Creating a Root File System for Linux on OMAP35x

ABSTRACT

A Linux kernel is not very useful without a root file system containing applications and settings. Root file systems can be created in several formats: mountable over a network (NFS), a RAMDISK, or stored in flash (flash file system). Instructions on building and using these file systems is scattered over the Internet and/or in the Documentation directory in a Linux distribution. This note will describe building a simple root file system in the following formats: mountable over NFS, RAMDISK, Journaling Flash File System 2 (JFFS2), and a read-only flash file system known as CRAMFS.

Introduction

The root file system built in this note is based on BusyBox, also known as “The Swiss Army Knife of Embedded Linux”. BusyBox contains reduced size versions of the most commonly used Unix utilities, all in a single executable. It is customizable so that only the utilities that are needed are built. This note guides the reader through the following:

- Downloading and building BusyBox
- Mounting the root file system over NFS
- Building and testing a RAMDISK from the BusyBox target
- Building and testing a JFFS2 file system from the BusyBox target
- Building and testing a CRAMFS file system from the BusyBox target

This note assumes that the reader has root privileges on a PC running Linux. The busybox executable and all symbolic links to it must have user and group ownership set to “root”. This is required because the kernel boots as root. If the generated root file system does not have user/group set to root, the kernel will not boot. There are utility programs on the Internet that can change this, but these are out of the scope of this document.

Nomenclature

For this note, all source and target files are assumed to be in the user’s home directory. For simplicity, this note will assume the user’s directory is “user”. The home directory is then

/home/user

All source code will be in the “src” directory, or

/home/user/src

Code will be built in the “build” directory, or

/home/user/build

The target root file system will be built in the “target” directory, or

/home/user/target

To create these directories, go to /home/user and enter

```
[root@localhost user]# cd /home/user
[root@localhost user]# mkdir src
[root@localhost user]# mkdir build
[root@localhost user]# mkdir target
```

Configure the Linux Kernel to Support File Systems

The Linux kernel must be configured to support the file systems described in this note. To configure the kernel enter "make menuconfig" on the command line. The sub-menus and options to select follow. The Linux kernel source used in this note is from the Texas Instruments V2.6.22.18-OMAP3 release. The sub-menus may be different for other Linux sources and versions.

Device Node Creation

In Linux root file systems the /dev directory contains device nodes – special files that give application access to devices on the system. One example is a serial port. The device node for the first serial port on the system could be /dev/ttyS0. An application could open/read/write/close the serial port. The problem when creating a root file system is which device nodes are needed and which are not?

Early on in Linux root file systems, the creator would statically build the nodes with the "mknod" command (e.g. mknod target/dev/ttyS0 c 4 64). This led to missing nodes, unnecessary nodes, or nodes with the wrong name. During the 2.4.x kernels, the devfs file system was introduced. When device drivers loaded they would register with the devfs file system and the device node would be created. There were problems with this so that starting at 2.6.11 kernels, devfs was removed and replaced.

The current method for dynamic device node creation is to use kernel hotplug with the sysfs and tmpfs file systems. During initialization, when file systems are getting mounted, the tmpfs file system is mounted on /dev. That way all device nodes are in virtual memory and are not saved between reboots of the board.

A driver registers with sysfs and gets added to the system. The kernel then runs the application pointed to by /proc/sys/kernel/hotplug to create the device node and to run any additional scripts (e.g. mounting a device). The application used by the Texas Instruments v2.6.22.x releases is built into busybox and is called "mdev." This will be described later.

The v2.6.22.x releases default to enabling hotplug, sysfs, and tmpfs. To make sure hotplug is set, type in "make menuconfig" and from the "Main menu" use the down arrow key to scroll down to the "General setup" entry and press the Enter key to select it. Scroll down to "Configure standard kernel features (for small systems)" and press the Enter key. Scroll down to "Support for hot-pluggable devices" and make sure it is set. The Space Bar selects and de-selects options. A sample sub-menu is shown below:

```
┌───────── Configure standard kernel features (for small systems) ─────────┐
│  Arrow keys navigate the menu.  <Enter> selects submenus --->.           │
│  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,  │
│  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>  │
│  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >        │
│ ┌──────────────────────────────────────────────────────────────────────┐ │
│ │    --- Configure standard kernel features (for small systems)        │ │
│ │    [*]   Enable 16-bit UID system calls                              │ │
│ │    [ ]   Sysctl syscall support                                      │ │
│ │    [*]   Load all symbols for debugging/ksymoops                     │ │
│ │    [ ]     Include all symbols in kallsyms                           │ │
│ │    [*]     Do an extra kallsyms pass                                 │ │
│ │    [*]   Support for hot-pluggable devices                           │ │
│ │    [*]   Enable support for printk                                   │ │
│ │    [*]   BUG() support                                               │ │
│ │    [*]   Enable ELF core dumps                                       │ │
│ └────v(+)──────────────────────────────────────────────────────────────┘ │
```

```
├──────────────────────────────────────────────────────────────────────────┤
│                    <Select>    < Exit >    < Help >                      │
```

To ensure sysfs and tmpfs are selected, start back at the “Main menu” and scroll down to the “File systems” option. Press Enter to select it and then scroll down to the “Pseudo filesystems” option and press Enter to select it. The file systems selected here should be: /proc, /proc/sys, sysfs, and “Virtual memory file system support.” That last one is also referred to as /tmpfs. A sample sub-menu is shown below.

```
┌────────────────────────── Pseudo filesystems ───────────────────────────┐
│  Arrow keys navigate the menu.  <Enter> selects submenus --->.          │
│  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, │
│  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> │
│  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >       │
│ ┌─────────────────────────────────────────────────────────────────────┐ │
│ │    [*] /proc file system support                                    │ │
│ │    [*]   Sysctl support (/proc/sys)                                 │ │
│ │    [*] sysfs file system support                                    │ │
│ │    [*] Virtual memory file system support (former shm fs)           │ │
│ │    [ ]   Tmpfs POSIX Access Control Lists                           │ │
│ │    < > Userspace-driven configuration filesystem (EXPERIMENTAL)     │ │
│ │                                                                     │ │
│ │                                                                     │ │
│ │                                                                     │ │
│ │                                                                     │ │
│ └─────────────────────────────────────────────────────────────────────┘ │
├─────────────────────────────────────────────────────────────────────────┤
│                    <Select>    < Exit >    < Help >                     │
└─────────────────────────────────────────────────────────────────────────┘
```

Configure the Linux Kernel for Root File System over NFS

In this configuration the root file system resides on a PC running a NFS server. From the “Main Menu” use the down arrow to scroll down to “File systems” and select it by pressing the Enter key. Under “File Systems” scroll down to “Network file systems” and select it. On this page make sure that “NFS file system support”, “Provide NFSv3 client support”, “Provide NFSv4 client support,” and “Root file system on NFS” are selected. A sample sub-menu is shown below.

```
┌───────────────────────── Network File Systems ──────────────────────────┐
│  Arrow keys navigate the menu.  <Enter> selects submenus --->.          │
│  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, │
│  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> │
│  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >       │
│ ┌─────────────────────────────────────────────────────────────────────┐ │
│ │    <*> NFS file system support                                      │ │
│ │    [*]   Provide NFSv3 client support                               │ │
│ │    [ ]     Provide client support for the NFSv3 ACL protocol extensi│ │
│ │    [*]   Provide NFSv4 client support (EXPERIMENTAL)                │ │
│ │    [ ]   Allow direct I/O on NFS files                              │ │
│ │    < > NFS server support                                           │ │
│ │    [*] Root file system on NFS                                      │ │
│ │    [ ] Support for rpcbind versions 3 & 4 (EXPERIMENTAL)         │ │
│ │    --- Secure RPC: Kerberos V mechanism (EXPERIMENTAL)           │ │
│ │    < > Secure RPC: SPKM3 mechanism (EXPERIMENTAL)                │ │
│ └────v(+)──────────────────────────────────────────────────────────┘ │
├──────────────────────────────────────────────────────────────────────┤
│                    <Select>    < Exit >    < Help >                  │
└──────────────────────────────────────────────────────────────────────┘
```

Configure the Linux Kernel for RAMDISK support

In this configuration the u-boot boot loader is used to download a binary image to RAM and pass parameters to the kernel telling it the root file system is on a ramdisk, where it is located, and its size. From the "Main menu" scroll down to "Device Drivers" and select it. Then scroll down to "Block devices" and select it. On this sub-menu make sure "RAM disk support" is selected. For the "Default RAM disk size" enter 16384. The RAM disk built in this note is 16M.

```
┌─────────────────────────────── Block devices ────────────────────────┐
│  Arrow keys navigate the menu.  <Enter> selects submenus --->.       │
│  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, │
│  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> │
│  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >    │
│ ┌──────────────────────────────────────────────────────────────────┐ │
│ │    <*> Loopback device support                                   │ │
│ │    < >   Cryptoloop Support                                      │ │
│ │    < > Network block device support                              │ │
│ │    < > Low Performance USB Block driver                          │ │
│ │    <*> RAM disk support                                          │ │
│ │    (16)  Default number of RAM disks                             │ │
│ │    (16384) Default RAM disk size (kbytes)                        │ │
│ │    (1024) Default RAM disk block size (bytes)                    │ │
│ │    < > Packet writing on CD/DVD media                            │ │
│ │    < > ATA over Ethernet support                                 │ │
│ └──────────────────────────────────────────────────────────────────┘ │
├──────────────────────────────────────────────────────────────────────┤
│                    <Select>    < Exit >    < Help >                  │
└──────────────────────────────────────────────────────────────────────┘
```

The step above configures the kernel for build in a ramdisk. Now go back to the "Main menu" and again select "General setup." Scroll down and make sure "Initial RAM filesystem and RAM disk (initramfs/initrd) support" is selected. By default in the T.I. kernel it is.

```
┌─────────────────────────────── General setup ────────────────────────┐
│  Arrow keys navigate the menu.  <Enter> selects submenus --->.       │
│  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, │
│  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> │
│  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >    │
│ ┌────^(-)──────────────────────────────────────────────────────────┐ │
│ │    [ ] Auditing support                                          │ │
│ │    <*> Kernel .config support                                    │ │
│ │    [*] Enable access to .config through /proc/config.gz          │ │
│ │    (14) Kernel log buffer size (16 => 64KB, 17 => 128KB)          │ │
│ │    [*] Create deprecated sysfs files                              │ │
│ │    [ ] Kernel->user space relay support (formerly relayfs)        │ │
│ │    [*] Initial RAM filesystem and RAM disk (initramfs/initrd) suppor│ │
│ │    ()    Initramfs source file(s)                                 │ │
│ │    [*] Optimize for size (Look out for broken compilers!)         │ │
│ │    [*] Configure standard kernel features (for small systems)  ---> │ │
│ │        Choose SLAB allocator (SLAB)  --->                         │ │
│ └───────────────────────────────────────────────────────────────────┘ │
├───────────────────────────────────────────────────────────────────────┤
│                    <Select>    < Exit >    < Help >                   │
└───────────────────────────────────────────────────────────────────────┘
```

Configure the Linux Kernel for Memory Technology Devices (MTD)

To use any of the flash file systems, such as JFFS2 or CRAMFS, the MTD layer must be configured. MTD are typically flash devices used for storage. Configuring the individual drivers can be tricky. If a device is CFI compliant then all that is needed is to select the CFI options. In this example, CFI is selected, as is Intel/Sharp just in case the part it not CFI compliant. Starting at the “Main menu” scroll down to “Device Drivers” and select it. Scroll down to “Memory Technology Devices (MTD) support” and select it. There are numerous options that can/should be selected. The screen captures below are lengthy.

```
┌─────────────────── Memory Technology Device (MTD) support ────────────────────┐
│  Arrow keys navigate the menu.  <Enter> selects submenus --->.                │
│  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,       │
│  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>       │
│  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >             │
│ ┌───────────────────────────────────────────────────────────────────────────┐ │
│ │    --- Memory Technology Device (MTD) support                             │ │
│ │    [ ]   Debugging                                                        │ │
│ │    <*>   MTD concatenating support                                        │ │
│ │    [*]   MTD partitioning support                                         │ │
│ │    < >     RedBoot partition table parsing                                │ │
│ │    [*]     Command line partition table parsing                           │ │
│ │    < >     ARM Firmware Suite partition parsing                           │ │
│ │    ---   User Modules And Translation Layers                              │ │
│ │    <*>   Direct char device access to MTD devices                         │ │
│ │    ---   Common interface to block layer for MTD 'translation layers│ │
│ │    <*>   Caching block device access to MTD devices                       │ │
│ │    < >   FTL (Flash Translation Layer) support                            │ │
│ │    < >   NFTL (NAND Flash Translation Layer) support                      │ │
│ │    < >   INFTL (Inverse NAND Flash Translation Layer) support             │ │
│ │    < >   Resident Flash Disk (Flash Translation Layer) support            │ │
│ │    < >   NAND SSFDC (SmartMedia) read only translation layer              │ │
│ │          RAM/ROM/Flash chip drivers  --->                                 │ │
│ │          Mapping drivers for chip access  --->                            │ │
│ │          Self-contained MTD device drivers  --->                          │ │
│ │    <*>   NAND Device Support  --->                                        │ │
│ │    <*>   OneNAND Device Support  --->                                     │ │
│ │          UBI - Unsorted block images  --->                         │ │
│ └────────────────────────────────────────────────────────────────────┘ │
├────────────────────────────────────────────────────────────────────────┤
│                    <Select>    < Exit >    < Help >                    │
└────────────────────────────────────────────────────────────────────────┘
```

Select the “RAM/ROM/Flash chip drivers” and ensure CFI and Intel/Sharp are selected.

```
┌──────────────────────── RAM/ROM/Flash chip drivers ────────────────────┐
│  Arrow keys navigate the menu.  <Enter> selects submenus --->.         │
│  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, │
│  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> │
│  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >      │
│ ┌────────────────────────────────────────────────────────────────────┐ │
│ │    <*> Detect flash chips by Common Flash Interface (CFI) probe    │ │
│ │    < > Detect non-CFI AMD/JEDEC-compatible flash chips             │ │
│ │    [ ] Flash chip driver advanced configuration options            │ │
│ │    <*> Support for Intel/Sharp flash chips                         │ │
│ │    < > Support for AMD/Fujitsu flash chips                         │ │
│ │    < > Support for ST (Advanced Architecture) flash chips          │ │
│ │    < > Support for RAM chips in bus mapping                        │ │
│ │    < > Support for ROM chips in bus mapping                        │ │
│ │    < > Support for absent chips in bus mapping                     │ │
│ │                                                                    │ │
│ │                                                                    │ │
│ └────────────────────────────────────────────────────────────────────┘ │
├────────────────────────────────────────────────────────────────────────┤
│                    <Select>    < Exit >    < Help >                    │
└────────────────────────────────────────────────────────────────────────┘
```

Finally, go back to the MTD menu and select the “Mapping drivers for chip access” option. This option enables the partitioning of the MTD into four partitions: u-boot, u-boot environment, kernel, and file system.

```
┌────────────────────── Mapping drivers for chip access ─────────────────┐
│  Arrow keys navigate the menu.  <Enter> selects submenus --->.         │
│  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, │
│  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> │
│  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >      │
│ ┌────────────────────────────────────────────────────────────────────┐ │
│ │    [ ] Support non-linear mappings of flash chips                  │ │
│ │    < > CFI Flash device in physical memory map                     │ │
│ │    < > CFI Flash device mapped on ARM Integrator/P720T             │ │
│ │    <*> TI OMAP board mappings                                      │ │
│ │    < > Map driver for platform device RAM (mtd-ram)                │ │
│ │                                                                    │ │
│ │                                                                    │ │
│ └────────────────────────────────────────────────────────────────────┘ │
├────────────────────────────────────────────────────────────────────────┤
│                    <Select>    < Exit >    < Help >                    │
```

```
└──────────────────────────────────────────────────────────────────────────┘
```

Configure the Linux Kernel for Flash File Systems (JFFS2 and CRAMFS)

JFFS2 is a second generation journaling flash file system with improved wear leveling and compression. CRAMFS is a compressed ROM file system that is read-only, therefore any writes will generate error messages on the console. If any of the logging utilities are started, they should re-direct their output to a file on a file system capable of read-write. Otherwise, do not start any of the logging utilities at boot. To configure the kernel, start at the “Main menu”, scroll down to the “File systems” entry, and select it. Scroll down to “Miscellaneous filesystems” and select it. Scroll down to “Journalling Flash File System v2 (JFFS2) support” and select it. There are other JFFS2 options to select – see the sub-menu below. Continue scrolling down and select “Compressed ROM file system support (cramfs)”.

```
┌────────────────────── Miscellaneous filesystems ─────────────────────────┐
│  Arrow keys navigate the menu.  <Enter> selects submenus --->.           │
│  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,  │
│  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>  │
│  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >        │
│ ┌────^(-)──────────────────────────────────────────────────────────────┐ │
│ │    <*> Journalling Flash File System v2 (JFFS2) support              │ │
│ │    (0)   JFFS2 debugging verbosity (0 = quiet, 2 = noisy)            │ │
│ │    [*]   JFFS2 write-buffering support                               │ │
│ │    [ ]   JFFS2 summary support (EXPERIMENTAL)                        │ │
│ │    [ ]   JFFS2 XATTR support (EXPERIMENTAL)                          │ │
│ │    [*]   Advanced compression options for JFFS2                     │ │
│ │    [*]     JFFS2 ZLIB compression support                            │ │
│ │    [*]     JFFS2 RTIME compression support                           │ │
│ │    [ ]     JFFS2 RUBIN compression support                           │ │
│ │            JFFS2 default compression mode (priority)  --->           │ │
│ │    <*> Compressed ROM file system support (cramfs)                   │ │
│ │    < > FreeVxFS file system support (VERITAS VxFS(TM) compatible)    │ │
│ └────v(+)──────────────────────────────────────────────────────────────┘ │
├──────────────────────────────────────────────────────────────────────────┤
│                    <Select>    < Exit >    < Help >                      │
└──────────────────────────────────────────────────────────────────────────┘
```

Download and Build BusyBox

The source to BusyBox can be downloaded from http://www.busybox.net/downloads. One of the smallest and most stable version at the time of this writing is busybox-1.11.1.tar.bz2. Download it to the /home/user/src directory. Change directory to the build directory and extract the source code with the tar utility. The command line options to tar are: x extract the files j the is a file zipped with bzip2 so use bunzip2 to uncompress v display the files as they are extracted f the following file contains the source

```
[root@localhost src]# cd ../build
[root@localhost build]# tar -xjvf ../src/busybox-1.2.2.1.tar.bz2
```

As the files are extracted, their names are displayed.

Under the /home/user/build directory there should now be a busybox-1.2.2.1 directory. Change directory to it to start selecting build options. To configure build options, type in make menuconfig.

```
[root@localhost build]# cd busybox-1.2.2.1
[root@localhost busybox-1.2.2.1]# make menuconfig
```

The following screen should be displayed:

```
┌──────────────────────── BusyBox Configuration ─────────────────────────┐
│  Arrow keys navigate the menu.  <Enter> selects submenus --->.         │
│  Highlighted letters are hotkeys.  Pressing <Y> selectes a feature,    │
│  while <N> will exclude a feature.  Press <Esc><Esc> to exit, <?> for  │
│  Help, </> for Search.  Legend: [*] feature is selected  [ ] feature is │
│ ┌────────────────────────────────────────────────────────────────────┐ │
│ │               Busybox Settings  --->                               │ │
│ │           --- Applets                                              │ │
│ │               Archival Utilities  --->                             │ │
│ │               Coreutils  --->                                      │ │
│ │               Console Utilities  --->                              │ │
│ │               Debian Utilities  --->                               │ │
│ │               Editors  --->                                        │ │
│ │               Finding Utilities  --->                              │ │
│ │               Init Utilities  --->                                 │ │
│ │               Login/Password Management Utilities  --->            │ │
│ │               Linux Ext2 FS Progs  --->                            │ │
│ │               Linux Module Utilities  --->                         │ │
│ └───────────v(+)─────────────────────────────────────────────────────┘ │
├────────────────────────────────────────────────────────────────────────┤
│                    <Select>    < Exit >    < Help >                    │
└────────────────────────────────────────────────────────────────────────┘
```

To save on space, a hierarchical listing of options that must be selected is shown below.

```
Busybox Settings
  General Configuration
     Show terse applet usage messages
     Store applet usage messages in compressed form
     Use the devpts filesystem for Unix98 PTYs
     (/proc/self/exe) Path to BusyBox executable
  Build Options
     Build BusyBox as a static binary (no shared libs)
     Build with Large File Support
     Do you want to build BusyBox with a Cross Compiler
     (/opt/arm-2007q3/bin/arm-none-linux-gnueabi-) Cross Compiler prefix
  Coreutils
     basename
     cat
     chmod
     cp
     cut
     echo
     false
```

```
   head
   hostid
   ln
   ls
   mkdir
   mkfifo
   mknod
   mv
   pwd
   rm
   sync
   tail
   touch
   true
   uname
   yes
Console Utilities
   clear
   reset
Editors
   awk
   sed
   vi
Finding Utilities
   find
   grep
Init Utilities
   init (keep the defaults that come with this selection)
   Support running commands with a controlling-tty
   poweroff, halt, and reboot
   mesg
Login/Password Management Utilities
   Use internal password and group functuions
   getty
   login
     (de-select Support for /etc/securetty)
Linux Module Utilities
   insmod
   rmmod
   lsmod
   modprobe
   Support version 2.6.x Linux kernels
Linux System Utilities
   dmesg
   mdev
   Support /etc/mdev.conf
   Support command execution at device addition/removal
```

```
     mkswap
     more
     mount
     swaponoff
     umount
     umount —a option
  Networking Utilities
     hostname
     ifconfig
     ping
     telnet
     telnetd
 Process Utilities
     kill
     killall
     pidof
     ps
  Shells
     ash (keep all the defaults)
     Expand prompt string
     ---Bourne Shell Options
        Command line editing
        History saving
        Tab completion
        Fancy shell prompts
  System Logging Utilities
     syslogd
     klogd
```

To build BusyBox, simply type in **make** at the command line. After a few minutes the compile should be complete. When compilation is complete install BusyBox to the target directory. The make and install commands are show below:

```
[root@localhost busybox-1.2.2.1]# make
[root@localhost busybox-1.2.2.1]# make CONFIG_PREFIX=/home/user/target install
```

Configure the New Target Root File System

Looking at the newly built root file system, there are only three subdirectories of binaries and a symbolic link of bin/busybox to linuxrc. More directories must be created before the file system can be used. Some device nodes should be created, too.

```
[root@localhost bin]# cd /home/user/target/
[root@localhost target]# dir
total 28
drwxr-xr-x  2 root root 4096 Nov 21 10:20 bin
lrwxrwxrwx  1 root root   11 Nov 21 10:20 linuxrc -> bin/busybox
drwxr-xr-x  2 root root 4096 Nov 21 10:20 sbin
drwxr-xr-x  4 root root 4096 Nov 21 10:20 usr
```

```
[root@localhost target]#
```

Create the dev, dev/pts, etc, etc/init.d, lib, mnt, opt, proc, root, sys, tmp, var, and var/log directories. Also create the device node for the initial console.

```
[root@localhost target]# mkdir dev
[root@localhost target]# mknod dev/console c 5 1
[root@localhost target]# mkdir dev/pts
[root@localhost target]# mkdir etc
[root@localhost target]# mkdir etc/init.d
[root@localhost target]# mkdir lib
[root@localhost target]# mkdir mnt
[root@localhost target]# mkdir opt
[root@localhost target]# mkdir proc
[root@localhost target]# mkdir root
[root@localhost target]# mkdir sys
[root@localhost target]# mkdir tmp
[root@localhost target]# mkdir var
[root@localhost target]# mkdir var/log
```

Since 2.6.11 the debugfs has been introduced and so you may optionally create a directory called debug for mounting the debugfs.

```
[root@localhost target]# mkdir debug
```

To have the /proc and /dev/pts file systems mounted at boot time, the file etc/fstab must be created in the etc directory. Use an editor to create the file. The example below uses vi to create the file. The two lines to enter into fstab are shown below the vi fstab line.

```
[root@localhost target]# cd etc
[root@localhost etc]# vi fstab
proc            /proc           proc    defaults        0 0
none            /dev/pts        devpts  mode=0622       0 0
```

The login utilities use the files group, hosts, and passwd in the etc directory for logging in. For now, only root needs to be defined in group and passwd while hosts just needs to have localhost defined. The contents of the three files are show below:

```
[root@localhost etc]# vi group
root:x:0:root

[root@localhost etc]# vi passwd
root::0:0:root:/root:/bin/ash

[root@localhost etc]# vi hosts
127.0.0.1       localhost
```

The kernel starts "/sbin/init" after it boots (actually the kernel attempts to execute several known programs until one succeeds). Init reads the etc/inittab file to determine what to do at start up, shutdown, or when a user logs in. These inittab files can get quite complicated. A simple one is shown below:

```
[root@localhost etc]# vi inittab
::sysinit:/etc/init.d/rcS

# /bin/ash
#
# Start an "askfirst" shell on the serial port
console::askfirst:-/bin/ash

# 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
```

The “sysinit” line tells init to run the /etc/init.d/rcS script to set up the system. The rest are self explanatory.

A simple etc/init.d/rcS file could assume file systems existed in the kernel and would simply mount the mount points as needed. A more complicated one would test for the existence of file systems and if found, mount them; if not found, find ways to still get the system booted.

The author has taken the etc/init.d/rcS and mdev.conf files from the V2.6.22.18-OMAP3 release for a simple example. The rcS script will test for the existence of file systems and mount them accordingly. The 1.11.1 version of BusyBox does not create a link for /bin/sh so change any /bin/sh to /bin/ash. The contents are shown below:

```
[root@localhost etc]# vi init.d/rcS

#!/bin/sh
#   ---------------------------------------------
#   Common settings
#   ---------------------------------------------
HOSTNAME=OMAP3EVM
VERSION=1.0.0

hostname $HOSTNAME

#   ---------------------------------------------
#   Prints execution status.
#
#   arg1 : Execution status
#   arg2 : Continue (0) or Abort (1) on error
#   ---------------------------------------------
status ()
{
       if [ $1 -eq 0 ] ; then
               echo "[SUCCESS]"
       else
               echo "[FAILED]"
```

```
                if [ $2 -eq 1 ] ; then
                        echo "... System init aborted."
                        exit 1
                fi
        fi

}

#   ---------------------------------------------
#   Get verbose
#   ---------------------------------------------
echo ""
echo "    System initialization..."
echo ""
echo "    Hostname       : $HOSTNAME"
echo "    Filesystem     : v$VERSION"
echo ""
echo ""
echo "    Kernel release : `uname -s` `uname -r`"
echo "    Kernel version : `uname -v`"
echo ""

#   ---------------------------------------------
#   MDEV Support
#   (Requires sysfs support in the kernel)
#   ---------------------------------------------
echo -n " Mounting /proc              : "
mount -n -t proc /proc /proc
status $? 1

echo -n " Mounting /sys               : "
mount -n -t sysfs sysfs /sys
status $? 1

echo -n " Mounting /dev               : "
mount -n -t tmpfs mdev /dev
status $? 1

echo -n " Mounting /dev/pts           : "
mkdir /dev/pts
mount -t devpts devpts /dev/pts
status $? 1

echo -n " Enabling hot-plug           : "
echo "/sbin/mdev" > /proc/sys/kernel/hotplug
status $? 0
```

```
echo -n " Populating /dev            : "
mkdir /dev/input
mkdir /dev/snd

mdev -s
status $? 0

#   ---------------------------------------------
#   Disable power management
#   (Requires sysfs support in the kernel)
#   ---------------------------------------------
# echo -n " Disabling Power mgmt       : "
# echo -n "1" > /sys/power/cpuidle_deepest_state
# status $? 1

#   ---------------------------------------------
#   Turn off LCD after 1 hour of inactivity
#   (Requires sysfs support in the kernel)
#   ---------------------------------------------
# echo -n " Turn off LCD after 1 hour  : "
# echo -n "3600" > /sys/power/fb_timeout_value
# status $? 1

#   ---------------------------------------------
#   Mount the default file systems
#   ---------------------------------------------
echo -n " Mounting other filesystems : "
mount -a
status $? 0

#   ---------------------------------------------
#   Set PATH
#   ---------------------------------------------
export PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin

#   ---------------------------------------------
#   Start other daemons
#   ---------------------------------------------
echo -n " Starting syslogd           : "
/sbin/syslogd
status $? 0

echo -n " Starting telnetd           : "
/usr/sbin/telnetd
status $? 0

#   ---------------------------------------------
#   Done!
#   ---------------------------------------------
echo ""
echo "System initialization complete."

#   ---------------------------------------------
#   Start demo app
#   ---------------------------------------------
#if [[ -x /etc/init.d/demo_start ]]; then
#       echo " Starting Demo Application..."
#       /etc/init.d/demo_start &
#       sleep 5
#fi
```

Please note that in more recent kernels (2.6.29 and newer) the sysfs entries shown above to disable power management and control the LCD inactivity timeout shown above have been removed. For latest details on using OMAP Power Management in Linux please refer to this article [1].

To automatically mount the debugfs on start-up the following can be added to the above rcS file. Please note that the debugfs needs to be selected in the kernel configuration and built into the kernel before you can mount it. The below script checks to see if the debugfs is supported by the kernel before attempting to mount the file-system. To enable the debugfs run "make menuconfig" and go to "Kernel Hacking" and select "Debug Filesystem".

```
cat /proc/filesystems | grep -q debugfs
if [ $? -eq 0 ] ; then
        echo -n " Mounting /debug             : "
        mount -n -t debugfs none /debug
        status $? 1
fi
```

After saving the file, change its access permissions so that it is executable for all.

```
[root@localhost etc]# vi mdev.conf

audio       0:5 0666
console     0:5 0600
control.*   0:0 0660 @/bin/mv /dev/$MDEV /dev/snd/
dsp         0:5 0666
event.*     0:0 0600 @/bin/mv /dev/$MDEV /dev/input/
fb          0:5 0666
nfs         0:5 0770
null        0:0 0777
pcm.*       0:0 0660 @/bin/mv /dev/$MDEV /dev/snd/
rtc         0:0 0666
tty         0:5 0660
tty0*       0:5 0660
tty1*       0:5 0660
tty2*       0:5 0660
tty3*       0:5 0660
tty4*       0:5 0660
```

```
tty5*        0:5 0660
tty6*        0:5 0660
ttyS*        0:5 0640
urandom      0:0 0444
zero         0:0 0666
```

Add the Shared Libraries Applications will Require

Various applications will be built using the libraries in the tool chain so the runtime libraries must be copied to the root file system. Care should be taken to only copy the libraries that are required so that the root file system does not grow based on unused libraries. This note will simply demonstrate how to copy all the shared libraries to the root file system and strip out any debug information from the libraries.

In the Code Sourcery tool chain, most of the libraries are found in the /opt/arm-2007q1/arm-none-linux-gnueabi/libc/lib directory. Simply copy these files to the root file system, maintaining symbolic links and then strip out unneeded debug information.

```
[root@localhost user]# cd /home/user/target/lib
[root@localhost lib]# cp –r /opt/arm-2007q1/arm-none-linux-gnueabi/libc/lib/* .
[root@localhost lib]# arm-none-linux-gnueabi-strip *
```

Some libraries might be in other directories in the tool chain; those would have to be found on a “trial and error” basis when applications do not load because of a missing shared library.

A minimal, working root file system has now been built. It will boot, but does not have kernel modules yet. These will be added later after testing.

Mounting the Root File System over NFS

In this scenario, the target device and the host system are connected via Ethernet, either directly through a crossover cable or to a hub or switch. The host must export the target file system through NFS and the NFS server daemon must be running. For configuring and running NFS on a Linux PC, see the HOWTOs at any Linux Documentation Internet site. In this simple example, the exported file system is /home/user/target and anybody can mount it via NFS. The /etc/exports file is shown below

```
/home/user/target        *(rw,no_root_squash)
```

On the target side, the u-boot environment variable bootargs must be set to boot from NFS and contain the mount address of the root file system. In the example below, the NFS server is at IP address 192.168.1.104.

```
[root@OMAP3EVM /] # setenv bootargs mem=64M ip=dhcp noinitrd
console=ttyS0,115200n8 root=/dev/nfs rw
nfsroot=192.168.1.104:/home/user/target,nolock,rsize=1024,wsize=1024
[root@OMAP3EVM /] # saveenv
```

After booting the kernel, the final boot messages should look similar to below.

```
Sending DHCP requests .<6>eth0: link up, 100Mbps, full-duplex, lpa 0xCDE1
., OK
IP-Config: Got DHCP answer from 255.255.255.255, my address is 128.247.107.23
IP-Config: Complete:
     device=eth0, addr=128.247.107.23, mask=255.255.254.0, gw=128.247.106.2,
    host=128.247.107.23, domain=am.dhcp.ti.com, nis-domain=(none),
    bootserver=255.255.255.255, rootserver=128.247.107.35, rootpath=
Looking up port of RPC 100003/2 on 128.247.107.35
Looking up port of RPC 100005/1 on 128.247.107.35
VFS: Mounted root (nfs filesystem).
Freeing init memory: 136K
init started: BusyBox v1.11.1 (2008-10-05 04:40:51 CDT)
starting pid 288, tty : '/etc/init.d/rcS'

   System initialization...

   Hostname       : OMAP3EVM
   Filesystem     : v1.0.0

   Kernel release : Linux 2.6.22.18-omap3
   Kernel version : #12 Mon Oct 6 01:22:49 CDT 2008

Mounting /proc             : [SUCCESS]
Mounting /sys              : [SUCCESS]
Mounting /dev              : [SUCCESS]
Mounting /dev/pts          : [SUCCESS]
Enabling hot-plug          : [SUCCESS]
Populating /dev            : [SUCCESS]
Disabling Power mgmt       : [SUCCESS]
Turn off LCD after 1 hour  : [SUCCESS]
Mounting other filesystems : [SUCCESS]
Starting syslogd           : [SUCCESS]
Starting telnetd           : [SUCCESS]

System initialization complete.

Please press Enter to activate this console.
```

Creating and Booting a RAMDISK

A ramdisk is simply an ext2 file system. The tools to build one are shipped with just about every Linux distribution. The steps to build the ramdisk from the target file system are as follows:

1. Change directory to the /home/user directory.
2. Use the dd utility to create a 16M file of “zero” that will contain the file system.
3. Use mke2fs to create the empty file system in the file from 2.
4. Mount the file to a mount point using the loop device.
5. Use tar to create an archive from the target and pipe it into tar to extract the target to the mount point.
6. Un-mount the file.
7. Copy it to the /tftpboot directory so that u-boot can download it to the target device.

```
[root@localhost user]# cd /home/user
[root@localhost user]# dd if=/dev/zero of=rd-ext2.bin bs=1k count=16384
[root@localhost user]# mke2fs -F -m0 rd-ext2.bin
[root@localhost user]# mount -t ext2 rd-ext2.bin /mnt -o loop
[root@localhost user]# tar -C target -cf - . | tar -C /mnt -xf -
```

```
[root@localhost user]# umount /mnt
[root@localhost user]# cp rd-ext2.bin /tftpboot
```

To boot the kernel with the root file system on the ramdisk, the u-boot environment variable “bootargs” must be modified. The “bootargs” variable contains information that gets passed to the kernel at boot up time and has information such as memory size, ip address, where the root file system is, what kind of file system it is, etc. For this note, a TI OMAP 3530 EVM is used so all RAM and flash addresses are specific to that platform.

Start up a terminal program on the host and turn on the target. It is assumed that u-boot has already been flashed to the target board. The ramdisk will be downloaded to RAM at address 0x81600000 and is 16M in size. At the u-boot prompt, change the bootargs variable as follows:

```
[root@OMAP3EVM /] # setenv bootargs mem=88M ip=dhcp console=ttyS0,115200n8
                    root=/dev/ram0 rw initrd=0x81600000,16M ramdisk_size=16384
[root@OMAP3EVM /] # saveenv
```

All the text should be entered as one line. Now download the kernel to 0x80000000 and the ramdisk to 0x81600000 and then boot the kernel.

```
[root@OMAP3EVM /] # tftp 81600000 rd-ext2.bin
[root@OMAP3EVM /] # tftp 80000000 uImage
[root@OMAP3EVM /] # bootm
```

After the kernel boots, the terminal should show the usual boot up messages and then finally indicates that the filesystem is mounted as a RAMDISK.

```
RAMDISK: ext2 filesystem found at block 0
RAMDISK: Loading 16384KiB [1 disk] into ram disk... done.
VFS: Mounted root (ext2 filesystem).
Freeing init memory: 136K
init started: BusyBox v1.11.1 (2008-10-05 04:40:51 CDT)
starting pid 287, tty : '/etc/init.d/rcS'

   System initialization...

   Hostname       : OMAP3EVM
   Filesystem     : v1.0.0

   Kernel release : Linux 2.6.22.18-omap3
   Kernel version : #12 Mon Oct 6 01:22:49 CDT 2008

Mounting /proc             : [SUCCESS]
Mounting /sys              : [SUCCESS]
Mounting /dev              : [SUCCESS]
Mounting /dev/pts          : [SUCCESS]
Enabling hot-plug          : [SUCCESS]
Populating /dev            : [SUCCESS]
Disabling Power mgmt       : [SUCCESS]
Turn off LCD after 1 hour  : [SUCCESS]
Mounting other filesystems : [SUCCESS]
Starting syslogd           : [SUCCESS]
Starting telnetd              : [SUCCESS]

System initialization complete.

Please press Enter to activate this console.
```

MTD Partitions

Before discussing flash file systems, “MTD partitions” must be explained. A flash device must be partitioned on an embedded device just as a hard disk must be partitioned on a Linux PC. On a Linux PC there would be a partition for /boot, a swap partition, a /usr partition, etc. For the embedded device the partitions are usually just for the boot loader, its environment variables, the kernel, and a flash file system.

There is no “fdisk” for flash devices like there is for a hard disk. Instead, the partitioning is done in software. The exact file that does the partitioning is different based on the chip, board, CPU, and Linux source code. For this note, the board used is the TI OMAP 3530 EVM and the kernel source is from the SDK1.0.0 release. The file containing the MTD driver and partitions is linux/kernel_org/2.6_kernel/arch/arm/mach-omap2/board-omap3evm-flash.c.

Looking at the file, the structure omap_partitions[ ] contains the partition table for the H4 board.

```
static struct mtd_partition omap3evm_onenand_partitions[] = {
       {
               .name           = "X-Loader-ONENAND",
               .offset         = 0,
               .size           = 4 * (64 * 2048),
               .mask_flags     = MTD_WRITEABLE /* force read-only */
       },
       {
               .name           = "U-Boot-ONENAND",
               .offset         = MTDPART_OFS_APPEND,
               .size           =  14*(64*2048),
               .mask_flags     = MTD_WRITEABLE /* force read-only */
       },
       {
               .name           = "Boot Env-ONENAND",
               .offset         = MTDPART_OFS_APPEND,
               .size           = 2*(64*2048),
       },
       {
               .name           = "Kernel-ONENAND",
               .offset         = MTDPART_OFS_APPEND,
               .size           = 40*(64*2048),
       },
       {
               .name           = "File System - ONENAND",
               .offset         = MTDPART_OFS_APPEND,
               .size           = MTDPART_SIZ_FULL,
       },
```

}; The boot loader has a partition size of 256k. The u-boot environment variables fit in a 128k partition for ES1.0 or 256k for ES2.0. The kernel partion is 2M while what is left is used for a file system.

When the kernel boots and the driver loads, device nodes are created based on the partitions. The device nodes take on the form /dev/mtdblock/x or /dev/mtdblockx.

```
                         Flash
Physical Adress Range  Bank  Device Node     Partition Name
0x00080000-0x00020000  1     /dev/mtdblock0  "X-Loader-NAND"
0x001c0000-0x00020000  1     /dev/mtdblock1  "U-Boot-NAND"
0x00040000-0x00020000  1     /dev/mtdblock2  "Boot Env-NAND"
0x00500000-0x00020000  1     /dev/mtdblock3  "Kernel-NAND"
0x0f880000-0x00020000  1     /dev/mtdblock4  "File System - NAND"
```

The physical address is set by u-boot. Look at the file u-boot/include/asm/arch/cpu.h to see the flash physical address mappings:

```
/* GPMC Mapping */
# define FLASH_BASE            0x10000000  /* NOR flash (aligned to 256 Meg) */
# define FLASH_BASE_SDPV1      0x04000000  /* NOR flash (aligned to 64 Meg) */
# define FLASH_BASE_SDPV2      0x10000000  /* NOR flash (aligned to 256 Meg) */
# define DEBUG_BASE            0x08000000  /* debug board */
# define NAND_BASE             0x10000000  /* NAND addr (actual size small port)*/
# define PISMO2_BASE           0x18000000  /* PISMO2 CS1/2 */
# define ONENAND_MAP           0x20000000  /* OneNand addr (actual size small port */
```

Creating and Booting a JFFS2 Root File System

Since jffs2 is used only on flash devices, a standard Linux distribution does not have the tools to make a jffs2 file system. The Internet site http://sources.redhat.com/jffs2 explains where and how to obtain the latest source to MTD code and file systems. mkfs.jffs2 is the tool needed to build a jffs2 file system and the source is in this code. All of the MTD code will be downloaded but only the code to build mkfs.jffs2 is built.

The source code is maintained in a CVS tree. CVS is version control software and is usually installed when the Linux distribution is installed. CVS does not work across firewalls so a direct connection to the Internet is required. From the command line enter the following:

```
[root@localhost user]# cd /home/user/build
[root@localhost build]# cvs –d :pserver:anoncvs@cvs.infradead.org:/home/cvs login CVS password: anoncvs
[root@localhost build]# cvs –d :pserver:anoncvs@cvs.infradead.org:/home/cvs co mtd
```

There should now be a mtd directory under /home/user/build. The source to mkfs.jffs2 is found in the util directory under mtd. To build mkfs.jffs2 simply change directory to mtd/util and enter “make mkfs.jffs2”. When the build is complete, copy the mkfs.jffs2 file to the /sbin directory; that is where the other utilities that make file systems reside.

```
[root@localhost build]# cd mtd/util
[root@localhost util]# make mkfs.jffs2
[root@localhost util]# cp mkfs.jffs2 /sbin
```

With the mkfs.jffs2 utility built and in place it is now time to make the jffs2 file system from of the target directory. Change to the /home/user directory and enter the mkfs.jffs2 command below. Probably the most important argument to the utility is the erase block size. For the NAND part on the OMAP3530 EVM board the erase block is 128k Consult either u-boot, the kernel, or the data manual for the flash part used to find the erase block size.

```
[root@localhost util]# cd /home/user
[root@localhost util]# mkfs.jffs2 -lqnp –e 128 -r target –o /tftpboot/rd-jffs2.bin
```

By building the file in the /tftpboot directory, the step of copying it over is eliminated. The file must now be written into flash at a specific location. In u-boot, the flash file system will get flashed to the physical address 0x780000 and will be mounted by the kernel from /dev/mtdblock4. The steps to perform this are:

1. Unprotect the flash area where the file system will reside.
2. Erase that area.
3. Download the rd-jffs2.bin file.
4. Write it to flash.
5. Modify bootargs to support a JFFS2 file system as root on /dev/mtdblock4.
6. Save the environment variables.

NOTE: (Not up to date - based on a different kernel – see SDK1.0.0 reprogramming scripts: C:\OMAP35x_SDK_1.0.0\board_utilities\scripts\ reflash-samsung.txt)

```
[root@OMAP3EVM /] # onenand unlock 0x0 0x8000000
[root@OMAP3EVM /] # mw.b 0x81600000 0xff 0x1400000
[root@OMAP3EVM /] # onenand erase block 60-1023
[root@OMAP3EVM /] # tftpboot 0x81600000 rd-jffs2.bin
[root@OMAP3EVM /] # onenand write 0x81600000 0x780000 0x1400000
[root@OMAP3EVM /] # setenv bootargs mem=88M noinitrd ip=dhcp console=ttyS0,115200n8  root=/dev/mtdblock4 rootfstype=jffs2
[root@OMAP3EVM /] # saveenv
```

Boot the kernel and near the end of boot-up the following messages should be seen:

```
IP-Config: Gateway not on directly connected network.
VFS: Mounted root (jffs2 filesystem).
Freeing init memory: 136K
eth0: link up, 100Mbps, full-duplex, lpa 0xCDE1
init started: BusyBox v1.11.1 (2008-10-05 04:40:51 CDT)
starting pid 288, tty : '/etc/init.d/rcS'

   System initialization...

   Hostname       : OMAP3EVM
   Filesystem     : v1.0.0

   Kernel release : Linux 2.6.22.18-omap3
   Kernel version : #1 Tue Jul 1 17:16:34 IST 2008

Mounting /proc             : [SUCCESS]
Mounting /sys              : [SUCCESS]
Mounting /dev              : [SUCCESS]
Mounting /dev/pts          : [SUCCESS]
Enabling hot-plug          : [SUCCESS]
Populating /dev            : [SUCCESS]
Disabling Power mgmt       : [SUCCESS]
Turn off LCD after 1 hour  : [SUCCESS]
Mounting other filesystems : [SUCCESS]
```

```
Starting syslogd            : [SUCCESS]
Starting telnetd            : [SUCCESS]

System initialization complete.

Please press Enter to activate this console.
```

Creating and Booting a CRAMFS Root File System

Since CRAMFS is another embedded file system, the host Linux distribution might not have the mkcramfs utility. The source to the cramfs tools can be found at http://sourceforge.net/projects/cramfs/.As usual, download, build, and install them.

The first step is to edit the etc/init.d/rcS file so that the logging daemons do not get started. Since CRAMFS is a read-only file system, the serial console would be filled with errors as these loggers try to write messages to disk. After that, change to the /home/user directory and use the mkcramfs utility to create the file image. The –v argument only says to be verbose and display messages as the file system is built.

```
[root@localhost home]# cd /home/user
[root@localhost user]# mkcramfs -v target /tftpboot/rd-cramfs.bin
```

By building the file in the /tftpboot directory, the step of copying it over is eliminated.

The file must now be written into flash at a specific location.

In u-boot, the flash file system will get flashed to the physical address 0x780000 and will be mounted by the kernel from /dev/mtdblock4. The steps to perform this are:

1. Unprotect the flash area where the file system will reside.
2. Erase that area.
3. Download the rd-cramfs.bin file.
4. Write it to flash.
5. Modify bootargs to support a CRAMFS file system as root on /dev/mtdblock4.
6. Save the environment variables.

```
[root@OMAP3EVM /] # onenand unlock 0x0 0x8000000

[root@OMAP3EVM /] # mw.b 0x81600000 0xff 0x1400000

[root@OMAP3EVM /] # onenand erase block 60-1023

[root@OMAP3EVM /] # tftpboot 0x81600000 rd-cramfs.bin

[root@OMAP3EVM /] # onenand write 0x81600000 0x780000 0x1400000

[root@OMAP3EVM /] # setenv bootargs mem=88M noinitrd ip=dhcp console=ttyS0,115200n8 root=/dev/mtdblock4 rootfstype=cramfs

[root@OMAP3EVM /] # saveenv
```

Boot the kernel and near the end of boot-up the following messages should be seen: NOTE: (Not up to date - based on a different kernel)

```
VFS: Mounted root (cramfs filesystem) readonly.
Freeing init memory: 136K
Mounting proc : OK
Mounting sysfs : OK
Mounting /dev : OK
Creating local mdev devices

   System initialization...
   Hostname       : OMAP3EVM
   Filesystem     : v1.0.0

   Kernel release : Linux 2.6.22.18-omap3
   Kernel version : #1 Tue Jul 1 17:16:34 IST 2008

Mounting devpts : OK
Setting up networking
Configuring lo : OK
Mounting filesystems : OK

System initialization complete.

Please press Enter to activate this console.
```

Enhancing the Root File System

The root file system built contains only the bare minimum files to operate. Kernel driver modules have not been added yet. Also, as applications are built and added, the shared libraries used by the applications will need to be added. The file system will get larger when these files are added.

Add the Kernel Driver Modules to the Root File System

Any drivers built as modules need to be added to the target root file system. This is fairly simple; after building the kernel, build the modules, and then use the same kernel Makefile to install the files to the target.

```
[root@localhost 2.6_kernel]# make modules
[root@localhost 2.6_kernel]# make INSTALL_MOD_PATH=/home/user/target modules_install
```

After the second make is performed, the kernel modules will be copied to the /home/user/target/lib/modules directory. One problem still exists – the modules still have debug information so they are larger than they need to be. When debug information is no longer needed, use the arm-none-linux-gnueabi-strip utility to remove it. Unfortunately the strip utility does not have a recursive option to go into each directory and strip the driver. If there are only a few drivers, the find utility can be used to find each .o file recursively and then pass the file name to strip. An example is shown below.

```
[root@localhost user]# cd /home/user/target/lib/modules
[root@localhost modules]# arm-none-linux-gnueabi-strip `find . –name “*.ko”`
```

Note that the ` character is the “tick” key that is on the left-hand side of the keyboard and is a lowercase ~.

RAMDISK Considerations

As files get added to the target root file system, the image size will grow beyond 16M. The kernel must be configured to the new RAMDISK size and re-built. See section “3.3 Configure the Linux Kernel for RAMDISK support” for more information on how to change the RAMDISK size in the kernel. The amount of RAM used for the RAMDISK should not be too big so that the kernel has enough RAM to load and run programs. For example, on a board with only 32M RAM, a 20M RAMDISK would only leave 12M for the kernel to use. This could cause a shortage of memory for programs to run. If a development platform has more then 32M, then that should also be reflected in the u-boot environment variable “bootargs”. For example, assume a development platform now has 64M of RAM. The bootargs variable could be changed from “mem=32M” to “mem=64M”. A RAMDISK of 20M would no longer be a problem. One last hint on keeping the RAMDISK image small. Use the arm-none-linux-gnueabi-strip program to strip off unneeded headers and debug information once an application or driver has been fully debugged and ready for inclusion in the image.

Trouble Shooting

If you are having problems booting the kernel please refer to this article for help.

References

[1] http://elinux.org/OMAP_Power_Management

Article Sources and Contributors

Creating a Root File System for Linux on OMAP35x *Source*: http://processors.wiki.ti.com/index.php?oldid=114243 *Contributors*: Anveshg, Archaeologist, Jefflance01, Jonhunter, Kevinsc

License