coreIPM-LINUX for OMAP Release Notes

1. INTRODUCTION

coreIPM-LINUX is a fully fledged Linux distribution for OMAP 35xx architecture with built in support for coreIPM management architecture.

The distribution has support for the following features:

SNMP

We use SNMP to notify management applications of system events. The coreSNMP module uses the “snmptrap” command to send SNMP traps. The Platform Event Trap format is used for sending a platform event in an SNMP Trap. The specification of the Platform Event Trap format is defined in “IPMI - Platform Event Trap Format Specification v1.0 Document Revision 1.0 December 7, 1998”
The coreIPM SNMP interface is built on the Net-SNMP package. There are two MIB files related to coreIPM in /usr/local/share/smnp/mibs directory: COREIPM-GROUP-SMI.txt defines our group enterprise number and our product identification OIDs. COREIPM-MIB.txt holds the coreIPM MIB.

These files should also be on the system that will be running the management applications or interpreting traps and informs.
There are also 2 C files coresnmp.h and coresnmp.c which are used for SNMP agent extension. This enables the SNMP agent to support our MIB. When queried for the objects we have defined in the MIB files, the agent extensions respond with the current values.

The agent snmpd is located in /opt/snmp/sbin directory and started on system startup.

Command Line Interface (CLI)

The command line interface gives you an easy way to configure and manage your shelf. CLI also allows you to script commands.

CLI command summary:

- fru [ipmc [fru]]
- deactivate ipmc fru
- activate ipmc fru
- frudata ipmc fru offset data1 [data2...data22]
- frudata [ipmc [fru]]
- help
- sendmod fruid netfn cmd data1 [data2...data17]
- sendcmd addr netfn cmd data1 [data2...data25]
- upgrade

- date [YYYY MM DD hh mm [ss]| hh mm [ss]]
- version
- ipmc [ipmc]
- sensordata [ipmc [[lun:]number]]
- sensor [ipmc [[lun:]number]]
- cooling policy [on|off]
- shelf fs
- shelf fans_state
- shelf cs
- shelf cooling_state
- fans [fru]
- alarm [minor|major|critical|clear]
- sel clear [ipmc]
- sel info [ipmc]
- sel [-v] [ipmc [record_count [starting_record]]]

RMCP

Enables network interface for sending and receiving IPMI commands. We support:

Multi-session access
User Privilege Levels
Cipher Suites

+ Authentication Algorithms
  - HMAC-SHA1
  - HMAC-MD5
+ Integrity Algorithms
  - HMAC-SHA1-96
  - HMAC-MD5-128
  - MD5-128
+ Confidentiality Algorithms
  - AES-CBC-128

OpenHPI server & plug-in

HPI provides a standard and hardware independent service to upper level management software to set and retrieve configuration or operational data about the hardware components, and to control the operation of those components.

HPI is defined as a library API of C-library functions . OpenHPI provides an open source implementation of the Service Availability Forum (SAF) Hardware Platform Interface (HPI). Open HPI includes a Plug-in Application Binary Interface (ABI): an internal interface designed for developers to easily write modules for a specific platform with ease.

We have our own plug-in implementation called coreHPI that provides the interface to the coreIPM architecture.

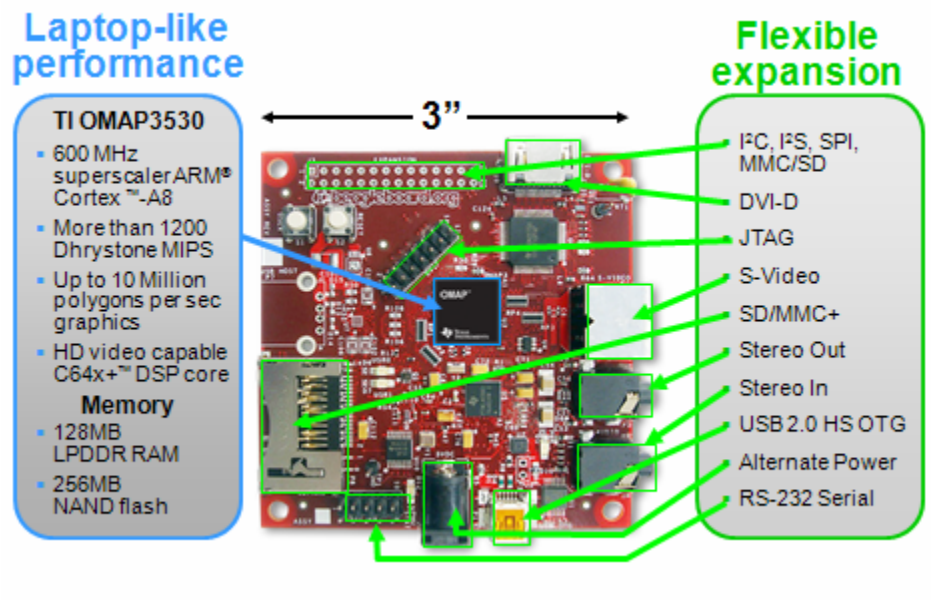
2. HARDWARE

OMAP3500 processors deliver laptop-like performance at handheld power levels with over 1,200 Dhrystone MIPS using the superscalar ARM Cortex-A8 with highly accurate branch prediction and 256KB L2 cache running at up to 600MHz.

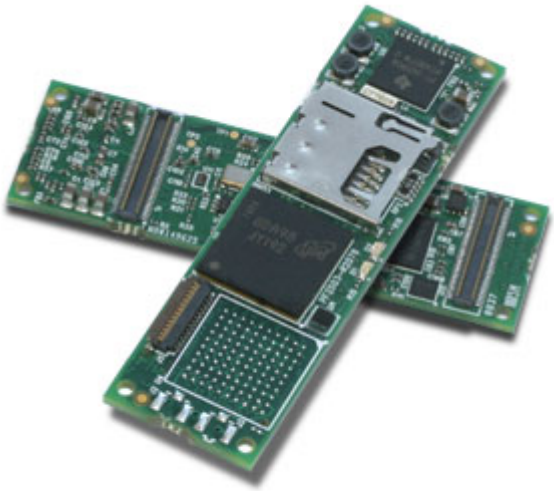
To connect to a 10/100 Ethernet network we support the USB200M from Linksys which attaches directly to a USB port.

We currently support the following development boards for our OMAP release, both are currently sold at around $150:

Beagle Board
Beagle Board is an ultra-low cost, high performance, low power OMAP3530 based platform designed by BeagleBoard.org community members and sold by Digi-Key. (http://beagleboard.org/hardware)



gumstix Overo™
OMAP 3503 Application Processor with ARM Cortex-A8 CPU at 600 MHz with 256MB RAM & 256MB Flash.
(http://www.gumstix.com/store/catalog/product_info.php?products_id=211)



3. SOFTWARE COMPILATION OPTIONS AND PROCEDURES

Boot Stages

Stage 1: Processor ROM Code

During power-on, or after a RESET operation, the OMAP processor runs its internal ROM code. This ROM code cannot be modified by the system designer. After a power-on-reset is initiated, the ROM code reads the SYS.BOOT register to determine the memory interface configuration and programs the general-purpose memory controller (GPMC) accordingly. Then the ROM code determines how the flash device is configured and whether this device is supported by the ROM code.

After the flash device configuration has been verified, the process of copying the x-loader from the flash device to the internal SRAM of the OMAP processor begins.

First, the ROM code reads bytes 1 through 4 of the x-loader to determine the size of the file; then it reads bytes 5 through 8 of the x-loader, which contain the destination address in SRAM where the x-loader will be shadowed. The ROM code then shadows the x-loader from the flash device to the OMAP processor SRAM, and finally, the system jumps to the SRAM address where the first byte of the x-loader is stored.

Stage 2: Bootstrap

x-loader is the stage 2 bootstrap code. The x-loader code is stored in the flash, and the ROM code copies it to the OMAP processor SRAM for execution. x-loader in turn bootstraps U-Boot.

Stage 3: Boot Loader
Stage 3 is the boot loader, which is used to copy the operating system code from the flash to the DRAM; in this case U-Boot is the boot loader code. The U-Boot code is stored in flash, and the stage 2 code copies it to the DRAM for execution.

Stage 4: Operating System
The Linux kernel, is stored in the flash, and the stage 3 code copies it to the DRAM, where it is executed. The boot process is complete after this stage as the OS takes control of the system.

3.1 Compiling x-loader

Compiling x-loader for NAND booting

- In file include/configs/omap3530beagle.h disable the "CFG_CMD_MMC" macro

```
/* For X-loader to be flashed on to NAND disable the below
macro */
//#define CFG_CMD_MMC              1
```

- Compile the x-loader

```
# make CROSS_COMPILE=arm-none-linux-gnueabi- distclean
# make CROSS_COMPILE=arm-none-linux-gnueabi-
omap3530beagle_config
# make CROSS_COMPILE=arm-none-linux-gnueabi-
```

File named "x-load.bin" will be generated

- Convert x-load.bin to x-load.bin.ift (required to FLASH x-loader to NAND) using the "SignGP" tool. signGP reads the x-load.bin file and writes out the x-load.bin.ift file. The signed image is the original pre-pended with the size of the image and the load address. If not entered on command line, file name is assumed to be x-load.bin in current directory and load address is 0x40200800.

```
# ./signGP x-load.bin
```

- Copy x-load.bin.ift to NAND or download it through UART.

Compiling x-loader for MMC booting

- In file include/configs/omap3530beagle.h enable the "CFG_CMD_MMC" macro

```
/* For X-loader to be flashed on to NAND disable the below
macro */
#define CFG_CMD_MMC              1
```

- Compile the x-loader

```
# make CROSS_COMPILE=arm-none-linux-gnueabi- distclean
# make CROSS_COMPILE=arm-none-linux-gnueabi-
omap3530beagle_config
# make CROSS_COMPILE=arm-none-linux-gnueabi-
```

File named "x-load.bin" will be generated

- Convert x-load.bin to x-load.bin.ift (required to FLASH x-loader to NAND) using the "SignGP" tool. signGP reads the x-load.bin file and writes out the x-load.bin.ift file. The signed image is the original pre-pended with the size of the image and the load address. If not entered on command line, file name is assumed to be x-load.bin in current directory and load address is 0x40200800.

```
# ./signGP x-load.bin
```

- Rename x-load.bin.ift to MLO (required for MMC booting)

- Copy MLO to MMC/SD card using a card reader/writer.

3.2 Compiling u-boot

Compiling u-boot for Flashing NAND automatically

- In file include/configs/omap3530beagle.h enable the CONFIG_BOOTCOMMAND macro and comment the CONFIG_BOOTCOMMAND below it

```
#define CONFIG_BOOTCOMMAND       \
   "mmcinit;fatload mmc 0 0x80200000 x-load.bin.ift;\
    nand unlock;nand ecc hw;nand erase 0 80000;nand write.i
0x80200000 0 80000;\
    fatload mmc 0 0x80200000 flash-uboot.bin; nand unlock;\
    nand ecc sw;nand erase 80000 160000; nand write.i
0x80200000 80000 160000;\0"
```

Comment the below line as shown below

```
/* #define CONFIG_BOOTCOMMAND "\0" */
```

- Build u-boot

```
# make CROSS_COMPILE=arm-none-linux-gnueabi- distclean
```

```
# make CROSS_COMPILE=arm-none-linux-gnueabi-
omap3530beagle_config
# make CROSS_COMPILE=arm-none-linux-gnueabi-
```

File named "u-boot.bin" will be generated

Compiling u-boot for regular Kernel Booting

- In file include/configs/omap3530beagle.h disable the CONFIG_BOOTCOMMAND macro and uncomment the CONFIG_BOOTCOMMAND macro below it

```
/*
#define CONFIG_BOOTCOMMAND       \
   "mmcinit;fatload mmc 0 0x80200000 x-load.bin.ift;\
   nand unlock;nand ecc hw;nand erase 0 80000;nand write.i
0x80200000 0 80000;\
   fatload mmc 0 0x80200000 flash-uboot.bin; nand unlock;\
   nand ecc sw;nand erase 80000 160000; nand write.i
0x80200000 80000 160000;\0"
*/
```

Uncomment CONFIG_BOOTCOMMAND macro below

```
#define CONFIG_BOOTCOMMAND "\0"
```

- Compile u-boot

```
# make CROSS_COMPILE=arm-none-linux-gnueabi- distclean
# make CROSS_COMPILE=arm-none-linux-gnueabi-
omap3530beagle_config
# make CROSS_COMPILE=arm-none-linux-gnueabi-
```

File "u-boot.bin" will be generated.

3.3 Compiling Kernel

- Compile the Kernel

```
# make CROSS_COMPILE=arm-none-linux-gnueabi- distclean
# make CROSS_COMPILE=arm-none-linux-gnueabi-
omap3_beagle_defconfig
# make CROSS_COMPILE=arm-none-linux-gnueabi- uImage
```

File named "uImage" will be generated in arch/arm/boot directory

3.4 Compiling MTDUtils

Source and dependencies

MTD utils are available from MTD utils git. You can get them by

- using gitweb "snapshot" feature (use "snapshot" link at latest commit at the right side)
- using git

```
git pull git://git.infradead.org/mtd-utils.git mtd-utils
```

Compiling MTD utils depend on zlib and LZO. Download latest archives using the given links. For this example we use

- zlib-1.2.3.tar.bz2
- lzo-2.03.tar.gz
- mtd-utils.git-snapshot-20081004.tar.gz

Cross compiling

In this example, we use

```
/home/user/mtd
```

as base directory. This example assumes you are in this directory and the above three source .tar.gz files are located here, too.

To not pollute the host file system, we install build results in local subdirectory:

```
> mkdir install
```

should result in /home/user/mtd/install (replace this with your real path below)

zlib

```
> tar xfj zlib-1.2.3.tar.bz2
> cd zlib-1.2.3/
zlib-1.2.3 > ./configure --prefix=/home/user/mtd/install
```

Edit resulting Makefile, e.g.

```
zlib-1.2.3 > emacs Makefile
```

and add toolchain prefix *arm-none-linux-gnueabi-* to gcc, ar and ranlib. Then you should be ready to compile:

```
zlib-1.2.3 > make
```

```
zlib-1.2.3 > make install
zlib-1.2.3 > cd ..
```

Result should be zlib.a in install/lib directory and zlib's headers in install/include. If this was successful, remove build directory:

```
> rm -rf zlib-1.2.3
```

lzo

```
> tar xfz lzo-2.03.tar.gz
> cd lzo-2.03/
lzo-2.03 > ./configure --host=arm-none-linux-gnueabi --
prefix=/home/user/mtd/install
lzo-2.03 > make
lzo-2.03 > make install
lzo-2.03 > cd ..
> rm -rf lzo-2.03
```

Result should be liblzo2.a in install/lib directory and lzo's headers in install/include/lzo.

mtd-utils

```
> tar xfz mtd-utils.git-snapshot-20081004.tar.gz
> cd mtd-uitls/
```

MTD-Utils don't have a configure script, so we have to edit Makefile again. Depending on the version of MTD Utils, make sure head of top level Makefile has:

```
mtd-uitls > emacs Makefile
PREFIX=/home/user/mtd/install
...
ZLIBCPPFLAGS=-I$(PREFIX)/include
LZOCPPFLAGS=-I$(PREFIX)/include/lzo

ZLIBLDFLAGS=-L$(PREFIX)/lib
LZOLDFLAGS=-L$(PREFIX)/lib

CROSS=arm-none-linux-gnueabi-
...
CFLAGS ?= -O2 -g $(ZLIBCPPFLAGS) $(LZOCPPFLAGS)
...
```

Then, you should be able to cross compile MTD Utils setting variable WITHOUT_XATTR:

```
mtd-uitls > WITHOUT_XATTR=1 make
mtd-uitls > make install
mtd-uitls > cd ..
> rm -rf mtd-utils
> cd install/sbin/
install/sbin > arm-none-linux-gnueabi-strip *
```

Directory install/sbin/ should now contain cross compiled MTD utils you can use at your target.

3.4 Source & Pre-built images

Source Archive	md5sum
x-loader	
u-boot 1.3.3	

Pre-Built Image	Description	md5sum
x-load.bin.ift_for_NAND	x-loader image used to flash on NAND for NAND booting	
MLO	x-loader image to be copied to MMC/SD cards for booting Beagle wit MMC/SD	6ae111d0b3bad7673697187a8e3ee4b6
u-boot.bin	u-boot Image	33ee8852dfdb091b74c7ddc2d9ad2445
u-boot.bin	u-boot Image that automatically flashes u-boot Image to NAND from MMC	
Kernel (uImage) 2.6.22.18	Linux Kernel Image with USB OTG mode enabled	
BusyBox (ramdisk) File System	8MB Ramdisk File System Image	
BusyBox FS with ALSA libraries	File system Image with ALSA libraries	

Note: To use MLO, U-Boot and kernel images from above, rename downloaded files to **MLO**, **u-boot.bin** and **uImage** at your SD card. I.e. remove the extensions to distinguish the download files.

Tools for OMAP3

Tool Path	Description
ARM Linux GCC	codesourcery tool chain (Select GNU/Linux and download version 2007q3)
signGP	x-loader Signing Tool Source is here
HP MMC/SD Disk Format Tool	HP USB Disk Storage Format Tool 2.0.6 for Windows. See LinuxBootDiskFormat for using Linux fdisk instead

3.5 Booting your development system

Currently, booting with MMC/SD is the only working way for first board bring up.

MMC/SD formatting

As described in above MMC/SD boot description, you have to create a bootable partition on MMC/SD Card. See Appendix C - MMC Boot Format on how this can be done with Linux tools.

Dual partition card

You can create a dual-partition card, booting from a FAT partition that can be read by the OMAP3 ROM bootloader and Windows, then utilizing an ext2 partition for the Linux root file system.

To mount second ext2 partition as root file system (e.g. containing contents of rd-ext2.bin) use kernel boot arguments (e.g. in uboot using setenv bootargs):

```
console=ttyS2,115200n8 root=/dev/mmcblk0p2 rw rootwait
```

U-Boot booting

If your MMC/SD card formatting is correct and you put MLO, u-boot.bin and uImage on the card you should get a u-boot prompt after booting beagle board. E.g. (output from terminal program with 115200 8N1):

```
...40T.........XH.H.U ..Instruments X-Loader 1.41
Starting on with MMC
Reading boot sector

717948 Bytes Read from MMC
Starting OS Bootloader from MMC...

U-Boot 1.1.4 (Apr  2 2008 - 13:42:13)

OMAP3430-GP rev 2, CPU-OPP2 L3-133MHz
TI 3430Beagle 2.0 Version + mDDR (Boot ONND)
```

```
DRAM:  128 MB
Flash:  0 kB
NAND:256 MiB
In:    serial
Out:   serial
Err:   serial
Audio Tone on Speakers  ... complete
#
```

Using this u-boot prompt, you now can start kernel uImage stored on MMC card manually:

```
# mmcinit
# fatload mmc 0:1 0x80000000 uimage
# bootm
```

If you like to make that happen every boot:

```
# set bootcmd 'mmcinit ; fatload mmc 0:1 0x80000000 uimage ;
bootm' ; saveenv
```

The following software parts can be stored and booted/run from NAND:

- X-Loader
- U-Boot (+ environment/configuration data)
- Linux kernel
- Linux file system

The memory partitioning:

```
0x00000000-0x00080000 : "X-Loader"
0x00080000-0x00260000 : "U-Boot"
0x00260000-0x00280000 : "U-Boot Env"
0x00280000-0x00680000 : "Kernel"
0x00680000-0x10000000 : "File System"
```

To be able to write something to (empty) NAND, you first need to boot from an other source, e.g. MMC/SD card boot. Besides the files you need for MMC/SD card boot (MLO & U-Boot), put the files you want to flash into first FAT partition of MMC/SD card, too. Then you can read them from there and write them to NAND. Note: If something goes wrong writing the initial X-Loader, your board might not boot any more without pressing the user button. See Appendix D- Board recovery on how to fix this.

X-Loader

Build or download binary (x-load.bin.ift_for_NAND) X-Loader. Put it at first (boot) FAT partition of MMC/SD card and boot from card. Then start boot from card and use the following to write X-Loader to NAND:

```
...40T.......

Texas Instruments X-Loader 1.41
Starting on with MMC
Reading boot sector

147424 Bytes Read from MMC
Starting OS Bootloader from MMC...

U-Boot 1.3.3-00411-g76fe13c-dirty (Jul 12 2008 - 17:12:05)

OMAP3530-GP rev 2, CPU-OPP2 L3-165MHz
OMAP3 Beagle Board + LPDDR/NAND
DRAM:  128 MB
NAND:  256 MiB
In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0

# mmcinit
# fatload mmc 0:1 80000000 x-load.bin.ift_for_NAND
reading x-load.bin.ift_for_NAND

9808 bytes read
# nand unlock
device 0 whole chip
nand_unlock: start: 00000000, length: 268435456!
NAND flash successfully unlocked
# nand ecc hw
# nand erase 0 80000

NAND erase: device 0 offset 0x0, size 0x80000
Erasing at 0x60000 -- 100% complete.
OK
# nand write 80000000 0 80000

NAND write: device 0 offset 0x0, size 0x80000
524288 bytes written: OK
#
```

Note: The command *nand ecc hw* is essential here! X-Loader is started by OMAP3 boot rom. This uses HW ECC while reading the NAND, so while writing,

we have to use OMAP3 HW ECC, too. If you don't use HW ECC boot ROM, you can't boot from NAND any more. See Appendix D - Board recovery.

U-Boot

Build or download binary (flash-uboot.bin) U-Boot. Put it at first (boot) FAT partition of MMC/SD card and boot from card. Then start boot from card and use the following to write U-Boot to NAND:

```
# mmcinit
# fatload mmc 0:1 80000000 u-boot.bin
reading u-boot.bin

147424 bytes read
# nand unlock
device 0 whole chip
nand_unlock: start: 00000000, length: 268435456!
NAND flash successfully unlocked
# nand ecc sw
# nand erase 80000 160000

NAND erase: device 0 offset 0x80000, size 0x160000
Erasing at 0x1c0000 -- 100% complete.
OK
# nand write 80000000 80000 160000

NAND write: device 0 offset 0x80000, size 0x160000
 1441792 bytes written: OK
#
```

Note: You can use the same u-boot.bin you use to boot from MMC/SD card for NAND. There are no differences between U-Boot used for MMC/SD card boot and NAND boot.

Note: Here, you don't need the *nand ecc hw* option. X-Loader which loads & starts U-Boot is able to understand SW ECC written by U-Boot.

Kernel

While X-Loader and U-Boot can be written only by U-Boot, for kernel and file system there are two ways to write them to NAND: Either by U-Boot (similar way as for X-Loader and U-Boot above) or from running kernel (e.g. booted from MMC card).

Note: X-Loader and U-Boot can't be written from already running kernel, too, because from kernel point of view X-loader and U-Boot NAND partitions are

marked as write only. See *omap3beagle_nand_partitions[]* configuration structure in kernel's arch/arm/mach-omap2 directory.

Writing kernel with U-Boot

```
# mmcinit
# fatload mmc 0:1 80000000 uImage
reading uImage

# nand ecc sw
# nand erase 280000 400000

NAND erase: device 0 offset 0x280000, size 0x400000
Erasing at 0x660000 -- 100% complete.
OK
# nand write 80000000 280000 400000

NAND write: device 0 offset 0x280000, size 0x400000
 4194304 bytes written: OK
#
```

Once you do this, use U-Boot commands to boot kernel (uImage) from NAND:

```
# nand read 80000000 280000 400000 ; bootm 80000000
```

These, you can store as bootcmd and your board will automagically boot uImage from NAND.

Writing kernel with kernel

Once you have a kernel booted, e.g. from MMC card, you can use it to write himself (uImage) to NAND and then switch from MMC boot to kernel NAND boot. For this, observe kernel's boot messages. These should have something like ...

```
omap2-nand driver initializing
NAND device: Manufacturer ID: 0x2c, Chip ID: 0xba (Micron
NAND 256MiB 1,8V 16-bit)
cmdlinepart partition parsing not available
Creating 5 MTD partitions on "omap2-nand":
0x00000000-0x00080000 : "X-Loader"
0x00080000-0x00260000 : "U-Boot"
0x00260000-0x00280000 : "U-Boot Env"
0x00280000-0x00680000 : "Kernel"
0x00680000-0x10000000 : "File System"
...
```

At kernel's prompt command *cat /proc/mtd* will give you similar output:

```
# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00080000 00020000 "X-Loader"
mtd1: 001e0000 00020000 "U-Boot"
mtd2: 00020000 00020000 "U-Boot Env"
mtd3: 00400000 00020000 "Kernel"
mtd4: 0f980000 00020000 "File System"
```

While the first three partitions (X-Loader, U-Boot and U-Boot Env) are read only from kernel point of view, Kernel and File System partition can be written by kernel itself. To do this, you need MTD User modules in your kernel's root file system.

In this example we mount boot (FAT) partition of MMC card (using a dual boot card) to read kernel image (uImage) from. If you have network connection in your kernel, you can use this, too. Or you put uImage in your root file system. Goal is to have access to uImage from running kernel to be able to write it to NAND.

```
# mkdir -p /mnt/fat
# mount /dev/mmcblk0p1 /mnt/fat/
# ls -la /mnt/fat
-rwxr-xr-x    1 root     root         16740 Oct  7 17:28 mlo
-rwxr-xr-x    1 root     root        717116 Oct 24  2008 u-
boot.bin
-rwxr-xr-x    1 root     root       2106940 Oct 26  2008
uImage
# cp /mnt/fat/uImage .
# ls -la
-rwxr-xr-x    1 root     root       2106940 Oct 22 00:30
uImage
# flash_eraseall /dev/mtd3
Erasing 128 Kibyte @ 3e0000 -- 96 % complete.
# nandwrite /dev/mtd3 uImage
Input file is not page aligned
Data did not fit into device, due to bad blocks
: Success
#
```

File system

As with kernel, while X-Loader and U-Boot can be written only by U-Boot, for file system there are two ways to write them to NAND: Either by U-Boot (similar way as for X-Loader and U-Boot above) or from running kernel (e.g. booted from MMC card). A lot of users report they have issues with writing (root) file system with U-Boot. Main issue is that U-Boot has to write file system *exactly* in format

kernel expects. If there are minor incompatibilities, kernel will later not be able to read file system written by U-Boot.

So, while we document here how to write file system with U-Boot, **recommended** way is to write (root) file system by kernel itself. With this, it is ensured that kernel writes a file system it will later be able to read.

Writing file system with U-Boot
This way is *not* recommended. See above.

```
# mmcinit
# fatload mmc 0:1 80000000 rootfs.jffs2
reading rootfs.jffs2

12976128 bytes read
# nand unlock
device 0 whole chip
nand_unlock: start: 00000000, length: 268435456!
NAND flash successfully unlocked
# nand ecc sw
# nand erase 680000 F980000

NAND erase: device 0 offset 0x680000, size 0xf980000
Erasing at 0xffe0000 -- 100% complete.
OK
# nand write.jffs2 80000000 680000 ${file_size}

NAND write: device 0 offset 0x680000, size 0xc60000

Writing data at 0x12df800 -- 100% complete.
 12976128 bytes written: OK
#
```

Writing file system with kernel

This is the *recommended* way. See above.

First, we boot kernel with (root) file system on SD card, write (root) file system using file system image at SD card to Beagle's NAND with running kernel. After this is done, we switch kernel's boot arguments to take root file system from NAND instead of SD card, then.
To be able to manipulate/erase/write NAND from kernel's user space, we need MTD Utils (e.g. flash_eraseall).

If you don't have them already, you can get them

- using the angstrom demo, you can install via opkg install mtd-utils

- cross compiling them your self. See Section 3.4

For file system in Beagle's NAND, we use JFFS2. As JFFS2 is part of the standard git kernel, only thing is to configure kernel to be able to use JFFS2 is to enable in make menuconfig (check if already enabled):

```
CONFIG_JFFS2_FS=y
CONFIG_JFFS2_FS_DEBUG=0
CONFIG_JFFS2_FS_WRITEBUFFER=y
CONFIG_JFFS2_ZLIB=y
CONFIG_JFFS2_RTIME=y
```

Having kernel supporting JFFS2 and MTD Utils, we now first erase file system partition and create JFFS2 into it:

```
# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00080000 00020000 "X-Loader"
mtd1: 001e0000 00020000 "U-Boot"
mtd2: 00020000 00020000 "U-Boot Env"
mtd3: 00400000 00020000 "Kernel"
mtd4: 0f980000 00020000 "File System"
# flash_eraseall -j /dev/mtd4
Erasing 128 Kibyte @ f960000 -- 99 % complete. Cleanmarker
written at f960000.
```

Then, we can mount "File system" partition:

```
# cd /mnt
# mkdirt nand
# mount -t jffs2 /dev/mtdblock4 /mnt/nand
```

```
and extract the root file system image to it:
```

```
# cd nand
# tar xfz
<where_ever_your_root_fs_image_is_at_sd_card>/rootfs.tar.gz
.
```
... wait ...

```
# cd ..
# sync
# umount nand
```

Now, you should reboot your board and edit bootargs in U-Boot to configure root fs in NAND:

```
... root=/dev/mtdblock4 rootfstype=jffs2 ...
```

Do the Hardware Setup for booting u-boot over NAND Flash.
NOTE: If u-boot is not Flashed on the NAND, then refer to NAND Flash Procedure to do the same

Booting the Linux Image

Once you have u-boot booted over NAND or MMC, a Linux kernel image can be booted on. The Linux Kernel Image (uImage) can be downloaded to DDR memory using UART (time consuming), MMC, NAND (if it was stored in it), USB (Not supported yet).

The Below procedure gives MMC based Linux Kernel Booting.
Compile the Linux Kernel Image "uImage".
Copy the uImage file in MMC/SD card pre-formated for FAT32.
Download the uImage:

```
# mmcinit
# fatload mmc 0 0x80300000 uImage
```

Set/Configure the boot arguments
The filesystem to be mounted could be present in MMC, RAM (Ramdisk), NAND (if copied), Ethernet (using Ethernet over USB Dongle on USB HOST machine).
The Bootargs for each of these is shown below:

Bootargs for RAMDISK File System

```
# setenv bootargs console=ttyS2,115200n8 ramdisk_size=8192
root=/dev/ram0 rw rootfstype=ext2 initrd=0x81600000,8M
nohz=off
```

Bootargs for MMC File System

```
# setenv bootargs console=ttyS2,115200n8 noinitrd
root=/dev/mmcblk0p1 rootfstype=ext2 rw rootdelay=1 nohz=off
```

Getting File System on Beagle Board

RAMDISK File system

```
# fatload mmc 0 0x81600000 rd-ext2.bin
```

NOTE: rd-ext2.bin should have been copied onto MMC Card.

MMC File system

- Copy Filesystem on MMC/SD card.
- Format an MMC/SD card for ext2/ext3 file system using Linux Machine
- Mount the MMC/SD card on Host Linux Machine
- UnTar the Pre-built Filesystem
- Un-Mount the MMC/SD card on Host Linux Machine
- Remove the MMC/SD card that had uImage, and insert the MMC/SD card that has Filesystem.

Booting the Kernel Image

```
# bootm 0x80300000
```

Appendix A – Beagleboard

Expansion Connector Signals

Pin	Option A	Option B	Option C	Option D
1	VIO_1V8	VIO_1V8	VIO_1V8	VIO_1V8
2	DC_5V	DC_5V	DC_5V	DC_5V
3	MMC2_DAT7	GPIO_139		
4	McBSP3_DX	GPIO_140	**UART2_CTS**	
5	MMC2_DAT6	GPIO_138		
6	McBSP3_CLKX	GPIO_141	**UART2_RTS**	
7	MMC2_DAT5	GPIO_137		
8	McBSP3_FSX	GPIO_143	**UART2_RX**	
9	MMC2_DAT4	GPIO_136		
10	McBSP3_DR	GPIO_142	**UART2_TX**	
11	MMC2_DAT3	**McSPI3_CS0**	GPIO_135	
12	McBSP1_DX	McSPI4_SIMO	McBSP3_DX	GPIO_158
13	MMC2_DAT2	**McSPI3_CS1**	GPIO_134	
14	McBSP1_CLKX	McBSP3_CLKX	GPIO_162	
15	MMC2_DAT1	GPIO_133		
16	McBSP1_FSX	McSPI4_CS0	McBSP3_FSX	GPIO_161
17	MMC2_DAT0	**McSPI3_SOMI**	GPIO_132	
18	McBSP1_DR	McSPI4_SOMI	McBSP3_DR	GPIO_159
19	MMC2_CMD	**McSPI3_SIMO**	GPIO_131	
20	McBSP1_CLKR	McSPI4_CLK	SIM_CD	GPIO_156
21	MMC2_CLKO	**McSPI3_CLK**	GPIO_130	
22	McBSP1_FSR	GPIO_157		
23	**I2C2_SDA**	GPIO_183		
24	**I2C2_SCL**	GPIO_168		
25	REGEN	REGEN	REGEN	REGEN
26	nRESET	nRESET	nRESET	nRESET
27	**GND**	GND	GND	GND
28	**GND**	GND	GND	GND

We utilize the signals indicated in bold.

JTAG connection

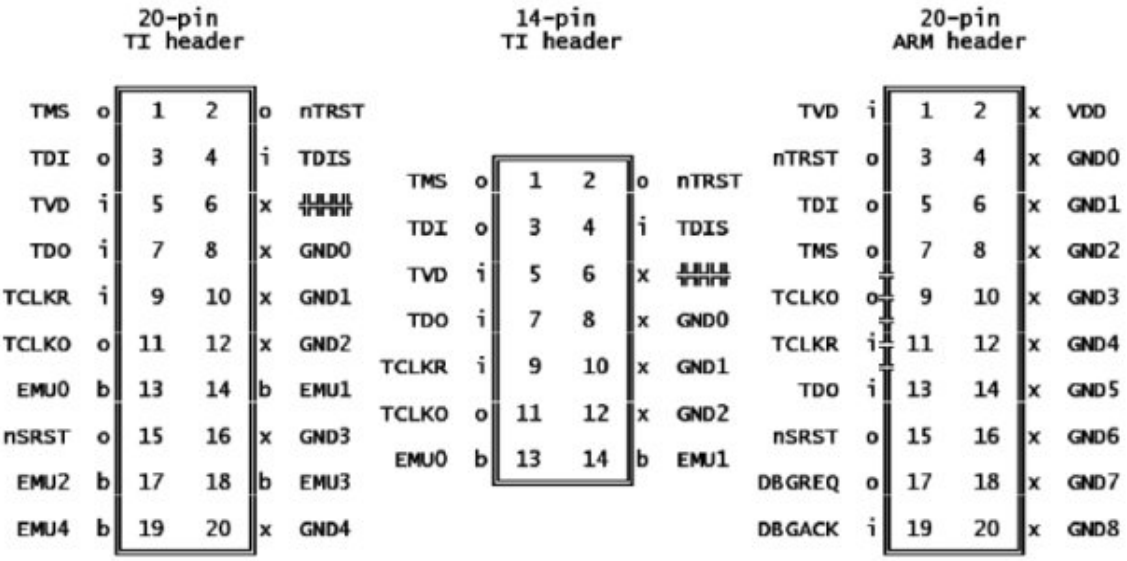
Note: JTAG on BeagleBoard uses 1.8V.

We use the BDI2000 with the omap35xx.cfg and regOMAP3500.def files that come with the BDI hardware.

Your BDI2000 needs firmware that supports Cortex-A8.

bdiGDB enhances the GNU debugger (GDB), with Background Debug Mode (BDM) and JTAG debugging via BDI2000

You can contact Ultimate Solutions, Inc. (www.ultsol.com) for BDI200 and firmware updates, they also carry a cable that supports the beagle board.



Texas Instruments uses (e.g. on BeagleBoard) a variation on the standard ARM 2x7 JTAG header:

ARM	PIN	TI
Vcc	1	TMS
GND	2	nTRST
nTRST	3	TDI
GND	4	GND
TDI	5	Vcc
GND	6	NC
TMS	7	TDO
GND	8	GND
TCK	9	RTCK
GND	10	GND
TDO	11	TCK

nRST	12	GND
Vcc	13	EMU-0
GND	14	EMU-1

User button

With the user button on BeagleBoard you can configure boot order. Depending on this button, the order used to scan boot devices is changed. The boot order is (the first is the default boot source):

- User button *not* pressed: NAND -> USB -> UART -> MMC
- User button *is* pressed: USB -> UART -> MMC -> NAND

Technically speaking, the user button configures pin SYS.BOOT[5].

Appendix B – Overo

Expansion Connectors

The bottom side of Overo has two (2) x 70-pin AVX 5602-14 connectors with 0.4mm pitch:
- connector J1 - features the LCD, PWM and analog signals.
- connector J4 - features the Extended Memory Bus and MMC signals.

Connector J1

Signal	Pin	Pin	Signal
N_MANUAL_RESET	1	70	GND
GPIO71_L_DD01	2	69	HSORF
GPIO70_L_DD00	3	68	HSOLF
GPIO73_L_DD03	4	67	USYSTEM
GPIO75_L_DD05	5	66	USYSTEM
GPIO72_L_DD02	6	65	POWERON
GPIO74_L_DD04	7	64	ADCIN2
GPIO127_TS_IRQ	8	63	
GPIO0_WAKEUP	9	62	
GPIO185_I2C3_SDA	10	61	GPIO93_L_DD23
GPIO80_L_DD10	11	60	GPIO82_L_DD12
GPIO81_L_DD11	12	59	SYSEN
GPIO184_L_I2C3_SCL	13	58	ADCIN2
GPIO128_GPS_PPS	14	57	MIC_MAIN_MF
GPIO92_L_DD22	15	56	GND
GPIO147_GPT8__PWM	16	55	GPIO145_GPT10_PWM
GPIO83_L_DD13	17	54	USBOTG_VBUS
GPIO144_GPT9_PWM	18	53	ADCIN6
GPIO84_L_DD14	19	52	VBACKUP
GPIO85_L_DD15	20	51	ADCIN5
GPIO146_GPT11_PWM	21	50	AGND
GPIO163_IR_CTS3	22	49	PWM1
GPIO91_L_DD21	23	48	ADCIN3
GPIO87_L_DD17	24	47	GPIO170_HDQ_1WIRE
GPIO88_L_DD18	25	46	USBOTG_ID

GPIO166_IR_TXD3	26	45	GPIO90_L_DD20
GPIO89_L_DD19	27	44	GPIO86_L_DD16
GPIO79_L_DD09	28	43	GPIO69_L_BIAS
GPIO77_L_DD07	29	42	PWM0
GPIO78_L_DD08	30	41	AUXRF
GPIO165_IR_RXD3	31	40	ADCIN4
GPIO66_L_PCLK	32	39	MIC_SUB_MF
GPIO76_L_DD06	33	38	AUXLF
GPIO68_L_FCLK	34	37	USBOTG_DM
GPIO67_L_LCLK	35	36	USBOTG_DP

Connector J4

Signal	Pin	Pin	Signal
VSYSTEM	1	70	EM_CLK
VSYSTEM	2	69	EM_NBF1
GND	3	68	EM_WAIT0
EM_NCS5_ETH0	4	67	EM_NCS6
EM_NCS4	5	66	EM_NCS0
EM_NWF	6	65	EM_NBF0
EM_NADV_ALE	7	64	EM_NCS1
EM_NOE	8	63	EM_NWP
GPIO65_ETH1_TRQ1	9	62	EM_A9
GPIO64_ETH0_NRESET	10	61	EM_A4
EM_A2	11	60	EM_A10
EM_A8	12	59	EM_A3
EM_A5	13	58	EM_A1
EM_A7	14	57	EM_A6
EM_D2	15	56	EM_D0
EM_D10	16	55	EM_D9
EM_D3	17	54	EM_D8
EM_D11	18	53	EM_D1
EM_D4	19	52	EM_D13
EM_D12	20	51	EM_D6
EM_D5	21	50	EM_D14
EM_D15	22	49	EM_D7

GPIO13_MMC3_CMD	23	48	GPIO151_RXD1
GPIO148_TXD1	24	47	GPIO150_MMC3_WP
GPIO176_ETH0_IRQ	25	46	GPIO49_MMC3_CD
GPIO18_MMC3_D0	26	45	GPIO173_SPI1_MISO
GPIO174_SPI1_CS0	27	44	GPIO172_SPI1_MOSI
GPIO168_USBH_CPEN	28	43	GPIO171_SPI1_CLK
GPIO14_MMC3_DAT4	29	42	GPIO175_SPI1_CS1
GPIO21_MMC3_DAT4	30	41	GPIO114_SPI1_NIRQ
GPIO17_MMC3_D	31	40	GPIO12_MMC3_CLK
USBH_UBUS	32	39	GPIO20_MMC3_D2
GND	33	38	GPIO23_MMC3_DAT5
USBH_DP	34	37	GPIO22_MMC3_DAT6
USBH_DM	35	36	GPIO19_MMC3_D1

Appendix C - MMC Boot Format

In order to create a bootable SD/MMC card under Linux compatible with OMAP3 boot ROM, you'd have to set a special geometry in the partition table, which is done through the fdisk "Expert mode".

First, lets clear the partition table:

```
# fdisk /dev/sdb

Command (m for help): o
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that, of course, the previous
content won't be recoverable.

Warning: invalid flag 0x0000 of partition table 4 will be corrected by
w(rite)
Print card info:
Command (m for help): p

Disk /dev/sdb: 128 MB, 128450560 bytes
....
```

Note card size in bytes. Needed later below.

Then go into "Expert mode":

```
Command (m for help): x
```

Now we want to set the geometry to 255 heads, 63 sectors and calculate the number of cylinders required for the particular SD/MMC card:

```
Expert command (m for help): h
Number of heads (1-256, default 4): 255

Expert command (m for help): s
Number of sectors (1-63, default 62): 63
Warning: setting sector offset for DOS compatiblity

Expert command (m for help): c
Number of cylinders (1-1048576, default 1011): 15
```

In this case 128MB card is used (reported as 128450560 bytes by fdisk above), thus 128450560 / 255 / 63 / 512 = 15.6 rounded down to 15 cylinders. Numbers there are 255 heads, 63 sectors, 512 bytes per sector.

Now, return to main mode and create a new partition:

```
Expert command (m for help): r
```

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-15, default 1): 1
Last cylinder or +size or +sizeM or +sizeK (1-15, default 15): 15
```

Mark it bootable:

```
Command (m for help): a
Partition number (1-4): 1
```

And change its type to FAT32:

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): c
Changed system type of partition 1 to c (W95 FAT32 (LBA))
```

The result is:

```
Command (m for help): p

Disk /dev/sdb: 128 MB, 128450560 bytes
255 heads, 63 sectors/track, 15 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *           1          15      120456    c  W95 FAT32
(LBA)
```

Now, really write configuration to card (until here, card is not changed):

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
```

What's left is to format our partition as FAT32 to be mounted and populated:

```
# mkfs.vfat -F 32 /dev/sdb1
mkfs.vfat 2.11 (12 Mar 2005)

# mount /dev/sdb1 /mnt/tmp
```
Note: If you use additional mkfs.vfat parameter -n you can give the card a name, e.g. for easier identification (i.e. mkfs.vfat -n omap3 -F 32 /dev/sdb1)
The SD/MMC card is now ready to be used to boot OMAP3 boards.

sfdisk
In order to format same card using sfdisk, one needs to do the following:

```
# sfdisk -H 255 -S 63 -C 15 /dev/sdb << EOF
> ,,b,*
> EOF
```

And follow with the mkfs.vfat commands above.

Appendix D - Board recovery

Normally, if you boot from MMC, you will get something like

...40T...

in terminal program connected to UART (115200 8N1). This is output from OMAP3's bootrom while scanning the UART for boot source before trying to boot from MMC card. If you don't get this, but want to boot from MMC, most probably bootrom doesn't reach the MMC boot stage any more. If you played with NAND before getting this, most probably NAND contains some broken content.

Depending on user button OMAP3 on BeagleBoard uses different boot order. Normal order if user button isn't pressed at power up is boot from

NAND -> USB -> UART -> MMC

in this order. Depending on the boot medium (e.g. MMC) this might fail if something bad is in NAND flash which confuses OMAP3 bootrom thus stopping it to reach MMC boot stage.

This might happen if you e.g. mess your NAND, e.g. something went wrong with NAND boot.

Recovery

First, we have to press user button at power up to switch boot order to

USB -> UART -> MMC -> NAND

to have option to boot from other sources than broken NAND (which is first if user button is not pressed).

Then, there are three options to boot from:

- MMC
- USB
- UART

MMC and USB recovery is described below. Goal of all ways is to get an U-Boot prompt again to erase the bad NAND content.

MMC recovery

MMC recovery should be straight forward. Press user button at power up and according to above boot order MMC boot is before NAND. With this, we should be able to boot as we did without pressing the user button before bricking the

board. But there are some broken MLO (x-loader) out there which fail to boot if something wrong is in NAND. E.g.:

```
...40T.........

Texas Instruments X-Loader 1.41
Starting on with MMC
Reading boot sector

150832 Bytes Read from MMC
Starting OS Bootloader from MMC...

U-Boot 1.3.3 (Jun 20 2008 - 17:06:22)

OMAP3530-GP rev 2, CPU-OPP2 L3-165MHz
OMAP3 Beagle Board + LPDDR/NAND
RAM Configuration:
Bank #0: 80000000 128 MB
Bank #1: 88000000  0 kB
NAND:  NAND device: Manufacturer ID: 0x2c, Chip ID: 0x01 ( AND
128MiB 3,3V 8-bit)
NAND bus width 16 instead 8 bit
0 MiB
<hang, no prompt>
```

This seems to happen with both MLO's from Beagle source code page (381MHz and 500MHz one) independent of U-Boot version.

Thus, you have to use a special (?) MLO for recovery to get a U-Boot prompt. Replacing MLO used above on MMC/SD card with this recovery MLO we get a U-Boot prompt while pressing the user button at power up:

```
...40T.........

Texas Instruments X-Loader 1.41
Starting on with MMC
Reading boot sector

150832 Bytes Read from MMC
Starting OS Bootloader from MMC...

U-Boot 1.3.3 (Jun 20 2008 - 17:06:22)

OMAP3530-GP rev 2, CPU-OPP2 L3-165MHz
OMAP3 Beagle Board + LPDDR/NAND
RAM Configuration:
Bank #0: 80000000 128 MB
```

```
Bank #1: 88000000  0 kB
NAND:  256 MiB
In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
OMAP3 beagleboard.org #
```

U-Boot version doesn't seem to matter. Then you can erase NAND start (e.g. using U-Boot 1.3.3 commands):

```
OMAP3 beagleboard.org # nand unlock
device 0 whole chip
nand_unlock: start: 00000000, length: 268435456!
NAND flash successfully unlocked
OMAP3 beagleboard.org # nand erase 0 80000

NAND erase: device 0 offset 0x0, size 0x80000
Erasing at 0x60000 -- 100% complete.
OK
OMAP3 beagleboard.org #
```

If you now re-power your board without pressing the user board it should work as before.

USB recovery

You can use USB boot utility together with U-Boot V2 and then use U-Boot V2's loadb to load U-Boot (V1). Binary: U-boot V2

Note: USB download can only load programs into OMAP3's internal SRAM. This is 64k, so too small for U-Boot (V1). But unfortunately, U-Boot V2 currently lacks NAND support. So we have to use:

USB download -> U-Boot V2 (SRAM) loadb -> U-Boot (V1) (SDRAM) NAND erase

For this, get usbload and U-Boot V2 using above links, start usbload tool at PC and while ... plug in USB OTG (power) cable. At host, this will result in:

```
> ./omap3_usbload uboot_v2.bin

TI OMAP3 USB boot ROM tool, version 0.1
(c) 2008 Martin Mueller <martinmm@pfump.org>

.........................................
```

```
found device!
download ok
>
```

And at target you will get:

```
U-Boot 2.0.0-rc5-git (Jun 30 2008 - 20:16:02)

Board: Texas Instrument's SDP343x
Malloc Space: 0x87bfff10 -> 0x87ffff10 (size  4 MB)
running /env/bin/init...
not found
X-load 343x>
```

Now, you can use this running U-Boot V2 to download U-Boot (V1) using loadb command:

```
X-load 343x> devinfo
devices:
|----uart3
|----ram0
|----filesystem: /
|----filesystem: /dev

drivers:
serial_ns16550
    ramfs
    devfs
      ram
X-load 343x> help loadb
[OPTIONS]
 -d device - which device to download - defaults to /dev/mem
 -o offset - what offset to download - defaults to 0
 -b baud   - baudrate at which to download - defaults to console
baudrate

X-load 343x> loadb -d /dev/ram0
## Ready for binary (kermit) download to 0x00000000 offset on
/dev/ram0 device at 115200 bps...
```

Now, send U-Boot (V1) binary (i.e. u-boot.bin) using kermit download of you terminal program. When this is finished:

```
## Total Size      = 0x00023d64 = 146788 Bytes

X-load 343x> help go
addr [arg ...]
```

```
   - start application at address 'addr'
     passing 'arg' as arguments

X-load 343x> go 0x80000000
## Starting application at 0x80000000 ...

U-Boot 1.3.3 (Jul  6 2008 - 10:33:59)

OMAP3530-GP rev 2, CPU-OPP2 L3-165MHz
OMAP3 Beagle Board + LPDDR/NAND
RAM Configuration:
Bank #0: 80000000 128 MB
Bank #1: 88000000  0 kB
NAND:  256 MiB
In:    serial
Out:   serial
Err:   serial
OMAP3 beagleboard.org #
```

Now, you have your U-Boot (V1) prompt. This can be used now to erase (broken) parts in NAND:

```
OMAP3 beagleboard.org # nand unlock
device 0 whole chip
nand_unlock: start: 00000000, length: 268435456!
NAND flash successfully unlocked
OMAP3 beagleboard.org # nand erase 0 80000

NAND erase: device 0 offset 0x0, size 0x80000
Erasing at 0x60000 -- 100% complete.
OK
OMAP3 beagleboard.org #
```

If you now re-power your board without pressing the user board it should work as before.