

The DENX U-Boot and Linux Guide (DULG) for enbw_cmc

Table of contents:

- [1. Abstract](#)
- [2. Introduction](#)
 - [2.1. Copyright](#)
 - [2.2. Disclaimer](#)
 - [2.3. Availability](#)
 - [2.4. Credits](#)
 - [2.5. Translations](#)
 - [2.6. Feedback](#)
 - [2.7. Conventions](#)
- [3. Embedded Linux Development Kit](#)
 - [3.1. ELDK Availability](#)
 - [3.2. ELDK Getting Help](#)
 - [3.3. Supported Host Systems](#)
 - [3.4. Supported Target Architectures](#)
 - [3.5. Installation](#)
 - [3.5.1. Product Packaging](#)
 - [3.5.2. Downloading the ELDK](#)
 - [3.5.3. Initial Installation](#)
 - [3.5.4. Installation and Removal of Individual Packages](#)
 - [3.5.5. Removal of the Entire Installation](#)
 - [3.6. Working with ELDK](#)
 - [3.6.1. Switching Between Multiple Installations](#)
 - [3.7. Mounting Target Components via NFS](#)
 - [3.8. Rebuilding ELDK Components](#)
 - [3.8.1. ELDK Source Distribution](#)
 - [3.8.2. Rebuilding Target Packages](#)
 - [3.8.3. Rebuilding ELDT Packages](#)
 - [3.9. ELDK Packages](#)
 - [3.9.1. List of ELDT Packages](#)
 - [3.9.2. List of Target Packages](#)
 - [3.10. Rebuilding the ELDK from Scratch](#)
 - [3.10.1. ELDK Build Process Overview](#)
 - [3.10.2. Setting Up ELDK Build Environment](#)
 - [3.10.3. build.sh Usage](#)
 - [3.10.4. Format of the cpkgs.lst and tpkgs.lst Files](#)
- [4. System Setup](#)
 - [4.1. Serial Console Access](#)
 - [4.2. Configuring the "cu" command](#)
 - [4.3. Configuring the "kermit" command](#)
 - [4.4. Using the "minicom" program](#)
 - [4.5. Permission Denied Problems](#)
 - [4.6. Configuration of a TFTP Server](#)
 - [4.7. Configuration of a BOOTP / DHCP Server](#)
 - [4.8. Configuring a NFS Server](#)
- [5. Das U-Boot](#)
 - [5.1. Current Versions](#)
 - [5.2. Unpacking the Source Code](#)
 - [5.3. Configuration](#)
 - [5.4. Installation](#)
 - [5.4.1. Before You Begin](#)
 - [5.4.1.1. Installation Requirements](#)
 - [5.4.1.2. Board Identification Data](#)
 - [5.4.2. Installation Using a BDM/JTAG Debugger](#)
 - [5.4.3. Installation using U-Boot](#)
 - [5.5. Tool Installation](#)
 - [5.6. Initialization](#)
 - [5.7. Initial Steps](#)
 - [5.8. The First Power-On](#)
 - [5.9. U-Boot Command Line Interface](#)
 - [5.9.1. Information Commands](#)
 - [5.9.1.1. bdfinfo - print Board Info structure](#)
 - [5.9.1.2. coninfo - print console devices and informations](#)
 - [5.9.1.3. flinfo - print FLASH memory information](#)
 - [5.9.1.4. iminfo - print header information for application image](#)
 - [5.9.1.5. help - print online help](#)
 - [5.9.2. Memory Commands](#)
 - [5.9.2.1. base - print or set address offset](#)
 - [5.9.2.2. crc32 - checksum calculation](#)
 - [5.9.2.3. cmp - memory compare](#)
 - [5.9.2.4. cp - memory copy](#)
 - [5.9.2.5. md - memory display](#)
 - [5.9.2.6. mm - memory modify \(auto-incrementing\)](#)
 - [5.9.2.7. mtest - simple RAM test](#)
 - [5.9.2.8. mw - memory write \(fill\)](#)
 - [5.9.2.9. nm - memory modify \(constant address\)](#)
 - [5.9.2.10. loop - infinite loop on address range](#)
 - [5.9.3. Flash Memory Commands](#)
 - [5.9.3.1. cp - memory copy](#)
 - [5.9.3.2. flinfo - print FLASH memory information](#)
 - [5.9.3.3. erase - erase FLASH memory](#)
 - [5.9.3.4. protect - enable or disable FLASH write protection](#)
 - [5.9.3.5. mtdparts - define a Linux compatible MTD partition scheme](#)
 - [5.9.3.6. UBI Usage in U-Boot](#)
 - [5.9.4. Execution Control Commands](#)
 - [5.9.4.1. source - run script from memory](#)
 - [5.9.4.2. bootm - boot application image from memory](#)
 - [5.9.4.3. go - start application at address 'addr'](#)
 - [5.9.5. Download Commands](#)
 - [5.9.5.1. bootp - boot image via network using BOOTP/TFTP protocol](#)
 - [5.9.5.2. dhcp - invoke DHCP client to obtain IP/boot params](#)
 - [5.9.5.3. loadb - load binary file over serial line \(kermit mode\)](#)
 - [5.9.5.4. loads - load S-Record file over serial line](#)
 - [5.9.5.5. rarpboot - boot image via network using RARP/TFTP protocol](#)
 - [5.9.5.6. tftpboot - boot image via network using TFTP protocol](#)

- [5.9.6. Environment Variables Commands](#)
 - [5.9.6.1. printenv- print environment variables](#)
 - [5.9.6.2. saveenv - save environment variables to persistent storage](#)
 - [5.9.6.3. setenv - set environment variables](#)
 - [5.9.6.4. run - run commands in an environment variable](#)
 - [5.9.6.5. bootd - boot default, i.e., run 'bootcmd'](#)
 - [5.9.7. Flattened Device Tree support](#)
 - [5.9.7.1. fdt addr - select FDT to work on](#)
 - [5.9.7.2. fdt list - print one level](#)
 - [5.9.7.3. fdt print - recursive print](#)
 - [5.9.7.4. fdt mknod - create new nodes](#)
 - [5.9.7.5. fdt set - set node properties](#)
 - [5.9.7.6. fdt rm - remove nodes or properties](#)
 - [5.9.7.7. fdt move - move FDT blob to new address](#)
 - [5.9.7.8. fdt chosen - fixup dynamic info](#)
 - [5.9.8. Special Commands](#)
 - [5.9.8.1. i2c - I2C sub-system](#)
 - [5.9.9. Storage devices](#)
 - [5.9.9.1. MMC devices](#)
 - [5.9.9.2. NAND devices](#)
 - [5.9.9.2.1. nand bad - show bad block information](#)
 - [5.9.9.2.2. nand erase - erase region](#)
 - [5.9.9.2.3. nand write - write to NAND device](#)
 - [5.9.9.2.4. nand read - read from NAND device](#)
 - [5.9.10. Miscellaneous Commands](#)
 - [5.9.10.1. date - get/set/reset date & time](#)
 - [5.9.10.2. echo - echo args to console](#)
 - [5.9.10.3. reset - Perform RESET of the CPU](#)
 - [5.9.10.4. sleep - delay execution for some time](#)
 - [5.9.10.5. version - print monitor version](#)
 - [5.9.10.6. ? - alias for 'help'](#)
 - [5.10. U-Boot Environment Variables](#)
 - [5.11. U-Boot Scripting Capabilities](#)
 - [5.12. U-Boot Standalone Applications](#)
 - [5.12.1. "Hello World" Demo](#)
 - [5.12.2. Timer Demo](#)
 - [5.13. U-Boot Image Formats](#)
 - [5.14. U-Boot Advanced Features](#)
 - [5.14.1. Boot Count Limit](#)
 - [6. Embedded Linux Configuration](#)
 - [6.1. Download and Unpack the Linux Kernel Sources](#)
 - [6.2. Kernel Configuration and Compilation](#)
 - [6.3. Installation](#)
 - [7. Booting Embedded Linux](#)
 - [7.1. Introduction](#)
 - [7.2. Flattened Device Tree Blob](#)
 - [7.3. Passing Kernel Arguments](#)
 - [7.4. Boot Arguments Unleashed](#)
 - [7.5. Networked Operation with Root Filesystem over NFS](#)
 - [7.5.1. Bootlog of tftp'd Linux kernel with Root Filesystem over NFS](#)
 - [7.6. Boot from NAND Flash Memory](#)
 - [7.7. Standalone Operation with Ramdisk Image](#)
 - [8. Building and Using Modules](#)
 - [9. Advanced Topics](#)
 - [9.1. Flash Filesystems](#)
 - [9.1.1. Memory Technology Devices](#)
 - [9.1.2. Journalling Flash File System](#)
 - [9.1.3. Second Version of JFFS](#)
 - [9.1.4. Compressed ROM Filesystem](#)
 - [9.1.5. UBI and UBIFS file systems](#)
 - [9.1.5.1. Create Device Files](#)
 - [9.1.5.2. Using UBI on NAND Flash:](#)
 - [9.1.5.3. Creating UBIFS File System Images](#)
 - [9.1.5.3.1. Determining the Parameters of the used Flash Types:](#)
 - [9.1.5.3.2. Create some Test File System Hierarchy](#)
 - [9.1.5.3.3. Installing UBIFS images into existing UBI Volume:](#)
 - [9.1.5.3.4. Installing UBI images \(if no UBI Volumes exist\):](#)
 - [9.2. The TMPFS Virtual Memory Filesystem](#)
 - [9.2.1. Mount Parameters](#)
 - [9.2.2. Kernel Support for tmpfs](#)
 - [9.2.3. Usage of tmpfs in Embedded Systems](#)
 - [9.3. Using MultiMediaCards in Linux](#)
 - [9.4. Splash Screen Support in Linux](#)
 - [9.5. Root File System: Design and Building](#)
 - [9.5.1. Root File System on a Ramdisk](#)
 - [9.5.2. Root File System on a JFFS2 File System](#)
 - [9.5.3. Root File System on a cramfs File System](#)
 - [9.5.4. Root File System on a Read-Only ext2 File System](#)
 - [9.5.5. Root File System on a Flash Card](#)
 - [9.5.6. Root File System in a Read-Only File in a FAT File System](#)
 - [9.6. Root File System Selection](#)
 - [9.7. Overlay File Systems](#)
 - [9.8. The Persistent RAM File system \(PRAMFS\)](#)
 - [9.8.1. Mount Parameters](#)
 - [9.8.2. Example](#)
- [10. Debugging](#)
 - [10.1. Debugging of U-Boot](#)
 - [10.1.1. Debugging of U-Boot Before Relocation](#)
 - [10.1.2. Debugging of U-Boot After Relocation](#)
 - [10.2. Linux Kernel Debugging](#)
 - [10.2.1. Linux Kernel and Statically Linked Device Drivers](#)
 - [10.2.2. Dynamically Loaded Device Drivers \(Modules\)](#)
 - [10.2.3. GDB Macros to Simplify Module Loading](#)
 - [10.3. GDB Startup File and Utility Scripts](#)
 - [10.4. Tips and Tricks](#)
 - [10.5. Application Debugging](#)
 - [10.5.1. Local Debugging](#)
 - [10.5.2. Remote Debugging](#)
 - [10.6. Debugging with Graphical User Interfaces](#)
- [11. Simple Embedded Linux Framework](#)

- [12. Books, Mailing Lists, Links, etc.](#)
 - [12.1. Application Notes](#)
 - [12.2. Further Reading](#)
 - [12.2.1. License Issues](#)
 - [12.2.2. Linux kernel](#)
 - [12.2.3. General Linux / Unix programming](#)
 - [12.2.4. Network Programming](#)
 - [12.2.5. C++ programming](#)
 - [12.2.6. Java programming](#)
 - [12.2.7. Power Architecture® Programming](#)
 - [12.2.8. Embedded Topics](#)
 - [12.3. Mailing Lists](#)
 - [12.4. Links](#)
 - [12.5. Tools](#)
- [13. Appendix](#)
 - [13.1. Flat Device Tree](#)
 - [13.2. BDI2000 Configuration file](#)
- [14. FAQ - Frequently Asked Questions](#)
 - [14.1. ELDK](#)
 - [14.1.1. ELDK Installation under FreeBSD](#)
 - [14.1.2. ELDK Installation Hangs](#)
 - [14.1.3. gvfs: Permission Denied](#)
 - [14.1.4. Installation on Local Harddisk](#)
 - [14.1.5. System Include Files Missing](#)
 - [14.1.6. patch: command not found](#)
 - [14.1.7. ELDK Include Files Missing](#)
 - [14.1.8. Using the ELDK on a 64 bit platform](#)
 - [14.1.9. How can I check if Floating Point support is working?](#)
 - [14.1.10. ELDK 2.x Installation Aborts](#)
 - [14.1.11. Enable SSH Access](#)
 - [14.2. U-Boot](#)
 - [14.2.1. Can U-Boot be configured such that it can be started in RAM?](#)
 - [14.2.2. Relocation cannot be done when using -mrelocatable](#)
 - [14.2.3. Source object has EABI version 4, but target has EABI version 0](#)
 - [14.2.4. U-Boot crashes after relocation to RAM](#)
 - [14.2.5. Warning - bad CRC, using default environment](#)
 - [14.2.6. Net: No ethernet found](#)
 - [14.2.7. Wrong debug symbols after relocation](#)
 - [14.2.8. Decoding U-Boot Crash Dumps](#)
 - [14.2.9. Porting Problem: cannot move location counter backwards](#)
 - [14.2.10. U-Boot Doesn't Run after Upgrading my Compiler](#)
 - [14.2.11. How Can I Reduce The Image Size?](#)
 - [14.2.12. Erasing Flash Fails](#)
 - [14.2.13. Ethernet Does Not Work](#)
 - [14.2.14. Where Can I Get a Valid MAC Address from?](#)
 - [14.2.15. Why do I get TFTP timeouts?](#)
 - [14.2.16. Why is my Ethernet operation not reliable?](#)
 - [14.2.17. How the Command Line Parsing Works](#)
 - [14.2.17.1. Old, simple command line parser](#)
 - [14.2.17.2. Hush shell](#)
 - [14.2.17.3. Hush shell scripts](#)
 - [14.2.17.4. General rules](#)
 - [14.2.18. How can I load and uncompress a compressed image](#)
 - [14.2.19. How can I create an ulmage from a ELF file](#)
 - [14.2.20. My standalone program does not work](#)
 - [14.2.21. Linux hangs after uncompressing the kernel](#)
 - [14.2.22. How can I implement automatic software updates?](#)
 - [14.3. Linux](#)
 - [14.3.1. Linux crashes randomly](#)
 - [14.3.2. Linux crashes when uncompressing the kernel](#)
 - [14.3.3. Linux Post Mortem Analysis](#)
 - [14.3.4. Linux kernel register usage](#)
 - [14.3.5. Linux Kernel Ignores my bootargs](#)
 - [14.3.6. Cannot configure Root Filesystem over NFS](#)
 - [14.3.7. Linux Kernel Panics because "init" process dies](#)
 - [14.3.8. Unable to open an initial console](#)
 - [14.3.9. System hangs when entering User Space \(ARM\)](#)
 - [14.3.10. Mounting a Filesystem over NFS hangs forever](#)
 - [14.3.11. Ethernet does not work in Linux](#)
 - [14.3.12. Loopback interface does not work](#)
 - [14.3.13. Linux kernel messages are not printed on the console](#)
 - [14.3.14. Linux ignores input when using the framebuffer driver](#)
 - [14.3.15. How to switch off the screen saver and the blinking cursor?](#)
 - [14.3.16. BogoMIPS Value too low](#)
 - [14.3.17. Linux Kernel crashes when using a ramdisk image](#)
 - [14.3.18. Ramdisk Greater than 4 MB Causes Problems](#)
 - [14.3.19. Combining a Kernel and a Ramdisk into a Multi-File Image](#)
 - [14.3.20. Adding Files to Ramdisk is Non Persistent](#)
 - [14.3.21. Kernel Configuration for PCMCIA](#)
 - [14.3.22. Configure Linux for PCMCIA Cards using the Card Services package](#)
 - [14.3.23. Configure Linux for PCMCIA Cards without the Card Services package](#)
 - [14.3.23.1. Using a MacOS Partition Table](#)
 - [14.3.23.2. Using a MS-DOS Partition Table](#)
 - [14.3.24. Boot-Time Configuration of MTD Partitions](#)
 - [14.3.25. Use NTP to synchronize system time against RTC](#)
 - [14.3.26. Configure Linux for XIP \(Execution In Place\)](#)
 - [14.3.26.1. XIP Kernel](#)
 - [14.3.26.2. Cramfs Filesystem](#)
 - [14.3.26.3. Hints and Notes](#)
 - [14.3.26.4. Space requirements and RAM saving, an example](#)
 - [14.3.27. Use SCC UART with Hardware Handshake](#)
 - [14.3.28. How can I access U-Boot environment variables in Linux?](#)
 - [14.3.29. The =appWeb= server hangs *OR* /dev/random hangs](#)
 - [14.3.30. Swapping over NFS](#)
 - [14.3.31. Using NFSv3 for NFS Root Filesystem](#)
 - [14.3.32. Using and Configuring the SocketCAN Driver](#)
 - [14.3.33. Telnet / SSH \(dropbear\) server not working](#)
 - [14.4. Self](#)
 - [14.4.1. How to Add Files to a SELF Ramdisk](#)
 - [14.4.2. How to Increase the Size of the Ramdisk](#)

- [14.5. RTAI](#)
 - [14.5.1. Conflicts with asm clobber list](#)
- [14.6. BDI2000](#)
 - [14.6.1. Where can I find BDI2000 Configuration Files?](#)
 - [14.6.2. How to Debug Linux Exceptions](#)
 - [14.6.3. How to single step through "RFI" instruction](#)
 - [14.6.4. Setting a breakpoint doesn't work](#)
 - [14.6.5. Remote 'g' packet reply is too long](#)
- [14.7. Motorola LITE5200 Board](#)
 - [14.7.1. LITE5200 Installation Howto](#)
 - [14.7.2. USB does not work on Lite5200 board](#)
- [15. Glossary](#)

1. Abstract

This is the DENX U-Boot and Linux Guide to Embedded [PowerPC](#), ARM and MIPS Systems.

The document describes how to configure, build and use the firmware **Das U-Boot** (typically abbreviated as just "U-Boot") and the operating system **Linux** for Embedded [PowerPC](#), ARM and MIPS Systems.

The focus of this version of the document is on enbw_cmc boards.

This document was generated at 09 Feb 2012 - 20:47.

- [2. Introduction](#)
 - [2.1. Copyright](#)
 - [2.2. Disclaimer](#)
 - [2.3. Availability](#)
 - [2.4. Credits](#)
 - [2.5. Translations](#)
 - [2.6. Feedback](#)
 - [2.7. Conventions](#)

2. Introduction

This document describes how to use the firmware U-Boot and the operating system Linux in Embedded [Power Architecture®](#), ARM and MIPS Systems.

There are many steps along the way, and it is nearly impossible to cover them all in depth, but we will try to provide all necessary information to get an embedded system running from scratch. This includes all the tools you will probably need to configure, build and run U-Boot and Linux.

First, we describe how to install the Cross Development Tools [Embedded Linux Development Kit](#) which you probably need - at least when you use a standard x86 PC running Linux or a Sun Solaris 2.6 system as build environment.

Then we describe what needs to be done to connect to the serial console port of your target: you will have to configure a terminal emulation program like `cu` or `kermit`.

In most cases you will want to load images into your target using ethernet; for this purpose you need [TFTP](#) and [DHCP](#) / [BOOTP](#) servers. A short description of their configuration is given.

A description follows of what needs to be done to configure and build the U-Boot for a specific board, and how to install it and get it working on that board.

The configuration, building and installing of Linux in an embedded configuration is the next step. We use **SELF**, our **Simple Embedded Linux Framework**, to demonstrate how to set up both a development system (with the root filesystem mounted over NFS) and an embedded target configuration (running from a ramdisk image based on `busybox`).

This document does **not** describe what needs to be done to port U-Boot or Linux to a new hardware platform. Instead, it is silently assumed that your board is already supported by U-Boot and Linux.

The focus of this document is on enbw_cmc boards.

2.1. Copyright

Copyright (C) 2001 - 2011 by Wolfgang Denk, DENX Software Engineering.
 Copyright (C) 2003 - 2011 by Detlev Zundel, DENX Software Engineering.
 Copyright (C) 2003 - 2011 by contributing authors

You have the freedom to distribute copies of this document in any format or to create a derivative work of it and distribute it provided that you:

- Distribute this document or the derivative work at no charge at all. It is **not** permitted to sell this document or the derivative work or to include it into any package or distribution that is not freely available to everybody.
- Send your derivative work (in the most suitable format such as sgml) to the author.
- License the derivative work with this same license or use [GPL](#). Include a copyright notice and at least a pointer to the license used.
- Give due credit to previous authors and major contributors.

It is requested that corrections and/or comments be forwarded to the author.

If you are considering to create a derived work other than a translation, it is requested that you discuss your plans with the author.

2.2. Disclaimer

Use the information in this document at your own risk. DENX disavows any potential liability for the contents of this document. Use of the concepts, examples, and/or other content of this document is entirely at your own risk. All copyrights are owned by their owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark. Naming of particular products or brands should not be seen as endorsements.

2.3. Availability

The latest version of this document is available in a number of formats:

- HTML http://www.denx.de/wiki/publish/DULG/DULG-enbw_cmc.html
- plain ASCII text http://www.denx.de/wiki/publish/DULG/DULG-enbw_cmc.txt
- PostScript European A4 format http://www.denx.de/wiki/publish/DULG/DULG-enbw_cmc.ps
- PDF European A4 format http://www.denx.de/wiki/publish/DULG/DULG-enbw_cmc.pdf

2.4. Credits

A lot of the information contained in this document was collected from several mailing lists. Thanks to anybody who contributed in one form or another.

2.5. Translations

None yet.

2.6. Feedback

Any comments or suggestions can be mailed to the author: Wolfgang Denk at wd@denx.de.

2.7. Conventions

| Descriptions | Appearance |
|---|----------------------------|
| Warnings | <input type="checkbox"/> |
| Hint | <input type="checkbox"/> |
| Notes | Note. |
| Information requiring special attention | Warning |
| File Names | <i>file.extension</i> |
| Directory Names | <i>directory</i> |
| Commands to be typed | a command |
| Applications Names | another application |
| Prompt of users command under bash shell | bash\$ |
| Prompt of root users command under bash shell | bash# |
| Prompt of users command under tcsh shell | tcsh\$ |
| Environment Variables | VARIABLE |
| Emphasized word | <i>word</i> |
| Code Example | ls -l |

- [3. Embedded Linux Development Kit](#)
 - [3.1. ELDK Availability](#)
 - [3.2. ELDK Getting Help](#)
 - [3.3. Supported Host Systems](#)
 - [3.4. Supported Target Architectures](#)
 - [3.5. Installation](#)
 - [3.5.1. Product Packaging](#)
 - [3.5.2. Downloading the ELDK](#)
 - [3.5.3. Initial Installation](#)
 - [3.5.4. Installation and Removal of Individual Packages](#)
 - [3.5.5. Removal of the Entire Installation](#)
 - [3.6. Working with ELDK](#)
 - [3.6.1. Switching Between Multiple Installations](#)
 - [3.7. Mounting Target Components via NFS](#)
 - [3.8. Rebuilding ELDK Components](#)
 - [3.8.1. ELDK Source Distribution](#)
 - [3.8.2. Rebuilding Target Packages](#)
 - [3.8.3. Rebuilding ELDT Packages](#)
 - [3.9. ELDK Packages](#)
 - [3.9.1. List of ELDT Packages](#)
 - [3.9.2. List of Target Packages](#)
 - [3.10. Rebuilding the ELDK from Scratch](#)
 - [3.10.1. ELDK Build Process Overview](#)
 - [3.10.2. Setting Up ELDK Build Environment](#)
 - [3.10.3. build.sh Usage](#)
 - [3.10.4. Format of the cpkgs.lst and tpkgs.lst Files](#)

3. Embedded Linux Development Kit

The Embedded Linux Development Kit (**ELDK**) includes the GNU cross development tools, such as the compilers, binutils, gdb, etc., and a number of pre-built target tools and libraries necessary to provide some functionality on the target system.

It is provided for free with full source code, including all patches, extensions, programs and scripts used to build the tools.

Some versions of [ELDK](#) (4.1) are available in two versions, which use Glibc resp. uClibc as the main C library for the target packages.

Packaging and installation is based on the RPM package manager.

3.1. [ELDK](#) Availability

The [ELDK](#) is available

- on DVD-ROM from [DENX Computer Systems](#)
- for download on the following server:

| FTP | HTTP |
|---|---|
| ftp://ftp.denx.de/pub/eldk/ | http://www.denx.de/ftp/pub/eldk/ |

- for download on the following mirrors:

| FTP | HTTP |
|---|---|
| ftp://ftp-stud.hs-esslingen.de/pub/Mirrors/eldk/ | http://ftp-stud.hs-esslingen.de/pub/Mirrors/eldk/ |
| ftp://mirror.switch.ch/mirror/eldk/ | http://mirror.switch.ch/ftp/mirror/eldk/ |
| not available | http://mira.sunsite.utk.edu/eldk/ |
| ftp://ftp.sunet.se/pub/Linux/distributions/eldk/ | http://ftp.sunet.se/pub/Linux/distributions/eldk/ |

3.2. [ELDK](#) Getting Help

Community support for the [ELDK](#) is available through the [ELDK Mailing List](#).
Previous postings to this mailing list are available from the [ELDK archives](#).

Commercial support is also available; please feel free to contact [DENX Software Engineering GmbH](#).

3.3. Supported Host Systems

The [ELDK](#) can be installed onto and operate with the following operating systems:

- [Fedora Core](#) 4, 5, 6 [Fedora](#) 7, 8, 9, 10, 11, 12
- [Red Hat](#) Linux 7.3, 8.0, 9
- [SuSE](#) Linux 8.x, 9.0, 9.1, 9.2, 9.3, 10.0
- [OpenSUSE](#) 10.2, 10.3 (32 Bit); OpenSUSE 11.0 (32 and 64 Bit)
- [Debian](#) 3.0 (Woody), 3.1 (Sarge) and 4.0 (Etch)
- [Ubuntu](#) 4.10, 5.04, 6.10, 8.04, 9.04, 9.10, 10.04
- [FreeBSD](#) 5.0

Users also reported successful installation and use of the [ELDK](#) on the following host systems:

- [Suse](#) Linux 7.2, 7.3
- [Mandrake](#) 8.2
- [Slackware](#) 8.1beta2
- [Gentoo](#) Linux 2006.1

Note: It may be necessary, and is usually recommended, to install the latest available software updates on your host system. For example, on [Fedora](#) systems, you can use `yum` or `apt-get` to keep your systems current.

3.4. Supported Target Architectures

The [ELDK](#) for ARM systems supports processors complying with the ARM architecture version 2 to 6. This includes ARM7, ARM9, XScale, AT91RM9200, i.MX31, S3C6400 and other ARM based systems.

The version of 4.2 and higher of [ELDK](#) has two ARM targets in distribution - one with the soft-float math support, and another one with the Vector Floating Point math support. Both targets comply with ARM Embedded Application Binary Interface ([EABI](#)).

The [ELDK](#) ARM target architectures are:

- arm-linux = soft-float math
- armVFP-linux = Vector Floating Point (VFP) math

There is also an [ELDK](#) for [PowerPC](#) and MIPS systems.

3.5. Installation

3.5.1. Product Packaging

Stable versions of the [ELDK](#) are distributed in the form of an ISO image, which can be either burned onto a DVD or mounted directly, using the loopback Linux device driver (Linux host only).

Development versions of the [ELDK](#) are available as directory trees so it is easy to update individual packages; instructions for download of these trees and creation of ISO images from it is described in section [3.5.2. Downloading the ELDK](#).

The [ELDK](#) contains an installation utility and a number of RPM packages, which are installed onto the hard disk of the cross development host by the installation procedure. The RPM packages can be logically divided into two parts:

- Embedded Linux Development Tools (ELDT)
- Target components

The first part contains the cross development tools that are executed on the host system. Most notably, these are the GNU cross compiler, binutils, and `gdb`. For a full list of the provided ELDT packages, refer to section [3.9.1. List of ELDT Packages](#) below.

The target components are pre-built tools and libraries which are executed on the target system. The [ELDK](#) includes necessary target components to provide a minimal working NFS-based environment for the target system. For a list of the target packages included in the [ELDK](#), refer to section [3.9.2. List of Target Packages](#) below.

The [ELDK](#) contains several independent sets of the target packages, one for each supported target [architecture CPU](#) family. Each set has been built using compiler code generation and optimization options specific to the respective target [CPU](#) family.

3.5.2. Downloading the [ELDK](#)

You can either download the ready-to-burn ISO-images from one of the mirror sites (see [3.1. ELDK Availability](#)), or you can download the individual files of the [ELDK](#) from the development directory tree and either use these directly for installation or create an ISO image that can be burned on DVD-ROM.

Change to a directory with sufficient free disk space; for the ARM version of the [ELDK](#) you need about 510 MiB, or twice as much (1.1 GiB) if you also want to create an ISO image in this directory.

To download the ISO image from the `arm-linux-x86/iso` directory of one of the mirror sites you can use standard tools like `wget` or `ncftpget`, for example:

```
bash$ wget ftp://ftp.sunet.se/pub/Linux/distributions/eldk/4.2/arm-linux-x86/iso/arm-2008-11-24.iso
```

If you want to download the whole [ELDK](#) directory tree instead you can - for example - use the `ncftp` [FTP](#) client:

```
bash$ ncftp ftp.sunet.se
...
ncftp / > cd /pub/Linux/distributions/eldk/4.2
ncftp /pub/Linux/distributions/eldk/4.2 > bin
ncftp /pub/Linux/distributions/eldk/4.2 > get -R arm-linux-x86/distribution
...
ncftp /pub/Linux/distributions/eldk/4.2 > bye
```

☐ If you don't find the `ncftp` tool on your system you can download the *NcFTP* client from <http://www.ncftp.com/download/>

There are a few executable files (binaries and scripts) in the [ELDK](#) tree. Make sure they have the execute permissions set in your local copy:

```
bash$ for file in \
>   tools/bin/rpm \
>   tools/usr/lib/rpm/rpmd \
>   install \
>   ELDK_MAKEDEV \
>   ELDK_FIXOWNER
> do
```

```
> chmod +x arm-linux-x86/distribution/$file
> done
```

Now create an ISO image from the directory tree:

```
bash$ mkisofs \
> -A "ELDK-4.2 -- Target: ARM -- Host: x86 Linux" \
> -publisher "(C) `date +%Y` DENX Software Engineering, www.denix.de" \
> -p "`id -nu`@`hostname` -- `date`" \
> -V arm-linux-x86 \
> -l -J -R -o eldk-arm-linux-x86.iso arm-linux-x86/distribution
```

This will create an ISO image *eldk-arm-linux-x86.iso* in your local directory that can be burned on DVD or mounted using the loopback device and used for installation as described above. Of course you can use the local copy of the directory tree directly for the installation, too.

Please refer to section [3.10.2. Setting Up ELDK Build Environment](#) for instructions on obtaining the build environment needed to re-build the [ELDK](#) from scratch.

3.5.3. Initial Installation

The initial installation is performed using the `install` utility located in the root of the [ELDK](#) ISO image directory tree. The `install` utility has the following syntax:

```
$ ./install [-d <dir>] [<cpu_family1>] [<cpu_family2>] ...
```

`-d <dir>` Specifies the root directory of the [ELDK](#) being installed. If omitted, the [ELDK](#) goes into the current directory.


`<cpu_family>` Specifies the target [CPU](#) family the user desires to install. If one or more `<cpu_family>` parameters are specified, only the target components specific to the respective [CPU](#) families are installed onto the host. If omitted, the target components for all supported target [architecture CPU](#) families are installed.

Note: Make sure that the `"exec"` option to the `mount` command is in effect when mounting the [ELDK](#) ISO image. Otherwise the `install` program cannot be executed. On some distributions, it may be necessary to modify the `/etc/fstab` file, adding the `"exec"` mount option to the `cdrom` entry - it may also be the case that other existing mount options, such as `"user"` prevent a particular configuration from mounting the [ELDK](#) DVD with appropriate `"exec"` permission. In such cases, consult your distribution documentation or mount the DVD explicitly using a command such as `"sudo mount -o exec /dev/cdrom /mnt/cdrom"` (sudo allows regular users to run certain privileged commands but may not be configured - run the previous command as root without `"sudo"` in the case that `"sudo"` has not been setup for use on your particular GNU/Linux system).

You can install the [ELDK](#) to any empty directory you wish, the only requirement being that you have to have write and execute permissions on the directory. The installation process does not require superuser privileges.

Depending on the parameters the `install` utility is invoked with, it installs one or more sets of target components. The ELDK packages are installed in any case.

Refer to section [3.6. Working with ELDK](#) for a sample usage of the [ELDK](#).

 **Note:** If you intend to use the installation as a root filesystem exported over NFS, then you now have to finish the configuration of the [ELDK](#) following the instructions in [3.7. Mounting Target Components via NFS](#).

 **Note:** Installation of the Glibc- and uClibc-based [ELDK](#) versions into one directory is not yet supported.

 **Note:** Installation of the 32-bit and 64-bit [ELDK](#) versions into one directory is not yet supported.

3.5.4. Installation and Removal of Individual Packages

The [ELDK](#) has an RPM-based structure. This means that on the ISO image, individual components of the [ELDK](#) are in the form of RPM packages, and after installation, the [ELDK](#) maintains its own database which contains information about installed packages. The RPM database is kept local to the specific [ELDK](#) installation, which allows you to have multiple independent [ELDK](#) installations on your host system. (That is, you can install several instances of [ELDK](#) under different directories and work with them independently). Also, this provides for easy installation and management of individual [ELDK](#) packages.

To list the installed [ELDK](#) RPM packages, use the following command:

```
bash$ ${CROSS_COMPILE}rpm -qa
```

To remove an [ELDK](#) package, use the following command:

```
bash$ ${CROSS_COMPILE}rpm -e <package_name>
```

To install a package, use the following command:

```
bash$ ${CROSS_COMPILE}rpm -i <package_file_name>
```


To update a package, use the following command:

```
bash$ ${CROSS_COMPILE}rpm -U <package_file_name>
```

For the above commands to work correctly, it is crucial that the correct `rpm` binary gets invoked. In case of multiple [ELDK](#) installations and RedHat-based host system, there may well be several `rpm` tools installed on the host system.

You must make sure, either by using an explicit path or by having set an appropriate `PATH` environment variable, that when you invoke `rpm` to install/remove components of a [ELDK](#) installation, it is the [ELDK](#)'s `rpm` utility that gets actually invoked. The `rpm` utility is located in the `bin` subdirectory relative to the [ELDK](#) root installation directory.

To avoid confusion with the host OS (RedHat) `rpm` utility, the [ELDK](#) creates symlinks to its `rpm` binary with the names such that it could be invoked using the `${CROSS_COMPILE}rpm` notation, for all supported [SCROSS_COMPILE](#) values.

 The standard (host OS) `rpm` utility allows various macros and configuration parameters to specified in user-specific `~/rpmrc` and `~/rpmmacros` files. The [ELDK](#) rpm tool also has this capability, but the names of the user-specific configuration files are `~/eldk_rpmrc` and `~/eldk_rpmmacros`, respectively.

3.5.5. Removal of the Entire Installation

To remove the entire [ELDK](#) installation, use the following command while in the [ELDK](#) root directory:

```
bash$ rm -rf <dir>
```

where `<dir>` specifies the root directory of the [ELDK](#) to be removed.

3.6. Working with ELDK

After the initial installation is complete, all you have to do to start working with the [ELDK](#) is to set and export the `CROSS_COMPILE` environment variable. Optionally, you may wish to add the `bin` and `usr/bin` directories of your [ELDK](#) installation to the value of your `PATH` environment variable. For instance, a sample [ELDK](#) installation and usage scenario looks as follows:

- Create a new directory where the [ELDK](#) is to be installed, say:

```
bash$ mkdir /opt/eldk
```

- Mount a CD or an ISO image with the distribution:

```
bash$ mount /dev/cdrom /mnt/cdrom
```

- Run the installation utility included on the distribution to install into that specified directory:

```
bash$ /mnt/cdrom/install -d /opt/eldk
```

- After the installation utility completes, export the `CROSS_COMPILE` variable:

```
bash$ export CROSS_COMPILE=arm-linux-gnueabi-
```

- ☐ The trailing '-' character in the `CROSS_COMPILE` variable value is optional and has no effect on the cross tools behavior. However, it is required when building Linux kernel and U-Boot images.

- Add the directories `/opt/eldk/usr/bin` and `/opt/eldk/bin` to `PATH`:

```
bash$ PATH=$PATH:/opt/eldk/usr/bin:/opt/eldk/bin
```

- Compile a file:

```
bash$ ${CROSS_COMPILE}gcc -o hello_world hello_world.c
```

- ☐ You can also call the cross tools using the generic prefix `arm-linux-` for example:

```
bash$ arm-linux-gcc -o hello_world hello_world.c
```

- or, equivalently:

```
bash /opt/eldk/usr/arm-linux/bin/gcc -o hello_world hello_world.c
```

The value of the `CROSS_COMPILE` variable must correspond to the target [CPU](#) family you want the cross tools to work for. Refer to the table below for the supported `CROSS_COMPILE` variable values:

3.6.A Table of possible values for `$CROSS_COMPILE`

| <code>CROSS_COMPILE</code> Value | Predefined Compiler Flag | FPU present or not |
|----------------------------------|---|--------------------|
| <code>arm-linux-</code> | <code>-mcpu=arm9 -msoft-float</code> | No |
| <code>armVFP-linux-</code> | <code>-mfpu=vfp -mfloat-abi=softfp</code> | Yes (VFP) |

3.6.1. Switching Between Multiple Installations

No special actions are required from the user to switch between multiple [ELDK](#) installations on the same host system. Which [ELDK](#) installation is used is determined entirely by the filesystem location of the binary that is being invoked. This approach can be illustrated using the following example.

Assume the directory `/work/denx_tools/usr/bin`, where the `arm-linux-gcc` compiler binary has been installed, is a part of the `PATH` environment variable. The user types the command as follows:

```
$ arm-linux-gcc -c myfile.c
```

To load the correct include files, find the correct libraries, spec files, etc., the compiler needs to know the [ELDK](#) root directory. The compiler determines this information by analyzing the shell command it was invoked with (`arm-linux-gcc` - without specifying the explicit path in this example) and, if needed, the value of the `PATH` environment variable. Thus, the compiler knows that it has been executed from the `/work/denx_tools/usr/bin` directory.

Then, it knows that the compiler is installed in the `usr/bin` subdirectory of the root installation directory, so the [ELDK](#), the compiler is a part of, has been installed in the subdirectories of the `/work/denx_tools` directory. This means that the target include files are in `/work/denx_tools/<target_cpu_variant>/usr/include`, and so on.

3.7. Mounting Target Components via NFS

The target components of the [ELDK](#) can be mounted via NFS as the root file system for your target machine. For instance, for an AT91-based target, and assuming the [ELDK](#) has been installed into the `/opt/eldk` directory, you can use the following directory as the NFS-based root file system:

```
/opt/eldk/arm
```

- ☐ Before the NFS-mounted root file system can work, you must create necessary device nodes in the `<ELDK_root>/<target_cpu_variant>/dev` directory. This process requires superuser privileges and thus cannot be done by the installation procedure (which typically runs as non-root). To facilitate creation of the device nodes, the [ELDK](#) provides a script named `ELDK_MAKEDEV`, which is located in the root of the [ELDK](#) distribution ISO image. The script accepts the following optional arguments:

- d <dir> Specifies the root directory of the [ELDK](#) being installed. If omitted, then the current directory is assumed.
- a <cpu_family> Specifies the target [CPU](#) family directory. If omitted, all installed target [architecture](#) directories will be populated with the device nodes.
- h Prints usage.

```
# /mnt/cdrom/ELDK_MAKEDEV -d /opt/eldk
```

NOTE: Compared to older versions of the [ELDK](#), options and behaviour of this command have been changed significantly. **Please read the documentation.**

- ☐ Some of the target utilities included in the [ELDK](#), such as `mount` and `su`, have the SUID bit set. This means that when run, they will have privileges of the file owner of these utilities. That is, normally, they will have the privileges of the user who installed the [ELDK](#) on the host system. However, for these utilities to work properly, they **must** have superuser privileges. This means that if the [ELDK](#) was not installed by the superuser, the file owner of the target [ELDK](#) utilities that have the SUID bit set must be changed to `root` before a target component may be mounted as the root file system. The [ELDK](#) distribution image contains an `ELDK_FIXOWNER` script, which you can use to change file owners of all the appropriate files of the [ELDK](#) installation to root. The script accepts the same arguments as the `ELDK_MAKEDEV` script above. Please note that you must have superuser privileges to run this script. For instance, if you have installed the [ELDK](#) in the `/opt/eldk` directory, you can use the following commands:

```
# cd /opt/eldk
# /mnt/cdrom/ELDK_FIXOWNER
```

Please note, that in the case that the installation directory, where the new [ELDK](#) distribution is being installed, is already populated with other [ELDK](#) distributions, the execution of the `ELDK_FIXOWNER` script without arguments will make the script work with all installed [ELDK](#) target [architecture](#) directories. This could take some time. To save the time, please use the `-a` argument to specify the appropriate target [architecture](#). For instance:

```
# cd /opt/eldk
# /mnt/cdrom/ELDK_FIXOWNER -a arm
```


3.8. Rebuilding [ELDK](#) Components

3.8.1. [ELDK](#) Source Distribution

The [ELDK](#) is distributed with the full sources of all the components, so you may rebuild any [ELDK](#) package. The sources are provided in the form of SRPM packages, distributed as a separate ISO image.

To rebuild a target or ELDT package, you must first install the appropriate source RPM package from the ISO image into the [ELDK](#) environment. This can be done using the following command:

```
$ ${CROSS_COMPILE}rpm -i /mnt/cdrom/SRPMS/<source_rpm_file_name>.src.rpm
```

After an [ELDK](#) source RPM is installed using the above command, its spec file and sources can be found in the subdirectories of the *<ELDK_root>/usr/src/denx* subdirectory.

The sections that follow provide detailed instructions on rebuilding ELDT and target components of the [ELDK](#).

3.8.2. Rebuilding Target Packages

All the target packages can be rebuilt from the provided source RPM packages. At first you have to install the Source RPM itself:

```
bash$ ${CROSS_COMPILE}rpm -iv <package_name>.src.rpm
```

Then you can rebuild the binary target RPM using the following command from the [ELDK](#) environment:

```
bash$ ${CROSS_COMPILE}rpmbuild -ba <package_name>.spec
```

In order for the rebuilding process to work correctly, the following conditions must be true:

- The [\\$CROSS_COMPILE](#) environment variable must be set as appropriate for the target [CPU](#) family.
- The *<ELDK_root>/usr/arm-linux/bin* directory must be in `PATH` before the */usr/bin* directory. This is to make sure that the command `gcc` results in the fact that the [ELDK](#) cross compiler is invoked, rather than the host `gcc`.

The newly built package can then be installed just as easily:

```
bash$ ${CROSS_COMPILE}rpm -i <package_name>.rpm
```

3.8.3. Rebuilding ELDT Packages

All the ELDT packages allow for rebuilding from the provided source RPM packages using the following command from the [ELDK](#) environment:

```
$ unset CROSS_COMPILE
$ <ELDK_root>/usr/bin/rpmbuild -ba <package_name>.spec
```

In order for the rebuilding process to work correctly, make sure all of the following is true:

- The [\\$CROSS_COMPILE](#) environment variable must **NOT** be set.
- Do **NOT** use the [\\$CROSS_COMPILE](#) command prefix.
- The *<ELDK_root>/usr/arm-linux/bin* directory must **NOT** be in `PATH`. This is to make sure that the command `gcc` causes invocation of the host `gcc`, rather than the [ELDK](#) cross compiler.

Note that the newly built package should be installed with the "global" `rpm`, not with the arch specific one:

```
bash$ <ELDK_root>/bin/rpm -i <package_name>.rpm
```

3.9. [ELDK](#) Packages

3.9.1. List of ELDT Packages

| Package Name | Package Version |
|------------------------------|---------------------------------|
| autoconf | 2.61-8 |
| automake | 1.10-5 |
| bison | 2.3-3 |
| crosstool-devel | 0.43-3 |
| dtc | 20070802-1 |
| elocaledef | 1-1 |
| ftdump | 20070802-1 |
| gdb | 6.7-2 |
| genext2fs | 1.4.1-1 |
| info | 4.8-15 |
| ldd | 0.1-1 |
| libtool | 1.5.22-11 |
| make | 3.81-6 |
| mkcramfs | 1.1-1 |
| mkimage | 1.3.1-1 |
| mtd-utils | 1.0.1-2 |
| rpm | 4.4.2-46_2 |
| rpm-build | 4.4.2-46_2 |
| sed | 4.1.4-1 |
| texinfo | 4.8-15 |

☐ Note: The **crosstool 0.43** ELDT package provides the following packages: **gcc 4.2.2**, **gcc-c++ 4.2.2**, **gcc-java 4.2.2**, **cpp 4.2.2** and **binutils 2.17.90**. For more information about the **crosstool** package please refer to <http://kegel.com/crosstool>.


3.9.2. List of Target Packages


| Package Name | Package Version |
|----------------------------|------------------|
| acl | 2.2.39-3.1 |
| appweb | 2.2.2-5 |
| attr | 2.4.32-2 |
| autoconf | 2.61-8 |
| bash | 3.2-9 |
| bc | 1.06-26 |
| bind | 9.4.1-8.P1 |
| binutils | 2.17.90-1 |
| binutils-devel | 2.17.90-1 |
| boa | 0.94.14-0.5.rc21 |
| busybox | 1.7.1-2 |
| byacc | 1.9.20050813-1 |
| bzip2 | 1.0.4-10 |
| bzip2-devel | 1.0.4-10 |
| bzip2-libs | 1.0.4-10 |
| ccid | 1.2.1-10 |
| chkconfig | 1.3.34-1 |
| coreutils | 6.9-3 |
| cpio | 2.6-27 |
| cpp | 4.2.2-2 |
| cracklib | 2.8.9-11 |
| cracklib-dicts | 2.8.9-11 |
| crosstool-targetcomponents | 0.43-3 |
| curl | 7.16.2-1 |
| cyrus-sasl | 2.1.22-6 |
| cyrus-sasl-devel | 2.1.22-6 |
| cyrus-sasl-lib | 2.1.22-6 |
| db4 | 4.5.20-5_2 |
| db4-devel | 4.5.20-5_2 |
| db4-utils | 4.5.20-5_2 |
| device-mapper | 1.02.17-7 |
| device-mapper-devel | 1.02.17-7 |
| device-mapper-libs | 1.02.17-7 |
| dhclient | 3.0.5-38 |
| dhcp | 3.0.5-38 |
| diffutils | 2.8.1-16 |
| directfb | 1.0.0-1 |
| dosfstools | 2.11-8 |
| dropbear | 0.50-1 |
| dtc | 20070802-1 |
| duma | 2.5.8-2 |
| e2fsprogs | 1.39-11 |
| e2fsprogs-devel | 1.39-11 |
| e2fsprogs-libs | 1.39-11 |
| ethtool | 5-1 |
| expat | 1.95.8-9 |
| expat-devel | 1.95.8-9 |
| file | 4.21-1 |
| file-libs | 4.21-1 |
| findutils | 4.2.29-2 |
| flex | 2.5.33-9 |
| freetype | 2.3.4-3 |
| freetype-devel | 2.3.4-3 |
| ftdump | 20070802-1 |
| ftp | 0.17-40 |
| gawk | 3.1.5-15 |
| gcc | 4.2.2-2 |


| | |
|--------------------|-----------------|
| gcc-c++ | 4.2.2-2 |
| gcc-java | 4.2.2-2 |
| gdb | 6.7-1 |
| glib | 1.2.10-26 |
| glib2 | 2.12.13-1 |
| glib2-devel | 2.12.13-1 |
| glib-devel | 1.2.10-26 |
| gmp | 4.1.4-12.3 |
| grep | 2.5.1-57 |
| groff | 1.18.1.4-2 |
| gzip | 1.3.11-2 |
| hdparm | 6.9-3 |
| httpd | 2.2.4-4.1 |
| httpd-devel | 2.2.4-4.1 |
| httpd-manual | 2.2.4-4.1 |
| initscripts | 8.54.1-1 |
| iproute | 2.6.20-2 |
| iptables | 1.3.8-2 |
| iputils | 20070202-3 |
| iscsitarget | 0.4.15-1 |
| kbd | 1.12-22 |
| kernel-headers | 2.6.24-1 |
| kernel-source | 2.6.24-1 |
| krb5-devel | 1.6.1-2.1 |
| krb5-libs | 1.6.1-2.1 |
| less | 394-9 |
| libattr | 2.4.32-2 |
| libattr-devel | 2.4.32-2 |
| libcap | 1.10-29 |
| libcap-devel | 1.10-29 |
| libpng | 1.2.16-1 |
| libpng-devel | 1.2.16-1 |
| libsysfs | 2.1.0-1 |
| libsysfs-devel | 2.1.0-1 |
| libtermcap | 2.0.8-46.1 |
| libtermcap-devel | 2.0.8-46.1 |
| libtirpc | 0.1.7-7_2 |
| libtirpc-devel | 0.1.7-7_2 |
| libtool | 1.5.22-11 |
| libtool-ltdl | 1.5.22-11 |
| libtool-ltdl-devel | 1.5.22-11 |
| libusb | 0.1.12-7 |
| libusb-devel | 0.1.12-7 |
| libuser | 0.56.2-1 |
| libuser-devel | 0.56.2-1 |
| libxml2 | 2.6.29-1 |
| logrotate | 3.7.5-3.1 |
| lrzsz | 0.12.20-22.1 |
| lsof | 4.78-5 |
| ltp | 20080131-eldk2 |
| lvm2 | 2.02.24-1 |
| m4 | 1.4.8-2 |
| mailcap | 2.1.23-1 |
| make | 3.81-6 |
| MAKEDEV | 3.23-1.2 |
| man | 1.6e-3 |
| mdadm | 2.6.2-4 |
| microwindows | 0.91-2 |
| microwindows-fonts | 0.91-1 |
| mingetty | 1.07-5.2.2 |
| mktemp | 1.5-25 |
| module-init-tools | 3.3-0.pre11.1.0 |

| | |
|----------------------|--------------|
| | 1.0.1-2 |
| ncompress | 4.2.4-49 |
| ncurses | 5.6-17 |
| ncurses-devel | 5.6-17 |
| net-snmp | 5.4-14 |
| net-snmp-devel | 5.4-14 |
| net-snmp-libs | 5.4-14 |
| net-snmp-utils | 5.4-14 |
| net-tools | 1.60-82 |
| newt | 0.52.6-30 |
| newt-devel | 0.52.6-30 |
| nfs-utils | 1.1.0-1 |
| ntp | 4.2.4p2-1 |
| open-iscsi | 2.0-865.15 |
| openldap | 2.3.34-3 |
| openldap-devel | 2.3.34-3 |
| openssl | 0.9.8b-12_2 |
| openssl-devel | 0.9.8b-12_2 |
| oprofile | 0.9.2-8_2 |
| pam | 0.99.7.1-5.1 |
| pam-devel | 0.99.7.1-5.1 |
| passwd | 0.74-3 |
| patch | 2.5.4-29.2.2 |
| pciutils | 2.2.4-3_2 |
| pciutils-devel | 2.2.4-3_2 |
| pcmciautils | 014-9_2 |
| pcre | 7.0-2 |
| pcsc-lite | 1.3.3-1.0 |
| pcsc-lite-devel | 1.3.3-1.0 |
| pcsc-lite-libs | 1.3.3-1.0 |
| perl | 5.8.8-18_2 |
| perl-libs | 5.8.8-18_2 |
| popt | 1.12-1 |
| portmap | 4.0-65_2 |
| postgresql | 8.2.4-1_2 |
| postgresql-devel | 8.2.4-1_2 |
| postgresql-libs | 8.2.4-1_2 |
| ppp | 2.4.4-7 |
| procps | 3.2.7-14 |
| psmisc | 22.3-2 |
| python | 2.5.1-1 |
| rdate | 1.4-6 |
| readline | 5.2-4 |
| readline-devel | 5.2-4 |
| routed | 0.17-12_1 |
| rpcbind | 0.1.4-6 |
| rpm | 4.4.2-46_2 |
| rpm-build | 4.4.2-46_2 |
| rpm-devel | 4.4.2-46_2 |
| rpm-libs | 4.4.2-46_2 |
| rsh | 0.17-40 |
| rsh-server | 0.17-40 |
| screen | 4.0.3-50 |
| sed | 4.1.5-7 |
| SELF | 1.0-13 |
| setup | 2.6.4-1_2 |
| shadow-utils | 4.0.18.1-15 |
| slang | 2.0.7-17 |
| slang-devel | 2.0.7-17 |
| smartmontools | 5.38-2 |
| strace | 4.5.15-1 |
| sysfsutils | 2.1.0-1 |

| | |
|--------------------|------------------|
| | 1.4.2-9 |
| sysvinit | 2.86-17 |
| tar | 1.15.1-26 |
| tcp_wrappers | 7.6-48 |
| tcp_wrappers-devel | 7.6-48 |
| tcp_wrappers-libs | 7.6-48 |
| telnet | 0.17-38 |
| telnet-server | 0.17-38 |
| termcap | 5.5-1.20060701.1 |
| tftp | 0.42-4 |
| tftp-server | 0.42-4 |
| thttpd | 2.25b-13 |
| time | 1.7-29 |
| u-boot | 1.3.1-1 |
| udev | 106-4.1 |
| unixODBC | 2.2.12-2 |
| unzip | 5.52-4 |
| util-linux | 2.13-0.52_2 |
| vim-common | 7.1.12-1 |
| vim-minimal | 7.1.12-1 |
| vixie-cron | 4.1-82 |
| vsftpd | 2.0.5-16_2 |
| which | 2.16-8 |
| wireless-tools | 28-4 |
| wpa_supplicant | 0.5.7-3 |
| wu-ftpd | 2.6.2-1 |
| xdd | 65.013007-1 |
| xenomai | 2.4.2-1 |
| xinetd | 2.3.14-12 |
| zip | 2.31-3 |
| zlib | 1.2.3-10 |
| zlib-devel | 1.2.3-10 |

 Note 1: Not all packages will be installed automatically; for example the **boa** and **thttpd** web servers are mutually exclusive - you will have to remove one package before you can (manually) install the other one.

 Note 2: The **crosstool 0.43** target package provides the following packages: **glibc 2.6**, **glibc-common 2.6**, **glibc-devel 2.6**, **libstdc++ 4.2.2**, **libgccj 4.2.2**, **libgccj-devel 4.2.2** and **libstdc++-devel 4.2.2**. For more information about the **crosstool** package please refer to <http://kegel.com/crosstool>

 Note 3: The Xenomai and gcc-java packages are unavailable in ARM [ELDK](#) version.

3.10. Rebuilding the [ELDK](#) from Scratch

In this section, you will find instructions on how to build the [ELDK](#) from scratch, using the pristine package sources available on the Internet, and patches, spec files, and build scripts provided on the [ELDK](#) source CD-ROM.

3.10.1. [ELDK](#) Build Process Overview

The [ELDK](#) uses the Fedora 7 Linux distribution as source code reference. Any modifications to Fedora's sources the [ELDK](#) has introduced are in the form of patches applied by the RPM tool while building the packages. Also, the [ELDK](#) uses modified spec files for its RPM packages. So, the sources of almost every [ELDK](#) package consist of the following parts:

- Fedora pristine sources **or**
- [ELDK](#) source tarball,
- [ELDK](#) patches,
- [ELDK](#) spec file.

The Fedora pristine sources may be obtained from the Internet, see <http://download.fedora.redhat.com/pub/fedora/linux/core>.

The [ELDK](#) patches and spec files are available on the [ELDK](#) source CD-ROM and from the DENX GIT repositories. Also, for convenience, the pristine Fedora sources are available here, too.

Please use the following commands to check out a copy of one of the modules:

```
git-clone git://www.denx.de/git/eldk/module
```

The following [ELDK](#) modules are available:

| Module Name | Contents |
|-----------------------------|--------------------------------------|
| tarballs | Source tarballs |
| build | Build tools, patches, and spec files |
| SRPMS | Fedora 7 sources |

Then you may switch to a specific release of the [ELDK](#) using the "git-checkout" command; for example, to get the files for [ELDK](#) release 4.1, please do the following from the *module* directory:

```
git-checkout ELDK_4_2
```


It must be noted that some of the packages which are included in the [ELDK](#) are not included in Fedora. Examples of such packages are `appWeb`, `microwindows`, and `wu-ftp`. For these packages tarballs are provided in the DENX GIT repository.

To facilitate building of the [ELDK](#), a build infrastructure has been developed. The infrastructure is composed of the following components:

- `ELDK_BUILD` script
- `build.sh` script
- `cpkgs.lst` file
- `tpkgs.lst` file
- `SRPMS.lst` file
- `tarballs.lst` file

The `ELDK_BUILD` script is the main script of the [ELDK](#) build procedure. It is the tool that you would normally use to build the [ELDK](#) from scratch. In the simplest case, the script may be invoked without arguments, and it will perform all necessary steps to build the [ELDK](#) in a fully automated way. You may pass the following optional arguments to the `ELDK_BUILD` script:

```
-a <arch>      target architecture: "arm", "ppc" or "ppc64", defaults to "ppc".
-n <build_name> an identification string for the build. Defaults to the value based on the build architecture and current date, and has the following format: <arch>-YYYY-MM-DD
-v <version>    ELDK version string
-u            build the uClibc-based ELDK version (on the platforms and versions where this is available).
-p <builddir>  Optional build directory. By default, build will place the work files and results in the current directory.
```

 Warning: The [ELDK](#) build scripts rely on standard behaviour of the RPM tool. Make sure you don't use non-standard settings in your personal `~/rpmmacros` file that might cause conflicts.

`build.sh` is a supplementary script that is called by `ELDK_BUILD` to accomplish certain steps of the build. Refer to section [3.10.3. build.sh Usage](#) below for more details.

The `cpkgs.lst` and `tpkgs.lst` files are read by `build.sh` and must contain lines describing sub-steps of the `elddt` and `trg` build procedure steps. Essentially, the files contain the list of the ELDT and target packages to be included in the [ELDK](#). The `SRPMS.lst` file contains the list of the Fedora source RPM packages used during the [ELDK](#) build. The `tarballs.lst` file contains the list of source tarballs of the packages that are included in the [ELDK](#) but are not present in Fedora 7.

For the `ELDK_BUILD` script to work correctly, it must be invoked from a certain build environment created on the host system. The build environment can be either checked out from the DENX GIT repository (see section [3.10.2. Setting Up ELDK](#) Build Environment below for details) or copied from the [ELDK](#) build environment CD-ROM.

To be more specific, the following diagram outlines the build environment needed for correct operation of the `ELDK_BUILD` script:

```
<some_directory>/
    build/cross_rpms/<package_name>/SPECS/...
                                SOURCES/...
    target_rpms/<package_name>/SPECS/...
                                SOURCES/...

    install/install.c
    Makefile
    misc/ELDK_MAKEDEV
    ELDK_FIXOWNER
    README.html
    cpkgs.lst
    tpkgs.lst
    build.sh

    ELDK_BUILD

    SRPMS.lst

    tarballs.lst

    tarballs/....

    SRPMS/....
    SRPMS-updates/....
```

In subdirectories of the `cross_rpms` and `target_rpms` directories, the sources and RPM spec files of, respectively, the ELDT and target packages are stored. The `install` subdirectory contains the sources of the installation utility which will be built and placed in the root of the ISO image. `tarballs` directory contains the source tarballs of the packages that are included in the [ELDK](#) but are not present in Fedora 7.

The SRPMS and SRPMS-updates directories may contain the source RPM packages of Fedora 7. The `ELDK_BUILD` script looks for a package in the SRPMS directory and then, if the package is not found, in the SRPMS-updates directory. If some (or all) of the Fedora SRPMS needed for the build are missing in the directories, the `ELDK_BUILD` script will download the source RPMs automatically from the Internet.

The [ELDK](#) build environment CD-ROM provides a ready-to-use [ELDK](#) build environment. Please refer to section [3.10.2. Setting Up ELDK](#) Build Environment below for detailed instructions on setting up the build environment.

The `ELDK_BUILD` script examines the contents of the `ELDK_PREFIX` environment variable to determine the root directory of the [ELDK](#) build environment. If the variable is not set when the script is invoked, it is assumed that the root directory of the [ELDK](#) build environment is `/opt/eldk`. To build the [ELDK](#) in the example directory layout given above, you must set and export the `ELDK_PREFIX` variable `<some_directory>` prior to invoking `ELDK_BUILD`.

After all the build steps are complete, the following subdirectories are created in the [ELDK](#) build environment:

```
build/<build_name>/work/      - full ELDK environment
build/<build_name>/logs/      - build procedure log files
build/<build_name>/results/b_cdrom/ - binary cdrom tree, ready for mkisofs
    results/s_cdrom/         - source cdrom tree, ready for mkisofs
    results/d_cdrom/         - debuginfo cdrom tree, ready for mkisofs
```

On Linux hosts, the binary and source ISO images are created automatically by the `ELDK_BUILD` script and placed in the `results` directory. On Solaris hosts, creating the ISO images is a manual step. Use the contents of the `b_cdrom` and `s_cdrom` directories for the contents of the ISO images.

3.10.2. Setting Up [ELDK](#) Build Environment


For your convenience, the [ELDK](#) build environment CD-ROM provides full [ELDK](#) build environment. All you need to do is copy the contents of the CD-ROM to an empty directory on your host system. Assuming the [ELDK](#) build environment CD-ROM is mounted at `/mnt/cdrom`, and the empty directory where you want to create the build environment is named `/opt/eldk`, use the following commands to create the build environment:

```
bash$ cd /opt/eldk
bash$ cp -r /mnt/cdrom/* .
```

These commands will create the directory structure as described in section [3.10.1. ELDK](#) Build Process Overview above. All necessary scripts and [ELDK](#) specific source files will be placed in the `build` subdirectory, and the required tarballs can be found in the `tarballs` subdirectory. In the `SRPMS` subdirectory, you will find all the Fedora 7 SRPMS needed to build the [ELDK](#).

Alternatively, you can obtain the [ELDK](#) build environment from the DENX GIT repository. Two modules are provided for check out: `build` and `tarballs`. The first one contains the files for the `build` subdirectory in the build environment, and the second one contains source tarballs of the packages that are included in the [ELDK](#) but are not present in Fedora 7. To create the [ELDK](#) build environment from the DENX GIT repository, use the following commands (the example below assumes that the root directory of the build environment is `/opt/eldk`):

```
bash$ cd /opt/eldk
bash$ git-clone git://www.denx.de/git/eldk/build
bash$ git-clone git://www.denx.de/git/eldk/tarballs
bash$ git-clone git://www.denx.de/git/eldk/SRPMS
```

 **Note:** To allow to install the [ELDK](#) on as many as possible Linux distributions (including old systems), we use a Red Hat 7.3 host system for building. Also, Fedora Core 5 is known to work as a build environment. Other, especially more recent Linux distributions, will most likely have problems. We therefor provide a Red Hat 7.3 based root file system image than can run in some virtualization environment (like qemu etc.). Here is an application note with detailed instructions: http://www.denx.de/wiki/DULG/AN2009_02_EldkReleaseBuildEnvironment

3.10.3. build.sh Usage

If you wish to perform only a part of the [ELDK](#) build procedure, for instance to re-build or update a certain package, it may sometimes be convenient to invoke the `build.sh` script manually, without the aid of the `ELDK_BUILD` script. Please note, however, that this approach is in general discouraged.

The whole build procedure is logically divided into six steps, and the `build.sh` must be told which of the build steps to perform. The build steps are defined as follows:

- *rpm* - build RPM
- *eldt* - build ELDT packages
- *seldt* - save ELDT SRPM packages to create a source ISO image later on
- *trg* - build target packages
- *biso* - prepare the file tree to create the binary ISO image
- *siso* - prepare the file tree to create the source ISO image
- *diso* - prepare the file tree to create the debuginfo ISO image


Further, the *eldt* and *trg* build steps are divided into sub-steps, as defined in the *cpkgs.lst* and *tpkgs.lst*

files (see below for details). You may specify which sub-steps of the build step are to be performed.

The formal syntax for the usage of `build.sh` is as follows:

```
bash$ ./build.sh [-a <arch>] [-n <name>] [-p <prefix>] [-r <result>] \
                [-w <work>] <step_name> [<sub_step_number>]
```

| | |
|--------------------------------------|--|
| <code>-a <arch></code> | target architecture: "ppc", "ppc64", "arm" or "mips", defaults to "ppc". |
| <code>-n <build_name></code> | an identification string for the build. It is used as a name for some directories created during the build. You may use for example the current date as the build name. |
| <code>-p <prefix></code> | is the name of the directory that contains the build environment. Refer to build overview above for description of the build environment. |
| <code>-r <result></code> | is the name of the directory where the resulting RPMs and SRPMs created on this step will be placed. |
| <code>-w <work></code> | is the name of the directory where the build is performed. |
| <code><stepname></code> | is the name of the build step that is to be performed. Refer to the list of the build procedure steps above. |
| <code><sub_step_number></code> | is an optional parameter which identifies sub-steps of the step which are to be performed. This is useful when you want to re-build only some specific packages. The numbers are defined in the <i>cpkgs.lst</i> and <i>tpkgs.lst</i> files discussed below. You can specify a range of numbers here. For instance, "2 5" means do steps from 2 to 5, while simply "2" means do all steps starting at 2. |

 Please note that you must never use `build.sh` to build the [ELDK](#) from scratch. For `build.sh` to work correctly, the script must be invoked from the build environment after a successful build using the `ELDK_BUILD` script. A possible scenario of `build.sh` usage is such that you have a build environment with results of a build performed using the `ELDK_BUILD` script and want to re-build certain ELDT and target packages, for instance, because you have updated sources of a package or added a new package to the build.

When building the target packages (during the *trg* buildstep), `build.sh` examines the contents of the `TARGET_CPU_FAMILY_LIST` environment variable, which may contain a list indicating which target **CPU** variants the packages must be built for. Possible **CPU** variants are *arm*. For example, the command below rebuilds the target RPM listed in the *tpckgs.lst* file under the number of 47 (see section [3.10.4. Format of the cpkgs.lst and tpkgs.lst Files](#) for description of the *tpckgs.lst* and *cpkgs.lst* files), for the *arm CPU*:

```
bash$ TARGET_CPU_FAMILY_LIST="arm" \
> /opt/eldk/build.sh -a arm \
>     -n 2007-01-21 \
>     -p /opt/eldk/build/arm-2007-01-21 \
>     -r /opt/eldk/build/arm-2007-01-21/results \
>     -w /opt/eldk/build/arm-2007-01-21/work \
>     trg 47 47
```

Note: If you are going to invoke `build.sh` to re-build a package that has already been built in the build environment by the `ELDK_BUILD` script, then you must first manually uninstall the package from [ELDK](#) installation created by the build procedure under the *work* directory of the build environment.

Note: It is recommended that you use the `build.sh` script only at the final stage of adding/updating a package to the [ELDK](#). For debugging purposes, it is much more convenient and efficient to build both ELDT and target packages using a working [ELDK](#) installation, as described in the sections [3.8.2. Rebuilding Target Packages](#) and [3.8.3. Rebuilding ELDT Packages](#) above.

3.10.4. Format of the cpkgs.lst and tpkgs.lst Files

Each line of these files has the following format:

```
<sub_step_number> <package_name> <spec_file_name> \
    <binary_package_name> <package_version>
```

The [ELDK](#) source CD-ROM contains the *cpkgs.lst* and *tpkgs.lst* files used to build this version of the [ELDK](#) distribution. Use them as reference if you want to include any additional packages into the [ELDK](#), or remove unneeded packages.

To add a package to the [ELDK](#) you must add a line to either the *cpkgs.lst* file, if you are adding a ELDT package, or to the *tpkgs.lst* file, if it is a target package. Keep in mind that the relative positions of packages in the *cpkgs.lst* and *tpkgs.lst* files (the sub-step numbers) are very important. The build procedure builds the packages sequentially as defined in the **.lst* files and installs the packages in the "work" environment as they are built. This implies that if a package depends on other packages, those packages must be specified earlier (with smaller sub-step numbers) in the **.lst* files.

Note: For *cpkgs.lst*, the *package_version* may be replaced by the special keyword "RHAUX". Such packages are used as auxiliary when building [ELDK](#) 4.2 on non-Fedora hosts. These packages will be built and used during the build process, but will not be put into the [ELDK](#) 4.2 distribution ISO images.

- [4. System Setup](#)
 - [4.1. Serial Console Access](#)
 - [4.2. Configuring the "cu" command](#)
 - [4.3. Configuring the "kermit" command](#)
 - [4.4. Using the "minicom" program](#)
 - [4.5. Permission Denied Problems](#)
 - [4.6. Configuration of a TFTP Server](#)
 - [4.7. Configuration of a BOOTP / DHCP Server](#)
 - [4.8. Configuring a NFS Server](#)

4. System Setup

Some tools are needed to install and configure U-Boot and Linux on the target system. Also, especially during development, you will want to be able to interact with the target system. This

section describes how to configure your host system for this purpose.

4.1. Serial Console Access

To use U-Boot and Linux as a development system and to make full use of all their capabilities you will need access to a serial console port on your target system. Later, U-Boot and Linux can be configured to allow for automatic execution without any user interaction.

There are several ways to access the serial console port on your target system, such as using a terminal server, but the most common way is to attach it to a serial port on your host. Additionally, you will need a terminal emulation program on your host system, such as `cu` or `kermit`.

4.2. Configuring the "cu" command

The `cu` command is part of the UUCP package and can be used to act as a dial-in terminal. It can also do simple file transfers, which can be used in U-Boot for image download.

On [RedHat](#) systems you can check if the UUCP package is installed as follows:

```
$ rpm -q uucp
```

If necessary, install the UUCP package from your distribution media.

To configure `cu` for use with U-Boot and Linux please make sure that the following entries are present in the `uucp` configuration files; depending on your target configuration the serial port and/or the console baudrate may be different from the values used in this example: (`/dev/ttyS0`, 115200 bps, 8N1):

- `/etc/uucp/sys`:

```
#
# /dev/ttyS0 at 115200 bps:
#
system      S0@115200
port        serial0_115200
time        any
```

- `/etc/uucp/port`:

```
#
# /dev/ttyS0 at 115200 bps:
#
port        serial0_115200
type        direct
device      /dev/ttyS0
speed       115200
hardflow    false
```

You can then connect to the serial line using the command

```
$ cu S0@115200
Connected.
```

To disconnect, type the escape character '~' followed by '.' at the beginning of a line.

See also: `cu(1)`, `info uucp`.

4.3. Configuring the "kermit" command

The name `kermit` stands for a whole family of communications software for serial and network connections. The fact that it is available for most computers and operating systems makes it especially well suited for our purposes.

`kermit` executes the commands in its initialization file, `.kermrc`, in your home directory before it executes any other commands, so this can be easily used to customize its behaviour using appropriate initialization commands. The following settings are recommended for use with U-Boot and Linux:

- `~/kermrc`:

```
set line /dev/ttyS0
set speed 115200
set carrier-watch off
set handshake none
set flow-control none
robust
set file type bin
set file name lit
set rec pack 1000
set send pack 1000
set window 5
```

This example assumes that you use the first serial port of your host system (`/dev/ttyS0`) at a baudrate of 115200 to connect to the target's serial console port.

You can then connect to the serial line:

```
$ kermit -c
Connecting to /dev/ttyS0, speed 115200.
The escape character is Ctrl-\ (ASCII 28, FS)
Type the escape character followed by C to get back,
or followed by ? to see other options.
-----
```

☐ Due to licensing conditions you will often find two `kermit` packages in your GNU/Linux distribution. In this case you will want to install the `ckermmit` package. The `gkermit` package is only a command line tool implementing the `kermit` transfer protocol.

☐ If you cannot find `kermit` on the distribution media for your Linux host system, you can download it from the `kermit` project home page: <http://www.columbia.edu/kermit/>

4.4. Using the "minicom" program

`minicom` is another popular serial communication program. Unfortunately, many users have reported problems using it with U-Boot and Linux, especially when trying to use it for serial image download. Its use is therefore discouraged.

4.5. Permission Denied Problems

The terminal emulation program must have write access to the serial port and to any locking files that are used to prevent concurrent access from other applications. Depending on the used Linux distribution you may have to make sure that:

- the serial device belongs to the same group as the `cu` command, and that the permissions of `cu` have the *setgid* bit set
- the `kermit` belongs to the same group as `cu` and has the *setgid* bit set
- the `/var/lock` directory belongs to the same group as the `cu` command, and that the write permissions for the group are set

4.6. Configuration of a [TFTP](#) Server

The fastest way to use U-Boot to load a Linux kernel or an application image is file transfer over Ethernet. For this purpose, U-Boot implements the *TFTP* protocol (see the `tftpboot` command in U-Boot).

To enable [TFTP](#) support on your host system you must make sure that the [TFTP](#) daemon program `/usr/sbin/in.tftpd` is installed. On [RedHat](#) systems you can verify this by running:

```
$ rpm -q tftp-server
```

If necessary, install the [TFTP](#) daemon program from your distribution media.

Most Linux distributions disable the [TFTP](#) service by default. To enable it for example on [RedHat](#) systems, edit the file `/etc/xinetd.d/tftp` and remove the line

```
disable = yes
```

or change it into a comment line by putting a hash character in front of it:

```
# default: off
# description: The tftp server serves files using the trivial file transfer
#               protocol. The tftp protocol is often used to boot diskless
#               workstations, download configuration files to network-aware printers,
#               and to start the installation process for some operating systems.
service tftp
{
    socket_type        = dgram
    protocol           = udp
    wait               = yes
    user               = root
    server             = /usr/sbin/in.tftpd
    server_args        = -s /tftpboot
    #
    disable            = yes
    per_source         = 11
    cps                = 100 2
}
```

Also, make sure that the `/tftpboot` directory exists and is world-readable (permissions at least "dr-xr-xr-x").

4.7. Configuration of a [BOOTP](#) / [DHCP](#) Server

BOOTP resp. **DHCP** can be used to automatically pass configuration information to the target. The only thing the target must "know" about itself is its own Ethernet hardware ([MAC](#)) address. The following command can be used to check if [DHCP](#) support is available on your host system:

```
$ rpm -q dhcp
```

If necessary, install the [DHCP](#) package from your distribution media.

Then you have to create the [DHCP](#) configuration file `/etc/dhcpd.conf` that matches your network setup. The following example gives you an idea what to do:

```
subnet 192.168.0.0 netmask 255.255.0.0 {
    option routers          192.168.1.1;
    option subnet-mask      255.255.0.0;

    option domain-name      "local.net";
    option domain-name-servers ns.local.net;

    host trgt {
        hardware ethernet    00:30:BF:01:02:D0;
        fixed-address        192.168.20.38;
        option root-path     "/opt/eldk-5.0/armv5te/rootfs";
        option host-name     "enbw_cmc";
        next-server          192.168.1.1;
        filename             "/tftpboot/duts/enbw_cmc/uImage";
    }
}
```

With this configuration, the [DHCP](#) server will reply to a request from the target with the ethernet address `00:30:BF:01:02:D0` with the following information:

- The target is located in the subnet `192.168.0.0` which uses the netmask `255.255.0.0`.
- The target has the hostname `enbw_cmc` and the IP address `192.168.20.38`.
- The host with the IP address `192.168.1.1` will provide the boot image for the target and provide NFS server function in cases when the target mounts its root filesystem over NFS.
 - ☐ The host listed with the `next-server` option can be different from the host that is running the [DHCP](#) server.
- The host provides the file `/tftpboot/duts/enbw_cmc/uImage` as boot image for the target.
- The target can mount the directory `/opt/eldk-5.0/armv5te/rootfs` on the NFS server as root filesystem.

4.8. Configuring a NFS Server

For a development environment it is very convenient when the host and the target can share the same files over the network. The easiest way for such a setup is when the host provides NFS server functionality and exports a directory that can be mounted from the target as the root filesystem.

Assuming NFS server functionality is already provided by your host, the only configuration that needs to be added is an entry for your target root directory to your `/etc/exports` file, for instance like this:

```
/opt/eldk-5.0/armv5te/rootfs    192.168.0.0/255.255.0.0(rw,no_root_squash,sync)
```

This line exports the `/opt/eldk-5.0/armv5te/rootfs` directory with read and write permissions to all hosts on the `192.168.0.0` subnet.

After modifying the `/etc/exports` file you must make sure the NFS system is notified about the change, for instance by issuing the command:

```
# /sbin/service nfs restart
```

- [5. Das U-Boot](#)
 - [5.1. Current Versions](#)
 - [5.2. Unpacking the Source Code](#)

- [5.3. Configuration](#)
- [5.4. Installation](#)
 - [5.4.1. Before You Begin](#)
 - [5.4.1.1. Installation Requirements](#)
 - [5.4.1.2. Board Identification Data](#)
 - [5.4.2. Installation Using a BDM/JTAG Debugger](#)
 - [5.4.3. Installation using U-Boot](#)
- [5.5. Tool Installation](#)
- [5.6. Initialization](#)
- [5.7. Initial Steps](#)
- [5.8. The First Power-On](#)
- [5.9. U-Boot Command Line Interface](#)
 - [5.9.1. Information Commands](#)
 - [5.9.1.1. bdfinfo - print Board Info structure](#)
 - [5.9.1.2. coninfo - print console devices and informations](#)
 - [5.9.1.3. flinfo - print FLASH memory information](#)
 - [5.9.1.4. iminfo - print header information for application image](#)
 - [5.9.1.5. help - print online help](#)
 - [5.9.2. Memory Commands](#)
 - [5.9.2.1. base - print or set address offset](#)
 - [5.9.2.2. crc32 - checksum calculation](#)
 - [5.9.2.3. cmp - memory compare](#)
 - [5.9.2.4. cp - memory copy](#)
 - [5.9.2.5. md - memory display](#)
 - [5.9.2.6. mm - memory modify \(auto-incrementing\)](#)
 - [5.9.2.7. mtest - simple RAM test](#)
 - [5.9.2.8. mw - memory write \(fill\)](#)
 - [5.9.2.9. nm - memory modify \(constant address\)](#)
 - [5.9.2.10. loop - infinite loop on address range](#)
 - [5.9.3. Flash Memory Commands](#)
 - [5.9.3.1. cp - memory copy](#)
 - [5.9.3.2. flinfo - print FLASH memory information](#)
 - [5.9.3.3. erase - erase FLASH memory](#)
 - [5.9.3.4. protect - enable or disable FLASH write protection](#)
 - [5.9.3.5. mtdparts - define a Linux compatible MTD partition scheme](#)
 - [5.9.3.6. UBI Usage in U-Boot](#)
 - [5.9.4. Execution Control Commands](#)
 - [5.9.4.1. source - run script from memory](#)
 - [5.9.4.2. bootm - boot application image from memory](#)
 - [5.9.4.3. go - start application at address 'addr'](#)
 - [5.9.5. Download Commands](#)
 - [5.9.5.1. bootp - boot image via network using BOOTP/TFTP protocol](#)
 - [5.9.5.2. dhcp - invoke DHCP client to obtain IP/boot params](#)
 - [5.9.5.3. loadb - load binary file over serial line \(kermit mode\)](#)
 - [5.9.5.4. loads - load S-Record file over serial line](#)
 - [5.9.5.5. rarpboot - boot image via network using RARP/TFTP protocol](#)
 - [5.9.5.6. tftpboot - boot image via network using TFTP protocol](#)
 - [5.9.6. Environment Variables Commands](#)
 - [5.9.6.1. printenv - print environment variables](#)
 - [5.9.6.2. saveenv - save environment variables to persistent storage](#)
 - [5.9.6.3. setenv - set environment variables](#)
 - [5.9.6.4. run - run commands in an environment variable](#)
 - [5.9.6.5. bootd - boot default, i.e., run 'bootcmd'](#)
 - [5.9.7. Flattened Device Tree support](#)
 - [5.9.7.1. fdt addr - select FDT to work on](#)
 - [5.9.7.2. fdt list - print one level](#)
 - [5.9.7.3. fdt print - recursive print](#)
 - [5.9.7.4. fdt mknode - create new nodes](#)
 - [5.9.7.5. fdt set - set node properties](#)
 - [5.9.7.6. fdt rm - remove nodes or properties](#)
 - [5.9.7.7. fdt move - move FDT blob to new address](#)
 - [5.9.7.8. fdt chosen - fixup dynamic info](#)
 - [5.9.8. Special Commands](#)
 - [5.9.8.1. i2c - I2C sub-system](#)
 - [5.9.9. Storage devices](#)
 - [5.9.9.1. MMC devices](#)
 - [5.9.9.2. NAND devices](#)
 - [5.9.9.2.1. nand bad - show bad block information](#)
 - [5.9.9.2.2. nand erase - erase region](#)
 - [5.9.9.2.3. nand write - write to NAND device](#)
 - [5.9.9.2.4. nand read - read from NAND device](#)
 - [5.9.10. Miscellaneous Commands](#)
 - [5.9.10.1. date - get/set/reset date & time](#)
 - [5.9.10.2. echo - echo args to console](#)
 - [5.9.10.3. reset - Perform RESET of the CPU](#)
 - [5.9.10.4. sleep - delay execution for some time](#)
 - [5.9.10.5. version - print monitor version](#)
 - [5.9.10.6. ? - alias for 'help'](#)
- [5.10. U-Boot Environment Variables](#)
- [5.11. U-Boot Scripting Capabilities](#)
- [5.12. U-Boot Standalone Applications](#)
 - [5.12.1. "Hello World" Demo](#)
 - [5.12.2. Timer Demo](#)
- [5.13. U-Boot Image Formats](#)
- [5.14. U-Boot Advanced Features](#)
 - [5.14.1. Boot Count Limit](#)

5. Das U-Boot

5.1. Current Versions

Das U-Boot (or just "U-Boot" for short) is Open Source Firmware for Embedded [Power Architecture®](#), ARM, MIPS, x86 and other processors. The U-Boot project is hosted by DENX, where you can also find the project home page: <http://www.denx.de/wiki/U-Boot/>

The current version of the U-Boot source code can be retrieved from the DENX "[git](#)" repository.

You can browse the "git" repositories at <http://git.denx.de/>

The trees can be accessed through the git, HTTP, and rsync protocols. For example you can use one of the following commands to create a local clone of one of the source trees:

```
git clone git://git.denx.de/u-boot.git u-boot/
git clone http://git.denx.de/u-boot.git u-boot/
git clone rsync://git.denx.de/u-boot.git u-boot/
```

For details please see [here](#).

Official releases of U-Boot are also available through [FTP](#). Compressed tar archives can be downloaded from the directory <ftp://ftp.denx.de/pub/u-boot/>.

5.2. Unpacking the Source Code

If you use GIT to get a copy of the U-Boot sources, here an example how you get the sources with git:

Note: Included topic DULGData_enbw_cmc.UBootGetSource does not exist yet

If you used [GIT](#) to get a copy of the U-Boot sources, then you can skip this next step since you already have an unpacked directory tree. If you downloaded a compressed tarball from the DENX [FTP](#) server, you can unpack it as follows:

```
$ cd /opt/eldk/usr/src
$ wget ftp://ftp.denx.de/pub/u-boot/2011.09.tar.bz2
$ rm -f u-boot
$ bunzip2 < 2011.09.tar.bz2 | tar xf -
$ ln -s 2011.09 u-boot
$ cd u-boot
```

5.3. Configuration

After changing to the directory with the U-Boot source code you should make sure that there are no build results from any previous configurations left:

```
$ make distclean
```

The following (model) command configures U-Boot for the enbw_cmc board:

```
$ make enbw_cmc_config
```

```
[hs@pollux]$
```

And finally we can compile the tools and U-Boot itself:

```
$ make all
```

By default the build is performed locally and the objects are saved in the source directory. One of the two methods can be used to change this behaviour and build U-Boot to some external directory:

1. Add O= to the make command line invocations:

```
make O=/tmp/build distclean
make O=/tmp/build enbw_cmc_config
make O=/tmp/build all
```

Note that if the 'O=output/dir' option is used then it must be used for all invocations of make.

2. Set environment variable BUILD_DIR to point to the desired location:

```
export BUILD_DIR=/tmp/build
make distclean
make enbw_cmc_config
make all
```

Note that the command line "O=" setting overrides the BUILD_DIR environment variable.

5.4. Installation

5.4.1. Before You Begin

5.4.1.1. Installation Requirements

The following section assumes that flash memory is used as the storage device for the firmware on your board. If this is not the case, the following instructions will not work - you will probably have to replace the storage device (probably ROM or EPROM) on such systems to install or update U-Boot.

5.4.1.2. Board Identification Data

All enbw_cmc boards use a serial number for identification purposes. Also, all boards have at least one ethernet ([MAC](#)) address assigned. You may lose your warranty on the board if this data gets lost. Before installing U-Boot or otherwise changing the software configuration of a board (like erasing some flash memory) you should make sure that you have all necessary information about such data.

5.4.2. Installation Using a BDM/JTAG Debugger

A fast and simple way to write new data to flash memory is via the use of a debugger or flash programmer with a [BDM](#) or [JTAG](#) interface. In cases where there is no running firmware at all (for instance on new hardware), this is usually the only way to install any software at all.

We use (and highly recommend) the BDI2000/BDI3000 by [Abatron](#).

Other [BDM](#) / [JTAG](#) debuggers may work too, but how to use them is beyond the scope of this document. Please see the documentation for the tool you want to use.

Before you can use the BDI2000 you have to configure it. A configuration file that can be used with enbw_cmc boards is included in section [13.2. BDI2000 Configuration file](#)

To install a new U-Boot image on your enbw_cmc board using a BDI2000, proceed as follows:

```
[hs@pollux bmk]$ telnet bdi4
Trying 192.168.10.4...
Connected to bdi4.
Escape character is '^'.
BDI Debugger for ARM
=====
```

```

MD      [<address>] [<count>]  display target memory as word (32bit)
MDH     [<address>] [<count>]  display target memory as half word (16bit)
[... ]
Core#0>
Core#0>
- CONFIG: loading configuration file passed
- CONFIG: loading register definition passed
- TARGET: processing reset request
- TARGET: BDI executes scan chain init string
- TARGET: Bypass check 0x00000001 => 0x00000002
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x07926001
- TARGET: All ICEBreaker access checks passed
- TARGET: BDI removes RESET
- TARGET: BDI waits for RESET inactive
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
CMC>
CMC>
CMC>
CMC>era
Erasing flash at 0x60000000
Erasing flash at 0x60004000
Erasing flash at 0x60006000
Erasing flash at 0x60008000
Erasing flash at 0x60010000
Erasing flash at 0x60020000
Erasing flash at 0x60030000
Erasing flash at 0x60040000
Erasing flash at 0x60050000
Erasing flash at 0x60060000
Erasing flash at 0x60070000
Erasing flash passed
CMC>prog 0x60000000 /tftpboot/duts/enbw_cmc/u-boot.bin BIN
Programming /tftpboot/duts/enbw_cmc/u-boot.bin , please wait ....
Programming flash passed
CMC>ti 0x60000004
      Core number      : 0
      Core state       : debug mode (ARM)
      Debug entry cause : Single Step
      Current PC       : 0x60000054
      Current CPSR     : 0x200000d3 (Supervisor)
CMC>g
CMC>

```

5.4.3. Installation using U-Boot

If U-Boot is already installed and running on your board, you can use these instructions to download another U-Boot image to replace the current one.

☐ Warning: Before you can install the new image, you have to erase the current one. If anything goes wrong your board will be dead. It is *strongly* recommended that:

- you have a backup of the old, working U-Boot image
- you know how to install an image on a virgin system

☐ Proceed as follows:

```

=> printenv load update
load=tftp ${u-boot_addr_r} ${u-boot}
update=protect off 0x60000000 +${filesize};erase 0x60000000 +${filesize};cp.b ${u-boot_addr_r} 0x60000000 ${filesize};protect on 0x60000000 +${filesize}
=> setenv u-boot /tftpboot/duts/enbw_cmc/u-boot.bin
=> run load update
Using DaVinci-EMAC device
TFTP from server 192.168.1.1; our IP address is 192.168.20.40
Filename '/tftpboot/duts/enbw_cmc/u-boot.bin'.
Load address: 0xc0000000
Loading: #####
done
Bytes transferred = 415480 (656f8 hex)
Un-Protected 10 sectors

..... done
Erased 10 sectors
Copy to Flash... done
Protected 10 sectors
=> reset
resetting ...

```

U-Boot 2011.09-03846-gccd5912 (Nov 23 2011 - 07:34:54)

```

Cores: ARM 456 MHz
DDR:    300 MHz
I2C:    ready
DRAM:   64 MiB
WARNING: Caches not enabled
Flash:  2 MiB
NAND:   128 MiB
MMC:    davinci: 0
In:     serial
Out:    serial
Err:    serial
Un-Protected 1 sectors
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
Protected 1 sectors
Net:    DaVinci-EMAC
key: 3
Hit any key to stop autoboot:  0
=> version

```

```

U-Boot 2011.09-03846-gccd5912 (Nov 23 2011 - 07:34:54)
arm-linux-gnueabi-gcc (GCC) 4.5.1
GNU ld (GNU Binutils) 2.21
=>

```

5.5. Tool Installation

U-Boot uses a special image format when loading the Linux kernel or ramdisk or other images. This image contains (among other things) information about the time of creation, operating system, compression type, image type, image name and CRC32 checksums.

The tool `mkimage` is used to create such images or to display the information they contain. When using the [ELDK](#), the `mkimage` command is already included with the other [ELDK](#) tools.

If you don't use the [ELDK](#) then you should install `mkimage` in some directory that is in your command search `PATH`, for instance:

```
$ cp tools/mkimage /usr/local/bin/
```

5.6. Initialization

To initialize the U-Boot firmware running on your `enbw_cmc` board, you have to connect a terminal to the board's serial console port.

The default configuration of the console port on the `enbw_cmc` board uses a baudrate of 115200/8N1 (115200 bps, 8 Bit per character, no parity, 1 stop bit, no handshake).

If you are running Linux on your host system we recommend either `kermit` or `cu` as terminal emulation programs. Do **not** use `minicom`, since this has caused problems for many users, especially for software download over the serial port.

For the configuration of your terminal program see section [4.1. Serial Console Access](#)

Make sure that both hardware and software flow control are **disabled**.

5.7. Initial Steps

In the default configuration, U-Boot operates in an interactive mode which provides a simple command line-oriented user interface using a serial console on port COM.2.

In the simplest case, this means that U-Boot shows a prompt (default: `=>`) when it is ready to receive user input. You then type a command, and press enter. U-Boot will try to run the required action(s), and then prompt for another command.

To see a list of the available U-Boot commands, you can type `help` (or simply `?`). This will print a list of all commands that are available in your current configuration. [Please note that U-Boot provides a *lot* of configuration options; not all options are available for all processors and boards, and some options might be simply not selected for your configuration.]

```
=>
=> he
```

With the command `help <command>` you can get additional information about most commands:

```
=> help tftpboot
tftpboot - boot image via network using TFTP protocol
```

```
Usage:
tftpboot [loadAddress] [[hostIPAddr:]bootfilename]
=> help setenv printenv
setenv - set environment variables
```

```
Usage:
setenv name value ...
- set environment variable 'name' to 'value ...'
setenv name
- delete environment variable 'name'
printenv - print environment variables
```

```
Usage:
printenv
- print values of all environment variables
printenv name ...
- print value of environment variable 'name'
=>
```


Most commands can be abbreviated as long as the string remains unambiguous:

```
=> help fli tftp
flinfo - print FLASH memory information
```

```
Usage:
flinfo
- print information for all FLASH memory banks
flinfo N
- print information for FLASH memory bank # N
tftpboot - boot image via network using TFTP protocol
```

```
Usage:
tftpboot [loadAddress] [[hostIPAddr:]bootfilename]
=>
```

5.8. The First Power-On

 Note: If you bought your `enbw_cmc` board with U-Boot already installed, you can skip this section since the manufacturer probably has already performed these steps.

Connect the port labeled COM.2 on your `enbw_cmc` board to the designated serial port of your host, start the terminal program, and connect the power supply of your `enbw_cmc` board. You should see messages like this:

```
=>
=> reset
resetting ...
```

```
U-Boot 2011.09-03846-gccd5912 (Nov 23 2011 - 07:34:54)
```

```
Cores: ARM 456 MHz
DDR: 300 MHz
I2C: ready
DRAM: 64 MiB
WARNING: Caches not enabled
Flash: 2 MiB
NAND: 128 MiB
MMC: davinci: 0
In: serial
Out: serial
Err: serial
Un-Protected 1 sectors
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
```

```
Writing to Flash... done
Protected 1 sectors
Protected 1 sectors
Net:    DaVinci-EMAC
key: 3
Hit any key to stop autoboot:  0
=>
```

You can interrupt the "Count-Down" by pressing any key. If you don't you will probably see some (harmless) error messages because the system has not been initialized yet.

☐ In some cases you may see a message

```
*** Warning - bad CRC, using default environment
```

This is harmless and will go away as soon as you have initialized and saved the *environment* variables.

At first you have to enter the serial number and the ethernet address of your board. Pay special attention here since these parameters are write protected and cannot be changed once saved (usually this is done by the manufacturer of the board). To enter the data you have to use the U-Boot command `setenv`, followed by the variable name and the data, all separated by white space (blank and/or TAB characters). Use the variable name `serial#` for the board ID and/or serial number, and `ethaddr` for the ethernet address, for instance:

```
=> setenv serial# DUTS
=> setenv ethaddr !!!!!FILL_THIS!!!!!!
=>
```

Use the `printenv` command to verify that you have entered the correct values:

```
=> printenv serial# ethaddr
serial#=DUTS
ethaddr=!!!!!!FILL_THIS!!!!!!
=>
```

Please double check that the printed values are correct! You will not be able to correct any errors later! If there is something wrong, reset the board and restart from the beginning; otherwise you can store the parameters permanently using the `saveenv` command:

```
=> saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
Protected 1 sectors
=>
```

5.9. U-Boot Command Line Interface

The following section describes the most important commands available in U-Boot. Please note that U-Boot is highly configurable, so not all of these commands may be available in the configuration of U-Boot installed on your hardware, or additional commands may exist. You can use the `help` command to print a list of all available commands for your configuration.

For most commands, you do not need to type in the full command name; instead it is sufficient to type a few characters. For instance, `help` can be abbreviated as `h`.

☐ The behaviour of some commands depends on the configuration of U-Boot and on the definition of some variables in your U-Boot environment.

☐ Almost all U-Boot commands expect numbers to be entered in hexadecimal input format. (Exception: for historical reasons, the `sleep` command takes it's argument in decimal input format.)

☐ Be careful not to use edit keys besides 'Backspace', as hidden characters in things like environment variables can be *very* difficult to find.

5.9.1. Information Commands

5.9.1.1. bdfinfo - print Board Info structure

```
=> help bdfinfo
bdfinfo - print Board Info structure
```

```
Usage:
bdfinfo
=>
```

The `bdfinfo` command (short: `bdi`) prints the information that U-Boot passes about the board such as memory addresses and sizes, clock frequencies, [MAC](#) address, etc. This information is mainly needed to be passed to the Linux kernel.

```
=> bdi
arch_number = 0x00000E01
boot_params = 0x00000000
DRAM_bank   = 0x00000000
-> start     = 0xC0000000
-> size      = 0x04000000
ethaddr      = 72:97:6f:6a:e4:e6
ip_addr      = 192.168.20.40
baudrate     = 115200 bps
TLB addr     = 0xC3FF0000
relocaddr    = 0xC3F52000
reloc off    = 0x63F52000
irq_sp       = 0xC3E3DF60
sp start     = 0xC3E3DF50
FB base      = 0x00000000
=>
```

5.9.1.2. coninfo - print console devices and informations

```
=> help conin
coninfo - print console devices and information
```

```
Usage:
coninfo
=>
```

The `coninfo` command (short: `conin`) displays information about the available console I/O devices.

```
=> conin
List of available devices:
logbuff  00000002 ..0
```



```
serial 80000003 SIO stdin stdout stderr
=>
```

The output contains the device name, flags, and the current usage. For example, the output

```
serial 80000003 SIO stdin stdout stderr
```

means that the `serial` device is a system device (flag `'S'`) which provides input (flag `'I'`) and output (flag `'O'`) functionality and is currently assigned to the 3 standard I/O streams `stdin`, `stdout` and `stderr`.

5.9.1.3. flinfo - print FLASH memory information

```
=> help flinfo
flinfo - print FLASH memory information

Usage:
flinfo
    - print information for all FLASH memory banks
flinfo N
    - print information for FLASH memory bank # N
=>
```

The command `flinfo` (short: `fli`) can be used to get information about the available flash memory (see Flash Memory Commands below).

```
=> fli
```

```
Bank # 1: CFI conformant flash (16 x 16) Size: 2 MB in 35 Sectors
AMD Standard command set, Manufacturer ID: 0x01, Device ID: 0x2249
Erase timeout: 8192 ms, write timeout: 1 ms

Sector Start Addresses:
60000000 RO 60004000 RO 60006000 RO 60008000 RO 60010000 RO
60020000 RO 60030000 RO 60040000 RO 60050000 RO 60060000 RO
60070000 60080000 RO 60090000 RO 600A0000 600B0000
600C0000 600D0000 600E0000 600F0000 60100000
60110000 60120000 60130000 60140000 60150000
60160000 60170000 60180000 60190000 601A0000
601B0000 601C0000 601D0000 601E0000 601F0000
=>
```

5.9.1.4. iminfo - print header information for application image

```
=> help iminfo
iminfo - print header information for application image

Usage:
iminfo addr [addr ...]
    - print header information for application image starting at
      address 'addr' in memory; this includes verification of the
      image contents (magic number, header and payload checksums)
=>
```

`iminfo` (short: `imi`) is used to print the header information for images like Linux kernels or ramdisks. It prints (among other information) the image name, type and size and verifies that the CRC32 checksums stored within the image are OK.

```
=> tftp ${ram_ws} ${bootfile}
Using DaVinci-EMAC device
TFTP from server 192.168.1.1; our IP address is 192.168.20.40
Filename '/tftpboot/duts/enbw_cmc/uImage'.
Load address: 0xc0300000
Loading: #####
          #####
          #####
done
Bytes transferred = 2285584 (22e010 hex)
=> imi ${ram_ws}

## Checking Image at c0300000 ...
Legacy image found
Image Name: Linux-3.1.0-00009-gd695872
Created: 2011-11-23 7:05:36 UTC
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 2285520 Bytes = 2.2 MiB
Load Address: c0008000
Entry Point: c0008000
Verifying Checksum ... OK
=>
```

☐ Like with many other commands, the exact operation of this command can be controlled by the settings of some U-Boot environment variables (here: the `verify` variable). See below for details.

5.9.1.5. help - print online help

```
=> help help
help - print command description/usage

Usage:
help
    - print brief description of all commands
help command ...
    - print detailed usage of 'command'
=>
```

The `help` command (short: `h` or `?`) prints online help. Without any arguments, it prints a list of all U-Boot commands that are available in your configuration of U-Boot. You can get detailed information for a specific command by typing its name as argument to the `help` command:

```
=> help protect
protect - enable or disable FLASH write protection

Usage:
protect on start end
    - protect FLASH from addr 'start' to addr 'end'
protect on start +len
    - protect FLASH from addr 'start' to end of sect w/addr 'start'+len'-1
protect on N:SF[-SL]
    - protect sectors SF-SL in FLASH bank # N
protect on bank N
    - protect FLASH bank # N
protect on <part-id>
```

```

- protect partition
protect on all
- protect all FLASH banks
protect off start end
- make FLASH from addr 'start' to addr 'end' writable
protect off start +len
- make FLASH from addr 'start' to end of sect w/addr 'start'+len'-1 wrtable
protect off N:SF[-SL]
- make sectors SF-SL writable in FLASH bank # N
protect off bank N
- make FLASH bank # N writable
protect off <part-id>
- make partition writable
protect off all
- make all FLASH banks writable
=>

```

5.9.2. Memory Commands

5.9.2.1. base - print or set address offset

```

=> help base
base - print or set address offset

Usage:
base
- print address offset for memory commands
base off
- set address offset for memory commands to 'off'
=>

```

You can use the `base` command (short: `ba`) to print or set a "base address" that is used as address offset for all memory commands; the default value of the base address is 0, so all addresses you enter are used unmodified. However, when you repeatedly have to access a certain memory region (like the internal memory of some embedded [Power Architecture®](#) processors) it can be very convenient to set the base address to the start of this area and then use only the offsets:

```

=> base
Base Address: 0x00000000
=> md 0 0xc
00000000: 00000000 00000000 00000000 00000000 .....
00000010: 00000000 00000000 00000000 00000000 .....
00000020: 00000000 00000000 00000000 00000000 .....
=> base 0xc0000000
Base Address: 0xc0000000
=> md 0 0xc
c0000000: ffffffff ffffffff ffffffff ffffffff .....
c0000010: ffffffff b ffffffff ffffffff 9 ffffffff 8 .....
c0000020: ffffffff 7 ffffffff 6 ffffffff 5 ffffffff 4 .....
=>

```

5.9.2.2. crc32 - checksum calculation

The `crc32` command (short: `crc`) can be used to calculate a CRC32 checksum over a range of memory:

```

=> crc 0xc0000004 0x3FC
CRC32 for c0000004 ... c00003ff ==> 3f285918
=>

```

When used with 3 arguments, the command stores the calculated checksum at the given address:

```

=> crc 0xc0000004 0x3FC 0xc0000000
CRC32 for c0000004 ... c00003ff ==> 3f285918
=> md 0xc0000000 4
c0000000: 3f285918 ffffffff ffffffff ffffffff .Y(????????
=>

```

As you can see, the CRC32 checksum was not only printed, but also stored at address 0x100000.

5.9.2.3. cmp - memory compare

```

=> help cmp
cmp - memory compare

Usage:
cmp [.b, .w, .l] addr1 addr2 count
=>

```

With the `cmp` command you can test if the contents of two memory areas is identical or not. The command will either test the whole area as specified by the 3rd (length) argument, or stop at the first difference.

```

=> cmp 0xc0000000 0xc0300000 0x400
Total of 1024 words were the same
=> md 0xc0000000 0xc
c0000000: 56190527 8779fae1 409bcc4e d0df2200 '. .V. .y.N. .@. " .
c0000010: 008000c0 008000c0 e4281fc4 00020205 .....(.....
c0000020: 756e694c 2e332d78 2d302e31 30303030 Linux-3.1.0-0000
=> md 0xc0300000 0xc
c0300000: 56190527 8779fae1 409bcc4e d0df2200 '. .V. .y.N. .@. " .
c0300010: 008000c0 008000c0 e4281fc4 00020205 .....(.....
c0300020: 756e694c 2e332d78 2d302e31 30303030 Linux-3.1.0-0000
=>

```

Like most memory commands the `cmp` can access the memory in different sizes: as 32 bit (long word), 16 bit (word) or 8 bit (byte) data. If invoked just as `cmp` the default size (32 bit or long words) is used; the same can be selected explicitly by typing `cmp.l` instead. If you want to access memory as 16 bit or word data, you can use the variant `cmp.w` instead; and to access memory as 8 bit or byte data please use `cmp.b`.

☐ Please note that the *count* argument specifies the number of data items to process, i. e. the number of long words or words or bytes to compare.

```

=> cmp.l 0xc0000000 0xc0300000 0x400
Total of 1024 words were the same
=> cmp.w 0xc0000000 0xc0300000 0x800
Total of 2048 halfwords were the same
=> cmp.b 0xc0000000 0xc0300000 0x1000
Total of 4096 bytes were the same
=>

```

5.9.2.4. cp - memory copy

```
=> help cp
cp - memory copy

Usage:
cp [.b, .w, .l] source target count
=>
```

The `cp` is used to copy memory areas.

```
=> cp 0xc0000000 0xc0300000 0x10000
=>
```

The `cp` understands the type extensions `.l`, `.w` and `.b`:

```
=> cp.l 0xc0300000 0xc0000000 0x10000
=> cp.w 0xc0300000 0xc0000000 0x20000
=> cp.b 0xc0300000 0xc0000000 0x40000
=>
```

5.9.2.5. md - memory display

```
=> help md
md - memory display
```

```
Usage:
md [.b, .w, .l] address [# of objects]
=>
```

The `md` can be used to display memory contents both as hexadecimal and ASCII data.

```
=> md 0xc0000000
c0000000: 56190527 8779fae1 409bcc4e d0df2200   '..V..y.N..@.'"..
c0000010: 008000c0 008000c0 e4281fc4 00020205   .....(.....
c0000020: 756e694c 2e332d78 2d302e31 30303030   Linux-3.1.0-0000
=>
c0000030: 64672d39 38353936 00003237 00000000   9-gd695872.....
c0000040: e1a00000 e1a00000 e1a00000 e1a00000   .....
c0000050: e1a00000 e1a00000 e1a00000 e1a00000   .....
=>
```

This command, too, can be used with the type extensions `.l`, `.w` and `.b`:

```
=>
=> md.w 0xc0000000
c0000000: 0527 5619 fae1 8779 cc4e 409b 2200 d0df   '..V..y.N..@.'"..
c0000010: 00c0 0080 00c0 0080      .....
=> d.b 0xc0000000
```

The last displayed memory address and the value of the count argument are remembered, so when you enter `md` again *without arguments* it will automatically continue at the next address, and use the same count again.

```
=>
=> md.b 0xc0000000 0x20
c0000000: 27 05 19 56 e1 fa 79 87 4e cc 9b 40 00 22 df d0   '..V..y.N..@.'"..
c0000010: c0 00 80 00 c0 00 80 00 c4 1f 28 e4 05 02 02 00   .....(.....
=> md.w 0xc0000000
c0000000: 0527 5619 fae1 8779 cc4e 409b 2200 d0df   '..V..y.N..@.'"..
c0000010: 00c0 0080 00c0 0080 1fc4 e428 0205 0002   .....(.....
c0000020: 694c 756e 2d78 2e33 2e31 2d30 3030 3030   Linux-3.1.0-0000
c0000030: 2d39 6467 3936 3835 3237 0000 0000 0000   9-gd695872.....
=> md 0xc0000000
```

5.9.2.6. mm - memory modify (auto-incrementing)

```
c0000080: 1a000001 e3a00017 ef123456 e10f2000   .....V4... ..
c0000090: e38220c0 e121f002 00000000 00000000   . ....!.....
c00000a0: e59f4764 eb00004d e28f0f43 e8901c4e   dG..M...C...N...
c00000b0: e590d01c e0400001 e0866000 e08aa000   .....@..`.....
c00000c0: e5da9000 e5dae001 e189940e e5dae002   .....
c00000d0: e5daa003 e189980e e1899c0a e08dd000   .....
c00000e0: e28da801 e28aa901 e154000a 2a000015   .....T....*
c00000f0: e084a009 e15a000f 9a000012 e28aab02   .....Z.....
=>
=> help m
```

The `mm` is a method to interactively modify memory contents. It will display the address and current contents and then prompt for user input. If you enter a legal hexadecimal number, this new value will be written to the address. Then the next address will be prompted. If you don't enter any value and just press ENTER, then the contents of this address will remain unchanged. The command stops as soon as you enter any data that is not a hex number (like `.`):

```
=>
=>
=> mm 0xc0000000
c0000000: 56190527 ? 0
c0000004: 8779fae1 ? 0xaabbccdd
c0000008: 409bcc4e ? 0x01234567
c000000c: d0df2200 ? .
=> md 0xc0000000 0x10
c0000000: 00000000 aabbccdd 01234567 d0df2200   .....gE#.'"..
c0000010: 008000c0 008000c0 e4281fc4 00020205   .....(.....
c0000020: 756e694c 2e332d78 2d302e31 30303030   Linux-3.1.0-0000
c0000030: 64672d39 38353936 00003237 00000000   9-gd695872.....
=>
```

Again this command can be used with the type extensions `.l`, `.w` and `.b`:

```
=>
=> mm.w 0xc0000000
c0000000: 0000 ? 0x0101
c0000002: 0000 ? 0x0202
c0000004: ccdd ? 0x4321
c0000006: aabb ? 0x8765
c0000008: 4567 ? .
=> md 0xc0000000 0x10
c0000000: 02020101 87654321 01234567 d0df2200   ....!Ce.gE#.'"..
c0000010: 008000c0 008000c0 e4281fc4 00020205   .....(.....
c0000020: 756e694c 2e332d78 2d302e31 30303030   Linux-3.1.0-0000
c0000030: 64672d39 38353936 00003237 00000000   9-gd695872.....
=>
```

```
=>
```

```
=> mm.b 0xc0000000
c0000000: 01 ? 0x48
c0000001: 01 ? 0x65
c0000002: 02 ? 0x6c
c0000003: 02 ? 0x6c
c0000004: 21 ? 0x6f
c0000005: 43 ? 0x20
c0000006: 65 ? 0x20
c0000007: 87 ? 0x20
c0000008: 67 ? .
=> md 0xc0000000 0x10
c0000000: 6c6c6548 2020206f 01234567 d0df2200 Hello gE#..."..
c0000010: 008000c0 008000c0 008000c0 e4281fc4 00020205 .....(.....
c0000020: 756e694c 2e332d78 2d302e31 30303030 Linux-3.1.0-0000
c0000030: 64672d39 38353936 00003237 00000000 9-gd695872.....
=>
```


5.9.2.7. mtest - simple RAM test


```
=> help mtest
mtest - simple RAM read/write test

Usage:
mtest [start [end [pattern [iterations]]]]
=>
```

The `mtest` provides a simple memory test.

```
=>
=> mtest 0xc0000000 0xc0100000
Pattern 00000000 Writing... Reading...Pattern FFFFFFFF Writing... Reading...Pattern 00000001 Writing... Reading...Pattern FFFFFFFE Writing... Readi
=>
```

 This tests writes to memory, thus modifying the memory contents. It will fail when applied to ROM or flash memory.

 This command may crash the system when the tested memory range includes areas that are needed for the operation of the U-Boot firmware (like exception vector code, or U-Boot's internal program code, stack or heap memory areas).

5.9.2.8. mw - memory write (fill)

```
=> help mw
mw - memory write (fill)

Usage:
mw [.b, .w, .l] address value [count]
=>
```

The `mw` is a way to initialize (fill) memory with some value. When called without a count argument, the value will be written only to the specified address. When used with a count, then a whole memory areas will be initialized with this value:

```
=> md 0xc0000000 0x10
c0000000: 0000000f 00000010 00000011 00000012 .....
c0000010: 00000013 00000014 00000015 00000016 .....
c0000020: 00000017 00000018 00000019 0000001a .....
c0000030: 0000001b 0000001c 0000001d 0000001e .....
=> mw 0xc0000000 0xaabbccdd
=> md 0xc0000000 0x10
c0000000: aabbccdd 00000010 00000011 00000012 .....
c0000010: 00000013 00000014 00000015 00000016 .....
c0000020: 00000017 00000018 00000019 0000001a .....
c0000030: 0000001b 0000001c 0000001d 0000001e .....
=> mw 0xc0000000 0 6
=> md 0xc0000000 0x10
c0000000: 00000000 00000000 00000000 00000000 .....
c0000010: 00000000 00000000 00000000 00000015 .....
c0000020: 00000017 00000018 00000019 0000001a .....
c0000030: 0000001b 0000001c 0000001d 0000001e .....
=>
```

This is another command that accepts the type extensions `.l`, `.w` and `.b` :

```
=> mw.w 0xc0000004 0x1155 6
=> md 0xc0000000 0x10
c0000000: 00000000 11551155 11551155 11551155 ....U.U.U.U.U.
c0000010: 00000000 00000000 00000015 00000016 .....
c0000020: 00000017 00000018 00000019 0000001a .....
c0000030: 0000001b 0000001c 0000001d 0000001e .....
=> mw.b 0xc0000007 0xff 7
=> md 0xc0000000 0x10
c0000000: 00000000 ff551155 ffffffff 1155ffff ....U.U.....U.
c0000010: 00000000 00000000 00000015 00000016 .....
c0000020: 00000017 00000018 00000019 0000001a .....
c0000030: 0000001b 0000001c 0000001d 0000001e .....
=>
```

5.9.2.9. nm - memory modify (constant address)

```
=> help nm
nm - memory modify (constant address)

Usage:
nm [.b, .w, .l] address
=>
```

The `nm` command (non-incrementing memory modify) can be used to interactively write different data several times to the same address. This can be useful for instance to access and modify device registers:

```
=>
=> nm.b 0xc0000000
c0000000: 00 ? 0x48
c0000000: 48 ? 0x65
c0000000: 65 ? 0x6c
c0000000: 6c ? 0x6c
c0000000: 6c ? 0x6f
c0000000: 6f ? .
=> md 0xc0000000 8
c0000000: 0000006f ff551155 ffffffff 1155ffff o...U.U.....U.
c0000010: 00000000 00000000 00000015 00000016 .....
=>
```

The `nm` command too accepts the type extensions `.l`, `.w` and `.b`.

5.9.2.10. loop - infinite loop on address range

```
=> help loop
loop - infinite loop on address range

Usage:
loop [.b, .w, .l] address number_of_objects
=>
```

The `loop` command reads in a tight loop from a range of memory. This is intended as a special form of a memory test, since this command tries to read the memory as fast as possible.

☐ This command will never terminate. There is no way to stop it but to reset the board!

```
=> loop 100000 8
```

5.9.3. Flash Memory Commands

5.9.3.1. cp - memory copy

```
=> help cp
cp - memory copy

Usage:
cp [.b, .w, .l] source target count
=>
```

The `cp` command "knows" about flash memory areas and will automatically invoke the necessary flash programming algorithm when the target area is in flash memory.

```
=> cp.b 0xc0000000 0x60100000 0x00050000
Copy to Flash... done
=>
```

☐ Writing to flash memory may fail when the target area has not been erased (see `erase` below), or if it is write-protected (see `protect` below).

```
=> cp.b 0xc0000000 0x60100000 0x00050000
Copy to Flash... Can't write to protected Flash sectors
=>
```

☐ Remember that the *count* argument specifies the number of items to copy. If you have a "length" instead (= byte count) you should use `cp.b` or you will have to calculate the correct number of items.

5.9.3.2. flinfo - print FLASH memory information

The command `flinfo` (short: `fli`) can be used to get information about the available flash memory. The number of flash banks is printed with information about the size and organization into flash "sectors" or *erase units*. For all sectors the start addresses are printed; write-protected sectors are marked as read-only (`RO`). Some configurations of U-Boot also mark empty sectors with an (`E`).

```
=> fli
```

```
Bank # 1: CFI conformant flash (16 x 16) Size: 2 MB in 35 Sectors
AMD Standard command set, Manufacturer ID: 0x01, Device ID: 0x2249
Erase timeout: 8192 ms, write timeout: 1 ms

Sector Start Addresses:
60000000 RO 60004000 RO 60006000 RO 60008000 RO 60010000 RO
60020000 RO 60030000 RO 60040000 RO 60050000 RO 60060000 RO
60070000 60080000 RO 60090000 RO 600A0000 600B0000
600C0000 600D0000 600E0000 600F0000 60100000
60110000 60120000 60130000 60140000 60150000
60160000 60170000 60180000 60190000 601A0000
601B0000 601C0000 601D0000 601E0000 601F0000
=>
```

5.9.3.3. erase - erase FLASH memory

```
=> help era
erase - erase FLASH memory

Usage:
erase start end
- erase FLASH from addr 'start' to addr 'end'
erase start +len
- erase FLASH from addr 'start' to the end of sect w/addr 'start'+ 'len'-1
erase N:SF[-SL]
- erase sectors SF-SL in FLASH bank # N
erase bank N
- erase FLASH bank # N
erase <part-id>
- erase partition
erase all
- erase all FLASH banks
=>
```

The `erase` command (short: `era`) is used to erase the contents of one or more sectors of the flash memory. It is one of the more complex commands; the `help` output shows this.

Probably the most frequent usage of this command is to pass the start and end addresses of the area to be erased:

```
=> era 0x60100000 0x6014ffff
..... done
Erased 5 sectors
=>
```

☐ Note that both the start and end addresses for this command must point **exactly** at the start resp. end addresses of flash sectors. Otherwise the command will not be executed.

Another way to select certain areas of the flash memory for the `erase` command uses the notation of flash *banks* and *sectors*:

Technically speaking, a *bank* is an area of memory implemented by one or more memory chips that are connected to the same *chip select* signal of the [CPU](#), and a flash *sector* or *erase unit* is the smallest area that can be erased in one operation.

For practical purposes it is sufficient to remember that with flash memory a bank is something that eventually may be erased as a whole in a single operation. This may be more efficient (faster)

than erasing the same area sector by sector.

[It depends on the actual type of flash chips used on the board if such a fast bank erase algorithm exists, and on the implementation of the flash device driver if it is actually used.]

In U-Boot, flash banks are numbered starting with 1, while flash sectors start with 0.

To erase the same flash area as specified using start and end addresses in the example above you could also type:

```
=> era 1:19-34
Erase Flash Sectors 19-34 in Bank # 1
..... done
=>
```

To erase a whole bank of flash memory you can use a command like this one:

```
=> era bank 1
Erase Flash Bank # 1 - Warning: 13 protected sectors will not be erased!
..... done
=>
```

☐ Note that a warning message is printed because some *write protected* sectors exist in this flash bank which were not erased.

With the command:

```
=> era all
Erase Flash Bank # 1 - Warning: 13 protected sectors will not be erased!
..... done
=>
```

the whole flash memory (except for the write-protected sectors) can be erased.

5.9.3.4. protect - enable or disable FLASH write protection

```
=> help protect
protect - enable or disable FLASH write protection
```

```
Usage:
protect on start end
- protect FLASH from addr 'start' to addr 'end'
protect on start +len
- protect FLASH from addr 'start' to end of sect w/addr 'start'+len'-1
protect on N:SF[-SL]
- protect sectors SF-SL in FLASH bank # N
protect on bank N
- protect FLASH bank # N
protect on <part-id>
- protect partition
protect on all
- protect all FLASH banks
protect off start end
- make FLASH from addr 'start' to addr 'end' writable
protect off start +len
- make FLASH from addr 'start' to end of sect w/addr 'start'+len'-1 wrtable
protect off N:SF[-SL]
- make sectors SF-SL writable in FLASH bank # N
protect off bank N
- make FLASH bank # N writable
protect off <part-id>
- make partition writable
protect off all
- make all FLASH banks writable
=>
```

The `protect` command is another complex one. It is used to set certain parts of the flash memory to read-only mode or to make them writable again. Flash memory that is "protected" (= read-only) cannot be written (with the `cp` command) or erased (with the `erase` command). Protected areas are marked as (RO) (for "read-only") in the output of the `flinfo` command:

```
=> flinfo
```

```
Bank # 1: CFI conformant flash (16 x 16) Size: 2 MB in 35 Sectors
AMD Standard command set, Manufacturer ID: 0x01, Device ID: 0x2249
Erase timeout: 8192 ms, write timeout: 1 ms
```

```
Sector Start Addresses:
60000000 RO 60004000 RO 60006000 RO 60008000 RO 60010000 RO
60020000 RO 60030000 RO 60040000 RO 60050000 RO 60060000 RO
60070000 RO 60080000 RO 60090000 RO 600A0000 600B0000
600C0000 600D0000 600E0000 600F0000 60100000
60110000 60120000 60130000 60140000 60150000
60160000 60170000 60180000 60190000 601A0000
601B0000 601C0000 601D0000 601E0000 601F0000
```

```
=> prot on 0x60100000 0x6014ffff
Protected 5 sectors
=> flinfo
```

```
Bank # 1: CFI conformant flash (16 x 16) Size: 2 MB in 35 Sectors
AMD Standard command set, Manufacturer ID: 0x01, Device ID: 0x2249
Erase timeout: 8192 ms, write timeout: 1 ms
```

```
Sector Start Addresses:
60000000 RO 60004000 RO 60006000 RO 60008000 RO 60010000 RO
60020000 RO 60030000 RO 60040000 RO 60050000 RO 60060000 RO
60070000 RO 60080000 RO 60090000 RO 600A0000 600B0000
600C0000 600D0000 600E0000 600F0000 60100000 RO
60110000 RO 60120000 RO 60130000 RO 60140000 RO 60150000
60160000 60170000 60180000 60190000 601A0000
601B0000 601C0000 601D0000 601E0000 601F0000
```

```
=> era 0x60100000 0x6014ffff
- Warning: 5 protected sectors will not be erased!
done
Erased 5 sectors
=> prot off 1:19-34
Un-Protect Flash Sectors 19-34 in Bank # 1
=> flinfo
```

```
Bank # 1: CFI conformant flash (16 x 16) Size: 2 MB in 35 Sectors
AMD Standard command set, Manufacturer ID: 0x01, Device ID: 0x2249
Erase timeout: 8192 ms, write timeout: 1 ms
```

```
Sector Start Addresses:
60000000 RO 60004000 RO 60006000 RO 60008000 RO 60010000 RO
60020000 RO 60030000 RO 60040000 RO 60050000 RO 60060000 RO
60070000 RO 60080000 RO 60090000 RO 600A0000 600B0000
```

```

600C0000      600D0000      600E0000      600F0000      60100000
60110000      60120000      60130000      60140000      60150000
60160000      60170000      60180000      60190000      601A0000
601B0000      601C0000      601D0000      601E0000      601F0000

```

```

=> era 1:19-34
Erase Flash Sectors 19-34 in Bank # 1
..... done
=>

```

□ The actual level of protection depends on the flash chips used on your hardware, and on the implementation of the flash device driver for this board. In most cases U-Boot provides just a simple software-protection, i. e. it prevents you from erasing or overwriting important stuff by accident (like the U-Boot code itself or U-Boot's environment variables), but it cannot prevent you from circumventing these restrictions - a nasty user who is loading and running his own flash driver code cannot and will not be stopped by this mechanism. Also, in most cases this protection is only effective while running U-Boot, i. e. any operating system will not know about "protected" flash areas and will happily erase these if requested to do so.

5.9.3.5. mtdparts - define a Linux compatible [MTD](#) partition scheme

U-Boot implements two different approaches to define a [MTD](#) partition scheme that can be shared easily with the linux kernel.

The first one is to define a single, static partition in your board config file, for example:

```

#undef CONFIG_JFFS2_CMDLINE
#define CONFIG_JFFS2_DEV      "nor0"
#define CONFIG_JFFS2_PART_SIZE 0xFFFFFFFF /* use whole device */
#define CONFIG_JFFS2_PART_SIZE 0x00100000 /* use 1MB */
#define CONFIG_JFFS2_PART_OFFSET 0x00000000

```

The second method uses the Linux kernel's `mtdparts` command line option and dynamic partitioning:

```

#define CONFIG_CMD_MTDPARTS
#define MTDIDS_DEFAULT      "nor1=zuma-1,nor2=zuma-2"
#define MTDPARTS_DEFAULT    "mtdparts=zuma-1:-(jffs2),zuma-2:-(user)"

```

Command line of course produces bigger images, and may be inappropriate for some targets, so by default it's off.

The `mtdparts` command offers an easy to use and powerful interface to define the contents of the environment variable of the same name that can be passed as boot argument to the Linux kernel:

```

=> help mtdparts
mtdparts - define flash/nand partitions

Usage:
mtdparts
  - list partition table
mtdparts delall
  - delete all partitions
mtdparts del part-id
  - delete partition (e.g. part-id = nand0,1)
mtdparts add <mtd-dev> <size>[@<offset>] [<name>] [<ro>]
  - add partition
mtdparts default
  - reset partition table to defaults

-----

this command uses three environment variables:

'partition' - keeps current partition identifier

partition := <part-id>
<part-id> := <dev-id>,part_num

'mtdids' - linux kernel mtd device id <-> u-boot device id mapping

mtdids=<idmap>[,<idmap>,...]

<idmap> := <dev-id>=<mtd-id>
<dev-id> := 'nand'|'nor'|'onenand'<dev-num>
<dev-num> := mtd device number, 0...
<mtd-id> := unique device tag used by linux kernel to find mtd device (mtd->name)

'mtdparts' - partition list

mtdparts=mtdparts=<mtd-def>[:<mtd-def>...]

<mtd-def> := <mtd-id>:<part-def>[,<part-def>...]
<mtd-id> := unique device tag used by linux kernel to find mtd device (mtd->name)
<part-def> := <size>[@<offset>][<name>][<ro-flag>]
<size> := standard linux memsize OR '-' to denote all remaining space
<offset> := partition start offset within the device
<name> := '(' NAME ') '
<ro-flag> := when set to 'ro' makes partition read-only (not used, passed to kernel)
=>

```

For example, on the `enbw_cmc` target system the `mtdparts` command display this information:

```

=> mtdparts
no such device nor0

device nand0 <davinci_nand.1>, # parts = 7
#: name      size      offset      mask_flags
0: uboot     0x000c0000    0x00000000    0
1: NVRAM     0x00040000    0x000c0000    0
2: k0        0x00300000    0x00100000    0
3: r0        0x01400000    0x00400000    0
4: k1        0x00300000    0x01800000    0
5: r1        0x01400000    0x01b00000    0
6: data      0x01100000    0x02f00000    0

active partition: nand0,0 - (uboot) 0x000c0000 @ 0x00000000

defaults:
mtdids : nor0=physmap-flash.0,nand0=davinci_nand.1
mtdparts: mtdparts=physmap-flash.0:512k(U-Boot),64k(env1),64k(env2),-(rest):davinci_nand.1:128k(dtb),3m(kernel),4m(rootfs),-(userfs)
=>

```

The partition table printed here obviously differs from the default value for the `mtdparts` variable printed in the last line. To verify this, we can check the current content of this variable:

```

=> print mtdparts
mtdparts=mtdparts=davinci_nand.1:0xc0000(uboot),0x40000(NVRAM),0x300000(k0),0x1400000(r0),0x300000(k1),0x1400000(r1),0x1100000(data)
=>

```


and we can see that it exactly matches the partition table printed above.

Then we delete the last 2 partitions ...

```
=> print mtdparts
mtdparts=mtdparts=davinci_nand.1:0xc0000(uboot),0x40000(NVRAM),0x300000(k0),0x1400000(r0),0x300000(k1),0x1400000(r1),0x1100000(data)
=> mtdparts del data
no such device nor0
=> mtdparts del r1
=> mtdparts
```

```
device nand0 <davinci_nand.1>, # parts = 5
#: name      size      offset      mask_flags
0: uboot      0x000c0000    0x00000000    0
1: NVRAM      0x00040000    0x000c0000    0
2: k0         0x00300000    0x00100000    0
3: r0         0x01400000    0x00400000    0
4: k1         0x00300000    0x01800000    0
```

```
active partition: nand0,0 - (uboot) 0x000c0000 @ 0x00000000
```

```
defaults:
mtdids : nor0=physmap-flash.0,nand0=davinci_nand.1
mtdparts: mtdparts=physmap-flash.0:512k(U-Boot),64k(env1),64k(env2),-(rest):davinci_nand.1:128k(dtb),3m(kernel),4m(rootfs),-(userfs)
=>
```

... and combine the free space into a single big partition:

```
=> print mtdparts
mtdparts=mtdparts=davinci_nand.1:768k(uboot),256k(NVRAM),3m(k0),20m(r0),3m(k1)
=> mtdparts add nand0 - newdata
=> mtdparts
```

```
device nand0 <davinci_nand.1>, # parts = 6
#: name      size      offset      mask_flags
0: uboot      0x000c0000    0x00000000    0
1: NVRAM      0x00040000    0x000c0000    0
2: k0         0x00300000    0x00100000    0
3: r0         0x01400000    0x00400000    0
4: k1         0x00300000    0x01800000    0
5: newdata    0x06500000    0x01b00000    0
```

```
active partition: nand0,0 - (uboot) 0x000c0000 @ 0x00000000
```

```
defaults:
mtdids : nor0=physmap-flash.0,nand0=davinci_nand.1
mtdparts: mtdparts=physmap-flash.0:512k(U-Boot),64k(env1),64k(env2),-(rest):davinci_nand.1:128k(dtb),3m(kernel),4m(rootfs),-(userfs)
=>
```

Now let's switch back to the default settings:

```
=> mtdparts default
=> mtdparts
```

```
device nor0 <physmap-flash.0>, # parts = 4
#: name      size      offset      mask_flags
0: U-Boot     0x00080000    0x00000000    0
1: env1       0x00010000    0x00080000    0
2: env2       0x00010000    0x00090000    0
3: rest       0x00160000    0x000a0000    0
```

```
device nand0 <davinci_nand.1>, # parts = 4
#: name      size      offset      mask_flags
0: dtb        0x00020000    0x00000000    0
1: kernel     0x00300000    0x00020000    0
2: rootfs     0x00400000    0x00320000    0
3: userfs     0x078e0000    0x00720000    0
```

```
active partition: nor0,0 - (U-Boot) 0x00080000 @ 0x00000000
```

```
defaults:
mtdids : nor0=physmap-flash.0,nand0=davinci_nand.1
mtdparts: mtdparts=physmap-flash.0:512k(U-Boot),64k(env1),64k(env2),-(rest):davinci_nand.1:128k(dtb),3m(kernel),4m(rootfs),-(userfs)
=>
```

5.9.3.6. UBI Usage in U-Boot

As in Linux, UBI access in U-Boot refers to [MTD](#) partitions, either through their partition number (like "nand0,7") or partition name (like "userfs"). So let's first check the partitions on the device:

```
=> mtdparts
```

```
device nor0 <physmap-flash.0>, # parts = 4
#: name      size      offset      mask_flags
0: U-Boot     0x00080000    0x00000000    0
1: env1       0x00010000    0x00080000    0
2: env2       0x00010000    0x00090000    0
3: rest       0x00160000    0x000a0000    0
```

```
device nand0 <davinci_nand.1>, # parts = 4
#: name      size      offset      mask_flags
0: dtb        0x00020000    0x00000000    0
1: kernel     0x00300000    0x00020000    0
2: rootfs     0x00400000    0x00320000    0
3: userfs     0x078e0000    0x00720000    0
```

```
active partition: nor0,0 - (U-Boot) 0x00080000 @ 0x00000000
```

```
defaults:
mtdids : nor0=physmap-flash.0,nand0=davinci_nand.1
mtdparts: mtdparts=physmap-flash.0:512k(U-Boot),64k(env1),64k(env2),-(rest):davinci_nand.1:128k(dtb),3m(kernel),4m(rootfs),-(userfs)
=>
```

U-Boot provides the following command line interface to UBI:

```
=> help ubi
ubi - ubi commands
```

```
Usage:
ubi part [part] [offset]
- Show or set current partition (with optional VID header offset)
ubi info [layout] - Display volume and ubi layout information
ubi create[vol] volume [size] [type] - create volume name with size
```

```

ubi write[vol] address volume size - Write volume from address with size
ubi read[vol] address volume [size] - Read volume to address with size
ubi remove[vol] volume - Remove volume
[Legends]
  volume: character name
  size: specified in bytes
  type: s[tatic] or d[ynamic] (default=dynamic)
=>

```

To make a UBI device available to U-Boot it needs to be attached. This is done using the "ubi part" command. If an UBI device exists on the specified [MTD](#) partition it will be attached, otherwise a new UBI device will be created.

*** WARNING *** "ubi part" will, without any warning, overwrite any existing data and create a new UBI device if you run it on a partition that does not contain an UBI device yet.

Let's attach the "userfs" partition to UBI:

```

=> ubi part userfs 2048
Creating 1 MTD partitions on "nand0":
0x000000720000-0x000008000000 : "mtd=3"
Bad block table found at page 65472, version 0x01
Bad block table found at page 65408, version 0x01
UBI: attaching mtd2 to ubi0
UBI: physical eraseblock size:   131072 bytes (128 KiB)
UBI: logical eraseblock size:   126976 bytes
UBI: smallest flash I/O unit:    2048
UBI: sub-page size:             512
UBI: VID header offset:         2048 (aligned 2048)
UBI: data offset:               4096
UBI: attached mtd2 to ubi0
UBI: MTD device name:           "mtd=3"
UBI: MTD device size:           120 MiB
UBI: number of good PEBs:       963
UBI: number of bad PEBs:        4
UBI: max. allowed volumes:      128
UBI: wear-leveling threshold:    4096
UBI: number of internal volumes: 1
UBI: number of user volumes:    1
UBI: available PEBs:            3
UBI: total number of reserved PEBs: 960
UBI: number of PEBs reserved for bad PEB handling: 9
UBI: max/mean erase counter: 5/2
=>

```

Now that the UBI device is attached, this device can be accessed using the following commands:

```

ubi info          Display volume and ubi layout information
ubi createvol     Create UBI volume on UBI device
ubi removevol     Remove UBI volume from UBI device
ubi read          Read data from UBI volume to memory
ubi write         Write data from memory to UBI volume

```

For example display volume and ubi layout information with ubi info:

```

=> ubi info l
UBI: volume information dump:
UBI: vol_id          0
UBI: reserved_pebs   947
UBI: alignment       1
UBI: data_pad        0
UBI: vol_type        3
UBI: name_len        6
UBI: usable_leb_size 126976
UBI: used_ebs        947
UBI: used_bytes      120246272
UBI: last_eb_bytes   126976
UBI: corrupted       0
UBI: upd_marker      0
UBI: name            userfs

UBI: volume information dump:
UBI: vol_id          2147479551
UBI: reserved_pebs   2
UBI: alignment       1
UBI: data_pad        0
UBI: vol_type        3
UBI: name_len        13
UBI: usable_leb_size 126976
UBI: used_ebs        2
UBI: used_bytes      253952
UBI: last_eb_bytes   2
UBI: corrupted       0
UBI: upd_marker      0
UBI: name            layout volume

=>

```

There is another set of commands to access UBIFS file systems:

```

ubifsmount - mount UBIFS volume
ubifsls    - list files in a directory
ubifsload  - load file from an UBIFS filesystem

```

First, we have to mount the UBIFS file system:

```

=> ubifsmount userfs
UBIFS: mounted UBI device 0, volume 0, name "userfs"
UBIFS: mounted read-only
UBIFS: file system size:   118849536 bytes (116064 KiB, 113 MiB, 936 LEBs)
UBIFS: journal size:      9023488 bytes (8812 KiB, 8 MiB, 72 LEBs)
UBIFS: media format:      w4/r0 (latest is w4/r0)
UBIFS: default compressor: LZO
UBIFS: reserved for root:  0 bytes (0 KiB)
=>

```

after successfully mounted we can access it:

```

=> ubifsls
      29 Wed Nov 23 08:39:41 2011  date_of_creation
      16 Wed Nov 23 08:39:40 2011  README
=> ubifsload 0xc0000000 date_of_creation
Loading file 'date_of_creation' to addr 0xc0000000 with size 29 (0x0000001d)...
Done
=> md 0xc0000000

```

```

c0000000: 3220694d 4e202e33 3020766f 39333a39      Mi 23. Nov 09:39
c0000010: 2031343a 20544543 31313032 ffffffff0a      :41 CET 2011....
c0000020: ffffffff7 ffffffff6 ffffffff5 ffffffff4      .....
c0000030: ffffffff3 ffffffff2 ffffffff1 ffffffff0      .....
c0000040: ffffffffef ffffffffef ffffffffed ffffffffec      .....
c0000050: ffffffffef ffffffffef ffffffffef ffffffffef      .....
c0000060: ffffffff7 ffffffff6 ffffffff5 ffffffff4      .....
c0000070: ffffffff3 ffffffff2 ffffffff1 ffffffff0      .....
c0000080: ffffffffdf ffffffffde ffffffffdd ffffffffdc      .....
c0000090: ffffffffdb ffffffffda ffffffff9 ffffffff8      .....
c00000a0: ffffffff7 ffffffff6 ffffffff5 ffffffff4      .....
c00000b0: ffffffff3 ffffffff2 ffffffff1 ffffffff0      .....
c00000c0: ffffffffcf ffffffffce ffffffffcd ffffffffcc      .....
c00000d0: ffffffffcb ffffffffca ffffffff9 ffffffff8      .....
c00000e0: ffffffff7 ffffffff6 ffffffff5 ffffffff4      .....
c00000f0: ffffffff3 ffffffff2 ffffffff1 ffffffff0      .....
=> ubifsload 0xc0000000 README
Loading file 'README' to addr 0xc0000000 with size 16 (0x00000010)...
Done
=> md 0xc0000000
c0000000: 7473754a 206e6120 6d617865 0a656c70      Just an example.
c0000010: 2031343a 20544543 31313032 ffffffff0a      :41 CET 2011....
c0000020: ffffffff7 ffffffff6 ffffffff5 ffffffff4      .....
c0000030: ffffffff3 ffffffff2 ffffffff1 ffffffff0      .....
c0000040: ffffffffef ffffffffef ffffffffed ffffffffec      .....
c0000050: ffffffffef ffffffffef ffffffffef ffffffffef      .....
c0000060: ffffffff7 ffffffff6 ffffffff5 ffffffff4      .....
c0000070: ffffffff3 ffffffff2 ffffffff1 ffffffff0      .....
c0000080: ffffffffdf ffffffffde ffffffffdd ffffffffdc      .....
c0000090: ffffffffdb ffffffffda ffffffff9 ffffffff8      .....
c00000a0: ffffffff7 ffffffff6 ffffffff5 ffffffff4      .....
c00000b0: ffffffff3 ffffffff2 ffffffff1 ffffffff0      .....
c00000c0: ffffffffcf ffffffffce ffffffffcd ffffffffcc      .....
c00000d0: ffffffffcb ffffffffca ffffffff9 ffffffff8      .....
c00000e0: ffffffff7 ffffffff6 ffffffff5 ffffffff4      .....
c00000f0: ffffffff3 ffffffff2 ffffffff1 ffffffff0      .....
=>

```

*** NOTE *** In U-Boot, there is no clean way to detach an UBI device; all you can do is to attach a different device - assuming there is another one; alternatively, you can give a non-existent partition name: this will give an error, but it will detach the previously attached UBI device:

```

=> ubi part foobar
incorrect device type in foobar
Partition foobar not found!
=>

```

Update of an UBI Volume using U-Boot:

```

=> ubi part userfs 2048
Unmounting UBIFS volume userfs!
UBI: mtd2 is detached from ubi0
Creating 1 MTD partitions on "nand0":
0x000000720000-0x000008000000 : "mtd=3"
UBI: attaching mtd2 to ubi0
UBI: physical eraseblock size: 131072 bytes (128 KiB)
UBI: logical eraseblock size: 126976 bytes
UBI: smallest flash I/O unit: 2048
UBI: sub-page size: 512
UBI: VID header offset: 2048 (aligned 2048)
UBI: data offset: 4096
UBI: attached mtd2 to ubi0
UBI: MTD device name: "mtd=3"
UBI: MTD device size: 120 MiB
UBI: number of good PEBs: 963
UBI: number of bad PEBs: 4
UBI: max. allowed volumes: 128
UBI: wear-leveling threshold: 4096
UBI: number of internal volumes: 1
UBI: number of user volumes: 1
UBI: available PEBs: 3
UBI: total number of reserved PEBs: 960
UBI: number of PEBs reserved for bad PEB handling: 9
UBI: max/mean erase counter: 5/2
=> ubifsmount userfs
UBIFS: mounted UBI device 0, volume 0, name "userfs"
UBIFS: mounted read-only
UBIFS: file system size: 118849536 bytes (116064 KiB, 113 MiB, 936 LEBs)
UBIFS: journal size: 9023488 bytes (8812 KiB, 8 MiB, 72 LEBs)
UBIFS: media format: w4/r0 (latest is w4/r0)
UBIFS: default compressor: LZO
UBIFS: reserved for root: 0 bytes (0 KiB)
=> ubifsls
      29 Wed Nov 23 08:39:41 2011 date_of_creation
      16 Wed Nov 23 08:39:40 2011 README
=> tftp 0xc0000000 enbw_cmc/image-userfs-new.ubifs
Using DaVinci-EMAC device
TFTP from server 192.168.1.1; our IP address is 192.168.20.40
Filename 'enbw_cmc/image-userfs-new.ubifs'.
Load address: 0xc0000000
Loading: #####
      #####
done
Bytes transferred = 1777664 (1b2000 hex)
=> ubi write 0xc0000000 userfs ${filesize}
1777664 bytes written to volume userfs
=> ubifsmount userfs
UBIFS: mounted UBI device 0, volume 0, name "userfs"
UBIFS: mounted read-only
UBIFS: file system size: 118849536 bytes (116064 KiB, 113 MiB, 936 LEBs)
UBIFS: journal size: 9023488 bytes (8812 KiB, 8 MiB, 72 LEBs)
UBIFS: media format: w4/r0 (latest is w4/r0)
UBIFS: default compressor: LZO
UBIFS: reserved for root: 0 bytes (0 KiB)
=> ubifsls
      29 Wed Nov 23 08:42:34 2011 date_of_creation
      29 Wed Nov 23 08:42:36 2011 date_of_modification
      16 Wed Nov 23 08:42:33 2011 README
=>

```

Useful definitions:

```

update_data=ubi part data;tftp c0000000 enbw_cmc/image-data.ubifs;ubi write c0000000 data ${filesize}
update_user=ubi part user;tftp c0000000 enbw_cmc/image-user.ubifs;ubi write c0000000 user ${filesize}

```

5.9.4. Execution Control Commands

5.9.4.1. source - run script from memory

```
=> help source
source - run script from memory

Usage:
source [addr]
    - run script starting at addr
    - A valid image header must be present

=>
```

With the `source` command you can run "shell" scripts under U-Boot: You create a U-Boot script image by simply writing the commands you want to run into a text file; then you will have to use the `mkimage` tool to convert this text file into a U-Boot image (using the image type `script`).

This image can be loaded like any other image file, and with `source` you can run the commands in such an image. For instance, the following text file:

```
[hs@pollux]$ echo
echo Network Configuration:
echo -----
echo Target:
printenv ipaddr hostname
echo
echo Server:
printenv serverip rootpath
echo
```

can be converted into a U-Boot script image using the `mkimage` command like this:

```
[hs@pollux]$ bash$ mkimage -A ppc -O linux -T script -C none -a 0 -e 0 \
> -n "autoscr example script" \
> -d ./testsystems/dulg/testcases/example.script /tftpboot/duts/enbw_cmc/example.scr
Image Name:      autoscr example script
Created:         Wed Nov 23 09:08:27 2011
Image Type:      PowerPC Linux Script (uncompressed)
Data Size:       157 Bytes = 0.15 kB = 0.00 MB
Load Address:    00000000
Entry Point:     00000000
Contents:
    Image 0: 149 Bytes = 0.15 kB = 0.00 MB
```

Now you can load and execute this script image in U-Boot:

```
=> tftp 0xc0000000 /tftpboot/duts/enbw_cmc/example.scr
Using DaVinci-EMAC device
TFTP from server 192.168.1.1: our IP address is 192.168.20.40
Filename '/tftpboot/duts/enbw_cmc/example.scr'.
Load address: 0xc0000000
Loading: #
done
Bytes transferred = 221 (dd hex)
=> imi
```

```
## Checking Image at c0000000 ...
Legacy image found
Image Name:      autoscr example script
Created:         2011-11-23   8:08:27 UTC
Image Type:      PowerPC Linux Script (uncompressed)
Data Size:       157 Bytes = 157 Bytes
Load Address:    00000000
Entry Point:     00000000
Contents:
    Image 0: 149 Bytes = 149 Bytes
Verifying Checksum ... OK
=> source 0xc0000000
## Executing script at c0000000
```

```
Network Configuration:
```

```
-----
Target:
ipaddr=192.168.20.40
hostname=enbw_cmc
```

```
Server:
serverip=192.168.1.1
rootpath=/opt/eldk-5.0/armv5te/rootfs
```

```
=>
```

5.9.4.2. bootm - boot application image from memory

```
=> help bootm
bootm - boot application image from memory
```

```
Usage:
bootm [addr [arg ...]]
    - boot application image stored in memory
      passing arguments 'arg ...'; when booting a Linux kernel,
      'arg' can be the address of an initrd image
      When booting a Linux kernel which requires a flat device-tree
      a third argument is required which is the address of the
      device-tree blob. To boot that kernel without an initrd image,
      use a '-' for the second argument. If you do not pass a third
      a bd_info struct will be passed instead
```

Sub-commands to do part of the bootm sequence. The sub-commands must be issued in the order below (it's ok to not issue all sub-commands):

```
start [addr [arg ...]]
loados  - load OS image
fdt     - relocate flat device tree
cmdline - OS specific command line processing/setup
bdt     - OS specific bd_t processing
prep    - OS specific prep before relocation or go
go      - start OS

=>
```

The `bootm` command is used to start operating system images. From the image header it gets information about the type of the operating system, the file compression method used (if any), the load and entry point addresses, etc. The command will then load the image to the required memory address, uncompressing it on the fly if necessary. Depending on the OS it will pass the

required boot arguments and start the OS at it's entry point.

The first argument to `bootm` is the memory address (in RAM, ROM or flash memory) where the image is stored, followed by optional arguments that depend on the OS.

For Linux, exactly one optional argument can be passed. If it is present, it is interpreted as the start address of a `initrd` ramdisk image (in RAM, ROM or flash memory). In this case the `bootm` command consists of three steps: first the Linux kernel image is uncompressed and copied into RAM, then the ramdisk image is loaded to RAM, and finally control is passed to the Linux kernel, passing information about the location and size of the ramdisk image.

To boot a Linux kernel image without a `initrd` ramdisk image, the following command can be used:

```
=> bootm ${kernel_addr}
```

If a ramdisk image shall be used, you can type:

```
=> bootm ${kernel_addr} ${ramdisk_addr}
```

Both examples of course imply that the variables used are set to correct addresses for a kernel and a `initrd` ramdisk image.

☐ When booting images that have been loaded to RAM (for instance using [TFTP](#) download) you have to be careful that the locations where the (compressed) images were stored do not overlap with the memory needed to load the uncompressed kernel. For instance, if you load a ramdisk image at a location in low memory, it may be overwritten when the Linux kernel gets loaded. This will cause undefined system crashes.

5.9.4.3. go - start application at address 'addr'

```
=> help go
go - start application at address 'addr'

Usage:
go addr [arg ...]
    - start application at address 'addr'
      passing 'arg' as arguments
=>
```

U-Boot has support for so-called *standalone applications*. These are programs that do not require the complex environment of an operating system to run. Instead they can be loaded and executed by U-Boot directly, utilizing U-Boot's service functions like console I/O or `malloc()` and `free()`.

This can be used to dynamically load and run special extensions to U-Boot like special hardware test routines or bootstrap code to load an OS image from some filesystem.

The `go` command is used to start such standalone applications. The optional arguments are passed to the application without modification. For more information see [5.12. U-Boot Standalone Applications](#).

5.9.5. Download Commands

5.9.5.1. bootp - boot image via network using [BOOTP](#)/TFTP protocol

```
=> help bootp
bootp - boot image via network using BOOTP/TFTP protocol

Usage:
bootp [loadAddress] [[hostIPAddr:]bootfilename]
=>
```

5.9.5.2. dhcp - invoke [DHCP](#) client to obtain IP/boot params

```
=> help dhcp
dhcp - boot image via network using DHCP/TFTP protocol

Usage:
dhcp [loadAddress] [[hostIPAddr:]bootfilename]
=>
```

5.9.5.3. loadb - load binary file over serial line (kermit mode)

```
=> help loadb
loadb - load binary file over serial line (kermit mode)

Usage:
loadb [ off ] [ baud ]
    - load binary file over serial line with offset 'off' and baudrate 'baud'
=>
```

With `kermit` you can download binary data via the serial line. Here we show how to download *uImage*, the Linux kernel image. Please make sure, that you have set up `kermit` as described in section [4.3. Configuring the "kermit" command](#) and then type:

```
=> loadb 100000
## Ready for binary (kermit) download ...
Ctrl-\c
(Back at denx.denx.de)
-----
C-Kermit 7.0.197, 8 Feb 2000, for Linux
Copyright (C) 1985, 2000,
    Trustees of Columbia University in the City of New York.
Type ? or HELP for help.
Kermit> send /bin /tftpboot/pImage
...
Kermit> connect
Connecting to /dev/ttyS0, speed 115200.
The escape character is Ctrl-\ (ASCII 28, FS)
Type the escape character followed by C to get back,
or followed by ? to see other options.
-----
= 550260 Bytes
## Start Addr      = 0x00100000
=> iminfo 100000

## Checking Image at 00100000 ...
Image Name:      Linux-2.4.4
Created:         2002-07-02 22:10:11 UTC
Image Type:      PowerPC Linux Kernel Image (gzip compressed)
Data Size:       550196 Bytes = 537 kB = 0 MB
Load Address:    00000000
Entry Point:     00000000
Verifying Checksum ... OK
```

5.9.5.4. loads - load S-Record file over serial line

```
=> help loads
loads - load S-Record file over serial line

Usage:
loads [ off ]
    - load S-Record file over serial line with offset 'off'
=>
```

5.9.5.5. rarpboot- boot image via network using RARP/TFTP protocol

```
=> help rarp
Unknown command 'rarp' - try 'help' without arguments for list of all known commands

=>
```

5.9.5.6. tftpbboot- boot image via network using [TFTP](#) protocol

```
=> help tftp
tftpbboot - boot image via network using TFTP protocol

Usage:
tftpbboot [loadAddress] [[hostIPAddr:]bootfilename]
=>
```

5.9.6. Environment Variables Commands

5.9.6.1. printenv- print environment variables

```
=> help printenv
printenv - print environment variables

Usage:
printenv
    - print values of all environment variables
printenv name ...
    - print value of environment variable 'name'
=>
```

The `printenv` command prints one, several or all variables of the U-Boot environment. When arguments are given, these are interpreted as the names of environment variables which will be printed with their values:

```
=> printenv ipaddr hostname netmask
ipaddr=192.168.20.40
hostname=enbw_cmc
netmask=255.255.0.0
=>
```

Without arguments, `printenv` prints all a list with all variables in the environment and their values, plus some statistics about the current usage and the total size of the memory available for the environment.

```
=> printenv
addcon=setenv bootargs ${bootargs} console=ttyS2,$(baudrate)n8
addip=setenv bootargs ${bootargs} ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}:${netdev}:off panic=1
addmttd=setenv bootargs ${bootargs} ${mtddparts}
bar=This is a new example.
baudrate=115200
bootcmd=run net_nfs
bootcount=3
bootdelay=3
bootfile=/tftpbboot/duts/enbw_cmc/uImage
cmp_addr_r=0xc0000000
cons_opts=console=tty0 console=ttyS0,$(baudrate)
ethact=Davinci-EMAC
ethaddr=72:97:6f:6a:e4:e6
fdt_addr_r=0xc0600000
fdt_file=/tftpbboot/duts/enbw_cmc/enbw_cmc.dtb
fileaddr=C0B00000
filesize=195377
flash_self=run load_nand ramargs addip addcon;bootm ${kernel_addr_r} ${ramdisk_addr_r} ${fdt_addr_r}
hostname=enbw_cmc
hwconfig=switch:lan=on,pwl=off
ipaddr=192.168.20.40
kernel_addr_r=0xc0700000
kernel_file=/tftpbboot/enbw_cmc/uImage-hs-20111121
key_cmd_0=echo key: 0
key_cmd_1=echo key: 1
key_cmd_2=echo key: 2
key_cmd_3=echo key: 3
key_magic_0=0
key_magic_1=1
key_magic_2=2
key_magic_3=3
load=tftp ${u-boot_addr_r} ${u-boot}
load_fdt=tftp ${fdt_addr_r} ${fdt_file}
load_kernel=tftp ${kernel_addr_r} ${kernel_file}
load_nand=run nand_id_ramdisk nand_id_kernel nand_id_fdt
logversion=2
machid=e01
magic_keys=0123
mem=65516k
mtdddevname=U-Boot
mtdddevnum=0
mtddids=nor0=physmap-flash.0,nand0=davinci_nand.1
mtddparts=mtddparts=physmap-flash.0:512k(U-Boot),64k(env1),64k(env2),-(rest);davinci_nand.1:128k(dtb),3m(kernel),4m(rootfs),-(userfs)
nand_erasesize=20000
nand_ld_fdt=nand read ${fdt_addr_r} 0 2000
nand_ld_kernel=nand read ${kernel_addr_r} 20000 300000
nand_ld_ramdisk=nand read ${ramdisk_addr_r} 320000 400000
nand_oobsize=40
nand_writesize=800
net_nfs=run load_fdt load_kernel; run nfsargs addip addcon addmttd;bootm ${kernel_addr_r} - ${fdt_addr_r}
netdev=eth0
netmask=255.255.0.0
nfsargs=setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath}
partition=nor0,0
preboot=echo key: 3
ramargs=setenv bootargs root=/dev/ram rw
ramdisk_addr_r=0xc0b00000
```

```

restartcount=83
rootpath=/opt/eldk-5.0/armv5te/rootfs
serverip=192.168.1.1
test=echo This is a test
test2=echo This is another Test
u-boot=/tftpboot/enbw_cmc/u-boot.bin-hs-20111117
u-boot_addr_r=c0000000
update=protect off 0x60000000 +${filesize};erase 0x60000000 +${filesize};cp.b ${u-boot_addr_r} 0x60000000 ${filesize};protect on 0x60000000 +${filesize}
ver=U-Boot 2011.09-03846-gccd5912 (Nov 23 2011 - 07:34:54)

Environment size: 2491/16379 bytes
=>

```

5.9.6.2. saveenv - save environment variables to persistent storage

```

=> help saveenv
saveenv - save environment variables to persistent storage

```

```

Usage:
saveenv
=>

```

All changes you make to the U-Boot environment are made in RAM only. They are lost as soon as you reboot the system. If you want to make your changes permanent you have to use the `saveenv` command to write a copy of the environment settings to persistent storage, from where they are automatically loaded during startup:

```

=> saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
Protected 1 sectors
=>

```

5.9.6.3. setenv - set environment variables

```

=> help setenv
setenv - set environment variables

Usage:
setenv name value ...
    - set environment variable 'name' to 'value ...'
setenv name
    - delete environment variable 'name'
=>

```

To modify the U-Boot environment you have to use the `setenv` command. When called with exactly one argument, it will delete any variable of that name from U-Boot's environment, if such a variable exists. Any storage occupied for such a variable will be automatically reclaimed:

```

=> setenv foo This is an example value.
=> printenv foo
foo=This is an example value.
=> setenv foo
=> printenv foo
## Error: "foo" not defined
=>

```

When called with more arguments, the first one will again be the name of the variable, and all following arguments will (concatenated by single space characters) form the value that gets stored for this variable. New variables will be automatically created, existing ones overwritten.

```

=> printenv bar
## Error: "bar" not defined
=> setenv bar This is a new example.
=> printenv bar
bar=This is a new example.
=>

```

Remember standard shell quoting rules when the value of a variable shall contain characters that have a special meaning to the command line parser (like the `$` character that is used for variable substitution or the semicolon which separates commands). Use the backslash (`\`) character to escape such special characters, or enclose the whole phrase in apostrophes (`'`). Use `"${name}"` for variable expansion (see [14.2.17. How the Command Line Parsing Works](#) for details).

```

=> setenv cons_opts 'console=tty0 console=ttyS0,{baudrate}'
=> printenv cons_opts
cons_opts=console=tty0 console=ttyS0,{baudrate}
=>

```

❏ There is no restriction on the characters that can be used in a variable name except the restrictions imposed by the command line parser (like using backslash for quoting, space and tab characters to separate arguments, or semicolon and newline to separate commands). Even strange input like `"=~/()+="` is a perfectly legal variable name in U-Boot.

❏ A common mistake is to write

```
setenv name=value
```

instead of

```
setenv name value
```

There will be no error message, which lets you believe everything went OK, but it didn't: instead of setting the variable *name* to the value *value* you tried to delete a variable with the name *name=value* - this is probably not what you intended! Always remember that name and value have to be separated by space and/or tab characters!

5.9.6.4. run - run commands in an environment variable

```

=> help run
run - run commands in an environment variable

Usage:
run var [...]
    - run the commands in the environment variable(s) 'var'
=>

```

You can use U-Boot environment variables to store commands and even sequences of commands. To execute such a command, you use the `run` command:

```

=> setenv test echo This is a test;printenv ipaddr;echo Done.
ipaddr=192.168.20.40

```



```

Done.
=> printenv test
test=echo This is a test
=> run test
This is a test
=>

```

You can call `run` with several variables as arguments, in which case these commands will be executed in sequence:

```

=> setenv test2 echo This is another Test;printenv hostname;echo Done.
hostname=enbw_cmc
Done.
=> printenv test test2
test=echo This is a test
test2=echo This is another Test
=> run test test2
This is a test
This is another Test
=>

```

☐ If a U-Boot variable contains several commands (separated by semicolon), and one of these commands fails when you "run" this variable, the remaining commands *will be executed anyway*.

☐ If you execute several variables with one call to `run`, any failing command will cause "run" to terminate, i. e. the remaining variables are *not* executed.

5.9.6.5. bootd - boot default, i.e., run 'bootcmd'

```

=> help bootd
bootd - boot default, i.e., run 'bootcmd'

```

```

Usage:
bootd
=>

```

The `bootd` (short: `boot`) executes the default boot command, i. e. what happens when you don't interrupt the initial countdown. This is a synonym for the `run bootcmd` command.

5.9.7. Flattened Device Tree support

U-Boot is capable of quite comprehensive handling of the flattened device tree blob, implemented by the `fdt` family of commands:

```

=> help fdt
fdt - flattened device tree utility commands

Usage:
fdt addr <addr> [<length>] - Set the fdt location to <addr>
fdt move <fdt> <newaddr> <length> - Copy the fdt to <addr> and make it active
fdt resize - Resize fdt to size + padding to 4k addr
fdt print <path> [<prop>] - Recursive print starting at <path>
fdt list <path> [<prop>] - Print one level starting at <path>
fdt set <path> <prop> [<val>] - Set <property> [to <val>]
fdt mknod <path> <node> - Create a new node after <path>
fdt rm <path> [<prop>] - Delete the node or <property>
fdt header - Display header info
fdt bootcpu <id> - Set boot cpuid
fdt memory <addr> <size> - Add/Update memory node
fdt rsvmem print - Show current mem reserves
fdt rsvmem add <addr> <size> - Add a mem reserve
fdt rsvmem delete <index> - Delete a mem reserves
fdt chosen [<start> <end>] - Add/update the /chosen branch in the tree
                             <start>/<end> - initrd start/end addr

NOTE: Dereference aliases by omitting the leading '/', e.g. fdt print ethernet0.
=>

```

5.9.7.1. fdt addr - select FDT to work on

First, the blob that is to be operated on should be stored in memory, and U-Boot has to be informed about its location by the `fdt addr` command. Once this command has been issued, all subsequent `fdt` handling commands will use the blob stored at the given address. This address can be changed later on by issuing `fdt addr` or `fdt move` command. Here's how to load the blob into memory and tell U-Boot its location:

```

=> print fdt_addr_r
fdt_addr_r=0xc0600000
=> print fdt_file
fdt_file=/tftpboot/duts/enbw_cmc/enbw_cmc.dtb
=> tftp ${fdt_addr_r} ${fdt_file}
Using DaVinci-EMAC device
TFTP from server 192.168.1.1; our IP address is 192.168.20.40
Filename '/tftpboot/duts/enbw_cmc/enbw_cmc.dtb'.
Load address: 0xc0600000
Loading: #
done
Bytes transferred = 5258 (148a hex)
=> fdt addr ${fdt_addr_r}
=>

```

5.9.7.2. fdt list - print one level

Having selected the device tree stored in the blob just loaded, we can inspect its contents. As an FDT usually is quite extensive, it is easier to get information about the structure by looking at selected levels rather than full hierarchies. `fdt list` allows us to do exactly this. Let's have a look at the hierarchy one level below the `cpus` node:

```

=> fdt list /memory
memory {
    device_type = "memory";
    reg = <0x0 0x0>;
};
=>

```

5.9.7.3. fdt print - recursive print

To print a complete subtree we use `fdt print`. In comparison to the previous example it is obvious that the whole subtree is printed:

```

=> fdt print /memory
memory {
    device_type = "memory";
    reg = <0x0 0x0>;
};
=>

```

5.9.7.4. fdt mknode - create new nodes

`fdt mknode` can be used to attach a new node to the tree. We will use the `fdt list` command to verify that the new node has been created and that it is empty:

```
=> fdt list /
/ {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    model = "EnBW CMC";
    compatible = "enbw,cmc";
    chosen {
    };
    aliases {
    };
    memory {
    };
    soc@1c00000 {
    };
    onchipram@8000000 {
    };
    emif@60000000 {
    };
};
=> fdt mknode / testnode
=> fdt list /
/ {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    model = "EnBW CMC";
    compatible = "enbw,cmc";
    testnode {
    };
    chosen {
    };
    aliases {
    };
    memory {
    };
    soc@1c00000 {
    };
    onchipram@8000000 {
    };
    emif@60000000 {
    };
};
=> fdt list /testnode
testnode {
};
=>
```

5.9.7.5. fdt set - set node properties

Now, let's create a property at the newly created node; again we'll use `fdt list` for verification:

```
=> fdt set /testnode testprop testvalue
=> fdt list /testnode
testnode {
    testprop = "testvalue";
};
=>
```

5.9.7.6. fdt rm - remove nodes or properties

The `fdt rm` command is used to remove nodes and properties. Let's delete the test property created in the previous paragraph and verify the results:

```
=> fdt rm /testnode testprop
=> fdt list /testnode
testnode {
};
=> fdt rm /testnode
=> fdt list /
/ {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    model = "EnBW CMC";
    compatible = "enbw,cmc";
    chosen {
    };
    aliases {
    };
    memory {
    };
    soc@1c00000 {
    };
    onchipram@8000000 {
    };
    emif@60000000 {
    };
};
=>
```

5.9.7.7. fdt move - move FDT blob to new address

To move the blob from one memory location to another we will use the `fdt move` command. Besides moving the blob, it makes the new address the "active" one - similar to `fdt addr`:

```
=> fdt move ${fdt_addr_r} 0xc0000000
=> fdt list /
/ {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    model = "EnBW CMC";
    compatible = "enbw,cmc";
    chosen {
    };
    aliases {
    };
    memory {
    };
    soc@1c00000 {
    };
    onchipram@8000000 {
    };
};
```

```

    };
    emif@60000000 {
    };
};
=> fdt mknod / foobar
=> fdt list /
/ {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    model = "EnBW CMC";
    compatible = "enbw,cmc";
    foobar {
    };
    chosen {
    };
    aliases {
    };
    memory {
    };
    soc@1c00000 {
    };
    onchipram@8000000 {
    };
    emif@60000000 {
    };
};
=> fdt addr ${fdt_addr_r}
=> fdt list /
/ {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    model = "EnBW CMC";
    compatible = "enbw,cmc";
    chosen {
    };
    aliases {
    };
    memory {
    };
    soc@1c00000 {
    };
    onchipram@8000000 {
    };
    emif@60000000 {
    };
};
=>

```

5.9.7.8. fdt chosen - fixup dynamic info

One of the modifications made by U-Boot to the blob before passing it to the kernel is the addition of the `/chosen` node. Linux 2.6 Documentation/powerpc/booting-without-of.txt says that this node is used to store "some variable environment information, like the arguments, or the default input/output devices." To force U-Boot to add the `/chosen` node to the current blob, `fdt chosen` command can be used. Let's now verify its operation:

```

=> fdt list /
/ {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    model = "EnBW CMC";
    compatible = "enbw,cmc";
    chosen {
    };
    aliases {
    };
    memory {
    };
    soc@1c00000 {
    };
    onchipram@8000000 {
    };
    emif@60000000 {
    };
};
=> fdt chosen
=> fdt list /
/ {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    model = "EnBW CMC";
    compatible = "enbw,cmc";
    chosen {
    };
    aliases {
    };
    memory {
    };
    soc@1c00000 {
    };
    onchipram@8000000 {
    };
    emif@60000000 {
    };
};
=> fdt list /chosen
chosen {
};
=>

```

Note: `fdt boardsetup` performs board-specific blob updates, most commonly setting clock frequencies, etc. Discovering its operation is left as an exercise for the reader.

5.9.8. Special Commands

5.9.8.1. i2c - I2C sub-system

```

=> help i2c
i2c - I2C sub-system

Usage:
i2c crc32 chip address[.0, .1, .2] count - compute CRC32 checksum
i2c loop chip address[.0, .1, .2] [# of objects] - looping read of device
i2c md chip address[.0, .1, .2] [# of objects] - read from I2C device
i2c mm chip address[.0, .1, .2] - write to I2C device (auto-incrementing)

```

```
i2c mw chip address[.0, .1, .2] value [count] - write to I2C device (fill)
i2c nm chip address[.0, .1, .2] - write to I2C device (constant address)
i2c probe - show devices on the I2C bus
i2c read chip address[.0, .1, .2] length memaddress - read to memory
i2c reset - re-init the I2C Controller
i2c speed [speed] - show or set I2C bus speed
=>
```

5.9.9. Storage devices

This chapter introduces commands to work with storage devices, i.e. ATA, CF, SATA, SCSI, USB, NAND, etc. connected to the board.

5.9.9.1. MMC devices

```
=> help mmc
mmc - MMC sub system
```

```
Usage:
mmc read addr blk# cnt
mmc write addr blk# cnt
mmc erase blk# cnt
mmc rescan
mmc part - lists available partition on current mmc device
mmc dev [dev] [part] - show or set current mmc device [partition]
mmc list - lists available devices
=>
```

The `mmc dev` command displays the current device

```
=> mmc dev
mmc0 is current device
=>
```

The `mmc list` command displays the available mmc devices

```
=> mmc list
davinci: 0
=>
```

With the `mmc rescan` command you can rescan the actual mmc device.

```
=> mmc rescan
=>
```

The `mmcinfo` command displays the information about the actual mmc device

```
=> mmcinfo
Device: davinci
Manufacturer ID: 3
OEM: 5344
Name: SU02G
Tran Speed: 25000000
Rd Block Len: 512
SD version 2.0
High Capacity: No
Capacity: 1.8 GiB
Bus Width: 4-bit
=>
```

The `mmc part` command displays the available partitions on the actual mmc device.

```
=> mmc part

Partition Map for MMC device 0 -- Partition Type: DOS

Partition      Start Sector      Num Sectors      Type
  1              63          273042          c
  2         273105         2120580          83
  3        2393685         1285200          83
=>
```

You can read data from the mmc with the `mmc read` command:

```
=> mmc read 0xc0000000 247 10

MMC read: dev # 0, block # 583, count 16 ... 16 blocks read: OK
=> md 0xc0000000
c0000000: 00000000 00000000 00000000 00000000 .....
c0000010: 00000000 00000000 00000000 00000000 .....
c0000020: 00000000 00000000 00000000 00000000 .....
c0000030: 00000000 00000000 00000000 00000000 .....
c0000040: 00000000 00000000 00000000 00000000 .....
c0000050: 00000000 00000000 00000000 00000000 .....
c0000060: 00000000 00000000 00000000 00000000 .....
c0000070: 00000000 00000000 00000000 00000000 .....
c0000080: 00000000 00000000 00000000 00000000 .....
c0000090: 00000000 00000000 00000000 00000000 .....
c00000a0: 00000000 00000000 00000000 00000000 .....
c00000b0: 00000000 00000000 00000000 00000000 .....
c00000c0: 00000000 00000000 00000000 00000000 .....
c00000d0: 00000000 00000000 00000000 00000000 .....
c00000e0: 00000000 00000000 00000000 00000000 .....
c00000f0: 00000000 00000000 00000000 00000000 .....
=>
```

You can erase/write data from the mmc with the `mmc erase` / `mmc write` commands:

```
=> mmc read 0xc0300000 247 10

MMC read: dev # 0, block # 583, count 16 ... 16 blocks read: OK
=> md 0xc0300000
c0300000: 00000000 00000000 00000000 00000000 .....
c0300010: 00000000 00000000 00000000 00000000 .....
c0300020: 00000000 00000000 00000000 00000000 .....
c0300030: 00000000 00000000 00000000 00000000 .....
c0300040: 00000000 00000000 00000000 00000000 .....
c0300050: 00000000 00000000 00000000 00000000 .....
c0300060: 00000000 00000000 00000000 00000000 .....
c0300070: 00000000 00000000 00000000 00000000 .....
=>
```

```

c0300080: 00000000 00000000 00000000 00000000 .....
c0300090: 00000000 00000000 00000000 00000000 .....
c03000a0: 00000000 00000000 00000000 00000000 .....
c03000b0: 00000000 00000000 00000000 00000000 .....
c03000c0: 00000000 00000000 00000000 00000000 .....
c03000d0: 00000000 00000000 00000000 00000000 .....
c03000e0: 00000000 00000000 00000000 00000000 .....
c03000f0: 00000000 00000000 00000000 00000000 .....
=> mw 0xc0000000 0xdeadface 1000
=> md 0xc0000000
c0000000: deadface deadface deadface deadface .....
c0000010: deadface deadface deadface deadface .....
c0000020: deadface deadface deadface deadface .....
c0000030: deadface deadface deadface deadface .....
c0000040: deadface deadface deadface deadface .....
c0000050: deadface deadface deadface deadface .....
c0000060: deadface deadface deadface deadface .....
c0000070: deadface deadface deadface deadface .....
c0000080: deadface deadface deadface deadface .....
c0000090: deadface deadface deadface deadface .....
c00000a0: deadface deadface deadface deadface .....
c00000b0: deadface deadface deadface deadface .....
c00000c0: deadface deadface deadface deadface .....
c00000d0: deadface deadface deadface deadface .....
c00000e0: deadface deadface deadface deadface .....
c00000f0: deadface deadface deadface deadface .....
=> mmc erase 247 10

```

```

MMC erase: dev # 0, block # 583, count 16 ... 16 blocks erase: OK
=> mmc write 0xc0000000 247 10

```

```

MMC write: dev # 0, block # 583, count 16 ... 16 blocks write: OK
=> mw 0xc0000000 0x0 1000
=> mmc read 0xc0000000 247 10

```

```

MMC read: dev # 0, block # 583, count 16 ... 16 blocks read: OK
=> md 0xc0000000
c0000000: deadface deadface deadface deadface .....
c0000010: deadface deadface deadface deadface .....
c0000020: deadface deadface deadface deadface .....
c0000030: deadface deadface deadface deadface .....
c0000040: deadface deadface deadface deadface .....
c0000050: deadface deadface deadface deadface .....
c0000060: deadface deadface deadface deadface .....
c0000070: deadface deadface deadface deadface .....
c0000080: deadface deadface deadface deadface .....
c0000090: deadface deadface deadface deadface .....
c00000a0: deadface deadface deadface deadface .....
c00000b0: deadface deadface deadface deadface .....
c00000c0: deadface deadface deadface deadface .....
c00000d0: deadface deadface deadface deadface .....
c00000e0: deadface deadface deadface deadface .....
c00000f0: deadface deadface deadface deadface .....
=> mmc erase 247 10

```

```

MMC erase: dev # 0, block # 583, count 16 ... 16 blocks erase: OK
=> mmc write 0xc0300000 247 10

```

```

MMC write: dev # 0, block # 583, count 16 ... 16 blocks write: OK
=> mmc read 0xc0000000 247 10

```

```

MMC read: dev # 0, block # 583, count 16 ... 16 blocks read: OK
=> md 0xc0000000
c0000000: 00000000 00000000 00000000 00000000 .....
c0000010: 00000000 00000000 00000000 00000000 .....
c0000020: 00000000 00000000 00000000 00000000 .....
c0000030: 00000000 00000000 00000000 00000000 .....
c0000040: 00000000 00000000 00000000 00000000 .....
c0000050: 00000000 00000000 00000000 00000000 .....
c0000060: 00000000 00000000 00000000 00000000 .....
c0000070: 00000000 00000000 00000000 00000000 .....
c0000080: 00000000 00000000 00000000 00000000 .....
c0000090: 00000000 00000000 00000000 00000000 .....
c00000a0: 00000000 00000000 00000000 00000000 .....
c00000b0: 00000000 00000000 00000000 00000000 .....
c00000c0: 00000000 00000000 00000000 00000000 .....
c00000d0: 00000000 00000000 00000000 00000000 .....
c00000e0: 00000000 00000000 00000000 00000000 .....
c00000f0: 00000000 00000000 00000000 00000000 .....
=>

```

if you have a dos partition on the device, you can list the content with:

```

=> fatls mmc 0 /
    17036    mlo
    202736  u-boot.bin
    2360344  uimage
           boot/
           date_of_creation
    1024    random.hex

```

```
5 file(s), 1 dir(s)
```

```
=>
```

and load a file in RAM with:

```

=> fatload mmc 0 0xc0000000 date_of_creation
reading date_of_creation

```

```

29 bytes read
=> md 0xc0000000
c0000000: 20646557 20766f4e 30203332 30343a39 Wed Nov 23 09:40
c0000010: 2039353a 2054454d 31313032 0000000a :59 MET 2011...
c0000020: 00000000 00000000 00000000 00000000 .....
c0000030: 00000000 00000000 00000000 00000000 .....
c0000040: 00000000 00000000 00000000 00000000 .....
c0000050: 00000000 00000000 00000000 00000000 .....
c0000060: 00000000 00000000 00000000 00000000 .....
c0000070: 00000000 00000000 00000000 00000000 .....
c0000080: 00000000 00000000 00000000 00000000 .....
c0000090: 00000000 00000000 00000000 00000000 .....
c00000a0: 00000000 00000000 00000000 00000000 .....
c00000b0: 00000000 00000000 00000000 00000000 .....
c00000c0: 00000000 00000000 00000000 00000000 .....
c00000d0: 00000000 00000000 00000000 00000000 .....

```

```
c00000e0: 00000000 00000000 00000000 00000000 .....
c00000f0: 00000000 00000000 00000000 00000000 .....
=>
```

5.9.9.2. NAND devices

U-Boot allows us to directly work with NAND devices attached directly or through a NAND controller. The commands are grouped under the `nand` subsystem:

```
=> help nand
nand - NAND sub-system

Usage:
nand info - show available NAND devices
nand device [dev] - show or set current device
nand read - addr off|partition size
nand write - addr off|partition size
             read/write 'size' bytes starting at offset 'off'
             to/from memory address 'addr', skipping bad blocks.
nand read.raw - addr off|partition
nand write.raw - addr off|partition
             Use read.raw/write.raw to avoid ECC and access the page as-is.
nand erase[.spread] [clean] off size - erase 'size' bytes from offset 'off'
             With '.spread', erase enough for given file size, otherwise,
             'size' includes skipped bad blocks.
nand erase.part [clean] partition - erase entire mtd partition'
nand erase.chip [clean] - erase entire chip'
nand bad - show bad blocks
nand dump[.oob] off - dump page
nand scrub [-y] off size | scrub.part partition | scrub.chip
             really clean NAND erasing bad blocks (UNSAFE)
nand markbad off [...] - mark bad block(s) at offset (UNSAFE)
nand biterr off - make a bit error at offset (UNSAFE)
=>
```

5.9.9.2.1. nand bad - show bad block information

As NAND devices can develop bad blocks over their lifetime and usually even are delivered with bad blocks already, the NAND commands need to be aware of this fact. Getting information of the current bad block list is easy:

```
=> nand bad

Device 0 bad blocks:
Bad block table found at page 65472, version 0x01
Bad block table found at page 65408, version 0x01
 07f80000
 07fa0000
 07fc0000
 07fe0000
=>
```

5.9.9.2.2. nand erase - erase region

Ensuring that the NAND device functions properly, we will use the basic commands to construct a testpattern in memory, write that to the device, of course erasing it first, and read the data back. A final comparison will show if all data was transferred correctly.

We begin by erasing 64k at the start of the NAND device:

```
=> nand erase 0 0x10000

NAND erase: device 0 offset 0x0, size 0x10000
Erasing at 0x0 -- 100% complete.
OK
=>
```

5.9.9.2.3. nand write - write to NAND device

Let's create a testpattern in memory and write that to the previously erased NAND area:

```
=> mw 0xc0000000 0x55aa55aa 0x4000
=> nand write 0xc0000000 0 0x10000

NAND write: device 0 offset 0x0, size 0x10000
65536 bytes written: OK
=>
```

5.9.9.2.4. nand read - read from NAND device

As everything worked ok, we ensure everything was fine by transferring the data to a different location in RAM and check this against the original written content:

```
=> nand read 0xc0300000 0 0x10000

NAND read: device 0 offset 0x0, size 0x10000
65536 bytes read: OK
=> md 0xc0300000
c0300000: 55aa55aa 55aa55aa 55aa55aa 55aa55aa .U.U.U.U.U.U.U.U
c0300010: 55aa55aa 55aa55aa 55aa55aa 55aa55aa .U.U.U.U.U.U.U.U
=> cmp 0xc0000000 0xc0300000 0x4000
Total of 16384 words were the same
=>
```

5.9.10. Miscellaneous Commands

5.9.10.1. date - get/set/reset date & time

```
=> help date
date - get/set/reset date & time

Usage:
date [MMDDhhmm[[CC]YY][.ss]]
date reset
  - without arguments: print date & time
  - with numeric argument: set the system date & time
  - with 'reset' argument: reset the RTC
=>
```

The `date` command is used to display the current time in a standard format, or to set the system date. On some systems it can also be used to reset (initialize) the system clock:

```
=> date reset
Reset RTC...
Date: 2011-11-23 (Wednesday)    Time:  8:57:07
=> date 112308572011
Date: 2011-11-23 (Wednesday)    Time:  8:57:00
=> date
Date: 2011-11-23 (Wednesday)    Time:  8:57:01
=>
```

5.9.10.2. echo - echo args to console

```
=> help echo
echo - echo args to console
```

```
Usage:
echo [args...]
    - echo args to console; \c suppresses newline
=>
```

The `echo` command echoes the arguments to the console:

```
=> echo The quick brown fox jumped over the lazy dog.
The quick brown fox jumped over the lazy dog.
=>
```

5.9.10.3. reset - Perform RESET of the [CPU](#)

```
=> help reset
reset - Perform RESET of the CPU
```

```
Usage:
reset
=>
```

The `reset` command reboots the system.

```
=>
=> reset
resetting ...
```

```
U-Boot 2011.09-03846-gccd5912 (Nov 23 2011 - 07:34:54)
```

```
Cores: ARM 456 MHz
DDR:    300 MHz
I2C:    ready
DRAM:   64 MiB
WARNING: Caches not enabled
Flash:  2 MiB
NAND:   128 MiB
MMC:    davinci: 0
In:     serial
Out:    serial
Err:    serial
Un-Protected 1 sectors
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
Protected 1 sectors
Net:    DaVinci-EMAC
key: 3
Hit any key to stop autoboot:  0
=>
```

5.9.10.4. sleep - delay execution for some time

```
=> help sleep
sleep - delay execution for some time
```

```
Usage:
sleep N
    - delay execution for N seconds (N is _decimal_ !!!)
=>
```

The `sleep` command pauses execution for the number of seconds given as the argument:

```
=> sleep 5
=>
```

5.9.10.5. version - print monitor version

```
=> help version
version - print monitor, compiler and linker version
```

```
Usage:
version
=>
```

You can print the version and build date of the U-Boot image running on your system using the `version` command (short: `vers`):

```
=> version

U-Boot 2011.09-03846-gccd5912 (Nov 23 2011 - 07:34:54)
arm-linux-gnueabi-gcc (GCC) 4.5.1
GNU ld (GNU Binutils) 2.21
=>
```

5.9.10.6. ? - alias for 'help'

You can use `?` as a short form for the `help` command (see description above).

5.10. U-Boot Environment Variables

The U-Boot environment is a block of memory that is kept on persistent storage and copied to RAM when U-Boot starts. It is used to store environment variables which can be used to configure the system. The environment is protected by a CRC32 checksum.

This section lists the most important environment variables, some of which have a special meaning to U-Boot. You can use these variables to configure the behaviour of U-Boot to your liking.

- **autoload**: if set to "no" (or any string beginning with 'n'), the `rarpb`, `bootp` or `dhcp` commands will perform *only* a configuration lookup from the [BOOTP](#) / [DHCP](#) server, but not try to load any image using [TFTP](#).
- **autostart**: if set to "yes", an image loaded using the `rarpb`, `bootp`, `dhcp`, `tftp`, `disk`, or `docb` commands will be automatically started (by internally calling the `bootm` command).
- **baudrate**: a decimal number that selects the console baudrate (in bps). Only a predefined list of baudrate settings is available. When you change the baudrate (using the "setenv baudrate ..." command), U-Boot will switch the baudrate of the console terminal and wait for a newline which must be entered with the *new* speed setting. This is to make sure you can actually type at the new speed. If this fails, you have to reset the board (which will operate at the old speed since you were not able to saveenv the new settings.)
If no "baudrate" variable is defined, the default baudrate of 115200 is used.
- **bootargs**: The contents of this variable are passed to the Linux kernel as boot arguments (aka "command line").
- **bootcmd**: This variable defines a command string that is automatically executed when the initial countdown is not interrupted. This command is only executed when the variable `bootdelay` is also defined!
- **bootdelay**: After reset, U-Boot will wait this number of seconds before it executes the contents of the `bootcmd` variable. During this time a countdown is printed, which can be interrupted by pressing any key.
Set this variable to 0 boot without delay. Be careful: depending on the contents of your `bootcmd` variable, this can prevent you from entering interactive commands again forever!
Set this variable to -1 to disable autoboot.
- **bootfile**: name of the default image to load with [TFTP](#)
- **cpuclock**: (Only with MPC859 / MPC866 / MPC885 processors) On some processors, the [CPU](#) clock frequency can be adjusted by the user (for example to optimize performance versus power dissipation). On such systems the `cpuclock` variable can be set to the desired [CPU](#) clock value, in MHz. If the `cpuclock` variable exists and its value is within the compile-time defined limits (`CFG_866_CPUCLK_MIN` and `CFG_866_CPUCLK_MAX` = minimum resp. maximum allowed [CPU](#) clock), then the specified value is used. Otherwise, the default [CPU](#) clock value is set.
- **ethaddr**: Ethernet [MAC](#) address for first/only ethernet interface (= `eth0` in Linux).
This variable can be set only once (usually during manufacturing of the board). U-Boot refuses to delete or overwrite this variable once it has been set.
- **eth1addr**: Ethernet [MAC](#) address for second ethernet interface (= `eth1` in Linux).
- **eth2addr**: Ethernet [MAC](#) address for third ethernet interface (= `eth2` in Linux).
...
- **initrd_high**: used to restrict positioning of initrd ramdisk images:
If this variable is not set, initrd images will be copied to the highest possible address in RAM; this is usually what you want since it allows for maximum initrd size. If for some reason you want to make sure that the initrd image is loaded below the `CFG_BOOTMAPSZ` limit, you can set this environment variable to a value of "no" or "off" or "0". Alternatively, you can set it to a maximum upper address to use (U-Boot will still check that it does not overwrite the U-Boot stack and data).
For instance, when you have a system with 16 MB RAM, and want to reserve 4 MB from use by Linux, you can do this by adding "mem=12M" to the value of the "bootargs" variable.
However, now you must make sure that the initrd image is placed in the first 12 MB as well - this can be done with

```
=> setenv initrd_high 00c00000
```

Setting `initrd_high` to the highest possible address in your system (0xFFFFFFFF) prevents U-Boot from copying the image to RAM at all. This allows for faster boot times, but requires a Linux kernel with zero-copy ramdisk support.

- **ipaddr**: IP address; needed for `tftp` command
- **loadaddr**: Default load address for commands like `tftp` or `loads`.
- **loads_echo**: If set to 1, all characters received during a serial download (using the `loads` command) are echoed back. This might be needed by some terminal emulations (like `cu`), but may as well just take time on others.
- **mtdparts**: This variable (usually defined using the [mtdparts](#) command) allows to share a common [MTD](#) partition scheme between U-Boot and the Linux kernel.
- **pram**: If the "Protected RAM" feature is enabled in your board's configuration, this variable can be defined to enable the reservation of such "protected RAM", i. e. RAM which is not overwritten by U-Boot. Define this variable to hold the number of kB you want to reserve for pRAM. Note that the board info structure will still show the full amount of RAM. If pRAM is reserved, a new environment variable "mem" will automatically be defined to hold the amount of remaining RAM in a form that can be passed as boot argument to Linux, for instance like that:

```
=> setenv bootargs ${bootargs} mem=\${mem}
=> saveenv
```

This way you can tell Linux not to use this memory, either, which results in a memory region that will not be affected by reboots.

- **serverip**: [TFTP](#) server IP address; needed for `tftp` command.
- **serial#**: contains hardware identification information such as type string and/or serial number.
This variable can be set only once (usually during manufacturing of the board). U-Boot refuses to delete or overwrite this variable once it has been set.
- **silent**: If the configuration option `CONFIG_SILENT_CONSOLE` has been enabled for your board, setting this variable to any value will suppress all console messages. Please see `doc/README.silent` for details.
- **verify**: If set to `n` or `no` disables the checksum calculation over the complete image in the `bootm` command to trade speed for safety in the boot process. Note that the header checksum is still verified.

The following environment variables may be used and automatically updated by the network boot commands (`bootp`, `dhcp`, or `tftp`), depending the information provided by your boot server:

- **bootfile**: see above
- **dnsip**: IP address of your Domain Name Server
- **gatewayip**: IP address of the Gateway (Router) to use
- **hostname**: Target hostname
- **ipaddr**: see above
- **netmask**: Subnet Mask
- **rootpath**: Pathname of the root filesystem on the NFS server
- **serverip**: see above
- **filesize**: Size (as hex number in bytes) of the file downloaded using the last `bootp`, `dhcp`, or `tftp` command.

5.11. U-Boot Scripting Capabilities

U-Boot allows to store commands or command sequences in a plain text file. Using the `mkimage` tool you can then convert this file into a **script image** which can be executed using U-Boot's `autoscr` command.

For example, assume that you will have to run the following sequence of commands on many boards, so you store them in a text file, say `"setenv-commands"`:

```
bash$ cat setenv-commands
setenv loadaddr 00200000
echo ===== U-Boot settings =====
setenv u-boot /tftpboot/TQM860L/u-boot.bin
setenv u-boot_addr 40000000
setenv load_u-boot 'tftp ${loadaddr} ${u-boot}'
setenv install_u-boot 'protect off ${u-boot_addr} +${filesize};era ${u-boot_addr} +${filesize};cp.b ${loadaddr} ${u-boot_addr} ${filesize};saveenv'
setenv update_u-boot run load_u-boot install_u-boot
echo ===== Linux Kernel settings =====
setenv bootfile /tftpboot/TQM860L/uImage
setenv kernel_addr 40040000
setenv load_kernel 'tftp ${loadaddr} ${bootfile};'
setenv install_kernel 'era ${kernel_addr} +${filesize};cp.b ${loadaddr} ${kernel_addr} ${filesize}'
setenv update_kernel run load_kernel install_kernel
echo ===== Ramdisk settings =====
setenv ramdisk /tftpboot/TQM860L/uRamdisk
setenv ramdisk_addr 40100000
setenv load_ramdisk 'tftp ${loadaddr} ${ramdisk};'
setenv install_ramdisk 'era ${ramdisk_addr} +${filesize};cp.b ${loadaddr} ${ramdisk_addr} ${filesize}'
setenv update_ramdisk run load_ramdisk install_ramdisk
echo ===== Save new definitions =====
saveenv
bash$
```


To convert the text file into a script image for U-Boot, you have to use the `mkimage` tool as follows:

```
bash$ mkimage -T script -C none -n 'Demo Script File' -d setenv-commands setenv.img
Image Name:      Demo Script File
Created:         Mon Jun  6 13:33:14 2005
Image Type:      PowerPC Linux Script (uncompressed)
Data Size:       1147 Bytes = 1.12 kB = 0.00 MB
Load Address:    0x00000000
Entry Point:     0x00000000
Contents:
  Image 0:       1139 Bytes = 1 kB = 0 MB
bash$
```

On the target, you can download this image as usual (for example, using the `"tftp"` command). Use the `"autoscr"` command to execute it:

```
=> tftp 100000 /tftpboot/TQM860L/setenv.img
Using FEC ETHERNET device
TFTP from server 192.168.3.1; our IP address is 192.168.3.80
Filename '/tftpboot/TQM860L/setenv.img'.
Load address: 0x100000
Loading: #
done
Bytes transferred = 1211 (4bb hex)
=> imi 100000
```

```
## Checking Image at 00100000 ...
  Image Name:      Demo Script File
  Created:         2005-06-06 11:33:14 UTC
  Image Type:      PowerPC Linux Script (uncompressed)
  Data Size:       1147 Bytes = 1.1 kB
  Load Address:   00000000
  Entry Point:     00000000
  Verifying Checksum ... OK
=> autoscr 100000
## Executing script at 00100000
===== U-Boot settings =====
===== Linux Kernel settings =====
===== Ramdisk settings =====
===== Save new definitions =====
Saving Environment to Flash...
Un-Protected 1 sectors
Un-Protected 1 sectors
Erasing Flash...
.. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
Protected 1 sectors
=>
```

 Hint: maximum flexibility can be achieved if you are using the Hush shell as command interpreter in U-Boot; see section [14.2.17. How the Command Line Parsing Works](#)

5.12. U-Boot Standalone Applications

U-Boot supports "standalone" applications, which are loaded dynamically; these applications can have access to the U-Boot console I/O functions, memory allocation and interrupt services.

A couple of simple examples are included with the U-Boot source code:

5.12.1. "Hello World" Demo

examples/hello_world.c contains a small "Hello World" Demo application; it is automatically compiled when you build U-Boot. It's configured to run at address 0x00040004, so you can play with it like that:

```
=> loads
## Ready for S-Record download ...
~>examples/hello_world.srec
1 2 3 4 5 6 7 8 9 10 11 ...
[file transfer complete]
[connected]
## Start Addr = 0x00040004

=> go 40004 Hello World! This is a test.
## Starting application at 0x00040004 ...
Hello World
argc = 7
argv[0] = "40004"
argv[1] = "Hello"
argv[2] = "World!"
argv[3] = "This"
argv[4] = "is"
argv[5] = "a"
```


```

argv[6] = "test."
argv[7] = ""
Hit any key to exit ...

## Application terminated, rc = 0x0

```

Alternatively, you can of course use TFTP to download the image over the network. In this case the binary image (`hello_world.bin`) is used.

 Note that the entry point of the program is at offset 0x0004 from the start of file, i. e. the download address and the entry point address differ by four bytes.


```

=> tftp 40000 /tftpboot/hello_world.bin
...
=> go 40004 This is another test.
## Starting application at 0x00040004 ...
Hello World
argc = 5
argv[0] = "40004"
argv[1] = "This"
argv[2] = "is"
argv[3] = "another"
argv[4] = "test."
argv[5] = ""
Hit any key to exit ...

## Application terminated, rc = 0x0

```

5.12.2. Timer Demo

 This example is only available on MPC8xx [CPUs](#).

5.13. U-Boot Image Formats

U-Boot operates on "image" files which can be basically anything, preceeded by a special header; see the definitions in *include/image.h* for details; basically, the header defines the following image properties:

- Target Operating System (Provisions for OpenBSD, NetBSD, FreeBSD, 4.4BSD, Linux, SVR4, Esix, Solaris, Irix, SCO, Dell, NCR, LynxOS, pSOS, QNX, RTEMS, U-Boot, ARTOS, Unity OS, Integrity; Currently supported: Linux, NetBSD, VxWorks, QNX, RTEMS, ARTOS, Unity OS, Integrity).
- Target [CPU](#) Architecture (Provisions for Alpha, ARM, AVR32, BlackFin, IA64, M68K, Microblaze, MIPS, MIPS64, NIOS, NIOS2, [Power Architecture](#)®, IBM S390, SuperH, Sparc, Sparc 64 Bit, Intel x86; Currently supported: ARM, AVR32, BlackFin, M68K, Microblaze, MIPS, MIPS64, NIOS, NIOS2, [Power Architecture](#)®, SuperH, Sparc, Sparc 64 Bit, Intel x86).
- Compression Type (Provisions for uncompressed, gzip, bzip2, lzo; Currently supported: uncompressed, gzip, bzip2, lzo).
- Load Address
- Entry Point
- Image Name
- Image Timestamp

The header is marked by a special Magic Number, and both the header and the data portions of the image are secured against corruption by CRC32 checksums.

5.14. U-Boot Advanced Features

5.14.1. Boot Count Limit

The Open Source Development Labs *Carrier Grade Linux Requirements Definition* version 2.0 (http://www.osdl.org/docs/carrier_grade_linux_requirements_definition_version_20_final_public_draft.pdf) contains the following requirement definition (ID PLT.4.0, p. 44):


CGL shall provide support for detecting a repeating reboot cycle due to recurring failures and will go to an offline state if this occurs.

This feature is available in U-Boot if you enable the `CONFIG_BOOTCOUNT_LIMIT` configuration option. The implementation uses the following environment variables:

- bootcount:**
This variable will be automatically created if it does not exist, and it will be updated at each reset of the processor. After a power-on reset, it will be initialized with **1**, and each reboot will increment the value by **1**.
- bootlimit:**
If this variable exists, its contents are taken as the maximum number of reboot cycles allowed.
- altbootcmd:**
If, after a reboot, the new value of **bootcount** exceeds the value of **bootlimit**, then instead of the standard boot action (executing the contents of **bootcmd**) an alternate boot action will be performed, and the contents of **altbootcmd** will be executed.

If the variable **bootlimit** is not defined in the environment, the Boot Count Limit feature is disabled. If it is enabled, but **altbootcmd** is not defined, then U-Boot will drop into interactive mode and remain there.

It is the responsibility of some application code (typically a Linux application) to reset the variable **bootcount**, thus allowing for more boot cycles.

 At the moment, the Boot Count Limit feature is available only for MPC8xx, MPC82xx and MPC5200 [Power Architecture](#)® processors.

- [ubootBootcountAccess.c](#): C-source: bootcount access through /proc file system
- [6. Embedded Linux Configuration](#)
 - [6.1. Download and Unpack the Linux Kernel Sources](#)
 - [6.2. Kernel Configuration and Compilation](#)
 - [6.3. Installation](#)

6. Embedded Linux Configuration

6.1. Download and Unpack the Linux Kernel Sources

You can download the Linux Kernel Sources from our anonymous [git](#) server at <http://git.denx.de/>. To checkout the module for the first time, proceed as follows:

```

bash$ cd /opt/eldk/usr/src
bash$ git clone git://git.denx.de/linux-denx.git linux-denx
bash$ cd linux-denx
bash$ git checkout -b duts remotes/origin/enbw_cmc

```

```
Branch duts set up to track remote branch enbw_cmc from origin.
Switched to a new branch 'duts'
bash$
```

6.2. Kernel Configuration and Compilation

The enbw_cmc board is fully supported by DENX Software Engineering. This means that you will always be able to build a working default configuration with just minimal interaction.

Please be aware that you will need the "arm" cross development tools for the following steps. Make sure that the directory which contains the binaries of your [ELDK](#) are in your **PATH**.


To be sure that no intermediate results of previous builds are left in your Linux kernel source tree you can clean it up as follows:

```
bash$ make mrproper
```

The following command selects a standard configuration for the enbw_cmc board that has been extensively tested. It is recommended to use this as a starting point for other, customized configurations:

```
bash$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- enbw_cmc_defconfig

[hs@pollux]$
```

 Note: The name of this default configuration file is **arch/arm/configs/XXX**. By (recursively) listing the contents of the **arch/arm/configs/** directory you can easily find out which other default configurations are available.

If you don't want to change the default configuration you can now continue to use it to build a kernel image:


```
bash$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- uImage
```

Otherwise you can modify the kernel configuration as follows:

```
bash$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- config
```

or

```
bash$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig
```

 Note: Because of problems (especially with some older Linux kernel versions) the use of "make xconfig" is **not** recommended.

```
bash$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- uImage
```

The **make** target **uImage** uses the tool **mkimage** (from the U-Boot package) to create a Linux kernel image in *arch/arm/boot/uImage* which is immediately usable for download and booting with U-Boot.

In case you need a DTB to boot your linux kernel, you need the following step:

```
bash$ make enbw_cmc.dtb
```

In case you configured modules you will also need to compile the modules:

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules
```

add install the modules (make sure to pass the correct root path for module installation):

```
bash$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- INSTALL_MOD_PATH=/opt/eldk-5.0/armv5te/rootfs modules_install
```

6.3. Installation

For now it is sufficient to copy the Linux kernel image into the directory used by your [TFTP](#) server:

```
bash$ cp arch/arm/boot/uImage /tftpboot/uImage
```

- [7. Booting Embedded Linux](#)
 - [7.1. Introduction](#)
 - [7.2. Flattened Device Tree Blob](#)
 - [7.3. Passing Kernel Arguments](#)
 - [7.4. Boot Arguments Unleashed](#)
 - [7.5. Networked Operation with Root Filesystem over NFS](#)
 - [7.5.1. Bootlog of tftp'd Linux kernel with Root Filesystem over NFS](#)
 - [7.6. Boot from NAND Flash Memory](#)
 - [7.7. Standalone Operation with Ramdisk Image](#)

7. Booting Embedded Linux

7.1. Introduction

In principle, if you have a Linux kernel image and the flattened device tree blob somewhere in system memory (RAM, ROM, flash...), then all you need to boot the system is the `bootm` command. Assume a Linux kernel image has been stored at address `0xC0700000` and the flattened device tree blob has been stored at address `0xC0600000` - then you can boot this image with the following command:

```
=> bootm C0700000 - C0600000
```

7.2. Flattened Device Tree Blob

Linux kernel expects certain information on the hardware that it runs on. For kernels compiled with fdt support, this information has the form of a device tree, which is based on the Open Firmware specification. Bootloaders like U-Boot that do not implement the Open Firmware API, are expected to pass to the kernel a binary form of the flattened device tree, commonly referred to as *FDT blob* or simply *the blob*.

Device trees are defined in human-readable text files, which are part of the Linux 2.6 source tree. Device tree source for the enbw_cmc board is found in `arch/arm/boot/dts/enbw_cmc.dts` file. Before the device tree can be passed to the kernel, it has to be compiled to the binary form by the `dtc` compiler. The `dtc` compiler is included with the Linux kernel since 2.6.25. Since 2.6.26 there is also a simple makefile rule to generate the blob:

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- enbw_cmc.dtb
```

After the blob has been compiled, it has to be transferred from where it was built ("`arch/arm/boot/enbw_cmc.dtb`") to target's memory, for example over the [TFTP](#) protocol using U-Boot's `tftp` command. Then, the blob is passed to the kernel by the `bootm` command, and its address in memory is one of the arguments to `bootm` - refer to the description of this command in [UBootCmdGroupExec](#) for more details. Note that U-Boot makes some automatic modifications to the blob before passing it to the kernel - mainly adding and modifying information that is

learnt at run-time.

U-Boot also has provisions to alter a flattened device tree in arbitrary ways from the command line, refer to the description of the `fdt` commands found in [UBootCmdGroupMisc](#).

Notes:

- Flattened Device Tree custodian's page at <http://www.denx.de/wiki/U-Boot/UBootFdtInfo> contains useful information, and a number of references.
- At the time of this writing (September 2007) blob handling is still a very fresh feature and undergonig frequent changes. Reader is encouraged to watch the `u-boot-users` and `linuxppc-dev` mailing lists for important news (required version of the `dtc` compiler, blob compilation options, flattened device tree source file structure, etc.).

7.3. Passing Kernel Arguments

In nearly all cases, you will want to pass additional information to the Linux kernel; for instance, information about the root device or network configuration.

In U-Boot, this is supported using the `bootargs` environment variable. Its contents are automatically passed to the Linux kernel as boot arguments (or "command line" arguments). This allows the use of the same Linux kernel image in a wide range of configurations. For instance, by just changing the contents of the `bootargs` variable you can use the very same Linux kernel image to boot with an `initrd` ramdisk image, with a root filesystem over NFS, with a [CompactFlash](#) disk or from a flash filesystem.

As one example, to boot the Linux kernel image at address `0xC0700000` using the `initrd` ramdisk image at address `0xC0A00000` as root filesystem, and with the flattened device tree blob at address `0xC0600000`, you can use the following commands:

```
=> setenv bootargs root=/dev/ram rw
=> bootm 0xC0700000 0xC0A00000 0xC0600000
```

To boot the same kernel image with a root filesystem over NFS, the following command sequence can be used. This example assumes that your NFS server has the IP address "192.168.1.1" and exports the directory `/opt/eldk-5.0/armv5te/rootfs` as root filesystem for the target. The target has been assigned the IP address "192.168.20.38" and the hostname "enbw_cmc". A netmask of "255.255.0.0" is used:

```
=> setenv bootargs root=/dev/nfs rw nfsroot=192.168.1.1:/opt/eldk-5.0/armv5te/rootfs ip=192.168.20.38:192.168.1.1:192.168.1.1:255.255.0.0:enbw_cmc::off
=> bootm 0xC0700000 - 0xC0600000
```

Please see also the files `Documentation/initrd.txt` and `Documentation/nfsroot.txt` in your Linux kernel source directory for more information about which options can be passed to the Linux kernel.

☐ Note: Once your system is up and running, if you have a simple shell login, you can normally examine the boot arguments that were used by the kernel for the most recent boot with the command:

```
$ cat /proc/cmdline
```

7.4. Boot Arguments Unleashed

Passing command line arguments to the Linux kernel allows for very flexible and efficient configuration which is especially important in Embedded Systems. It is somewhat strange that these features are nearly undocumented everywhere else. One reason for that is certainly the very limited capabilities of other boot loaders.

It is especially U-Boot's capability to easily define, store, and use environment variables that makes it such a powerful tool in this area. In the examples above we have already seen how we can use for instance the `root` and `ip` boot arguments to pass information about the root filesystem or network configuration. The `ip` argument is not only useful in configurations with root filesystem over NFS; if the Linux kernel has the `CONFIG_IP_PNP` configuration enabled (IP kernel level autoconfiguration), this can be used to enable automatic configuration of IP addresses of devices and of the routing table during kernel boot, based on either information supplied on the kernel command line or by [BOOTP](#) or RARP protocols.

The advantage of this mechanism is that you don't have to spend precious system memory (RAM and flash) for network configuration tools like `ifconfig` or `route` - especially in Embedded Systems where you seldom have to change the network configuration while the system is running.

We can use U-Boot environment variables to store all necessary configuration parameters:

```
=> setenv ipaddr 192.168.20.38
=> setenv serverip 192.168.1.1
=> setenv netmask 255.255.0.0
=> setenv hostname enbw_cmc
=> setenv rootpath /opt/eldk-5.0/armv5te/rootfs
=> saveenv
```

Then you can use these variables to build the boot arguments to be passed to the Linux kernel:

```
=> setenv nfsargs 'root=/dev/nfs rw nfsroot=${serverip}:${rootpath}'
```

Note how apostrophes are used to delay the substitution of the referenced environment variables. This way, the current values of these variables get inserted when assigning values to the "bootargs" variable itself later, i. e. when it gets assembled from the given parts before passing it to the kernel. This allows us to simply redefine any of the variables (say, the value of "ipaddr" if it has to be changed), and the changes will automatically propagate to the Linux kernel.

☐ **Note:** You cannot use this method **directly** to define for example the "bootargs" environment variable, as the implicit usage of this variable by the "bootm" command will **not** trigger variable expansion - this happens **only** when using the "setenv" command.

In the next step, this can be used for a flexible method to define the "bootargs" environment variable by using a function-like approach to build the boot arguments step by step:

```
=> setenv ramargs setenv bootargs root=/dev/ram rw
=> setenv nfsargs 'setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath}'
=> setenv addip 'setenv bootargs ${bootargs} ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}::off'
=> setenv ram_root 'run ramargs addip;bootm ${kernel_addr} ${ramdisk_addr} ${fdt_addr} '
=> setenv nfs_root 'run nfsargs addip;bootm ${kernel_addr} - ${fdt_addr} '
```

In this setup we define two variables, `ram_root` and `nfs_root`, to boot with root filesystem from a ramdisk image or over NFS, respectively. The variables can be executed using U-Boot's `run` command. These variables make use of the `run` command itself:

- First, either `run ramargs` or `run nfsargs` is used to initialize the `bootargs` environment variable as needed to boot with ramdisk image or with root over NFS.
- Then, in both cases, `run addip` is used to append the `ip` parameter to use the Linux kernel IP autoconfiguration mechanism for configuration of the network settings.
- Finally, the `bootm` command is used with three resp. two address arguments to boot the Linux kernel image with resp. without a ramdisk image. (We assume here that the variables `kernel_addr`, `ramdisk_addr` and `fdt_addr` have already been set.)

This method can be easily extended to add more customization options when needed.

If you have used U-Boot's network commands before (and/or read the documentation), you will probably have recognized that the names of the U-Boot environment variables we used in the examples above are exactly the same as those used with the U-Boot commands to boot over a network using [DHCP](#) or [BOOTP](#). That means that, instead of manually setting network configuration parameters like IP address, etc., these variables will be set automatically to the values retrieved with the network boot protocols. This is explained in detail in the sections about the respective U-Boot commands.

7.5. Networked Operation with Root Filesystem over NFS

This section will show how to boot the target into Linux with no more than U-Boot residing on it. For this we will use the `tftp` command of U-Boot to transfer a Linux kernel and boot it with the NFS rootfilesystem provided by the [ELDK](#).

For this to work, we rely on some U-Boot environment variables to be set up correctly, i.e. the network parameters, the names of files to transfer via tftp and last but not least some scripts easing the assembly of the Linux command line. The whole process is packaged up into one script shown before we actually execute it.

Note that the Linux kernel will also output the command line used, so you can easily check if everything worked like expected. The command line in this example passes at least the following information to the:

- `root=/dev/nfs rw`: the root filesystem will be mounted using NFS, and it will be writable.
- `nfsroot=192.168.1.1:/opt/eldk-5.0/armv5te/rootfs`: the NFS server has the IP address 192.168.1.1, and exports the directory `/opt/eldk-5.0/armv5te/rootfs` for our system to use as root filesystem.
- `ip=192.168.20.38:192.168.1.1:192.168.1.1:255.255.0.0:enbw_cmc::off`: the target has the IP address 192.168.20.38; the NFS server is 192.168.1.1; there is a gateway at IP address 192.168.1.1; the netmask is 255.255.0.0 and our hostname is `enbw_cmc`. The first ethernet interface (`eth0`) will be used, and the Linux kernel will immediately use this network configuration and not try to re-negotiate it (IP autoconfiguration is `off`).

See *Documentation/nfsroot.txt* in you Linux kernel source directory for more information about these parameters and other options.

7.5.1. Bootlog of tftp'd Linux kernel with Root Filesystem over NFS

```
=> setenv bootfile /tftpboot/duts/enbw_cmc/uImage
=> setenv rootpath /opt/eldk-5.0/armv5te/rootfs
=> printenv net_nfs
net_nfs=run load_fdt load_kernel; run nfsargs addip addcon addmtd;bootm ${kernel_addr_r} - ${fdt_addr_r}
=> run net_nfs
Using DaVinci-EMAC device
TFTP from server 192.168.1.1; our IP address is 192.168.20.40
Filename '/tftpboot/duts/enbw_cmc/enbw_cmc.dtb'.
Load address: 0xc0600000
Loading: #
done
Bytes transferred = 5258 (148a hex)
Using DaVinci-EMAC device
TFTP from server 192.168.1.1; our IP address is 192.168.20.40
Filename '/tftpboot/duts/enbw_cmc/uImage'.
Load address: 0xc0700000
Loading: #####
#####
#####
#####
done
Bytes transferred = 2285584 (22e010 hex)
## Booting kernel from Legacy Image at c0700000 ...
   Image Name:   Linux-3.1.0-00009-gd695872
   Created:      2011-11-23   7:05:36 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2285520 Bytes = 2.2 MiB
   Load Address: c0008000
   Entry Point:  c0008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at c0600000
   Booting using the fdt blob at 0xc0600000
   Loading Kernel Image ... OK
OK
Using machid 0xe01 from environment
   Loading Device Tree to c3e39000, end c3e3d489 ... OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
Linux version 3.1.0-00009-gd695872 (hs@pollux.denx.de) (gcc version 4.5.1 (GCC) ) #1 PREEMPT Wed Nov 23 08:05:33 CET 2011
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00053177
CPU: VIVT data cache, VIVT instruction cache
Machine: EnBW CMC, model: EnBW CMC
Memory policy: ECC disabled, Data cache writeback
DaVinci da850/omap-l138/aml8x variant 0x1
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 16256
Kernel command line: root=/dev/nfs rw nfsroot=192.168.1.1:/opt/eldk-5.0/armv5te/rootfs ip=192.168.20.40:192.168.1.1::255.255.0.0:enbw_cmc:eth0:off panic=1
PID hash table entries: 256 (order: -2, 1024 bytes)
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 64MB = 64MB total
Memory: 60384k/60384k available, 5152k reserved, 0K highmem
Virtual kernel memory layout:
   vector : 0xffff0000 - 0xffff1000   (   4 kB)
   fixmap : 0xffff0000 - 0xffffe000   (  896 kB)
   DMA     : 0xfff00000 - 0xfffe0000   (   14 MB)
   vmalloc : 0xc4800000 - 0xf0000000   ( 930 MB)
   lowmem  : 0xc0000000 - 0xc4000000   (   64 MB)
   modules : 0xbf000000 - 0xc0000000   (   16 MB)
   .text   : 0xc0008000 - 0xc03f81e0   (4033 kB)
   .init   : 0xc03f9000 - 0xc041f000   (   152 kB)
   .data   : 0xc0420000 - 0xc0445b40   (   151 kB)
   .bss    : 0xc0445b64 - 0xc046b3c8   (   151 kB)
SLUB: Genslabs=13, HWalign=32, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
Preemptible hierarchical RCU implementation.
NR_IRQS:245
Calibrating delay loop... 227.32 BogoMIPS (lpj=1136640)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 512
CPU: Testing write buffer coherency: ok
DaVinci: 144 gpio irqs
NET: Registered protocol family 16
bio: create slab <bio-0> at 0
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
Switching to clocksource timer0_1
Switched to NOHz mode on CPU #0
musb-hdrc: version 6.0, ?dma?, otg (peripheral+host)
Waiting for USB PHY clock good...
musb-hdrc musb-hdrc: USB OTG mode controller at fee00000 using PIO, IRQ 58
NET: Registered protocol family 2
IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
TCP established hash table entries: 2048 (order: 2, 16384 bytes)
TCP bind hash table entries: 2048 (order: 1, 8192 bytes)
TCP: Hash tables configured (established 2048 bind 2048)
```

```

TCP reno registered
UDP hash table entries: 256 (order: 0, 4096 bytes)
UDP-Lite hash table entries: 256 (order: 0, 4096 bytes)
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
EMAC: MII PHY configured, RMII PHY will not be functional
msgmni has been set to 117
io scheduler noop registered (default)
start plist test
end plist test
Serial: 8250/16550 driver, 3 ports, IRQ sharing disabled
serial8250.0: ttyS0 at MMIO 0x1c42000 (irq = 25) is a 16550A
serial8250.0: ttyS1 at MMIO 0x1d0c000 (irq = 53) is a 16550A
serial8250.0: ttyS2 at MMIO 0x1d0d000 (irq = 61) is a 16550A
console [ttyS2] enabled
created "/proc/driver/bootcount"
brd: module loaded
loop: module loaded
physmap platform flash device: 02000000 at 60000000
physmap-flash.0: Found 1 x16 devices at 0x0 in 16-bit bank. Manufacturer ID 0x000001 Chip ID 0x002249
Amd/Fujitsu Extended Query Table at 0x0040
  Amd/Fujitsu Extended Query version 1.3.
number of CFI chips: 1
Using physmap partition information
Creating 3 MTD partitions on "physmap-flash.0":
0x0000000000000-0x0000000080000 : "bootloaders"
0x0000000080000-0x00000000a0000 : "env + red"
0x00000000a0000-0x0000000200000 : "rest"
NAND device: Manufacturer ID: 0xad, Chip ID: 0xf1 (Hynix NAND 128MiB 3,3V 8-bit)
6 cmdlinepart partitions found on MTD device davinci_nand.1
Creating 6 MTD partitions on "davinci_nand.1":
0x0000000000000-0x00000000c0000 : "uboot"
0x00000000c0000-0x0000000100000 : "NVRAM"
0x0000000100000-0x0000000400000 : "k0"
0x0000000400000-0x0000000800000 : "r0"
0x0000000800000-0x0000000b00000 : "k1"
0x0000000b00000-0x0000000800000 : "newdata"
davinci_nand davinci_nand.1: controller rev. 2.5
Fixed MDIO Bus: probed
davinci_mdio davinci_mdio.0: davinci mdio revision 1.5
davinci_mdio davinci_mdio.0: detected phy mask ffffffff
davinci_mdio.0: probed
davinci_mdio davinci_mdio.0: phy[1]: device 0:01, driver unknown
davinci_mdio davinci_mdio.0: phy[2]: device 0:02, driver unknown
davinci_mdio davinci_mdio.0: phy[3]: device 0:03, driver unknown
davinci_mdio davinci_mdio.0: phy[4]: device 0:04, driver unknown
davinci_mdio davinci_mdio.0: phy[5]: device 0:05, driver unknown
usbcore: registered new interface driver uas
Initializing USB Mass Storage driver...
usbcore: registered new interface driver usb-storage
USB Mass Storage support registered.
usbcore: registered new interface driver libusual
  gadget: using random self ethernet address
  gadget: using random host ethernet address
usb0: MAC 4a:03:f1:0f:17:31
usb0: HOST MAC 0a:80:42:35:63:a4
  gadget: Ethernet Gadget, version: Memorial Day 2008
  gadget: g_ether ready
musb-hdrc musb-hdrc: MUSB HDRC host driver
musb-hdrc musb-hdrc: new USB bus registered, assigned bus number 1
usb usb1: New USB device found, idVendor=1d6b, idProduct=0002
usb usb1: New USB device strings: Mfr=3, Product=2, SerialNumber=1
usb usb1: Product: MUSB HDRC host driver
usb usb1: Manufacturer: Linux 3.1.0-00009-gd695872 musb-hcd
usb usb1: SerialNumber: musb-hdrc
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
omap_rtc omap_rtc: rtc core: registered omap_rtc as rtc0
omap_rtc: RTC power up reset detected
omap_rtc: already running
i2c /dev entries driver
lm75 1-0048: hwmon0: sensor 'lm75'
WD: Software Watchdog Timer 1.1.0 timeout 300 sec.
davinci_mmc davinci_mmc.1: Using PIO, 4-bit mode
TCP cubic registered
NET: Registered protocol family 10
IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
omap_rtc omap_rtc: setting system clock to 2000-01-01 00:04:12 UTC (946685052)
net eth0: attached PHY driver [Generic PHY] (mii_bus:phy_addr=1:00, id=0)
ADDRCONF(NETDEV_UP): eth0: link is not ready
mmc0: new SD card at address eabb
mmcblk0: mmc0:eabb SU02G 1.84 GiB
  mmcblk0: p1 p2 p3
PHY: 1:00 - Link is Up - 100/Full
ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
IP-Config: Complete:
    device=eth0, addr=192.168.20.40, mask=255.255.0.0, gw=255.255.255.255,
    host=enbw_cmc, domain=, nis-domain=(none),
    bootserver=192.168.1.1, rootserver=192.168.1.1, rootpath=
VFS: Mounted root (nfs filesystem) on device 0:13.
Freeing init memory: 152K
hub 1-0:1.0: unable to enumerate USB device on port 1
modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

usb 1-1: new high speed USB device number 3 using musb-hdrc
INIT: version 2.88 booting
usb 1-1: New USB device found, idVendor=13fe, idProduct=1f00
usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
usb 1-1: Product: DataTraveler 2.0
usb 1-1: Manufacturer: Kingston
usb 1-1: SerialNumber: 5B840400019B
scsi0 : usb-storage 1-1:1.0
modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

Error Cannot open /dev/tty0: No such device or address
modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

```

```

Please wait: booting...
modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

Error opening /dev/fb0: No such device or address
Starting udevscsi 0:0:0:0: Direct-Access Kingston DataTraveler 2.0 PMPAQ: 0 ANSI: 0 CCS
udev[936]: starting version 164
sd 0:0:0:0: [sda] 1953792 512-byte logical blocks: (1.00 GB/954 MiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] No Caching mode page present
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] No Caching mode page present
sd 0:0:0:0: [sda] Assuming drive cache: write through
sda: sdal
sd 0:0:0:0: [sda] No Caching mode page present
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] Attached SCSI removable disk

modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or dFAT-fs (sda): invalid media value (0x00)
irectory

FAT-fs (sda): Can't find a valid FAT filesystem

udevadm settle - timeout of 3 seconds reached, the event queue contains:
/sys/devices/platform/musb-da8xx/musb-hdrc/usbl/1-1/1-1:1.0/host0/target0:0:0:0:0:0/block/sda (950)
Configuring network interfaces... done.
Starting portmap daemon...
net.ipv4.conf.default.rp_filter = 1
net.ipv4.conf.all.rp_filter = 1
INIT: Entering runlevel: 5
Starting system message bus: Unknown username "xuser" in message bus configuration file
dbus.
Starting Connection Manager
Starting Xserver
modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

fbset: can't open '/dev/fb0': No such device or address
modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

fbset: can't open '/dev/fb0': No such device or address
modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

fbset: can't open '/dev/fb0': No such device or address

modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

modprobe: FATAL: Could not load /lib/modules/3.1.0-00009-gd695872/modules.dep: No such file or directory

Fatal server error:
LinuxInit: Cannot open /dev/tty0 (No such device or address)

xinit: giving up
xinit: unable to connect to X server: Connection refused
xinit: server error
Timeout waiting for org.matchbox_project.desktop.Loaded
Starting Dropbear SSH server: dropbear.
Starting Distributed Compiler Daemon: distcc.
Starting syslogd/klogd: done
* Starting Avahi mDNS/DNS-SD Daemon: avahi-daemon
A74G[ ok ]
Starting tcf-agent: OK
getty: ioctl() TIOCPGR
EDLK (Built by Poky 5.0) 5.0 armv5te console

armv5te login: root
-sh: can't access tty; job control turned off
root@armv5te:~#

```

7.6. Boot from NAND Flash Memory

The previous section described how to load the Linux kernel image over ethernet using [TFTP](#). This is especially well suited for your development and test environment, when the kernel image is still undergoing frequent changes, for instance because you are modifying kernel code or configuration.

Later in your development cycle you will work on application code or device drivers, which can be loaded dynamically as modules. If the Linux kernel remains the same then you can save the time needed for the [TFTP](#) download and put the kernel image into the NAND flash memory of your enbw_cmc board.

After having deleted the target flash area, you can download the Linux image and write it to flash. Below is a transcript of the complete operation with a final `iminfo` command to check the newly placed Linux kernel image in the flash memory.

```

=> setenv kernel_addr_r 0xc0700000
=> setenv nand_off 0x20000
=> nand erase 0x20000 0x300000

NAND erase: device 0 offset 0x20000, size 0x300000
Erasing at 0x20000 -- 4% complete.Erasing at 0x40000 -- 8% complete.Erasing at 0x60000 -- 12% complete.Erasing at 0x80000 -- 16% complete.Erasing at
OK
=> tftp 0xc0700000 /tftpboot/duts/enbw_cmc/uImage
Using DaVinci-EMAC device
TFTP from server 192.168.1.1; our IP address is 192.168.20.40
Filename '/tftpboot/duts/enbw_cmc/uImage'.
Load address: 0xc0700000
Loading: #####
#####
#####
done
Bytes transferred = 2285584 (22e010 hex)
=> iminfo 0xc0700000

## Checking Image at c0700000 ...
Legacy image found
Image Name: Linux-3.1.0-00009-gd695872
Created: 2011-11-23 7:05:36 UTC
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 2285520 Bytes = 2.2 MiB
Load Address: c0008000
Entry Point: c0008000

```



```

    Verifying Checksum ... OK
=> nand write ${kernel_addr_r} ${nand_off} ${filesize}

NAND write: device 0 offset 0x20000, size 0x22e010
2285584 bytes written: OK
=> setenv nand_off
=> saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
Protected 1 sectors
=>

```

Note how the `filesize` variable (which gets set by the [TFTP](#) transfer) is used to automatically adjust for the actual image size.

Since kernel requires the flattened device tree blob to be passed at boot time, you have to also write the blob to the flash memory. Below is a transcript of this operation.

```

=> setenv fdt_addr_r 0xc0600000
=> setenv nand_off 0x0
=> setenv nand_len 0x2000
=> setenv cmp_addr_r 0xc0000000
=> nand erase ${nand_off} ${nand_len}

NAND erase: device 0 offset 0x0, size 0x2000
Erasing at 0x0 -- 100% complete.
OK
=> tftp 0xc0600000 /tftpboot/duts/enbw_cmc/enbw_cmc.dtb
Using DaVinci-EMAC device
TFTP from server 192.168.1.1; our IP address is 192.168.20.40
Filename '/tftpboot/duts/enbw_cmc/enbw_cmc.dtb'.
Load address: 0xc0600000
Loading: #
done
Bytes transferred = 5258 (148a hex)
=> nand write ${fdt_addr_r} ${nand_off} ${filesize}

NAND write: device 0 offset 0x0, size 0x148a
5258 bytes written: OK
=> setenv nand_off
=> setenv nand_len
=> saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
Protected 1 sectors
=>

```

Now we can boot directly from flash. All we need to do is passing the in-flash address of the image (C0700000) and the in-flash address of the flattened device tree (C0600000) with the `bootm` command; we also make the definition of the `bootargs` variable permanent now:

```

=> setenv bootcmd bootm C0700000 - C0600000
=> setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath} ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}::off

```

Use `printenv` to verify that everything is OK before you save the environment settings:

```

=> printenv
bootdelay=5
baudrate=115200
stdin=serial
stdout=serial
stderr=serial
bootcmd=bootm C0700000 - C0600000
bootargs=root=/dev/nfs rw nfsroot=192.168.1.1:/opt/eldk-5.0/armv5te/rootfs
ip=192.168.20.38:192.168.1.1:192.168.1.1:255.255.0.0:enbw_cmc::off
....

=> saveenv

```

To test booting from flash you can now reset the board (either by power-cycling it, or using the U-Boot command `reset`), or you can manually call the `boot` command which will run the commands in the `bootcmd` variable:

```

=> run nand_selfnand

NAND read: device 0 offset 0x320000, size 0x400000
Bad block table found at page 65472, version 0x01
Bad block table found at page 65408, version 0x01
4194304 bytes read: OK

NAND read: device 0 offset 0x20000, size 0x300000
3145728 bytes read: OK

NAND read: device 0 offset 0x0, size 0x2000
8192 bytes read: OK
## Booting kernel from Legacy Image at c0700000 ...
  Image Name:   Linux-3.1.0-00009-gd695872
  Created:      2011-11-23   7:05:36 UTC
  Image Type:   ARM Linux Kernel Image (uncompressed)
  Data Size:    2285520 Bytes = 2.2 MiB
  Load Address: c0008000
  Entry Point:  c0008000
  Verifying Checksum ... OK
## Loading init Ramdisk from Legacy Image at c0b00000 ...
  Image Name:   Simple Embedded Linux Framework
  Created:      2008-11-24   16:44:17 UTC
  Image Type:   ARM Linux RAMDisk Image (gzip compressed)
  Data Size:    1659703 Bytes = 1.6 MiB
  Load Address: 00000000
  Entry Point:  00000000
  Verifying Checksum ... OK
## Flattened Device Tree blob at c0600000
   Booting using the fdt blob at 0xc0600000
   Loading Kernel Image ... OK
OK
Using machid 0xe01 from environment

```



```

Loading Ramdisk to c3ca8000, end c3e3d337 ... OK
Loading Device Tree to c3ca3000, end c3ca7489 ... OK

```

```
Starting kernel ...
```

```

Uncompressing Linux... done, booting the kernel.
Linux version 3.1.0-00009-gd695872 (hs@pollux.denx.de) (gcc version 4.5.1 (GCC) ) #1 PREEMPT Wed Nov 23 08:05:33 CET 2011
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00053177
CPU: VIVT data cache, VIVT instruction cache
Machine: EnBW CMC, model: EnBW CMC
Memory policy: ECC disabled, Data cache writeback
DaVinci da850/omap-1138/aml8x variant 0x1
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 16256
Kernel command line: root=/dev/ram rw ip=192.168.20.40:192.168.1.1::255.255.0.0:enbw_cmc:eth0:off panic=1 console=ttyS2,115200n8
PID hash table entries: 256 (order: -2, 1024 bytes)
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 64MB = 64MB total
Memory: 58760k/58760k available, 6776k reserved, 0K highmem
Virtual kernel memory layout:
 vector : 0xffff0000 - 0xffff1000 ( 4 kB)
 fixmap : 0xffff0000 - 0xffffe000 ( 896 kB)
 DMA : 0xff000000 - 0xffe00000 ( 14 MB)
 vmalloc : 0xc4800000 - 0xfea00000 ( 930 MB)
 lowmem : 0xc0000000 - 0xc4000000 ( 64 MB)
 modules : 0xbf000000 - 0xc0000000 ( 16 MB)
 .text : 0xc0008000 - 0xc03f81e0 (4033 kB)
 .init : 0xc03f9000 - 0xc041f000 ( 152 kB)
 .data : 0xc0420000 - 0xc0445b40 ( 151 kB)
 .bss : 0xc0445b64 - 0xc046b3c8 ( 151 kB)
SLUB: Genslabs=13, HWalign=32, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
Preemptible hierarchical RCU implementation.
NR_IRQS:245
Calibrating delay loop... 227.32 BogoMIPS (lpj=1136640)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 512
CPU: Testing write buffer coherency: ok
DaVinci: 144 gpio irqs
NET: Registered protocol family 16
bio: create slab <bio-0> at 0
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
Switching to clocksource timer0_1
Switched to NOHz mode on CPU #0
musb-hdrc: version 6.0, ?dma?, otg (peripheral+host)
Waiting for USB PHY clock good...
musb-hdrc musb-hdrc: USB OTG mode controller at fee00000 using PIO, IRQ 58
NET: Registered protocol family 2
IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
TCP established hash table entries: 2048 (order: 2, 16384 bytes)
TCP bind hash table entries: 2048 (order: 1, 8192 bytes)
TCP: Hash tables configured (established 2048 bind 2048)
TCP reno registered
UDP hash table entries: 256 (order: 0, 4096 bytes)
UDP-Lite hash table entries: 256 (order: 0, 4096 bytes)
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
Trying to unpack rootfs image as initramfs...
rootfs image is not initramfs (no cpio magic); looks like an initrd
Freeing initrd memory: 1620K
EMAC: MII PHY configured, RMII PHY will not be functional
msgmni has been set to 117
io scheduler noop registered (default)
start plist test
end plist test
Serial: 8250/16550 driver, 3 ports, IRQ sharing disabled
serial8250.0: ttyS0 at MMIO 0x1c42000 (irq = 25) is a 16550A
serial8250.0: ttyS1 at MMIO 0x1d0c000 (irq = 53) is a 16550A
serial8250.0: ttyS2 at MMIO 0x1d0d000 (irq = 61) is a 16550A
console [ttyS2] enabled
created "/proc/driver/bootcount"
brd: module loaded
loop: module loaded
physmap platform flash device: 02000000 at 60000000
physmap-flash.0: Found 1 x16 devices at 0x0 in 16-bit bank. Manufacturer ID 0x000001 Chip ID 0x002249
Amd/Fujitsu Extended Query Table at 0x0040
Amd/Fujitsu Extended Query version 1.3.
number of CFI chips: 1
Using physmap partition information
Creating 3 MTD partitions on "physmap-flash.0":
0x00000000000000-0x00000000800000 : "bootloaders"
0x00000000800000-0x00000000a00000 : "env + red"
0x00000000a00000-0x00000002000000 : "rest"
NAND device: Manufacturer ID: 0xad, Chip ID: 0xf1 (Hynix NAND 128MiB 3,3V 8-bit)
Creating 4 MTD partitions on "davinci_nand.1":
0x00000000000000-0x00000000200000 : "dtb"
0x00000000200000-0x00000003200000 : "kernel"
0x00000003200000-0x00000007200000 : "rootfs"
0x00000007200000-0x00000008000000 : "filesystem"
davinci_nand davinci_nand.1: controller rev. 2.5
Fixed MDIO Bus: probed
davinci_mdio davinci_mdio.0: davinci mdio revision 1.5
davinci_mdio davinci_mdio.0: detected phy mask fffffffc1
davinci_mdio.0: probed
davinci_mdio davinci_mdio.0: phy[1]: device 0:01, driver unknown
davinci_mdio davinci_mdio.0: phy[2]: device 0:02, driver unknown
davinci_mdio davinci_mdio.0: phy[3]: device 0:03, driver unknown
davinci_mdio davinci_mdio.0: phy[4]: device 0:04, driver unknown
davinci_mdio davinci_mdio.0: phy[5]: device 0:05, driver unknown
usbcore: registered new interface driver uas
Initializing USB Mass Storage driver...
usbcore: registered new interface driver usb-storage
USB Mass Storage support registered.
usbcore: registered new interface driver libusual
gadget: using random self ethernet address
gadget: using random host ethernet address
usb0: MAC c6:c9:12:c0:a2:55
usb0: HOST MAC 52:bb:d3:31:2f:6e
gadget: Ethernet Gadget, version: Memorial Day 2008
gadget: g_ether ready
musb-hdrc musb-hdrc: MUSB HDRC host driver
musb-hdrc musb-hdrc: new USB bus registered, assigned bus number 1
usb usb1: New USB device found, idVendor=1d6b, idProduct=0002

```

```

usb usb1: New USB device strings: Mfr=3, Product=2, SerialNumber=1
usb usb1: Product: MUSB HSDRC host driver
usb usb1: Manufacturer: Linux 3.1.0-00009-gd695872 musb-hcd
usb usb1: SerialNumber: musb-hdrc
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
omap_rtc omap_rtc: rtc core: registered omap_rtc as rtc0
omap_rtc: RTC power up reset detected
omap_rtc: already running
i2c /dev entries driver
lm75 1-0048: hwmon0: sensor 'lm75'
WD: Software Watchdog Timer 1.1.0 timeout 300 sec.
davinci_mmc davinci_mmc.1: Using PIO, 4-bit mode
TCP cubic registered
NET: Registered protocol family 10
IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
omap_rtc omap_rtc: setting system clock to 2011-11-23 10:24:11 UTC (1322043851)
net eth0: attached PHY driver [Generic PHY] (mii_bus:phy_addr=1:00, id=0)
ADDRCONF(NETDEV_UP): eth0: link is not ready
mmc0: new SD card at address eabb
mmcblk0: mmc0:eabb SU02G 1.84 GiB
mmcblk0: p1 p2 p3
PHY: 1:00 - Link is Up - 100/Full
ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
IP-Config: Complete:
    device=eth0, addr=192.168.20.40, mask=255.255.0.0, gw=255.255.255.255,
    host=enbw_cmc, domain=, nis-domain=(none),
    bootserver=192.168.1.1, rootserver=192.168.1.1, rootpath=
RAMDISK: gzip image found at block 0
hub 1-0:1.0: unable to enumerate USB device on port 1
usb 1-1: new high speed USB device number 3 using musb-hdrc
usb 1-1: New USB device found, idVendor=13fe, idProduct=1f00
usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
usb 1-1: Product: DataTraveler 2.0
usb 1-1: Manufacturer: Kingston
usb 1-1: SerialNumber: 5B840400019B
scsi0 : usb-storage 1-1:1.0
VFS: Mounted root (ext2 filesystem) on device 1:0.
Freeing init memory: 152K
init started: BusyBox v1.7.1 (2008-11-24 17:40:49 MET)
starting pid 894, tty '': '/etc/rc.sh'
starting pid 900, tty '': '/bin/sh'
starting pid 899, tty '': '/bin/application'
### Application running ...
~ #
~ #

```

-- [HeikoSchocher](#) - 04 Nov 2011

7.7. Standalone Operation with Ramdisk Image

When your application development is completed, you usually will want to run your Embedded System *standalone*, i. e. independent from external resources like NFS filesystems. Instead of mounting the root filesystem from a remote server you can use a compressed ramdisk image, which is stored in flash memory and loaded into RAM when the system boots.

Load the ramdisk image into RAM and write it to flash as follows:

```

=> setenv ramdisk_addr_r 0xc0b00000
=> setenv nand_off 0x320000
=> nand erase 0x320000 0x400000

NAND erase: device 0 offset 0x320000, size 0x400000
Erasing at 0x320000 -- 3% complete.Erasing at 0x340000 -- 6% complete.Erasing at 0x360000 -- 9% complete.Erasing at 0x380000 -- 12% complete.Erasing
OK
=> tftp 0xc0b00000 /tftpboot/duts/enbw_cmc/uRamdisk
Using DaVinci-EMAC device
TFTP from server 192.168.1.1: our IP address is 192.168.20.40
Filename '/tftpboot/duts/enbw_cmc/uRamdisk'.
Load address: 0xc0b00000
Loading: #####
          #####
done
Bytes transferred = 1659767 (195377 hex)
=> iminfo 0xc0b00000

## Checking Image at c0b00000 ...
Legacy image found
Image Name: Simple Embedded Linux Framework
Created: 2008-11-24 16:44:17 UTC
Image Type: ARM Linux RAMDisk Image (gzip compressed)
Data Size: 1659703 Bytes = 1.6 MiB
Load Address: 00000000
Entry Point: 00000000
Verifying Checksum ... OK
=> nand write ${ramdisk_addr_r} ${nand_off} ${filesize}

NAND write: device 0 offset 0x320000, size 0x195377
1659767 bytes written: OK
=> setenv nand_off
=> saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
Protected 1 sectors
=>

```

To tell the Linux kernel to use the ramdisk image as root filesystem you have to modify the command line arguments passed to the kernel, and to pass two arguments to the `bootm` command, the first is the memory address of the Linux kernel image, the second that of the ramdisk image:

```

=> run nand_selfnand

NAND read: device 0 offset 0x320000, size 0x400000
Bad block table found at page 65472, version 0x01
Bad block table found at page 65408, version 0x01
4194304 bytes read: OK

NAND read: device 0 offset 0x20000, size 0x300000
3145728 bytes read: OK

```

```

NAND read: device 0 offset 0x0, size 0x2000
8192 bytes read: OK
## Booting kernel from Legacy Image at c0700000 ...
   Image Name:   Linux-3.1.0-00009-gd695872
   Created:      2011-11-23   7:05:36 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2285520 Bytes = 2.2 MiB
   Load Address: c0008000
   Entry Point:  c0008000
   Verifying Checksum ... OK
## Loading init Ramdisk from Legacy Image at c0b00000 ...
   Image Name:   Simple Embedded Linux Framework
   Created:      2008-11-24   16:44:17 UTC
   Image Type:   ARM Linux RAMDisk Image (gzip compressed)
   Data Size:    1659703 Bytes = 1.6 MiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
## Flattened Device Tree blob at c0600000
   Booting using the fdt blob at 0xc0600000
   Loading Kernel Image ... OK

OK
Using machid 0xe01 from environment
   Loading Ramdisk to c3ca8000, end c3e3d337 ... OK
   Loading Device Tree to c3ca3000, end c3ca7489 ... OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
Linux version 3.1.0-00009-gd695872 (hs@pollux.denx.de) (gcc version 4.5.1 (GCC) ) #1 PREEMPT Wed Nov 23 08:05:33 CET 2011
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00053177
CPU: VIVT data cache, VIVT instruction cache
Machine: EnBW CMC, model: EnBW CMC
Memory policy: ECC disabled, Data cache writeback
DaVinci da850/omap-1138/aml8x variant 0x1
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 16256
Kernel command line: root=/dev/ram rw ip=192.168.20.40:192.168.1.1::255.255.0.0:enbw_cmc:eth0:off panic=1 console=ttyS2,115200n8
PID hash table entries: 256 (order: -2, 1024 bytes)
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 64MB = 64MB total
Memory: 58760k/58760k available, 6776k reserved, 0K highmem
Virtual kernel memory layout:
   vector : 0xffff0000 - 0xffff1000   (   4 kB)
   fixmap : 0xffff0000 - 0xffffe000   ( 896 kB)
   DMA     : 0xff000000 - 0xffe00000   (  14 MB)
   vmalloc : 0xc4800000 - 0xfea00000   ( 930 MB)
   lowmem  : 0xc0000000 - 0xc4000000   (  64 MB)
   modules : 0xbf000000 - 0xc0000000   (  16 MB)
   .text   : 0xc0008000 - 0xc03f81e0   (4033 kB)
   .init   : 0xc03f9000 - 0xc041f000   (  152 kB)
   .data   : 0xc0420000 - 0xc0445b40   (  151 kB)
   .bss    : 0xc0445b64 - 0xc046b3c8   (  151 kB)
SLUB: Genslabs=13, HWalign=32, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
Preemptible hierarchical RCU implementation.
NR_IRQS:245
Calibrating delay loop... 227.32 BogoMIPS (lpj=1136640)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 512
CPU: Testing write buffer coherency: ok
DaVinci: 144 gpio irqs
NET: Registered protocol family 16
bio: create slab <bio-0> at 0
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
Switching to clocksource timer0_1
Switched to NOHz mode on CPU #0
musb-hdrc: version 6.0, ?dma?, otg (peripheral+host)
Waiting for USB PHY clock good...
musb-hdrc musb-hdrc: USB OTG mode controller at fee00000 using PIO, IRQ 58
NET: Registered protocol family 2
IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
TCP established hash table entries: 2048 (order: 2, 16384 bytes)
TCP bind hash table entries: 2048 (order: 1, 8192 bytes)
TCP: Hash tables configured (established 2048 bind 2048)
TCP reno registered
UDP hash table entries: 256 (order: 0, 4096 bytes)
UDP-Lite hash table entries: 256 (order: 0, 4096 bytes)
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
Trying to unpack rootfs image as initramfs...
rootfs image is not initramfs (no cpio magic); looks like an initrd
Freeing initrd memory: 1620K
EMAC: MII PHY configured, RMII PHY will not be functional
msgmni has been set to 117
io scheduler noop registered (default)
start plst test
end plst test
Serial: 8250/16550 driver, 3 ports, IRQ sharing disabled
serial8250.0: ttyS0 at MMIO 0x1c42000 (irq = 25) is a 16550A
serial8250.0: ttyS1 at MMIO 0x1d0c000 (irq = 53) is a 16550A
serial8250.0: ttyS2 at MMIO 0x1d0d000 (irq = 61) is a 16550A
console [ttyS2] enabled
created "/proc/driver/bootcount"
brd: module loaded
loop: module loaded
physmap platform flash device: 02000000 at 60000000
physmap-flash.0: Found 1 x16 devices at 0x0 in 16-bit bank. Manufacturer ID 0x000001 Chip ID 0x002249
Amd/Fujitsu Extended Query Table at 0x0040
   Amd/Fujitsu Extended Query version 1.3.
number of CFI chips: 1
Using physmap partition information
Creating 3 MTD partitions on "physmap-flash.0":
0x0000000000000-0x0000000080000 : "bootloaders"
0x0000000080000-0x00000000a0000 : "env + red"
0x00000000a0000-0x0000000200000 : "rest"
NAND device: Manufacturer ID: 0xad, Chip ID: 0xf1 (Hynix NAND 128MiB 3,3V 8-bit)
Creating 4 MTD partitions on "davinci_nand.1":
0x0000000000000-0x0000000020000 : "dtb"
0x0000000020000-0x0000000320000 : "kernel"
0x0000000320000-0x0000000720000 : "rootfs"
0x0000000720000-0x0000008000000 : "filesystem"
davinci_nand davinci_nand.1: controller rev. 2.5

```

```

Fixed MDIO Bus: probed
davinci_mdio davinci_mdio.0: davinci mdio revision 1.5
davinci_mdio davinci_mdio.0: detected phy mask fffffffc1
davinci_mdio.0: probed
davinci_mdio davinci_mdio.0: phy[1]: device 0:01, driver unknown
davinci_mdio davinci_mdio.0: phy[2]: device 0:02, driver unknown
davinci_mdio davinci_mdio.0: phy[3]: device 0:03, driver unknown
davinci_mdio davinci_mdio.0: phy[4]: device 0:04, driver unknown
davinci_mdio davinci_mdio.0: phy[5]: device 0:05, driver unknown
usbcore: registered new interface driver uas
Initializing USB Mass Storage driver...
usbcore: registered new interface driver usb-storage
USB Mass Storage support registered.
usbcore: registered new interface driver libusual
  gadget: using random self ethernet address
  gadget: using random host ethernet address
usb0: MAC c6:c9:12:c0:a2:55
usb0: HOST MAC 52:bb:d3:31:2f:6e
  gadget: Ethernet Gadget, version: Memorial Day 2008
  gadget: g_ether ready
musb-hdrc musb-hdrc: MUSB HDRC host driver
musb-hdrc musb-hdrc: new USB bus registered, assigned bus number 1
usb usb1: New USB device found, idVendor=1d6b, idProduct=0002
usb usb1: New USB device strings: Mfr=3, Product=2, SerialNumber=1
usb usb1: Product: MUSB HDRC host driver
usb usb1: Manufacturer: Linux 3.1.0-00009-gd695872 musb-hcd
usb usb1: SerialNumber: musb-hdrc
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
omap_rtc omap_rtc: rtc core: registered omap_rtc as rtc0
omap_rtc: RTC power up reset detected
omap_rtc: already running
i2c /dev entries driver
lm75 1-0048: hwmmon0: sensor 'lm75'
WD: Software Watchdog Timer 1.1.0 timeout 300 sec.
davinci_mmc davinci_mmc.1: Using PIO, 4-bit mode
TCP cubic registered
NET: Registered protocol family 10
IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
omap_rtc omap_rtc: setting system clock to 2011-11-23 10:24:11 UTC (1322043851)
net eth0: attached PHY driver [Generic PHY] (mii_bus:phy_addr=1:00, id=0)
ADDRCONF(NETDEV_UP): eth0: link is not ready
mmc0: new SD card at address eabb
mmcblk0: mmc0:eabb SU02G 1.84 GiB
  mmcblk0: p1 p2 p3
PHY: 1:00 - Link is Up - 100/Full
ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
IP-Config: Complete:
    device=eth0, addr=192.168.20.40, mask=255.255.0.0, gw=255.255.255.255,
    host=enbw_cmc, domain=, nis-domain=(none),
    bootserver=192.168.1.1, rootserver=192.168.1.1, rootpath=
RAMDISK: gzip image found at block 0
hub 1-0:1.0: unable to enumerate USB device on port 1
usb 1-1: new high speed USB device number 3 using musb-hdrc
usb 1-1: New USB device found, idVendor=13fe, idProduct=1f00
usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
usb 1-1: Product: DataTraveler 2.0
usb 1-1: Manufacturer: Kingston
usb 1-1: SerialNumber: 5B840400019B
scsi0 : usb-storage 1-1:1.0
VFS: Mounted root (ext2 filesystem) on device 1:0.
Freeing init memory: 152K
init started: BusyBox v1.7.1 (2008-11-24 17:40:49 MET)
starting pid 894, tty '': '/etc/rc.sh'
starting pid 900, tty '': '/bin/sh'
starting pid 899, tty '': '/bin/application'
### Application running ...
~ #
~ #

```

8. Building and Using Modules

This section still needs to be written (this is a wiki, so please feel free to contribute!).

In the meantime, please refer to file [documentation/kbuild/modules.txt](#) in the Linux source tree.

- [9. Advanced Topics](#)
 - [9.1. Flash Filesystems](#)
 - [9.1.1. Memory Technology Devices](#)
 - [9.1.2. Journalling Flash File System](#)
 - [9.1.3. Second Version of JFFS](#)
 - [9.1.4. Compressed ROM Filesystem](#)
 - [9.1.5. UBI and UBIFS file systems](#)
 - [9.1.5.1. Create Device Files](#)
 - [9.1.5.2. Using UBI on NAND Flash:](#)
 - [9.1.5.3. Creating UBIFS File System Images](#)
 - [9.1.5.3.1. Determining the Parameters of the used Flash Types:](#)
 - [9.1.5.3.2. Create some Test File System Hierarchy](#)
 - [9.1.5.3.3. Installing UBIFS images into existing UBI Volume:](#)
 - [9.1.5.3.4. Installing UBI images \(if no UBI Volumes exist\):](#)
- [9.2. The TMPFS Virtual Memory Filesystem](#)
 - [9.2.1. Mount Parameters](#)
 - [9.2.2. Kernel Support for tmpfs](#)
 - [9.2.3. Usage of tmpfs in Embedded Systems](#)
- [9.3. Using MultiMediaCards in Linux"](#)
- [9.4. Splash Screen Support in Linux](#)
- [9.5. Root File System: Design and Building](#)
 - [9.5.1. Root File System on a Ramdisk](#)
 - [9.5.2. Root File System on a JFFS2 File System](#)
 - [9.5.3. Root File System on a cramfs File System](#)
 - [9.5.4. Root File System on a Read-Only ext2 File System](#)
 - [9.5.5. Root File System on a Flash Card](#)
 - [9.5.6. Root File System in a Read-Only File in a FAT File System](#)
- [9.6. Root File System Selection](#)
- [9.7. Overlay File Systems](#)
- [9.8. The Persistent RAM File system \(PRAMFS\)](#)
 - [9.8.1. Mount Parameters](#)
 - [9.8.2. Example](#)

9. Advanced Topics


This section lists some advanced topics of interest to users of U-Boot and Linux.

9.1. Flash Filesystems

9.1.1. Memory Technology Devices

All currently available flash filesystems are based on the Memory Technology Devices **MTD** layer, so you must enable (at least) the following configuration options to get flash filesystem support in your system:

```
CONFIG_MTD=y
CONFIG_MTD_PARTITIONS=y
CONFIG_MTD_CHAR=y
CONFIG_MTD_BLOCK=y
CONFIG_MTD_CFI=y
CONFIG_MTD_GEN_PROBE=y
CONFIG_MTD_CFI_AMDSTD=y
CONFIG_MTD_ROM=y
CONFIG_MTD_enbw_cmc=y
```

 Note: this configuration uses **CFI** conformant AMD flash chips; you may need to adjust these settings on other boards.

The partition layout of the flash devices is contained in the flat device tree for the system (see [13.1. Flat Device Tree](#)).

Informational messages of the MTD subsystem can be found in the Linux bootlog, i.e. see section [7.5.1. Bootlog of tftp'd Linux kernel with Root Filesystem over NFS](#).

One can discover this information in a running system using the `proc` filesystem:

```
root@armv5te:~# cat /proc/mtd
dev:      size  erasesize  name
mtd0: 00080000 00010000 "bootloaders"
mtd1: 00020000 00010000 "env + red"
mtd2: 00160000 00010000 "rest"
mtd3: 000c0000 00020000 "uboot"
mtd4: 00040000 00020000 "NVRAM"
mtd5: 00300000 00020000 "k0"
mtd6: 01400000 00020000 "x0"
mtd7: 00300000 00020000 "k1"
mtd8: 06500000 00020000 "newdata"
root@armv5te:~#
```

Now we can run some basic tests to verify that the flash driver routines and the partitioning works as expected:

```
root@armv5te:~# hexdump -C /dev/mtd3 | head -4
00000000  53 61 74 20 4a 61 6e 20 20 31 20 30 30 3a 32 33  |Sat Jan  1 00:23|
00000010  3a 35 34 20 4d 45 54 20 32 30 30 30 0a 00 00 00  |:54 MET 2000....|
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |.....|
*
root@armv5te:~#
```

In the hex-dumps of the [MTD](#) devices you can identify some strings that verify that we indeed see an U-Boot environment, a Linux kernel, a ramdisk image and an empty partition to play with.

The last output shows the partition to be empty. We can try write some data into it:

```
root@armv5te:~# date > /tmp/tempfile
root@armv5te:~# dd if=/dev/zero of=/tmp/tempfile bs=1 count=4096 seek=50
4096+0 records in
4096+0 records out
4096 bytes (4.1 kB) copied, 0.0767632 s, 53.4 kB/s
root@armv5te:~# dd if=/tmp/tempfile of=/dev/mtd3 bs=4096 count=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 0.000950875 s, 4.3 MB/s
root@armv5te:~# head -1 /dev/mtd3
Wed Nov 23 09:34:35 MET 2011
root@armv5te:~# dd if=/tmp/tempfile of=/dev/mtd3 bs=4096 count=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 0.000756458 s, 5.4 MB/s
root@armv5te:~#
```

As you can see it worked the first time. When we tried to write the (new date) again, we got an error. The reason is that the date has changed (probably at least the seconds) and flash memory cannot be simply overwritten - it has to be erased first.

You can use the `eraseall` Linux commands to erase a whole [MTD](#) partition:

```
root@armv5te:~# flash_erase -q /dev/mtd3 0 0
root@armv5te:~# date > /tmp/tempfile
root@armv5te:~# dd if=/dev/zero of=/tmp/tempfile bs=1 count=4096 seek=50
4096+0 records in
4096+0 records out
4096 bytes (4.1 kB) copied, 0.0824337 s, 49.7 kB/s
root@armv5te:~# dd if=/tmp/tempfile of=/dev/mtd3 bs=4096 count=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 0.00093767 s, 4.4 MB/s
root@armv5te:~# head -1 /dev/mtd3
Wed Nov 23 09:34:23 MET 2011
root@armv5te:~#
```

We have now sufficient proof that the [MTD](#) layer is working as expected, so we can try creating a flash filesystem.

9.1.2. Journalling Flash File System

At the moment it seems that the Journalling Flash File System **JFFS** is the best choice for filesystems in flash memory of embedded devices. You must enable the following configuration options to get [JFFS](#) support in your system:

```
CONFIG_JFFS_FS=y
CONFIG_JFFS_FS_VERBOSE=0
```

If the flash device is erased, we can simply mount it, and the creation of the [JFFS](#) filesystem is performed automagically.

☐ Note: For simple accesses like direct read or write operations or erasing you use the *character* device interface (*/dev/mtd**) of the [MTD](#) layer, while for filesystem operations like mounting we must use the *block* device interface (*/dev/mtdblock**).

```
# eraseall /dev/mtd2
Erased 4096 Kibyte @ 0 -- 100% complete.
# mount -t jffs /dev/mtdblock2 /mnt
# mount
/dev/root on / type nfs (rw,v2,rsize=4096,wsiz=4096,hard,udp,nolock,addr=10.0.0.2)
proc on /proc type proc (rw)
devpts on /dev/pts type devpts (rw)
/dev/mtdblock2 on /mnt type jffs (rw)
# df
Filesystem            1k-blocks      Used Available Use% Mounted on
/dev/root              2087212      1232060    855152   60% /
/dev/mtdblock2         3584          0        3584    0% /mnt
```

Now you can access the files in the [JFFS](#) filesystem in the */mnt* directory.

9.1.3. Second Version of [JFFS](#)

Probably even more interesting for embedded systems is the second version of [JFFS](#), **JFFS2**, since it not only fixes a few design issues with [JFFS](#), but also adds transparent compression, so that you can save a lot of precious flash memory.

The `mkfs.jffs2` tool is used to create a JFFS2 filesystem image; it populates the image with files from a given directory. For instance, to create a JFFS2 image for a flash partition of 3 MB total size and to populate it with the files from the */tmp/flashtools* directory you would use:

```
# mkfs.jffs2 --pad=3145728 --eraseblock=262144 \
--root=/tmp/flashtools/ --output image.jffs2
# eraseall /dev/mtd4
Erased 3072 Kibyte @ 0 -- 100% complete.
\# dd if=image.jffs2 of=/dev/mtd4 bs=256k
12+0 records in
12+0 records out
# mount -t jffs2 /dev/mtdblock4 /mnt
# df /mnt
Filesystem            1k-blocks      Used Available Use% Mounted on
/dev/mtdblock4        3072        2488      584   81% /mnt
```

☐ Note: Especially when you are running time-critical applications on your system you should carefully study if the behaviour of the flash filesystem might have any negative impact on your application. After all, a flash device is not a normal harddisk. This is especially important when your flash filesystem gets full; JFFS2 acts a bit weird then:

- You will note that an increasing amount of [CPU](#) time is spent by the filesystem's garbage collection kernel thread.
- Access times to the files on the flash filesystem may increase drastically.
- Attempts to truncate a file (to free space) or to rename it may fail:

```
...
# cp /bin/bash file
cp: writing `file': No space left on device
# >file
bash: file: No space left on device
# mv file foo
mv: cannot create regular file `foo': No space left on device
```

You will have to use `rm` to actually delete a file in this situation.

This is especially critical when you are using the flash filesystem to store log files: when your application detects some abnormal condition and produces lots of log messages (which usually are especially important in this situation) the filesystem may fill up and cause *extreme* long delays - if your system crashes, the most important messages may never be logged at all.

9.1.4. Compressed ROM Filesystem

In some cases it is sufficient to have read-only access to some files, and if the files are big enough it becomes desirable to use some method of compression. The Compressed ROM Filesystem **CramFs** might be a solution here.

☐ Please note that [CramFs](#) has - beside the fact that it is a read-only filesystem - some severe limitations (like missing support for timestamps, hard links, and 16/32 bit uid/gids), but there are many situations in Embedded Systems where it's still useful.

To create a [CramFs](#) filesystem a special tool `mkcramfs` is used to create a file which contains the [CramFs](#) image. Note that the [CramFs](#) filesystem can be written and read only by kernels with `PAGE_CACHE_SIZE == 4096`, and some versions of the `mkcramfs` program may have other restrictions like that the filesystem must be written and read with architectures of the same endianness. Especially the endianness requirement makes it impossible to build the [CramFs](#) image on x86 PC host when you want to use it on a [Power Architecture](#)® target. The endianness problem has been fixed in the version of `mkcramfs` that comes with the [ELDK](#).

In some cases you can use a target system running with root filesystem mounted over NFS to create the [CramFs](#) image on the native system and store it to flash for further use.

☐ Note: The normal version of the `mkcramfs` program tries to initialize some entries in the filesystem's superblock with random numbers by reading */dev/random*; this may hang permanently on your target because there is not enough input (like mouse movement) to the entropy pool. You may want to use a modified version of `mkcramfs` which does not depend on */dev/random*.

To create a [CramFs](#) image, you put all files you want in the filesystem into one directory, and then use the `mkcramfs` program as follows:

```
$ mkdir /tmp/test
$ cp ... /tmp/test
$ du -sk /tmp/test
64      /tmp/test
$ mkcramfs /tmp/test test.cramfs.img
Super block: 76 bytes
  erase
  eraseall
  mkfs.jffs
  lock
  unlock
Directory data: 176 bytes
-54.96% (-4784 bytes)  erase
-55.46% (-5010 bytes)  eraseall
-51.94% (-8863 bytes)  mkfs.jffs
-58.76% (-4383 bytes)  lock
-59.68% (-4215 bytes)  unlock
Everything: 24 kilobytes
$ ls -l test.cramfs.img
-rw-r--r--    1 wd      users          24576 Nov 10 23:44 test.cramfs.img
```

As you can see, the [CramFs](#) image *test.cramfs.img* takes just 24 kB, while the input directory contained 64 kB of data. Savings of some 60% like in this case are typical [CramFs](#).

Now we write the [CramFs](#) image to a partition in flash and test it:

```
# cp test.cramfs.img /dev/mtd3
# mount -t cramfs /dev/mtdblock3 /mnt
# mount
/dev/root on / type nfs (rw,v2,rsize=4096,wsiz=4096,hard,udp,nolock,addr=10.0.0.2)
proc on /proc type proc (rw)
devpts on /dev/pts type devpts (rw)
/dev/mtdblock3 on /mnt type cramfs (rw)
# ls -l /mnt
total 54
-rwxr-xr-x 1 wd      users      8704 Jan  9 16:32 erase
-rwxr-xr-x 1 wd      users      9034 Jan  1 01:00 eraseall
-rwxr-xr-x 1 wd      users      7459 Jan  1 01:00 lock
-rwxr-xr-x 1 wd      users     17063 Jan  1 01:00 mkfs.jffs
-rwxr-xr-x 1 wd      users      7063 Jan  1 01:00 unlock
```

Note that all the timestamps in the [CramFs](#) filesystems are bogus, and so is for instance the output of the `df` command for such filesystems:

```
# df /mnt
Filesystem      1k-blocks      Used Available Use% Mounted on
/dev/mtdblock3          0          0          0  -  /mnt
```

9.1.5. UBI and UBIFS file systems

UBIFS is a flash filesystem, which work on top of the Linux [MTD](#) layer. UBI itself is a software layer which basically is a volume management and wear-leveling layer. It provides so called UBI volumes which is a higher level abstraction than a [MTD](#) device.

For more documentation about UBI/UBIFS see:

- linux source:Documentation/filesystems/ubifs.txt
- <http://www.linux-mtd.infradead.org/doc/ubi.html>

This document illustrates the usage of UBI/UBIFS for the `enbw_cmc` board.

9.1.5.1. Create Device Files

First we have to create some device files, which are necessary for using UBI/UBIFS:

```
root@armv5te:~# mknod /dev/ubi_ctrl c 10 63
root@armv5te:~# mknod /dev/ubi0 c 253 0
root@armv5te:~# for i in $(seq 0 9); do mknod /dev/ubi0_$i c 253 $((i + 1)); done
root@armv5te:~# mknod /dev/ubil c 252 0
root@armv5te:~# for i in $(seq 0 9); do mknod /dev/ubil_$i c 252 $((i + 1)); done
root@armv5te:~# ls -l /dev/ubi*
crw-r--r-- 1 root root 253, 0 Nov 23 09:38 /dev/ubi0
crw-r--r-- 1 root root 253, 1 Nov 23 09:38 /dev/ubi0_0
crw-r--r-- 1 root root 253, 2 Nov 23 09:38 /dev/ubi0_1
crw-r--r-- 1 root root 253, 3 Nov 23 09:38 /dev/ubi0_2
crw-r--r-- 1 root root 253, 4 Nov 23 09:38 /dev/ubi0_3
crw-r--r-- 1 root root 253, 5 Nov 23 09:38 /dev/ubi0_4
crw-r--r-- 1 root root 253, 6 Nov 23 09:38 /dev/ubi0_5
crw-r--r-- 1 root root 253, 7 Nov 23 09:38 /dev/ubi0_6
crw-r--r-- 1 root root 253, 8 Nov 23 09:38 /dev/ubi0_7
crw-r--r-- 1 root root 253, 9 Nov 23 09:38 /dev/ubi0_8
crw-r--r-- 1 root root 253, 10 Nov 23 09:38 /dev/ubi0_9
crw-r--r-- 1 root root 252, 0 Nov 23 09:38 /dev/ubil
crw-r--r-- 1 root root 252, 1 Nov 23 09:38 /dev/ubil_0
crw-r--r-- 1 root root 252, 2 Nov 23 09:38 /dev/ubil_1
crw-r--r-- 1 root root 252, 3 Nov 23 09:38 /dev/ubil_2
crw-r--r-- 1 root root 252, 4 Nov 23 09:38 /dev/ubil_3
crw-r--r-- 1 root root 252, 5 Nov 23 09:38 /dev/ubil_4
crw-r--r-- 1 root root 252, 6 Nov 23 09:38 /dev/ubil_5
crw-r--r-- 1 root root 252, 7 Nov 23 09:38 /dev/ubil_6
crw-r--r-- 1 root root 252, 8 Nov 23 09:38 /dev/ubil_7
crw-r--r-- 1 root root 252, 9 Nov 23 09:38 /dev/ubil_8
crw-r--r-- 1 root root 252, 10 Nov 23 09:38 /dev/ubil_9
crw-r--r-- 1 root root 10, 63 Nov 23 09:38 /dev/ubi_ctrl
root@armv5te:~#
```

9.1.5.2. Using UBI on NAND Flash:

Erase the flash partition:

```
root@armv5te:~# flash_erase -q /dev/mtd7 0 0
root@armv5te:~#
```

and attach it to UBI:

```
root@armv5te:~# ubiattach /dev/ubi_ctrl -m 7 -O 2048
UBI: attaching mtd7 to ubi0
UBI: physical eraseblock size: 131072 bytes (128 KiB)
UBI: logical eraseblock size: 126976 bytes
UBI: smallest flash I/O unit: 2048
UBI: sub-page size: 512
UBI: VID header offset: 2048 (aligned 2048)
UBI: data offset: 4096
UBI: empty MTD device detected
UBI: max. sequence number: 0
UBI: create volume table (copy #1)
UBI: create volume table (copy #2)
UBI: attached mtd7 to ubi0
UBI: MTD device name: "userfs"
UBI: MTD device size: 120 MiB
UBI: number of good PEBs: 963
UBI: number of bad PEBs: 4
UBI: number of corrupted PEBs: 0
UBI: max. allowed volumes: 128
UBI: wear-leveling threshold: 4096
UBI: number of internal volumes: 1
UBI: number of user volumes: 0
UBI: available PEBs: 950
UBI: total number of reserved PEBs: 13
UBI: number of PEBs reserved for bad PEB handling: 9
UBI: max/mean erase counter: 0/0
UBI: image sequence number: 1499480099
UBI: background thread "ubi_bgt0d" started, PID 1239
UBI device number 0, total 963 LEBs (122277888 bytes, 116.6 MiB), available 950 LEBs (120627200 bytes, 115.0 MiB), LEB size 126976 bytes (124.0 KiB)
root@armv5te:~#
```

As this is done, we check if things are done correct:

```
root@armv5te:~# ubinfo
UBI version:          1
Count of UBI devices: 1
UBI control device major/minor: 10:63
Present UBI devices:  ubi0
root@armv5te:~#
```

Create a Volume on the UBI Device:

```
root@armv5te:~# ubimkvol /dev/ubi0 -N userfs -m
Set volume size to 120627200
Volume ID 0, size 950 LEBs (120627200 bytes, 115.0 MiB), LEB size 126976 bytes (124.0 KiB), dynamic, name "userfs", alignment 1
root@armv5te:~#
```

As we intend to create just a single volume, we use maximum size ("-m" option).

Mount and use it:

```
root@armv5te:~# mkdir /mnt/userfs
root@armv5te:~# mount -t ubifs /dev/ubi0_0 /mnt/userfs
UBIFS: default file-system created
UBIFS: mounted UBI device 0, volume 0, name "userfs"
UBIFS: file system size: 119357440 bytes (116560 KiB, 113 MiB, 940 LEBs)
UBIFS: journal size: 5967872 bytes (5828 KiB, 5 MiB, 47 LEBs)
UBIFS: media format: w4/r0 (latest is w4/r0)
UBIFS: default compressor: lzo
UBIFS: reserved for root: 4952683 bytes (4836 KiB)
root@armv5te:~#
```

Check with "-a" option:

```
root@armv5te:~# ubinfo -a
UBI version:          1
Count of UBI devices: 1
UBI control device major/minor: 10:63
Present UBI devices:  ubi0

ubi0
Volumes count:        1
Logical eraseblock size: 126976 bytes, 124.0 KiB
Total amount of logical eraseblocks: 963 (122277888 bytes, 116.6 MiB)
Amount of available logical eraseblocks: 0 (0 bytes)
Maximum count of volumes 128
Count of bad physical eraseblocks: 4
Count of reserved physical eraseblocks: 9
Current maximum erase counter value: 2
Minimum input/output unit size: 2048 bytes
Character device major/minor: 253:0
Present volumes:      0

Volume ID: 0 (on ubi0)
Type:      dynamic
Alignment: 1
Size:      950 LEBs (120627200 bytes, 115.0 MiB)
State:     OK
Name:      userfs
Character device major/minor: 253:1
root@armv5te:~#
```

List which partitions we have mounted, and how many space we have available:

```
root@armv5te:~# df -h
Filesystem                Size      Used Available Use% Mounted on
192.168.1.1:/opt/eldk-5.0/armv5te/rootfs/
315.0G    109.9G    189.1G    37% /
tmpfs                40.0K          0    40.0K    0% /mnt/.psplash
none                 29.6M    100.0K    29.5M    0% /dev
tmpfs                29.6M    40.0K    29.5M    0% /var/volatile
tmpfs                29.6M          0    29.6M    0% /media/ram
/dev/sdal             952.1M    756.6M    195.5M    79% /media/sdal
/dev/ubi0_0           104.6M     24.0K    99.9M    0% /mnt/userfs
root@armv5te:~#
```

9.1.5.3. Creating UBIFS File System Images

9.1.5.3.1. Determining the Parameters of the used Flash Types:

The "mkfs.ubifs" requires a few parameters that describe the specific features of the underlying flash chips. The easiest way to determine these parameters is to run the "mtdinfo" utility on a running Linux system./dev/mtd7 is NAND flash:

```
root@armv5te:~# cat /proc/mtd
dev:   size  erasesize name
mtd0: 00080000 00010000 "U-Boot"
mtd1: 00010000 00010000 "env1"
mtd2: 00010000 00010000 "env2"
mtd3: 00160000 00010000 "rest"
mtd4: 00020000 00020000 "dtb"
mtd5: 00300000 00020000 "kernel"
mtd6: 00400000 00020000 "rootfs"
mtd7: 078e0000 00020000 "userfs"
root@armv5te:~#
```

```
root@armv5te:~# mtdinfo -u /dev/mtd7
mtd7
Name:      userfs
Type:      nand
Eraseblock size: 131072 bytes, 128.0 KiB
Amount of eraseblocks: 967 (126746624 bytes, 120.9 MiB)
Minimum input/output unit size: 2048 bytes
Sub-page size: 512 bytes
OOB size: 64 bytes
Character device major/minor: 90:14
Bad blocks are allowed: true
Device is writable: true
Default UBI VID header offset: 512
Default UBI data offset: 2048
Default UBI LEB size: 129024 bytes, 126.0 KiB
Maximum UBI volumes count: 128
```

```
root@armv5te:~# ubinfo /dev/ubi0
```



```
ubi0
Volumes count: 1
Logical eraseblock size: 126976 bytes, 124.0 KiB
Total amount of logical eraseblocks: 963 (122277888 bytes, 116.6 MiB)
Amount of available logical eraseblocks: 0 (0 bytes)
Maximum count of volumes 128
Count of bad physical eraseblocks: 4
Count of reserved physical eraseblocks: 9
Current maximum erase counter value: 2
Minimum input/output unit size: 2048 bytes
Character device major/minor: 253:0
Present volumes: 0
root@armv5te:~#
```

The interesting parameters are:

```
- min-io-size: corresponds to "Minimum input/output unit size"
- max-leb-cnt: corresponds to "Amount of eraseblocks"
```

One more needed parameter from the "ubininfo -a" command:

```
- leb-size: corresponds to "Logical eraseblock size"
```

9.1.5.3.2. Create some Test File System Hierarchy

```
[hs@pollux]$ cd /tmp
[hs@pollux]$ cd duts
[hs@pollux]$ mkdir fs
[hs@pollux]$ echo Just an example >fs/README
[hs@pollux]$ date >fs/date_of_creation
[hs@pollux]$ ls -l fs
insgesamt 8
-rw-rw-r-- 1 hs hs 29 23. Nov 09:39 date_of_creation
-rw-rw-r-- 1 hs hs 16 23. Nov 09:39 README
[hs@pollux]$
```

Create UBIFS Images for ""userfs"" (NAND)

```
[hs@pollux]$ mkfs.ubifs --root=/tmp/duts/fs --min-io-size=2048 --leb-size=126976 --max-leb-cnt=950 -o /tmp/duts/image-userfs.ubifs
[hs@pollux]$ ls -lh /tmp/duts/image-userfs.ubifs
-rw-rw-r-- 1 hs hs 1,7M 23. Nov 09:39 /tmp/duts/image-userfs.ubifs
[hs@pollux]$ cp /tmp/duts/image-userfs.ubifs /opt/eldk-5.0/armv5te/rootfs/home/duts/
[hs@pollux]$
```

9.1.5.3.3. Installing UBIFS images into existing UBI Volume:

```
root@armv5te:~# ubiattach /dev/ubi_ctrl -m 7 -O 2048
UBI: attaching mtd7 to ubi0
UBI: physical eraseblock size: 131072 bytes (128 KiB)
UBI: logical eraseblock size: 126976 bytes
UBI: smallest flash I/O unit: 2048
UBI: sub-page size: 512
UBI: VID header offset: 2048 (aligned 2048)
UBI: data offset: 4096
UBI: max. sequence number: 12
UBI: attached mtd7 to ubi0
UBI: MTD device name: "userfs"
UBI: MTD device size: 120 MiB
UBI: number of good PEBs: 963
UBI: number of bad PEBs: 4
UBI: number of corrupted PEBs: 0
UBI: max. allowed volumes: 128
UBI: wear-leveling threshold: 4096
UBI: number of internal volumes: 1
UBI: number of user volumes: 1
UBI: available PEBs: 0
UBI: total number of reserved PEBs: 963
UBI: number of PEBs reserved for bad PEB handling: 9
UBI: max/mean erase counter: 2/1
UBI: image sequence number: 1499480099
UBI: background thread "ubi_bgt0d" started, PID 1276
UBI device number 0, total 963 LEBs (122277888 bytes, 116.6 MiB), available 0 LEBs (0 bytes), LEB size 126976 bytes (124.0 KiB)
root@armv5te:~# ubiupdatevol /dev/ubi0_0 /home/duts/image-userfs.ubifs
root@armv5te:~# mount -t ubifs /dev/ubi0_0 /mnt/userfs
UBIFS: mounted UBI device 0, volume 0, name "userfs"
UBIFS: file system size: 119230464 bytes (116436 KiB, 113 MiB, 939 LEBs)
UBIFS: journal size: 9023488 bytes (8812 KiB, 8 MiB, 72 LEBs)
UBIFS: media format: w4/r0 (latest is w4/r0)
UBIFS: default compressor: lzo
UBIFS: reserved for root: 0 bytes (0 KiB)
root@armv5te:~# ls -l /mnt/userfs
total 8
-rw-rw-r-- 1 515 515 16 Nov 23 2011 README
-rw-rw-r-- 1 515 515 29 Nov 23 2011 date_of_creation
root@armv5te:~# hexdump -C /mnt/userfs/date_of_creation
00000000 4d 69 20 32 33 2e 20 4e 6f 76 20 30 39 3a 33 39 |Mi 23. Nov 09:39|
00000010 3a 34 31 20 43 45 54 20 32 30 31 31 0a          |:41 CET 2011.|
0000001d
root@armv5te:~#
```

9.1.5.3.4. Installing UBI images (if no UBI Volumes exist):

If the UBI device is not already formatted to contain suitable volumes, we have to generate UBI images. An UBI image may contain one or more UBI volumes, which in turn may be used to store an UBIFS file system. This is done using the "ubinize" tool, which unfortunately is a bit weird to use: an input configuration ini-file defines all the UBI volumes - their characteristics and the contents, but it does not define the characteristics of the flash flash device - these have to be specified as command-line options.

Also, we should keep in mind that we should no longer use "flash_eraseall" to erase the [MTD](#) device, as this has no knowledge about the UBI erase counters. Instead, we should use "ubiformat" to erase the flash and install a new UBI image.

Unmount and detach previously used UBI devices:

```
root@armv5te:~# umount /mnt/userfs
UBIFS: un-mount UBI device 0, volume 0
root@armv5te:~# ubidetach /dev/ubi_ctrl -m 7
UBI: mtd7 is detached from ubi0
root@armv5te:~#
```

Create UBI ini-file and create UBI images:

Note: for the volume sizes we use the byte count previously determined by running "ubininfo -a" above.

```
[hs@pollux]$ cat ubi-userfs.cfg
[ubifs]
mode=ubi
image=image-userfs.ubifs
vol_id=0
vol_size=120246272
vol_type=dynamic
vol_name=userfs
[hs@pollux]$ ubinize --min-io-size=2048 --peb-size=128KiB -s 2048 -o image-userfs.ubi ubi-userfs.cfg
[hs@pollux]$ cp /tmp/duts/image-userfs.ubi /opt/eldk-5.0/armv5te/rootfs/home/duts/
[hs@pollux]$
```

Note that for the NAND device we must pass the "-s 2048" option to ubinize; if we don't, an attempt to attach the created UBI device will result in error messages like these:

```
root@armv5te:~# ubiattach /dev/ubi_ctrl -m 7
UBI: attaching mtd7 to ubi0
UBI: physical eraseblock size: 131072 bytes (128 KiB)
UBI: logical eraseblock size: 129024 bytes
UBI: smallest flash I/O unit: 2048
UBI: sub-page size: 512
UBI: VID header offset: 512 (aligned 512)
UBI: data offset: 2048
UBI error: validate_ec_hdr: bad VID header offset 2048, expected 512
UBI error: validate_ec_hdr: bad EC header
UBI error: ubi_io_read_ec_hdr: validation failed for PEB 0
ubiattach: error!: cannot attach mtd7
          error 22 (Invalid argument)
root@armv5te:~#
```

For background information please see http://www.linux-mtd.infradead.org/faq/ubi.html#L_vid_offset_mismatch

Erase [MTD](#) partitions and install UBI images:

```
root@armv5te:/home/duts# ubiformat -q -s 2048 -f /home/duts/image-userfs.ubi /dev/mtd7
root@armv5te:/home/duts#
```

Verify that it worked:

```
root@armv5te:/home/duts# ubiattach /dev/ubi_ctrl -m 7 -O 2048
UBI: attaching mtd7 to ubi0
UBI: physical eraseblock size: 131072 bytes (128 KiB)
UBI: logical eraseblock size: 126976 bytes
UBI: smallest flash I/O unit: 2048
UBI: sub-page size: 512
UBI: VID header offset: 2048 (aligned 2048)
UBI: data offset: 4096
UBI: max. sequence number: 0
UBI: attached mtd7 to ubi0
UBI: MTD device name: "userfs"
UBI: MTD device size: 120 MiB
UBI: number of good PEBs: 963
UBI: number of bad PEBs: 4
UBI: number of corrupted PEBs: 0
UBI: max. allowed volumes: 128
UBI: wear-leveling threshold: 4096
UBI: number of internal volumes: 1
UBI: number of user volumes: 1
UBI: available PEBs: 3
UBI: total number of reserved PEBs: 960
UBI: number of PEBs reserved for bad PEB handling: 9
UBI: max/mean erase counter: 5/2
UBI: image sequence number: 2135855804
UBI: background thread "ubi_bgt0d" started, PID 1299
UBI device number 0, total 963 LEBs (122277888 bytes, 116.6 MiB), available 3 LEBs (380928 bytes, 372.0 KiB), LEB size 126976 bytes (124.0 KiB)
root@armv5te:/home/duts# mount -t ubifs /dev/ubi0_0 /mnt/userfs
UBIFS: mounted UBI device 0, volume 0, name "userfs"
UBIFS: file system size: 118849536 bytes (116064 KiB, 113 MiB, 936 LEBs)
UBIFS: journal size: 9023488 bytes (8812 KiB, 8 MiB, 72 LEBs)
UBIFS: media format: w4/r0 (latest is w4/r0)
UBIFS: default compressor: lzo
UBIFS: reserved for root: 0 bytes (0 KiB)
root@armv5te:/home/duts# ls -al /mnt/userfs
total 12
drwxr-xr-x 2 root root 304 Nov 23 09:39 .
drwxr-xr-x 7 root root 4096 Nov 23 09:39 ..
-rw-rw-r-- 1 515 515 16 Nov 23 09:39 README
-rw-rw-r-- 1 515 515 29 Nov 23 09:39 date_of_creation
root@armv5te:/home/duts# hexdump -C /mnt/userfs/date_of_creation
00000000 4d 69 20 32 33 2e 20 4e 6f 76 20 30 39 3a 33 39 |Mi 23. Nov 09:39|
00000010 3a 34 31 20 43 45 54 20 32 30 31 31 0a          |:41 CET 2011.|
0000001d
root@armv5te:/home/duts#
```

9.2. The TMPFS Virtual Memory Filesystem

The `tmpfs` filesystem, formerly known as `shmfs`, is a filesystem keeping all files in virtual memory.

Everything in `tmpfs` is temporary in the sense that no files will be created on any device. If you unmount a `tmpfs` instance, everything stored therein is lost.

`tmpfs` puts everything into the kernel internal caches and grows and shrinks to accommodate the files it contains and is able to swap unneeded pages out to swap space. It has maximum size limits which can be adjusted on the fly via `'mount -o remount ...'`

If you compare it to `ramfs` (which was the template to create `tmpfs`) you gain swapping and limit checking. Another similar thing is the RAM disk (`/dev/ram*`), which simulates a fixed size hard disk in physical RAM, where you have to create an ordinary filesystem on top. Ramdisks cannot swap and you do not have the possibility to resize them.

9.2.1. Mount Parameters

`tmpfs` has a couple of mount options:

- `size`: The limit of allocated bytes for this `tmpfs` instance. The default is half of your physical RAM without swap. If you oversize your `tmpfs` instances the machine will deadlock since the OOM handler will not be able to free that memory.
- `nr_blocks`: The same as `size`, but in blocks of `PAGECACHE_SIZE`.
- `nr_inodes`: The maximum number of inodes for this instance. The default is half of the number of your physical RAM pages.

These parameters accept a suffix `k`, `m` or `g` for kilo, mega and giga and can be changed on `remount`.

To specify the initial root directory you can use the following mount options:

- mode: The permissions as an octal number
- uid: The user id
- gid: The group id

These options do not have any effect on remount. You can change these parameters with `chmod(1)`, `chown(1)` and `chgrp(1)` on a mounted filesystem.

So the following mount command will give you a `tmpfs` instance on `/mytmpfs` which can allocate 12MB of RAM/SWAP and it is only accessible by root.

```
mount -t tmpfs -o size=12M,mode=700 tmpfs /mytmpfs
```

9.2.2. Kernel Support for tmpfs

In order to use a `tmpfs` filesystem, the `CONFIG_TMPFS` option has to be enabled for your kernel configuration. It can be found in the `Filesystems` configuration group. You can simply check if a running kernel supports `tmpfs` by searching the contents of `/proc/filesystems`:

```
bash# grep tmpfs /proc/filesystems
nodev    tmpfs
bash#
```

9.2.3. Usage of tmpfs in Embedded Systems

In embedded systems `tmpfs` is very well suited to provide read and write space (e.g. `/tmp` and `/var`) for a read-only root file system such as [CramFs](#) described in section [9.1.4. Compressed ROM Filesystem](#). One way to achieve this is to use symbolic links. The following code could be part of the startup file `/etc/rc.sh` of the read-only ramdisk:

```
#!/bin/sh
...
# Won't work on read-only root: mkdir /tmpfs
mount -t tmpfs tmpfs /tmpfs
mkdir /tmpfs/tmp /tmpfs/var
# Won't work on read-only root: ln -sf /tmpfs/tmp /tmpfs/var /
...
```

The commented out sections will of course fail on a read-only root filesystem, so you have to create the `/tmpfs` mount-point and the symbolic links in your root filesystem beforehand in order to successfully use this setup.

9.3. Using MultiMediaCards in Linux"

The MultiMediaCard (MMC) is a flash memory card standard.

Booting Linux with a MMC card connected to the `enbw_cmc` you should find in the bootlog something like that:

```
mmc0: new high speed SD card at address 0002
mmcblk0: mmc0:0002 00000 1.90 GiB
mmcblk0: p1
```

If the mmc card is detected, you should see at least the following device files:

```
/dev/mmcblk0
```

If there are partitions on it, you see the following device files:

```
/dev/mmcblk0pX
```

X=[1..n] with n=number of partitions

mount the partition:

```
root@armv5te:~# cd /tmp/duts
root@armv5te:/var/volatile/tmp/duts# ls -al mmc
total 0
drwxr-xr-x 2 root root 40 Nov 23 09:40 .
drwxr-xr-x 3 root root 60 Nov 23 09:40 ..
root@armv5te:/var/volatile/tmp/duts# mount
rootfs on / type rootfs (rw)
192.168.1.1:/opt/eldk-5.0/armv5te/rootfs/ on / type nfs (rw,relatime,vers=2,rsize=4096,wsiz=4096,namlen=255,hard,nolock,proto=udp,timeo=11,retrans=3,sec=s
proc on /proc type proc (rw,relatime)
tmpfs on /mnt/.psplash type tmpfs (rw,relatime,size=40k)
sysfs on /sys type sysfs (rw,relatime)
none on /dev type tmpfs (rw,relatime,mode=755)
devpts on /dev/pts type devpts (rw,relatime,gid=5,mode=620)
tmpfs on /var/volatile type tmpfs (rw,relatime)
tmpfs on /media/ram type tmpfs (rw,relatime)
/dev/sdal on /media/sdal type vfat (rw,sync,relatime,fmask=0022,dmask=0022,codepage=cp437,iocharset=iso8859-1,shortname=mixed,errors=remount-ro)
/dev/ubi0_0 on /mnt/userfs type ubifs (rw,relatime)
root@armv5te:/var/volatile/tmp/duts# df
Filesystem            1k-blocks      Used Available Use% Mounted on
192.168.1.1:/opt/eldk-5.0/armv5te/rootfs/
330279532 115243860 198258456 37% /
tmpfs                  40          0          40 0% /mnt/.psplash
none                  30268       100       30168 0% /dev
tmpfs                  30268       44       30224 0% /var/volatile
tmpfs                  30268        0       30268 0% /media/ram
/dev/sdal              974956     774772     200184 79% /media/sdal
/dev/ubi0_0            106656      24      106632 0% /mnt/userfs
root@armv5te:/var/volatile/tmp/duts# mount -t vfat /dev/mmcblk0p1 /tmp/duts/mmc
root@armv5te:/var/volatile/tmp/duts# mount
rootfs on / type rootfs (rw)
192.168.1.1:/opt/eldk-5.0/armv5te/rootfs/ on / type nfs (rw,relatime,vers=2,rsize=4096,wsiz=4096,namlen=255,hard,nolock,proto=udp,timeo=11,retrans=3,sec=s
proc on /proc type proc (rw,relatime)
tmpfs on /mnt/.psplash type tmpfs (rw,relatime,size=40k)
sysfs on /sys type sysfs (rw,relatime)
none on /dev type tmpfs (rw,relatime,mode=755)
devpts on /dev/pts type devpts (rw,relatime,gid=5,mode=620)
tmpfs on /var/volatile type tmpfs (rw,relatime)
tmpfs on /media/ram type tmpfs (rw,relatime)
/dev/sdal on /media/sdal type vfat (rw,sync,relatime,fmask=0022,dmask=0022,codepage=cp437,iocharset=iso8859-1,shortname=mixed,errors=remount-ro)
/dev/ubi0_0 on /mnt/userfs type ubifs (rw,relatime)
/dev/mmcblk0p1 on /var/volatile/tmp/duts/mmc type vfat (rw,relatime,fmask=0022,dmask=0022,codepage=cp437,iocharset=iso8859-1,shortname=mixed,errors=remount
root@armv5te:/var/volatile/tmp/duts# df
Filesystem            1k-blocks      Used Available Use% Mounted on
192.168.1.1:/opt/eldk-5.0/armv5te/rootfs/
330279532 115243860 198258456 37% /
tmpfs                  40          0          40 0% /mnt/.psplash
none                  30268       100       30168 0% /dev
tmpfs                  30268       44       30224 0% /var/volatile
tmpfs                  30268        0       30268 0% /media/ram
/dev/sdal              974956     774772     200184 79% /media/sdal
```

```

/dev/ubi0_0          106656         24    106632    0% /mnt/userfs
/dev/mmcblk0p1       134404        8706    125698    6% /var/volatile/tmp/duts/mmc
root@armv5te:/var/volatile/tmp/duts# ls -al mmc
total 2522
drwxr-xr-x 3 root root    512 Jan  1  1970 .
drwxr-xr-x 3 root root    60 Nov 23  09:40 ..
drwxr-xr-x 4 root root    512 Jan 21  2011 boot
-rwxr-xr-x 1 root root  17036 Mar 24  2011 mlo
-rwxr-xr-x 1 root root  202736 Mar 24  2011 u-boot.bin
-rwxr-xr-x 1 root root  2360344 Mar 24  2011 uImage
root@armv5te:/var/volatile/tmp/duts#

```

write some files to the MMC:

```

root@armv5te:/var/volatile/tmp/duts# date > /tmp/duts/mmc/date_of_creation
root@armv5te:/var/volatile/tmp/duts# dd if=/dev/urandom of=/tmp/duts/random.hex
bs=1024 count=1
1+0 records in
1+0 records out
1024 bytes (1.0 kB) copied, 0.00206346 s, 496 kB/s
root@armv5te:/var/volatile/tmp/duts# cp random.hex mmc/
root@armv5te:/var/volatile/tmp/duts# cmp random.hex mmc/random.hex
root@armv5te:/var/volatile/tmp/duts# ls -al mmc/
total 2523
drwxr-xr-x 3 root root    512 Nov 23  09:41 .
drwxr-xr-x 3 root root    80 Nov 23  09:41 ..
drwxr-xr-x 4 root root    512 Jan 21  2011 boot
-rwxr-xr-x 1 root root    29 Nov 23  09:40 date_of_creation
-rwxr-xr-x 1 root root  17036 Mar 24  2011 mlo
-rwxr-xr-x 1 root root   1024 Nov 23  09:41 random.hex
-rwxr-xr-x 1 root root  202736 Mar 24  2011 u-boot.bin
-rwxr-xr-x 1 root root  2360344 Mar 24  2011 uImage
root@armv5te:/var/volatile/tmp/duts#

```

unmount the partition with:

```

root@armv5te:/var/volatile/tmp/duts# umount /tmp/duts/mmc
root@armv5te:/var/volatile/tmp/duts# mount
rootfs on / type rootfs (rw)
192.168.1.1:/opt/eldk-5.0/armv5te/rootfs/ on / type nfs (rw,relatime,vers=2,rsize=4096,wsiz=4096,namlen=255,hard,nolock,proto=udp,timeo=11,retrans=3,sec=s
proc on /proc type proc (rw,relatime)
tmpfs on /mnt/.psplash type tmpfs (rw,relatime,size=40k)
sysfs on /sys type sysfs (rw,relatime)
none on /dev type tmpfs (rw,relatime,mode=755)
devpts on /dev/pts type devpts (rw,relatime,gid=5,mode=620)
tmpfs on /var/volatile type tmpfs (rw,relatime)
tmpfs on /media/ram type tmpfs (rw,relatime)
/dev/sd1 on /media/sd1 type vfat (rw,sync,relatime,fmask=0022,dmask=0022,codepage=cp437,iocharset=iso8859-1,shortname=mixed,errors=remount-ro)
/dev/ubi0_0 on /mnt/userfs type ubifs (rw,relatime)
root@armv5te:/var/volatile/tmp/duts# df
Filesystem            1K-blocks      Used Available Use% Mounted on
192.168.1.1:/opt/eldk-5.0/armv5te/rootfs/
330279532 115243860 198258456   37% /
tmpfs                  40          0         40    0% /mnt/.psplash
none                 30268       100     30168    0% /dev
tmpfs                 30268       48     30220    0% /var/volatile
tmpfs                 30268        0     30268    0% /media/ram
/dev/sd1              974956     774772    200184   79% /media/sd1
/dev/ubi0_0           106656        24     106632    0% /mnt/userfs
root@armv5te:/var/volatile/tmp/duts#

```

9.4. Splash Screen Support in Linux

To complement the [U-Boot Splash Screen](#) feature the new configuration option "CONFIG_FB_PRE_INIT_FB" was added to the Linux kernel. This allows the Linux kernel to skip certain parts of the framebuffer initialization and to reuse the framebuffer contents that was set up by the U-Boot firmware. This allows to have an image displayed nearly immediately after power-on, so the delay needed to boot the Linux kernel is masked to the user.

The current implementation has some limitations:

- We did not succeed in reusing the previously allocated framebuffer contents directly. Instead, Linux will allocate a new framebuffer, copy the contents, and then switch the display. This adds a minimal delay to the boot time, but is otherwise invisible to the user.
- Linux manages its own colormap, and we considered it too much effort to keep the same settings as used by U-Boot. Instead we use the "trick" that U-Boot will fill the color map table backwards (top down). This works pretty well for images which use no more than 200...225 colors. If the images uses more colors, a bad color mapping may result.
 - ☐ We strongly recommend to convert all images that will be loaded as Linux splash screens to use no more than 225 colors. The "ppmquant" tool can be used for this purpose (see [Bitmap Support in U-Boot](#) for details).
- Usually there will be a Linux device driver that is used to adjust the brightness and contrast of the display. When this driver starts, a visible change of brightness will happen if the default settings as used by U-Boot differ.
 - ☐ We recommend to store settings of brightness and contrast in U-Boot environment variables that can be shared between U-Boot and Linux. This way it is possible (assuming adequate driver support) to adjust the display settings correctly already in U-Boot and thus to avoid any flicker of the display when Linux takes over control.

9.5. Root File System: Design and Building

It is not an easy task to design the root file system for an embedded system. There are three major problems to be solved:

1. what to put in it
2. which file system type to use
3. where to store and how to boot it

For now we will assume that the contents of the root file system is already known; for example, it is given to us as a directory tree or a tarball which contains all the required files.

We will also assume that our system is a typical resource-limited embedded system so we will especially look for solutions where the root file system can be stored on on-board flash memory or other flash memory based devices like CompactFlash or SD cards, MMC or USB memory sticks.

A widespread approach to build a root file system is to use some Linux distribution (like the [ELDK](#)) and to remove things not needed. This approach may be pretty common, but it is almost always terribly wrong. You also don't build a family home by taking a skyscraper and removing parts. Like a house, a root file system should be built bottom up, starting from scratch and adding things you know you need. Never add anything where you don't exactly know what it's needed for.


But our focus here is on the second item: the options we have for choosing a file system type and the consequences this has.

In all cases we will base our experiments on the same content of the root filesystem; we use the images of the [SELF](#) (Simple Embedded Linux Framework) that come with the [ELDK](#). In a first step we will transform the [SELF](#) images into a tarball to meet the requirements mentioned above:

In a [ELDK](#) installation, the [SELF](#) images can be found in the `/opt/eldk/<architecture>/images/` directory. There is already a compressed ramdisk image in this directory, which we will use (`ramdisk_image.gz`):

1. Uncompress ramdisk image:

```
bash$ gzip -d -c -v /opt/eldk/ppc_8xx/images/ramdisk_image.gz >/tmp/ramdisk_image
/opt/eldk/ppc_8xx/images/ramdisk_image.gz: 61.4%
```

 Note: The following steps require root permissions!

2. Mount ramdisk image:

```
bash# mount -o loop /tmp/ramdisk_image /mnt/tmp
```

3. Create tarball; to avoid the need for `root` permissions in the following steps we don't include the device files in our tarball:

```
bash# cd /mnt/tmp
bash# tar -zc --exclude='dev/*' -f /tmp/rootfs.tar.gz *
```

4. Instead, we create a separate tarball which contains only the device entries so we can use them when necessary (with `cramfs`):

```
bash# tar -zcf /tmp/devices.tar.gz dev/
bash# cd /tmp
```

5. Unmount ramdisk image:

```
bash# umount /mnt/tmp
```

We will use the `/tmp/rootfs.tar.gz` tarball as master file in all following experiments.

9.5.1. Root File System on a Ramdisk

Ram disks are used very often to hold the root file system of embedded systems. They have several advantages:

- well-known
- well-supported by the Linux kernel
- simple to build
- simple to use - you can even combine the ramdisk with the Linux kernel into a single image file
- RAM based, thus pretty fast
- writable file system
- original state of file system after each reboot = easy recovery from accidental or malicious data corruption etc.

On the other hand, there are several disadvantages, too:

- big memory footprint: you always have to load the complete filesystem into RAM, even if only small parts of are actually used
- slow boot time: you have to load (and uncompress) the whole image before the first application process can start
- only the whole image can be replaced (not individual files)
- additional storage needed for writable persistent data

Actually there are only very few situations where a ramdisk image is the optimal solution. But because they are so easy to build and use we will discuss them here anyway.

In almost all cases you will use an `ext2` file system in your ramdisk image. The following steps are needed to create it:

1. Create a directory tree with the content of the target root filesystem. We do this by unpacking our master tarball:

```
$ mkdir rootfs
$ cd rootfs
$ tar xzf /tmp/rootfs.tar.gz
```

2. We use the `genext2fs` tool to create the ramdisk image as this allows to use a simple text file to describe which devices shall be created in the generated file system image. That means that no `root` permissions are required at all. We use the following device table `rootfs_devices.tab`:

| #<name> | <type> | <mode> | <uid> | <gid> | <major> | <minor> | <start> | <inc> | <count> |
|---------------|--------|--------|-------|-------|---------|---------|---------|-------|---------|
| /dev | d | 755 | 0 | 0 | - | - | - | - | - |
| /dev/console | c | 640 | 0 | 0 | 5 | 1 | - | - | - |
| /dev/fb0 | c | 640 | 0 | 0 | 29 | 0 | - | - | - |
| /dev/full | c | 640 | 0 | 0 | 1 | 7 | - | - | - |
| /dev/hda | b | 640 | 0 | 0 | 3 | 0 | - | - | - |
| /dev/hda | b | 640 | 0 | 0 | 3 | 1 | 1 | 1 | 16 |
| /dev/kmem | c | 640 | 0 | 0 | 1 | 2 | - | - | - |
| /dev/mem | c | 640 | 0 | 0 | 1 | 1 | - | - | - |
| /dev/mtd | c | 640 | 0 | 0 | 90 | 0 | 0 | 2 | 16 |
| /dev/mtdblock | b | 640 | 0 | 0 | 31 | 0 | 0 | 1 | 16 |
| /dev/mtdr | c | 640 | 0 | 0 | 90 | 1 | 0 | 2 | 16 |
| /dev/nftla | b | 640 | 0 | 0 | 93 | 0 | - | - | - |
| /dev/nftla | b | 640 | 0 | 0 | 93 | 1 | 1 | 1 | 8 |
| /dev/nftlb | b | 640 | 0 | 0 | 93 | 16 | - | - | - |
| /dev/nftlb | b | 640 | 0 | 0 | 93 | 17 | 1 | 1 | 8 |
| /dev/null | c | 640 | 0 | 0 | 1 | 3 | - | - | - |
| /dev/ptyp | c | 640 | 0 | 0 | 2 | 0 | 0 | 1 | 10 |
| /dev/ptypa | c | 640 | 0 | 0 | 2 | 10 | - | - | - |
| /dev/ptypb | c | 640 | 0 | 0 | 2 | 11 | - | - | - |
| /dev/ptypc | c | 640 | 0 | 0 | 2 | 12 | - | - | - |
| /dev/ptypd | c | 640 | 0 | 0 | 2 | 13 | - | - | - |
| /dev/ptype | c | 640 | 0 | 0 | 2 | 14 | - | - | - |
| /dev/typf | c | 640 | 0 | 0 | 2 | 15 | - | - | - |
| /dev/ram | b | 640 | 0 | 0 | 1 | 0 | 0 | 1 | 2 |
| /dev/ram | b | 640 | 0 | 0 | 1 | 1 | - | - | - |
| /dev/rtc | c | 640 | 0 | 0 | 10 | 135 | - | - | - |
| /dev/tty | c | 640 | 0 | 0 | 4 | 0 | 0 | 1 | 4 |
| /dev/tty | c | 640 | 0 | 0 | 5 | 0 | - | - | - |
| /dev/ttyS | c | 640 | 0 | 0 | 4 | 64 | 0 | 1 | 8 |
| /dev/tty | c | 640 | 0 | 0 | 3 | 0 | 0 | 1 | 10 |
| /dev/ttya | c | 640 | 0 | 0 | 3 | 10 | - | - | - |
| /dev/ttyb | c | 640 | 0 | 0 | 3 | 11 | - | - | - |
| /dev/ttyc | c | 640 | 0 | 0 | 3 | 12 | - | - | - |
| /dev/ttyd | c | 640 | 0 | 0 | 3 | 13 | - | - | - |
| /dev/ttye | c | 640 | 0 | 0 | 3 | 14 | - | - | - |
| /dev/ttyf | c | 640 | 0 | 0 | 3 | 15 | - | - | - |
| /dev/zero | c | 640 | 0 | 0 | 1 | 5 | - | - | - |

A description of the format of this table is part of the manual page for the `genext2fs` tool, `genext2fs(8)`.

3. We can now create an `ext2` file system image using the `genext2fs` tool:

```
$ ROOTFS_DIR=rootfs           # directory with root file system content
$ ROOTFS_SIZE=3700            # size of file system image
$ ROOTFS_FREE=100             # free space wanted
$ ROOTFS_INODES=380           # number of inodes
$ ROOTFS_DEVICES=rootfs_devices.tab # device description file
```

```
$ ROOTFS_IMAGE=ramdisk.img          # generated file system image

$ genext2fs -U \
  -d ${ROOTFS_DIR} \
  -D ${ROOTFS_DEVICES} \
  -b ${ROOTFS_SIZE} \
  -r ${ROOTFS_FREE} \
  -i ${ROOTFS_INODES} \
  ${ROOTFS_IMAGE}
```

4. Compress the file system image:

```
$ gzip -v9 ramdisk.img
rootfs.img:      55.6% -- replaced with ramdisk.img.gz
```

5. Create an U-Boot image file from it:

```
$ mkimage -T ramdisk -C gzip -n 'Test Ramdisk Image' \
> -d ramdisk.img.gz uRamdisk
Image Name:      Test Ramdisk Image
Created:         Sun Jun 12 16:58:06 2005
Image Type:      PowerPC Linux RAMDisk Image (gzip compressed)
Data Size:       1618547 Bytes = 1580.61 kB = 1.54 MB
Load Address:    0x00000000
Entry Point:     0x00000000
```

We now have a root file system image `uRamdisk` that can be used with U-Boot.

9.5.2. Root File System on a JFFS2 File System

JFFS2 (Journalling Flash File System version 2) was specifically designed for use on flash memory devices in embedded systems. It is a log-structured file system which means that it is robust against loss of power, crashes or other unordered shutdowns of the system ("robust" means that data that is just being written when the system goes down may be lost, but the file system itself does not get corrupted and the system can be rebooted without need for any kind of file system check).

Some of the advantages of using JFFS2 as root file system in embedded systems are:

- file system uses compression, thus making efficient use of flash memory
- log-structured file system, thus robust against unordered shutdown
- flash sector wear-leveling
- writable flash file system

Disadvantages are:

- long mount times (especially older versions)
- slow when reading: files to be read get uncompressed on the fly which eats [CPU](#) cycles and takes time
- slow when writing: files to be written get compressed, which eats [CPU](#) cycles and takes time, but it may even take much longer until data gets actually stored in flash if the file system becomes full and blocks must be erased first or - even worse - if garbage collection becomes necessary
- The garbage collector thread may run at any time, consuming [CPU](#) cycles and blocking accesses to the file system.

Despite the aforementioned disadvantages, systems using a JFFS2 based root file system are easy to build, make efficient use of the available resources and can run pretty reliably.

To create a JFFS2 based root file system please proceed as follows:

1. Create a directory tree with the content of the target root filesystem. We do this by unpacking our master tarball:

```
$ mkdir rootfs
$ cd rootfs
$ tar xzf /tmp/rootfs.tar.gz
```

2. We can now create a JFFS2 file system image using the `mkfs.jffs2` tool:

```
$ ROOTFS_DIR=rootfs          # directory with root file system content
$ ROOTFS_EBSIZE=0x20000      # erase block size of flash memory
$ ROOTFS_ENDIAN=b           # target system is big endian
$ ROOTFS_DEVICES=rootfs_devices.tab # device description file
$ ROOTFS_IMAGE=jffs2.img     # generated file system image

$ mkfs.jffs2 -U \
  -d ${ROOTFS_DIR} \
  -D ${ROOTFS_DEVICES} \
  -${ROOTFS_ENDIAN} \
  -e ${ROOTFS_EBSIZE} \
  -o ${ROOTFS_IMAGE}
mkfs.jffs2: skipping device_table entry '/dev': no parent directory!
```

☐ **Note:** When you intend to write the JFFS2 file system image to a NAND flash device, you should also add the `"-n"` (or `"--no-cleanmarkers"`) option, as cleanmarkers are not needed then.

When booting the Linux kernel prints the following messages showing the default partition map which is used for the flash memory on the TQM8xxL boards:

```
TQM flash bank 0: Using static image partition definition
Creating 7 MTD partitions on "TQM8xxL0":
0x00000000-0x00040000 : "u-boot"
0x00040000-0x00100000 : "kernel"
0x00100000-0x00200000 : "user"
0x00200000-0x00400000 : "initrd"
0x00400000-0x00600000 : "cramfs"
0x00600000-0x00800000 : "jffs"
0x00400000-0x00800000 : "big_fs"
```

We use U-Boot to load and store the JFFS2 image into the last partition and set up the Linux boot arguments to use this as root device:

1. Erase flash:

```
=> era 40400000 407FFFFF
..... done
Erased 35 sectors
```

2. Download JFFS2 image:

```
=> tftp 100000 /tftpboot/TQM860L/jffs2.img
Using FEC ETHERNET device
TFTP from server 192.168.3.1; our IP address is 192.168.3.80
Filename '/tftpboot/TQM860L/jffs2.img'.
Load address: 0x100000
Loading: #####
```

```
#####
#####
#####
#####
#####
#####
#####
```

```
done
Bytes transferred = 2033888 (1f08e0 hex)
```

3. Copy image to flash:

```
=> cp.b 100000 40400000 ${filesize}
Copy to Flash... done
```

4. set up boot arguments to use flash partition 6 as root device:

```
=> setenv mtd_args setenv bootargs root=/dev/mtdblock6 rw rootfstype=jffs2
=> printenv addip
addip=setenv bootargs ${bootargs} ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}:${netdev}:off panic=1
=> setenv flash_mtd 'run mtd_args addip;bootm ${kernel_addr}'
=> run flash_mtd
Using FEC ETHERNET device
TFTP from server 192.168.3.1: our IP address is 192.168.3.80
Filename '/tftpboot/TQM860L/uImage'.
Load address: 0x200000
Loading: #####
#####
#####
done
Bytes transferred = 719233 (af981 hex)
## Booting image at 40040000 ...
Image Name: Linux-2.4.25
Created: 2005-06-12 16:32:24 UTC
Image Type: PowerPC Linux Kernel Image (gzip compressed)
Data Size: 782219 Bytes = 763.9 kB
Load Address: 00000000
Entry Point: 00000000
Verifying Checksum ... OK
Uncompressing Kernel Image ... OK
Linux version 2.4.25 (wd@xpert) (gcc version 3.3.3 (DENX ELDK 3.1.1 3.3.3-9)) #1 Sun Jun 12 18:32:18 MEST 2005
On node 0 totalpages: 4096
zone(0): 4096 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/mtdblock6 rw rootfstype=jffs2 ip=192.168.3.80:192.168.3.1::255.255.255.0:tqm8601:eth1:off panic=1
Decrementer Frequency = 187500000/60
Calibrating delay loop... 49.86 BogoMIPS
...
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
VFS: Mounted root (jffs2 filesystem).
Freeing unused kernel memory: 56k init
```

```
BusyBox v0.60.5 (2005.03.07-06:54+0000) Built-in shell (msh)
Enter 'help' for a list of built-in commands.
```

```
# ### Application running ...
# mount
rootfs on / type rootfs (rw)
/dev/mtdblock6 on / type jffs2 (rw)
/proc on /proc type proc (rw)
# df /
Filesystem          1k-blocks      Used Available Use% Mounted on
rootfs                4096          2372        1724    58% /
```

9.5.3. Root File System on a cramfs File System

cramfs is a compressed, read-only file system.

Advantages are:

- file system uses compression, thus making efficient use of flash memory
- Allows for quick boot times as only used files get loaded and uncompressed

Disadvantages are:

- only the whole image can be replaced (not individual files)
- additional storage needed for writable persistent data
- *mkcramfs* tool does not support device table, so we need root permissions to create the required device files

To create a *cramfs* based root file system please proceed as follows:

1. Create a directory tree with the content of the target root filesystem. We do this by unpacking our master tarball:

```
$ mkdir rootfs
$ cd rootfs
$ tar -zxf /tmp/rootfs.tar.gz
```

2. Create the required device files. We do this here by unpacking a special tarball which holds only the device file entries. ☐ Note: this requires root permissions!

```
# cd rootfs
# tar -zxf /tmp/devices.tar.gz
```

3. Many tools require some storage place in a filesystem, so we must provide at least one (small) writable filesystem. For all data which may be lost when the system goes down, a *"tmpfs"* filesystem is the optimal choice. To create such a writable *tmpfs* filesystem we add the following lines to the */etc/rc.sh* script:

```
# mount TMPFS because root-fs is readonly
/bin/mount -t tmpfs -o size=2M tmpfs /tmpfs
```

Some tools require write permissions on some device nodes (for example, to change ownership and permissions), or dynamically (re-) create such files (for example, */dev/log* which is usually a Unix Domain socket). The files are placed in a writable filesystem; in the root filesystem symbolic links are used to point to their new locations:

```
dev/ptyp0    → /tmpfs/dev/ptyp0      dev/ttyp0    → /tmpfs/dev/ttyp0
dev/ptyp1    → /tmpfs/dev/ptyp1      dev/ttyp1    → /tmpfs/dev/ttyp1
dev/ptyp2    → /tmpfs/dev/ptyp2      dev/ttyp2    → /tmpfs/dev/ttyp2
dev/ptyp3    → /tmpfs/dev/ptyp3      dev/ttyp3    → /tmpfs/dev/ttyp3
dev/ptyp4    → /tmpfs/dev/ptyp4      dev/ttyp4    → /tmpfs/dev/ttyp4
```

```

dev/ptyp5      → /tmpfs/dev/ptyp5      dev/ttyp5      → /tmpfs/dev/ttyp5
dev/ptyp6      → /tmpfs/dev/ptyp6      dev/ttyp6      → /tmpfs/dev/ttyp6
dev/ptyp7      → /tmpfs/dev/ptyp7      dev/ttyp7      → /tmpfs/dev/ttyp7
dev/ptyp8      → /tmpfs/dev/ptyp8      dev/ttyp8      → /tmpfs/dev/ttyp8
dev/ptyp9      → /tmpfs/dev/ptyp9      dev/ttyp9      → /tmpfs/dev/ttyp9
dev/ptypa      → /tmpfs/dev/ptypa      dev/ttypa      → /tmpfs/dev/ttypa
dev/ptypb      → /tmpfs/dev/ptypb      dev/ttypb      → /tmpfs/dev/ttypb
dev/ptypc      → /tmpfs/dev/ptypc      dev/ttypc      → /tmpfs/dev/ttypc
dev/ptypd      → /tmpfs/dev/ptypd      dev/ttypd      → /tmpfs/dev/ttypd
dev/ptype      → /tmpfs/dev/ptype      dev/ttype      → /tmpfs/dev/ttype
dev/ptypf      → /tmpfs/dev/ptypf      dev/ttypf      → /tmpfs/dev/ttypf
tmp            → /tmpfs/tmp            var            → /tmpfs/var
dev/log        → /var/log/log

```

In case you use dhclient also:

```
etc/dhclient.conf → /tmpfs/var/lib/dhclient.conf  etc/resolv.conf → /tmpfs/var/lib/resolv.conf
```

To place the corresponding directories and device files in the `tmpfs` file system, the following code is added to the `/etc/rc.sh` script:

```

mkdir -p /tmpfs/tmp /tmpfs/dev \
        /tmpfs/var/lib/dhclient /tmpfs/var/lock /tmpfs/var/run

while read name minor
do
    mknod /tmpfs/dev/ptyp$name c 2 $minor
    mknod /tmpfs/dev/ttyp$name c 3 $minor
done << __EOD__
0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
a 10
b 11
c 12
d 13
e 14
f 15
__EOD__
chmod 0666 /tmpfs/dev/*

```

4. We can now create a `cramfs` file system image using the `mkcramfs` tool:

```

$ ROOTFS_DIR=rootfs          # directory with root file system content
$ ROOTFS_ENDIAN="-r"         # target system has reversed (big) endianness
$ ROOTFS_IMAGE=cramfs.img    # generated file system image

PATH=/opt/eldk/usr/bin:$PATH
mkcramfs ${ROOTFS_ENDIAN} ${DEVICES} ${ROOTFS_DIR} ${ROOTFS_IMAGE}
Swapping filesystem endian-ness
  bin
  dev
  etc
...
-48.78% (-86348 bytes)  in.ftpd
-46.02% (-16280 bytes)  in.telnetd
-45.31% (-74444 bytes)  xinetd
Everything: 1864 kilobytes
Super block: 76 bytes
CRC: c166be6d
warning: gids truncated to 8 bits. (This may be a security concern.)

```

5. We can use the same setup as before for the JFFS2 filesystem, just changing the bootargument to `"rootfstype=cramfs"`

9.5.4. Root File System on a Read-Only ext2 File System

When storing the root file system in on-board flash memory it seems only natural to look for special flash filesystems like JFFS2, or for other file system types that are designed for such environments like `cramfs`. It seems to be a bad idea to use a standard `ext2` file system because it contains neither any type of wear leveling which is needed for writable file systems in flash memory, nor is it robust against unordered shutdowns.

The situation changes if we use an `ext2` file system which we mount **read-only**. Such a configuration can be very useful in some situations.

Advantages:

- very fast
- low RAM memory footprint

Disadvantages:

- high flash memory footprint because no compression

To create an `ext2` image that can be used as a read-only root file system the following steps are necessary:

1. Create a directory tree with the content of the target root filesystem. We do this by unpacking our master tarball:

```

$ mkdir rootfs
$ cd rootfs
$ tar -xzf /tmp/rootfs.tar.gz

```

2. Like with the `cramfs` root file system, we use `"tmpfs"` for cases where a writable file system is needed and add the following lines to the `/etc/rc.sh` script:

```

# mount TMPFS because root-fs is readonly
/bin/mount -t tmpfs -o size=2M tmpfs /tmpfs

```

We also create the same symbolic links for device files that must be placed in a writable filesystem:

```

dev/ptyp0      → /tmpfs/dev/ptyp0      dev/ttyp0      → /tmpfs/dev/ttyp0
dev/ptyp1      → /tmpfs/dev/ptyp1      dev/ttyp1      → /tmpfs/dev/ttyp1
dev/ptyp2      → /tmpfs/dev/ptyp2      dev/ttyp2      → /tmpfs/dev/ttyp2

```



```
=> ide part

Partition Map for IDE device 0 -- Partition Type: DOS

Partition    Start Sector    Num Sectors    Type
    1              32         500704         6
=> ide write 100000 20 lce8

IDE write: device 0 block # 32, count 7400 ... 7400 blocks written: OK
```

Note that the "ide write" command takes parameters as hex numbers, and the write count is in terms of disk blocks of 512 bytes each. So we have to use 0x20 for the starts sector of the first partition, and 3788800 / 512 = 7400 = 0x1CE8 for the block count.

- We now prepare the Linux boot arguments to take this partition as read-only root device:

```
=> setenv cf_args setenv bootargs root=/dev/hdal ro
=> setenv flash_cf 'run cf_args addip;bootm ${kernel_addr}'
=> setenv bootcmd run flash_cf
```

- ...and boot the system:

```
...
Linux version 2.4.25 (wd@xpert) (gcc version 3.3.3 (DENX ELDK 3.1.1 3.3.3-9)) #1 Sun Jun 12 18:32:18 MEST 2005
On node 0 totalpages: 4096
zone(0): 4096 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/hdal ro ip=192.168.3.80:192.168.3.1::255.255.255.0:tqm8601:eth1:off panic=1
Decrementer Frequency = 187500000/60
Calibrating delay loop... 49.86 BogoMIPS
...
```

9.5.6. Root File System in a Read-Only File in a FAT File System

This is a more complicated example that shows that - depending on project requirements - many other alternatives for choosing a root file system for your embedded system exist.

The scenario is as follows: on your embedded device you use a cheap and popular storage medium like CompactFlash, MMC or SD cards or USB memory sticks to store both the Linux kernel and your root file system. You want to distribute software updates over the internet: your customers can download the file from your web site, or you sent the images by email. Your customers may use any flash card or memory stick they happen to find, so you have no information about brand or size of the storage device.

Unfortunately most of your customers use Windows systems. And they don't want to be bothered with long instructions how to create special partitions on the storage device or how to write binary images or things like that. A simple "copy file" operation is nearly exhausting their capabilities.

What to do? Well, if copying a file is all your customers can do we should not ask for more. Storage devices like CompactFlash cards etc. typically come with a single partition on it, which holds a FAT or VFAT file system. This cannot be used as a Linux root file system directly, so we have to use some trickery.

Here is one possible solution: Your software distribution consists of two files: The first file is the Linux kernel with a minimal ramdisk image attached (using the multi-file image format for U-Boot); U-Boot can load and boot such files from a FAT or VFAT file system. The second file is your root file system. For convenience and speed we use again an image of an `ext2` file system. When Linux boots, it will initially use the attached ramdisk as root file system. The programs in this ramdisk will mount the FAT or VFAT file system - read-only. Then we can use a loop device (see *losetup(8)*) to associate the root file system image with a block device which can be used as a mount point. And finally we use *pivot_root(8)* to change the root file system to our image on the CF card.

This sounds not so complicated, and actually it is quite simple once you understand what needs to be done. Here is a more detailed description:

- The root file system image is easy: as mentioned before, we will use an `ext2` file system image, and to avoid wearing the flash storage device we will use it in read-only mode - we did a read-only `ext2` root file system image before, and here we can just re-use the existing image file.
- The initial ramdisk image that performs the *pivot_root* step must be created from scratch, but we already know how to create ramdisk images, so we just have to figure out what to put in it.

The most important tool here is *nash*, a script interpreter that was specifically designed for such purposes (see *nash(8)*). We don't need any additional tools, and if we use static linking, then the *nash* binary plus a small script to control it is all we need for our initial ramdisk.

To be precise, we need a couple of (empty) directories (`bin`, `dev`, etc, `lib`, `loopfs`, `mnt`, `proc`, and `sysroot`), the `bin/nash` binary, the `linuxrc` script and a symbolic link `sbin` pointing to `bin`:

```
drwxr-xr-x  2 wd      users      4096 Apr 13 01:11 bin
-rwxr-xr-x  1 wd      users      469512 Apr 11 22:47 bin/nash
drwxr-xr-x  2 wd      users      4096 Apr 12 00:04 dev
drwxr-xr-x  2 wd      users      4096 Apr 12 00:04 etc
drwxr-xr-x  2 wd      users      4096 Apr 12 00:04 lib
-rwxr-xr-x  1 wd      users      511 Apr 13 01:28 linuxrc
drwxr-xr-x  2 wd      users      4096 Apr 12 00:04 loopfs
drwxr-xr-x  2 wd      users      4096 Apr 12 00:09 mnt
drwxr-xr-x  2 wd      users      4096 Apr 12 00:04 proc
lrwxrwxrwx  1 wd      users        3 Jun 12 18:54 sbin -> bin
drwxr-xr-x  2 wd      users      4096 Apr 12 00:04 sysroot
```

- We also need only a minimal device table for creating the initial ramdisk:

```
#<name>    <type> <mode> <uid> <gid> <major> <minor> <start> <inc> <count>
/dev       d 755 0    0    -    -    -    -    -
/dev/console c 640 0    0    5    1    -    -    -
/dev/hda   b 640 0    0    3    0    -    -    -
/dev/hda   b 640 0    0    3    1    1    1    8
/dev/loop  b 640 0    0    7    0    0    1    4
/dev/null  c 640 0    0    1    3    -    -    -
/dev/ram   b 640 0    0    1    0    0    1    2
/dev/ram   b 640 0    0    1    1    -    -    -
/dev/tty   c 640 0    0    4    0    0    1    4
/dev/tty   c 640 0    0    5    0    -    -    -
/dev/ttyS  c 640 0    0    4    64   0    1    4
/dev/zero  c 640 0    0    1    5    -    -    -
```

- To create the initial ramdisk we perform the usual steps:

```
$ INITRD_DIR=initrd
$ INITRD_SIZE=490
$ INITRD_FREE=0
$ INITRD_INODES=54
$ INITRD_DEVICES=initrd_devices.tab
$ INITRD_IMAGE=initrd.img

$ genext2fs -U \
  -d ${INITRD_DIR} \
  -D ${INITRD_DEVICES} \
  -b ${INITRD_SIZE} \
  -r ${INITRD_FREE} \
  -i ${INITRD_INODES} \
```

```

${INITRD_IMAGE}

$ gzip -v9 ${INITRD_IMAGE}

```

The result is a really small (233 kB) compressed ramdisk image.

- Assuming you already have your Linux kernel image, you can now use `mkimage` to build an U-Boot multi-file image that combines the Linux kernel and the initial ramdisk:

```

$ LINUX_KERNEL=linuxppc_2_4_devel/arch/ppc/boot/images/vmlinux.gz
$ mkimage -A ppc -O Linux -T multi -C gzip \
> -n 'Linux with Pivot Root Helper' \
> -d ${LINUX_KERNEL}:${INITRD_IMAGE}.gz linux.img
Image Name:      Linux with Pivot Root Helper
Created:         Mon Jun 13 01:48:11 2005
Image Type:      PowerPC Linux Multi-File Image (gzip compressed)
Data Size:       1020665 Bytes = 996.74 kB = 0.97 MB
Load Address:    0x00000000
Entry Point:     0x00000000
Contents:
  Image 0:       782219 Bytes = 763 kB = 0 MB
  Image 1:       238433 Bytes = 232 kB = 0 MB

```

The newly created file `linux.img` is the second image we have to copy to the CF card.

We are done.

But wait - one essential part was not mentioned yet: the `linuxrc` script in our initial ramdisk image which contains all the magic. This script is quite simple:

```

#!/bin/nash

echo Mounting /proc filesystem
mount -t proc /proc /proc

echo Creating block devices
mkdevices /dev

echo Creating root device
mkrootdev /dev/root
echo 0x0100 > /proc/sys/kernel/real-root-dev

echo Mounting flash card
mount -o noatime -t vfat /dev/hda1 /mnt


echo losetup for filesystem image
losetup /dev/loop0 /mnt/rootfs.img

echo Mounting root filesystem image
mount -o defaults --ro -t ext2 /dev/loop0 /sysroot

echo Running pivot_root
pivot_root /sysroot /sysroot/initrd
umount /initrd/proc

```

Let's go through it step by step:

- The first line says that it's a script file for the `/bin/nash` interpreter.
 Note: even if this file looks like a shell script it is NOT interpreted by a shell, but by the `nash` interpreter. For a complete list of available `nash` commands and their syntax please refer to the manual page, *nash(8)*.
- The first action is to mount the `/proc` pseudo file system which is needed to find out some required information.
- Then we create block device entries for all partitions listed in `/proc/partitions` (`mkdevices` command).
- In the next step a block device for our new root file system is created (`mkrootdev` command).
- Then we mount the CF card. We assume that there is only a single partition on it (`/dev/hda1`) which is of type `VFAT` (which also will work with `FAT` file systems). These assumptions work fine with basically all memory devices used under Windows.
- We further assume that the file name of the root file system image on the CF card is `"rootfs.img"` - this file now gets mounted using a loop device (`losetup` and `mount` commands).
- Our file system image, is now mounted on the `/sysroot` directory. In the last step we use `pivot_root` to make this the new root file system.
- As a final cleanup we unmount the `/proc` file system which is not needed any more.

There is one tiny flaw in this method: since we mount the CF card on a directory in the ramdisk to be able to access to root file system image. This means that we cannot unmount the CF card, which in turn prevents us from freeing the space for the initial ramdisk. The consequence is that you permanently lose approx. 450 kB of RAM for the ramdisk. [We could of course re-use this ramdisk space for temporary data, but such optimization is beyond the scope of this document.]

And how does this work on our target?

- First we copy the two images to the CF card; we do this on the target under Linux:

```

bash-2.05b# fdisk -l /dev/hda

Disk /dev/hda: 256 MB, 256376832 bytes
16 heads, 32 sectors/track, 978 cylinders
Units = cylinders of 512 * 512 = 262144 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/hda1 *          1           978       250352    6   FAT16
bash-2.05b# mkfs.vfat /dev/hda1
mkfs.vfat 2.8 (28 Feb 2001)
bash-2.05b# mount -t vfat /dev/hda1 /mnt
bash-2.05b# cp -v linux.img rootfs.img /mnt/
`linux.img' -> `/mnt/linux.img'
`rootfs.img' -> `/mnt/rootfs.img'
bash-2.05b# ls -l /mnt
total 4700
-rwxr--r--  1 root    root      1020729 Jun 14 05:36 linux.img
-rwxr--r--  1 root    root      3788800 Jun 14 05:36 rootfs.img
bash-2.05b# umount /mnt

```

- We now prepare U-Boot to load the `"uMulti"` file (combined Linux kernel and initial ramdisk) from the CF card and boot it:

```

=> setenv fat_args setenv bootargs rw
=> setenv fat_boot 'run fat_args addip;fatload ide 0:1 200000 linux.img;bootm'
=> setenv bootcmd run fat_boot

```

- And finally we try it out:

```

U-Boot 1.1.3 (Jun 13 2005 - 02:24:00)

CPU:   XPC86xxx2PnnD4 at 50 MHz: 4 kB I-Cache 4 kB D-Cache FEC present
Board: TQM860LDB0A3-T50.202
DRAM:  16 MB
FLASH:  8 MB
In:     serial

```

```

Out:    serial
Err:    serial
Net:    SCC ETHERNET, FEC ETHERNET [PRIME]
PCMCIA: 3.3V card found: Transcend    256M
        Fixed Disk Card
        IDE interface
        [silicon] [unique] [single] [sleep] [standby] [idle] [low power]

Bus 0: OK
  Device 0: Model: Transcend    256M Firm: 1.1 Ser#: SSSC256M04Z27A25906T
            Type: Removable Hard Disk
            Capacity: 244.5 MB = 0.2 GB (500736 x 512)

Type "run flash_nfs" to mount root filesystem over NFS

Hit any key to stop autoboot:  0
reading linux.img

1025657 bytes read
## Booting image at 00200000 ...
   Image Name:   Linux with Pivot Root Helper
   Created:      2005-06-13   0:32:41 UTC
   Image Type:   PowerPC Linux Multi-File Image (gzip compressed)
   Data Size:    1025593 Bytes = 1001.6 kB
   Load Address: 00000000
   Entry Point:  00000000
   Contents:
     Image 0:    787146 Bytes = 768.7 kB
     Image 1:    238433 Bytes = 232.8 kB
   Verifying Checksum ... OK
   Uncompressing Multi-File Image ... OK
   Loading Ramdisk to 00f3d000, end 00f77361 ... OK
Linux version 2.4.25 (wd@xpert) (gcc version 3.3.3 (DENX ELDK 3.1.1 3.3.3-9)) #1 Mon Jun 13 02:32:10 MEST 2005
On node 0 totalpages: 4096
zone(0): 4096 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: rw ip=192.168.3.80:192.168.3.1::255.255.255.0:tqm8601:eth1:off panic=1
Decrementer Frequency = 187500000/60
Calibrating delay loop... 49.86 BogoMIPS
...
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
RAMDISK: Compressed image found at block 0
Freeing initrd memory: 232k freed
VFS: Mounted root (ext2 filesystem).
Red Hat nash version 4.1.18 starting
Mounting /proc filesystem
Creating block devices
Creating root device
Mounting flash card
hda: hda1
hda: hda1
losetup for filesystem image
Mounting root filesystem image
Running pivot_root
Freeing unused kernel memory: 60k init

BusyBox v0.60.5 (2005.03.07-06:54+0000) Built-in shell (msh)
Enter 'help' for a list of built-in commands.

# ### Application running ...

```

9.6. Root File System Selection

Now we know several options for file systems we can use, and know how to create the corresponding images. But how can we decide which one to chose?

For practical purposes in embedded systems the following criteria are often essential:

- boot time (i. e. time needed from power on until application code is running)
- flash memory footprint
- RAM memory footprint
- effects on software updates

The following data was measured for the different configurations. All measurements were performed on the same TQM860L board (MPC860 [CPU](#) at 50 MHz, 16 MB RAM, 8 MB flash, 256 MB CompactFlash card):

| File System Type | Boot Time | Free Mem | Updates | while running |
|----------------------------------|---------------------------|--------------------------|-------------------------|-------------------------------|
| ramdisk | 16.3 sec | 6.58 MB | whole image | yes |
| JFFS2 | 21.4 sec | 10.3 MB | per file | only non-active files |
| cramfs | 10.8 sec | 10.3 MB | whole image | no |
| ext2 (ro) | 9.1 sec | 10.8 MB | whole image | no |
| ext2 on CF (ro) | 9.3 sec | 10.9 MB | whole image | no |
| File on FAT fs | 11.4 sec | 7.8 MB | whole image | yes |

As you can see, the ramdisk solution is the worst of all in terms of RAM memory footprint; also it takes a pretty long time to boot. However, it is one of the few solutions that allow an in-situ update while the system is running.

JFFS2 is easy to use as it's a writable file system but it takes a **long** time to boot.

A read-only ext2 file system shines when boot time and RAM memory footprint are important; you pay for this with an increased flash memory footprint.

External flash memory devices like CompactFlash cards or USB memory sticks can be cheap and efficient solutions especially when lots of data need to be stored or when easy update procedures are required. -

9.7. Overlay File Systems

Introduction

Overlay File Systems provide an interesting approach to several frequent problems in Embedded Systems. For example, **mini_fo** is a virtual kernel file system that can make read-only file systems writable. This is done by redirecting modifying operations to a writeable location called "storage directory", and leaving the original data in the "base directory" untouched. When reading, the file system merges the modified and original data so that only the newest versions will appear. This occurs transparently to the user, who can access the data like on any other read-write file system.

What it is good for?

In embedded systems the main use of **mini_fo** is to overlay the root file system. This means it is mounted on top of the regular root file system, thereby allowing applications or users to transparently make modifications to it but redirecting these to a different location.

Some examples of why this is usefull are explained in the following sections.

Making a *read-only* root filesystem *writable*

Root file systems stored in flash are often read only, such as [cramfs](#) or [read only ext2](#). While this offers major advantages in terms of speed and flash memory footprint, it nevertheless is often desirable to be able to modify the root file system, for example to

- apply (small) software updates without having to burn a whole new root file system image to flash
- make modifications during development when frequent changes to the root file system occur.

This can be achieved by mounting **mini_fo** on top of the root file system and using a (probably small) writeable partition as the storage file system. This could be either a JFFS2 flash file system, or during development even an external hard disk. This has the following advantages:

- read-only file systems (fast, small memory footprint) can be used like persistent writable file systems (in contrast to a ramdisk)
- slow flash journalling file systems with large flash memory footprint can be avoided.

Non persistant changes

Ramdisks are often used when the root file system needs to be modified non-persistantly. This works well, but downsides are the large RAM memory footprint and the time costly operation of copying the ramdisk into RAM during startup. These can be avoided by overlaying the root file system as in the previous example but with the difference that the [tmpfs file system](#) is used as storage. Thus *only* modified files are stored in RAM, and can even be swapped out if neccessary. This saves boot time and RAM!

Resetable changes

mini_fo can be easily used to implement a "reset to factory defaults" function by overlaying the default root file system. When configuration changes are made, these are automatically directed to the storage file system and take precedence over the original files. Now, to restore the system to factory defaults, all that needs to be done is delete the contents of the storage directory. This will remove all changes made to the root file system and return it to the original state.

☐ Note: Deleting the contents of the storage directory should only be done when the overlay file system is unmounted.

Examples

Generally, there are two different ways of overlaying the root file system, which both make sense in different scenarios.

Starting a single application in a chrooted overlayed environment

This is easy. Let's assume "/" is the read-only root file system and /dev/mtdblock5 contains a small JFFS2 flash partition that shall be used to store modifications made by application "/usr/bin/autoPilot":

```
# mount -t jffs2 /dev/mtdblock5 /tmp/sto
# insmod mini_fo.o
# mount -t mini_fo -o base=/,sto=/tmp/sto/ /mnt/mini_fo/
# cd /mnt/mini_fo/
# chroot . /usr/bin/autoPilot
```

The **mini_fo** file system is mounted with "/" as base directory, "/tmp/sto/" as storage directory to the mount point "/mnt/mini_fo". After that, `chroot(1)` is used to start the application with the new file system root "/mnt/mini_fo". All modifications made by the application will be stored to the JFFS2 file system in /tmp/sto.

Starting the whole system system in chrooted overlayed environment

This is more interesting, and a bit trickier, as mounting needs to be done during system startup *after* the root file system has been mounted, but *before* init is started. The best way to do this is to have a script that mounts the mini_fo file system on top of root and then starts init in the chrooted overlayed environment. For example assume the following script "overlay_init", stored in /sbin/:

```
#!/bin/bash
#
# mount mini_fo overlay file system and execute init
#

# make sure these exist in the read-only file system
STORAGE=/tmp/sto
MOUNT_POINT=/mnt/mini_fo/

# mount tmpfs as storage file system with a maximum size of 32MB
mount -t tmpfs -o rw,size=32M none $STORAGE

/sbin/modprobe mini_fo
mount -t mini_fo -o base=/,sto=$STORAGE / $MOUNT_POINT

exec /usr/sbin/chroot $MOUNT_POINT /sbin/init

echo "exec chroot failed, bad!"
exec /bin/sh

exit 1
```

Now its easy to choose between a **mini_fo** overlayed and the regular non overlayed system just by setting the "init" kernel parameter in the boot loader to "init=/sbin/overlay_init".

Tips

- `pivot_root(1)` can be used with chroot if there is need to access the original non overlayed root file system from the chrooted overlayed environment.

Performance overhead

The **mini_fo** file system is inserted as an additional layer between the VFS and the native file system, and thus creates some overhead that varies strongly depending of the operation performed.

1. modifying a regular file for the first time
This results in a copy of the original file beeing created in the storage directory, that is then modified. Overhead depends on the size of the modified file.
2. Reading from files, creating new files, modifying already modified files
These operations are passed directly through to the lower native layer, and only impose an overhead of 1-2%.

Further information

This section discusses how the **mini_fo** overlay file system can be used in embedded systems. More general information is available at the **mini_fo** project page:

http://www.denx.de/wiki/publish/DULG/DULG-enbw_cmc.html

16/02/2012

<http://www.denx.de/wiki/Know/MiniFOHome>.

9.8. The Persistent RAM File system (PRAMFS)

The `pramfs` file system supports persistent memory devices such as SRAM. Instead of having a block emulation layer over such a memory area and using a normal file system on top of that, `pramfs` seeks to induce minimal overhead in this situation. Most important in this respect is that the normal block layer caching of the Linux kernel is circumvented in `pramfs`.

9.8.1. Mount Parameters

The most important parameters for normal usage are

- `physaddr`: The physical address of the static memory.
- `init`: When given, it will initialize the file system to that size.

9.8.2. Example

We will show a sample usage of `pramfs` in this section using normal DRAM on a board with at least 256MB of memory. For `pramfs` we reserve the upper 32MB by appending `mem=224M` to the kernel command line.

First off we generate some testdata on a persistent file system (`/tmp`) to demonstrate that `pramfs` survives a reboot (of course with power always applied to keep the DRAM refreshed):

```
bash-3.00# dd if=/dev/urandom bs=1M count=8 of=/tmp/testdata
8+0 records in
8+0 records out
bash-3.00#
```

Next we mount the 32MB that we reserved and initialize it to be 32MB in size and copy the testfile. A final compare shows that the copy was indeed successful so we can reboot:

```
bash-3.00# mount -t pramfs -o physaddr=0xe000000,init=0x2000000 none /mnt
bash-3.00# cp /tmp/testdata /mnt
bash-3.00# cmp /tmp/testdata /mnt/testdata
bash-3.00# reboot
```

Having rebooted (using `mem=224M` on the kernel command line again of course) we mount the file system but this time without the `init` parameter because it is preinitialized. We then check the contents again:

```
bash-3.00# mount -t pramfs -o physaddr=0xe000000 none /mnt
bash-3.00# ls /mnt
testdata
bash-3.00# cmp /tmp/testdata /mnt/testdata
bash-3.00#
```

- [10. Debugging](#)
 - [10.1. Debugging of U-Boot](#)
 - [10.1.1. Debugging of U-Boot Before Relocation](#)
 - [10.1.2. Debugging of U-Boot After Relocation](#)
 - [10.2. Linux Kernel Debugging](#)
 - [10.2.1. Linux Kernel and Statically Linked Device Drivers](#)
 - [10.2.2. Dynamically Loaded Device Drivers \(Modules\)](#)
 - [10.2.3. GDB Macros to Simplify Module Loading](#)
 - [10.3. GDB Startup File and Utility Scripts](#)
 - [10.4. Tips and Tricks](#)
 - [10.5. Application Debugging](#)
 - [10.5.1. Local Debugging](#)
 - [10.5.2. Remote Debugging](#)
 - [10.6. Debugging with Graphical User Interfaces](#)

10. Debugging

The purpose of this document is not to provide an introduction into programming and debugging in general. We assume that you know how to use the GNU debugger [gdb](#) and probably it's graphical frontends like [ddd](#). We also assume that you have access to adequate tools for your work, i. e. a BDI2000 [BDM](#)/JTAG debugger. The following discussion assumes that the host name of your BDI2000 is `bdi`.

Please note that there are several limitations in earlier versions of GDB. The version of GDB as distributed with the [ELDK](#) contains several bug fixes and extensions. If you find that your GDB behaves differently, have a look at the GDB sources and patches that come with the [ELDK](#) source.

10.1. Debugging of U-Boot

When U-Boot starts it is running from ROM space. Running from flash would make it nearly impossible to read from flash while executing code from flash not to speak of updating the U-Boot image in flash itself. To be able to do just that, U-Boot relocates itself to RAM. We therefore have two phases with different program addresses. The following sections show how to debug U-Boot in both phases.

10.1.1. Debugging of U-Boot Before Relocation

Before relocation, the addresses in the ELF file can be used without any problems, so debugging U-Boot in this phase with the BDI2000 is quite easy:

```
bash[0]# ${CROSS_COMPILE}gdb u-boot
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i386-redhat-linux --target=ppc-linux"...

(gdb) target remote bdi:2001
Remote debugging using bdi:2001
0xffffffff in ?? ()
(gdb) b cpu_init_f
Breakpoint 1 at 0xffffd3310: file cpu_init.c, line 136.
(gdb) c
Continuing.

Breakpoint 1, cpu_init_f () at cpu_init.c:136
136      asm volatile(" bl      0f"          ::: "lr");
(gdb) s
137      asm volatile("0:      mflr    3"          ::: "r3");
(gdb)
138      asm volatile(" addi    4, 0, 14"          ::: "r4");
(gdb)
```

`cpu_init_f` is the first C function called from the code in `start.C`.

10.1.2. Debugging of U-Boot After Relocation

For debugging U-Boot after relocation we need to know the address to which U-Boot relocates itself to. When no exotic features like `PRAM` are used, this address usually is `<MAXMEM>` - `CONFIG_SYS_MONITOR_LEN`. In our example with 16MB RAM and `CONFIG_SYS_MONITOR_LEN` = 192KB this yields the address `0x1000000 - 0x30000 = 0xFD0000`.

In other cases, check the source code, and apply some common sense. For example, on Power Architecture® we use "r2" to hold a pointer to the "global data" structure ("`struct global_data`"); this structure contains a field

```
unsigned long    reloc_off;        /* Relocation Offset */
```

which is the offset between the image addresses in flash and in RAM. You can easily print this value in gdb like that:

```
(gdb) print/x ((gd_t *)$r2)->reloc_off
```

Then add this value to the value of `TEXT_BASE` as defined in your board's `config.mk` file, and you get the start address of the U-Boot image in RAM.

With this knowledge, we can instruct gdb to forget the old symbol table and reload the symbols with our calculated offset:

```
(gdb) symbol-file
Discard symbol table from `/home/dzu/denx/cvs-trees/u-boot/u-boot'? (y or n) y
No symbol file now.
(gdb) add-symbol-file u-boot 0xfd0000
add symbol table from file "u-boot" at
        .text_addr = 0xfd0000
(y or n) y
Reading symbols from u-boot...done.
(gdb) b board_init_r
Breakpoint 2 at 0xfd99ac: file board.c, line 533.
(gdb) c
Continuing.
```

```
Breakpoint 2, board_init_r (id=0xfbb1f0, dest_addr=16495088) at board.c:533
533      {
(gdb)
```

`board_init_r` is the first C routine running in the newly relocated C friendly RAM environment.

The simple example above relocates the symbols of only one section, `.text`. Other sections of the executable image (like `.data`, `.bss`, etc.) are not relocated and this prevents gdb from accessing static and global variables by name. See more sophisticated examples in section [10.3. GDB Startup File and Utility Scripts](#).

10.2. Linux Kernel Debugging

10.2.1. Linux Kernel and Statically Linked Device Drivers

10.2.2. Dynamically Loaded Device Drivers (Modules)

First start GDB in the root directory of your Linux kernel, using the `vmlinux` kernel image as file to debug:

```
bash$ cd <linux-root>
bash$ ${CROSS_COMPILE}gdb vmlinux
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i386-redhat-linux --target=ppc-linux".
(gdb)
```

Now attach to the target and start execution with the commands:

```
(gdb) target remote bdi:2001
Remote debugging using bdi:2001
0x00000100 in ?? ()
(gdb) c
Continuing.
```

Now the target should boot Linux as usual. Next you need to load your kernel module on the target:

```
bash# insmod -m ex_sw.o
Sections:
      Size      Address   Align
.text      00000060    cf030000  2**2
.rodata     000002f4    cf030060  2**2
.data       00000134    cf030354  2**2
.sdata      00000000    cf030488  2**0
.sbss       0000000c    cf030488  2**2
.kstrtab    00000085    cf030494  2**0
.bss        00000000    cf030519  2**0
.sbss       00000008    cf03051c  2**2
...
```

The option `-m` prints out the addresses of the various code and data segments (`.text`, `.data`, `.sdata`, `.bss`, `.sbss`) after relocation. GDB needs these addresses to know where all the symbols are located. We now interrupt GDB to load the symbol table of the module as follows:

```
(gdb) ^C
Program received signal SIGSTOP, Stopped (signal).
...
(gdb) add-symbol-file <path-to-module-dir>/ex_sw.o 0xcf030060\
-s .rodata 0xcf030354\
-s .data 0xcf030488\
-s .sdata 0xcf030488\
-s .bss 0xcf030519\
-s .sbss 0xcf03051c
add symbol table from file "<path-to-module-dir>/ex_sw.o" at
        .text_addr = 0xcf030060
        .rodata_addr = 0xcf030354
        .data_addr = 0xcf030488
        .sdata_addr = 0xcf030488
        .bss_addr = 0xcf030519
        .sbss_addr = 0xcf03051c
(y or n) y
Reading symbols from <path-to-module-dir>/ex_sw.o...done.
```

Now you can list the source code of the module, set break points or inspect variables as usual:

```
(gdb) l fun
61     static RT_TASK *thread;
62
63     static int cpu_used[NR_RT_CPUS];
64
65     static void fun(int t)
66     {
67         unsigned int loops = LOOPS;
68         while(loops--) {
69             cpu_used[hard_cpu_id()];
70             rt_leds_set_mask(1,t);
(gdb)
(gdb) b ex_sw.c:69
Breakpoint 1 at 0xcf03007c: file ex_sw.c, line 69.
(gdb) c
Continuing.
Breakpoint 1, fun (t=1) at ex_sw.c:69
69             cpu_used[hard_cpu_id()];
(gdb) p ntasks
$1 = 16
(gdb) p stack_size
$2 = 3000
```

The next section demonstrates a way to automate the symbol table loading procedure.

10.2.3. GDB Macros to Simplify Module Loading

The following GDB macros and scripts help you to load kernel modules into GDB in a half-automatic way. It assumes, that the module on the target has been installed with the command:

```
bash# insmod -m my_module.o > my_module.o.map
```

In your \$HOME directory you need the scripts *add-symbol-file.sh* and the GDB startup file *.gdbinit*, which are listed in [10.3. GDB Startup File and Utility Scripts](#) below.

Now you can include the symbol definition into GDB with:

```
bash$ ${CROSS_COMPILE}gdb vmlinux
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i386-redhat-linux --target=ppc-linux".
0x00000100 in ?? ()
c
Continuing.
^C
Program received signal SIGSTOP, Stopped (signal).
0xcf02a91c in ?? ()
(gdb) add-module rtai4/examples/sw/ex_sw.o
add symbol table from file "/HHL/8xx/target/home/wolf/rtai4/examples/sw/ex_sw.o" at
      .text_addr = 0xcf030060
      .rodata_addr = 0xcf030340
      .data_addr = 0xcf030464
      .sdata_addr = 0xcf030464
      .bss_addr = 0xcf0304f5
      .sbss_addr = 0xcf0304f8
(gdb) b ex_sw.c:69
Breakpoint 1 at 0xcf03007c: file ex_sw.c, line 69.
(gdb) c
Continuing.

Breakpoint 1, fun (t=0x1) at ex_sw.c:69
69             cpu_used[hard_cpu_id()];
(gdb) p/d loops
$2 = 999986939
(gdb) p t
$3 = 0x1
(gdb) d b
Delete all breakpoints? (y or n) y
(gdb) c
Continuing.
```

10.3. GDB Startup File and Utility Scripts

In addition to the *add-module* macro, the followin example GDB startup file contains a few other useful settings and macros, which you may want to adjust to your local environment:

```
set output-radix 16

target remote bdi:2001

define reset
    detach
    target remote bdi:2001
end

define add-module
    shell ~/add-symbol-file.sh $arg0
    source ~/add-symbol-file.gdb
end
document add-module
    Usage: add-module <module>

    Do add-symbol-file for module <module> automatically.
    Note: A map file with the extension ".map" must have
    been created with "insmod -m <module> > <module>.map"
    in advance.
end
```

The following shell script *~/add-symbol-file.sh* is used to run the GDB *add-symbol-file* command automatically:

```
#!/bin/sh
#
# Constructs the GDB "add-symbol-file" command string
# from the map file of the specified kernel module.

add_sect() {
    ADDR=`awk '/^$1' / {print $3}' $MAPFILE`
    if [ "$ADDR" != "" ]; then
```



```

        echo "-s $1 0x`awk '/^'$1' / {print $3}' $MAPFILE`"
    fi
}

[ $# == 1 ] && [ -r "$1" ] || { echo "Usage: $0 <module>" >&2 ; exit 1 ; }

MAPFILE=$1.map

ARGS="0x`awk '/^.text / {print $3}' $MAPFILE`
`add_section .rodata`
`add_section .data`
`add_section .sdata`
`add_section .bss`
`add_section .sbss`
"

echo "add-symbol-file $1 $ARGS" > ~/add-symbol-file.gdb

```

10.4. Tips and Tricks

- To prevent GDB from jumping around in the code when trying to single step, i. e. when it seems as if the code is not executing line by line, you can recompile your code with the following additional compiler options:

```
-fno-schedule-insns -fno-schedule-insns2
```

- On some systems (like the MPC8xx or MPC8260) you can only define one hardware breakpoint. Therefore you must delete an existing breakpoint before you can define a new one:

```

(gdb) d b
Delete all breakpoints? (y or n) y
(gdb) b ex_preempt.c:63
Breakpoint 2 at 0xcf030080: file ex_preempt.c, line 63.

```

10.5. Application Debugging

10.5.1. Local Debugging

In case there is a native GDB available for your target you can use it for application debugging as usual:

```

bash$ gcc -Wall -g -o hello hello.c
bash$ gdb hello
...
(gdb) l
1      #include <stdio.h>
2
3      int main(int argc, char* argv[])
4      {
5          printf ("Hello world\n");
6          return 0;
7      }
(gdb) break 5
Breakpoint 1 at 0x8048466: file hello.c, line 5.
(gdb) run
Starting program: /opt/eldk/ppc_8xx/tmp/hello

Breakpoint 1, main (argc=0x1, argv=0xbffff9f4) at hello.c:5
5      printf ("Hello world\n");
(gdb) c
Continuing.
Hello world

Program exited normally.

```

10.5.2. Remote Debugging

`gdbserver` allows you to connect your program with a remote GDB using the "target remote" command. On the target machine, you need to have a copy of the program you want to debug. `gdbserver` does not need your program's symbol table, so you can strip the program if necessary to save space. `GDB` on the host system does all the symbol handling. Here is an example:

```

bash$ ${CROSS_COMPILE}gcc -Wall -g -o hello hello.c
bash$ cp -p hello <directory-shared-with-target>/hello-stripped
bash$ ${CROSS_COMPILE}strip <directory-shared-with-target>/hello-stripped

```

To use the server, you must tell it how to communicate with `GDB`, the name of your program, and the arguments for your program. To start a debugging session via network type on the target:

```

bash$ cd <directory-shared-with-host>
bash$ gdbserver 192.168.1.1:12345 hello-stripped
Process hello-stripped created; pid = 353

```

And then on the host:

```

bash$ ${CROSS_COMPILE}gdb hello
...
(gdb) set solib-absolute-prefix /opt/eldk/${CROSS_COMPILE}
(gdb) dir /opt/eldk/${CROSS_COMPILE}
Source directories searched:
/opt/eldk/${CROSS_COMPILE}:${cdir}:${cwd}
(gdb) target remote 192.168.1.99:12345
Remote debugging using 192.168.1.99:12345
0x30012748 in ?? ()
...
(gdb) l
1      #include <stdio.h>
2
3      int main(int argc, char* argv[])
4      {
5          printf ("Hello world\n");
6          return 0;
7      }
(gdb) break 5
Breakpoint 1 at 0x10000498: file hello.c, line 5.
(gdb) continue
Continuing.

Breakpoint 1, main (argc=1, argv=0x7ffffbe4) at hello.c:5
5      printf ("Hello world\n");
(gdb) p argc
$1 = 1
(gdb) continue
Continuing.

```

Program exited normally.

 If the target program you want to debug is linked against shared libraries, you *must* tell GDB where the proper target libraries are located. This is done using the `set solib-absolute-prefix` GDB command. If this command is omitted, then, apparently, GDB loads the host versions of the libraries and gets crazy because of that.

10.6. Debugging with Graphical User Interfaces

It is convenient to use `DDD`, a Graphical User Interface to GDB, for debugging as it allows to define and execute frequently used commands via buttons. You can start `DDD` with the command:

```
bash$ ddd --debugger ${CROSS_COMPILE}gdb &
```

If `DDD` is not already installed on your Linux system, have a look at your distribution media.

11. Simple Embedded Linux Framework

12. Books, Mailing Lists, Links, etc.

This section provides references on where to find more information

Contents:

- [12. Books, Mailing Lists, Links, etc.](#)
 - [12.1. Application Notes](#)
 - [12.2. Further Reading](#)
 - [12.2.1. License Issues](#)
 - [12.2.2. Linux kernel](#)
 - [12.2.3. General Linux / Unix programming](#)
 - [12.2.4. Network Programming](#)
 - [12.2.5. C++ programming](#)
 - [12.2.6. Java programming](#)
 - [12.2.7. Power Architecture® Programming](#)
 - [12.2.8. Embedded Topics](#)
 - [12.3. Mailing Lists](#)
 - [12.4. Links](#)
 - [12.5. Tools](#)

12.1. Application Notes

A collection of [Application Notes](#) relevant for embedded computing can be found on the DENX web server.

12.2. Further Reading

12.2.1. License Issues

Articles

- <http://www.gnu.org/licenses/gpl-2.0.html>: GNU General Public License, version 2
- <http://www.gnu.org/copyleft/gpl.html>: GNU General Public License
- <http://www.softwarefreedom.org/resources/2008/compliance-guide.html>: A Practical Guide to [GPL](#) Compliance
- <http://article.gmane.org/gmane.comp.video.dri.devel/52751> **Alan Cox** about combining [GPL](#) device drivers with closed source user space libraries

12.2.2. Linux kernel

Books

- **Karim Yaghmour, Jon Masters, Gilad Ben-Yosef, Philippe Gerum**: "Building Embedded Linux Systems 2nd edition", Paperback: 462 pages, O'Reilly & Associates; (August 2008); ISBN 10: 0-596-52968-6; ISBN 13: 9780596529680 ISBN 059600222X - IMHO **the** best book about Embedded Linux so far. An absolute must have.
- **Greg Kroah-Hartman**: "Linux Kernel in a Nutshell", 198 pages, O'Reilly ("In Nutshell" series), (December 2006), ISBN 10: 0-596-10079-5; ISBN 13: 9780596100797
 - Tarball of PDF files (3 MB): http://www.kernel.org/pub/linux/kernel/people/gregkh/lkn/lkn_pdf.tar.bz2
 - Tarball of DocBook files (1 MB): http://www.kernel.org/pub/linux/kernel/people/gregkh/lkn/lkn_xml.tar.bz2
- **Craig Hollabaugh**: "Embedded Linux: Hardware, Software, and Interfacing", Paperback: 432 pages; Addison Wesley Professional; (March 7, 2002); ISBN 0672322269
- **Christopher Hallinan**: "Embedded Linux Primer: A Practical Real-World Approach", 576 pages, Prentice Hall, September 2006, ISBN-10: 0-13-167984-8; ISBN-13: 978-0-13-167984-9
- **Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman**: "[Linux Device Drivers](#)", 3rd Edition; Paperback: 636 pages; O'Reilly & Associates; 3rd edition (February 2005); ISBN: 0-596-00590-31 - **The** reference book for writing Linux device drivers. An absolute must have. => [Read online](#)
- **Jürgen Quade, Eva-Katharina Kunst**: "[Linux-Treiber entwickeln](#)"; Broschur: 436 pages; dpunkt.verlag, Juni 2004; ISBN 3898642380
 - focused on kernel 2.6, unfortunately German only
 - => [Read online](#)
- **Sreekrishnan Venkateswaran**: "Essential Linux Device Drivers", 744 pages, Prentice Hall, March 2008, ISBN-10: 0-13-239655-6; ISBN-13: 978-0-13-239655-4
 - => [Read online](#)

Articles

- [The Linux Kernel](#) - describing most aspects of the Linux Kernel. Probably, the first reference for beginners. Lots of illustrations explaining data structures use and relationships. In short: a must have.
- [Linux Kernel Module Programming Guide](#) - Very nice 92 pages [GPL](#) book on the topic of modules programming. Lots of examples.
- **LWN: Porting device drivers to the 2.6 kernel** - Series of articles (37) in Linux Weekly News: <http://lwn.net/Articles/driver-porting/>
- MIPS Linux Porting Guide: <http://linux.junsun.net/porting-howto/porting-howto.html>
- Andries Brouwers remarks to the linux kernel: <http://www.win.tue.nl/~aeb/linux/lk/lk.html>

12.2.3. General Linux / Unix programming

Books

http://www.denx.de/wiki/publish/DULG/DULG-enbw_cmc.html

- **W. Richard Stevens:** "Advanced Programming in the UNIX Environment", Addison Wesley, ISBN 0-201-56317-7
- **Eric S. Raymond:** "The Art of Unix Programming", Addison Wesley, ISBN 0131429019 => [Read online](#). If you don't want to read the whole book then at least look at the [Basics of the Unix philosophy](#) condensing lots of experience into a few rules. This is essential reading.
- **David R. Butenhof:** "Programming with POSIX Threads", Addison Wesley, ISBN 0-201-63392-2.
- **Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell:** "Pthreads Programming", O'Reilly & Associates
- **"Git Community Book"**
See <http://book.git-scm.com/> or download the [PDF version](#).

Articles

- The GNU C Library: http://www.linuxselfhelp.com/gnu/glibc/html_chapter/libc_toc.html
- General Linux Programming: <http://www.linuxselfhelp.com/cats/programming.html>
- Multi-Threaded Programming With POSIX Threads:
<http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html>
- **Brad Hards:** [The Linux USB Input Subsystem, Part I](#)
<http://www.linuxjournal.com/article/6396>
- **Brad Hards:** [Using the Input Subsystem, Part II](#)
<http://www.linuxjournal.com/article/6429>
- **Ulrich Drepper:** Position Independent Binaries: ["Text Relocations"](#)
<http://people.redhat.com/drepper/textrelocs.html>
- **Ulrich Drepper:** ["How to Write Shared Libraries"](#)
<http://people.redhat.com/drepper/dsohowto.pdf>
- **Ulrich Drepper:** ["What Every Programmer Should Know About Memory"](#)
<http://people.redhat.com/drepper/cpumemory.pdf>
- **David Goldberg:** ["What Every Computer Scientist Should Know About Floating-Point Arithmetic"](#)
http://www.physics.ohio-state.edu/~dws/grouplinks/floating_point_math.pdf
- More Ulrich Drepper stuff: <http://people.redhat.com/drepper/>
- How to [optimize DSOs](#) by identifying unused non-exported functions and data.
<http://blog.flameeyes.eu/articles/2008/01/17/today-how-to-identify-unused-exported-functions-and-variables>
- A quite complete history of the UNIX family can be found here: <http://www.levenez.com/unix/>
- Unix Manual, first edition, 3 November 1971: <http://cm.bell-labs.com/cm/cs/who/dmr/1stEdman.html>
- **John Graham-Cumming:** Debugging Makefiles
<http://newsletter.embedded.com/cgi-bin4/DM/y/e4Kd0G1ErD0FrY0E3FS0EZ>
- Binutils / ld documentation: [Linker Scripts](#)
- git ready - learn one git command at a time: <http://gitready.com/>

Standards:

- POSIX.1-2008, IEEE Std 1003.1™-2008, The Open Group Technical Standard Base Specifications, Issue 7: <http://pubs.opengroup.org/onlinepubs/9699919799/mindex.html>
- Linux Standard Base: <http://refspecs.freestandards.org/lsb.shtml>
- [Single UNIX Specification, Version 3](#) (needs registration even for online viewing)
- [Single UNIX Specification, Version 2](#)
- PCI Bus Bindings - Standard for Boot Firmware:
http://playground.sun.com/1275/bindings/pci/pci2_1.pdf
- International standardization working group for the programming language C: <http://www.open-std.org/jtc1/sc22/WG14/>

12.2.4. Network Programming

Books

- **W. Richard Stevens:** "TCP/IP Illustrated, Volume 1 - The Protocols", Addison Wesley, ISBN 0-201-63346-9
- **Gary R. Wright, W. Richard Stevens:** "TCP/IP Illustrated, Volume 2 - The Implementation", Addison Wesley, ISBN 0-201-63354-X
- **W. Richard Stevens:** "TCP/IP Illustrated, Volume 3 - TCP for Transactions", Addison Wesley, ISBN 0-201-63495-3
- **W. Richard Stevens:** "UNIX Network Programming, Volume 1 - Networking APIs: Sockets and XTI", 2nd ed., Prentice Hall, ISBN-0-13-490012-X
- **W. Richard Stevens:** "UNIX Network Programming, Volume 2 - Interprocess Communication", 2nd ed., Prentice Hall, ISBN-0-13-081081-9

Articles

- Linux Networking topics (like [NAPI](#), [GSO](#), [VLAN](#), [IPsec](#) etc.):
http://linux-net.osdl.org/index.php/Main_Page

12.2.5. C++ programming

Books

- **Scott Meyers:** "Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)", Addison-Wesley, May 20, 2005, ISBN: 0321334876

12.2.6. Java programming

Books

- **Joshua Bloch:** "Effective Java -- Programming Language Guide", 2001, Addison Wesley, ISBN 0-201-31005-8, 250 pages

12.2.7. Power Architecture® Programming

Books

- Programming Environments Manual for 32-Bit Implementations of the [PowerPC](#) architecture: <http://www.freescale.com/files/product/doc/MPCFPE32B.pdf>
- IBM PDF file (600+ page book) on PowerPC assembly language: <http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2>
- Power.org™ Standard for Embedded Power Architecture™ Platform Requirements (ePAPR): http://www.power.org/resources/downloads/Power_ePAPR_APPROVED_v1.0.pdf

Articles

- Introduction to Assembly on the PowerPC: <http://www-106.ibm.com/developerworks/library/l-ppc/?t=gr.lnxw09=PowPC>
- IBM PDF compiler writers guide on PPC asm tuning etc.: <http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2>
- A developer's guide to the POWER architecture: <http://www-128.ibm.com/developerworks/linux/library/l-powarch/index.html>
- PowerPC [EABI](#) Calling Sequence: <http://sourceware.redhat.com/pub/binutils/ppc-docs/ppc-eabi-calling-sequence>
- PowerPC Embedded Application Binary Interface (32-Bit Implementation): <http://sourceware.redhat.com/pub/binutils/ppc-docs/ppc-eabi-1995-01.pdf>
- Developing PowerPC Embedded Application Binary Interface ([EABI](#)) Compliant Programs <http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569970071B0D6>
- System V Application Binary Interface - PowerPC Processor Supplement: http://refspecs.freestandards.org/elf/elfspec_ppc.pdf
- Device Tree Wiki: http://devicetree.org/Main_Page
- Device Tree Usage: http://devicetree.org/Device_Tree_Usage

- Linux for PowerPC Embedded Systems HOWTO (very old): <http://penguinppc.org/embeddedd/howto/PowerPC-Embeddedd-HOWTO.html>
- Linux for PowerPC Embedded Systems HOWTO (old): <http://www.denx.de/twiki/bin/view/PPCEmbedded>
- Understanding MPC5200 Bestcomm Firmware: [Posting](#) on linuxppc-embedded@ozlabs.org mailing list (see also the mailing list [archive entry](#)), source code [disasm.c](#) for a disassembler, and ["SmartDMA Hand-Assembly Guides"](#) document.

12.2.8. Embedded Topics

Articles

- Things you always wanted to know about NAND flash but never dared to ask: [Micron Application Note](#)
- The ultimate goal of [Embedded C++](#) is to provide embedded systems programmers with a subset of C++ that is easy for the average C programmer to understand and use.
- Our contribution to the Darwin year 2009: Hardware designs that will **not** replicate: [Topic in DENX Wiki](#)

12.3. Mailing Lists

These are some mailing lists of interest. If you are new to mailing lists then please take the time to read at least [RFC 1855](#).

- [linux-arm-kernel](#) - Communications among developers and users of Linux on arm boards
- [linuxppc-embedded](#) - Communications among developers and users of Linux on embedded [Power Architecture](#)® boards
☐ This mailing list has been merged into the `linuxppc-dev` mailing list below and thus **does not exist anymore**.
- [linuxppc-dev](#) - Communications among active developers of Linux on 32 bit [Power Architecture](#)® platforms. Not intended for user support.
- [u-boot](#) - Support for "U-Boot" Universal Bootloader
- [ELDK](#) - Support for DENX Embedded Linux Development Kit

12.4. Links

Linux Kernel Resources:

- The Linux Documentation Project : <http://www.tldp.org/>
- Generic ("official") Linux Kernel sources:
git: <http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=tree>
FTP: <ftp://ftp.kernel.org/pub/linux/kernel/v2.6/>
- Full git history of Linux: <http://thread.gmane.org/gmane.linux.kernel/690811>
- Generic kernel sources for [Power Architecture](#)™ systems: <http://penguinppc.org/dev/kernel.shtml>
- DENX kernel sources: <http://git.denx.de/?p=linux-denix.git;a=summary>
- Cross-Referencing the Linx Kernel: <http://lxr.linux.no/source/?a=ppc>
- Starting point for Linux based asm (mostly x86): <http://linuxassembly.org/>

Realtime, Xenomai, [RTAI](#):

- Xenomai Home Page: <http://www.xenomai.org/>
- [Hackbench](#), a commonly used system stress tool
- [Calibrator](#), determines cache
- [RTAI](#) Home Page: <http://www.rtai.org/>
- DENX [RTAI](#) Patches: <ftp://ftp.denx.de/pub/RTAI/> sizes at runtime, rendering it another useful system stress tool

U-Boot:

- U-Boot Project Page: <http://www.denx.de/wiki/U-Boot/WebHome>
☐ Note that the old [SourceForge](#) page is not maintained anymore.
- DENX U-Boot and Linux Guide: <http://www.denx.de/twiki/bin/view/DULG>

Cross Development Tools:

- DENX Embedded Linux Development Kit: <http://www.denx.de/twiki/bin/view/DULG/ELDK>

Miscellaneous or unsorted material:

- BDI2000 [List of supported Flash Memories](#): This document not only lists the currently supported flash chips, but also the required settings in the BDI config file.
- BDI2000 configuration files: <ftp://78.31.64.234/bdigdb/config/>

12.5. Tools

- <http://lxr.linux.no/source/> - Cross-Referencing the Linux Kernel - using a versatile hypertext cross-referencing tool for the Linux Kernel source tree (the Linux Cross-Reference project)
- <ftp://ftp.denx.de/pub/tools/backtrace> - Decode Stack Backtrace - Perl script to decode the Stack Backtrace printed by the Linux Kernel when it panics
- ftp://ftp.denx.de/pub/tools/clone_tree - "Clone" a Source Tree - Perl script to create a working copy of a source tree (for example the Linux Kernel) which contains mainly symbolic links (and automatically omits "unwanted" files like [CVS](#) repository data, etc.)
- [13. Appendix](#)
 - [13.1. Flat Device Tree](#)
 - [13.2. BDI2000 Configuration file](#)

13. Appendix

13.1. Flat Device Tree

```
[hs@pollux]$ /*
 * Device Tree for the EnBW CMC platform
 *
 * Copyright 2011 DENX Software Engineering GmbH
 * Heiko Schocher <hs@denx.de>
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the
 * Free Software Foundation; either version 2 of the License, or (at your
 * option) any later version.
 */
/dts-v1/;
/include/ "skeleton.dtsi"
```

```

/ {
    model = "EnBW CMC";
    compatible = "enbw,cmc";

    aliases {
        ethernet0 = &eth0;
    };

    soc@1c00000 {
        compatible = "arm,davinci";
        #address-cells = <1>;
        #size-cells = <1>;
        ranges = <0x0 0x01c00000 0x400000>;

        eth0: emac@1e22000 {
            compatible = "davinci,emac";
            local-mac-address = [ 00 00 00 00 00 00 ];
        };
        wdt@1c21000 {
            compatible = "davinci,wdt";
            reg = <0x00021000 0x1000>;
            period = <100>;
        };
        bootcount@1c23000 {
            compatible = "uboot,bootcount";
            reg = <0x23060 0x20>;
        };
    };

    onchipram@8000000 {
        compatible = "davinci,onchipram";
        #address-cells = <1>;
        #size-cells = <1>;
        ranges = <0x0 0x80000000 0x20000>;
    };

    emif@60000000 {
        compatible = "davinci,emifa";
        #address-cells = <2>;
        #size-cells = <1>;
        reg = <0x68000000 0x80000>;
        ranges = <2 0 0x60000000 0x02000000
                  3 0 0x62000000 0x02000000
                  4 0 0x64000000 0x02000000
                  5 0 0x66000000 0x02000000>;

        flash@2,0 {
            compatible = "cfi-flash";
            reg = <2 0x0 0x400000>;
            #address-cells = <1>;
            #size-cells = <1>;
            bank-width = <2>;
            device-width = <2>;
        };
    };
};

```

13.2. BDI2000 Configuration file

```

[hs@pollux]$ ; bdiGDB configuration for BMK / EnBW "CMC" board
; -----
;
; Using a faster JTAG clock than 1 MHz is only possible if the
; ARM core clock has been speed-up after a reset. Either by letting
; U-boot run for some time or by configuring the clock system via
; the appropriate init list entries.
;
; Info about the JTAG clock frequency:
; -----
; BDI2000:
;   0=Adaptive,
;   1=16MHz, 2=8MHz, 3=4MHz,
;   4= 1MHz, 5=500kHz, 6=200kHz, 7=100kHz, 8=50kHz,
;   9=20kHz, 10=10kHz, 11=5kHz, 12=2kHz, 13=1kHz
; BDI3000:
;   0=Adaptive,
;   1=32MHz, 2=16MHz, 3=11MHz, 4=8MHz, 5=5MHz, 6=4MHz,
;   7=1MHz, 8=500kHz, 9=200kHz, 10=100kHz, 11=50kHz,
;   12=20kHz, 13=10kHz, 14=5kHz, 15=2kHz, 16=1kHz,
;
;
[INIT]
;
; CS2CFG slowest timing 16 Bit
;WM32 0x68000010 0x3fffffff

; Pinmux setup for EMIFA
;WM32 0x01c14134 0x11111111
;WM32 0x01c14138 0x11111111
;WM32 0x01c1413c 0x11111111
;WM32 0x01c14140 0x11111111

[TARGET]
CPUTYPE      ARM926E
;CLOCK        7                ;BDI2000: JTAG clock : 100kHz
;CLOCK        4 7              ;BDI2000: JTAG clock : start with 100 kHz then use 1 MHz
CLOCK        4 6              ;BDI2000: JTAG clock : start with 100 kHz then use ? MHz
;CLOCK        0 7              ;BDI2000: JTAG clock : start with 100 kHz then use adaptive
TRST         PUSH_PULL        ;TRST driver type (OPENDRAIN | PUSH_PULL)
RESET        HARD 1000         ;NONE | HARD <n> (ms), asserted via SCANINIT
ENDIAN        LITTLE          ;memory model (LITTLE | BIG)
;STARTUP      STOP 50          ;let U-boot setup the system
STARTUP       RESET           ;No code is executed after reset.
WAKEUP        500
;VECTOR       CATCH 0x0f       ;catch P_Abort, SWI, Undefined and Reset
VECTOR        CATCH 0x02       ;catch P_Abort, SWI, Undefined and Reset
;
SCANPRED      1 6              ;count for ICEPick TAP
SCANSUCC      0 0              ;no device after ARM926e
;
;
; Configure ICEPick module to make ARM926 TAP visible
SCANINIT      r1:t1:w100000:t0:r0:w1000: ;Toggle reset
SCANINIT      w1000:ch10:w1000:          ;clock TCK with TMS high and wait
SCANINIT      i6=07:d8=89:i6=02:          ;connect and select router
SCANINIT      d32=81000080:                ;set IP control
SCANINIT      d32=a2002008:                ;configure TAP2

```

```

SCANINIT d32=a2002108: ;enable TAP2
SCANINIT c110:i10=ffff ;clock 10 times in RTI, scan bypass
;
;

[HOST]
IP 192.168.1.1
FILE cmc/u-boot.bin
FORMAT BIN 0x00000000
LOAD MANUAL ;load code MANUAL or AUTO after reset
PROMPT CMC>

[FLASH]
WORKSPACE 0x80000000 ;workspace in internal RAM (!!! boot mode UART !!! )
;CHIPTYPE I28BX16 ; Numonyx M28W320CT90N6 - 4 MB
CHIPTYPE AM29BX16
;CHIPSIZE 0x00400000 ; Size is 4MB
CHIPSIZE 0x00200000 ; Size is 2MB
BUSWIDTH 16
FILE cmc/u-boot.bin
FORMAT BIN
ERASE 0x60000000
ERASE 0x60004000
ERASE 0x60006000
ERASE 0x60008000
ERASE 0x60010000
ERASE 0x60020000
ERASE 0x60030000
ERASE 0x60040000
ERASE 0x60050000
ERASE 0x60060000
ERASE 0x60070000

[REGS]
FILE BDI2000/reg926e.def

```

- [14. FAQ - Frequently Asked Questions](#)
 - [14.1. ELDK](#)
 - [14.1.1. ELDK Installation under FreeBSD](#)
 - [14.1.2. ELDK Installation Hangs](#)
 - [14.1.3. gvfs: Permission Denied](#)
 - [14.1.4. Installation on Local Harddisk](#)
 - [14.1.5. System Include Files Missing](#)
 - [14.1.6. patch: command not found](#)
 - [14.1.7. ELDK Include Files Missing](#)
 - [14.1.8. Using the ELDK on a 64 bit platform](#)
 - [14.1.9. How can I check if Floating Point support is working?](#)
 - [14.1.10. ELDK 2.x Installation Aborts](#)
 - [14.1.11. Enable SSH Access](#)
 - [14.2. U-Boot](#)
 - [14.2.1. Can U-Boot be configured such that it can be started in RAM?](#)
 - [14.2.2. Relocation cannot be done when using -mrelocatable](#)
 - [14.2.3. Source object has EABI version 4, but target has EABI version 0](#)
 - [14.2.4. U-Boot crashes after relocation to RAM](#)
 - [14.2.5. Warning - bad CRC, using default environment](#)
 - [14.2.6. Net: No ethernet found](#)
 - [14.2.7. Wrong debug symbols after relocation](#)
 - [14.2.8. Decoding U-Boot Crash Dumps](#)
 - [14.2.9. Porting Problem: cannot move location counter backwards](#)
 - [14.2.10. U-Boot Doesn't Run after Upgrading my Compiler](#)
 - [14.2.11. How Can I Reduce The Image Size?](#)
 - [14.2.12. Erasing Flash Fails](#)
 - [14.2.13. Ethernet Does Not Work](#)
 - [14.2.14. Where Can I Get a Valid MAC Address from?](#)
 - [14.2.15. Why do I get TFTP timeouts?](#)
 - [14.2.16. Why is my Ethernet operation not reliable?](#)
 - [14.2.17. How the Command Line Parsing Works](#)
 - [14.2.17.1. Old, simple command line parser](#)
 - [14.2.17.2. Hush shell](#)
 - [14.2.17.3. Hush shell scripts](#)
 - [14.2.17.4. General rules](#)
 - [14.2.18. How can I load and uncompress a compressed image](#)
 - [14.2.19. How can I create an uImage from a ELF file](#)
 - [14.2.20. My standalone program does not work](#)
 - [14.2.21. Linux hangs after uncompressing the kernel](#)
 - [14.2.22. How can I implement automatic software updates?](#)
 - [14.3. Linux](#)
 - [14.3.1. Linux crashes randomly](#)
 - [14.3.2. Linux crashes when uncompressing the kernel](#)
 - [14.3.3. Linux Post Mortem Analysis](#)
 - [14.3.4. Linux kernel register usage](#)
 - [14.3.5. Linux Kernel Ignores my bootargs](#)
 - [14.3.6. Cannot configure Root Filesystem over NFS](#)
 - [14.3.7. Linux Kernel Panics because "init" process dies](#)
 - [14.3.8. Unable to open an initial console](#)
 - [14.3.9. System hangs when entering User Space \(ARM\)](#)
 - [14.3.10. Mounting a Filesystem over NFS hangs forever](#)
 - [14.3.11. Ethernet does not work in Linux](#)
 - [14.3.12. Loopback interface does not work](#)
 - [14.3.13. Linux kernel messages are not printed on the console](#)
 - [14.3.14. Linux ignores input when using the framebuffer driver](#)
 - [14.3.15. How to switch off the screen saver and the blinking cursor?](#)
 - [14.3.16. BogoMIPS Value too low](#)
 - [14.3.17. Linux Kernel crashes when using a ramdisk image](#)
 - [14.3.18. Ramdisk Greater than 4 MB Causes Problems](#)
 - [14.3.19. Combining a Kernel and a Ramdisk into a Multi-File Image](#)
 - [14.3.20. Adding Files to Ramdisk is Non Persistent](#)
 - [14.3.21. Kernel Configuration for PCMCIA](#)
 - [14.3.22. Configure Linux for PCMCIA Cards using the Card Services package](#)
 - [14.3.23. Configure Linux for PCMCIA Cards without the Card Services package](#)
 - [14.3.23.1. Using a MacOS Partition Table](#)
 - [14.3.23.2. Using a MS-DOS Partition Table](#)
 - [14.3.24. Boot-Time Configuration of MTD Partitions](#)
 - [14.3.25. Use NTP to synchronize system time against RTC](#)

- [14.3.26. Configure Linux for XIP \(Execution In Place\)](#)
 - [14.3.26.1. XIP Kernel](#)
 - [14.3.26.2. Cramfs Filesystem](#)
 - [14.3.26.3. Hints and Notes](#)
 - [14.3.26.4. Space requirements and RAM saving, an example](#)
- [14.3.27. Use SCC UART with Hardware Handshake](#)
- [14.3.28. How can I access U-Boot environment variables in Linux?](#)
- [14.3.29. The =appWeb= server hangs *OR* /dev/random hangs](#)
- [14.3.30. Swapping over NFS](#)
- [14.3.31. Using NFSv3 for NFS Root Filesystem](#)
- [14.3.32. Using and Configuring the SocketCAN Driver](#)
- [14.3.33. Telnet / SSH \(dropbear\) server not working](#)
- [14.4. Self](#)
 - [14.4.1. How to Add Files to a SELF Ramdisk](#)
 - [14.4.2. How to Increase the Size of the Ramdisk](#)
- [14.5. RTAI](#)
 - [14.5.1. Conflicts with asm clobber list](#)
- [14.6. BDI2000](#)
 - [14.6.1. Where can I find BDI2000 Configuration Files?](#)
 - [14.6.2. How to Debug Linux Exceptions](#)
 - [14.6.3. How to single step through "RFT" instruction](#)
 - [14.6.4. Setting a breakpoint doesn't work](#)
 - [14.6.5. Remote 'g' packet reply is too long](#)
- [14.7. Motorola LITE5200 Board](#)
 - [14.7.1. LITE5200 Installation Howto](#)
 - [14.7.2. USB does not work on Lite5200 board](#)

14. FAQ - Frequently Asked Questions

This is a collection of questions which came up repeatedly. Give me more feedback and I will add more stuff here.

The items are categorized whether they concern U-Boot itself, the Linux kernel or the [SELF](#) framework.

14.1. [ELDK](#)

14.1.1. [ELDK](#) Installation under [FreeBSD](#)

Question:

How can I install [ELDK](#) on a [FreeBSD](#) system?

Answer:

[Thanks to Rafal Jaworowski for these detailed instructions.] This is a short tutorial how to host [ELDK](#) on [FreeBSD](#) 5.x and 6.x. The procedure described below was tested on 5.2.1, 5.3 and 6-current releases; we assume the reader is equipped with the [ELDK](#) 3.x CDROM or ISO image for installation, and is familiar with [FreeBSD](#) basic administration tasks like ports/packages installation.

1. Prerequisites:

1. Install `linux_base`

The first step is to install the Linux compatibility layer from ports `/usr/ports/emulators/linux_base/` or packages

`ftp://ftp.freebsd.org/pub/FreeBSD/ports/i386/packages/emulators/`

☐ Please make sure to install version 7.1_5 (`linux_base-7.1_5.tbz`) or later; in particular, version 6.1.5 which can also be found in the ports tree does **not** work properly!

The compatibility layer is activated by

```
# kldload linux
```

2. Install `bash`

Since [ELDK](#) and Linux build scripts are organised around `bash` while [FreeBSD](#) does not have it in base, this shell needs to be installed either from ports `/usr/ports/shells/bash2/` or packages collection `ftp://ftp.freebsd.org/pub/FreeBSD/ports/i386/packages/shells/`

The installation puts the `bash` binary in `/usr/local/bin`. It is a good idea to create a symlink in `/bin` so that hash bang from scripts (`#!/bin/bash`) works without modifications:

```
# cd /bin
# ln -s /usr/local/bin/bash
```

2. Prepare [ELDK](#)

☐ This step is only needed for [ELDK](#) release 3.1 and older versions.

Copy the install files from the CDROM or ISO image to a writable location. Brand the [ELDK](#) installer as Linux ELF file:

```
# cd <elkd_install_dir>
# brandelf -t Linux ./install
```

☐ **Note:** The following workaround might be a good alternative for the tedious copying of the installation CDROM to a writable location and manual branding: you can set a fallback branding in [FreeBSD](#) - when the loader cannot recognise the ELF brand it will switch to the last resort defined.

```
# sysctl -w kern.elf32.fallback_brand=3
kern.elf32.fallback_brand: -1 -> 3
```

With this setting, the normal [ELDK](#) CDROM images should work.

3. Install [ELDK](#) normally as described in [3.5.3. Initial Installation](#)
4. Set environment variables and `PATH` as needed for [ELDK](#) (in `bash`); for example:

```
bash$ export CROSS_COMPILE=ppc_8xx-
bash$ export PATH=${PATH}:/opt/eldk/bin:/opt/eldk/usr/bin
```

5. Hints for building U-Boot:

[FreeBSD](#) normally uses BSD-style `'make'` in base, but in order to compile U-Boot `'gmake'` (GNU make) has to be used; this is installed as part of the `'linux_base'` package (see above).

U-Boot should build according to standard [ELDK](#) instructions, for example:

```
bash$ cd /opt/eldk/ppc_8xx/usr/src/u-boot-1.1.2
```

```
bash$ gmake TQM823L_config
bash$ gmake all
```

6. Hints for building Linux:

There are three issues with the Makefile in the Linux kernel source tree:

- o GNU make has to be used.
- o The 'expr' utility in [FreeBSD](#) base behaves differently from the version than is used in Linux so we need to modify the Makefile to explicitly use the Linux version (which is part of the Linux compatibility package). This is best achieved with defining "EXPR = /compat/linux/usr/bin/expr" somewhere at =Makefile's beginning and replacing all references to 'expr' with the variable \${EXPR}.
- o Some build steps (like when running 'scripts/mkdep' can generate very long arguments lists (especially is the Linux kernel tree is in a directory with long absolute filenames). A solution is to use xargs to split such long commands into several with shorter argument lists.

The Linux kernel can then be built following the standard instructions, for example:

```
bash$ cd /opt/eldk/ppc_8xx/usr/src/linux-2.4.25/
bash$ gmake mrproper
bash$ gmake TQM823L_config
bash$ gmake oldconfig
bash$ gmake dep
bash$ gmake -j6 uImage
```

ELDK Installation Hangs

Question:

I try to install the [ELDK](#) on a Linux PC, and the installation hangs. It starts fine, but then it freezes like this:

```
...
Preparing... ##### [100%]
 1:db4-devel-ppc_4xx ##### [100%]
Preparing... ##### [100%]
 1:db4-utils-ppc_4xx ##### [100%]
Preparing... ##### [100%]
 1:glib2-ppc_4xx ##### [100%]
Preparing... ##### [100%]
 1:glib2-devel-ppc_4xx ##### [100%]
Preparing... ##### [100%]

<hangs here>
```

Answer:

This is almost certainly a FUTEX problem. To verify this, please wait until the process grinds to a halt, then use ps to find the pid of the "rpm" process that was started by the "install" program (use "ps -axf" which gives you a nice hierarchy, look for the "install" process, then for "rpm") and then attach to it with "strace -p". Most probably you will see the something like this:

```
# strace -p 21197
Process 21197 attached - interrupt to quit
futex(0x96fe17c, FUTEX_WAIT_PRIVATE, 1, NULL
```

i. e. the process is hanging in a futex call.

We have seen this more than once with differing Linux systems, but unfortunately we don't know a clean and reliable way to fix it yet. We suspect that it is a kernel/libc combination problem because it usually went away usually after changing the exact used kernel version.

The only workaround we can recommend so far is to update your host system and install more recent versions of the Linux kernel and/or the glibc C library (assuming such are available for your Linux distribution; if not, falling back to a previous kernel version may help, too).

Note: This is only needed for the installer, the problem does not happen with the regular use of the [ELDK](#).

14.1.3. .gvfs: Permission Denied

Question:

When trying to install the [ELDK](#), I get error messages like this for each and every package that gets installed:

```
Preparing... ##### 100%
1: rpm... ##### 100%
Error: Failed to stat /home/wd/.gvfs: Permission Denied
```

This happens even though I run the installer as root.

Answer:

Even though flagged as an error, these messages are harmless warnings that can be safely ignored. Before the RPM tool starts to install a package, it checks if there is sufficient space for it in the file system. Unfortunately it is dumb and checks all mounted file systems for space, but the permissions of the ".gvfs" directory (the mount point for the Gnome Virtual File System) do not permit this.

Note:

Actually the messages are not printed despite the fact that you are running as root, but **because** you run as root. You have permissions to check the "\$HOME/.gvfs" directory, while root gets an error:

```
$ df -h /home/wd/.gvfs
Filesystem      Size  Used Avail Use% Mounted on
gvfs-fuse-daemon 0      0      0  -  /home/wd/.gvfs
$ sudo df -h /home/wd/.gvfs
df: '/home/wd/.gvfs': Permission denied
df: no file systems processed
```

14.1.4. Installation on Local Harddisk

Question:

I have a local harddisk drive connected to my target board. Can I install the [ELDK](#) on it and run it like a standard Linux distribution?

Answer:

Yes, this is possible. It requires only minor adjustments. The following example assumes you are using a SCSI disk drive, but the same can be done with standard SATA or PATA drives, too:

1. Boot the target with root file system over NFS.
2. Create the necessary partitions on your disk drive: you need at last a swap partition and a file system partition.

```
bash-3.00# fdisk -l

Disk /dev/sda: 36.9 GB, 36951490048 bytes
64 heads, 32 sectors/track, 35239 cylinders
Units = cylinders of 2048 * 512 = 1048576 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1            1           978     1001456   82  Linux swap / Solaris
/dev/sda2           979        12423     11719680   83  Linux
/dev/sda3        12424        23868     11719680   83  Linux
```



```
/dev/sda4      23869      35239      11643904      83  Linux
```

3. Format the partitions:

```
bash-3.00# mkswap /dev/sda1
bash-3.00# mke2fs -j -m1 /dev/sda2
```

4. Mount the file system:

```
bash-3.00# mount /dev/sda2 /mnt
```

5. Copy the content of the (NFS) root file system into the mounted file system:

```
bash-3.00# tar --one-file-system -c -f - / | ( cd /mnt ; tar xpf - )
```

6. Adjust `/etc/fstab` for the disk file system:

```
bash-3.00# vi /mnt/etc/fstab
bash-3.00# cat /mnt/etc/fstab
/dev/sda2      /          ext3          defaults      1 1
/dev/sda1      swap       swap          defaults      0 0
proc           /proc     proc          defaults      0 0
sysfs          /sys      sysfs         defaults      0 0
```

7. Adjust `/etc/rc.sysinit` for running from local disk; remove the following comments:

```
bash-3.00# diff -u /mnt/etc/rc.sysinit.ORIG /mnt/etc/rc.sysinit
--- /mnt/etc/rc.sysinit.ORIG      2007-01-21 04:37:00.000000000 +0100
+++ /mnt/etc/rc.sysinit 2007-03-02 10:58:22.000000000 +0100
@@ -460,9 +460,9 @@

# Remount the root filesystem read-write.
update_boot_stage RMountfs
-#state=`LC_ALL=C awk '/ \/ / && ($3 != /rootfs/) { print $4 }' /proc/mounts`
-#[ "$state" != "rw" -a "$READONLY" != "yes" ] && \
-# action $"Remounting root filesystem in read-write mode: " mount -n -o remount,rw /
+state=`LC_ALL=C awk '/ \/ / && ($3 != /rootfs/) { print $4 }' /proc/mounts`
+[ "$state" != "rw" -a "$READONLY" != "yes" ] && \
+ action $"Remounting root filesystem in read-write mode: " mount -n -o remount,rw /

# Clean up SELinux labels
if [ -n "$SELINUX" ]; then
```

8. Unmount disk:

```
bash-3.00# umount /mnt
```

9. Reboot, and adjust boot arguments to use disk partition as root file system

```
=> setenv diskargs setenv bootargs root=/dev/sda2 ro
=> setenv net_disk 'tftp ${loadaddr} ${bootfile};run diskargs addip addcons;bootm'
=> saveenv
```

10. Boot with these settings

```
=> run net_disk
```

14.1.5. System Include Files Missing

Question:

when installing [ELDK](#) on Ubuntu 6.06 dapper drake I get the following error messages....

```
Preparing... ##### [100%]
1:kernel-source-ppc_6xx ##### [100%]
Configuring kernel...
scripts/basic/fixdep.c:107:23: error: sys/types.h: No such file or directory
scripts/basic/fixdep.c:108:22: error: sys/stat.h: No such file or directory
scripts/basic/fixdep.c:109:22: error: sys/mman.h: No such file or directory
...
```

Answer:

The Linux installation on your host is missing essential files that are needed to perform software development and use a C compiler. On Ubuntu, check for example if you miss a "libc6-dev" package. The specific package name differs from distribution to distribution; on Fedora, you need for example the "glibc-headers" package.

If you want to work with a Linux kernel you will probably also need other packages.

14.1.6. patch: command not found

Question:

When installing [ELDK](#) on Ubuntu I get the following error message:

```
...prepare-kernel.sh: line 376: patch: command not found
```

Answer:

The error message contains clear hints for the solution: the "patch" command cannot be found on your system, so most probably it has not been installed yet. Please try:

```
$ sudo apt-get install patch
```

14.1.7. [ELDK](#) Include Files Missing

Question:

After configuring and compiling a Linux kernel in the kernel source tree that comes with the [ELDK](#), I cannot compile user space programs any more - I get error messages because many #include file like <errno.h> etc. are missing.
This is with [ELDK](#) 4.0 or 4.1.

Answer:

This problem is caused by the way how the [ELDK](#) is packaged. At the moment, the [ELDK](#) kernel headers are not packed into a separate "kernel-headers" RPM to avoid duplication, because the kernel source tree is always installed. Instead, the [ELDK](#) "kernel-headers" package is just a set of symlinks. This worked fine in the past, but fails with the new support for ARCH=powerpc systems.

The next version of the [ELDK](#) will contain a real kernel-headers RPM, which will fix this problem.

As a workaround on current systems, you can install the real kernel include files into the "include/asm", "include/linux" and "include/mtd" directories.

To do this, the following commands can be used:

```
bash$ <eldkroot>/bin/rpm -e kernel-headers-ppc_<target>
bash$ cd <eldkroot>/ppc_<target>
bash$ rm usr/include/asm
bash$ tar -xvzf kernel-headers-powerpc.tar.gz
```

The tarball mentioned above can be downloaded [here](#). It contains the include files that get installed by running the "make ARCH=powerpc headers_install" command in the Linux kernel tree.

This problem is fixed in [ELDK](#) 4.2 and later releases.

14.1.8. Using the [ELDK](#) on a 64 bit platform

As the [ELDK](#) is compiled for 32-bit host systems, a compatibility layer is required on 64-bit systems. This package is usually called `ia32-libs`. So on a Debian or Ubuntu system a

```
sudo apt-get install ia32-libs
```

should be enough to make the [ELDK](#) work.

On the U-Boot mailing list, it was reported that for a 64 bit Fedora 11 the following should be enough:

```
sudo yum -y install glibc.i686 zlib.i686
```

14.1.9. How can I check if Floating Point support is working?

Question:

The floating point performance of my P2020 QorIQ processor is really poor. I am not using the [ELDK](#), but a tool chain from FOOBAR. Can this be a problem? What can I do to verify this?

Answer:

The P20xx QorIQ processors use an e500v2 core which does *not* include a normal Floating Point Unit (FPU), but instead a *Signal Processing Engine* (SPE Version 2). You can run FP calculations on the SPE, but there are no special FP registers available as on a normal FPU, so General Purpose Registers must be used for passing of FP operands. While this is still much faster than pure soft-float emulation, it is missing the advantages and the speed of a full-blown, separate standard FPU with a full FP register set.

Also, your tool chain needs to be aware of this feature, and must contain support for it. Eventually special compiler options are needed - check your documentation.

With the [ELDK](#), the needed settings are automatically pre-set when you just chose the correct target architecture packages, cf. [3.4. Supported Target Architectures](#)

To test what your tool chain is doing, you best compile a small test program and check the generated code. The following examples were done with [ELDK](#) 4.2 for [Power Architecture](#)® targets:

1. Test Program:

```
$ cat fp_test.c
double foo (double x, double y)
{
    double z;

    z = (x + y) / (x * y);

    return z;
}
```

2. Build for normal FPU support (using the `ppc_6xx` target architecture):

```
$ export CROSS_COMPILE=ppc_6xx-
$ ppc_6xx-gcc -S -O fp_test.c
```

Check results:

```
$ cat fp_test.s
.file "fp_test.c"
.section ".text"
.align 2
.globl foo
.type foo, @function
foo:
    fadd 0,1,2
    fmul 1,1,2
    fdiv 1,0,1
    blr
.size foo, .-foo
.ident "GCC: (GNU) 4.2.2"
.section .note.GNU-stack,"",@progbits
```

The use of floating point machine instructions ("fadd", "fmul", "fdiv") and the fact that no additional register use is needed is a clear indication that full support for the hardware FPU is available in this configuration.

3. Build for soft-float emulation (using the `ppc_8xx` target architecture):

```
$ export CROSS_COMPILE=ppc_8xx-
$ ppc_8xx-gcc -S -O fp_test.c
```

```
$ cat fp_test.s
.file "fp_test.c"
.globl __adddf3
.globl __muldf3
.globl __divdf3
.section ".text"
.align 2
.globl foo
.type foo, @function
foo:
    stwu 1,-48(1)
    mflr 0
    stw 24,16(1)
    stw 25,20(1)
    stw 26,24(1)
    stw 27,28(1)
    stw 28,32(1)
    stw 29,36(1)
    stw 0,52(1)
    mr 28,3
    mr 29,4
    mr 26,5
    mr 27,6
    bl __adddf3
    mr 24,3
    mr 25,4
```

```

mr 3,28
mr 4,29
mr 5,26
mr 6,27
bl __muldf3
mr 5,3
mr 6,4
mr 3,24
mr 4,25
bl __divdf3
lwz 0,52(1)
mtlr 0
lwz 24,16(1)
lwz 25,20(1)
lwz 26,24(1)
lwz 27,28(1)
lwz 28,32(1)
lwz 29,36(1)
addi 1,1,48
blr
.size    foo,.-foo
.ident   "GCC: (GNU) 4.2.2"
.section .note.GNU-stack,"",@progbits

```

The fact that the compiler is calling helper functions (`__adddf3`, `__muldf3`, `__divdf3`) combined with heavy use of the General Purpose Registers is a clear indication for software-emulated FP support - and explains why this is so slow compared to a real FPU.

4. Build for SPE v2 support (as needed for example for a P2020 QorIQ processor, using the `ppc_85xxDP` target architecture):

```

$ export CROSS_COMPILE=ppc_85xxDP-
$ ppc_85xxDP-gcc -S -O fp_test.c

$ cat fp_test.s
.file    "fp_test.c"
.section    ".text"
.align 2
.globl foo
.type     foo, @function
foo:
    stwu 1,-48(1)
    stw 3,8(1)
    stw 4,12(1)
    stw 5,16(1)
    stw 6,20(1)
    evmergelo 0,3,4
    evmergelo 9,5,6
    efdadd 11,0,9
    efdmul 0,0,9
    efddiv 11,11,0
    evstdd 11,24(1)
    evmergehi 9,11,11
    mr 10,11
    stw 9,32(1)
    stw 10,36(1)
    mr 3,9
    mr 4,10
    addi 1,1,48
    blr
.size    foo,.-foo
.ident   "GCC: (GNU) 4.2.2"
.section .note.GNU-stack,"",@progbits

```

Here we can see moderate use of General Purpos Registers combined with the use of SPE machine instructions (`evmergelo`, `efdadd`, `efdmul`, `efddiv`, `evstdd`, `evmergehi`) which proves that the compiler really generates code that supports the SPE.

14.1.10. [ELDK 2.x](#) Installation Aborts

Question:

I tried to install [ELDK](#) version 2.x on a [SuSE](#) 8.2 / [SuSE](#) 9 / [RedHat](#)-9 Linux host but failed - it terminated without installing any packages. Why?

Answer:

Newer Linux distributions use libraries that are incompatible to those used by the [ELDK](#)'s installation tools. This problem was fixed in later releases of the [ELDK](#) (version 3.0 and later). It is therefore recommended to use a more recent version of the [ELDK](#). If you really want to install an old version, the following back-port is available:

Please download the file <ftp://ftp.denx.de/pub/tmp/ELDK-update-2.2.0.tar.bz2>

Then change into the source tree with the [ELDK](#) files and perform the following operations:

```

bash$ rm RPMS/rpm-4.0.3-1.03b_2.i386.rpm \
    RPMS/rpm-build-4.0.3-1.03b_2.i386.rpm \
    RPMS/rpm-devel-4.0.3-1.03b_2.i386.rpm \
    tools/usr/lib/rpm/rpmpopt-4.0.3
bash$ tar jxf /tmp/ELDK-update-2.2.0.tar.bz2

```

Then build the ISO image as documented, and try again.

14.1.11. Enable SSH Access

Question:

How can I enable SSH access to the target system when running with the [ELDK](#) file system mounted over NFS?

Answer:

The [ELDK](#) includes the dropbear SSH server and client packages (see <http://matt.ucc.asn.au/dropbear/dropbear.html>). To enable SSH access, you must first generate the SSH host keys for RSA and DSS:

```

# dropbearkey -t rsa -f /etc/dropbear/dropbear_rsa_host_key
Will output 1024 bit rsa secret key to '/etc/dropbear/dropbear_rsa_host_key'
Generating key, this may take a while...
Public key portion is:
ssh-rsa ...
Fingerprint: md5 ...
# dropbearkey -t dss -f /etc/dropbear/dropbear_dss_host_key
Will output 1024 bit dss secret key to '/etc/dropbear/dropbear_dss_host_key'
Generating key, this may take a while...
Public key portion is:
ssh-dss ...
Fingerprint: md5 ...

```

Then you can start the dropbear daemon:

```
# dropbear
```

Now you should be able to access the target system through SSH.

14.2. U-Boot

14.2.1. Can U-Boot be configured such that it can be started in RAM?

Question:

I don't want to erase my flash memory because I'm not sure if my new U-Boot image will work. Is it possible to configure U-Boot such that I can load it into RAM instead of flash, and start it from my old boot loader?

Answer:

No. (Unless you're using a Blackfin processor, but you probably aren't.)

Question:

But I've been told it **is** possible??

Answer:

Well, yes. Of course this is possible. This is software, so **everything** is possible. But it is difficult, unsupported, and fraught with peril. You are on your own if you choose to do it. And it will not help you to solve your problem.

Question:

Why?

Answer:

U-Boot expects to see a virgin [CPU](#), i. e. the [CPU](#) state must match what you see if the processor starts executing the first instructions when it comes out of reset. If you want to start U-Boot from another boot loader, you must disable a lot of code, i. e. all initialization parts that already have been performed by this other boot loader, like setting up the memory controller, initializing the [SDRAM](#), initializing the serial port, setting up a stack frame etc. Also you must disable the relocation to RAM and adjust the link addresses etc.

This requires a **lot** of experience with U-Boot, and the fact that you had to ask if this can be done means that you are not in a position to do this.

The code you have to disable contains the most critical parts in U-Boot, i. e. these are the areas where 99% or more of all errors are located when you port U-Boot to a new hardware. In the result, your RAM image may work, but in the end you will need a full image to program the flash memory with it, and then you will have to enable all this highly critical and completely untested code.

You see? You **cannot** use a RAM version of U-Boot to avoid testing a flash version, so you can save all this effort and just burn your image to flash.

Question:

So how can I test an U-Boot image and recover my system if it doesn't work?

Answer:

Attach a BDI2000 (or any appropriate [JTAG](#) ICE) to your board, burn the image to flash, and debug it in its natural environment, i.e. U-Boot being the boot loader of the system and taking control over the [CPU](#) right as it comes out of reset. If something goes wrong, erase the flash and program a new image. This is a routine job using a BDI2000.

14.2.2. Relocation cannot be done when using -mrelocatable

Question:

I use [ELDK](#) version 3.0. When I build U-Boot I get error messages like this:

```
{standard input}: Assembler messages:
{standard input}:4998: Error: Relocation cannot be done when using -mrelocatable
...
```

Answer:

[ELDK](#) 3.0 uses GCC-3.2.2; your U-Boot sources are too old for this compiler. GCC-3.x requires a few adaption which were added in later versions of U-Boot. Use for example the source tree (1.0.2) which is included with the [ELDK](#), or download the latest version from [CVS](#).

14.2.3. Source object has [EABI](#) version 4, but target has [EABI](#) version 0

Question:

When trying to build U-Boot with an [EABI](#) compliant tool chain, I get such error messages:

```
arm-ld: ERROR: Source object ... has EABI version 4, but target ... has EABI version 0
```

What does that mean, and how can I fix that?

Answer:

"EABI version 0" means the "apcs-gnu" [ABI](#), while "EABI version 4" is the "aapcs-linux" [ABI](#), aka "gnueabi".

All U-Boot ARM sources are built with "-mapcs-gnu" option set in "cpu/arm/config.mk", while libgcc.a modules are built in "gnueabi" format, which is for example the ARM GCC default in [ELDK](#) Release 4.2.

So the real problem is compatibility between toolchain [ABI](#) and U-Boot ARM [ABI](#). In the Linux kernel there is a special kernel config option for [EABI](#)-enabled tool chains (CONFIG_AEABI), which enables special pieces of code in ARM assembler modules. We could follow this approach, reworking existing assembler sources and respective config.mk files in U-Boot.

Alternatively, the tool chain could provide a separate version of libgcc.a built with old [ABI](#). This could be done using the multilib approach. The advantage here is that no U-boot changes will be required.

14.2.4. U-Boot crashes after relocation to RAM

Question:

I have ported U-Boot to a custom board. It starts OK, but crashes or hangs after relocating itself to RAM. Why?

Answer:

Your [SDRAM](#) initialization is bad, and the system crashes when it tries to fetch instructions from RAM. Note that simple read and write accesses may still work, it's the **burst** mode that is failing. This only shows up when caches are enabled because cache is the primary (or only) user of burst operations in U-Boot. In Linux, burst accesses may also result from [DMA](#). For example, it is typical that a system may crash under heavy network load if the Ethernet controller uses [DMA](#) to memory.

☐ It is **NOT** sufficient to program the memory controller of your [CPU](#); each [SDRAM](#) chip also requires a specific initialization sequence which you must adhere to **to the letter** - check with the chip manufacturer's manual.

It has been observed that some operating systems like pSOS+ or VxWorks do not stress the memory subsystem as much as Linux or other UNIX systems like LynxOS do, so just because your board appears to work running another OS does not mean it is 100% OK.

Standard memory tests are not effective in identifying this type of problem because they do not cause stressful cache burst read/write operations.

With this caveat in mind, reportedly this program has found memory problems before: <http://pyropus.ca/software/memtester/>

Argument:

But my board ran fine with bootloader XYZ and/or operating system ABC.

Answer:

Double-check your configuration that you claim runs properly...

1. Are you **sure** the [SDRAM](#) is initialized using the same init sequence and values?
2. Are you **sure** the memory controlling registers are set the same?
3. Are you **sure** your other configuration uses caches and/or [DMA](#)? If it doesn't, it isn't a valid comparison.

14.2.5. Warning - bad CRC, using default environment

Question:

I have ported U-Boot to a custom board. It seems to boot OK, but it prints:

```
*** Warning - bad CRC, using default environment
```

Why?

Answer:

Most probably everything is OK. The message is printed because the flash sector or ERPROM containing the environment variables has never been initialized yet. The message will go away as soon as you save the environment variables using the `saveenv` command.

14.2.6. Net: No ethernet found

Question:

I have ported U-Boot to a custom board. It seems to boot OK, but it prints:

```
Net:      No ethernet found.
```

Why?

Answer:

Some network drivers (especially on ppc4xx) respond with such a message when no valid [MAC](#) address has been defined. Please make sure that a valid [MAC](#) address has been defined in the environment (using the `setenv` command). Then store the environment (using the `saveenv` command). After the next reboot, the Ethernet interface should be available.

14.2.7. Wrong debug symbols after relocation

Question:

I want to debug U-Boot after relocation to RAM, but it doesn't work since all the symbols are at wrong addresses now.

Answer:

To debug parts of U-Boot that are running from ROM/flash, i. e. **before** relocation, just use a command like `powerpc-linux-gdb uboot` as usual.

For parts of U-Boot that run from RAM, i. e. **after** relocation, use `powerpc-linux-gdb` *without* arguments, and use the `add-symbol-file` command in GDB to load the symbol table at the relocation address in RAM. The only problem is that you need to know that address, which depends on RAM size, length reserved for U-Boot, size of "protected RAM" area, etc. If in doubt, enable DEBUG mode when building U-Boot so it prints the address to the console.

☐ Hint: I use definitions like these in my `.gdbinit` file:

```
define rom
    symbol-file
    file u-boot
end

define ram
    symbol-file
    add-symbol-file u-boot 0x01fe0000
end
```

Note: when you want to switch modes during one debug session (i. e. without restarting GDB) you can "delete" the current symbol information by using the `symbol-file` command without arguments, and then either using `symbol-file u-boot` for code before relocation, or `add-symbol-file u-boot _offset_` for code after relocation.

14.2.8. Decoding U-Boot Crash Dumps

When you are porting U-Boot to new hardware, or implementing extensions, you might run into situations where U-Boot crashes and prints a register dump and a stack trace, for example like this:

```
Bus Fault @ 0x00f8d70c, fixup 0x00000000
Machine check in kernel mode.
Caused by (from msr): regs 00f52cf8 Unknown values in msr
NIP: 00f8d70c XER: 0000005f LR: 00f8d6f4 REGS: 00f52cf8 TRAP: 0200 DAR: f9f68c00
MSR: 00009002 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 00

GPR00: 00016acc 00f52de8 00000000 f9f68c00 00fa38ec 00000001 f9f68bf8 0000000b
GPR08: 00000002 00f55470 00000000 00f52d94 44004024 00000000 00fa2f00 c0f75000
GPR16: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24: 00000000 00fa38ec 00f553c0 00f55480 00000000 00f52f80 00fa41c0 00000001
Call backtrace:
00000000 00f8f998 00f8fa88 00f8faf8 00f90b5c 00f90cf8 00f8385c
00f79e6c 00f773b0
machine check
```

To find out what happened, you can try to decode the stack backtrace (the list of addresses printed after the `Call backtrace:` line. The [backtrace](#) tool can be used for this purpose. However, there is a little problem: the addresses printed for the stack backtrace are **after relocation** of the U-Boot code to RAM; to use the `backtrace` tool you need to know U-Boot's *address offset* (the difference between the start address of U-Boot in flash and its relocation address in RAM).

The easiest way to find out the relocation address is to enable debugging for the U-Boot source file `lib_*/board.c` - U-Boot will then print some debug messages

```
...
Now running in RAM - U-Boot at: 00f75000
...
```

Now you have to calculate the address offset between your link address (The value of the `TEXT_BASE` definition in your `board/?.config.mk` file). In our case this value is `0x40000000`, so the address offset is `0x40000000 - 0x00f75000 = 0x3f08b000`

Now we use the `backtrace` script with the `System.map` file in the U-Boot source tree and this address offset:

```
-> backtrace System.map 0x3f08b000
Reading symbols from System.map
```

```

Using Address Offset 0x3f08b000
0x3f08b000 -- unknown address
0x4001a998 -- 0x4001a8d0 + 0x00c8   free_pipe
0x4001aa88 -- 0x4001aa2c + 0x005c   free_pipe_list
0x4001aaf8 -- 0x4001aad0 + 0x0028   run_list
0x4001bb5c -- 0x4001ba68 + 0x00f4   parse_stream_outer
0x4001bcf8 -- 0x4001bcd8 + 0x0020   parse_file_outer
0x4000e85c -- 0x4000e6f8 + 0x0164   main_loop
0x40004e6c -- 0x40004b9c + 0x02d0   board_init_r
0x400023b0 -- 0x400023b0 + 0x0000   trap_init

```

In this case the last "good" entry on the stack was in `free_pipe...`

14.2.9. Porting Problem: cannot move location counter backwards

Question:

I'm trying to port U-Boot to a new board and the linker throws an error message like this:

```
board/<your_board>/u-boot.lds:75 cannot move location counter backwards (from 00000000b0008010 to 00000000b0008000)
```

Answer:

Check your linker script `board/your_board/u-boot.lds` which controls how the object files are linked together to build the U-Boot image.

It looks as if your board uses an "embedded" environment, i. e. the flash sector containing the environment variables is surrounded by code. The `u-boot.lds` tries to collect as many as possible code in the first part, making the gap between this first part and the environment sector as small as possible. Everything that does not fit is then placed in the second part, after the environment sector.

Some your modifications caused the code that was put in this first part to grow, so that the linker finds that it would have to overwrite space that is already used.

Try commenting out one (or more) line(s) *before* the line containing the `"common/environment.o"` statement. [`"lib_generic/zlib.o"` is usually a good candidate for testing as it's **big**]. Once you get U-Boot linked, you can check in the `u-boot.map` file how big the gap is, and which object files could be used to fill it up again.

14.2.10. U-Boot Doesn't Run after Upgrading my Compiler

Question:

I encountered a big problem that U-Boot 1.1.4 compiled by [ELDK](#) 4.1 for MPC82xx crashed.

But if I build it using gcc-3.4.6 based cross tools, U-Boot on my board boots correctly.

The same U-Boot code built by [ELDK](#) 4.1 (gcc-4.0) failed, nothing occurs on the serial port.

Answer:

This is often a missing `volatile` attribute on shared variable references, particularly hardware registers. Newer compiler versions optimize more aggressively, making missing `volatile` attributes visible.

If you use -O0 (no optimization) does it fix the problem?

If it does, it most likely is an optimization/volatile issue. The hard part is figuring out where. Device handling and board-specific code is the place to start.

14.2.11. How Can I Reduce The Image Size?

Question:

I am trying to reduce the size of the `u-boot.bin` file so that it fits into 256 KB. I disabled all the drivers that I didn't need but the binary size is still 512KB, it seems to be a hard number coded in somewhere. Where can the image size be altered from?

Answer:

Some processors have a fixed reset vector address at `0xFFFFF0FC`, so the U-Boot image has to include that address, i. e. it covers the full range from the start address to the end of the 32 bit address space. In such a case, the start address must be changed - check the setting of `TEXT_BASE` in your board/*<name>*/config.mk file.

14.2.12. Erasing Flash Fails

Question:

I tried to erase the flash memory like

```
erase 40050000 40050100
```

It fails. What am I doing wrong?

Answer:

Remember that flash memory cannot be erased in arbitrary areas, but only in so called "erase regions" or "sectors". If you have U-Boot running you can use the `flinfo` (Flash information, short `fli`) command to print information about the flash memory on your board, for instance:

```

=> fli

Bank # 1: AMD AM29LV160B (16 Mbit, bottom boot sect)
Size: 4 MB in 35 Sectors
Sector Start Addresses:
 40000000 (RO) 40008000 (RO) 4000C000 (RO) 40010000 (RO) 40020000 (RO)
 40040000      40060000      40080000      400A0000      400C0000
 400E0000      40100000      40120000      40140000      40160000
 40180000      401A0000      401C0000      401E0000      40200000
 40220000      40240000      40260000      40280000      402A0000
 402C0000      402E0000      40300000      40320000      40340000
 40360000      40380000      403A0000      403C0000      403E0000

```

In the example above, the area `40050000 ... 40050100` lies right in the middle of a erase unit (`40040000 ... 4005FFFF`), so you cannot erase it without erasing the whole sector, i. e. you have to type

```
=> erase 40040000 4005FFFF
```

Also note that there are some sectors marked as read-only (`(RO)`); you cannot erase or overwrite these sectors without un-protecting the sectors first (see the U-Boot `protect` command).

14.2.13. Ethernet Does Not Work

Question:

Ethernet does not work on my board.

Answer:

Maybe you forgot to set a [MAC](#) address? Check if the "ethaddr" environment variable is defined, and if it has a sane value. If there are more than one Ethernet interfaces on your board, you may also have to check the [MAC](#) addresses for these, i. e. check the "eth1addr", "eth2addr", etc. variables, too.

Question:

I have configured a [MAC](#) address of 01:02:03:04:05:06, and I can see that an ARP packet is sent by U-Boot, and that an ARP reply is sent by the server, but U-Boot never receives any packets. What's wrong?

Answer:

You have chosen a [MAC](#) address which, according to the ANSI/IEEE 802-1990 standard, has the multicast bit set. Under normal conditions a network interface discards such packets, and this is what U-Boot is doing. This is not a bug, but correct behaviour.

Please use only valid [MAC](#) addresses that were assigned to you.

For bring-up testing in the lab you can also use so-called *locally administered ethernet addresses*. These are addresses that have the 2nd LSB in the most significant byte of [MAC](#) address set. The `gen_eth_addr` tool that comes with U-Boot (see "tools/gen_eth_addr") can be used to generate random addresses from this pool.

14.2.14. Where Can I Get a Valid [MAC](#) Address from?

Question:

Where can I get a valid [MAC](#) address from?

Answer:

You have to buy a block of 4096 [MAC](#) addresses (IAB = Individual Address Block) or a block of 16M [MAC](#) addresses (OUI = Organizationally Unique Identifier, also referred to as 'company id') from IEEE Registration Authority. The current cost of an IAB is \$550.00, the cost of an OUI is \$1,650.00. See <http://standards.ieee.org/regauth/oui/index.shtml>

You can buy Eeproms containing [MAC](#) addresses from: [Maxim](#) or [Microchip](#).

You can set the "locally administered" bit to make your own [MAC](#) address (no guarantee of uniqueness, but pretty good odds if you don't do something dumb). Ref: [Wikipedia](#)

Universally administered and locally administered addresses are distinguished by setting the second least significant bit of the most significant byte of the address. If the bit is 0, the address is universally administered. If it is 1, the address is locally administered. The bit is 0 in all OUIs. For example, 02-00-00-00-00-01. The most significant byte is 02h. The binary is 00000010 and the second least significant bit is 1. Therefore, it is a locally administered address.

In U-Boot, you can use the "gen_eth_addr" tool to generate a random "locally administered" [MAC](#) address. Here are the needed commands:

```
$ make tools/gen_eth_addr
cc      tools/gen_eth_addr.c  -o tools/gen_eth_addr
$ tools/gen_eth_addr
ba:d0:4a:9c:4e:ce
```

14.2.15. Why do I get [TFTP](#) timeouts?

Question 1:: When trying to download a file from the [TFTP](#) server I always get timeouts like these:

```
...
Loading: #####T #####T#####T#####T #####T ##T #
#####T #T #####T #####T #####T #####T ##T #####T #####T #####T #####T
#####T ##T #####T #####T #####T #####T ##T #####T #####T #####T
#####T
done
```

If the target is connected directly to the host PC (i. e. without a switch inbetween) the problem goes away or is at least less incisive.

What's wrong?

Answer 1:: Most probably you have a full duplex/half duplex problem. Verify that U-Boot is setting the ethernet interface on your board to the proper duplex mode (full/half). I'm guessing your board is half duplex but your switch is full (typical of a switch ;-).

The switch sends traffic to your board while your board is transmitting... that is a collision (late collision at that) to your board but is OK to the switch. This doesn't happen nearly as much with a direct link to your PC since then you have a dedicated link without much asynchronous traffic.

The software (U-Boot/Linux) needs to poll the [PHY](#) chip for duplex mode and then (re)configure the [MAC](#) chip (separate or built into the [CPU](#)) to match. If the poll isn't happening or has a bug, you have problems like described above.

Question 2:: When I use tftp, there are some problems. My terminal always displays "Loading: T T T T T T T T T T T T T T T T T". The whole information as follows:

```
U-Boot 1.1.4_XT (Jun  6 2006 - 17:36:18)
U-Boot code: 0C300000 -> 0C31AD70 BSS: -> 0C31EF98
RAM Configuration:
Bank #0: 0c000000  8 MB
Bank #1: 0c800000  8 MB
Flash:  2 MB
*** Warning - bad CRC, using default environment
In:      serial
Out:     serial
Err:     serial
Hit any key to stop autoboot:  0
XT=> help tftp
tftpboot [loadAddress] [bootfilename]
XT=> tftpboot 0x0c700000 image.bin
TFTP from server 192.168.0.23; our IP address is 192.168.0.70
Filename 'image.bin'.
Load address: 0xc700000
Loading: T T T T T T T T T T T T T T T T T
Retry count exceeded; starting again
TFTP from server 192.168.0.23; our IP address is 192.168.0.70
```

Would someone give me some suggestions?

Answer 2:: (1) Verify your [TFTP](#) server is working. On a machine (not the [TFTP](#) server nor your development board) use tftp to read the target file.

```
$ tftp 192.168.0.23 get image.bin
```

If this doesn't work, fix your [TFTP](#) server configuration and make sure it is running.

(2) If your [TFTP](#) server is working, run ethereal (or equivalent ethernet sniffing) to see what ethernet packets are being sent by your development board. It usually works best to run ethereal on your [TFTP](#) server (if you run it on a different machine and you use an ethernet switch, the third machine likely won't see the tftp packets).

14.2.16. Why is my Ethernet operation not reliable?

Question:

My ethernet connection is not working reliable. On one switch it works fine, but on another one it doesn't.

or:

Question:

I always see transmit errors or timeouts for the first packet of a download, but then it works well.

or:

Question:

I cannot mount the Linux root file system over NFS; especially not with recent Linux kernel versions (older kernel versions work better). Specifying "proto=tcp" as mount option greatly improves the situation.

etc.

Answer:

There are many possible explanations for such problems. After eliminating the obvious sources (like broken cables etc.) you should check the configuration of your Ethernet [PHY](#). One common cause of problems is if your [PHY](#) is hard configured in duplex mode (for example 100baseTX Full Duplex or 10baseT Full Duplex). If such a setup is combined with a autonegotiating switch, then trouble is ahead.

[Jerry Van Baren](#) explained this as follows:

Ignoring the configuration where both ends are (presumably correctly) manually configured, you end up with five cases, two of them misconfigured and WRONG:

- 1) Autonegotiation <-> autonegotiation - reliable.
- 2) 10bT half duplex <-> autonegotiation - reliable.
- 3) 100bT half duplex <-> autonegotiation - reliable.
- 4) 10bT *FULL* duplex <-> autonegotiation - *UNreliable*.
- 5) 100bT *FULL* duplex <-> autonegotiation - *UNreliable*.

The problem that I've observed is that the *humans* (the weak links) that do the manual configuration don't understand that "parallel detection" *must be* half duplex by definition in the spec (it is hard to define a reliable algorithm to detect full duplex capability so the spec writers punted). As a result, the human invariably picks "full duplex" because everybody knows full duplex is better... and end up as case (4) or (5). They inadvertently end up with a slower unreliable link (lots of "collisions" resulting in runt packets) rather than the faster better link they thought they were picking (d'oh!). The really bad thing is that the network works fine in testing on an isolated LAN with no traffic and absolutely craps its pants when it hits the real world.

That is my reasoning behind my statement that we can generally ignore the autonegotiation <-> fixed configuration case because the odds of it working properly are poor anyway.

Rule:

Always try to set up your [PHY](#) for autonegotiation.

If you must use some fixed setting, then set it to half duplex mode.

If you really must use a fixed full-duplex setting, then you absolutely must make sure that the link partner is configured exactly the same.

14.2.17. How the Command Line Parsing Works

There are two different command line parsers available with U-Boot: the old "simple" one, and the much more powerful "hush" shell:

14.2.17.1. Old, simple command line parser

- supports environment variables (through `setenv` / `saveenv` commands)
- several commands on one line, separated by ';'
 - ☐ **NOTE:** Older versions of U-Boot used "\${...}" for variable substitution. Support for this syntax is still present in current versions, but will be removed soon. Please use "\${...}" instead, which has the additional benefit that your environment definitions are compatible with the Hush shell, too.
- special characters ('\$', ' ', ' ') can be escaped by prefixing with '\', for example:

```
setenv bootcmd bootm \${address}
```

- You can also escape text by enclosing in single apostrophes, for example:

```
setenv addip 'setenv bootargs ${bootargs} ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}:${netdev}:off'
```

14.2.17.2. Hush shell

- similar to Bourne shell, with control structures like `if...then...else...fi`, `for...do...done`, `while...do...done`, `until...do...done`, ...
- supports environment ("global") variables (through `setenv` / `saveenv` commands) and local shell variables (through standard shell syntax `name=value`); only environment variables can be used with the `run` command, especially as the variable to run (i. e. the first argument).
- In the current implementation, the local variables space and global environment variables space are separated. Local variables are those you define by simply typing like `name=value`. To access a local variable later on, you have to write '`$name`' or '`${name}`'; to execute the contents of a variable directly you can type '`$name`' at the command prompt. Note that local variables can only be used for simple commands, not for compound commands etc.
- Global environment variables are those you can set and print using `setenv` and `printenv`. To run a command stored in such a variable, you need to use the `run` command, and you *must* not use the '\$' sign to access them.
- To store commands and special characters in a variable, use single quotation marks surrounding the whole text of the variable, instead of the backslashes before semicolons and special symbols.
- Be careful when using the hash ('#') character - like with a "real" Bourne shell it is the comment character, so you have to escape it when you use it in the value of a variable.

Examples:

```
setenv bootcmd bootm \${address}
setenv addip 'setenv bootargs $bootargs ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}:${netdev}:off'
```

14.2.17.3. Hush shell scripts

Here are a few examples for the use of the advanced capabilities of the hush shell in U-Boot environment variables or scripts:

Example:

```
=> setenv check 'if imi $addr; then echo Image OK; else echo Image corrupted!!; fi'
=> print check
check=if imi $addr; then echo Image OK; else echo Image corrupted!!; fi
```



```
=> addr=0 ; run check

## Checking Image at 00000000 ...
  Bad Magic Number
Image corrupted!!
=> addr=40000 ;run check

## Checking Image at 00040000 ...
Image Name:   ARM Linux-2.4.18
Created:      2003-06-02 14:10:54 UTC
Image Type:   ARM Linux Kernel Image (gzip compressed)
Data Size:    801609 Bytes = 782.8 kB
Load Address: 0c008000
Entry Point:  0c008000
Verifying Checksum ... OK
Image OK
```

Instead of "echo Image OK" there could be a command (sequence) to boot or otherwise deal with the correct image; instead of the "echo Image corrupted!!" there could be a command (sequence) to (load and) boot an alternative image, etc.

Example:

```
=> addr1=0
=> addr2=10
=> bootm $addr1 || bootm $addr2 || tftpboot $loadaddr $loadfile && bootm
## Booting image at 00000000 ...
Bad Magic Number
## Booting image at 00000010 ...
Bad Magic Number
TFTP from server 192.168.3.1; our IP address is 192.168.3.68
Filename '/tftpboot/TRAB/uImage'.
Load address: 0xc400000
Loading: #####
          #####
done
Bytes transferred = 801673 (c3b89 hex)
## Booting image at 0c400000 ...
Image Name:   ARM Linux-2.4.18
```

This will check if the image at (flash?) address "addr1" is ok and boot it; if the image is not ok, the alternative image at address "addr2" will be checked and booted if it is found to be OK. If both images are missing or corrupted, a new image will be loaded over [TFTP](#) and booted.

14.2.17.4. General rules

1. If a command line (or an environment variable executed by a `run` command) contains several commands separated by semicolons, and one of these commands fails, the remaining commands will *still* be executed.
2. If you execute several variables with one call to `run` (i. e. calling `run` with a list of variables as arguments), any failing command will cause `run` to terminate, i. e. the remaining variables are not executed.

14.2.18. How can I load and uncompress a compressed image

Question:

Can I use U-Boot to load and uncompress a compressed image from flash into RAM? And can I choose whether I want to automatically run it at that time, or wait until later?

Answer:

Yes to both questions. First, you should generate your image as type "*standalone*" (using "`mkimage ... -T standalone ...`"). When you use the `bootm` command for such an image, U-Boot will automatically uncompress the code while it is storing it at that image's *load address* in RAM (given by the `-a` option to the `mkimage` command).

As to the second question, by default, unless you say differently, U-Boot will automatically start the image by jumping to its entry point (given by the `-e` option to `mkimage`) after loading it. If you want to prevent automatic execution, just set the environment variable "autostart" to "no" ("`setenv autostart no`") before running `bootm`.

14.2.19. How can I create an uImage from a ELF file

Question:

I would like to run a standard distribution kernel on my target, but I can find only ELF kernel images or even RPM files. How can I use these?

Answer:

If you have just the kernel ELF file, this may be difficult, as you will usually also need a bunch of kernel modules that the distribution of your choice probably bundles with this kernel file. Try to locate and install these first.

If you have a kernel RPM, this usually includes both the kernel ELF file and the required modules. Install these in the [ELDK](#) root file system so you can use this for example mounted over NFS. The following example uses a Fedora kernel RPM on a 4xxFP target:

```
$ cd /tmp/
$ wget http://download.fedora.redhat.com/pub/fedora/linux/updates/11/ppc/kernel-2.6.30.9-90.fc11.ppc.rpm
```

After downloading the RPM we install it (manually using "`rpm2cpio`" and "`cpio`" in the root of the [ELDK](#) file system, "`/opt/eldk/ppc_4xxFP/`") :

```
$ cd /opt/eldk/ppc_4xxFP/
$ rpm2cpio /tmp/kernel-2.6.30.9-90.fc11.ppc.rpm | sudo cpio -vidum
```

This installs a lot of kernel modules in "`./lib/modules/`" and a kernel ELF file in "`./boot`" :

```
$ ls -l boot
total 8792
-rw-r--r-- 1 root root 1226119 Oct 17 17:31 System.map-2.6.30.9-90.fc11.ppc
-rw-r--r-- 1 root root 96224 Oct 17 17:31 config-2.6.30.9-90.fc11.ppc
-rwxr-xr-x 1 root root 7673768 Oct 17 18:20 vmlinuz-2.6.30.9-90.fc11.ppc
```

Now convert the ELF kernel image into an uImage file:

```
$ ppc_4xxFP-objcopy -O binary boot/vmlinuz-2.6.30.9-90.fc11.ppc /tmp/vmlinux.bin
$ gzip -v9 /tmp/vmlinux.bin
/tmp/vmlinux.bin: 58.1% -- replaced with /tmp/vmlinux.bin.gz
$ mkimage -A ppc -O linux -T kernel -C gzip \
> -a 0x00000000 -e 0x00000000 \
> -n Linux-2.6.30.9-90.fc11.ppc \
> -d /tmp/vmlinux.bin.gz /tftpboot/uImage-2.6.30.9-90.fc11.ppc
Image Name:   Linux-2.6.30.9-90.fc11.ppc
Created:      Sun Nov 1 17:00:37 2009
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    3187431 Bytes = 3112.73 kB = 3.04 MB
```

```
Load Address: 0x00000000
Entry Point: 0x00000000
```

There you go.

☐ Note: you still need the Device Tree Blob for your specific target board. This usually does not come with any of the standard distributions. Also, you may find that you need a ramdisk image to get some modules loaded that might be needed to mount your root file system.

14.2.20. My standalone program does not work

Question:

I tried adding some new code to the `hello_world.c` demo program. This works well as soon as I only add code to the existing `hello_world()` function, but as soon as I add some functions of my own, things go all haywire: the code of the `hello_world()` function does not get executed correctly, and my new function gets called with unexpected arguments. What's wrong?

Answer:

You probably failed to notice that any code you add to the example program may shift the entry point address. You should check this using the `nm` program:

```
$ ${CROSS_COMPILE}nm -n examples/hello_world
0000000000040004 T testfunc
0000000000040058 T hello_world
000000000004016c t dummy
...
```

As you can see, the entry point (function `hello_world()`) is no longer at `0x40004` as it was before and as it's documented. Instead, it is now at `0x40058`. So you have to start your standalone program at this address, and everything should work well.

14.2.21. Linux hangs after uncompressing the kernel

Question:

I am using U-Boot with a Linux kernel version $v < 2.4.5\text{-pre5}$, but the last message I see is

```
Uncompressing Kernel Image ... OK
```

Then the system hangs.

Answer:

Most probably you pass bad parameters to the Linux kernel.

There are several possible reasons:

- Bad device tree; for example, check that the memory map set up by the boot loader (like mapping of IMMR, PCI addresses etc.) is consistent with what is encoded in your device tree specification.

Here some possible reasons for older Linux kernel versions:

Linux:

arch/ppc:

- arch/ppc: Bad definition of the `bd_info` structure
You must make sure that your machine specific header file (for instance `include/asm-ppc/tqm8xx.h`) includes the same definition of the Board Information structure as we define in `include/ppcboot.h`, and make sure that your definition of `IMAP_ADDR` uses the same value as your U-Boot configuration in `CFG_IMMR`.
- Bad clock information
Before kernel version 2.4.5-pre5 (BitKeeper Patch 1.1.1.6, 22MAY2001) the kernel expected the clock information in MHz, but recent kernels expect it in Hz instead. U-Boot passes the clock information in Hz by default. To switch to the old behaviour, you can set the environment variable `"clocks_in_mhz"` in U-Boot:

```
=> setenv clocks_in_mhz 1
=> saveenv
```

For recent kernel the `"clocks_in_mhz"` variable **must not be set**. If it is present in your environment, you can delete it as follows:

```
=> setenv clocks_in_mhz
=> saveenv
```

☐ A common error is to try `"setenv clocks_in_mhz 0"` or to some other value - this will not work, as **the value of the variable is not important at all**. It is the **existence** of the variable that will be checked.

- - Inconsistent memory map
Some boards may need correct mappings for some special hardware devices like BCSR (Board Control and Status Registers) etc. Verify that the mappings expected by Linux match those created by U-Boot.

14.2.22. How can I implement automatic software updates?

Question:

How I can use U-Boot to make it easy for my end users to upgrade the firmware of the device?

Answer:

This depends mainly on how you intend to distribute your software updates, and which physical interfaces are present (or usable for this purpose) on your device. Typically you will either distribute the software over the network, or you can use some type of storage device like USB mass storage device (memory stick etc.), SD cards etc.

U-Boot already supports such a feature on several boards. You will probably have to adapt the code and/or the actual behaviour to your specific hardware and/or requirements. Please see here for a starting point:

Network:

When auto-update support over [TFTP](#) is enabled, U-Boot will test in the initialization sequence if a specific image file is present on the [TFTP](#) server. If this is the case, the image will be downloaded and, if it is considered ok, installed into flash. For details, please read [doc/README.update](#) and/or see commits [4bae9090](#) and [e83cc063](#).

USB:

Several boards implement this feature, all in a slightly different way; see [board/trab/auto_update.c](#), [board/mcc200/auto_update.c](#) and [board/esd/common/auto_update.c](#).

With this feature enabled, U-Boot will check during the init sequence if a USB mass storage device is plugged in, if this contains a readable file system, and check if this contains one or more known image files. Additionally it is possible to check if the image versions on the USB device are more recent than those already stored in flash. If all programmed criteria match, and if the images can be read without error, the content of the on-board storage (flash, NAND, etc.) gets automatically updated. Adaption for other storage devices (like SD card etc.) should be trivial to implement.

14.3. Linux

14.3.1. Linux crashes randomly

Question:

On my board, Linux crashes randomly or has random exceptions (especially floating point exceptions if it is a [Power Architecture®](#) processor). Why?

Answer:

Quite likely your [SDRAM](#) initialization is bad. See [UBootCrashAfterRelocation](#) for more information.

On a [Power Architecture®](#), the instructions beginning with 0xFF are floating point instructions. When your memory subsystem fails, the [Power Architecture®](#) is reading bad values (0xFF) and thus executing illegal floating point instructions.

14.3.2. Linux crashes when uncompressing the kernel

Question:

When I try to boot Linux, it crashes during uncompressing the kernel image:

```
=> bootm 100000
## Booting image at 00100000 ...
Image Name: Linux-2.4.25
Image Type: PowerPC Linux Kernel Image (gzip compressed)
Data Size: 1003065 Bytes = 979.6 kB
Load Address: 00000000
Entry Point: 00000000
Verifying Checksum ... OK
Uncompressing Kernel Image ... Error: inflate() returned -3
GUNZIP ERROR - must RESET board to recover
```

Answer:

Your kernel image is quite big - nearly 1 MB compressed; when it gets uncompressed it will need 2.5 ... 3 MB, starting at address 0x0000. But your compressed image was stored at 1 MB (0x100000), so the uncompressed code will overwrite the (remaining) compressed image. The solution is thus simple: just use a higher address to download the compressed image into RAM. For example, try:

```
=> bootm 400000
```

14.3.3. Linux Post Mortem Analysis

You may find yourself in a situation where the Linux kernel crashes or hangs without any output on the console. The first attempt to get more information in such a situation is a **Post Mortem** dump of the log buffer - often the Linux kernel has already collected useful information in its console I/O buffer which just does not get printed because the kernel does not run until successful initialization of the console port.

Proceed as follows:

1. Find out the virtual address of the log buffer; For 2.4 Linux kernels search for "log_buf":
2.4 Linux:

```
bash$ grep log_buf System.map
c0182f54 b log_buf
```

Here the virtual address of the buffer is 0xc0182f54
For 2.6 kernels "*__log_buf*" must be used:

```
bash$ grep __log_buf System.map
c02124c4 b __log_buf
```

Here the virtual address of the buffer is 0xc02124c4

2. Convert to physical address: on [Power Architecture®](#) systems, the kernel is usually configured for a virtual address of kernel base (**CONFIG_KERNEL_START**) of 0xc0000000. Just subtract this value from the address you found. In our case we get:

```
physical address = 0xc0182f54 - 0xc0000000 = 0x00182f54
```

3. Reset your board - do **not** power-cycle it!
4. Use your boot loader (you're running U-Boot, right?) to print a memory dump of that memory area:

```
=> md 0x00182f54
```

This whole operation is based on the assumption that your boot loader does not overwrite the RAM contents - U-Boot will take care not to destroy such valuable information.

14.3.4. Linux kernel register usage

For the [Power Architecture®](#) architecture, the Linux kernel uses the following registers:

R1:
stack pointer
R2:
pointer to *task_struct* for the current task
R3-R4:
parameter passing and return values
R5-R10:
parameter passing
R13:
small data area pointer
R30:
GOT pointer
R31:
frame pointer

A function can use *r0* and *r3 - r12* without saving and restoring them. *r13 - r31* have to be preserved so they must be saved and restored when you want to use them. Also, *cr2 - cr4* must be preserved, while *cr0*, *cr1*, *cr5 - cr7*, *lr*, *ctr* and *xer* can be used without saving & restoring them. [Posted Tue, 15 Jul 2003 by Paul Mackerras to linuxppc-embedded@lists.linuxppc.org].

See also the (E)ABI specifications for the [Power Architecture®](#) architecture, [Developing PowerPC Embedded Application Binary Interface \(EABI\) Compliant Programs](#)

14.3.5. Linux Kernel Ignores my bootargs

Question:

Why doesn't the kernel use the command-line options I set in the "bootargs" environment variable in U-Boot when I boot my target system?

Answer:

This problem is typical for ARM systems only. The following discussion is ARM-centric:

First, check to ensure that you have configured your U-Boot build so that `CONFIG_CMDLINE_TAG` is enabled. (Other tags like `CONFIG_SETUP_MEMORY_TAGS` or `CONFIG_INITRD_TAG` may be needed, too.) This ensures that u-boot will boot the kernel with a command-line tag that incorporates the kernel options you set in the "bootargs" environment variable.

If you have the `CONFIG_CMDLINE_TAG` option configured, the problem is almost certainly with your kernel build. You have to instruct the kernel to pick up the boot tags at a certain address. This is done in the machine descriptor macros, which are found in the processor start-up C code for your architecture. For the Intel DBPXA250 "Lubbock" development board, the machine descriptor macros are located at the bottom of the file `arch/arm/mach-pxa/lubbock.c`, and they look like this:

```
MACHINE_START(LUBBOCK, "Intel DBPXA250 Development Platform")
    MAINTAINER("MontaVista Software Inc.")
    BOOT_MEM(0xa0000000, 0x40000000, io_p2v(0x40000000))
    FIXUP(fixup_lubbock)
    MAPIO(lubbock_map_io)
    INITIRQ(lubbock_init_irq)
MACHINE_END
```

The machine descriptor macros for your machine will be located in a similar file in your kernel source tree. Having located your machine descriptor macros, the next step is to find out where U-Boot puts the kernel boot tags in memory for your architecture. On the Lubbock, this address turns out to be the start of physical RAM plus 0x100, or 0xa0000100. Add the "BOOT_PARAMS" macro with this address to your machine descriptor macros; the result should look something like this:

```
MACHINE_START(LUBBOCK, "Intel DBPXA250 Development Platform")
    MAINTAINER("MontaVista Software Inc.")
    BOOT_PARAMS(0xa0000100)
    BOOT_MEM(0xa0000000, 0x40000000, io_p2v(0x40000000))
    FIXUP(fixup_lubbock)
    MAPIO(lubbock_map_io)
    INITIRQ(lubbock_init_irq)
MACHINE_END
```

If there is already a `BOOT_PARAMS` macro in your machine descriptor macros, modify it so that it has the correct address. Then, rebuild your kernel and re-install it on your target. Now the kernel should be able to pick up the kernel options you have set in the "bootargs" environment variable.

14.3.6. Cannot configure Root Filesystem over NFS

Question:

I want to configure my system with root filesystem over NFS, but I cannot find any such configuration option.

Answer:

What you are looking for is the `CONFIG_ROOT_NFS` configuration option, which depends on `CONFIG_IP_PNP`.

To enable root filesystem over NFS you must enable the "IP: kernel level autoconfiguration" option in the "Networking options" menu first.

14.3.7. Linux Kernel Panics because "init" process dies

Question:

I once had a running system but suddenly, without any changes, the Linux kernel started crashing because the "init" process was dying each time I tried to boot the system, for example like that:

```
...
VFS: Mounted root (nfs filesystem).
Freeing unused kernel memory: 140k init
init has generated signal 11 but has no handler for it
Kernel panic - not syncing: Attempted to kill init!
```

Answer:

You probably run your system with the root file system mounted over NFS. Change into the root directory of your target file system, and remove the file "etc/ld.so.cache". That should fix this problem:

```
# cd /opt/eldk/ppc_6xx/
# rm -f etc/ld.so.cache
```

Explanation:

Normally, the file "etc/ld.so.cache" contains a compiled list of system libraries. This file is used by the dynamic linker/loader `ld.so` to cache library information. If it does not exist, rebuilt automatically. For some reason, a corrupted or partial file was written to your root file system. This corrupt file then confused the dynamic linker so that it crashed when trying to start the `init` process.

Question:

I cannot boot into my freshly installed [ELDK](#) Root-NFS because `init` dies with an unhandled signal like this:

```
Freeing unused kernel memory: 124k init
PHY: e0103320:00 - Link is Up - 100/Full
init has generated signal 4 but has no handler for it
Kernel panic - not syncing: Attempted to kill init!
Rebooting in 1 seconds..
```

Answer:

Your [CPU](#) does not have a floating point unit, your kernel has no math emulation (`CONFIG_MATH_EMULATION`) enabled but you still try to boot into a rootfilesystem intended for FPU systems.

This is to be expected for example if you try to use a `ppc_6xx` rootfilesystem on an 8xx system.

14.3.8. Unable to open an initial console

Question:

The Linux kernel boots, but then hangs after printing: "Warning: unable to open an initial console".

Answer:

Most probably you have one or missing entries in the `/dev` directory in your root filesystem. If you are using the [ELDK](#)'s root filesystem over NFS, you probably forgot to run the `ELDK_MAKEDEV` and `ELDK_FIXOWNER` scripts as described in [3.7. Mounting Target Components via NFS](#).

14.3.9. System hangs when entering User Space (ARM)

Question:

The Linux kernel boots, but then the system hangs after printing: "Freeing init memory: 120K". I'm using the root file system from [ELDK](#) 4.2 (or later) on an ARM system.

Answer:

[ELDK](#) 4.2 (and later) for ARM provide an [EABI](#) compliant User Space environment. You must enable [EABI](#) support in your Linux kernel configuration, or the system will hang as described. Your kernel's "config" file should contain "CONFIG_AEABI=y" for this.

14.3.10. Mounting a Filesystem over NFS hangs forever

Question:

We use the `SELF` ramdisk image that comes with the [ELDK](#). When we try to mount a filesystem over NFS from the server, for example:

```
# mount -t nfs 192.168.1.1:/target/home /home
```

the command waits nearly 5 minutes in uninterruptable sleep. Then the mount finally succeeds. What's wrong?

Answer:

The default configuration of the [SELF](#) was not designed to mount additional filesystems with file locking over NFS, so no portmap daemon is running, which is causing your problems. There are two solutions for the problem:

1. Add the portmap daemon (`/sbin/portmap`) to the target filesystem and start it as part of the init scripts.
2. Tell the "mount" program and the kernel that you don't need file locking by passing the "nolock" option to the mount call, i. e. use

```
# mount -o nolock -t nfs 192.168.1.1:/target/home /home
```

Explanation:

If you call the mount command like above (i. e. without the "nolock" option) an RPC call to the "portmap" daemon will be attempted which is required to start a *lockd* kernel thread which is necessary if you want to use file locking on the NFS filesystem. This call will fail only after a **very** long timeout.

14.3.11. Ethernet does not work in Linux

Question:

Ethernet does not work on my board. But everything is fine when I use the ethernet interface in U-Boot (for example by performing a [TFTP](#) download). This is a bug in U-Boot, right?

Answer:

No. It's a bug in the Linux ethernet driver.

In some cases the Linux driver fails to set the [MAC](#) address. That's a buggy driver then - Linux ethernet drivers are supposed to read the [MAC](#) address at startup. On ->open, they are supposed to reprogram the [MAC](#) address back into the chip (but not the EEPROM, if any) whether or not the address has been changed.

In general, a Linux driver shall not make any assumptions about any initialization being done (or not done) by a boot loader; instead, that driver is responsible for performing all of the necessary initialization itself.

And U-Boot shall not touch any hardware it does not access itself. If you don't use the ethernet interface in U-Boot, it won't be initialized by U-Boot.

A pretty extensive discussion of this issue can be found in the thread [ATAG for MAC address](#) on the ARM Linux mailing list. [archive 1](#) [archive 2](#)

Some current methods for handling the [MAC](#) address programming:

- use custom ATAGs (ARM systems)
- use a [Flattened Device Tree](#) (if your arch/port supports it)
- parse the U-Boot environment directly
- pass it via the command line

If your device driver does not support one of these sources directly, then do it yourself:

- add an init board hook
- program it from user space ('ifconfig hw ...')
- for people who need to do NFS root or similar, then use initramfs -- this is what it was designed for !

14.3.12. Loopback interface does not work

Question:

When I boot Linux I get a "socket: Address family not supported by protocol" error message when I try to configure the loopback interface. What's wrong?

Answer:

This is most probably a problem with your kernel configuration. Make sure that the `CONFIG_PACKET` option is selected.

14.3.13. Linux kernel messages are not printed on the console

Question:

I expect to see some Linux kernel messages on the console, but there aren't any.

Answer:

This is absolutely normal when using the [ELDK](#) with root filesystem over NFS. The [ELDK](#) startup routines will start the syslog daemon, which will collect all kernel messages and write them into a logfile (`/var/log/messages`).

If you want to see the messages at the console, either run `tail -f /var/log/messages &` on the console window, or stop the syslog daemon by issuing a `/etc/rc.d/init.d/syslog stop` command. Another alternative is to increase the `console_loglevel` of the kernel (any message with log level less than `console_loglevel` will be printed to the console). With the following command the `console_loglevel` could be set at runtime: `echo 8 > /proc/sys/kernel/printk`. Now all messages are displayed on the console.

14.3.14. Linux ignores input when using the framebuffer driver

Question:

When using the framebuffer driver the console output goes to the LCD display, but I cannot input anything. What's wrong?

Answer:

You can define "console devices" using the `console=` boot argument. Add something like this to your `bootargs` setting:

```
... console=tty0 console=ttyS0,{baudrate} ...
```

This will ensure that the boot messages are displayed on both the framebuffer (`/dev/tty0`) and the serial console (`/dev/ttyS0`); the last device named in a `console=` option will be the one that takes input, too, so with the settings above you can use the serial console to enter commands etc. For a more detailed description see <http://www.tldp.org/HOWTO/Remote-Serial-Console-HOWTO/configure-kernel.html>

14.3.15. How to switch off the screen saver and the blinking cursor?

Question:

I'm using a splash screen on my frame buffer display, but it is disturbed by a blinking cursor, and after a while the screen is blanked. How can I prevent this?

Answer:

Screen saver and blinking cursor can be turned off (and on) using escape sequences.

To turn off the screen saver, send the sequence "\E[9;0]" to the terminal="/dev/tty1". For example, output the content of file ["/etc/blank_off"](#) in one of your *init* scripts:

```
# cat /etc/blank_off
```

To turn off the blinking cursor, send the sequence "\E[?251\E[?1c" to the terminal. For example, copy the content of file ["/etc/init/tty"](#) to the terminal:

```
# cat /etc/init/tty
```

For details, please see *"man 4 console_codes"*.

14.3.16. BogoMIPS Value too low

Question:

We are only seeing 263.78 bogomips on a MPC5200 running at 396 MHz.
Doesn't this seem way to low ?? With a 603e core I'd expect 1 bogomip per MHz or better.

Answer:

No, the values you see is correct. Please keep in mind that there is a good reason for the name **BogoMIPS**.

On [Power Architecture®](#), the bogomips calculation is measuring the speed of a `dbnz` instruction. On some processors like the MPC8xx it takes 2 clocks per `dbnz` instruction, and you get 1 BogoMIP/MHz. The MPC5200 takes 3 clocks per `dbnz` in this loop, so you get .67 BogoMIP/MHz.

See also [The frequently asked questions about BogoMips](#) (note: this document is somewhat outdated).

Question:

But I have a MPC8572 running at 1.5GHz, amd it shows only 150 bogomips. This cannot be correct?

Answer:

This value is indeed correct.

"With recent kernels, when build with ARCH=powerpc, we now use the hardware timebase instead of bogus processor loops for short timings. Thus our bogomips value is no longer the speed at which the processor runs empty loops, but the actual processor timebase value as obtained after calibration at boot. " - Benjamin Herrenschmidt

14.3.17. Linux Kernel crashes when using a ramdisk image

Question:

I have a [Power Architecture®](#) board with 1 GiB of RAM (or more). It works fine with root file system over NFS, but it will crash when I try to use a ramdisk.

Answer:

Check where your ramdisk image gets loaded to. In the standard configuration, the Linux kernel can access only 768 MiB of RAM, so your ramdisk image must be loaded **below** this limit. Check your boot messages. You are hit by this problem when U-Boot reports something like this:

```
Loading Ramdisk to 3fdab000, end 3ff2ff9d ... OK
```

and then Linux shows a message like this:

```
mem_pieces_remove: [3fdab000,3ff2ff9d) not in any region
```

To fix, just tell U-Boot to load the ramdisk image below the 768 MB limit:

```
=> setenv initrd_high 30000000
```

14.3.18. Ramdisk Greater than 4 MB Causes Problems

Question:

I built a ramdisk image which is bigger than 4 MB. I run into problems when I try to boot Linux with this image, while other (smaller) ramdisk images work fine.

Answer:

The Linux kernel has a default maximum ramdisk size of 4096 kB. To boot with a bigger ramdisk image, you must raise this value. There are two methods:

- Dynamical adjustment using boot arguments:
You can pass a boot argument `ramdisk_size=<size-in-kB>` to the Linux kernel to overwrite the configured maximum. Note that this argument needs to be before any `root` argument. A flexible way to to this is using U-Boot environment variables. For instance, to boot with a ramdisk image of 6 MB (6144 kB), you can define:

```
=> setenv rd_size 6144
=> setenv bootargs ... ramdisk_size=${rd_size} ...
=> saveenv
```

If you later find out that you need an even bigger ramdisk image, or that a smaller one is sufficient, all that needs changing is the value of the `"rd_size"` environment variable.

- - Increasing the Linux kernel default value:
When configuring your Linux kernel, adjust the value of the `CONFIG_BLK_DEV_RAM_SIZE` parameter so that it contains a number equal or larger than your ramdisk (in kB). (In the 2.4 kernel series, you'll find this setting under the "Block devices" menu choice while, in the 2.6 series, it will be under "Device drivers" -> "Block devices".)

14.3.19. Combining a Kernel and a Ramdisk into a Multi-File Image

Question:

I used to build a `zImage.initrd` file which combined the Linux kernel with a ramdisk image. Can I do something similar with U-Boot?

Answer:

Yes, you can create "Multi-File Images" which contain several images, typically an OS (Linux) kernel image and one or more data images like RAMDisks. This construct is useful for instance when you want to boot over the network using [BOOTP](#) etc., where the boot server provides just a single image file, but you want to get for instance an OS kernel and a RAMDisk image.

The typical way to build such an image is:

```
bash$ mkimage -A ppc -O Linux -T multi -C gzip \
-n 'Linux Multiboot-Image' -e 0 -a 0 \
-d vmlinux.gz:ramdisk_image.gz pMulti
```

See also the usage message you get when you call `"mkimage"` without arguments.

14.3.20. Adding Files to Ramdisk is Non Persistent

Question:

I want to add some files to my ramdisk, but every time I reboot I lose all my changes. What can I do?

Answer:

To add your files or modifications permanently, you have to rebuild the ramdisk image. You may check out the sources of our [SELF](#) package (Simple Embedded Linux Framework) to see how this can be done, see for example <ftp://ftp.denx.de/pub/LinuxPPC/usr/src/SELF/> or check out the sources for [ELDK](#) (module *eldk_build* from our [CVS](#) server, see <http://www.denx.de/re/linux.html>).

See also section [14.4.1. How to Add Files to a SELF Ramdisk](#) for another way to change the ramdisk image.

For further hints about the creation and use of initial ramdisk images see also the file *Documentation/initrd.txt* in your Linux kernel source directory.

14.3.21. Kernel Configuration for [PCMCIA](#)

Question:

Which kernel configuration options are relevant to support [PCMCIA](#) cards under Linux?

Answer:

The following kernel configuration options are required to support miscellaneous [PCMCIA](#) card types with Linux and the [PCMCIA](#) CS package:

- [PCMCIA](#) IDE cards (CF and true-IDE)
To support the IDE CardService client, the kernel has to be configured with general ATA IDE support. The MPC8xx IDE support (`CONFIG_BLK_DEV_MPC8XX_IDE` flag) must be turned off.
- [PCMCIA](#) modem cards
The kernel has to be configured with standard serial port support (`CONFIG_SERIAL` flag). After the kernel bootstrap the following preparation is needed:

```
bash# mknod /dev/ttySp0 c 240 64
```

This creates a new special device for the modem card; please note that `/dev/ttyS0 ... S4` and `TTY_MAJOR 4` are already used by the standard 8xx [UART](#) driver). `/dev/ttySp0` becomes available for use as soon as the CardServices detect and initialize the [PCMCIA](#) modem card.

- [PCMCIA](#) Wireless LAN cards
Enable the "Network device support" --> "Wireless LAN (non-hamradio)" --> "Wireless LAN (non-hamradio)" option in the kernel configuration (`CONFIG_NET_RADIO` flag).

14.3.22. Configure Linux for PCMCIA Cards using the Card Services package

The following kernel configuration options are required to support miscellaneous [PCMCIA](#) card types with Linux and the [PCMCIA](#) CS package:

1. [PCMCIA](#) IDE cards ([CompactFlash](#) and true-IDE)
General setup -> Support for hot-pluggable devices (enable: **Y**) -> [PCMCIA](#)/CardBus support -> [PCMCIA](#)/CardBus support (enable: **M**) -> MPC8XX [PCMCIA](#) host bridge support (select)
2. [PCMCIA](#) Modem Cards
3. [PCMCIA](#) Network Cards
4. [PCMCIA](#) WLAN Cards

Build and install modules in target root filesystem, shared over NFS:

```
bash$ make modules modules_install INSTALL_MOD_PATH=/opt/eldk/ppc_8xx
```

Adjust [PCMCIA](#) configuration file (`/opt/eldk/ppc_8xx/etc/sysconfig/pcmcia`):

```
PCMCIA=yes
PCIC=m8xx_pcmcia
PCIC_OPTS=
CORE_OPTS=
CARDMGR_OPTS=
```

Start [PCMCIA](#) Card Services:

```
bash-2.05# sh /etc/rc.d/init.d/pcmcia start
```

14.3.23. Configure Linux for PCMCIA Cards without the Card Services package

For "disk" type PC Cards ([FlashDisks](#), [CompactFlash](#), Hard Disk Adapters - basically anything that looks like an ordinary IDE drive), an alternative solution is available: direct support within the Linux kernel. This has the big advantage of minimal memory footprint, but of course it comes with a couple of disadvantages, too:

- It works only with "disk" type PC Cards - no support for modems, network cards, etc; for these you still need the [PCMCIA](#) Card Services package.
- There is no support for "hot plug", i. e. you cannot insert or remove the card while Linux is running. (Well, of course you can do this, but either you will not be able to access any card inserted, or when you remove a card you will most likely crash the system. Don't do it - you have been warned!)
- The code relies on initialization of the [PCMCIA](#) controller by the firmware (of course U-Boot will do exactly what's required).

On the other hand these are no real restrictions for use in an Embedded System.

To enable the "direct IDE support" you have to select the following Linux kernel configuration options:

```
CONFIG_IDE=y
CONFIG_BLK_DEV_IDE=y
CONFIG_BLK_DEV_IDEDISK=y
CONFIG_IDEDISK_MULTI_MODE=y
CONFIG_BLK_DEV_MPC8XX_IDE=y
CONFIG_BLK_DEV_IDE_MODES=y
```

and, depending on which partition types and languages you want to support:

```
CONFIG_PARTITION_ADVANCED=y
CONFIG_MAC_PARTITION=y
CONFIG_MSDOS_PARTITION=y
CONFIG_NLS=y
CONFIG_NLS_DEFAULT="y"
CONFIG_NLS_ISO8859_1=y
CONFIG_NLS_ISO8859_15=y
```

With these options you will see messages like the following when you boot the Linux kernel:

```
...
Uniform Multi-Platform E-IDE driver Revision: 6.31
ide: Assuming 50MHz system bus speed for PIO modes; override with idebus=xx
PCMCIA slot B: phys mem e0000000...ec000000 (size 0c000000)
Card ID: CF 128MB CH
Fixed Disk Card
IDE interface
[silicon] [unique] [single] [sleep] [standby] [idle] [low power]
hda: probing with STATUS(0x50) instead of ALTSTATUS(0x41)
```



```
hda: CF 128MB, ATA DISK drive
ide0 at 0xc7000320-0xc7000327,0xc3000106 on irq 13
hda: 250368 sectors (128 MB) w/16KiB Cache, CHS=978/8/32
Partition check:
hda: hda1 hda2 hda3 hda4
...
```

You can now access your PC Card "disk" like any normal IDE drive. If you start with a new drive, you have to start by creating a new partition table. For Power Architecture® systems, there are two commonly used options:

14.3.23.1. Using a MacOS Partition Table

A MacOS partition table is the "native" partition table format on Power Architecture® systems; most desktop Power Architecture® systems use it, so you may prefer it when you have Power Architecture® development systems around.

To format your "disk" drive with a MacOS partition table you can use the `pdisk` command:

We start printing the help menu, re-initializing the partition table and then printing the new, empty partition table so that we know the block numbers when we want to create new partitions:

```
# pdisk /dev/hda
Edit /dev/hda -
Command (? for help): ?
Notes:
  Base and length fields are blocks, which vary in size between media.
  The base field can be <nth>p; i.e. use the base of the nth partition.
  The length field can be a length followed by k, m, g or t to indicate
  kilo, mega, giga, or tera bytes; also the length can be <nth>p; i.e. use
  the length of the nth partition.
  The name of a partition is descriptive text.
Commands are:
h      help
p      print the partition table
P      (print ordered by base address)
i      initialize partition map
s      change size of partition map
c      create new partition (standard MkLinux type)
C      (create with type also specified)
n      (re)name a partition
d      delete a partition
r      reorder partition entry in map
w      write the partition table
q      quit editing (don't save changes)
Command (? for help): i
map already exists
do you want to reinit? [n/y]: y
Command (? for help): p
Partition map (with 512 byte blocks) on '/dev/hda'
#:          type name      length  base    ( size )
1: Apple_partition_map Apple    63 @ 1
2:          Apple_Free Extra 1587536 @ 64      (775.2M)
Device block size=512, Number of Blocks=1587600 (775.2M)
DeviceType=0x0, DeviceId=0x0
```

At first we create two small partitions that will be used to store a Linux boot image; a compressed Linux kernel is typically around 400 ... 500 kB, so choosing a partition size of 2 MB is more than generous. 2 MB corresponds to 4096 disk blocks of 512 bytes each, so we enter:

```
Command (? for help): C
First block: 64
Length in blocks: 4096
Name of partition: boot0
Type of partition: PPCBoot
Command (? for help): p
Partition map (with 512 byte blocks) on '/dev/hda'
#:          type name      length  base    ( size )
1: Apple_partition_map Apple    63 @ 1
2:          PPCBoot boot0    4096 @ 64      ( 2.0M)
3:          Apple_Free Extra 1583440 @ 4160   (773.2M)
Device block size=512, Number of Blocks=1587600 (775.2M)
DeviceType=0x0, DeviceId=0x0
```

To be able to select between two kernel images (for instance when we want to do a field upgrade of the Linux kernel) we create a second boot partition of exactly the same size:

```
Command (? for help): C
First block: 4160
Length in blocks: 4096
Name of partition: boot1
Type of partition: PPCBoot
Command (? for help): p
Partition map (with 512 byte blocks) on '/dev/hda'
#:          type name      length  base    ( size )
1: Apple_partition_map Apple    63 @ 1
2:          PPCBoot boot0    4096 @ 64      ( 2.0M)
3:          PPCBoot boot1    4096 @ 4160   ( 2.0M)
4:          Apple_Free Extra 1579344 @ 8256   (771.2M)
Device block size=512, Number of Blocks=1587600 (775.2M)
DeviceType=0x0, DeviceId=0x0
```

Now we create a swap partition - 64 MB should be more than sufficient for our Embedded System; 64 MB means $64 \cdot 1024 \cdot 2 = 131072$ disk blocks of 512 bytes:

```
Command (? for help): C
First block: 8256
Length in blocks: 131072
Name of partition: swap
Type of partition: swap
Command (? for help): p
Partition map (with 512 byte blocks) on '/dev/hda'
#:          type name      length  base    ( size )
1: Apple_partition_map Apple    63 @ 1
2:          PPCBoot boot0    4096 @ 64      ( 2.0M)
3:          PPCBoot boot1    4096 @ 4160   ( 2.0M)
4:          swap swap      131072 @ 8256   ( 64.0M)
5:          Apple_Free Extra 1448272 @ 139328 (707.2M)
Device block size=512, Number of Blocks=1587600 (775.2M)
DeviceType=0x0, DeviceId=0x0
```

Finally, we dedicate all the remaining space to the root partition:

```
Command (? for help): C
First block: 139328
Length in blocks: 1448272
Name of partition: root
Type of partition: Linux
```



```

Command (? for help): p
Partition map (with 512 byte blocks) on '/dev/hda'
#:          type name      length  base    ( size )
1:  Apple_partition_map Apple    63 @ 1
2:      PPCBoot boot0      4096 @ 64    ( 2.0M)
3:      PPCBoot boot1      4096 @ 4160   ( 2.0M)
4:      swap swap        131072 @ 8256    ( 64.0M)
5:      Linux root       1448272 @ 139328  (707.2M)
Device block size=512, Number of Blocks=1587600 (775.2M)
DeviceType=0x0, DeviceId=0x0

```

To make our changes permanent we must write the new partition table to the disk, before we quit the pdisk program:

```

Command (? for help): w
Writing the map destroys what was there before. Is that okay? [n/y]: y
hda: [mac] hda1 hda2 hda3 hda4 hda5
hda: [mac] hda1 hda2 hda3 hda4 hda5
Command (? for help): q

```

Now we can initialize the swap space and the filesystem:

```

# mkswap /dev/hda4
Setting up swapspace version 1, size = 67104768 bytes
# mke2fs /dev/hda5
mke2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
90624 inodes, 181034 blocks
9051 blocks (5.00%) reserved for the super user
First data block=0
6 block groups
32768 blocks per group, 32768 fragments per group
15104 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840
Writing inode tables: done
Writing superblocks and filesystem accounting information: done

```

14.3.23.2. Using a MS-DOS Partition Table

The MS-DOS partition table is especially common on PC type computers, which these days means nearly everywhere. You will prefer this format if you want to exchange your "disk" media with any PC type host system.

The fdisk command is used to create MS-DOS type partition tables; to create the same partitioning scheme as above you would use the following commands:

```

# fdisk /dev/hda
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
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.
The number of cylinders for this disk is set to 1575.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)
Command (m for help): m
Command action
a  toggle a bootable flag
b  edit bsd disklabel
c  toggle the dos compatibility flag
d  delete a partition
l  list known partition types
m  print this menu
n  add a new partition
o  create a new empty DOS partition table
p  print the partition table
q  quit without saving changes
s  create a new empty Sun disklabel
t  change a partition's system id
u  change display/entry units
v  verify the partition table
w  write table to disk and exit
x  extra functionality (experts only)

```

```

Command (m for help): n
Command action
e  extended
p  primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1575, default 1):
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-1575, default 1575): +2M
Command (m for help): p
Disk /dev/hda: 16 heads, 63 sectors, 1575 cylinders
Units = cylinders of 1008 * 512 bytes
    Device Boot      Start         End      Blocks   Id  System
/dev/hda1            1             5       2488+    83  Linux

```

```

Command (m for help): n
Command action
e  extended
p  primary partition (1-4)
p
Partition number (1-4): 2
First cylinder (6-1575, default 6):
Using default value 6
Last cylinder or +size or +sizeM or +sizeK (6-1575, default 1575): +2M
Command (m for help): p
Disk /dev/hda: 16 heads, 63 sectors, 1575 cylinders
Units = cylinders of 1008 * 512 bytes
    Device Boot      Start         End      Blocks   Id  System
/dev/hda1            1             5       2488+    83  Linux
/dev/hda2            6            10       2520     83  Linux

```

```

Command (m for help): n
Command action
e  extended
p  primary partition (1-4)
p
Partition number (1-4): 3

```

```

First cylinder (11-1575, default 11):
Using default value 11
Last cylinder or +size or +sizeM or +sizeK (11-1575, default 1575): +64M
Command (m for help): t
Partition number (1-4): 3
Hex code (type L to list codes): 82
Changed system type of partition 3 to 82 (Linux swap)
Command (m for help): p
Disk /dev/hda: 16 heads, 63 sectors, 1575 cylinders
Units = cylinders of 1008 * 512 bytes
   Device Boot      Start         End      Blocks   Id  System
/dev/hda1            1           5       2488+    83   Linux
/dev/hda2            6          10       2520     83   Linux
/dev/hda3           11         141      66024     82   Linux swap

```

Note that we had to use the `t` command to mark this partition as swap space.

```

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 4
First cylinder (142-1575, default 142):
Using default value 142
Last cylinder or +size or +sizeM or +sizeK (142-1575, default 1575):
Using default value 1575
Command (m for help): p
Disk /dev/hda: 16 heads, 63 sectors, 1575 cylinders
Units = cylinders of 1008 * 512 bytes
   Device Boot      Start         End      Blocks   Id  System
/dev/hda1            1           5       2488+    83   Linux
/dev/hda2            6          10       2520     83   Linux
/dev/hda3           11         141      66024     82   Linux swap
/dev/hda4          142        1575     722736     83   Linux

Command (m for help): w
The partition table has been altered!
Calling ioctl() to re-read partition table.
 hda: hda1 hda2 hda3 hda4
 hda: hda1 hda2 hda3 hda4
WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.

```

Now we are ready to initialize the partitions:

```

# mkswap /dev/hda3
Setting up swapspace version 1, size = 67604480 bytes
# mke2fs /dev/hda4
mke2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
90432 inodes, 180684 blocks
9034 blocks (5.00%) reserved for the super user
First data block=0
6 block groups
32768 blocks per group, 32768 fragments per group
15072 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840
Writing inode tables: done
Writing superblocks and filesystem accounting information: done

```

14.3.24. Boot-Time Configuration of MTD Partitions

Instead of defining a static partition map as described in section [Memory Technology Devices](#) you can define the partitions for your flash memory at boot time using command line arguments. To do that you have to enable the `CONFIG_MTD_CMDLINE_PARTS` kernel configuration option. With this option enabled, the kernel will recognize a command line argument `mtdparts` and decode it as follows:

```

mtdparts=<mtddef>[;<mtddef>]
<mtddef>  := <mtid-id>:<partdef>[,<partdef>]
<partdef> := <size>[@offset][<name>][ro]
<mtid-id> := unique id used in mapping driver/device (number of flash bank)
<size>    := standard linux memsize OR "-" to denote all remaining space
<name>    := '(' NAME ')'
```

For example, instead of using a static partition map like this:

```

0x00000000-0x00060000 : "U-Boot"
0x00060000-0x00080000 : "Environment 1"
0x00080000-0x000A0000 : "Environment 2"
0x000A0000-0x000C0000 : "ASIC Images"
0x000C0000-0x001C0000 : "Linux Kernel"
0x001C0000-0x005C0000 : "Ramdisk Image"
0x005C0000-0x01000000 : "User Data"
```

you can pass a command line argument as follows:

```
mtdparts=0:384k(U-Boot),128k(Env1),128k(Env2),128k(ASIC),1M(Linux),4M(Ramdisk),-(User_Data)
```

14.3.25. Use NTP to synchronize system time against RTC

If a system has a real-time clock (RTC) this is often used only to initialize the system time when the system boots. From then, the system time is running independently. The RTC will probably only be used again at shutdown to save the current system time. Such a configuration is used in many workstation configurations. It is useful if time is not really critical, or if the system time is synchronized against some external reference clock like when using the Network Time Protocol (NTP) to access time servers on the network.

But some systems provide a high-accuracy real-time clock (RTC) while the system clocks are not as accurate, and sometimes permanent access to the net is not possible or wanted. In such systems it makes more sense to use the RTC as reference clock (Stratum 1 NTP server - cf. <http://www.ntp.org/>). To enable this mode of operation you must edit the NTP daemon's configuration file `/etc/ntp.conf` in your target's root file system. Replace the lines

```
server 127.127.1.0      # local clock
fudge 127.127.1.0 stratum 10
```

by

```
server 127.127.43.0      # standard Linux RTC
```

Then make sure to start the NTP daemon on your target by adding it to the corresponding init scripts and restart it if it is already running.

☐ The "address" of the RTC (127.127.43.0 in the example above) is **not** an IP address, but actually used as an index into an internal array of supported reference clocks in the NTP daemon code. You may need to check with your **ntpd** implementation if the example above does not work as expected.

14.3.26. Configure Linux for XIP (Execution In Place)

This document describes how to setup and use XIP in the kernel and the cramfs filesystem. (A patch to add XIP support to your kernel can be found at the bottom of this page.)

14.3.26.1. XIP Kernel

To select XIP you must enable the `CONFIG_XIP` option:

```
$ cd <xip-linux-root>
$ make menuconfig
...
MPC8xx CPM Options  --->
[*] Make a XIP (eXecute in Place) kernel
(40100000) Physical XIP kernel address
(c1100000) Virtual XIP kernel address
(64) Image header size e.g. 64 bytes for PPCBoot
```

The physical **and** virtual address of the flash memory used for XIP must be defined statically with the macros `CONFIG_XIP_PHYS_ADDR` and `CONFIG_XIP_VIRT_ADDR`. The virtual address usually points to the end of the kernel virtual address of the system memory. The physical and virtual address must be aligned relative to an 8 MB boundary:

```
CONFIG_XIP_PHYS_ADDR = FLASH-base-address + offset-in-FLASH
CONFIG_XIP_VIRT_ADDR = 0xc0000000 + DRAM-size + offset-in-FLASH
```

The default configuration parameters shown above are for a system with 16MB of DRAM and the XIP kernel image located at the physical address 0x40100000 in flash memory.

Note that the FLASH and [MTD](#) driver must be disabled.

You can then build the "uImage", copy it to `CONFIG_XIP_PHYS_ADDR` in flash memory and boot it from `CONFIG_XIP_PHYS_ADDR` as usual.

14.3.26.2. Cramfs Filesystem

The cramfs filesystem enhancements:

- They allow cramfs optional direct access to a cramfs image in memory (ram, rom, flash). It eliminates the unnecessary step of passing data through an intermediate buffer, as compared to accessing the same image through a memory block device like `mtdblock`.
- They allow optional cramfs linear root support. This eliminates the requirement of having to provide a block device to use a linear cramfs image as the root filesystem.
- They provide optional XIP. It extends `mkcramfs` to store files marked "+" uncompressed and page-aligned. Linux can then `mmap` those files and execute them in-place without copying them entirely to ram first.

Note: the current implementation can only be used together with a XIP kernel, which provides the appropriate XIP memory (FLASH) mapping.

To configure a root file system on linear cramfs with XIP select:

```
$ cd <xip-linux-root>
$ make menuconfig
...
File systems  --->"
...
<*> Compressed ROM file system support
[*] Use linear addressing for cramfs
(40400000) Physical address of linear cramfs
[*] Support XIP on linear cramfs
[*] Root file system on linear cramfs
```

This defines a cramfs filesystem located at the physical address 0x40400000 in FLASH memory.

After building the kernel image "pImage" as usual, you will want to build a filesystem using the `mkcramfs` executable (it's located in `/scripts/cramfs`). If you do not already have a reasonable sized disk directory tree you will need to make one. The ramdisk directory of [SELF](#) (the Simple Embedded Linux Framework from DENX at ftp.denx.de) is a good starting point. Before you build your cramfs image you must mark the binary files to be executed in place later on with the "t" permission:

```
$ mkcramfs -r ramdisk cramfs.img
```

and copy it to the defined place in FLASH memory.

You can then boot the XIP kernel with the cramfs root filesystem using the boot argument:

```
$ setenv bootargs root=/dev/cramfs ...
```

Be aware that cramfs is a read-only filesystem.

14.3.26.3. Hints and Notes

- XIP conserves RAM at the expense of flash. This might be useful if you have a big flash memory and little RAM.
- Flash memory used for XIP must be readable **all** the time e.g. this excludes installation and usage the character device or [MTD](#) flash drivers, because they do device probing, sector erase etc.
- The XIP extension is currently only available for PowerQUICC™ 8xx but can easily be extended to other architectures.
- Currently only up to 8 MB of ROM/Flash are supported.
- The original work was done for the amanda system.
- Special thanks goes to David Petersen for collecting the available XIP extension sources and highlighting how to put all the pieces together.

14.3.26.4. Space requirements and RAM saving, an example

For ppc 8xx, all figures are in bytes:

- Normal kernel + linear cramfs (patched):

```
pImage: 538062
cramfs: 1081344

          total:      used:      free:  shared: buffers:  cached:
Mem: 14921728 3866624 11055104 2781184      0 2240512
```

- XIP kernel + linear cramfs:

```
pImage: 1395952
cramfs: 1081344

          total:      used:      free:  shared: buffers:  cached:
Mem:  16175104  3940352 12234752  2822144         0  2240512
```

- XIP kernel + XIP cramfs (chmod +t: busybox, initd, libc):

```
pImage: 1395952
cramfs: 1871872

          total:      used:      free:  shared: buffers:  cached:
Mem:  16175104  2367488 13807616  610304         0  671744
```

The actual RAM saving is here approximately $1.1\text{MB} + 1.5\text{M} = 2.6\text{MB}$.

Have fun with XIP.

Wolfgang Grandegger (wg@denx.de)

- linux-2.4.4-2002-03-21-xip.patch.gz: Linux patches for XIP on MPC8xx

14.3.27. Use SCC UART with Hardware Handshake

Question:

I am using a SCC port of a MPC8xx / MPC82xx as UART; for the Linux UART driver I have configured support for hardware handshake. Then I used a null-modem cable to connect the port to the serial port of my PC. But this does not work. What am I doing wrong?

Answer:

There is absolutely no way to connect a MPC8xx / MPC82xx SCC port to any DTE and use RS-232 standard hardware flow control.

Explanation:

The serial interface of the SCC ports in MPC8xx / MPC82xx processors is designed as a DTE circuitry and the RS-232 standard hardware flow control can not be used in the DTE to DTE connection with the null-modem cable (with crossed RTS/CTS signals).

The RS-232 standard specifies a DTE to DCE connection and its hardware handshaking is designed for this specific task. The hardware flow control signals in the PC (and similar equipment) are implemented as software readable/writable bits in a control register and therefore may be arbitrary treated. Unlike that, in the 8xx/82xx the handshake protocol is handled by the CPM microcode. The meaning of the signals is fixed for the RS-232 standard with no way for user to change it.

In widely spread DTE-to-DTE connections over the so called 'null-modem' cable with the hardware flow control lines the meaning of the handshake signals is changed with respect to the RS-232 standard. Therefore this approach may not be used with the 8xx/82xx.

Question:

I succeeded in activating hardware handshake on the transmit side of the SCC using the CTS signal. However I have problems in the receive direction.

Answer:

This is caused by the semantics of the RTS signal as implemented on the SCC controllers: the CPM will assert this signal when it wants to **send** out data. This means you **cannot** use RTS to enable the transmitter on the other side, because it will be enabled only when the SCC is sending data itself.

Conclusions:

If you want to use 8xx/82xx based equipment in combination with RS-232 hardware control protocol, you must have a DCE device (modem, plotter, printer, etc) on the other end.

Hardware flow control on a SCC works only in transmit direction; when receiving data the driver has to be fast enough to prevent data overrun conditions (normally this is no problem though).

14.3.28. How can I access U-Boot environment variables in Linux?

Question:

I would like to access U-Boot's environment variables from my Linux application. Is this possible?

Answer:

Yes, you can. The environment variables must be stored in flash memory, and your Linux kernel must support flash access through the MTD layer. In the U-Boot source tree you can find the environment tools in the directory `tools/env`, which can be built with command:

```
make env
```

For building against older versions of the MTD headers (meaning before v2.6.8-rc1) it is required to pass the argument "MTD_VERSION=old" to make:

```
make MTD_VERSION=old env
```

The resulting binary is called `fw_printenv`, but actually includes support for setting environment variables too. To achieve this, the binary behaves according to the name it is invoked as, so you will have to create a link called `fw_setenv` to `fw_printenv`.

These tools work exactly like the U-Boot commands `printenv` resp. `setenv`. You can either build these tools with a fixed configuration selected at compile time, or you can configure the tools using the `/etc/fw_env.config` configuration file in your target root filesystem. Here is an example configuration file:

```
# Configuration file for fw_(printenv/saveenv) utility.
# Up to two entries are valid, in this case the redundand
# environment sector is assumed present.

#####
# For TQM8xxL modules:
#####
# MTD device name      Device offset  Env. size  Flash sector size
/dev/mtd0              0x8000      0x4000      0x4000
/dev/mtd0              0xC000      0x4000      0x4000

#####
# For NSCU:
#####
# MTD device name      Device offset  Env. size  Flash sector size
/dev/mtd1              0x0000      0x8000      0x20000
/dev/mtd2              0x0000      0x8000      0x20000

#####
# For LWMON
#####
# MTD device name      Device offset  Env. size  Flash sector size
/dev/mtd1              0x0000      0x2000      0x40000
```

14.3.29. The appWeb server hangs OR /dev/random hangs

Question:

I try to run the appWeb server, but it hangs, because read accesses to `/dev/random` hang forever. What's wrong?

Answer:

Your configuration of the Linux kernel does not contain drivers that feed enough entropy for `/dev/random`. Often mouse or keyboard drivers are used for this purpose, so on an embedded system without such devices `/dev/random` may not provide enough random numbers for your application.

Workaround:

As a quick workaround you can use `/dev/urandom` instead; i. e. try the following commands on your system:

```
# cd /dev
# rm -f random
# ln -s urandom random
```

Solution:

The correct solution for the problem is of course to feed sufficient entropy into `/dev/random`. To do so you can modify one or more appropriate device drivers on your system; for example if you know that there is sufficient traffic on network or on a serial port than adding `SA_SAMPLE_RANDOM` to the 3rd argument when calling the `request_irq()` function in your ethernet and/or serial driver(s) will cause the inter-interrupt times to be used to build up entropy for `/dev/random`.

14.3.30. Swapping over NFS

In case that the available memory is not sufficient, i.e. for compiling the X.org server, and no hard-drive can be attached to the system it *is* possible to swap over NFS, although it is not quite straightforward.

Usually one would create a blank file, *mkswap* it and simply do a *swapon swapfile*. Doing this on a filesystem mounted over NFS, i.e. the [ELDK](#) root filesystem, fails however.

With one level of indirection we can trick the kernel into doing it anyway. First we create a filesystem image (ext2 will do) on the NFS filesystem and mount it with the aid of the loopback device. Then we create a blank swapfile *inside* of this filesystem and turn on swapping:

```
bash-2.05b# mount
/dev/nfs on / type nfs (rw)
none on /proc type proc (rw)
bash-2.05b# cd /tmp
bash-2.05b# dd if=/dev/zero of=ext2.img bs=1M count=66
66+0 records in
66+0 records out
bash-2.05b# mkfs.ext2 ext2.img
mke2fs 1.27 (8-Mar-2002)
ext2.img is not a block special device.
Proceed anyway? (y,n) y
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
16920 inodes, 67584 blocks
3379 blocks (5.00%) reserved for the super user
First data block=1
9 block groups
8192 blocks per group, 8192 fragments per group
1880 inodes per group
Superblock backups stored on blocks:
    8193, 24577, 40961, 57345

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

This filesystem will be automatically checked every 26 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
bash-2.05b# for i in `seq 0 9` ; do mknod /dev/loop$i b 7 $i ; done
bash-2.05b# mkdir /mnt2
bash-2.05b# mount -o loop ext2.img /mnt2
bash-2.05b# cd /mnt2
bash-2.05b# dd if=/dev/zero of=swapfile bs=1M count=62
62+0 records in
62+0 records out
bash-2.05b# mkswap swapfile
Setting up swapspace version 1, size = 65007 kB
bash-2.05b# free
              total                used              free              shared            buffers           cached
Mem:           14556                14260                296                  0                772              9116
-/+ buffers/cache:           4372                10184
Swap:              0                  0                  0

bash-2.05b# swapon swapfile
bash-2.05b# free
              total                used              free              shared            buffers           cached
Mem:           14556                14172                384                  0                784              9020
-/+ buffers/cache:           4368                10188
Swap:           63480                  0                63480
bash-2.05b#
```

Because the [ELDK](#) right now has no device nodes for the loopback driver we create them along the way. It goes without saying that the *loop* driver has to be included in the kernel configuration. You can check this by looking for a driver for major number 7 (block devices) in */proc/devices*.

14.3.31. Using NFSv3 for NFS Root Filesystem

Question:

My NFS server only allows the protocol in version 3. Even though my kernel has "NFS client support for NFS version 3" compiled in, I cannot use this as a root filesystem. The boot process stops like this:

```
Looking up port of RPC 100003/2 on 10.0.0.136
Looking up port of RPC 100005/1 on 10.0.0.136
Root-NFS: Server returned error -22 while mounting /opt/eldk/ppc_6xx
VFS: Unable to mount root fs via NFS, trying floppy.
VFS: Cannot open root device "nfs" or unknown-block(2,0)
```

Answer:

In addition to the kernel support, you need to specify the "nfsvers=3" option to use NFS protocol version 3 as a rootfilesystem. So include something like the following in your kernel commandline:

```
nfsroot=[<server-ip>:]<root-dir>,nfsvers=3
```

14.3.32. Using and Configuring the SocketCAN Driver

Question:

When trying to start the SocketCAN interfaces I get error messages like "bit-timing not yet defined", but when trying to configure the bit timing the directory `"/sys/class/net/can0/can_bittiming"` does not exist.

Answer:

The SysFS CAN interface has not been accepted for inclusion into the mainline Linux kernel tree. We had to switch to a Netlink based interface as described in ["Documentation/networking/can.txt"](#)

You need a recent version of the IPROUTE2 utility suite. You can get and build it as follows:

```
$ git clone \
  git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iproute2.git
$ cd iproute2
$ make CC=${CROSS_COMPILE}gcc
```

14.3.33. Telnet / SSH (dropbear) server not working

Question:

The telnet server is running on the target but when I try to login I get this error message:

```
$ telnet 192.168.20.12
telnet 192.168.20.12
Trying 192.168.20.12...
Connected to 192.168.20.12.
Escape character is '^'.
telnetd: All network ports in use.
Connection closed by foreign host.
```

The dropbear ssh server fails in a similar fashion:

```
$ ssh root@192.168.20.12
root@192.168.20.12's password:
PTY allocation request failed on channel 0
shell request failed on channel 0
```

Answer:

Application software on the target cannot open a PTY (pseudo terminal) to handle the incoming request. To understand the problem, we have to be aware that the linux kernel and glibc support two schemes to handle PTYs. The deprecated scheme is hooked to the kernel option `CONFIG_LEGACY_PTYS` and at runtime uses (static) device files `/dev/ptyxx` and `/dev/ttyxx` for the master- and the slave ends of the PTYs. For this to work, you need the kernel support and `/dev/[pt]tyxx` pairs (where `xx` usually is a letter in the range p-z followed by a hexadecimal digit) in the target file system.

The regular support is coupled to the linux kernel option `CONFIG_UNIX98_PTYS` and the `devpts` virtual filesystem which has to be mounted on the target. Together with the device special file `/dev/ptmx` this will dynamically create device files for the allocated PTYs below the mount point. To use it, the device file has to exist and the filesystem needs to be mounted, e.g. like this:

```
# mkdir /dev/pts
# mknod c 5 2 /dev/ptmx
# mount -t devpts devpts /dev/pts
```

14.4. Self

14.4.1. How to Add Files to a [SELF](#) Ramdisk

It is not always necessary to rebuild a [SELF](#) based ramdisk image if you want to modify or to extend it. Especially during development it is often easier to unpack it, modify it, and re-pack it again. To do so, you have to understand the internal structure of the `uRamdisk` (resp. `pRamdisk`) images files as used with the U-Boot (old: PPCBoot) boot loader:

The `uRamdisk` image contains two parts:

- a 64 byte U-Boot header
- a (usually `gzip` compressed) ramdisk image

To modify the contents you have to extract, uncompress and mount the ramdisk image. This can be done as follows:

1. Extract compressed ramdisk image (`ramdisk.gz`)

```
bash$ dd if=uRamdisk bs=64 skip=1 of=ramdisk.gz
21876+1 records in
21876+1 records out
```

2. Uncompress ramdisk image (if it was a compressed one)

```
bash$ gunzip -v ramdisk.gz
ramdisk.gz: 66.6% -- replaced with ramdisk
```

3. Mount ramdisk image

```
bash# mount -o loop ramdisk /mnt/tmp
```

Now you can add, remove, or modify files in the `/mnt/tmp` directory. If you are done, you can re-pack the ramdisk into a U-Boot image:

1. Unmount ramdisk image:

```
bash# umount /mnt/tmp
```

2. Compress ramdisk image

```
bash$ gzip -v9 ramdisk
ramdisk: 66.6% -- replaced with ramdisk.gz
```

3. Create new U-Boot image (new-uRamdisk)

```
bash$ mkimage -T ramdisk -C gzip -n 'Simple Embedded Linux Framework' \
> -d ramdisk.gz new-uRamdisk
Image Name: Simple Embedded Linux Framework
Created: Sun May 4 13:23:48 2003
Image Type: PowerPC Linux RAMDisk Image (gzip compressed)
Data Size: 1400121 Bytes = 1367.31 kB = 1.34 MB
Load Address: 0x00000000
Entry Point: 0x00000000
```

Instead of re-packing into a U-boot ramdisk image you can of course also just extract the contents of the [SELF](#) image and re-use it as base of a (known to be working) root filesystem.

- For example, you can create a JFFS2 filesystem using the `mkfs.jffs2` command that comes with the MTD Tools:

```
bash# mkfs.jffs2 -r /mnt/tmp -e 0x10000 -o image.jffs2
```

- Or you can create a CramFS filesystem with `mkcramfs`:

```
bash# mkcramfs -r /mnt/tmp image.cramfs
Swapping filesystem endian-ness
...
Everything: 1656 kilobytes
Super block: 76 bytes
CRC: 7f34cae4
```

14.4.2. How to Increase the Size of the Ramdisk

1. Extract compressed ramdisk image (ramdisk.gz) from U-Boot image:

```
bash$ dd if=uRamdisk bs=64 skip=1 of=ramdisk.gz
21876+1 records in
21876+1 records out
```

2. Uncompress ramdisk image

```
bash$ gunzip -v ramdisk.gz
ramdisk.gz:      66.6% -- replaced with ramdisk
```

3. Mount ramdisk image
As root:

```
bash# mkdir -p /mnt/tmp
bash# mount -o loop ramdisk /mnt/tmp
```

4. Create new ramdisk image, say 8 MB big:

```
bash$ dd if=/dev/zero of=new_ramdisk bs=1024k count=8
bash$ /sbin/mke2fs -F -m0 new_ramdisk
bash$ /sbin/tune2fs -c 0 -i 0 new_ramdisk
```

As root:

```
bash# mkdir -p /mnt/new
bash# mount -o loop new_ramdisk /mnt/new
```

5. Copy files from old ramdisk to new ramdisk:
As root:

```
bash# cd /mnt/tmp
bash# find . -depth -print | cpio -VBpdm /mnt/new
```

Now you can add, remove, or modify files in the `/mnt/new` directory. If you are done, you can re-pack the ramdisk into a U-Boot image:

6. Unmount ramdisk images:
As root:

```
bash# umount /mnt/tmp
bash# umount /mnt/new
```

7. Compress new ramdisk image

```
bash$ gzip -v9 new_ramdisk
ramdisk:      66.6% -- replaced with new_ramdisk.gz
```

8. Create new U-Boot image (new-uRamdisk)

```
bash$ mkimage -T ramdisk -C gzip -n 'New Simple Embedded Linux Framework' \
> -d new_ramdisk.gz new_uRamdisk
Image Name:   Simple Embedded Linux Framework
Created:      Sun May  4 13:23:48 2003
Image Type:   PowerPC Linux RAMDisk Image (gzip compressed)
Data Size:    1400121 Bytes = 1367.31 kB = 1.34 MB
Load Address: 0x00000000
Entry Point:  0x00000000
```

☐ Remember that Linux by default supports only ramdisks up to a size of 4 MB. For bigger ramdisks, you have to either modify your Linux kernel configuration (parameter `CONFIG_BLK_DEV_RAM_SIZE` in the "Block devices" menu), or pass a `"ramdisk_size="` boot argument to the Linux kernel.

14.5. RTAI

14.5.1. Conflicts with asm clobber list

Question:

When I try to compile my Linux kernel after applying the RTAI patch, I get a strange "asm-specifier for variable `__sc_3' conflicts with asm clobber list" error message. What does that mean?

Answer:

You are using an old version of the Linux kernel / RTAI patch in combination with a more recent version of the cross compiler. Please use a recent kernel tree (and the corresponding RTAI patch), or apply the attached patch to fix this problem.

See: <http://www.denx.de/wiki/pub/DULG/ConflictsWithAsmClobberList/patch>

14.6. BDI2000

14.6.1. Where can I find BDI2000 Configuration Files?

The configuration files provided by Abatron can be found here: <ftp://94.230.212.16/bdigdb/config/>

A collection of configuration files for the [BDI2000](#) [BDM](#)/JTAG debugger by [Abatron](#) can be found at <ftp://ftp.denx.de/pub/BDI2000/>

A list of FAQ (with answers) can be found at <http://www.ultsol.com/faqs.htm>

A list of supported flash chips (and the needed matching entries for the config file) can be found at http://www.abatron.ch/fileadmin/user_upload/products/pdf/flashsupp.pdf

14.6.2. How to Debug Linux Exceptions

Question:

I am trying to single step into a Linux exception handler. This does not seem to work. Setting a breakpoint does not work either.

Answer:

The problem is bit complex on a MPC8xx target. Debug mode entry is like an exception and therefore only safe at locations in the code where an exception does not lead to an unrecoverable state. Another exception can only be accepted if SRR0 and SRR1 are saved. The MSR[RI] should indicate if currently an exception is safe. MSR[RI] is cleared automatically at exception entry.

The MPC8xx hardware breakpoints do only trigger if MSR[RI] is set in order to prevent non-recoverable state.

The problem is that the Linux exception handler does not take all this into account. First priority has speed, therefore neither SRR0 nor SRR1 are saved immediately. Only after EXCEPTION_PROLOG this registers are saved. Also Linux does not handle the MSR[RI] bit.

☐ Hint: Use STEPMODE HWBP when debugging Linux. This allows the TLB Miss Exception handler to update the TLB while you are single stepping.

Conclusion:

You cannot debug Linux exception entry and exit code. Because of speed, DataStoreTLBMiss does not even make use of EXCEPTION_PROLOG, and SRR0/SRR1 are never saved. Therefore you cannot debug DataStoreTLBMiss unless you change it's code (save SRR0/SRR1, set MSR[RI]).

14.6.3. How to single step through "RFI" instruction

Question:

I am trying to debug Linux on an IBM 405GP processor. Linux boots fine and I can step through the code until the "rfi" instruction in head_4xx.S; then I get the following:

```
- TARGET: target has entered debug mode
  Target state      : debug mode
  Debug entry cause : JTAG stop request
  Current PC       : 0x00000700
  Current CR       : 0x28004088
  Current MSR      : 0x00000000
  Current LR       : 0x000007a8
# Step timeout detected
```

Answer:

Your single step problem most likely comes from the fact that GDB accesses some non-existent memory (at least some versions do/did in the past). This exception is stored in some way within the 405 and when you step "rfi" it triggers. This is because some instructions like "rfi" are always stepped using a hardware breakpoint and not with the [JTAG](#) single step feature.

Probably you can step over the "rfi" instruction when using the BDI2000's `telnet` command interface instead of GDB.

Similar problems have also been reported when stepping through "mtmsr" or "mfmsr" during initial boot code. The problem comes also from the fact that GDB accesses non-existent memory (maybe it tries to read a non-existent stack frame).

To debug the Linux kernel, I recommend that you run to a point where the [MMU](#) is on before you connect with GDB.

To debug boot code where the [MMU](#) is off I recommend to use the MMAP feature of the BDI to prevent illegal memory accesses from GDB.

14.6.4. Setting a breakpoint doesn't work

Question:

I am trying to set a breakpoint using the BDI2000 `telnet` interface. However, the code does not stop at the breakpoint.

Answer:

Make sure that the [CPU](#) has been stopped before setting the breakpoint. You can verify this by issuing the "info" command before setting the breakpoint. If the target state is "running" you must use the "halt" command to stop the [CPU](#) before you can successfully set the breakpoint.

14.6.5. Remote 'g' packet reply is too long

Question:

I'm trying to debug U-Boot for a PPC4xx processor, but I get the following error:

```
(gdb) target remote xx.xx.xx.xx:2001
Remote debugging using xx.xx.xx.xx:2001

Remote 'g' packet reply is too long:...
```

I believe this error is caused by GDB being configured to the wrong architecture. So I did the following:

```
$ ppc_4xx-gdb u-boot
...
The target architecture is set automatically (currently powerpc:403)
(gdb) show arch
The target architecture is set automatically (currently powerpc:e500)
(gdb) set arch powerpc:403
The target architecture is assumed to be powerpc:e500
(gdb) show arch
The target architecture is assumed to be powerpc:e500
(gdb) set arch powerpc:common
The target architecture is assumed to be powerpc:e500
(gdb) show arch
The target architecture is assumed to be powerpc:e500
(gdb)
```

As you see, initially GDB says the target architecture is "powerpc:403". But the "show arch" command claims it is the "powerpc:e500". And any commands that try to change it from "powerpc:e500" do not appear to be working.

What's wrong, and why am I not able to change the architecture?

Answer:

Some versions of GDB are hard coded to try and read the architecture from the ELF file and set the arch based on that at startup. When this happens you cannot change the arch again. In this case it looks like it has set it incorrectly from the ELF file as "powerpc:e500".

Workaround:

The following procedure can be used to work around the problem:

- Start GDB without a file argument, i. e. do **not** give the the name of the ELF file (here "u-boot") on the command line
- use "set arch" to set the appropriate architecture

- use the "add-symbol-file" command to load the ELF file (here "u-boot") manually
- double check using "show arch" to make sure the arch hasn't changed; change it back in case it has
- connect to the BDI using the "target remote" command as usual and start debugging

Note:

When you see this problem with the GDB version 6.7-1rh as included with [ELDK](#) release 4.2, you may want to install the update to version 6.7-4rh which can be found here: <ftp://ftp.denx.de/pub/eldk/4.2/ppc-linux-x86/updates/RPMS/gdb-ppc-6.7-4.i386.rpm>

14.7. Motorola LITE5200 Board

14.7.1. LITE5200 Installation Howto

A nice "Application Note: Installing Embedded Linux on the Motorola MPC5200 Lite Evaluation Board" which covers the installation of U-Boot and Linux can be found at:

[http://emsys.denayer.wenk.be/emcam/Linux_on_MPC5200_\(UK\).pdf](http://emsys.denayer.wenk.be/emcam/Linux_on_MPC5200_(UK).pdf)

14.7.2. USB does not work on Lite5200 board

Question:

USB does not work on my Lite5200 board. Also, the green LED behind the USB connector remains always off. Why?

Answer:

This is a hardware problem. The green LED must be on as soon as you power on the Lite5200 board. As a workaround you can short-circuit resistor R164 (bottom side of the board, close to the USB connector). Please note that you will probably lose all warranty and/or may ruin the board. You have been warned.

15. Glossary

[ABI](#)

- Application Binary Interface

The convention for register usage and C linkage commonly used on desktop [Power Architecture](#)® machines. Similar, but not identical to the [EABI](#).

Includes binding specific ppc registers to certain fixed purposes, even though there may be no technical reason to enforce such binding, simplifying the process of linking together two separate sets of object code. e.g the [ABI](#) states that r1 shall be the stack pointer.

[BANK](#)

- also "memory bank"

A bank of memory (flash or RAM) consists of all those memory chips on your system that are controlled by the same chip select signal.

For example, a system might consist of one flash chip with a 8 bit bus interface, which is attached to the CS0 chip select signal, 2 flash chips with a 16 bit bus interface, which are attached to the CS1 chip select signal, and 2 [SDRAM](#) chips with a 16 bit bus interface, which are attached to the CS2 chip select signal.

This setup results in a system with 3 banks of memory:

- 1 bank of flash, 8 bit wide (CS0)
- 1 bank of flash, 32 bit wide (CS1)
- 1 bank of [SDRAM](#), 32 bit wide (CS2)

[BDM](#)

- Background Debug Mode

An on-chip debug interface supported by a special hardware port on some processors. It allows to take full control over the [CPU](#) with minimal external hardware, in many cases eliminating the need for expensive tools like In-Circuit-Emulators.

[BOOTP](#)

- Boot Protocol

A network protocol which can be used to inquire a server about information for the intended system configuration (like IP address, host name, netmask, name server, routing, name of a boot image, address of NFS server, etc).

[CFI](#)

- Common Flash Interface

[CFI](#) is a standard for flash chips that allows to create device independend drivers for such chips.

[CPM](#)

- Communications Processor Module

The magic communications co-processor in Motorola PowerQUICC devices. It contains [SCCs](#) and [SMCs](#), and performs [SDMA](#) and [IDMA](#).

[CPU](#)

- Central Processor Unit

Depending on the context, this may refer to the processor core itself, or the physical processor device (including peripherals like memory controller, Ethernet controller, [UARTs](#), LCD controller, ..., packaging etc.) as a single unit. The latter is today often called "system on chip" ("SoC").

[CramFs](#)

- Compressed ROM File System

Cramfs is designed to be a simple, small, and compressed file system for ROM based embedded systems. [CramFs](#) is read-only, limited to 256MB file systems (with 16MB files), and doesn't support 16/32 bits uid/gid, hard links and timestamps.

CVS

- Concurrent Versions System

[CVS](#) is a version control system; it can be used to record the history of files, so that it is for instance possible to retrieve specific versions of a source tree.

DHCP

- Dynamic Host Configuration Protocol

A network protocol which can be used to inquire a server about information for the intended system configuration (like IP address, host name, netmask, name server, routing, name of a boot image, address of NFS server, etc.). Successor of [BOOTP](#)

DMA

- Direct Memory Access

A form a data transfer directly between memory and a peripheral or between memory and memory, without normal program intervention.

EABI

- Embedded Application Binary Interface

The convention for register usage and C linkage commonly used on embedded Power Architecture® machines, derived from the [ABI](#).

ELDK

- Embedded Linux Development Kit

A package which contains everything you need to get startet with an Embedded Linux project on your hardware:

- cross development tools (like compiler, assembler, linker etc.) that are running on a [Host](#) system while generating code for a [Target](#) system
- native tools and libraries that can be use to build a system running on the target; they can also be exported on a NFS server and used as root filesystem for the target
- source code and binary images for PPCBoot and Linux
- Our [SELF](#) package as example configuration for an embedded system.

FEC

- Fast Ethernet Controller

The 100 Mbps (100Base) Ethernet controller, present on 'T' devices such as the 860T and 855T.

FTP

- File Transfer Protocol

A protocol that can be used to transfer files over a network.

GPL

/ LGPL - GNU General Public License/Lesser General Public License

The full license text can be found at <http://www.gnu.org/copyleft/gpl.html>.

The licenses under which the Linux kernel and much of the utility and library code necessary to build a complete system may be copied, distributed and modified. Each portion of the software is copyright by its respected copyright holder, and you must comply with the terms of the license in order to legally copy (and hence use) it. One significant requirement is that you freely redistribute any modifications you make; if you can't cope with this, embedded Linux isn't for you.

Host

The computer system which is used for software development. For instance it is used to run the tools of the [ELDK](#) to build software packages.

IDMA

- Independent [DMA](#)

A general purpose [DMA](#) engine with relatively limited throughput provided by the microcoded [CPM](#), for use with external peripherals or memory-to-memory transfers.

JFFS

- Journalling Flash File System

[JFFS](#) (developed by Axis Communicartion AB, Sweden) is a log-based filesystem on top of the [MTD](#) layer; it promises to keep your filesystem and data in a consistent state even in cases of sudden power-down or system crashes. That's why it is especially useful for embedded devices where a regular shutdown procedure cannot always be guaranteed.

JFFS2

- Second version of the Journalling Flash File System

Like [JFFS](#) this is a journalling flash filesystem that is based on the [MTD](#) layer; it fixes some design problems of [JFFS](#) and adds transparent compression.

JTAG

- Joint Test Action Group

A standard (see "IEEE Standard 1149.1") that defines how to control the pins of [JTAG](#) compliant devices.

Here: An on-chip debug interface supported by a special hardware port on some processors. It allows to take full control over the [CPU](#) with minimal external hardware, in many cases

eliminating the need for expensive tools like In-Circuit-Emulators.

MII

- Media Independent Interface

The IEEE Ethernet standard control interface used to communicate between the Ethernet controller ([MAC](#)) and the external [PHY](#).

MMU

- Memory Management Unit

[CPU](#) component which maps kernel- and user-space virtual addresses to physical addresses, and is an integral part of Linux kernel operation.

MTD

- Memory Technology Devices

The [MTD](#) functions in Linux support memory devices like flash or Disk-On-Chip in a device-independent way so that the higher software layers (like filesystem code) need no knowledge about the actual hardware properties.

PC

Card

PC Cards are self-contained extension cards especially for laptops and other types of portable computers. In just about the size of a credit card they provide functions like LAN cards (including wireless LAN), modems, ISDN cards, or hard disk drives - often "solid-state" disks based on flash chips.

The PC Card technology has been has been developed and standardized by the Personal Computer Memory Card International Association ([PCMCIA](#)), see <http://www.pcmcia.org/pccard.htm> .

PCMCIA

- Personal Computer Memory Card International Association

[PCMCIA](#) is an abbreviation that can stand for several things: the association which defines the standard, the specification itself, or the devices. The official term for the devices is [PC-Card](#).

PHY

- Physical Interface

The physical layer transceiver which implements the IEEE Ethernet standard interface between the ethernet wires (twisted pair, 50 ohm coax, *etc.*) and the ethernet controller ([MAC](#)). [PHYs](#) are often external transceivers but may be integrated in the [MAC](#) chip or in the [CPU](#).

The [PHY](#) is controlled more or less transparently to software via the [MI](#).

RTOS

- Real-Time Operating System

SCC

- Serial Communications Controller

The high performance module(s) within the [CPM](#) which implement the lowest layer of various serial protocols, such as Asynchronous serial ([UART](#)), 10 Mbps Ethernet, HDLC etc.

SDMA

- Serial [DMA](#)

[DMA](#) used to transfer data to and from the [SCCs](#).

SELF

- Simple Embedded Linux Framework

A simple default configuration for Embedded Linux systems that is suitable as starting point for building your own systems. It is based on [BusyBox](#) to provide an *init* process, shell, and many common tools (from `cat` and `ls` to `vi`), plus some other tools to provide network connectivity, allowing to access the system over the internet using `telnet` and `FTP` services.

SIU

- System Interface Unit

Provides much of the external interfacing logic. It's the other major module on Motorola PowerQUICC devices alongside the [CPU](#) core and [CPM](#).

SMC

- Serial Management Controller

A lower performance version of the [SCCs](#) with more limited functionality, particularly useful for serial debug ports and low throughput serial protocols.

SPI

- Serial Peripheral Interface

A relatively simple synchronous serial interface for connecting low speed external devices using minimal wires.

S-Record

- Motorola S-Record Format

Motorola S-records are an industry-standard format for transmitting binary files to target systems and PROM programmers.

See also: <http://pmon.groupbsd.org/Info/srec.htm>

Target

The computer system which will be used later in you application environment, for instance an Embedded System. In many cases it has a different architecture and much more limited resources than a typical [Host](#) system, so it is often not possible to develop the software directly (native) on this system.

TFTP

- Trivial File Transfer Protocol

A simple network protocol for file transfer; used in combination with [BOOTP](#) or [DHCP](#) to load boot images etc. over the network.

UART

- Universal Asynchronous Receiver Transmitter

Generically, this refers to any device capable of implementing a variety of asynchronous serial protocols, such as RS-232, HDLC and SDLC. In this context, it refers to the operating mode of the [SCCs](#) which provides this functionality.

UPM

- User Programmable Machine

A highly flexible bus interfacing machine unit allowing external peripherals with an extremely wide variety of interfacing requirements to be connected directly to the [CPU](#).

YellowDog

More information about the [YellowDog](#) GNU/Linux distribution for [Power Architecture](#)® systems can be found at <http://www.yellowdoglinux.com>.