

## Embedded Linux on H3

---

Magnus Aman

WTBU

### ABSTRACT

This document is a shortlist of all steps needed to build and run an embedded version of Linux 2.6 open source on OMAP1710 SDP (H3 board).

### Contents

<b>1</b>	<b>Overview .....</b>	<b>2</b>
1.1	Environment description.....	2
1.2	Local Desktop Utilities.....	2
1.3	Sources description.....	2
<b>2</b>	<b>Building the GNU-ARM compiler .....</b>	<b>3</b>
<b>3</b>	<b>Setting the environment variables .....</b>	<b>4</b>
<b>4</b>	<b>Building the bootloader U-Boot.....</b>	<b>5</b>
<b>5</b>	<b>Building the embedded Linux kernel .....</b>	<b>6</b>
<b>6</b>	<b>Building the Busybox shell toolbox.....</b>	<b>7</b>
<b>7</b>	<b>Creating a file system image with the embedded environment.....</b>	<b>8</b>
7.1	Journalling flash file system, JFFS2.....	11
7.2	Immediate initial ramdisk file system, RAMDISK .....	12
7.3	Indirect initial ramdisk file system, INITRD.....	13
7.4	Network based filesystem, NFS .....	15

## 1 Overview

The embedded Linux development is done on a remote Linux server but also a local desktop PC is used to host the H3 board interfaces.

### 1.1 Environment description

- Remote Desktop PC running RedHat R9 Linux server. Main interface is the network. The services nfs,nfslock, wsftpd, xinetd, telnet must be running.
- Local Desktop PC running WinXP. Main interface is the MMI and but also the network. Board interfaces are USB, COM1 and JTAG.

### 1.2 Local Desktop Utilities

- *TeraTerm* for both network (TCP/IP telnet) terminal to the remote Linux server, and serial (RS232 console) monitor of the H3 board.
- *FileZilla* (freeware) for FTP transfers to and from the remote Linux server.
- *SolarWinds TFTP Server* (freeware) for uploading images to the H3 board from the TFTP root directory on this PC.
- *OST Host* for flashing the bootloader onto the H3 board.
- *VI* simple text editor is called via TeraTerm for editing files on the Linux server.

### 1.3 Sources description

To build the H3 Bootloader and embedded Linux environment we need to get stuff from the Linux community. The filesystem on the Linux server host both the RedHat Intel386 environment as well as the embedded ARM build environment. Further the Linux server implies that the root directory structure is for privileged use (Super-user or Root) and any user logging in will get a user home root directory that we will refer to as **\$HOME**.

In this case **HOME = /home /h3usr**

1. We need to build the GNU-ARM compiler and C-library GLIBC-ARM with the host GNU-I386 compiler.
2. We need to build the bootloader U-Boot for H3 with the GNU-ARM compiler.
3. We need to build the Linux-ARM Kernel sources for H3 with the GNU-ARM compiler.
4. We need to build the embedded shell BusyBox with the GNU-ARM compiler.
5. We need to create and format an initial ramdisk image hosting BusyBox and startup scripts for the Linux-ARM kernel to invoke the system.

## 2 Building the GNU-ARM compiler

A set of scripts, together with GNU-I386 compiler and the network Linux utility *wget*, will automate download and build of the embedded cross-compiler GNU-ARM GCC with its C-library.

Enter directory `$HOME` and create/edit the file `.wgetrc` to locate the proxy to enable the *wget* utility access to the GNU code base:

```
ftp_proxy=www.euproxy.itg.ti.com:80
http_proxy=www.euproxy.itg.ti.com:80
```

Create directory `$HOME/src` where you download the file **`crosstool-0.32.tar.gz`** from <http://kegel.com/crosstool>. Decompress it in this directory and enter `$HOME/src/crosstool-0.32`.

Edit file **`demo-arm.sh`**:

```
TARBALLS_DIR = $HOME/downloads
:
RESULT_TOP = /usr/local
:
uncomment the line calling gcc-3.4.2-glibc-2.2.5.dat
```

Edit file **`gcc-3.4.2-glibc-2.2.5.dat`**:

```
LINUX_HEADERS = 2.6.9
```

Add write permissions to the result directory, then start the build by the commands:

```
su
chmod a+w /usr/local
exit
sh demo-arm.sh
```

After one hour or so you should have the cross-compiler tool chain installed at `/usr/local/arm-unknown-linux-gnu/gcc-3.4.2-glibc-2.2.5/...`

**NOTE:** This absolute location is very important since many build dependencies will need this path later on, also the cross-compiler cannot work if relocated from this point.

### 3 Setting the environment variables

Enter `$HOME` and edit either of the hidden file `.bashrc` or `.bash_profile` to host the needed environment variables needed to build sources:

```
PATH=$PATH:$HOME/bin
PATH=$PATH:/home/common_omap/bin
export TOOLCHAIN=/usr/local/arm-unknown-linux-gnu/gcc-3.4.2-glibc-2.2.5
export TARGET_NAME=arm-unknown-linux-gnu
export TARGET=/usr/local/arm-unknown-linux-gnu/gcc-3.4.2-glibc-2.2.5/arm-unknown-linux-gnu
export KERNEL=/home/h3usr/linux
export PATH=/usr/local/arm-unknown-linux-gnu/gcc-3.4.2-glibc-2.2.5/bin:$PATH
```

**NOTE:** In order for these to take effect you need to logoff and login again.

## 4 Building the bootloader U-Boot

Enter `$HOME/src` where you download the v1.1.1 file `u-boot.tar.gz` from <http://linux.omap.com>  
 Note that the official versions at <http://u-boot.sourceforge.net/> or <ftp://ftp.denx.de/pub/u-boot/> has no support for OMAP1710.

Decompress it and enter `$HOME/src/u-boot`.

If you want to set the default bootloader environment parameters you can do so by editing the file `omap1710h3.h` found in `$HOME/src/u-boot/include/configs`:

```
#define CONFIG_BOOTARGS "whatever you need"
#define CONFIG_IPADDR <ddd.ddd.ddd.ddd>
#define CONFIG_ETHADDR "<xx:xx:xx:xx:xx:xx>"
:
```

In `$HOME/src/u-boot` you need to edit the **Makefile** to find the right compiler:

```
CROSS_COMPILE = arm-unknown-linux-gnu-
```

From the same location you should now clean previous, config and build with these commands:

```
make mrproper
make omap1710h3_config
make
```

The resulting formatting utility file **mkimage** must now be copied from `$HOME/src/u-boot/tools` to `/usr/local/bin` in order to be available later when the kernel is built. This utility converts the compressed kernel image `zImage` into the file `uImage` that is compatible with the U-Boot invocation sequence.

In `$HOME/src/u-boot` you should rename the bootloader binary object file **u-boot.bin** to **u-boot.bin.raw** to get a file that *OST Host* can accept and flash directly into the H3 NOR flash.

Alternatively the ELF-symbolic object file **u-boot** can be processed to become accepted by *Code Composer Studio* and loadable into H3 SDRAM instead.

This is done from `$HOME/src/u-boot` with the command:

```
arm-unknown-linux-gnu-gcc -strip u-boot
```

## 5 Building the embedded Linux kernel

Enter `$HOME/src` where you download **linux-2.6.9.tgz** for OMAP1710 from <http://linux.omap.com>. Decompress it and enter `$HOME/src/linux`.

Edit the **Makefile**:

```
EXTRAVERSION := $(EXTRAVERSION)-H3<your name>
:
CROSS_COMPILE ?= arm-unknown-linux-gnu-
```

Continue the following operations in the same directory, starting by configuring the kernel source for OMAP1710 defaults with the commands:

**make mrproper**

**make omap\_h3\_1710\_defconfig**

Now you have a current hidden **.config** file corresponding to the template located in [\\$HOME/src/linux/arch/arm/configs](#) called *omap\_h3\_1710\_defconfig*.

In order to change drivers (e.g remove USB support) and other stuff you can manually edit items in the configuration file by the command:

**make menuconfig** or

**make xconfig** if you are working directly on the Linux server.

Now you should be able to build a kernel image with the command:

**make**

The resulting compressed kernel image **zImage** will now be located in [\\$HOME/src/linux/arch/arm/boot](#), and it needs to be converted to the file **ulmage** at the same location, for being invocable from the bootloader. The below command will call the U-Boot formatting utility *mkimage* that we previously copied to `/usr/local/bin`:

**make ulmage**

Now copy the **ulmage** to the TFTP root directory for retrieval by the H3 board over the network later.

## 6 Building the Busybox shell toolbox

The embedded Linux system needs shell commands for housekeeping and for this we choose the BusyBox.

Enter `$HOME/src` and download the file **busybox-1.00.tar.gz** from <http://busybox.net>. Decompress it and enter `$HOME/src/busybox-1.00` where you run the command:

**make menuconfig**

```
Select Build Options
*Build BusyBox as a static binary
*Select and Edit the cross-compiler prefix to become:
/usr/local/arm-unknown-linux-gnu/gcc-3.4.2-glibc-2.2.5/bin/arm-unknown-linux-gnu-

Select Installation Options
*Don't use /usr

Leave the rest of the options as default unless you know what to do
```

Now build build BusyBox with the command:

**make install**

Now you will find the resulting binary files and scripts and links located in

```
$HOME/src/busybox-1.00/_install/
$HOME/src/busybox-1.00/_install/bin
$HOME/src/busybox-1.00/_install/sbin
```

ready to be copied to the ramdisk image (created next) root with corresponding locations.

```
<rdroot>/
<rdroot>/bin
<rdroot>/sbin
```

## 7 Creating a file system image with the embedded environment

Create and enter `$HOME/images`.

```
cd $HOME/images
```

Create a zero-filled 4MB empty file with the command:

```
dd if=/dev/zero of=rdroot.img bs=1k count=4096
```

Format it to become a ramdisk image file using the common EXT2 file format:

```
/sbin/mkfs.ext2 -L "/" -F -m0 rdroot.img
```

To add stuff into the file **rdroot.img** it has to be appended into the Linux server filesystem via a mounting point, and since this is a single file it has to be done via the loop device.

Switch to become super-user to have the correct file system permissions and create a mounting point there:

```
su    (now the user's $HOME variable stop working)
```

```
mkdir rdrootx
```

Mount the ramdisk file to be open for writing stuff into:

```
mount -o loop /home/h3usr/images/rdroot.img /home/h3usr/images/rdrootx
```

Enter `/home/h3usr/images/rdrootx` and create the sub-directories:

```
/bin , /sbin, /proc, /sys, /etc/init.d, /usr/local, /var
```

Copy and paste the busybox stuff

```
$HOME/src/busybox-1.00/_install/      to /
$HOME/src/busybox-1.00/_install/bin   to /bin
$HOME/src/busybox-1.00/_install/sbin  to /sbin
```

You need startup scripts too and the below files need to be added as well.

Seen from the root `/home/h3usr/images/rdrootx` you must create the files

`./etc/inittab` with content:

```
# This is run first except when booting in single-user mode
::sysinit:/etc/init.d/rcS

# /bin/sh

# Start an "askfirst" shell on the console (whatever that may be)
::askfirst:-/bin/sh

# Stuff to do when restarting the init process
::restart:/sbin/init

# Stuff to do before rebooting
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
::shutdown:/sbin/swapoff -a
```

`./etc/init.d/rcS` with content:

```
#!/bin/sh
PATH=/bin:/sbin:/usr/bin:/usr/sbin
HOSTNAME=OMAP1710-SDP

hostname $HOSTNAME
echo ""
echo "*****"
echo "Starting System Init for $HOSTNAME"
echo "*****"

# goto the init.d directory
cd /etc/init.d

# Mount the default file systems
mount -a # Mount the default file systems
```

`./etc/fstab` with content:

```
/dev/root    /      auto  defaults,errors=remount-ro 0    0
proc         /proc  proc  defaults                    0    0
```

Finally the ramdisk image should be closed, zipped and formatted for usage with the Linux kernel. Close/unmount the ramdisk file:

**sync**            *(to make sure the latest changes gets written into the loop'ed file)*

**umount /home/h3usr/images/rdrootx**

**exit**            *(switch back from root and become a normal user)*

## 7.1 Journalling flash file system, JFFS2

The benefits of JFFS2 are its non-volatile, crash safe, compressed and randomly distributed flash space usage that avoid wearing out the flash. Any changes to its contents are persistent. A drawback is that it is slow to access.

Mount the ramdisk file **rdroot.img** again to create a JFFS2 file with the utility **mkfs.jffs2**. This utility can be found in the Linux community and can be copied to [/sbin](#) to be generally available:

```
su
mount -o loop /home/h3usr/images/rdroot.img /home/h3usr/images/rdrootx
```

The file system can now be converted into a compressed JFFS2 :

```
mkfs.jffs2 -e 128 -r /home/h3usr/images/rdrootx -o root.jffs2
```

Now copy **root.jffs2** to the TFTP root directory for retrieval by the H3 board bootloader U-Boot. The Linux kernel boot arguments should be changed in U-Boot as follows:

```
# setenv bootargs mem=32M console=ttyS0,115200n8 root=/dev/mtdblock3 rw \
    rootfstype=jffs2 devfs=mount
```

Load and flash the kernel first:

```
# tftp 0x10400000 ulmage (make a note of the retrieved filesize hex value)
#protect off 1:8-23
#erase 1:8-23          One sector is 128kbytes , sector 8-23 (0x100000-0x2FFFFFF)
#cp.b 0x10400000 0x00100000 <filesize1>
```

Then load and flash the filesystem:

```
# tftp 0x10600000 root.jffs2 (make a note of the retrieved filesize hex value)
#protect off 1:24-31
#erase 1:24-31       Sector 24-31 (0x300000-0x3FFFFFF)
#cp.b 0x10600000 0x00300000 <filesize2>
```

Now you can boot your JFFS2 based system from flash:

```
# bootm 0x00100000
```

**NOTE:** The first invocation is delayed a bit since the Linux kernel scans the memory to find the JFFS2 file system location.

## 7.2 Immediate initial ramdisk file system, RAMDISK

It is possible to use the ramdisk file directly to have a volatile but fast file system for debug purposes, the drawback is that you cannot use this approach in a standalone system since it is executed-in-place and disappears after powering off. This is the way to do it:

The uncompressed **rdroot.img** file can be copied to the TFTP root directory for retrieval by the H3 board bootloader U-Boot. The Linux kernel invocation sequence gets the physical ramdisk entry point address in RAM from the boot arguments. The following U-Boot sequence provides this option:

```
# setenv bootargs mem=32M console=ttyS0,115200n8 root=/dev/ram0 rw \  
initrd=0x10800000,4M devfs=mount
```

Download kernel and file system to RAM:

```
# tftp 0x10400000 ulmage  
# tftp 0x10800000 rdroot.img
```

Invoke the Linux kernel:

```
# bootm 0x10400000
```

### 7.3 Indirect initial ramdisk file system, INITRD

For standalone systems one way is to compress the ramdisk, combine it with the Linux kernel in a *BootP* image, and let the kernel decompress it to the final destination in RAM. The benefit is the low latency when accessing the file system, and the drawback is that changes to its contents will not be persistent when powering off since the Initrd in flash is read only once at boot.

Enter `$HOME/src/arch/arm/boot` and copy and compress the ramdisk file to **rdroot.img.gz** :

```
cp -p $HOME/src/images/rdroot.img .
gzip -9 rdroot.img
```

In that location, edit the boot **Makefile** with the added lines:

```
INITRD_PHYS := $(initrd_phys-y)
INITRD = ./arch/arm/boot/rdroot.img.gz
export INITRD
export ZRELADDR INITRD_PHYS PARAMS_PHYS
```

Enter `$HOME/src/linux` and build the combined kernel+fs BootP image:

```
make bootImage
```

Enter `$HOME/src/linux/arch/arm/boot/bootp` where you will find the combined kernel and file system image **bootp**, that you now need to post-process to work with U-Boot:

```
arm-unknown-linux-gnu-objcopy -O binary -R .note -R .comment \
-S bootp linux.bin

gzip -9 linux.bin

mkimage -A arm -O linux -T kernel -C gzip -a 0x10008000 -e 0x10008000 \
-n "Linux INITRD" -d linux.bin.gz ulmage.cc
```

Copy the image **ulmage.cc** to the TFTP root directory for retrieval by the H3 board.

The following U-Boot sequences prepare the kernel bootargs:

```
# setenv bootargs mem=32M console=ttyS0,115200n8 root=/dev/rd/0 rw \  
init=/bin/sh
```

Download the image to RAM:

```
# tftp 0x10400000 ulmage.cc (make a note of the retrieved filesize hex value)
```

Erase and flash it:

```
# protect off 1:8-23
```

```
# erase 1:8-23 One sector is 128kbytes , sector 8-23 (0x100000-0x2FFFFFF)
```

```
# cp.b 0x10400000 0x00100000 <filesize> Flash the compressed 1.5MB Kernel
```

Then invoke the embedded Linux system:

```
# bootm 0x00100000
```

## 7.4 Network based filesystem, NFS

The uncompressed **rdroot.img** file content can also be made directly accessible from the Linux server via its NFS service so that the H3 board Linux kernel invocation sequence finds it and mounts it as a local filesystem. To preserve the file itself we copy the contents to an empty new directory that will be our development target directory:

Become root and create a new directory:

```
su
cd /home/h3usr
mkdir nfs
```

Mount the **rdroot.img** and tar the directory structure into a file in the home dir:

```
mount -o loop /home/h3usr/images/rdroot.img ./nfs
cd ./nfs
tar -cvf ../nfs.tar ./*
cd ../
```

Unmount the image file and copy back contents from the tar file:

```
umount ./nfs
cd ./nfs
tar -xvf ../nfs.tar
exit
```

Setup the Linux NFS Server (System Tools) configuration to export the directory [/home/h3usr/nfs](#) as remote filesystem for the board. I use default options and wildcard for the host description ("\*").

The following U-Boot sequence provides access to the NFS:

```
# setenv bootargs mem=32M console=ttyS0,115200n8 noinitrd \
  root=/dev/nfs rw nfsroot=137.167.42.7:/home/h3usr/nfs,nolock \
  devfs=mount ip=dhcp
# tftp 0x10400000 ulmage
# bootm 0x10400000
```