Building Linux from Scratch for the OMAP35x

Application Note 412

Logic // Products
Published: February 2010

Abstract

This Application Note will walk readers through building a functional Linux system targeting the Texas Instruments Zoom™ OMAP35x Development Kit. The document focuses on building the entire system from scratch using only patches to freely downloadable, open source software.

Revision History

REV	EDITOR	DESCRIPTION	APPROVAL	DATE
A	JCA	Initial release; General formatting and grammatical changes	JCA	02/05/10

Table of Contents

1 Introduction

Building an embedded Linux system for real target hardware can be an extremely satisfying, yet daunting task. As a product solutions company, we at Logic have helped many customers successfully field devices based on Linux and would sincerely like to discuss how we might help you with your project. Please feel free to engage with us as you start to investigate fielding an embedded Linux-based device. We would love to help in any way possible—see Section 13 for contact information.

Internally, Logic maintains a private Linux tool and software distribution based on LTIB (Linux Target Image Builder) [http://savannah.nongnu.org/projects/ltib]. Logic makes these tools available to select customers based on their experience with prior Linux development and opportunity for engagement with Logic. The rules regarding who is given or denied access to our internal tools are flexible and the tools themselves are typically offered free of charge. If you would like to learn more, please contact Logic sales [product.sales@logicpd.com]. The reasoning behind not simply offering the tools to everyone basically comes down to support risk. Any product that comes from Logic is backed by a firm commitment of support. Logic prefers to focus on offering its customers product solutions—so supporting Linux tools creates a definite risk of distracting our resources from their main mission of providing solutions to our customers.

Logic also maintains an active partnership with Timesys [http://linuxlink.timesys.com/3/Linux/Logic] to provide mature and professionally supported tools to our customers. Logic works very closely with Timesys and, as a result, Timesys always has access to the latest source code and patches for our hardware. Logic's customers have been successful using the Timesys tools and all are quite satisfied with their experience. Many of Logic's customers are eligible for free demonstrations of Timesys' tools and we strongly encourage everyone to look closely at the Timesys offerings.

This Application Note describes a third method of building a Linux system for Logic's Zoom OMAP35x Development Kits; creating one from scratch. Logic offers downloads that contain the patches to the open-source software that it works with internally. Experienced embedded Linux developers may prefer to work with the patches and *pristine* sources using the tools and methods they are familiar with. We also find that inexperienced developers, often idealistically, also want to build entire systems from scratch. Though Logic would never suggest trying to launch an actual product using this method, this Application Note should fully document the steps necessary to get a basic system in place.

1.1 Audience

This Application Note was written for software engineers with experience pulling together the software components necessary to build an embedded Linux system. The document should provide enough information concerning tools, versions, and patches that seasoned Linux developers can work with Logic's output to accomplish their goals.

It is almost a given that novice developers will also read this Application Note and undertake the processes described therein. Effort was made to accurately document each step in sufficient detail that simply repeating the commands listed should achieve the desired result. However, readers should be cautioned that no attempt has been made to note the myriad of system administrative details necessary on the desktop-side to properly configure, build, and deploy software using the methods described below. Friedrich Nietzsche said; "That which does not kill us makes us stronger." Trying to build an entire Linux system from scratch without sufficient system administration knowledge definitely shouldn't kill you, but you will feel some pain. The author can only suggest that when you encounter problems you remember that patience is a virtue and Google is your friend.

1.2 Scope and Background Information

This Application Note assumes that you have experience administering your own Linux workstation. At the very least, you should be able to understand when an error is caused by something missing, or out-of-date, on your development system and how to correct the situation. If your command-line Fu is weak, plan to spend some time boning up on BASH-scripting and prepare to read many MAN pages.

1.3 Hardware Used

This Application Note uses the Texas Instruments Zoom™ OMAP35x Development Kit from Logic. Specifically, the development was verified on:

SOM Model Number:	SOMOMAP3530-10-1672IFCR-A
SOM Part Number:	1010194

1.4 Software Used

This Application Note refers to the following revisions of embedded software:

Package	**Version**
Linux Kernel:	2.6.28-rc8
U-Boot Universal Bootloader:	1.1.4
BusyBox:	1.14.1
GNU Compiler Collection (GCC):	4.4.0
GNU Binutils:	2.19.1
GNU C Library (GLIBC):	2.9
OMAP35x Patches from Logic:	1.5
LogicLoader Bootloader/Monitor (LoLo):	2.4.6-OMAP3503 0001

All development was done using Ubuntu 8.10 [http://www.ubuntu.com] - the *Intrepid Ibex* - released in October 2008. All of the system's packages were up-to-date at the time of writing. Several development packages were installed to allow building the various components (automake, awk, make, texinfo, etc.). Again, the document assumes that readers have enough UNIX/Linux system administration experience to successfully figure out when a necessary tool is absent (or out-of-date) and how to install (or upgrade) it as necessary.

2 *Mise en Place*

"*Mise en place* is the primary organizational principle in all cooking. It means first things first or, literally, 'everything in its place'" (Reinhart 49). As in the kitchen, we will do in the lab. This section will set the stage by defining some basic environment variables and a directory structure we will use throughout the rest of our project.

To start, we will create a series of environment variables and directories which will be used throughout this project. The environment variables are often used as *shortcuts* to the various directories and packages we will use.

2.1 Project Environment Variables

Create the project's environment variables.

```
$ export  PRJROOT=$HOME/olfs
$ export  ARCHIVE=$PRJROOT/archive
$ export  BUILDTOOLS=$PRJROOT/build-tools
$ export  KERNEL=$PRJROOT/kernel
$ export  PATCHES=$PRJROOT/patches
$ export  ROOTFS=$PRJROOT/rootfs
$ export  TOOLS=$PRJROOT/tools
$ export  UBOOT=$PRJROOT/u-boot
```

2.2 Project Directory Structure

Create the project's directories.

```
$ mkdir  $PRJROOT
$ mkdir  $ARCHIVE
$ mkdir  $BUILDTOOLS
$ mkdir  $KERNEL
$ mkdir  $PATCHES
$ mkdir  $ROOTFS
$ mkdir  $TOOLS
$ mkdir  $UBOOT
```

At this point, you probably want to go ahead and enter the root of your project tree.

```
$ cd  $PRJROOT
$ ls

archive      kernel   rootfs  u-boot
build-tools  patches  tools
```

3 Download the Project's Packages

Now that we have a project directory structure in place, we will download all of the packages we need.

Logic's suggestion is that you keep *everything* you download in the $ARCHIVE directory for future reference. The rest of this tutorial assumes you have stored the packages in this directory.

```
$ cd  $ARCHIVE
```

3.1 Package Overview

We will be using the following software packages throughout the remainder of this tutorial.

- Logic-provided Linux and U-Boot Patches - http://support.logicpd.com/auth/
- linux 2.6.28-rc8 - http://www.kernel.org
- U-Boot 1.1.4 - http://www.denx.de/wiki/U-Boot
- BusyBox 1.14.1 - http://www.busybox.net
- CodeSourcery Sourcery G++ Lite arm-2009q1-203 - http://www.codesourcery.com
- Binutils 2.19.1 - http://www.gnu.org/
- GCC 4.4.0- http://gcc.gnu.org/
- GLIBC 2.9 - http://www.gnu.org

3.2 Download Logic Patches

Logic provides patches for the Linux kernel and U-Boot. To access the patches, please ensure that you have created an account *and* registered your development kit on http://www.logicpd.com. Once you have logged in and gained access to the kit's dedicated downloads page, find the zip file containing the patches. You should be able to find the patch set by going to http://support.logicpd.com/auth/downloads/OMAP35x Zoom Development Kit/#linux and downloading the file denoted by the *OMAP35x Linux Demo Image Patch Set* link. At the time of writing, the version of the patch file found at the link above was 1013108_OMAP35x_Patches_v1.5.zip.

Download the Logic patches and save the zip file in the $ARCHIVE directory.

3.3 Download Linux Kernel

The Linux kernel can be downloaded from many different websites. For this tutorial, we will download a pristine kernel from http://www.kernel.org. Optionally, you may also download the kernel's signature file and verify that you received a properly signed source code package.

```
$ wget \
> http://www.kernel.org/pub/linux/kernel/v2.6/testing/v2.6.28/ \
> linux-2.6.28-rc8.tar.bz2
```

3.3.1 Verifying Downloaded Software

If you would like to verify the digital signature of the software packages you download, you may do so using GPG. To verify, download the *.sign or *.sig file that corresponds to the source code package. Use GPG --verify to check the signature against the package. If you have not loaded the appropriate public key into your keyring, GPG will report an error, but also tell you the hexadecimal fingerprint of the key used to sign the package. You may then import the reported key into your local keyring and use GPG to verify the downloaded source code.

A general and specific example are given below.

```
$  gpg --verify  package.tar.bz2.sig

gpg: Signature made Sun 17 May 2009 08:02:08 PM CDT using DSA key ID WWXXYYZZ
gpg: Can't check signature: public key not found

$ gpg  --keyserver wwwkeys.pgp.net  --recv-keys 0xWWXXYYZZ

$ wget \
> http://www.kernel.org/pub/linux/kernel/v2.6/testing/v2.6.28/ \
> linux-2.6.28-rc8.tar.bz2.sign

$  gpg --verify linux-2.6.28-rc8.tar.bz2.sign

gpg: Signature made Sun 17 May 2009 08:02:08 PM CDT using DSA key ID 517D0F0E
gpg: Can't check signature: public key not found

$ gpg  --keyserver wwwkeys.pgp.net  --recv-keys 0x517D0F0E
gpg: requesting key 517D0F0E from hkp server wwwkeys.pgp.net
gpg: key 517D0F0E: public key "Linux Kernel Archives Verification Key"
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg: Total number processed: 1
gpg:               imported: 1

$ gpg  --verify linux-2.6.28-rc8.tar.bz2.sign

gpg: Signature made Wed 10 Dec 2008 06:49:04 PM CST using DSA key ID 517D0F0E
gpg: Good signature from "Linux Kernel Archives Verification Key"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: C75D C40A 11D7 AF88 9981  ED5B C86B A06A 517D 0F0E
```

3.4 Download U-Boot

Das U-Boot http://www.denx.de/wiki/U-Boot is a bootloader commonly used to load Linux on embedded devices. We will use it to load a Linux kernel and root file system onto our target hardware and boot the system.

```
$ wget  ftp://ftp.denx.de/pub/u-boot/u-boot-1.1.4.tar.bz2
```

3.5 Download BusyBox

BusyBox combines tiny versions of many common UNIX utilities into a single small executable. It provides replacements for most of the utilities you usually find in GNU fileutils, shellutils, etc. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins; however, the options that are included provide the expected functionality and behave very much like their GNU counterparts. BusyBox provides a fairly complete environment for any small or embedded system.

BusyBox has been written with size-optimization and limited resources in mind. It is also extremely modular so you can easily include or exclude commands (or features) at compile time. This makes it easy to customize your embedded systems. To create a working system, just add some device nodes in /dev, a few configuration files in /etc, and a Linux kernel.

BusyBox is maintained by Denys Vlasenko, and licensed under the GNU GENERAL PUBLIC LICENSE version 2.

–www.busybox.net/about.html

```
$ wget  http://www.busybox.net/downloads/busybox-1.14.1.tar.bz2
```

3.6 Download CodeSourcery Tools; Sourcery G++ Lite Edition

You have a choice of tools to use. If you want to use the pre-built tools from CodeSourcery, download the files in this section. This tutorial will use the Lite Edition of CodeSourcery's [http://www.codesourcery.com] excellent cross-compiling tool chain.

```
$ # Note, use "-O" switch to avoid PHP items finding their
$ # way into the name of the downloaded file.

$ wget  -O  arm-2009q1-203-arm-none-linux-gnueabi.bin \
> http://www.codesourcery.com/sgpp/lite/arm/portal/package4573/public/
\
> arm-none-linux-gnueabi/arm-2009q1-203-arm-none-linux-gnueabi.bin

$ # Download the user manual while we are at it

$ wget \
> http://www.codesourcery.com/sgpp/lite/arm/portal/doc4337/ \
> getting-started.pdf
```

3.7 Download Pure GNU Tools

If you plan to build your toolchain from scratch, you will need to download several items to build your GNU cross-toolchain.

```
$ wget  http://ftp.gnu.org/pub/gnu/binutils/binutils-2.19.1.tar.bz2
$ wget  http://ftp.gnu.org/pub/gnu/gcc/gcc-4.4.0/gcc-4.4.0.tar.bz2
$ wget  http://ftp.gnu.org/pub/gnu/glibc/glibc-2.9.tar.bz2
$ wget  hhttp://ftp.gnu.org/pub/gnu/glibc/glibc-ports-2.9.tar.bz2
```

If you would like, you may follow the instructions detailed in Section 3.3.1 "Verifying Downloaded Software" to verify the digital signatures on the GNU tools.

4 Install the CodeSourcery Tools

Before you install the CodeSourcery tools, you should read the Getting Started Guide that comes with them. Chapter 4 of the *Sourcery G++ Getting Started Guide* discusses installing and configuring the tools. This tutorial will assume that you have read that guide and are following the detailed instructions listed therein. This tutorial should only be used as a *CliffsNotes* version of the installation instructions.

4.1 Dealing with the DASH Shell

If you are using Ubuntu or Debian, you may need to get the dash shell "out of the way" before installing these tools.

> Installing on Ubuntu and Debian GNU/Linux Hosts
>
> The Sourcery G++ graphical installer is incompatible with the dash shell, which is the default /bin/sh for recent releases of the Ubuntu and Debian GNU/Linux distributions. To install Sourcery G++ Lite on these systems, you must make /bin/sh a symbolic link to one of the supported shells: bash, csh, tcsh, zsh, or ksh.
>
> For example, on Ubuntu systems, the recommended way to do this is:
>
> ```
> > sudo  dpkg-reconfigure  -plow dash
> ```
>
> Install as /bin/sh? No
>
> This is a limitation of the installer and uninstaller only, not of the installed Sourcery G++ Lite toolchain.
>
> –*CodeSourcery Getting Started Guide*; Chapter 4

4.2 Install Sourcery G++ Lite

Again, refer to the *CodeSourceryGetting Started Guide* for complete details. Steps to install the tools from the command line are repeated below for your convenience.

```
$ cd  $ARCHIVE
$ /bin/sh  ./arm-2009q1-203-arm-none-linux-gnueabi.bin  -i console

Preparing to install...
Extracting the JRE from the installer archive...
Unpacking the JRE...
Extracting the installation resources from the installer archive...
Configuring the installer for this system's environment...

Launching installer...

Preparing CONSOLE Mode Installation...

=================================================================
Sourcery G++ Lite for ARM GNU/Linux(created with InstallAnywhere
-----------------------------------------------------------------
            ...
```

You should accept the license agreement and the standard installation options; the installer will do the rest. You can verify that the tools were installed by using the following commands:

```
$ ls  $HOME
... CodeSourcery  ...
```

```
$ ls  $HOME/CodeSourcery
Sourcery_G++_Lite  Sourcery_G++_Lite_for_ARM_GNU_Linux

$ ls  $HOME/CodeSourcery/Sourcery_G++_Lite/bin
arm-none-linux-gnueabi-addr2line  arm-none-linux-gnueabi-gprof
arm-none-linux-gnueabi-ar         arm-none-linux-gnueabi-ld
arm-none-linux-gnueabi-as         arm-none-linux-gnueabi-nm
arm-none-linux-gnueabi-c++        arm-none-linux-gnueabi-objcopy
arm-none-linux-gnueabi-c++filt    arm-none-linux-gnueabi-objdump
arm-none-linux-gnueabi-cpp        arm-none-linux-gnueabi-ranlib
arm-none-linux-gnueabi-g++        arm-none-linux-gnueabi-readelf
arm-none-linux-gnueabi-gcc        arm-none-linux-gnueabi-size
         ...
```

4.3 Adjust PATH Environment Variable

You will need to adjust your shell's PATH variable to ensure these newly installed tools can be found.

```
$ export  PATH=$HOME/CodeSourcery/Sourcery_G++_Lite/bin:$PATH
$ which   arm-none-linux-gnueabi-gcc

/home/logic/CodeSourcery/Sourcery_G++_Lite/bin/arm-none-linux-gnueabi-
gcc

$ arm-none-linux-gnueabi-gcc  --version

arm-none-linux-gnueabi-gcc (Sourcery G++ Lite 2009q1-203) 4.3.3
Copyright (C) 2008 Free Software Foundation, Inc.
...
```

NOTE: You may wish to add the above commands to your $HOME/.bashrc file so that PATH is properly set next time you login.

5 Build Toolchain from Scratch

About once each year the author finds himself trying to build a complete GNU cross-toolchain from scratch using the methods described below. It is never a pleasant experience. The classic computer science text *Compilers: Principles, Techniques, and Tools* by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman is commonly known as the *Dragon Book* because at the time of its publication, designing compilers was one of the most complex undertakings in the field. Certainly designing compilers has gotten easier for those employed in that trade; however, for those of us that just want to generate a working cross compiler, the path still feels fraught with danger.

Any successful cross build of GCC from scratch owes a great debt to the many people who have taken the time to document the arcane bits of knowledge necessary to pull three complex pieces of software together into a working toolchain. The author's list of creditors includes; Bill Gatliff, Dan Kegel, Karim Yaghmour, Peter Barada, Kai Ruottu, and Steve Papacharalambous.

Specifically, the sed commands to fix the built linker scripts comes directly from work Steve Papacharalambous published as part of the LTIB project.

5.1 Extract and Patch the GNU Tools

5.1.1 Extract the GNU Tools

Unpack the source code for the GNU tools into the project's "build-tools" directory. Note that "glibc-ports" is extracted *into* the "glibc" directory.

```
$ cd  $ARCHIVE
$ tar  -xjf binutils-2.19.1.tar.bz2  -C $BUILDTOOLS
$ tar  -xjf gcc-4.4.0.tar.bz2  -C $BUILDTOOLS
$ tar  -xjf glibc-2.9.tar.bz2  -C $BUILDTOOLS
$ tar  -xjf glibc-ports-2.9.tar.bz2  -C $BUILDTOOLS/glibc-2.9

$ cd  $BUILDTOOLS
$ ls

binutils-2.19.1  gcc-4.4.0  glibc-2.9
```

5.1.2 Extract the GNU Patches

There are a couple of small patch files we need to apply to binutils and GCC so that they may build for the ARM architecture without warning.

```
$ cd  $ARCHIVE
$ tar  -xjf gnu-patches.tar.bz2  -C $PATCHES

$ cd  $PATCHES/gnu-patches
$ ls

binutils-2.19.1  gcc-4.4.0
```

5.1.3 Patch Binutils

Apply the patches, *in the designated order*, to the U-Boot source code. You MUST be in the root directory of the binutils source code for the commands below to work. You should be in the same directory as the "COPYING" and "MAINTAINERS" files.

```
$ cd  $BUILDTOOLS/binutils-2.19.1/
$ for p in $PATCHES/gnu-patches/binutils-2.19.1/*.patch
> do
> echo "Applying patch ${p}"
> patch -p1 < ${p}
> done
```

5.1.4 Patch GCC

Using the same method as above, apply the patches to the GCC source code.

```
$ cd  $BUILDTOOLS/gcc-4.4.0/
$ for p in $PATCHES/gnu-patches/gcc-4.4.0/*.patch
> do
> echo "Applying patch ${p}"
> patch -p1 < ${p}
> done
```

5.2 Create Temporary Build Directories

The GNU tools are built in directories separate from their source code. We will make several temporary directories now.

```
$ cd  $BUILDTOOLS
$ mkdir  build-binutils  build-gcc{1,2,3}  build-glibc
$ ls

binutils-2.19.1  build-gcc1  build-gcc3   gcc-4.4.0
build-binutils   build-gcc2  build-glibc  glibc-2.9
```

5.3 Configure and Build Binutils

Binutils is a relatively painless and easy package to build and install.

```
$ cd  $BUILDTOOLS/build-binutils
$ ../binutils-2.19.1/configure \
> --target=$TARGET \
> --prefix=$PREFIX \
> --with-sysroot=$SYSROOT
$ make  all
$ make  install
$ ls  $PREFIX/bin

arm-none-linux-gnueabi-addr2line  arm-none-linux-gnueabi-objcopy
arm-none-linux-gnueabi-ar         arm-none-linux-gnueabi-objdump
arm-none-linux-gnueabi-as         arm-none-linux-gnueabi-ranlib
      ...

$ arm-none-linux-gnueabi-as  --version
```

```
GNU assembler (GNU Binutils) 2.19.1
Copyright 2007 Free Software Foundation, Inc.
...
```

5.4 Configure and Build GCC Bootstrap Compiler

GCC needs to be bootstrapped. You will note that we disable several items in the configuration command below. To read more on what each of these options does, consult the GCC installation manual at http://gcc.gnu.org/install. Specifically, review the configuration section http://gcc.gnu.org/install/configure.html.

Briefly, the disabled options below are removed for the following reasons:

libssp – libssp is "stack smashing protection," we disable it because the library's configuration script tries to build and run an executable file as a test. However, since we are building a cross-compiler, any programs it builds will not be able to be executed on our host machine—thus this will fail.

libgomp – libgomp [http://gcc.gnu.org/onlinedocs/libgomp/index.html] is the GNU implementation of the OpenMP (API) [http://openmp.org/wp/] for multi-platform shared-memory parallel programming in C/C++ and Fortran. We disable it here because it requires a threading model and our bootstrapped compiler lacks this *(--disable-threads and --with-newlib)*.

libmudflap – libmudflap provides runtime bounds checking for the C-language when -fmudflap is specified on the GCC command line. It needs a C-library to function and we haven't built that yet, thus it too must be disabled.

```
$ cd  $BUILDTOOLS/build-gcc1
$ ../gcc-4.4.0/configure \
> --target=$TARGET \
> --prefix=$PREFIX \
> --without-headers \
> --with-newlib \
> --disable-shared \
> --disable-threads \
> --disable-libssp \
> --disable-libgomp \
> --disable-libmudflap \
> --enable-languages=c

$ make
$ make  install
$ ls  $PREFIX/bin

...  arm-none-linux-gnueabi-gcc  ...

$ arm-none-linux-gnueabi-gcc  --version

arm-none-linux-gnueabi-gcc (GCC) 4.4.0
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
```

5.5 Configure and Install the Kernel Headers

Full versions of GLIBC and GCC need the kernel headers to build. We will configure and install them now. Jump ahead to Chapter 8. "Extract, Patch, and Build Linux" and follow those instructions up to building the kernel before proceeding.

```
$ cd  $KERNEL/linux-2.6.28-rc8-omap3530lv-som
$ make  ARCH=arm \
> CROSS_COMPILE=$TARGET- \
> INSTALL_HDR_PATH=${SYSROOT}/usr \
> headers_install

$ ls  $SYSROOT/usr/include

asm  asm-generic  drm  linux  mtd  rdma  sound  video
```

5.6 Understanding GCC Multilib

MULTILIB_OPTIONS

For some targets, invoking GCC in different ways produces objects that cannot be linked together. For example, for some targets GCC produces both big and little endian code. For these targets, you must arrange for multiple versions of libgcc.a to be compiled, one for each set of incompatible options. When GCC invokes the linker, it arranges to link in the right version of libgcc.a, based on the command line options used.

–GCC Internals
[http://gcc.gnu.org/onlinedocs/gcc-4.4.0/gccint/Target-Fragment.html#Target-Fragment]

In the case of building an ARM GCC Cross-Compiler, there are several different ways that GCC can generate code. GCC can generate code which adheres to different Application Binary Interface (ABI) formats, various versions of the ARM and Thumb instruction sets (ARMv2–ARMv7), code that supports calling between ARM and Thumb instructions (interworked), actual floating point instructions versus calls to integer-based emulation libraries, and even code specifically tuned to a particular ARM-based processor. It is important that software compiled using specific options such as those just mentioned is also *linked* with libraries compiled in a compatible manner. To state an obvious example, you can't compile little-endian code for a "hello, world" application and link it with a big-endian version of "printf."

To find the multilibs that GCC will build by default for a given target compiler, follow the example below.

Example 5.1: Mulitilibs and GCC Configuration

1. Using any text editor open the file: gcc-4.4.0/gcc/config.gcc
2. Scroll to line #715 and notice the lines:

```
arm*-*-linux-*eabi)
    tm_file="$tm_file arm/bpabi.h arm/linux-eabi.h"
    tmake_file="$tmake_file arm/t-arm-elf arm/t-bpabi arm/t-linux-eabi"
```

   The "tmake_file" line tells the configuration process to include the three listed files in the final, auto-generated Makefile.

3. Open gcc-4.4.0/gcc/config/arm/t-arm-elf and scroll to line #16.

4. Note the MULTILIB_OPTIONS and MULTILIB_DIRNAMES options. These imply that GCC will build two multilibs, one for "arm" and another for "thumb."
5. Open gcc-4.4.0/gcc/config/arm/t-linux-eabi and scroll to line #6
6. Note the newer MULTILIB_OPTIONS and MULTILIB_DIRNAMES. These values will overwrite the previous settings, which tells us that building GCC for an arm-none-linux-gnueabi target is going to generate a default library and another named "vfp."

Example 5.2: Multilib Output from GCC

If you happen to have a version of GCC around, you can pass it the -print-multi-lib option as so:

```
$ arm-none-linux-gnueabi-gcc  -print-multi-lib

.;
vfp;@mfpu=vfp
```

The above output tells us that this particular installation of GCC can compile code against two libraries: the default and one named "vfp" if the -mfpu compiler switch is set to "vfp." Let's look at this a little closer with a simple program.

Create a sample program named "foo.c":

```
      foo.c
      -----
      #include <stdio.h>
      void main (void)
      {
          int i;
          float f = 0.0;

          for ( i = 0; i < 10; ++i )
          {
              printf(" %d : %f\n", i, f);
              f += 0.1;
          }
      }
```

```
$ arm-none-linux-gnueabi-gcc -v -S ./foo.c &>  test-one
$ arm-none-linux-gnueabi-gcc -v -S -mfpu=vfp ./foo.c &> test-two
$ diff  test-one  test-two
```

Note that the line xxxxx/cc1 will contain "-imultilib vfp" in the output of test-two. This means that GCC recognized that the "-mfpu=vfp" switch calls for the generated code to be linked to a different library.

Example 5.3: A Final Example

Here is another example to show how GCC options generate different code and link with different libraries as needed.

```
      my_square.c
      -----------
      float my_square( float x)
      {
          return ( x * x );
      }
```

Using the default compilation options, the tools will call libraries to emulate the floating point operations using integer math.

```
$ arm-none-linux-gnueabi-gcc  -S  my_square.c

my_square.s
-----------
my_square:
 1:   @ args = 0, pretend = 0, frame = 8
 2:   @ frame_needed = 1, uses_anonymous_args = 0
 3:   stmfd sp!, {fp, lr}
 4:   add   fp, sp, #4
 5:   sub   sp, sp, #8
 6:   str   r0, [fp, #-8]     @ float
 7:   ldr   r0, [fp, #-8]     @ float
 8:   ldr   r1, [fp, #-8]     @ float
 9:   bl    __aeabi_fmul
10:   mov   r3, r0
11:   mov   r0, r3
12:   sub   sp, fp, #4
13:   ldmfd sp!, {fp, pc}
14:   .size my_square, .-my_square
15:   .ident      "GCC: (GNU) 4.4.0"
16:   .section    .note.GNU-stack,"",%progbits
```

Note how line #9 contains a call to a function named "__aeabi_fmul()" which will actually perform the floating point multiplication.

Using VFP switches generate actual floating point code.

```
$ arm-none-linux-gnueabi-gcc  -mfpu=vfp  -mfloat-abi=softfp  -S
square.c

my_square.s
-----------
my_square:
 1:   @ args = 0, pretend = 0, frame = 8
 2:   @ frame_needed = 1, uses_anonymous_args = 0
 3:   @ link register save eliminated.
 4:   str   fp, [sp, #-4]!
 5:   add   fp, sp, #0
 6:   sub   sp, sp, #12
 7:   str   r0, [fp, #-8]     @ float
 8:   flds  s14, [fp, #-8]
 9:   flds  s15, [fp, #-8]
10:   fmuls s15, s14, s15
11:   fmrs  r3, s15
12:   mov   r0, r3      @ float
13:   add   sp, fp, #0
14:   ldmfd sp!, {fp}
15:   bx    lr
16:   .size my_square, .-my_square
17:   .ident      "GCC: (GNU) 4.4.0"
18:   .section    .note.GNU-stack,"",%progbits
```

Here, in lines #8–11, we can see that GCC has emitted real floating point instructions which will execute immediately.

Given the discussion above, you will understand why in the following sections we seem to do everything with GLIBC twice.

5.7 Install the Default GLIBC Headers

GCC will need versions of GLIBC headers. You will probably want to review the GLIBC configuration and installation documentation [http://www.gnu.org/software/libc/manual/html_node/Configuring-and-compiling.html#Configuring-and-compiling].

```
$ cd  $BUILDTOOLS/build-glibc
$ echo  "libc_cv_forced_unwind=yes"  >  config.cache
$ echo  "libc_cv_c_cleanup=yes"     >>  config.cache

$ BUILD_CC=gcc \
> CC=$TARGET-gcc \
> CXX=$TARGET-g++ \
> AR=$TARGET-ar \
> RANLIB=$TARGET-ranlib \
> ../glibc-2.9/configure \
> --prefix=/usr \
> --with-headers=$SYSROOT/usr/include \
> --build=i686-pc-linux-gnu \
> --host=$TARGET \
> --without-fp \
> --disable-profile \
> --without-gd \
> --without-cvs \
> --cache-file=config.cache \
> --enable-add-ons

$ make  install-headers \
> install_root=$SYSROOT \
> install-bootstrap-headers=yes

$ mkdir  -p $SYSROOT/usr/include/gnu
$ touch  $SYSROOT/usr/include/gnu/stubs.h
$ cp  bits/stdio_lim.h  $SYSROOT/usr/include/bits/stdio_lim.h

$ mkdir -p $SYSROOT/usr/lib

$ make  csu/subdir_lib
$ cp  csu/crt1.o  csu/crti.o  csu/crtn.o  $SYSROOT/usr/lib
$ $TARGET-gcc  -nostdlib  -nostartfiles  -shared  -x c \
> -o $SYSROOT/usr/lib/libc.so \
> /dev/null
```

5.8 Install the VFP Multilib Headers

Please refer to the discussion in Example 5.3 regarding multilib if you are not sure why we need to regenerate the headers.

```
$ cd  $BUILDTOOLS/build-glibc
$ mkdir  -p ${SYSROOT}/vfp

$ # Clean up previous configuration files
$ rm -rf *

$ echo  "libc_cv_forced_unwind=yes"  >  config.cache
$ echo  "libc_cv_c_cleanup=yes"     >>  config.cache

$ BUILD_CC=gcc \
> CC=$TARGET-gcc \
> CXX=$TARGET-g++ \
> AR=$TARGET-ar \
> RANLIB=$TARGET-ranlib \
> ../glibc-2.9/configure \
> --prefix=/usr \
> --with-headers=$SYSROOT/usr/include \
> --build=i686-pc-linux-gnu \
> --host=$TARGET \
> --disable-profile \
> --without-gd \
> --without-cvs \
> --cache-file=config.cache \
> --enable-add-ons

$ make  install-headers \
> install_root=$SYSROOT/vfp \
> install-bootstrap-headers=yes

$ mkdir  -p $SYSROOT/vfp/usr/include/gnu
$ touch  $SYSROOT/vfp/usr/include/gnu/stubs.h
$ cp  bits/stdio_lim.h  $SYSROOT/vfp/usr/include/bits/stdio_lim.h

$ mkdir -p $SYSROOT/vfp/usr/lib

$ make  csu/subdir_lib
$ cp  csu/crt1.o  csu/crti.o  csu/crtn.o  $SYSROOT/vfp/usr/lib
$ $TARGET-gcc  -nostdlib  -nostartfiles  -shared  -x c \
> -o $SYSROOT/vfp/usr/lib/libc.so \
> /dev/null
```

5.9 Configure and Build GCC-2

Now that we have kernel and glibc headers in place, we can build a more functional GCC.

```
$ cd  $BUILDTOOLS/build-gcc2
$ ../gcc-4.4.0/configure \
> --target=$TARGET \
> --prefix=$PREFIX \
> --with-sysroot=${SYSROOT} \
> --disable-libssp \
> --disable-libgomp \
> --disable-libmudflap \
> --enable-shared \
> --enable-threads \
> --enable-languages=c
```

```
$ make
$ make  install
```

5.10 Build and Install the Default Cross-GLIBC Library

The second build of GCC is complete enough to do a full build of GLIBC.

```
$ cd  $BUILDTOOLS/build-glibc
$ # Clean up previous builds
$ rm -rf *

$ echo  "libc_cv_forced_unwind=yes"  >  config.cache
$ echo  "libc_cv_c_cleanup=yes"     >>  config.cache

$ BUILD_CC=gcc \
> CFLAGS=' -O -mabi=aapcs-linux -march=armv5te -mtune=cortex-a8 ' \
> CFLAGS=' -mfloat-abi=softfp -msoft-float ' $CFLAGS \
> CC=$TARGET-gcc \
> CXX=$TARGET-g++ \
> AR=$TARGET-ar \
> RANLIB=$TARGET-ranlib \
> ../glibc-2.9/configure \
> --prefix=/usr \
> --with-headers=$SYSROOT/usr/include \
> --build=i686-pc-linux-gnu \
> --host=$TARGET \
> --without-fp \
> --with-abi=aapcs-linux \
> --with-arch=armv5te \
> --disable-profile \
> --without-gd \
> --without-cvs \
> --cache-file=config.cache \
> --enable-add-ons

$ make
$ make  install  install_root=$SYSROOT
```

NOTE: We need to fix up some generated linker scripts. The command below can be confusing if you aren't used to scripting with BASH and using SED. Thus, I'll briefly explain. Essentially, this "script" searches for 8 files; lib/libc.so, lib/libpthread.so, lib64/libc.so, lib64/libpthread.so, usr/lib/libc.so, usr/lib/libpthread.so, usr/lib64/libc.so, and usr/lib64/libpthread.so. If it finds the files, it modifies each one using sed. The modifications are to remove the references to the above directories.

It's probably easier to visualize if you write each sed command out individually as the author has done below. Remember that the first character after the substitution (s) command is the delimiter. So the first few commands use a comma as a delimeter rather than the typical "/" character since they are searching for directories:

```
sed s,/usr/lib/,,g  <  ${SYSROOT}/${lib}/${file}_orig  >
${SYSROOT}/${lib}/${file}
```

```
      sed s,/usr/lib64/,,g  <  ${SYSROOT}/${lib}/${file}_orig  >
      ${SYSROOT}/${lib}/${file}

      sed s,/lib/,,g  <  ${SYSROOT}/${lib}/${file}_orig  >
      ${SYSROOT}/${lib}/${file}

      sed s,/lib64/,,g  <  ${SYSROOT}/${lib}/${file}_orig  >
      ${SYSROOT}/${lib}/${file}

      sed /BUG in libc.scripts.output-format.sed/d  <
      ${SYSROOT}/${lib}/${file}_orig  >
      ${SYSROOT}/${lib}/${file}
```

```
$ for  file  in libc.so  libpthread.so
> do
> for  lib in  lib  lib64  usr/lib usr/lib64
> do
> if [ -f ${SYSROOT}/${lib}/${file} ]  && [ ! -h
${SYSROOT}/${lib}/${file} ]
> then
> mv  ${SYSROOT}/${lib}/${file}  ${SYSROOT}/${lib}/${file}_orig
> sed 's,/usr/lib/,,g;s,/usr/lib64/,,g;s,/lib/,,g;s,/lib64/,,g;/BUG in
libc.scripts.output-format.sed/d' < ${SYSROOT}/${lib}/${file}_orig >
${SYSROOT}/${lib}/${file}
> fi
> done
> done
```

5.11 Build and Install the VFP Multilib Cross-GLIBC Library

Again, review the previous discussion on multilib if you are unsure why this step needs to be taken.

NOTE: Notice that we specify -mfpu=vfp this time around.

```
$ cd  $BUILDTOOLS/build-glibc
$ # Clean up previous builds
$ rm -rf *

$ echo  "libc_cv_forced_unwind=yes"  >  config.cache
$ echo  "libc_cv_c_cleanup=yes"     >>  config.cache

$ BUILD_CC=gcc \
> CFLAGS=' -O -mabi=aapcs-linux -march=armv5te -mtune=cortex-a8 ' \
> CFLAGS=' -mfloat-abi=softfp -mfpu=vfp ' $CFLAGS \
> CC=$TARGET-gcc \
> CXX=$TARGET-g++ \
> AR=$TARGET-ar \
> RANLIB=$TARGET-ranlib \
> ../glibc-2.9/configure \
> --prefix=/usr \
> --with-headers=$SYSROOT/usr/include \
> --build=i686-pc-linux-gnu \
> --host=$TARGET \
> --with-abi=aapcs-linux \
> --with-arch=armv5te \
> --disable-profile \
> --without-gd \
> --without-cvs \
> --cache-file=config.cache \
> --enable-add-ons

$ make
$ make  install  install_root=$SYSROOT/vfp
```

NOTE: See the discussion above regarding fixing up the linker script.

```
$ for  file  in libc.so  libpthread.so
> do
> for  lib in  lib  lib64  usr/lib usr/lib64
> do
> if [ -f ${SYSROOT}/vfp/${lib}/${file} ]  && [ ! -h
${SYSROOT}/vfp/${lib}/${file} ]
> then
> mv  ${SYSROOT}/vfp/${lib}/${file}  ${SYSROOT}/vfp/${lib}/${file}_orig
> sed 's,/usr/lib/,,g;s,/usr/lib64/,,g;s,/lib/,,g;s,/lib64/,,g;/BUG in
libc.scripts.output-format.sed/d' < ${SYSROOT}/vfp/${lib}/${file}_orig
> ${SYSROOT}/vfp/${lib}/${file}
> fi
> done
> done
```

5.12 Configure and Build Final GCC

Now that GLIBC has been built, we can build a fully-operational GCC—including support for C++.

```
$ cd  $BUILDTOOLS/build-gcc3
$ ../gcc-4.4.0/configure \
> --target=$TARGET \
> --prefix=$PREFIX \
> --with-sysroot=${SYSROOT} \
> --disable-libssp \
> --enable-shared \
> --enable-threads \
> --enable-languages=c,c++ \
> --enable-__cxa_atexit



$ make
$ make  install
```

5.13 Copy GCC Libraries

We need to copy some GCC libraries into our SYSROOT.

```
$ cd  ${PREFIX}/${TARGET}/lib
$ for  file  in  `ls`
```

```
> do
> if [ ! -d ${file} ]
> then
> cp  -d ${file}  ${SYSROOT}/lib
> fi
> done

$ cd  ${PREFIX}/${TARGET}/lib/vfp
$ for  file  in  `ls`
> do
> if [ ! -d ${file} ]
> then
> cp  -d ${file}  ${SYSROOT}/vfp/lib
> fi
> done
```

6 Extract Logic Patches

We need to extract the Logic patches from the archive we downloaded from http://support.logicpd.com/auth/ so we can apply them to U-Boot and the Linux kernel in upcoming steps.

```
$ cd  $ARCHIVE
$ unzip  1013108_OMAP35x_Patches_v1.5.zip  -d  $PATCHES
$ cd  $PATCHES
$ ls

1013108_OMAP35x_Patches_v1.5.0

$ ls  1013108_OMAP35x_Patches_v1.5.0

linux-2.6.28-rc8  u-boot-1.1.4
```

7 Extract, Patch, and Build U-Boot

7.1 Extract U-Boot

Unpack the U-Boot source code that was previously downloaded from http://www.denx.de/wiki/U-Boot

```
$ cd  $ARCHIVE
$ tar  -xjf u-boot-1.1.4.tar.bz2  -C $UBOOT
$ cd  $UBOOT
$ ls

u-boot-1.1.4

$ cd  u-boot-1.1.4
$ ls

arm_config.mk  drivers         lib_microblaze
board          dtt             lib_mips
CHANGELOG      examples        lib_nios
common         fs              lib_nios2
config.mk      i386_config.mk  lib_ppc
  ...
```

7.2 Patch U-Boot

7.2.1 Apply Logic-Supplied Patches

Apply the Logic-supplied patches, *in the designated order*, to the U-Boot source code. You MUST be in the root directory of the U-Boot source code for the commands below to work. You should be in the same directory as the "COPYING" and "MAINTAINERS" files.

```
$ for p in $PATCHES/1013108_OMAP35x_Patches_v1.5.0/u-boot-1.1.4/*.patch
> do
> echo "Applying patch ${p}"
> patch -p1 < ${p}
> done
```

7.2.2 Manually Modify a U-Boot File

At the time of this writing, you also need to manually make a minor patch to U-Boot.

Patching lib_arm/board.c

The following change is necessary to build U-Boot.

1. Using any text editor, open lib_arm/board.c
2. Scroll to approximately line 82
3. Insert the following function declaration

```
    #if defined(CONFIG_3430LV_SOM)
    void init_vaux1_voltage(void);
    #endif
```

7.2.3 Rename U-Boot Directory (optional)

You may wish to change the name of your U-Boot directory to denote the fact that it has been patched and is no longer a "stock" U-Boot source tree.

```
$ cd ..
$ mv u-boot-1.1.4 u-boot-1.1.4-omap3430-lv-som
$ cd u-boot-1.1.4-omap3430-lv-som
```

7.3 Configure and Build U-Boot

The steps below outline how to configure and build U-Boot for Logic's OMAP35x System on Module (SOM). You are encouraged to read the U-Boot documentation for a complete understanding of the process.

```
$ make distclean
$ make \
> ARCH=arm \
> CROSS_COMPILE=arm-none-linux-gnueabi- \
> omap3530lv_som_config

$ make \
> ARCH=arm \
> CROSS_COMPILE=arm-none-linux-gnueabi- \
> all

$ ls

... u-boot ...

$ file u-boot

u-boot: ELF 32-bit LSB executable, ARM, version 1, statically linked

$ # Copy the u-boot file to /tftpboot for later download
$ cp u-boot /tftpboot
```

7.4 Build and Install the U-Boot mkimage Tool

U-Boot expects the Linux kernel to be packaged in a certain format, commonly named "uImage." The source code for the tool which repackages the kernel is included in the U-Boot package. Here we will build the tool and copy it to the directory where our other compilation tools live. That way, the kernel's build-system can find and use it.

7.5 Build mkimage

Build the tool.

NOTE: It is unclear why the tool doesn't automatically build with the rest of the U-Boot source tree. However, the manual build steps below should work.

```
$ cd $UBOOT
$ cd u-boot-1.1.4-omap3430-lv-som
$ cd tools
$ gcc  -g  -I../include  -Wall  -c crc32.c
$ gcc  -g  -I../include  -Wall  -c mkimage.c
$ gcc  -Wall  -o mkimage  mkimage.o  crc32.o

$ file  mkimage

mkimage: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
```

7.6 Make the Tool Available

We need to copy the mkimage tool to a directory included in our PATH so the kernel build system can find it. We can just copy it into the same directory as the cross tools as we assume that anytime we are building this code, that directory will be available.

NOTE: The commands below assume you installed the CodeSourcery tools into their default location or built your own toolchain from scratch following the instructions in this document.

```
$ cp  mkimage  $HOME/CodeSourcery/Sourcery_G++_Lite/bin

------------- OR -------------

$ cp  mkimage  $TOOLS/bin

$ which  mkimage

--verify that your shell found the proper program--
```

8 Extract, Patch, and Build Linux

8.1 Extract Linux Kernel

Unpack the Linux source code that was previously downloaded from http://www.kernel.org

```
$ cd  $ARCHIVE
$ tar  -xjf linux-2.6.28-rc8.tar.bz2  -C $KERNEL
$ cd  $KERNEL
$ ls

linux-2.6.28-rc8

$ cd  linux-2.6.28-rc8
$ ls

arch     crypto         fs       Kbuild       Makefile
block    Documentation  include  kernel       mm
COPYING  drivers        init     lib          net
CREDITS  firmware       ipc      MAINTAINERS  README
```

8.2 Patch the Linux Kernel

8.2.1 Apply Logic-Supplied Patches

Apply the Logic-supplied patches, *in the designated order*, to the Linux source code. You MUST be in the root directory of the Linux source code for the commands below to work. You should be in the same directory as the "COPYING" and "MAINTAINERS" files.

```
$ for p in $PATCHES/1013108_OMAP35x_Patches_v1.5.0/linux-2.6.28-
rc8/*.patch
> do
> echo "Applying patch ${p}"
> patch -p1 < ${p}
> done
```

8.2.2 Manually Modify the Linux Kernel File

At the time of this writing, you also need to manually make a minor patch to the Linux kernel source code if you are using the version of the CodeSourcery tools denoted here.

Patching arch/arm/Makefile

The following change will allow the kernel to compile with the latest release of CodeSourcery tools.

1. Using any text editor, open arch/arm/Makefile
2. Scroll to approximately line 47
3. Find the lines that read

```
# Note that GCC does not numerically define an architecture version
# macro, but instead defines a whole series of macros which makes
# testing for a specific architecture or later rather impossible.
arch-$(CONFIG_CPU_32v7) :=-D__LINUX_ARM_ARCH__=7 $(call cc-option, \
                          -march=armv7a,-march=armv5t -Wa$(comma) \
                          -march=armv7a)
```

NOTE: The above Makefile instruction should be contained on a single line. This document breaks the line into several lines for readability.

4. Change both instances of armv7a to armv7-a

```
arch-$(CONFIG_CPU_32v7) :=-D__LINUX_ARM_ARCH__=7 $(call cc-option, \
                          -march=armv7-a,-march=armv5t -Wa$(comma) \
                          -march=armv7-a)
```

8.2.3 Rename Linux Kernel Directory (optional)

You may wish to change the name of your Linux kernel directory to denote the fact that it has been patched and is no longer a "stock" Linux source tree.

```
$ cd  ..
$ mv  linux-2.6.28-rc8  linux-2.6.28-rc8-omap3530lv-som
$ cd  linux-2.6.28-rc8-omap3530lv-som
```

8.3 Configure and Build the Linux Kernel

The steps below outline how to configure and build the Linux kernel for Logic's OMAP35x SOM. You are encouraged to read the kernel documentation for a complete understanding of the process.

Logic has provided a default, working kernel configuration with the patch-set. Start by using that and then modify to suit your needs once you have a working system.

NOTE: The uImage target below assumes that you followed the steps in the previous chapter so the kernel build system can find the mkimage program.

```
$ cp  $PATCHES/1013108_OMAP35x_Patches_v1.5.0/linux-2.6.28-rc8/ \
linux-2.6.28-rc8-pm-omap3530lv_som.config .config

$ make  \
> ARCH=arm  \
> CROSS_COMPILE=arm-none-linux-gnueabi- \
> menuconfig
```

You should now be looking at the Linux Kernel Configuration utility. We are going to use an initial RAM disk (initrd) as our device's root file system. This requires that two kernel options be enabled. Use the configuration utility to set the CONFIG_BLK_DEV_RAM and CONFIG_BLK_DEV_INITRD options as described below.

General setup → Initial RAM filesystem and RAM disk support

Device Drivers → Block devices → RAM block device support

```
$ make  \
> ARCH=arm  \
> CROSS_COMPILE=arm-none-linux-gnueabi- \
> uImage

$ ls  arch/arm/boot

... uImage ...

$ file  arch/arm/boot/uImage

arch/arm/boot/uImage: u-boot/PPCBoot image

$ # Copy the uImage file to /tftpboot for later download
$ cp  arch/arm/boot/uImage  /tftpboot
```

9 Extract, Configure, and Build BusyBox

9.1 Extract BusyBox

Unpack the BusyBox source code that was previously downloaded from http://www.busybox.net

```
$ cd  $ARCHIVE
$ tar  --xjf busybox-1.14.1.tar.bz2  -C $ROOTFS
$ cd  $ROOTFS
$ ls

busybox-1.14.1

$ cd  busybox-1.14.1
$ ls

applets         docs        libbb
arch            e2fsprogs   libpwdgrp
archival        editors     LICENSE
AUTHORS         examples    loginutils
Config.in       findutils   mailutils
console-tools   include     Makefile
coreutils       init        Makefile.custom
debianutils     INSTALL     Makefile.flags
```

9.2 Configure and Build BusyBox

The steps below outline how to configure and build BusyBox for Logic's OMAP35x SOM. You are encouraged to read the BusyBox documentation for a complete understanding of the process.

```
$ make  \
> ARCH=arm  \
> CROSS_COMPILE=arm-none-linux-gnueabi-  \
> defconfig

$ make  \
> ARCH=arm  \
> CROSS_COMPILE=arm-none-linux-gnueabi- \
> menuconfig
```

At this point you should be viewing BusyBox's configuration utility. There is one extra configuration option that we want to set for this tutorial. Since we aren't building a system with complete libraries, we need to make sure that BusyBox can stand on its own. We can't have it trying to load dynamic libraries at runtime; therefore, we want to select the option:

Busybox Settings → Build Options → Build BusyBox as a static binary

```
$ make  \
> ARCH=arm  \
> CROSS_COMPILE=arm-none-linux-gnueabi-

$ make  \
> ARCH=arm  \
> CROSS_COMPILE=arm-none-linux-gnueabi-  \
> install

$ ls  _install

bin  linuxrc  sbin  usr

$ ls  _install/bin

addgroup  date      getopt    iptunnel
adduser   dd        grep      kill
  ...

$ ls  -l _install/bin

lrwxrwxrwx 1 logic logic      7 2009-06-24 21:42 addgroup -> busybox
lrwxrwxrwx 1 logic logic      7 2009-06-24 21:42 adduser -> busybox
lrwxrwxrwx 1 logic logic      7 2009-06-24 21:42 ash -> busybox
-rwxr-xr-x 1 logic logic 847388 2009-06-24 21:42 busybox
            ...
```

If the build succeeded, you should have almost an entire root file system inside the _install directory. Notice how all of the programs in _install/bin are actually pointers to the program BusyBox? That is what is meant by "BusyBox is a multi-call binary."

10 Create a Root File System

This section will show you how to create a simple root file system using initrd and BusyBox.

10.1 Create an Empty Root Filesytem

Start the process by creating a file, filling it with zeros, and then formatting the file as an ext2 file system.

```
$ cd  $ROOTFS
$ RDSIZE=4000
$ BLKSIZE=1024
$ dd  if=/dev/zero  of=ramdisk.img  bs=$BLKSIZE  count=$RDSIZE
$ sudo  mke2fs  -F  -m  0  -b  $BLKSIZE  ramdisk.img  $RDSIZE
[sudo] password for logic:
mke2fs 1.41.3 (12-Oct-2008)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
1000 inodes, 4000 blocks
0 blocks (0.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=4194304
1 block group
8192 blocks per group, 8192 fragments per group
1000 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 21 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.

$ file  ramdisk.img

ramdisk.img: Linux rev 1.0 ext2 filesystem data
```

10.2 Mount the Root File System

Assuming the steps above worked, you now have a 4MB file in the $ROOTFS directory which has been formatted with ext2 data. We will use Linux's loopback device to mount that file as if it were any other file system.

```
$ mkdir  mnt
$ sudo  mount  -t ext2  -o loop  ./ramdisk.img  ./mnt
$ ls  mnt

lost+found
```

10.3 Populate the Root File System

Now it is time to populate the root file system with the directories, files, links, and programs we want on the end device.

NOTE: Many of the commands below are prefixed by "sudo" because we need the items created to be "owned" by root.

10.3.1 Create Standard Directories

Start by creating a standard set of subdirectories for the root file system.

```
$ sudo  mkdir  ./mnt/dev
$ sudo  mkdir  -m  777  ./mnt/etc
$ sudo  mkdir  ./mnt/home
$ sudo  mkdir  ./mnt/lib
$ sudo  mkdir  ./mnt/mnt
$ sudo  mkdir  ./mnt/opt
$ sudo  mkdir  ./mnt/proc
$ sudo  mkdir  ./mnt/root
$ sudo  mkdir  ./mnt/sys
$ sudo  mkdir  ./mnt/tmp
$ sudo  mkdir  ./mnt/var
```

10.3.2 Create Standard Devices

Create some standard /dev files.

```
$ cd  ./mnt/dev
$ sudo  mknod  -m  660  console c  5  1
$ sudo  mknod  -m  660  fb0     c  29 0
$ sudo  mknod  -m  660  kmem    c  1  2
$ sudo  mknod  -m  660  mem     c  1  1
$ sudo  mknod  -m  666  null    c  1  3
$ sudo  mknod  -m  660  ram0    b  1  0
$ sudo  mknod  -m  664  random  c  1  8
$ sudo  mknod  -m  600  ttyS0   c  4  64
$ sudo  mknod  -m  664  urandom c  1  9
$ sudo  mknod  -m  666  zero    c  1  5
```

10.3.3 Copy BusyBox Binaries to Root File System

BusyBox built almost everything we need for a small root file system, copy it into our image.

```
$ cd  $ROOTFS/mnt/
$ sudo  cp  -dpRv  $ROOTFS/busybox-1.14.1/_install/*  .
$ cd  $ROOTFS
```

10.3.4 Create a Startup Script

For now, we will create a simple welcome message.

```
$ sudo  mkdir  -m 777  ./mnt/etc/init.d
$ touch  ./mnt/etc/init.d/rcS
$ cat  >>  ./mnt/etc/init.d/rcS  <<  EOF
> #!/bin/ash
> echo "Hello World!"
> mount  -t proc  /proc  /proc
> mount  -t sysfs  none  /sys
> EOF
```

10.3.5 Fix Ownership and Group Settings

Oftentimes, copying the files from BusyBox will keep the owner and group settings that they were built with. In general, we want all of the files that we've copied to our root file system so far to be "owned" by root. Use the chown and chgrp commands to change these settings.

```
$ cd  $ROOTFS
$ sudo  chown  root  -h  --recursive  -L  ./mnt
$ sudo  chgrp  root  -h  --recursive  -L  ./mnt
```

10.4 Finish the Root File System

The last things to do are unmount the newly create file system, compress it, and use U-Boot's mkimage tool to format it.

10.4.1 Unmount the file system.

```
$ cd  $ROOTFS
$ sudo  umount  ./mnt
```

10.4.2 Compress the Root File System

Now that we have un-mounted the file system, we can work directly with the file again.

```
$ gzip  -9  ramdisk.img
```

10.4.3 Repackage the Root File System

As mentioned previously, U-Boot expects the binary objects it loads to be packaged in a specific manner. The root file system is no different than the Linux kernel. We can use the mkimage tool to package it.

```
$ mkimage  \
> -A arm  \
> -O linux  \
> -T ramdisk \
> -C gzip  \
> -n uboot ext2 ramdisk rootfs  \
> -d ramdisk.img.gz  \
> uInitrd

Image Name:
Created:      Wed Jun 24 23:29:26 2009
Image Type:   ARM Linux RAMDisk Image (gzip compressed)
Data Size:    980445 Bytes = 957.47 kB = 0.94 MB
Load Address: 0x00000000
Entry Point:  0x00000000

$ file  uInitrd

uInitrd: u-boot/PPCBoot image

$ cp  uInitrd  /tftpboot
```

11 Download and Execute U-Boot and Linux

If all has gone well so far, you are ready to start running things on real hardware.

11.1 Prepare Development Kit

Prepare to connect to the development hardware.

1. Connect the development kit to the network via Ethernet.
2. Connect the development kit's serial port to your PC using the serial cable provided.
3. Power on the development kit.
4. Launch minicom (or other terminal emulator) on your PC.

When you are properly connected, you should see the LogicLoader splash screen (version numbers may be different than what appears below):

```
NoLo Version : 2.4.6-OMAP3503 0001
NoLo Build   : LPD386 Tue Nov 25 15:00:19 CST 2008
NoLo Compiler: gcc version 4.2.1
Image type   : Elf
Boot Device  : NAND


*****************************************************************

                       LogicLoader

 (c) Copyright 2002-2008, Logic Product Development, Inc.
 All Rights Reserved.
 Version 2.4.6-OMAP3503 0001
*****************************************************************

losh>
```

11.2 Erase Previous U-Boot Environment

When U-Boot is run on the development kit, it saves its environment in a sector of on-board flash. Since we are just getting started with our newly built system, let's erase any possible remnants of previous boots.

```
losh> erase  /dev/nand0  B2047  B1
erasing nand:  100%
erased '/dev/nand0' start=0x7ff: len=0x1 bytes/blocks skipped 0
```

11.3 Download and Launch U-Boot

There are many ways to load the U-Boot file that we've created. We can load it over the serial port, from a CompactFlash or SD/MMC memory card, or from a previously created flash partition (please see the *LogicLoader User Manual* for complete information). For the purposes of this tutorial, we will download all items to the kit using TFTP.

```
losh> ifconfig  sm0  dhcp
losh> load  elf  /tftp/187.199.127.1:u-boot
loading from /tftp/187.199.127.1:u-boot:
.................................................................
ELF section 0: download address: 0x80208000 load address: 0x80e80000
loaded 133528 @ 0x80e80000 Ram
...done
file loaded

losh> exec
U-Boot 1.1.4 (Jun 24 2009 - 23:57:03)

OMAP3430-GP rev 2, CPU-OPP2 L3-133MHz
OMAP3430LV_SOM 0.1 Version + mDDR (Boot NAND)
DRAM:  128 MB
FLASH: initialize in sync mode
NAND:  256 MiB
*** Warning - bad CRC or NAND, using default environment

Read production data: done
Part Number  : 1010194
Model Name   : SOMOMAP3530-10-1672IFCR-A
Serial Number: 3308M00295
In:    serial
Out:   serial
Err:   serial
=====================NOTICE============================
This is the first time that you boot up this board. You are
required to set a valid display for your LCD panel.
Enter the display number of the LCD panel(none for no LCD panel)
Pick one of:
2 == LQ121S1DG31 TFT SVGA (12.1) Sharp
3 == LQ036Q1DA01 TFT QVGA (3.6) Sharp w/ASIC
5 == LQ064D343 TFT VGA (6.4) Sharp
7 == LQ10D368 TFT VGA (10.4) Sharp
15 == LQ043T1DG01 TFT WQVGA (4.3) Sharp
MAKE SURE YOUR DISPLAY IS CORRECTLY ENTERED!
Please enter your LCD display number:
```

When U-Boot launches, enter the proper display number, and then interrupt its "autoboot" mechanism by pressing a key before the 6-second timeout.

11.4 Create a New U-Boot Environment

Since we erased any previously stored U-Boot environment using the LogicLoader's erase command, we need to reestablish a default and sane group of settings. The steps below should be treated as a guideline for your own work by adapting the IP addresses for your own network. Please remember to read the U-Boot documentation and user manual for more information; it should be the definitive reference for your scenario.

```
=> set  ipaddr   187.199.127.107
=> set  serverip 187.199.127.1
=> set  netmask  255.255.255.0
=> save
Saving Environment to NAND...
Erasing Nand...Writing to Nand... done
```

```
=> tftp  ${loadaddr}   uImage
=> tftp  ${rootfsaddr} uInitrd
=> bootm  ${loadaddr}  ${rootfsaddr}
## Booting image at 81000000 ...
   Image Name:   Linux-2.6.28-rc8-omap1
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2100416 Bytes =  2 MB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
OK
## Loading Ramdisk Image at 81300000 ...
   Image Name:
   Image Type:   ARM Linux RAMDisk Image (gzip compressed)
   Data Size:    980445 Bytes = 957.5 kB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK

Starting kernel ...

Uncompressing Linux.......................................
Linux version 2.6.28-rc8-omap1
      ...
```

12 References

12.1 Books

Reinhart, Peter. *The Bread Baker's Apprentice: Mastering the Art of Extraordinary Bread*. Ten Speed Press, 2004. ISBN 1-58008-268-8.

Stallman, Richard M. and GCC Development Community. *Using the GNU Compiler Collection: A GNU Manual For GCC Version 4.4.0*. Free Software Foundation, Inc., 2008.

von Hagen, William. *The Definitive Guide to GCC*. Second Edition. Apress, 2006. ISBN 1-59059-585-8.

Yaghmour, Karim. *Building Embedded Linux Systems*. O'Reilly & Associates, Inc., 2003. ISBN 0-596-00222-X.

12.2 Online Resources

Jones, Tim M. "Linux initial RAM disk (initrd) overview." *IBM developerWorks*, 2006, http://www.ibm.com/developerworks/linux/library/l-initrd.html (accessed July 6, 2009).

Kegel, Dan. "Building and Testing gcc/glibc cross toolchains." *kegel.com*, 2003, http://www.kegel.com/crosstool/ (accessed July 1, 2009).

13 Contact Information

Please visit our website where you can download software, documentation, and participate in on-line discussions: http://www.logicpd.com

For more information regarding Logic's services or products, please send an email to: product.sales@logicpd.com

For technical support regarding any of Logic's products, please review the options on the product support page: http://www.logicpd.com/product-support

Appendix A: Sample Scripts

This appendix includes several sample BASH scripts that you may use to automate many of the steps discussed in this tutorial.

Project Environment Variables

Below is a BASH script you may use to setup the project's environment variables as detailed in this document.

```
#!/bin/bash
#
# !!! NOTE !!! You *must* "source" this script to properly export the
# variables it declares to the shell from which it was invoked.
#
# To do so, run the script using one of the following methods:
#     bash$ .  ./this_script
#          - or -
#     bash$ source ./this_script
#
echo
echo "Setting up some environment variables to use as shortcuts"
echo "throughout the rest of the tutorial."
echo

export PRJROOT=$HOME/olfs
export ARCHIVE=$PRJROOT/archive
export PATCHES=$PRJROOT/patches
export KERNEL=$PRJROOT/kernel
export UBOOT=$PRJROOT/u-boot
export ROOTFS=$PRJROOT/rootfs

# The following environment variables are used if you decide to try
# building the cross-compilation toolchain from scratch.
export TARGET=arm-none-linux-gnueabi
export PREFIX=$PRJROOT/tools
export BUILDTOOLS=$PRJROOT/build-tools
export TARGET_PREFIX=$PREFIX/$TARGET
export SYSROOT=$PREFIX/$TARGET/sysroot

# The next export assumes that you've either installed the CodeSourcery
# tools into their default location or that you followed the
instructions
# to build your own toolchain from scratch.  Uncomment the appropriate
# choice, or adjust accordingly.
# export PATH=$HOME/CodeSourcery/Sourcery_G++_Lite/bin:$PATH
export PATH=$PREFIX/bin:$PATH

echo "Exported the following environment varibles:"
echo "    PRJROOT    => $PRJROOT"
echo "    ARCHIVE    => $ARCHIVE"
echo "    PATCHES    => $PATCHES"
echo "    KERNEL     => $KERNEL"
echo "    UBOOT      => $UBOOT"
echo "    ROOTFS     => $ROOTFS"
echo "    PREFIX     => $PREFIX"
echo "    BUILDTOOLS => $BUILDTOOLS"
echo "    SYSROOT    => $SYSROOT"
echo "    TARGET     => $TARGET"
echo

# Check to see if PATH can actually find our cross-compiler.
COMPILER=`which $TARGET-gcc`
if [ -n "$COMPILER" ]
then
        echo "Using $TARGET cross-platform development tools found
here:"
        echo "    $COMPILER"
else
        echo "Can't find $TARGET tools."
        echo "Either your PATH variable isn't set correctly, or you
have not"
        echo "yet built/installed the cross-platform development
toolchain."
fi
echo
```

Project Directory Structure

Below is a sample BASH script you may use to setup the project's directory structure as detailed in this document.

```
#!/bin/bash

if test "${PRJROOT+set}" != set; then
        echo "!!! Environment variable PRJROOT _not_ set !!!"
        PRJROOT=$HOME/olfs
        echo "Defaulting to $PRJROOT."
        echo "Please double check this if you are using other scripts
to"
        echo "setup the environment for this tutorial."
else
        echo "Creating project directory tree at $PRJROOT"
fi

echo
echo "Creating OLFS directory structure"
echo

if [ ! -d $PRJROOT ]; then
        mkdir $PRJROOT
fi

if [ ! -d $PRJROOT/archive ]; then
        mkdir $PRJROOT/archive
fi

if [ ! -d $PRJROOT/patches ]; then
        mkdir $PRJROOT/patches
fi

if [ ! -d $PRJROOT/kernel ]; then
        mkdir $PRJROOT/kernel
```

```
fi

if [ ! -d $PRJROOT/u-boot ]; then
        mkdir $PRJROOT/u-boot

if [ ! -d $PRJROOT/rootfs ]; then
        mkdir $PRJROOT/rootfs
fi

if [ ! -d $PRJROOT/build-tools ]; then
      mkdir $PRJROOT/build-tools
fi

if [ ! -d $PRJROOT/tools ]; then
      mkdir $PRJROOT/tools
fi

echo "Project created at $PRJROOT"
ls $PRJROOT
```

Download Packages

Below is a sample BASH script you may use to download all of the packages used in this tutorial. Please note that this script will *not* download the patches from Logic. You will need to login to your account at http://support.logicpd.com/auth/ to access them.

```
#!/bin/bash

FAIL=0

if test "${ARCHIVE+set}" != set; then
        echo "!!! Environment variable ARCHIVE _not_ set !!!"
        echo "Please double check to see that you have your"
        echo "environment variables properly exported."
        exit
fi

echo
echo "Downloading source code packages:"
echo "  linux-2.6.28-rc8.tar.bz2"
echo "  u-boot-1.1.4.tar.bz2"
echo "  busybox-1.14.1.tar.bz2"
echo "  CodeSourcery Sourcery G++ Lite 2009q1-203 for ARM GNU/Linux"
echo
echo "This could take some time..."

pushd  $ARCHIVE

# Download Linux kernel version 2.6.28-rc8
wget -c -t 5 -N
http://www.kernel.org/pub/linux/kernel/v2.6/testing/v2.6.28/linux-
2.6.28-rc8.tar.bz2

if (( $? )) ; then FAIL=1 ; fi

# Download kernel signature file (optional)
```

```
wget -c -t 5 -N
http://www.kernel.org/pub/linux/kernel/v2.6/testing/v2.6.28/linux-
2.6.28-rc8.tar.bz2.sign

if (( $? )) ; then FAIL=1 ; fi

# Download u-boot version 1.1.4
wget -c -t 5 -N ftp://ftp.denx.de/pub/u-boot/u-boot-1.1.4.tar.bz2

if (( $? )) ; then FAIL=1 ; fi

# Download busybox version 1.14.1
wget -c -t 5 -N http://www.busybox.net/downloads/busybox-1.14.1.tar.bz2

if (( $? )) ; then FAIL=1 ; fi

# Download the CodeSourcery (www.codesourcery.com) tools.
# Note, we rename the downloaded file using the "-O" switch to wget
because
# the CodeSourcery site will accidentally insert a ? and some other PHP
items
# in the downloaded file name.  They don't hurt anything, but this just
# makes it cleaner.
wget -c -t 5 -O arm-2009q1-203-arm-none-linux-gnueabi.bin
http://www.codesourcery.com/sgpp/lite/arm/portal/package4573/public/arm
-none-linux-gnueabi/arm-2009q1-203-arm-none-linux-gnueabi.bin

if (( $? )) ; then FAIL=1 ; fi

# Download CodeSourcery's outstanding "Getting Started Guide."
wget -c -t 5 -N
http://www.codesourcery.com/sgpp/lite/arm/portal/doc4337/getting-
started.pdf

if (( $? )) ; then FAIL=1 ; fi

if (( $FAIL )); then
        echo "!!! !!! !!!"
        echo "Some downloads may have failed.  Please re-run the
script"
        echo "to try and continue fetching the packages."
        echo "!!! !!! !!!"
fi

echo
echo "Don't forget to login to http://www.logicpd.com and download the"
echo "appropriate zip file containing u-boot and kernel patches."
echo "Check this link:"
echo "
http://support.logicpd.com/auth/downloads/OMAP35x%20Zoom%20Development%
20Kit/#linux"
echo "Look for the \"OMAP35x Linux Demo Image Patch Set\" link."
echo
echo
echo "Downloaded files are in $ARCHIVE"
echo
```

```
popd
```

Build Root File System

Below is a sample BASH script you may use to automate the steps used to create the root file system. This script assumes that you have already configured and built BusyBox.

```
#!/bin/bash

if test "${ROOTFS+set}" != set; then
        echo "!!! Environment variable ROOTFS _not_ set !!!"
        ROOTFS=$HOME/olfs/rootfs
        echo "Defaulting to $ROOTFS."
        echo "Please double check this if you are using other scripts
to"
        echo "setup the environment for this tutorial."
else
        echo "Creating root file system at $ROOTFS"
fi

pushd $ROOTFS

# Clean up previous builds
rm -fv ramdisk.img
rm -fv ramdisk.img.gz
rm -fv uInitrd

# RAMDISK variables
RDSIZE=4000
BLKSIZE=1024

# Create the empty ramdisk image
dd if=/dev/zero of=./ramdisk.img bs=$BLKSIZE count=$RDSIZE
sudo mke2fs -F -m 0 -b $BLKSIZE ./ramdisk.img $RDSIZE

# Mount so we can populate
if [ ! -d ./mnt ]; then
        mkdir ./mnt
fi

sudo mount -t ext2 -o loop ./ramdisk.img ./mnt

# Populate a set of standard sub-directories
sudo mkdir ./mnt/dev
sudo mkdir -m 777 ./mnt/etc
sudo mkdir ./mnt/home
sudo mkdir ./mnt/lib
sudo mkdir ./mnt/mnt
sudo mkdir ./mnt/opt
sudo mkdir ./mnt/proc
sudo mkdir ./mnt/root
sudo mkdir ./mnt/sys
sudo mkdir ./mnt/tmp
sudo mkdir ./mnt/var

# Create standard devices
```

```
cd ./mnt/dev
sudo  mknod  -m  660  console c  5  1
sudo  mknod  -m  660  fb0     c  29 0
sudo  mknod  -m  660  kmem    c  1  2
sudo  mknod  -m  660  mem     c  1  1
sudo  mknod  -m  666  null    c  1  3
sudo  mknod  -m  660  ram0    b  1  0
sudo  mknod  -m  664  random  c  1  8
sudo  mknod  -m  600  ttyS0   c  4  64
sudo  mknod  -m  664  urandom c  1  9
sudo  mknod  -m  666  zero    c  1  5

# Copy BusyBox into the root file system
cd $ROOTFS/mnt/
sudo cp -dpR $ROOTFS/busybox-1.14.1/_install/*  .
cd $ROOTFS

# Create a simple startup message
sudo mkdir -m 777 ./mnt/etc/init.d
touch ./mnt/etc/init.d/rcS

cat >> ./mnt/etc/init.d/rcS  << EOF
#!/bin/ash
echo
echo "Hello, world!"
echo
mount -t proc  /proc /proc
mount -t sysfs none  /sys
EOF

sudo chmod +x ./mnt/etc/init.d/rcS

# Double check ownership and group settings
cd $ROOTFS
sudo chown root -h --recursive -L  ./mnt
sudo chgrp root -h --recursive -L  ./mnt

# Finish up
sudo umount ./mnt
gzip -9 ./ramdisk.img
mkimage -A arm -O linux -T ramdisk -C gzip -d ramdisk.img.gz uInitrd

cp -v ./uInitrd /tftpboot
echo "Root file system should be in /tftpboot/uInitrd"

popd
```