look for a `sys-include` directory; I don't know why, but it needs to be there. It's simply a symlink to the `include` directory you just created:

  ```
  $ ln -s include sys-include
  ```

  Next we prepare the kernel `include` files:

  ```
  $ cd ${BASE}/src/linux/linux-2.6.10
  $ make include/linux/version.h
  ```

  Then we copy these to the required location:

  ```
  $ cd ${BASE}/toolchain/powerpc-603e-linux-gnu/powerpc-603e-linux-gnu/include
  $ rm -fr linux asm
  $ cp -a ${BASE}/src/linux/linux-2.6.10/include/asm-ppc asm
  $ cp -a ${BASE}/src/linux/linux-2.6.10/include/asm-generic .
  $ cp -a ${BASE}/src/linux/linux-2.6.10/include/linux .
  ```

gcc-3.4.3 Static

In this step we create a small cross-**gcc**, just enough to build a small **glibc** which will allow us to then compile a full cross-**gcc** (which we then use to build a full cross-**glibc**):

```
$ cd ${BASE}/src/gcc/build-powerpc-603e-static
$ ../gcc-3.4.3/configure
```

```
    --prefix=${BASE}/toolchain/powerpc-603e-linux-gnu
    --target=powerpc-603e-linux-gnu
    --with-cpu=603e
    --disable-altivec
    --enable-languages=c
    --disable-threads
    --disable-shared
    --disable-shared-libgcc
    --without-stabs
    --without-dwarf2
    --disable-multilib
    --disable-nls
    2>&1 | tee LOG.make
```

```
$ make 2>&1 | tee LOG.make
```

```
$ make install 2>&1 | tee LOG.install
```

glibc-2.3.3 Toolchain

Now we are about to compile a small set of libraries targeting the PowerPC 603e. After we change into the build directory,

```
$ cd ${BASE}/src/glibc/build-powerpc-603e-toolchain
```

the first thing we need to do is to create a file named `configparms` and put the following line into this file:

```
gnulib := -lgcc
```

Now we're ready to configure...

```
$ ../glibc-2.3.3/configure
    --prefix=${BASE}/toolchain/powerpc-603e-linux-gnu/powerpc-603e-linux-gnu
    --build=i686-pc-linux-gnu
    --host=powerpc-603e-linux-gnu
    --enable-add-ons=linuxthreads
    --disable-profile
    --without-cvs
    --disable-debug
    --without-gd
    --without-tls
    --without-__thread
    2>&1 | tee LOG.configure
```

```
$ make 2>&1 | tee LOG.make
```

```
$ make install 2>&1 | tee LOG.install
```

Full gcc-3.4.3

Now we're ready to build a full cross-compiler:

```
$ cd ${BASE}/src/gcc/build-powerpc-603e-full
```

```
$ ../gcc-3.4.3/configure
    --prefix=${BASE}/target/powerpc-603e-linux-gnu
    --target=powerpc-603e-linux-gnu
    --with-cpu=603e
    --disable-nls
    --enable-long-long
    --enable-c99
    --enable-__cxa_atexit
    --enable-threads=posix
    --enable-languages=c,c++,objc,f77
    --enable-symvers=gnu
    --disable-altivec
    2>&1 | tee LOG.configure
```

```
$ make 2>&1 | tee LOG.make
```

```
$ make install 2>&1 | tee LOG.install
$ make prefix=${BASE}/target/powerpc-603e-linux-gnu install 2>&1 | tee LOG.install.full
```

glibc-2.3.3 for the Target

```
$ cd ${BASE}/src/build-powerpc-603e-target
```

```
$ ../glibc-2.3.3/configure
    --prefix=/usr
    --build=i686-pc-linux-gnu
    --host=powerpc-603e-linux-gnu
    --enable-add-ons=linuxthreads
```

```
    2>&1 | tee LOG.configure
```

```
$ make 2>&1 | tee LOG.make
```

```
$ make install_root=${BASE}/target/powerpc-603e-linux-gnu/powerpc-603e-linux-gnu instal
```

Conclusion

Note that there are a lot of subtle differences in the options between the different invocations. For example, building **binutils** and **gcc** requires a `--target=` option whereas **glibc** requires `--build=` and `--host=` options. There are also subtle differences in the `--prefix=` options, especially with the full build of **glibc** where you lie and say they're going to be installed in one place but then override a `Makefile` variable on the command-line at install-time to install the libraries in a different place. This is very important since the shared library mechanism on the target machine will not work properly if you don't do it this way (or at the very least you'll need to place your system-wide shared libraries in a peculiar place on the target machine for your binaries to work).

So what do we have now?

- In `${BASE}/toolchain/powerpc-603e-linux-gnu/bin` are the tools necessary to cross-compile applications for the target machine, the KuroBox.
- In `${BASE}/target/powerpc-603e-linux-gnu/powerpc-603e-linux-gnu` we have the skeleton of the target filesystem image.

We can demonstrate something is different by cross-compiling a simple "Hello, world!" application.

```
[cross-test]$ cat hello.c
#include <stdio.h>

int
main (void)
{
        printf ("Hello, world!\n");
        return 0;
}
[cross-test]$ gcc -o hello hello.c
[cross-test]$ file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, d
[cross-test]$ ./hello
Hello, world!
[cross-test]$
```

But:

```
[cross-test]$ export PATH=$PATH:${BASE}/toolchain/powerpc-603e-linux-gnu/bin
[cross-test]$ powerpc-603e-linux-gnu-gcc -o hello hello.c
[cross-test]$ file hello
hello: ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV), for GNU/Linu
[cross-test]$ ./hello
bash: ./hello: cannot execute binary file
```

Notes

1. http://ftp.gnu.org/gnu/binutils/binutils-2.15.tar.bz2
2. http://gcc.gnu.org
3. http://ftp.gnu.org/gnu/gcc/gcc-3.4.3/gcc-3.4.3.tar.bz2
4. http://ftp.gnu.org/gnu/glibc/glibc-2.3.3.tar.bz2
5. http://ftp.gnu.org/gnu/glibc/glibc-linuxthreads-2.3.3.tar.bz2
6. http://www.kernel.org
7. http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.10.tar.bz2
8. http://www.busybox.net/
9. http://www.busybox.net/downloads/busybox-1.00.tar.bz2

Chapter 3. Filling, Configuring, and Pruning the Base File Image

Preparation

The directory `${BASE}/target/powerpc-603e-linux-gnu/powerpc-603e-linux-gnu` contains a boilerplate files and directories for our target system's image. But many additional files and directories need to be added before it is a usable image. Personally, I prefer to leave this directory in its pristine state and rather build the target image someplace else: ${IMAGE}. This way I always have the boilerplate directories and files to come back to in case I mess something up. Therefore after creating the ${IMAGE} directory I:

```
$ cp -a ${BASE}/target/powerpc-603e-linux-gnu/powerpc-603e-linux-gnu/* ${IMAGE}
```

Busybox

If we look in the directory ${IMAGE} (`${BASE}/target/powerpc-603e-linux-gnu/powerpc-603e-linux`) we'll see that we have the beginnings of our target filesystem. But what we're missing are things like: an **init** system, a shell, and all the dozens of little utilities that make UNIX life easier.

When the Linux kernel is done booting it will look in a number of locations for the **init** program and run it. If it can't find the **init** program it will panic. The **init** system is generally responsible for running a bunch of user-space scripts that help to initialise the system (e.g. configure networking, starting daemons, etc). Sometimes **init** systems incorporate the idea of "run-levels" which a user can select from to determine the type of system and range of services to start or terminate. A shell is useful not just as a program to run after you logon, but is also often used by scripts, especially those which are part of the **init** process. A system will often run a number of daemons such as **getty** processes or **sshd**.

There are two ways to go about getting these things, we can either search in dozens of places downloading all the sources to the individual programs themselves or we can use something like *Busybox*. *Busybox* was designed for this purpose with embedded systems in mind. As such it provides many of the things we are now looking for in one convenient package. Since it was developed with code size considerations in mind, many of the utilities aren't quite as full-featured as the commands we might be accustomed to, but it's still a good place to start.

Unfortunately *Busybox* doesn't support building in a different directory, so you'll want different *Busybox* directories for each platform you want to support. To build *Busybox* go into the directory containing its sources and type:

```
$ make menuconfig
```

You will then be presented with a curses-based screen that will present you with various categories. Inside those categories are items you can select or unselect for inclusion into your system. Offering specific advice here is difficult because it depends on what you're looking for and what you want out of your system. However, some pieces of advice I can offer are:

- enable "`General Configuration -> Use the devpts filesystem for Unix98 PTYs`"
- disable "`General Configuration -> Support for devfs`"
- disable "`Build Options -> Build BusyBox as a static binary (no shared libs)`"
- enable "`Build Options -> Do you want to build BusyBox with a Cross Compiler?`" and fill in the next dialog box with the fully-qualified path value of "`${BASE}/toolchain/powerpc-603e-linux-gnu/bin/powerpc-603e-linux-gnu-`" (no extra CFLAGS are necessary)
- select "`Installation Options -> Don't use /usr`" and input a fully-qualified path name for the "`BusyBox installation prefix`"
- enable "`Networking Utilities -> telnetd`" but don't enable the subsequent "`Support call from inetd only`" option
- be sure to add in "`Init Utilities -> halt, poweroff, and reboot`"
- add in more, rather than less, things; you've got a HDD, space probably isn't an issue

Once you're done the regular "`make dep`", "`make`", and "`make install`" should be all you need.

Additional Files

Your image filesystem should be starting to look much better. Now you have libraries, the basic layout, an **init** system (you did ask *Busybox* to build one, right?), and a bunch of standard UNIX utilities. There are still a few small things you need to tie everything together.

${IMAGE}/dev

With a 2.4.x kernel in the KuroBox and not using devfs we need to create all the necessary device nodes for the system in the image file itself. The easiest way to do this is to simply copy the `${IMAGE}/dev` directory from the Kuro image. Note that you have to be root to create (or copy) device nodes.

Since I suggested enabling the use of devpts above in *Busybox* we also have to create the directory `${IMAGE}/dev/pts`.

More Directories

The following additional directories will need to be created:

- `${IMAGE}/etc/init.d` because the *Busybox* init system will look in this directory for the init script (which we'll create later)
- `${IMAGE}/var/log` which is used by the various loggers
- `${IMAGE}/proc` which is required by the kernel
- `${IMAGE}/tmp`

${IMAGE}/etc Files

Create the following files with the specified contents:

> **Note:** The blank for `hosts` is not a mistake. I'm not sure if things have changed, but I remember there was a time when *Busybox* wouldn't work right if that file wasn't present, even though it might be empty.

- **${IMAGE}/etc/group**

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

- **${IMAGE}/etc/passwd**

```
root::0:0:root:/:/bin/sh
```

- **${IMAGE}/etc/hosts**

- **${IMAGE}/etc/fstab**

```
/dev/hda1       /               ext3    defaults,noatime,errors=remount-ro      0 0
proc            /proc           proc    defaults                        0 0
none            /dev/pts        devpts  mode=20                         0 0
/dev/hda2       swap            swap    defaults                        0 0
```

Init Script

By default the *Busybox* init mechanism will look for an init script at `/etc/init.d/rcS`. Use this file to setup everything that needs to be setup and to run any daemons your system needs to run.

Here is what mine looks like (of course change the XXX and YYY in the eth0 IP address to match your needs):

```
#! /bin/sh

# mount filesystems and turn on swap
/bin/mount -a
/sbin/swapon -a

# networking
/sbin/ifconfig lo 127.0.0.1 up
/sbin/ifconfig eth0 192.168.XXX.YYY up

# telnet server
/sbin/telnetd &

# kuro voodoo
/sbin/ppc_uartd

# hdd rw
/bin/mount -o remount -o rw /dev/hda1
```

> **Warning**
>
> Make extra extra sure that you **chmod +x** this file!!

Kuro Voodoo

Due to the relationship between the Kuro and its ATMEL AVR chip the following modifications should be made:

- Add the program **ppc_uartd** from the Kuro image into your image (the init script will want to run this program).
- Change `${IMAGE}/sbin/halt` to:

```
#!/bin/ash
echo -n "EEEE" > /dev/ttyS1
/bin/busybox halt
```

- Change `${IMAGE}/sbin/poweroff` to:

```
#!/bin/ash
echo -n "EEEE" > /dev/ttyS1
/bin/busybox poweroff
```

- Change `${IMAGE}/sbin/reboot` to:

```
#!/bin/ash
echo -n "CCCC" > /dev/ttyS1
/bin/busybox reboot
```

Getting the Image Together

Now that your image is all set and populated with all the files your base installation requires there are just two things that are left to do:

- Make all the files of your image owned by root:

```
# cd ${IMAGE}
# chown -R root.root .
```

- Package the whole image up into a tarball:

```
# cd ${IMAGE}
# tar -c . > ../kuroimg.tar
```

If the KuroBox and your machine are far apart and separated by a very slow network you can additionally gzip the resulting tar file to help reduce the transfer time. But

if the network between your machine and the KuroBox is fast this just wastes more time for no benefit.

Conclusion

You should now have a very basic filesystem image that's ready to install onto your KuroBox which has been cross-compiled by the latest compiler tools and uses the latest base system libraries.

I should probably point out that what I've demonstrated here is a very bare-bones minimal system. It is very likely that you will want to add more things to your setup. If you want your KuroBox to operate on a network you'll probably want to create a `${IMAGE}/etc/resolv.conf` and add at least one **route** command to your init script (for the default route).

Chapter 4. Installing your Image

After installing a new HDD into the KuroBox (or perhaps one from another machine) the KuroBox will boot into a reserved mode. This reserved mode will run a telnet daemon and an ftp daemon which you can use to transfer your new image to the disk.

Preparation

After you've telnet'ed into your KuroBox, the first thing to do is to prepare the HDD to accept an image. For this we will run the **mfdisk -c** command on our HDD:

```
# mfdisk -c /dev/hda
```

This command works exactly like the well-known **fdisk** command so I won't bother with a detailed explanation. Just make sure that you reflect in the `${IMAGE}/etc/fstab` file the same work you do partitioning the disk. You must also make sure that the mount point directories exist before **mount** tries to mount the disk partitions during the init script.

For example purposes for my own use I simply created two partitions:

partition	purpose
/dev/hda1	root filesystem
/dev/hda2	swap

After you've created all your partitions you then have to format them. I'm not 100% sure of all the formats that are supported by the KuroBox kernel, but I do know that `ext3` works and is a good filesystem for most purposes. The command to format your partitions with `ext3` is **mkfs -j**:

```
# mkfs -j /dev/hda1
```

Be sure create and format your swap partition too:

```
# mkswap /dev/hda2
```

Now that your partitions are ready to go, mount them:

```
# mkdir /mnt/root
# mount /dev/hda1 /mnt/root
```

Installation

We're ready to transfer and install the image to the KuroBox. With `/dev/hda1` still mounted, **ftp** to the box (from your development PC) and upload the image:

```
[system-base]$ ftp 192.168.x.y
```

```
Connected to 192.168.x.y (192.168.x.y).
220 KURO-BOX-EM FTP server (Version 6.4/OpenBSD/Linux-ftpd-0.17) ready.
Name (192.168.x.y:trevor): root
331 Password required for root.
Password:
230- Linux 2.4.17 ppc unknown
230 User root logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd /mnt/root
250 CWD command successful.
ftp> put kuroimg.tar
local: kuroimg.tar remote: kuroimg.tar
227 Entering Passive Mode (192,168,x,y,4,3)
150 Opening BINARY mode data connection for 'kuroimg.tar'.
226 Transfer complete.
85084160 bytes sent in 18.6 secs (4.5e+03 Kbytes/sec)
ftp> quit
221 Goodbye.
[system-base]$
```

Back on the KuroBox (through telnet) unpack the image:

```
# cd /mnt/root
# tar -xf kuroimg.tar
```

We no longer need the `kuroimg.tar` file so we can delete it.

Have a look around your new disk, make sure everything is owned by `root`, make sure the things that need to be executable have their execute bit set.

There is one more thing to do: copy **ppc_uartd** from the reserved mode to your new image:

```
# cp /usr/sbin/ppc_uartd /mnt/root/sbin
```

Testing

Before taking the big plunge (rebooting) we can run a few tests to make sure things are working reasonably well. Remember that **hello** program that we cross-compiled? We can **ftp put** that to the new system and make sure it runs:

```
# cd /mnt/root/bin
# chmod +x hello
# ./hello
./hello: error while loading shared libraries: libgcc_s.so.1: cannot load shared object
```

Whoops! The dynamic loader doesn't know where to find our libraries. We can help it out:

```
# export LD_LIBRARY_PATH=/mnt/root/lib
# ./hello
./hello: /lib/ld.so.1: version 'GLIBC_PRIVATE' not found (required by /mnt/root/lib/lib
```

Damn! Still doesn't work. That's because by default we're using the old dynamic loader. To use the new loader:

```
# /mnt/root/lib/ld.so.1 ./hello
Hello, world!
```

Cool.

```
# /mnt/root/lib/ld.so.1 --list ./hello
        libgcc_s.so.1 => /mnt/root/lib/libgcc_s.so.1 (0x0ffd3000)
        libc.so.6 => /mnt/root/lib/libc.so.6 (0x0fe70000)
        /lib/ld.so.1 => /mnt/root/lib/ld.so.1 (0x08000000)
```

```
# ls -l
ls: /lib/ld.so.1: version 'GLIBC_PRIVATE' not found (required by /mnt/root/lib/libc.so.
# ifconfig
ifconfig: /lib/ld.so.1: version 'GLIBC_PRIVATE' not found (required by /mnt/root/lib/li
# vi
vi: /lib/ld.so.1: version 'GLIBC_PRIVATE' not found (required by /mnt/root/lib/libc.so.
```

Oh no! Nothing works anymore!!

Oh yea...

```
# unset LD_LIBRARY_PATH
```

```
:-)
```

So we're now sure that what we cross-compiled will actually work on this architecture. We're also sure that the dynamic loader and shared libraries work. We have a good deal of confidence that this new image will work well. But there's only one way to really know...

The Final Test

To get the KuroBox to try booting our new image on the HDD we need to modify some flash data which tells the booting mechanism to load the image off the HDD instead of using the reserve mode image:

```
# echo -n "OKOK" > /dev/fl3
# reboot
```

We now wait a couple seconds. Wait for the `LINK/ACT` light to blink. Then from our development PC...

```
[trevor]$ telnet 192.168.x.y
Trying 192.168.x.y...
```

```
Connected to 192.168.x.y.
Escape character is '^]'.

(none) login: root


BusyBox v1.00 (2005.01.14-05:42+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

~ # mount
/dev/hda1 on / type ext3 (rw)
proc on /proc type proc (rw)
none on /dev/pts type devpts (rw)
~ # hello
Hello, world!
~ # ldd /bin/hello
        libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x0ffd3000)
        libc.so.6 => /lib/libc.so.6 (0x0fe70000)
        /lib/ld.so.1 => /lib/ld.so.1 (0x30000000)
~ #
```

YEA!!!

> **Note:** We didn't setup any password with this system so all you have to do is type `root` at the `login:` prompt and you'll get a virtual terminal.

Appendix A. Appendix

Disclaimer

The steps and instructions outlined in this document worked for me; as a result no person or property was damaged, no losses were incurred, and my house did not burn down. In no way do I imply (either implicitly or explicitly) that if you read the information contained in this document, and follow all the steps as described, that you will be so lucky. :-)

By following these instructions, all risks assumed are your own.

Additional Information

If you'd like more information about the topics covered in this document you might find the following links and hints useful. I'm sure it's only a matter of time before these links become obsolete, in that case don't forget to try google. Also, for information regarding various programs, don't forget to read their **man** and **info** pages. Never forget to read the documentation that comes with various source packages, and the fact that *you have the source!* (read it too!). Many projects also have mailing lists which offer a lot of useful information and tips.

- Information about **gcc** can be found on the gcc website[1].
- *Busybox* comes with great documentation. You can also find information on the Busybox website[2].
- Two great resources for information about developing crosstools are:
  - Crosstools[3] by Dan Kegel
  - The CrossGCC FAQ[4]

Overall Impressions of the KuroBox

In a word: disappointment. Very disappointed.

I like the PowerPC architecture, always have. I've been fond of it ever since it just a rumour. I've been looking forward and hoping for a moderately-priced, open, PowerPC-based embedded device for development purposes -- this is not it. It's not even close.

The KuroBox is being hailed as an open development platform. The mere fact something runs Linux doesn't automatically imply it's an open development platform. Take, for example, the serial console port. It isn't populated. Even once you populate the connector and the required resistor you still don't have anything that's useful because you now have signals that are only 3.3V and inverted. You still need additional, external logic in order for it to be usable. Then there's the JTAG port which requires a surface mount resistor pack, not something everyone's going to have handy. Provided you've tackled these hurdles you still need specialised, closed hardware to connect to the JTAG which only works with Windows (R) software.

There's no board bring-up, boot-monitor, bootloader (or whatever you want to call it). Therefore if you happen to install a bad kernel your whole box has now been rendered useless. In other words, don't bother trying to develop for this board, just use the tools provided and be happy. This is not an open development system. You can play in user-space all you want, if you mess something up just reformat the HDD on another machine and you're ready to start over. Mess up the kernel? You're out

of luck. And if you wanted to ignore all that and still work on the kernel, you don't have a serial console, so you just have to reboot, wait, and hope it all worked out well enough and that your telnet server will start up soon.

If that's not enough, the KuroBox features an ATMEL chip in it that controls many aspects of the running device. For example, if you don't perform specific communications with this ATMEL chip it will simply shutdown your box after a short period. No information regarding this ATMEL chip is officially published.

In short, the search for a cheap, open, PowerPC-based development system continues.

Saving Space on the Target

The newly installed image running on the KuroBox consumes 115.4 MB of disk space (according to **df -h**). This isn't bad, but it is actually quite large. There are a number of files that can be removed from the distribution if you need more space, or just want to see how small you can make things go.

For starters all the `*.a` files in the various `lib` directories are all only necessary when compiling code (and even at that, they're only necessary if you're specifically statically compiling your binaries). Since the development PC can be used to compile your KuroBox applications these files really aren't necessary on the target. If you compile in your development PC, you'll also not need the `include` directories.

There are many other things. For example you can prune what you don't need from the Busybox executable. If you aren't going to be running Fortran, Objective-C, or C++ applications you can remove their support libraries too. You probably don't need the `info` files.

A Word Regarding Linux Threads

Writing threaded applications requires support from the OS and the language library, in Linux's case the C library, glibc. Linux's first, widely-used, quasi-POSIX attempt at threads was called "linuxthreads", and was supplied as part of glibc. Recently, Ingo Molnar and Ulrich Drepper (both working for RedHat Inc.) studied the "linuxthreads" implementation, found many problems with it, and started something new called the NPTL (Native POSIX Thread Library). In every situation NPTL has been found to be superior to the old linuxthreads. NPTL, however, has not yet been ported to all architectures, in fact it looks like it only works on x86-ish machines. This is why our compilation above specifies using linuxthreads when building glibc instead of using the superior NPTL.

Notes

1. http://gcc.gnu.org
2. http://www.busybox.net
3. http://kegel.com/crosstool/
4. http://vmlinux.org/joachim/mirror/www.objsw.com/CrossGCC/