The DENX U-Boot and Linux Guide (DULG) for TQM8xxL

Table of contents:

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 TQM8xxL boards.

This document was generated at 01 Mar 2008 - 16:53.

2. Introduction

This document describes how to use the firmware U-Boot and the operating system Linux in Embedded PowerPC, 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 TQM8xxL boards.

2.1. Copyright

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-tqm8xxl.html
- plain ASCII text http://www.denx.de/wiki/publish/DULG/DULG-tqm8xxl.txt

* PostScript European A4 format http://www.denx.de/wiki/publish/DULG/DULG-tqm8xxl.ps
* PDF European A4 format http://www.denx.de/wiki/publish/DULG/DULG-tqm8xxl.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	
Hint	
Notes	Note.
Information requiring special attention	Warning
File Names	*file.extension*
Directory Names	*directory*
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	*word*
Code Example	`ls -l`

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.

Starting from version 4.1, the ELDK is 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 CD-ROM from DENX Computer Systems
- for download on the following server:

FTP
ftp://ftp.denx.de/pub/eldk/

- for download on the following mirrors:

FTP	HTTP
ftp://mirror.switch.ch/mirror/eldk/eldk/	http://mirror.switch.ch/ftp/mirror/eldk/eldk/
ftp://sunsite.utk.edu/pub/linux/eldk/	http://sunsite.utk.edu/ftp/pub/linux/eldk/
ftp://ftp.sunet.se/pub/Linux/distributions/eldk/	http://ftp.sunet.se/pub/Linux/distributions/eldk/

3.2. Supported Host Systems

The ELDK can be installed onto and operate with the following operating systems:

- Fedora Core 4, 5, 6, Fedora 7
- Red Hat Linux 7.3, 8.0, 9

- SuSE Linux 8.x, 9.0, 9.1, 9.2, 9.3
- openSUSE openSUSE 10.2
- Debian 3.0 (Woody), 3.1 (Sarge) and 4.0 (Etch)
- Ubuntu 4.10, 5.04, 6.10
- 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 Core systems, you can use one of `yum`, `apt-get` or `up2date` to keep your systems current.

3.3. Supported Target Architectures

The ELDK includes target components and supports code generation for the following PowerPC types of processors:

- ppc_4xx = AMCC 4xx processors without FPU
- ppc_4xxFP = AMCC 4xx processors with FPU (440EP, 440EPx)
- ppc_6xx = PowerPC processors based on 60x cores
  (This includes support for MPC5xxx, 7xx, 82xx and 83xx processors).
- ppc_74xx = 74xx processors
  (This includes support for MPC86xx processors).
- ppc_8xx = MPC8xx processors
- ppc_85xx = MPC85xx processors

There is also an ELDK for ARM and MIPS systems.

3.4. Installation

3.4.1. Product Packaging

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

For the PowerPC target, the ELDK distribution was split into two independent ISO images: one targeting the 4xx family of processors (AMCC), and another one for the 8xx, 6xx, 74xx and 85xx families (Freescale). This makes the ISO images fit on standard CDROM media.

If you are not bound by the CDROM size limitiation there is still a single image containing all targets.

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 3.4.2. 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 3.8.1. 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 3.8.2. 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.

3.4.2. 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 CD-ROM.

Change to a directory with sufficient free disk space; for the PowerPC version of the ELDK you need about 780 MB, or twice as much (1.6 GB) if you also want to create an ISO image in this directory.

To download the ISO image from the *ppc-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/4.1/ppc-linux-x86/iso/ppc-2007-01-19.i
```

Note: The size of this ISO image is more than 790 MB, so it does not fit on CDROM media. If you don't need support for all PowerPC processors then you can use one of the following alternative images which can be writen to standard CDROM media:

ISO Image	Content
ppc-2007-01-19_amcc.iso	ISO image including support for AMCC 4xx / 4xxFP processors
ppc-2007-01-19_freescale.iso	ISO image including support for the remaining PowerPC processors (5xxx, 6xx, 7xx, 74xx, 8xx, 85xx)

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/4.1
ncftp /pub/Linux/distributions/eldk/4.1 > bin
ncftp /pub/Linux/distributions/eldk/4.1 > get -R ppc-linux-x86/distribution
...
ncftp /pub/Linux/distributions/eldk/4.1 > bye
```

Depending on your combination of host and target architecture, you should download one of the following directories:

- *ppc-linux-x86/iso* resp.
  ppc-linux-x86/distribution for PowerPC targets and x86 Linux hosts,
- *mips-linux-x86/iso* resp.
  mips-linux-x86/distribution for MIPS targets and x86 Linux hosts, or
- *arm-linux-x86/iso* resp.
  arm-linux-x86/distribution for ARM targets and x86 Linux hosts.

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 ppc-linux-x86/distribution/$file
> done
```

Now create an ISO image from the directory tree:

```
bash$ mkisofs \
> -A "ELDK-4.1 -- Target: PowerPC -- Host: x86 Linux" \
> -P "(C) `date "+%Y"` DENX Software Engineering,   www.denx.de" \
> -p "`id -nu`@`hostname` -- `date`" \
> -V ppc-linux-x86 \
> -l -J -R -o eldk-ppc-linux-x86.iso ppc-linux-x86/distribution
```

This will create an ISO image *eldk-ppc-linux-x86.iso* in your local directory that can be burned on CD or DVD (depending on size) 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 3.9.2. Setting Up ELDK Build Environment for instructions on obtaining the build environment needed to re-build the ELDK from scratch.

3.4.3. 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 CD with appropriate "exec" permission. In such cases, consult your distribution documentation or mount the CD 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 ELDT packages are installed in any case.

Refer to section 3.5. 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 3.6. Mounting Target Components via NFS.

Note: Installation of the Glibc- and uClibc-based ELDK versions into one directory is not yet supported.

3.4.4. 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 $CROSS_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.

3.4.5. 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.

3.5. 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=ppc_8xx-
  ```

  The trailing '-' character in the `CROSS_COMPILE` variable value is optional and has no effect on the cross tools behavior.
- 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 *ppc-linux-* for example:

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

* or, equivalently:

```
bash$ /opt/eldk/usr/ppc-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:

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

`CROSS_COMPILE` Value	Predefined Compiler Flag	FPU present or not
ppc_4xx-	-mcpu=403	No
ppc_4xxFP-	-mcpu=405fp	Yes
ppc_6xx-	-mcpu=603	Yes
ppc_74xx-	-mcpu=7400	Yes
ppc_8xx-	-mcpu=860	No
ppc_85xx-	-mcpu=8540	Yes

For compatibility with older versions of the ELDK and with other toolkits the following values for `$CROSS_COMPILE` can be used, too: `ppc_7xx-` and `ppc_82xx-`. These are synonyms for `ppc_6xx`.

3.5.1. 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 `ppc-linux-gcc` compiler binary has been installed, is a part of the `PATH` environment variable. The user types the command as follows:

```
$ ppc_8xx-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 (`ppc_8xx-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.

3.6. 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 8xx-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/ppc_8xx
```

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 acccepts 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.

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 ppc_8xx
```

3.7. Rebuilding ELDK Components

3.7.1. 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.

3.7.2. 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/ppc-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`.

3.7.3. 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/ppc-linux/bin* directory must **NOT** be in `PATH`. This is to make sure that the command `gcc` causes invokation of the host `gcc`, rather than the ELDK cross compiler.

3.8. ELDK Packages

3.8.1. List of ELDT Packages

Package Name	Package Version
crosstool	0.35-9
gdb	6.3.0.0-1.21_3
genext2fs	1.3-8
ldd	0.1-1
make	3.80-7_1
make-doc	3.80-7_1
mkcramfs	0.0.1-1
mkimage	1.2.0-1

mtd_utils	2-2
rpm	4.4.1-21_5
rpm-build	4.4.1-21_5

Note: The `crosstool 0.35` ELDT package provides the following packages: `gcc 4.0.0`, `gcc-c++ 4.0.0`, `cpp 4.0.0` and `binutils 2.16.1`. For more information about the `crosstool` package please refer to http://kegel.com/crosstool.

3.8.2. List of Target Packages

Package Name	Package Version
appWeb	1.2.2-1_6
autoconf	2.59-5_1
bash	3.0-31_2
binutils	2.16.1-2
boa	0.94.14rc19-2
busybox	1.3.0-1
byacc	1.9-29_1
bzip2	1.0.2-16_1
bzip2-devel	1.0.2-16_1
bzip2-libs	1.0.2-16_1
coreutils	5.2.1-48.1_1
cpio	2.6-7_1
cpp	4.0.0-4
cracklib	2.8.2-1
cracklib-dicts	2.8.2-1
crosstool	0.35-9
db4	4.3.27-3_1
db4-devel	4.3.27-3_1
db4-utils	4.3.27-3_1
dhclient	3.0.2-12_2
dhcp	3.0.2-12_2
diffutils	2.8.1-15_2
dosfstools	2.10-3_1
dropbear	0.43-1_2
e2fsprogs	1.38-0.FC4.1_2
e2fsprogs-devel	1.38-0.FC4.1_2

expat	1.95.8-6_1
expat-devel	1.95.8-6_1
file	4.13-4_1
findutils	4.2.20-1_1
flex	2.5.4a-34_1
ftp	0.17-26_1
gawk	3.1.4-5_1
gcc	4.0.0-4
gcc-c++	4.0.0-4
gdb	6.3.0.0-1.21_4
glib	1.2.10-16_1
glib2	2.6.6-1_1
glib2-devel	2.6.6-1_1
glib-devel	1.2.10-16_1
grep	2.5.1-48.2_1
groff	1.18.1.1-5_1
gzip	1.3.5-6_1
httpd	2.0.54-10.2_2
httpd-devel	2.0.54-10.2_2
httpd-manual	2.0.54-10.2_2
initscripts	8.11.1-1_3
iproute	2.6.11-1_1
iputils	20020927-22_2
kernel-headers	2.6.19-1
kernel-source	2.6.19-1
krb5-devel	1.4.1-5_2
krb5-libs	1.4.1-5_2
less	382-7_1
libcap	1.10-22_1
libcap-devel	1.10-22_1
libtermcap	2.0.8-41_1
libtermcap-devel	2.0.8-41_1
libtool	1.5.16.multilib2-2_2
libtool-ltdl	1.5.16.multilib2-2_2

libuser	0.53.7-1_2
libuser-devel	0.53.7-1_2
logrotate	3.7.1-10_2
lrzsz	0.12.20-21_1
m4	1.4.3-1_2
mailcap	2.1.19-1_1
make	3.80-7_1
man	1.5p-4_1
microwindows	0.90-7
microwindows-fonts	0.90-1
mingetty	1.07-5_1
mktemp	1.5-23_1
module-init-tools	3.1-4_1
modutils	2.4.22-8_2
modutils-devel	2.4.22-8_2
mtd_utils	1-4
ncompress	4.2.4-42_1
ncurses	5.4-17_1
ncurses-devel	5.4-17_1
net-snmp	5.2.1.2-1_2
net-snmp-devel	5.2.1.2-1_2
net-snmp-libs	5.2.1.2-1_2
net-snmp-utils	5.2.1.2-1_2
net-tools	1.60-52_2
nfs-utils	1.0.7-12_3
ntp	4.2.0.a.2004061-8_1
openssl	0.9.7f-7.10_1
openssl-devel	0.9.7f-7.10_1
pam	0.79-9.5_2
pam-devel	0.79-9.5_2
passwd	0.69-3_2
patch	2.5.4-24_1
pciutils	2.1.99.test8-10_1
pciutils-devel	2.1.99.test8-10_1

pcmcia-cs	3.2.8-1_1
popt	1.7-3
portmap	4.0-65_2
procps	3.2.5-6.3_2
psmisc	21.5-5_2
rdate	1.4-4_1
readline	5.0-3_1
readline-devel	5.0-3_1
routed	0.17-12_1
rpm	4.4.1-22_4
rpm-build	4.4.1-22_4
rpm-devel	4.4.1-22_4
rpm-libs	4.4.1-22_4
rsh	0.17-29_1
rsh-server	0.17-29_1
sed	4.1.4-1_1
SELF	1.0-11
setup	2.5.44-1.1_1
slang	1.4.9-17_2
slang-devel	1.4.9-17_2
strace	4.5.11-1_3
sysklogd	1.4.1-30_2
SysVinit	2.85-39_1
tar	1.15.1-10_2
tcp_wrappers	7.6-39_2
telnet	0.17-35_1
telnet-server	0.17-35_1
termcap	5.4-7_1
tftp	0.40-6_1
tftp-server	0.40-6_1
u-boot	1.2.0-1
util-linux	2.12p-9.12_3
vim-common	6.3.086-0_1
vim-minimal	6.3.086-0_1

wireless-tools	28-1_1
wu-ftpd	2.6.1-3
xenomai	2.3.0-1
xinetd	2.3.13-6_2
zlib	1.2.2.2-3_1
zlib-devel	1.2.2.2-3_1

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 0.35`** target package provides the following packages: **`glibc 2.3.5`**, **`glibc-common 2.3.5`**, **`glibc-devel 2.3.5`**, **`libstdc++ 4.0.0`** and **`libstdc++-devel 4.0.0`**. For more information about the **`crosstool`** package please refer to http://kegel.com/crosstool

3.9. 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.

3.9.1. ELDK Build Process Overview

The ELDK uses the Fedora Core 4 Linux distribution as source code reference. Any modifications to Fedora Core'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 Core pristine sources (SRPMs) **or**
- ELDK source tarball,
- ELDK patches,
- ELDK spec file.

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

The ELDK patches and spec files are available on the ELDK source CD-ROM and from the DENX =git= repository.

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

```
$ git-clone git://www.denx.de/git/module_name  your_repository_name/
```

The following ELDK repositories are available:

Module Name	Contents
eldk/build.git	Build tools, patches, and spec files
eldk/tarballs.git	Source tarballs
eldk/SRPMS.git	Source Packages (SRPMS)

After cloning the repository, you can use standard =git= commands to check out any specific release of the ELDK; for example, to get the files for ELDK release 4.1, please run the command

```
$ git-checkout ELDK_4_1
```

It must be noted that some of the packages which are included in the ELDK are not included in Fedora Core. Examples of such packages are `appWeb`, `microwindows`, and `wu-ftpd`. For these packages tarballs are provided in the DENX `git` repository. We also provide a copy of the original Fedora SRPMS to make sure these remain available permanently.

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:

`-d <arch>`	target architecture: "ppc", "arm" or "mips", 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*
`-p <build_dir>`	(optional) the name of a directory that will be used to store all the build results; used for out-of-tree building
`-u`	build the uClibc-based ELDK version.

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 3.9.3. 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 *eldt* 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 Core 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 Core 4.

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 CVS (see section 3.9.2. 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/....
```

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 Core 4.

The SRPMS directory may contain the source RPM packages of Fedora Core 4. If some (or all) of the Fedora Core SRPMs needed for the build are missing in the directory, 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 3.9.2. 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
```

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.

3.9.2. 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 3.9.1. 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 Core 4 SRPMS needed to build the ELDK.

Alternatively, you can obtain the ELDK build environment from the DENX `git` repository. Three modules are provided: *eldk/build.git*, *eldk/tarballs.git* and *eldk/SRPMS.git*. The first one contains the files for the *build* subdirectory in the build environment; the second one contains source tarballs of the packages that are included in the ELDK but are not present in Fedora, and the last one contains the original Fedora SRPMS. To create the ELDK build environment from the DENX `git` repository, please use the following commands (the example below assumes that the root directory of the build environment is */opt/eldk*):

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

Any Fedora source RPM packages that should be missing will, if required, be automatically downloaded by the `ELDK_BUILD` script.

3.9.3. 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

Further, the *eldt* and *trg* build steps are devided 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>]
```

`-a <arch>`	target architecture: "ppc", "arm" or "mips", defaults to "ppc".
`-n <build_name>`	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.

`-p <prefix>`	is the name of the directory that contains the build environment. Refer to build overview above for description of the build environment.
`-r <result>`	is the name of the directory where the resulting RPMs and SRPMs created on this step will be placed.
`-w <work>`	is the name of the directory where the build is performed.
`<stepname>`	is the name of the build step that is to be performed. Refer to the list of the build procedure steps above.
`<sub_step_number>`	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 *cpkgs.lst* and *tpkgs.lst* 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.

By default, the invocation of `build.sh` assumes that the Glibc-based ELDK version is being built. For the uClibc-based ELDK build, set the `UCLIBC` environment variable to `1` prior to running `build.sh` :

```
bash$ export UCLIBC=1
```

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 *4xx*, *4xxFP*, *6xx*, *74xx*, *8xx* and *85xx*. For example, the command below rebuilds the target RPM listed in the *tpckgs.lst* file under the number of 47 (see section 3.9.4. Format of the cpkgs.lst and tpkgs.lst Files for description of the *tpckgs.lst* and *cpkgs.lst* files), for the *8xx* and *85xx* CPUs:

```
bash$ TARGET_CPU_FAMILY_LIST="8xx 85xx" \
> /opt/eldk/build.sh -a ppc \
>                    -n 2007-01-19 \
>                    -p /opt/eldk/build/ppc-2007-01-19 \
>                    -r /opt/eldk/build/ppc-2007-01-19/results \
>                    -w /opt/eldk/build/ppc-2007-01-19/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 3.7.2. Rebuilding Target Packages and 3.7.3. Rebuilding ELDT Packages above.

3.9.4. 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 4.0 on non-Fedora hosts. These packages will be built and used during the build process, but will not be put into the ELDK 4.0 distribution ISO images.

3.10. Notes for Solaris 2.x Host Environment

If you use a Solaris 2.x host environment, you need additional freeware packages (mostly GNU tools) to install and especially to build the ELDK packages. The following table lists all required packages that must be installed on the Solaris host system **before** attempting to build and/or install the ELDK. All these files except those marked with (**) (and the RPM and *zlib-1.1.2* packages, which are available at ftp://rpmfind.net/linux/solaris are available for free download at ftp://ftp.sunfreeware.com/pub/freeware/sparc/2.6/

Necessary Freeware Packages:

Package	Version	Instance	File Name
autoconf(**)	2.13	SMCautoc	autoconf-2.13-sol26-sparc-local.gz
automake(**)	1.4	SMCautom	automake-1.4-sol26-sparc-local.gz
bash	2.05	SMCbash	bash-2.05-sol26-sparc-local.gz
binutils	2.11.2	SMCbinut	binutils-2.11.2-sol26-sparc-local.gz
bison	1.28	SMCbison	bison-1.28-sol26-sparc-local.gz
bzip2	1.0.1	SMCbzip2	bzip2-1.0.1-sol26-sparc-local.gz
ddd(*)	3.0	TUBddd	ddd-3.0-sol26-sparc-local.gz
diffutils	2.7	GNUdiffut	diffutils-2.7-sol26-sparc-local.gz
expect(*)	5.25	NTexpect	expect-5.25-sol26-sparc-local.gz
fileutils	4.0	SMCfileu	fileutils-4.0-sol26-sparc-local.gz
flex	2.5.4a	FSFflex	flex-2.5.4a-sol26-sparc-local.gz
gawk	3.1.0	SMCgawk	gawk-3.1.0-sol26-sparc-local.gz
gcc	2.95.3	SMCgcc	gcc-2.95.3-sol26-sparc-local.gz
gettext	0.10.37	SMCgtext	gettext-0.10.37-sol26-sparc-local.gz
gzip	1.3	SMCgzip	gzip-1.3-sol26-sparc-local
libiconv	1.6.1	SMClibi	libiconv-1.6.1-sol26-sparc-local.gz
libtool	1.4	SMClibt	libtool-1.4-sol26-sparc-local.gz
m4	1.4	SMCm4	m4-1.4-sol26-sparc-local.gz
make(**)	3.79.1	SMCmake	make-3.79.1-sol26-sparc-local.gz
ncurses	5.2	SMCncurs	ncurses-5.2-sol26-sparc-local.gz
patch	2.5	FSFpatch	patch-2.5-sol26-sparc-local.gz
perl(**)	5.005_03	SMCperl	perl-5.005_03-sol26-sparc-local.gz
python	1.5.2	SMCpython	python-1.5.2-sol26-sparc-local.gz

rpm	2.5.2	RPM	rpm-2.5.2.pkg
sed	3.02	SMCsed	sed-3.02-sol26-sparc-local.gz
tar	1.13.19	SMCtar	tar-1.13.19-sol26-sparc-local.gz
tcl(*)	8.3.3	SMCtcl	tcl-8.3.3-sol26-sparc-local.gz
texinfo	4.0	SMCtexi	texinfo-4.0-sol26-sparc-local.gz
textutils	2.0	SMCtextu	textutils-2.0-sol26-sparc-local.gz
unzip	5.32	IZunzip	unzip-5.32-sol26-sparc-local.gz
wget	1.7	SMCwget	wget-1.7-sol26-sparc-local.gz
zlib(**)	1.0.4	SMCzlib	zlib-1.0.4-sol26-sparc-local.gz
zlib	1.1.2	-	zlib-1.1.2.tar.gz

The packages marked "(*)" are not absolutely required, but sooner or later you will need them anyway so we recommend to install them.

The packages marked "(**)" are older versions of the ones currently available at ftp://ftp.sunfreeware.com/pub/freeware/sparc/2.6/. You can obtain them from the DENX public FTP server.

The following symbolic links must be created in order to be able to build the ELDK on a Solaris machine:

```
/usr/local/bin/cc  --> /usr/local/bin/gcc
/usr/lib/libiconv.so.2 --> /usr/local/lib/libiconv.so.2
/usr/lib/libncurses.so.5 --> /usr/local/lib/libncurses.so.5
```

Additionally, to be able to build the ELDK on Solaris, you must place newer GNU `gettext` macros to the */usr/local/share/aclocal* directory. This can be accomplished as follows:

- Download the http://www.ibiblio.org/pub/packages/solaris/sparc/GNUgettext.0.10.40.SPARC.32bit.Solaris.8.pkg.tgz package.
- Untar the package to a temporary directory and copy the macros to the */usr/local/share/aclocal* directory:

  ```
  $ cp GNUgettext/root/usr/local/share/aclocal/*.m4 /usr/local/share/aclocal
  ```


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 `ckermit` 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. It's 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 10.0.0.0 netmask 255.0.0.0 {
        option routers          10.0.0.2;
        option subnet-mask      255.0.0.0;

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

        host trgt {     hardware ethernet       00:30:BF:01:02:D0;
                        fixed-address           10.0.0.99;
                        option root-path        "/opt/eldk/ppc_8xx";
                        option host-name        "tqm";
                        next-server             10.0.0.2;
                        filename                "/tftpboot/TQM8xxL/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 `10.0.0.0` which uses the netmask `255.0.0.0`.
- The target has the hostname `tqm` and the IP address `10.0.0.99`.
- The host with the IP address `10.0.0.2` will provide the boot image for the target and provide NFS server function in cases when the target mounts it's 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/TQM8xxL/uImage* as boot image for the target.
- The target can mount the directory */opt/eldk/ppc_8xx* 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/ppc_8xx       10.0.0.0/255.0.0.0(rw,no_root_squash,sync)
```

This line exports the */opt/eldk/ppc_8xx* directory with read and write permissions to all hosts on the `10.0.0.0` 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

Das U-Boot (or just "U-Boot" for short) is Open Source Firmware for Embedded PowerPC, 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/UBoot

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://www.denx.de/cgi-bin/gitweb.cgi

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

For details please see here.

The U-Boot source code can also be retrieved from our CVS repository using *anonymous (pserver) CVS*. Press the "Enter" key when asked for the password for user "anonymous":

```
$ cvs -d:pserver:anonymous@www.denx.de:/cvsroot login

$ cvs -z6 -d:pserver:anonymous@www.denx.de:/cvsroot co -P u-boot
```

Official releases of U-Boot are also available through FTP. Compressed `tar` archives can downloaded from the directory ftp://ftp.denx.de/pub/u-boot/.

Those poor people sitting behind a restrictive firewall may use http tunneling to access the repositories. Here is an example for `cvsgrab`, available from http://cvsgrab.sourceforge.net/, to access the U-Boot repository:

```
cvsgrab -quiet -proxyHost <http_proxy> -proxyPort <proxy_port> -proxyUser <proxy_user> \
        -cvsRoot :pserver:anonymous@www.denx.de:/cvsroot \
        -rootUrl http://www.denx.de/cvsweb/ -packagePath u-boot -packageDir u-boot
```

Of course you have to set `http_proxy` , `proxy_port` and `proxy_user` properly.

5.2. Unpacking the Source Code

If you used CVS 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/u-boot-0.4.5.tar.bz2
$ rm -f u-boot
$ bunzip2 < u-boot-0.4.5.tar.bz2 | tar xf -
$ ln -s u-boot-0.4.5 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 TQM8xxL board:

```
$ make tqm8xxl_config
```

The TQM8xxL boards are available in many configurations (different CPUs, clock frequencies, with or without LCD display, with or without Fast Ethernet interface). Depending on the board configuration chose one of the following `make` targets:

TQM823L_config	TQM823L_66MHz_config	TQM823L_80MHz_config
TQM823L_LCD_config	TQM823L_LCD_66MHz_config	TQM823L_LCD_80MHz_config
TQM850L_config	TQM850L_66MHz_config	TQM850L_80MHz_config

TQM855L_config	TQM855L_66MHz_config	TQM855L_80MHz_config
TQM860L_config	TQM860L_66MHz_config	TQM860L_80MHz_config
TQM862L_config	TQM862L_66MHz_config	TQM862L_80MHz_config
TQM855M_config	TQM855M_66MHz_config	TQM855M_80MHz_config
TQM860M_config	TQM860M_66MHz_config	TQM860M_80MHz_config
TQM862M_config	TQM862M_66MHz_config	TQM862M_80MHz_config
TQM862M_100MHz_config		

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 tqm8xxl_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 tqm8xxl_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 TQM8xxL 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` 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 TQM8xxL boards is included in section 13.1. BDI2000 Configuration file

To install a new U-Boot image on your TQM8xxL board using a BDI2000, proceed as follows:

```
BDI>reset
BDI>- TARGET: processing user reset request
BDI>- TARGET: reseting target passed
BDI>- TARGET: processing target init list ....
BDI>- TARGET: processing target init list passed
BDI>md 0x1FFC0
0001ffc0 : 54514d38 36304c44 44424133 2d503530  TQM860LDDBA3-P50
0001ffd0 : 2e323033 20313032 32363132 32203030  .203 10226122 00
0001ffe0 : 44303933 30303238 38312034 00000000  D093002881 4....
0001fff0 : 00000000 00000000 00000000 00000000  ................
00020000 : ffffffff ffffffff ffffffff ffffffff  ................
\...
BDI>rm der 0x2006000f
BDI>erase 00000000
Erasing flash at 0x00000000
Erasing flash passed
BDI>erase 0x008000
Erasing flash at 0x00008000
Erasing flash passed
BDI>erase 0x00c000
Erasing flash at 0x0000c000
Erasing flash passed
BDI>erase 0x010000
Erasing flash at 0x00010000
Erasing flash passed
BDI>erase 0x020000
Erasing flash at 0x00020000
Erasing flash passed
BDI>prog 0 uboot.bin bin
Programming uboot.bin , please wait ....
Programming flash passed
BDI>rm der 0x2002000f
```

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:

```
=> tftp 100000 /tftpboot/uboot.bin
ARP broadcast 1
TFTP from server 10.0.0.2; our IP address is 10.0.0.100
Filename '/tftpboot/uboot.bin'.
Load address: 0x100000
Loading: ###############################
done
Bytes transferred = 155376 (25ef0 hex)
=> protect off 40000000 4003FFFF
Un-Protected 5 sectors
=> era 40000000 4003FFFF
Erase Flash from 0x40000000 to 0x4003ffff
......... done
Erased 5 sectors
=> cp.b 100000 40000000 ${filesize}
Copy to Flash... done
=> setenv filesize
=> saveenv
Saving Enviroment to Flash...
Un-Protected 1 sectors
Erasing Flash...
.. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
=> reset
```

5.4.4. Installation using Linux

If you have Linux running on your TQM8xxL system and your Linux configuration includes a flash device driver, then you can use this to install a U-Boot image to the appropriate address in flash memory:

```
# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00040000 00020000 "uboot"
mtd1: 000c0000 00020000 "kernel"
mtd2: 00100000 00020000 "user"
mtd3: 00200000 00020000 "initrd"
mtd4: 00200000 00020000 "cramfs"
mtd5: 00200000 00020000 "jffs"
# eraseall /dev/mtd0
Erased 256 Kibyte @ 0 -- 100% complete.
# dd if=/tmp/uboot.bin of=/dev/mtd0 bs=128k conv=sync
1+1 records in
2+0 records out
```

5.4.5. Installation using firmware

Connect to the SMC1 port of the tqm8xxl board using the `cu` program. See the hints for configuring `cu` above. Make sure you can communicate with the MON8xx firmware: reset the board and hit `ENTER` a couple of times until you see the MON8xx prompt (`MON:>`). Then proceed as follows:

5.4.5.1. Read Board ID and MAC Address

The same information is also printed on labels on the module, but often these labels are on the underside of the module so you have to remove it from the carrier board to read the text.

```
MON8xx.105 on TQM860L - (C) TQ-Systems 1998-2000
CPU speed: 50 MHz
MON:>

MON:>read 4001ff80

4001FF80:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
4001FF90:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
4001FFA0:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
4001FFB0:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
4001FFC0:  54 51 4D 38 36 30 4C 43 42 30 41 33 2D 53 52 35  TQM860LCB0A3-SR5
4001FFD0:  30 2E 32 30 32 20 31 30 31 33 34 38 37 33 20 30  0.202 10134873 0
4001FFE0:  30 44 30 39 33 30 30 31 32 33 34 20 34 00 00 00  0D093001234 4...
4001FFF0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
40020000:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
40020010:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
40020020:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
40020030:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
40020040:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
40020050:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
40020060:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
40020070:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................

MON:>
```

In the memory dump you can identify 4 strings of ASCII characters, separated by space characters: "TQM860LCB0A3-SR50.202", "10134873", "00D093001234", and "4". These have the following meaning:

- Module Type and Revision
- Serial Number
- Ethernet Address
- Number of additional Ethernet Addresses reserved for this board

In PPCBoot this is stored in two environment variables:

- Serial Number: `serial#` = TQM860LCB0A3-SR50.202 10134873 4
- Ethernet Address: `ethaddr` = 00D093001234 (==> 00:D0:93:00:12:34)

5.4.5.2. Test Download

This step is to make sure that you can download the U-Boot image to the flash memory. We load the U-Boot image to another (free) position in flash memory.

```
MON:>erase 40100000 4013ffff
* Erasing FLASH from 40100000h to 4013FFFFh
* Please wait

MON:>load 100000 flash
* Ready for s-record download to FLASH ...
~>ppcboot.srec
1 2 3 4 5 6 7 8 9 10 11 12 ...
\...
\... 6619 6620 6621 6622 6623
[file transfer complete]
[connected]
```

```
* Start address 40000000
MON:>
```

5.4.5.3. Verify Download

To make sure that the download and flash programming worked we dump the start of the U-Boot image. You should be able to read the U-Boot header information like that:

```
MON:>read 40100000

40100000:  27 05 19 56 50 50 43 42 6F 6F 74 20 31 2E 30 2E  '..VPPCBoot 1.0.
40100010:  30 2D 70 72 65 32 20 28 4A 75 6E 20 20 33 20 32  0-pre2 (Jun  3 2
40100020:  30 30 31 20 2D 20 32 33 3A 35 38 3A 34 30 29 00  001 - 23:58:40).
40100030:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
\...
MON:>
```

5.4.5.4. Erase MON8xx Firmware

The MON8xx Firmware is write-protected. We un-protect and erase it:

```
MON:>protect 1234
* Protection for sectors containing MON8xx disabled

MON:>erase 40000000 4003ffff
* Erasing FLASH from 40000000h to 4003FFFFh
* Please wait

MON:>
```

5.4.5.5. Load U-Boot

Now we load PPCBoot at it's correct position.

```
MON:>load 0 flash
* Ready for s-record download to FLASH ...
~>ppcboot.srec
1 2 3 4 5 6 7 8 9 10 11 12 ...
\...
\... 6619 6620 6621 6622 6623
[file transfer complete]
[connected]
* Start address 40000000
MON:>
```

5.4.5.6. Verify Download

```
MON:>read 40000000

40000000:  27 05 19 56 50 50 43 42 6F 6F 74 20 31 2E 30 2E  '..VPPCBoot 1.0.
40000010:  30 2D 70 72 65 32 20 28 4A 75 6E 20 20 33 20 32  0-pre2 (Jun  3 2
40000020:  30 30 31 20 2D 20 32 33 3A 35 38 3A 34 30 29 00  001 - 23:58:40).
40000030:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
\...
MON:>
```

In case anything goes wrong: Do NOT reset the board! Do NOT switch off the power! Instead, recover the old TQ monitor which is still running in RAM:

5.4.5.7. Recover Old MON8xx Firmware

```
MON:>erase 40000000 4003ffff
* Erasing FLASH from 40000000h to 4003FFFFh
* Please wait

MON:>copy monitor

Copy monitor

MON:>sethwi TQM860LCB0A3-SR50.202 10134873 00D093001234 4
* Hardware information written to 4001FFC0
MON:>
```

5.4.5.8. Reset Board, and Re-Initialize

```
PPCBoot 1.0.0-pre2 (Jun  3 2001 - 23:58:40)

Initializing...
  CPU:   XPC860xxZPnnD3 at 50 MHz: 4 kB I-Cache 4 kB D-Cache FEC present
  Board: ### No HW ID - assuming TQM8xxL
  DRAM:  16 MB
  FLASH:  4 MB
  PCMCIA:   No Card found
  In:    serial
  Out:   serial
  Err:   serial

Hit any key to stop autoboot:  0
=> setenv serial# TQM860LCB0A3-SR50.202 10134873
=> setenv ethaddr 00:D0:93:00:12:34
=> saveenv
Un-Protected 1 sectors
Erasing Flash...
\.. done
Erased 1 sectors
Saving Environment to Flash...
Protected 1 sectors
=> reset
```

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 TQM8xxL board, you have to connect a terminal to the board's serial console port.

The default configuration of the console port on the TQM8xxL 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.1 (X.18)".

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.]

```
=> help
askenv  - get environment variables from stdin
autoscr - run script from memory
base    - print or set address offset
bdinfo  - print Board Info structure
bootm   - boot application image from memory
bootp   - boot image via network using BootP/TFTP protocol
bootd   - boot default, i.e., run 'bootcmd'
cmp     - memory compare
coninfo - print console devices and informations
cp      - memory copy
crc32   - checksum calculation
date    - get/set/reset date & time
dhcp    - invoke DHCP client to obtain IP/boot params
diskboot- boot from IDE device
echo    - echo args to console
erase   - erase FLASH memory
flinfo  - print FLASH memory information
go      - start application at address 'addr'
help    - print online help
ide     - IDE sub-system
iminfo  - print header information for application image
loadb   - load binary file over serial line (kermit mode)
loads   - load S-Record file over serial line
loop    - infinite loop on address range
md      - memory display
mm      - memory modify (auto-incrementing)
mtest   - simple RAM test
```

```
mw      - memory write (fill)
nm      - memory modify (constant address)
printenv- print environment variables
protect - enable or disable FLASH write protection
rarpboot- boot image via network using RARP/TFTP protocol
reset   - Perform RESET of the CPU
run     - run commands in an environment variable
saveenv - save environment variables to persistent storage
setenv  - set environment variables
sleep   - delay execution for some time
tftpboot- boot image via network using TFTP protocol
               and env variables ipaddr and serverip
version - print monitor version
?       - alias for 'help'
=>
```

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

```
=> help tftpboot
tftpboot [loadAddress] [bootfilename]

=> help setenv printenv
setenv name value ...
    - set environment variable 'name' to 'value ...'
setenv name
    - delete environment variable 'name'

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 information for all FLASH memory banks
flinfo N
    - print information for FLASH memory bank # N

tftpboot [loadAddress] [bootfilename]

=>
```

5.8. The First Power-On

Note: If you bought your TQM8xxL 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.1 (X.18)" on your TQM8xxL board to the designated serial port of your host, start the terminal program, and connect the power supply of your TQM8xxL board. You should see messages like this:

```
Connecting to /dev/ttyS1, 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.
----------------------------------------------------
^@
```

```
PPCBoot 1.1.5 (Mar 21 2002 - 19:55:04)

CPU:   XPC860xxZPnnD3 at 50 MHz: 16 kB I-Cache 8 kB D-Cache FEC present
Board: TQM860LDDBA3-P50.203
DRAM:  64 MB
FLASH:  8 MB
In:    serial
Out:   serial
Err:   serial
PCMCIA:   No Card found

Type "run flash_nfs" to mount root filesystem over NFS

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# TQM860LDB0A3-P.200 10061684 4

=> setenv ethaddr 00:D0:93:00:05:B5
```

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

```
=> printenv serial# ethaddr
serial#=TQM860LDDBA3-P50.203 10226122 4
ethaddr=00:D0:93:00:28:81
=>
```

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 Enviroment to Flash...
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
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 of the configuration of U-Boot and on the definition of some variables in your U-Boot environment.

All U-Boot commands expect numbers to be entered in hexadecimal 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. bdinfo - print Board Info structure

```
=> help bdinfo
bdinfo - No help available.

=>
```

The `bdinfo` 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
memstart    = 0x00000000
memsize     = 0x04000000
flashstart  = 0x40000000
flashsize   = 0x00800000
flashoffset = 0x00030000
sramstart   = 0x00000000
sramsize    = 0x00000000
immr_base   = 0xFFF00000
bootflags   = 0x00000001
intfreq     =     50 MHz
busfreq     =     50 MHz
ethaddr     = 00:D0:93:00:28:81
IP addr     = 10.0.0.99
baudrate    = 115200 bps
=>
```

5.9.1.2. coninfo - print console devices and informations

```
=> help conin
coninfo
=>
```

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

```
=> conin
List of available devices:
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 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: FUJITSU 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

Bank # 2: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40400000      40408000      4040C000      40410000      40420000
    40440000      40460000      40480000      404A0000      404C0000
    404E0000      40500000      40520000      40540000      40560000
    40580000      405A0000      405C0000      405E0000      40600000
    40620000      40640000      40660000      40680000      406A0000
    406C0000      406E0000      40700000      40720000      40740000
    40760000      40780000      407A0000      407C0000      407E0000
=>
```

5.9.1.4. iminfo - print header information for application image

```
=> help iminfo
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.

```
=> imi 100000

## Checking Image at 00100000 ...
   Image Name:   Linux-2.4.4
   Created:      2002-04-07  21:31:59 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    605429 Bytes = 591 kB = 0 MB
   Load Address: 00000000
   Entry Point:  00000000
   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 [command ...]
    - show help information (for 'command')
'help' prints online help for the monitor commands.

Without arguments, it prints a short usage message for all commands.

To get detailed help information for specific commands you can type
'help' with one or more command names as arguments.

=>
```

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 on  start end
    - protect FLASH from addr 'start' to addr 'end'
protect on  N:SF[-SL]
    - protect sectors SF-SL in FLASH bank # N
protect on  bank N
    - protect FLASH bank # N
protect on  all
    - protect all FLASH banks
protect off start end
    - make FLASH from addr 'start' to addr 'end' writable
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 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 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 PowerPC 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 c
00000000: feffffff 00000000 7cbd2b78 7cdc3378    ........|.+x|.3x
00000010: 3cfb3b78 3b000000 7c0002e4 39000000    <.;x;...|...9...
00000020: 7d1043a6 3d000400 7918c3a6 3d00c000    }.C.=...y...=...
=> base 40000000
Base Address: 0x40000000
=> md 0 c
40000000: 27051956 50504342 6f6f7420 312e312e    '..VPPCBoot 1.1.
40000010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
40000020: 2031393a 35353a30 34290000 00000000     19:55:04)......
=>
```

5.9.2.2. crc32 - checksum calculation

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

```
=> crc 100004 3FC
CRC32 for 00100004 ... 001003ff ==> d433b05b
=>
```

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

```
=> crc 100004 3FC 100000
CRC32 for 00100004 ... 001003ff ==> d433b05b
=> md 100000 4
00100000: d433b05b ec3827e4 3cb0bacf 00093cf5    .3.[.8'.<.....<.
=>
```

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 [.b, .w, .l] addr1 addr2 count
    - compare memory

=>
```

With the cmp command you can test of 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 100000 40000000 400
word at 0x00100004 (0x50ff4342) != word at 0x40000004 (0x50504342)
Total of 1 word were the same
=> md 100000 C
00100000: 27051956 50ff4342 6f6f7420 312e312e    '..VP.CBoot 1.1.
00100010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
00100020: 2031393a 35353a30 34290000 00000000     19:55:04)......
=> md 40000000 C
40000000: 27051956 50504342 6f6f7420 312e312e    '..VPPCBoot 1.1.
40000010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
40000020: 2031393a 35353a30 34290000 00000000     19:55:04)......
=>
```

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 explicitely 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 100000 40000000 400
word at 0x00100004 (0x50ff4342) != word at 0x40000004 (0x50504342)
Total of 1 word were the same
=> cmp.w 100000 40000000 800
halfword at 0x00100004 (0x50ff) != halfword at 0x40000004 (0x5050)
Total of 2 halfwords were the same
=> cmp.b 100000 40000000 1000
byte at 0x00100005 (0xff) != byte at 0x40000005 (0x50)
Total of 5 bytes were the same
=>
```

5.9.2.4. cp - memory copy

```
=> help cp
cp [.b, .w, .l] source target count
    - copy memory

=>
```

The cp is used to copy memory areas.

```
=> cp 40000000 100000 10000
=>
```

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

Note: Included topic DULGData.tqm8xxlUBootCpExt does not exist yet

5.9.2.5. md - memory display

```
=> help md
md [.b, .w, .l] address [# of objects]
    - memory display

=>
```

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

```
=> md 100000
00100000: 27051956 50504342 6f6f7420 312e312e    '..VPPCBoot 1.1.
00100010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
00100020: 2031393a 35353a30 34290000 00000000     19:55:04)......
00100030: 00000000 00000000 00000000 00000000    ................
00100040: 00000000 00000000 00000000 00000000    ................
00100050: 00000000 00000000 00000000 00000000    ................
00100060: 00000000 00000000 00000000 00000000    ................
00100070: 00000000 00000000 00000000 00000000    ................
00100080: 00000000 00000000 00000000 00000000    ................
00100090: 00000000 00000000 00000000 00000000    ................
001000a0: 00000000 00000000 00000000 00000000    ................
001000b0: 00000000 00000000 00000000 00000000    ................
001000c0: 00000000 00000000 00000000 00000000    ................
001000d0: 00000000 00000000 00000000 00000000    ................
001000e0: 00000000 00000000 00000000 00000000    ................
001000f0: 00000000 00000000 00000000 00000000    ................
=>
00100100: 3c60fff0 7c7e9ba6 3aa00001 4800000c    <`..|~..:...H...
00100110: 3aa00002 48000004 38601002 7c600124    :...H...8`..|`.$
00100120: 7c7b03a6 7c7422a6 7c000278 7c1c23a6    |{..|t".|..x|.#.
00100130: 7c1d23a6 7c1623a6 7c1723a6 7c708aa6    |.#.|.#.|.#.|p..
00100140: 7c788aa6 3c600a00 7c708ba6 7c788ba6    |x..<`..|p..|x..
00100150: 3c600c00 7c708ba6 7c788ba6 3c600400    <`..|p..|x..<`..
00100160: 7c788ba6 3c600200 7c708ba6 7c0002e4    |x..<`..|p..|...
00100170: 4c00012c 3c604000 60630000 38630188    L..,<`@.`c..8c..
00100180: 7c6803a6 4e800020 3c60fff0 60612ec0    |h..N.. <`..`a..
00100190: 9401fffc 9401fffc 38400007 7c5e23a6    ........8@..|^#.
001001a0: 3c400000 60420000 7c5523a6 48000005    <@..`B..|U#.H...
001001b0: 7dc802a6 800e22bc 7dc07214 48019d41    }.....".}.r.H..A
001001c0: 7ea3ab78 4800c05d 00000000 00000000    ~..xH..]........
001001d0: 00000000 00000000 00000000 00000000    ................
001001e0: 00000000 00000000 00000000 00000000    ................
001001f0: 00000000 00000000 00000000 00000000    ................
=>
```

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

```
=> md.w 100000
00100000: 2705 1956 5050 4342 6f6f 7420 312e 312e    '..VPPCBoot 1.1.
00100010: 3520 284d 6172 2032 3120 3230 3032 202d    5 (Mar 21 2002 -
00100020: 2031 393a 3535 3a30 3429 0000 0000 0000     19:55:04)......
00100030: 0000 0000 0000 0000 0000 0000 0000 0000    ................
00100040: 0000 0000 0000 0000 0000 0000 0000 0000    ................
00100050: 0000 0000 0000 0000 0000 0000 0000 0000    ................
00100060: 0000 0000 0000 0000 0000 0000 0000 0000    ................
00100070: 0000 0000 0000 0000 0000 0000 0000 0000    ................
=> md.b 100000
00100000: 27 05 19 56 50 50 43 42 6f 6f 74 20 31 2e 31 2e    '..VPPCBoot 1.1.
00100010: 35 20 28 4d 61 72 20 32 31 20 32 30 30 32 20 2d    5 (Mar 21 2002 -
00100020: 20 31 39 3a 35 35 3a 30 34 29 00 00 00 00 00 00     19:55:04)......
00100030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
=>
```

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 100000 20
00100000: 27 05 19 56 50 50 43 42 6f 6f 74 20 31 2e 31 2e    '..VPPCBoot 1.1.
00100010: 35 20 28 4d 61 72 20 32 31 20 32 30 30 32 20 2d    5 (Mar 21 2002 -
=> md.w 100000
00100000: 2705 1956 5050 4342 6f6f 7420 312e 312e    '..VPPCBoot 1.1.
00100010: 3520 284d 6172 2032 3120 3230 3032 202d    5 (Mar 21 2002 -
```

```
00100020: 2031 393a 3535 3a30 3429 0000 0000 0000     19:55:04)......
00100030: 0000 0000 0000 0000 0000 0000 0000 0000    ................
=> md 100000
00100000: 27051956 50504342 6f6f7420 312e312e    '..VPPCBoot 1.1.
00100010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
00100020: 2031393a 35353a30 34290000 00000000     19:55:04)......
00100030: 00000000 00000000 00000000 00000000    ................
00100040: 00000000 00000000 00000000 00000000    ................
00100050: 00000000 00000000 00000000 00000000    ................
00100060: 00000000 00000000 00000000 00000000    ................
00100070: 00000000 00000000 00000000 00000000    ................
=>
```

5.9.2.6. mm - memory modify (auto-incrementing)

```
=> help mm
mm [.b, .w, .l] address
    - memory modify, auto increment address

=>
```

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 100000
00100000: 27051956 ? 0
00100004: 50504342 ? AABBCCDD
00100008: 6f6f7420 ? 01234567
0010000c: 312e312e ? .
=> md 100000 10
00100000: 00000000 aabbccdd 01234567 312e312e    .........#Eg1.1.
00100010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
00100020: 2031393a 35353a30 34290000 00000000     19:55:04)......
00100030: 00000000 00000000 00000000 00000000    ................
=>
```

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

```
=> mm.w 100000
00100000: 0000 ? 0101
00100002: 0000 ? 0202
00100004: aabb ? 4321
00100006: ccdd ? 8765
00100008: 0123 ? .
=> md 100000 10
00100000: 01010202 43218765 01234567 312e312e    ....C!.e.#Eg1.1.
00100010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
00100020: 2031393a 35353a30 34290000 00000000     19:55:04)......
00100030: 00000000 00000000 00000000 00000000    ................
=>

=> mm.b 100000
00100000: 01 ? 48
00100001: 01 ? 61
00100002: 02 ? 6c
00100003: 02 ? 6c
00100004: 43 ? 6f
00100005: 21 ? 20
00100006: 87 ? 20
00100007: 65 ? 20
```

```
00100008: 01 ? .
=> md 100000 10
00100000: 48616c6c 6f202020 01234567 312e312e    Hallo   .#Eg1.1.
00100010: 3520284d 61722032 31203230 3032202d    5 (Mar 21 2002 -
00100020: 2031393a 35353a30 34290000 00000000     19:55:04)......
00100030: 00000000 00000000 00000000 00000000    ................
=>
```

5.9.2.7. mtest - simple RAM test

```
=> help mtest
mtest [start [end [pattern]]]
    - simple RAM read/write test

=>
```

The `mtest` provides a simple memory test.

```
=> mtest 100000 200000
Testing 00100000 ... 00200000:
Pattern 0000000F  Writing...  Reading...
=>
```

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 firnware (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 [.b, .w, .l] address value [count]
    - write memory

=>
```

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 100000 10
00100000: 0000000f 00000010 00000011 00000012    ................
00100010: 00000013 00000014 00000015 00000016    ................
00100020: 00000017 00000018 00000019 0000001a    ................
00100030: 0000001b 0000001c 0000001d 0000001e    ................
=> mw 100000 aabbccdd
=> md 100000 10
00100000: aabbccdd 00000010 00000011 00000012    ................
00100010: 00000013 00000014 00000015 00000016    ................
00100020: 00000017 00000018 00000019 0000001a    ................
00100030: 0000001b 0000001c 0000001d 0000001e    ................
=> mw 100000 0 6
=> md 100000 10
00100000: 00000000 00000000 00000000 00000000    ................
00100010: 00000000 00000000 00000015 00000016    ................
00100020: 00000017 00000018 00000019 0000001a    ................
00100030: 0000001b 0000001c 0000001d 0000001e    ................
=>
```

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

```
=> mw.w 100004 1155 6
=> md 100000 10
00100000: 00000000 11551155 11551155 11551155    .....U.U.U.U.U.U
00100010: 00000000 00000000 00000015 00000016    ................
00100020: 00000017 00000018 00000019 0000001a    ................
00100030: 0000001b 0000001c 0000001d 0000001e    ................
=> mw.b 100007 ff 7
=> md 100000 10
00100000: 00000000 115511ff ffffffff ffff1155    .....U.........U
00100010: 00000000 00000000 00000015 00000016    ................
00100020: 00000017 00000018 00000019 0000001a    ................
00100030: 0000001b 0000001c 0000001d 0000001e    ................
=>
```

5.9.2.9. nm - memory modify (constant address)

```
=> help nm
nm [.b, .w, .l] address
    - memory modify, read and keep 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 100000
00100000: 00 ? 48
00100000: 48 ? 61
00100000: 61 ? 6c
00100000: 6c ? 6c
00100000: 6c ? 6f
00100000: 6f ? .
=> md 100000 8
00100000: 6f000000 115511ff ffffffff ffff1155    o....U.........U
00100010: 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 [.b, .w, .l] address number_of_objects
    - loop on a set of addresses

=>
```

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 [.b, .w, .l] source target count
    - copy memory

=>
```

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 100000 40000000 10000
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 100000 40000000 10000
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: FUJITSU 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

Bank # 2: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40400000      40408000      4040C000      40410000      40420000
    40440000      40460000      40480000      404A0000      404C0000
    404E0000      40500000      40520000      40540000      40560000
    40580000      405A0000      405C0000      405E0000      40600000
    40620000      40640000      40660000      40680000      406A0000
    406C0000      406E0000      40700000      40720000      40740000
    40760000      40780000      407A0000      407C0000      407E0000
=>
```

5.9.3.3. erase - erase FLASH memory

```
=> help era
erase start end
    - erase FLASH from addr 'start' to addr 'end'
erase N:SF[-SL]
    - erase sectors SF-SL in FLASH bank # N
erase bank N
    - erase FLASH bank # N
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 40040000 402FFFFF
Erase Flash from 0x40040000 to 0x402fffff
.............. done
Erased 22 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 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 1:6-8
Erase Flash Sectors 6-8 in Bank # 1
.. done
=>
```

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

Note: Included topic DULGData.tqm8xxlUBootEraseBank does not exist yet

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: 5 protected sectors will not be erased!
................... done
Erase Flash Bank # 2
......................... 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 on  start end
    - protect FLASH from addr 'start' to addr 'end'
protect on  N:SF[-SL]
    - protect sectors SF-SL in FLASH bank # N
protect on  bank N
    - protect FLASH bank # N
protect on  all
    - protect all FLASH banks
protect off start end
    - make FLASH from addr 'start' to addr 'end' writable
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 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:

```
=> fli

Bank # 1: FUJITSU 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

Bank # 2: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40400000      40408000      4040C000      40410000      40420000
    40440000      40460000      40480000      404A0000      404C0000
    404E0000      40500000      40520000      40540000      40560000
    40580000      405A0000      405C0000      405E0000      40600000
    40620000      40640000      40660000      40680000      406A0000
    406C0000      406E0000      40700000      40720000      40740000
```

```
    40760000      40780000      407A0000      407C0000      407E0000
=> protect on 40100000 401FFFFF
Protected 8 sectors
=> fli

Bank # 1: FUJITSU 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 (RO) 40120000 (RO) 40140000 (RO) 40160000 (RO)
    40180000 (RO) 401A0000 (RO) 401C0000 (RO) 401E0000 (RO) 40200000
    40220000      40240000      40260000      40280000      402A0000
    402C0000      402E0000      40300000      40320000      40340000
    40360000      40380000      403A0000      403C0000      403E0000

Bank # 2: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40400000      40408000      4040C000      40410000      40420000
    40440000      40460000      40480000      404A0000      404C0000
    404E0000      40500000      40520000      40540000      40560000
    40580000      405A0000      405C0000      405E0000      40600000
    40620000      40640000      40660000      40680000      406A0000
    406C0000      406E0000      40700000      40720000      40740000
    40760000      40780000      407A0000      407C0000      407E0000
=> era 40100000 401FFFFF
Erase Flash from 0x40100000 to 0x401fffff - Warning: 8 protected sectors will not be erased!
 done
Erased 8 sectors
=> protect off 1:11
Un-Protect Flash Sectors 11-11 in Bank # 1
=> fli

Bank # 1: FUJITSU 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 (RO) 40140000 (RO) 40160000 (RO)
    40180000 (RO) 401A0000 (RO) 401C0000 (RO) 401E0000 (RO) 40200000
    40220000      40240000      40260000      40280000      402A0000
    402C0000      402E0000      40300000      40320000      40340000
    40360000      40380000      403A0000      403C0000      403E0000

Bank # 2: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40400000      40408000      4040C000      40410000      40420000
    40440000      40460000      40480000      404A0000      404C0000
    404E0000      40500000      40520000      40540000      40560000
    40580000      405A0000      405C0000      405E0000      40600000
    40620000      40640000      40660000      40680000      406A0000
    406C0000      406E0000      40700000      40720000      40740000
    40760000      40780000      407A0000      407C0000      407E0000
=> era 1:11
Erase Flash Sectors 11-11 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_JFFS2_CMDLINE
#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
    - 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'<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 some target system the `mtdparts` command might display this information:

```
=> mtdparts

device nor0 <TQM5200-0>, # parts = 4
 #: name                        size            offset          mask_flags
 0: firmware            0x00100000      0x00000000      1
 1: kernel              0x00180000      0x00100000      0
 2: small-fs            0x00d80000      0x00280000      0
 3: big-fs              0x01000000      0x01000000      0

active partition: nor0,0 - (firmware) 0x00100000 @ 0x00000000

defaults:
mtdids  : nor0=TQM5200-0
mtdparts: mtdparts=TQM5200-0:1m(firmware),1536k(kernel),3584k(small-fs),2m(initrd),8m(misc),16m(b
```

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=TQM5200-0:1024k(firmware)ro,1536k(kernel),13824k(small-fs),16m(big-fs)
```

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

Now let's switch back to the default settings:

```
=> mtdparts default
=> mtdparts

device nor0 <TQM5200-0>, # parts = 6
 #: name                        size            offset          mask_flags
 0: firmware            0x00100000      0x00000000      0
 1: kernel              0x00180000      0x00100000      0
 2: small-fs            0x00380000      0x00280000      0
 3: initrd              0x00200000      0x00600000      0
 4: misc                0x00800000      0x00800000      0
 5: big-fs              0x01000000      0x01000000      0

active partition: nor0,0 - (firmware) 0x00100000 @ 0x00000000

defaults:
mtdids  : nor0=TQM5200-0
mtdparts: mtdparts=TQM5200-0:1m(firmware),1536k(kernel),3584k(small-fs),2m(initrd),8m(misc),16m(b
=> print mtdparts
mtdparts=mtdparts=TQM5200-0:1m(firmware),1536k(kernel),3584k(small-fs),2m(initrd),8m(misc),16m(bi
```

Then we delete the last 4 partitions ("small-fs", "initrd", "misc" and "big-fs") ...

```
=> mtdparts del small-fs
=> mtdparts del initrd
=> mtdparts del misc
```

```
=> mtdparts del big-fs
=> mtdparts

device nor0 <TQM5200-0>, # parts = 2
 #: name                        size            offset          mask_flags
 0: firmware            0x00100000      0x00000000      0
 1: kernel              0x00180000      0x00100000      0

active partition: nor0,0 - (firmware) 0x00100000 @ 0x00000000

defaults:
mtdids  : nor0=TQM5200-0
mtdparts: mtdparts=TQM5200-0:1m(firmware),1536k(kernel),3584k(small-fs),2m(initrd),8m(misc),16m(b
```

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

```
=> mtdparts add nor0 - new-part
=> mtdparts

device nor0 <TQM5200-0>, # parts = 3
 #: name                        size            offset          mask_flags
 0: firmware            0x00100000      0x00000000      0
 1: kernel              0x00180000      0x00100000      0
 2: new-part            0x01d80000      0x00280000      0

active partition: nor0,0 - (firmware) 0x00100000 @ 0x00000000

defaults:
mtdids  : nor0=TQM5200-0
mtdparts: mtdparts=TQM5200-0:1m(firmware),1536k(kernel),3584k(small-fs),2m(initrd),8m(misc),16m(b
=> print mtdparts
mtdparts=mtdparts=TQM5200-0:1m(firmware),1536k(kernel),30208k(new-part)
```

5.9.4. Execution Control Commands

5.9.4.1. autoscr - run script from memory

```
=> help autoscr
autoscr [addr] - run script starting at addr. A valid autoscr header must be present

=>
```

With the `autoscr` 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 `autoscr` you can run the commands in such an image. For instance, the following text file:

```
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:

```
bash$ mkimage -A ppc -O linux -T script -C none -a 0 -e 0 \
> -n "autoscr example script" \
> -d /tftpboot/TQM860L/example.script /tftpboot/TQM860L/example.img
Image Name:   autoscr example script
Created:      Mon Apr  8 01:15:02 2002
Image Type:   PowerPC Linux Script (uncompressed)
Data Size:    157 Bytes = 0.15 kB = 0.00 MB
Load Address: 0x00000000
Entry Point:  0x00000000
Contents:
   Image 0:      149 Bytes =    0 kB = 0 MB
```

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

```
=> tftp 100000 /tftpboot/TQM860L/example.img
ARP broadcast 1
TFTP from server 10.0.0.2; our IP address is 10.0.0.99
Filename '/tftpboot/TQM860L/example.img'.
Load address: 0x100000
Loading: #
done
Bytes transferred = 221 (dd hex)
=> autoscr 100000
## Executing script at 00100000

Network Configuration:
----------------------
Target:
ipaddr=10.0.0.99
hostname=tqm

Server:
serverip=10.0.0.2
rootpath=/opt/hardhat/devkit/ppc/8xx/target

=>
```

5.9.4.2. bootm - boot application image from memory

```
=> help bootm
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

=>
```

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 controll 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 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 informatoin 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 [loadAddress] [bootfilename]

=>
```

5.9.5.2. dhcp - invoke DHCP client to obtain IP/boot params

```
=> help dhcp
dhcp

=>
```

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

```
=> help loadb
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 [ 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
rarpboot [loadAddress] [bootfilename]

=>
```

5.9.5.6. tftpboot- boot image via network using TFTP protocol

```
=> help tftp
tftpboot [loadAddress] [bootfilename]

=>
```

5.9.6. Environment Variables Commands

5.9.6.1. printenv- print environment variables

```
=> help printenv
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=10.0.0.99
hostname=tqm
netmask=255.0.0.0
=>
```

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
baudrate=115200
serial#=TQM860LDDBA3-P50.203 10226122 4
ethaddr=00:D0:93:00:28:81
bootdelay=5
loads_echo=1
clocks_in_mhz=1
load=tftp 100000 /tftpboot/ppcboot.bin
update=protect off all;era 1:0-4;cp.b 100000 40000000 ${filesize};setenv filesize;saveenv
rtai=tftp 100000 /tftpboot/pImage.rtai;run nfsargs;run addip;bootm
preboot=echo;echo Type "run flash_nfs" to mount root filesystem over NFS;echo
nfsargs=setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath}
addip=setenv bootargs ${bootargs} ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}:${
flash_nfs=run nfsargs;run addip;bootm ${kernel_addr}
kernel_addr=40040000
netdev=eth0
hostname=tqm
rootpath=/opt/hardhat/devkit/ppc/8xx/target
ramargs=setenv bootargs root=/dev/ram rw
flash_self=run ramargs;run addip;bootm ${kernel_addr} ${ramdisk_addr}
ramdisk_addr=40100000
bootcmd=run flash_self
stdin=serial
stderr=serial
stdout=serial
filesize=dd
netmask=255.0.0.0
```

```
ipaddr=10.0.0.99
serverip=10.0.0.2

Environment size: 992/16380 bytes
=>
```

5.9.6.2. saveenv - save environment variables to persistent storage

```
=> help saveenv
saveenv - No help available.

=>
```

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 Enviroment to Flash...
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
=>
```

5.9.6.3. setenv - set environment variables

```
=> help setenv
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:

```
=> 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 `$` character that is used for variable substitution or the semicolon which separates commands). Use the backslash (`\`) character to escape such special characters, or enclose the whole phrase in apstrophes (`'`). Use `"${name}"` for variable expansion (see 14.2.11. 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 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.
=> printenv test
test=echo This is a test;printenv ipaddr;echo Done.
=> run test
This is a test
ipaddr=10.0.0.99
Done.
=>
```

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 serial#\;echo Done.
=> printenv test test2
test=echo This is a test;printenv ipaddr;echo Done.
test2=echo This is another Test;printenv serial#;echo Done.
=> run test test2
This is a test
ipaddr=10.0.0.99
```

```
Done.
This is another Test
serial#=TQM860LDDBA3-P50.203 10226122 4
Done.
=>
```

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 boot
bootd - No help available.

=>
```

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. Special Commands

5.9.7.1. i2c - I2C sub-system

```
=> help i2c
Unknown command 'i2c' - try 'help' without arguments for list of all known commands

=>
```

5.9.7.2. ide - IDE sub-system

```
=> help ide
ide reset - reset IDE controller
ide info  - show available IDE devices
ide device [dev] - show or set current device
ide part [dev] - print partition table of one or all IDE devices
ide read  addr blk# cnt
ide write addr blk# cnt - read/write `cnt' blocks starting at block `blk#'
    to/from memory address `addr'

=>
```

5.9.7.3. diskboot- boot from IDE device

```
=> help disk
diskboot loadAddr dev:part

=>
```

5.9.8. Miscellaneous Commands

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

```
=> help date
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
Date: 1970-01-01 (Thursday)    Time:  0:-1:-18
=> date 040723152002.35
Date: 2002-04-07 (Sunday)    Time: 23:15:35
=> date reset
Reset RTC...
Date: 2002-04-07 (Sunday)    Time: 23:15:36
=>
```

5.9.8.2. echo - echo args to console

```
=> help echo
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.
=>
```

5.9.8.3. reset - Perform RESET of the CPU

```
=> help reset
reset - No help available.

=>
```

The `reset` command reboots the system.

```
*** MISSING ***
```

5.9.8.4. sleep - delay execution for some time

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

=>
```

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

```
=> date ; sleep 5 ; date
```

```
Date: 2002-04-07 (Sunday)    Time: 23:15:40
Date: 2002-04-07 (Sunday)    Time: 23:15:45
=>
```

5.9.8.5. version - print monitor version

```
=> help version
version - No help available.

=>
```

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

```
=> version

PPCBoot 1.1.5 (Mar 21 2002 - 19:55:04)
=>
```

5.9.8.6. ? - 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

- **cpuclk**: (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 `cpuclk` variable can be set to the desired CPU clock value, in MHz. If the `cpuclk` 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 hass 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
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} ${files
setenv update_ramdisk run load_ramdisk install_ramdisk
echo ===== Save new definitions =====
saveenv
bash$
```

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 14.2.11. How the Command Line Parsing Works

5.12. U-Boot Standalone Applications

U-Boot allows to dynamically load and run "standalone" applications, which can use some resources of U-Boot like console I/O functions, memory allocation or 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.

This example, which demonstrates how to register a CPM interrupt handler with the U-Boot code, can be found in *examples/timer.c*. Here, a CPM timer is set up to generate an interrupt every second. The interrupt service routine is trivial, just printing a '.' character, but this is just a demo program. The application can be controlled by the following keys:

```
   ? - print current values og the CPM Timer registers
   b - enable interrupts and start timer
   e - stop timer and disable interrupts
   q - quit application

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

 => go 40004
   ## Starting application at 0x00040004 ...
   TIMERS=0xfff00980
   Using timer 1
     tgcr @ 0xfff00980, tmr @ 0xfff00990, trr @ 0xfff00994, tcr @ 0xfff00998, tcn @ 0xfff0099c, t

Hit 'b':
   [q, b, e, ?] Set interval 1000000 us
   Enabling timer
Hit '?':
   [q, b, e, ?] ........
   tgcr=0x1, tmr=0xff1c, trr=0x3d09, tcr=0x0, tcn=0xef6, ter=0x0
Hit '?':
   [q, b, e, ?] .
   tgcr=0x1, tmr=0xff1c, trr=0x3d09, tcr=0x0, tcn=0x2ad4, ter=0x0
Hit '?':
   [q, b, e, ?] .
   tgcr=0x1, tmr=0xff1c, trr=0x3d09, tcr=0x0, tcn=0x1efc, ter=0x0
Hit '?':
   [q, b, e, ?] .
   tgcr=0x1, tmr=0xff1c, trr=0x3d09, tcr=0x0, tcn=0x169d, ter=0x0
Hit 'e':
   [q, b, e, ?] ...Stopping timer
Hit 'q':
   [q, b, e, ?] ## Application terminated, rc = 0x0
```

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, ARTOS, Unity OS; Currently supported: Linux, NetBSD, VxWorks, QNX, RTEMS, ARTOS, Unity OS).

- Target CPU Architecture (Provisions for Alpha, ARM, Intel x86, IA64, MIPS, MIPS, PowerPC, IBM S390, SuperH, Sparc, Sparc 64 Bit, M68K, NIOS; Currently supported: ARM, PowerPC, MIPS, MIPS64, M68K, NIOS).
- Compression Type (Provisions for uncompressed, gzip, bzip2; Currently supported: uncompressed, gzip, bzip2).
- 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 and MPC82xx PowerPC processors.

5.14.2. Bitmap Support

By adding the **`CFG_CMD_BMP`** option to your **`CONFIG_COMMANDS`** command selections you can enable support for bitmap images in U-Boot. This will add **bmp** to the list of commands in your configuration of U-Boot:

```
=> help bmp
bmp info <imageAddr>    - display image info
bmp display <imageAddr> - display image
```

This command can be used to show information about bitmap images or to display the images on your screen.

Example:

```
=> tftp 100000 /tftpboot/LWMON/denk_startup.bmp
TFTP from server 192.168.3.1; our IP address is 192.168.3.74
Filename '/tftpboot/LWMON/denk_startup.bmp'.
Load address: 0x100000
Loading: #############################################################
done
Bytes transferred = 308278 (4b436 hex)
=> bmp info 100000
Image size    : 640 x 480
Bits per pixel: 8
Compression   : 0
=> bmp display 100000
```

To keep the code in U-Boot simple and as fast as possible, the bitmap images must match the color depth of your framebuffer device. For example, if your display is configured for a color depth of 8 bpp (bit per pixel) then the `bmp` command will complain if you try to load images with a different color depth:

```
=> tftp 100000 /tftpboot/LWMON/Bergkirchen.bmp
TFTP from server 192.168.3.1; our IP address is 192.168.3.74
Filename '/tftpboot/LWMON/Bergkirchen.bmp'.
Load address: 0x100000
Loading: #################################################################
         #################################################################
         ###################################################
done
Bytes transferred = 921654 (e1036 hex)
=> bmp i 100000
Image size    : 640 x 480
Bits per pixel: 24
Compression   : 0
=> bmp d 100000
Error: 8 bit/pixel mode, but BMP has 24 bit/pixel
```

(As you can see above, the sub-commands "`info`" and "`display`" can be abbreviated as "`i`" resp. "`d`" .)

Images that are bigger than your framebuffer device will be clipped on the top and right hand side.

Images that are smaller than the display will be loaded into the top left corner.

Since loading an image will define a new color map, the remainder of the display will appear with incorrect colors. It is therefore recommended that all images match exactly the size of the current display device. We accepted these restrictions since speed was top priority, and all attempts to implement scaling or optimizing the color maps would slow down the display too much. It is much easier to perform the necessary transformations on the development host, where a plethora of tools is available.

For example, to convert existing images to bitmap files with the required color depth (here: 8 bpp), the **"PBM"** -Tools can be used (PBM = portable pix map - see "`man 5 ppm`"):

```
bash$ jpegtopnm Bergkirchen.jpg | \
> ppmquant 256 | \
> ppmtobmp -bpp 8 >Bergkirchen-8bit.bmp
jpegtopnm: WRITING PPM FILE
ppmquant: making histogram...
ppmquant: too many colors!
ppmquant: scaling colors from maxval=255 to maxval=127 to improve clustering...
ppmquant: making histogram...
ppmquant: too many colors!
ppmquant: scaling colors from maxval=127 to maxval=63 to improve clustering...
ppmquant: making histogram...
ppmquant: 9760 colors found
ppmquant: choosing 256 colors...
ppmquant: mapping image to new colors...
ppmtobmp: analyzing colors...
ppmtobmp: 231 colors found
ppmtobmp: Writing 8 bits per pixel with a color pallette
```

This gives the following results on the target:

```
=> tftp 100000 /tftpboot/LWMON/Bergkirchen-8bit.bmp
TFTP from server 192.168.3.1; our IP address is 192.168.3.74
Filename '/tftpboot/LWMON/Bergkirchen-8bit.bmp'.
Load address: 0x100000
Loading: #############################################################
done
Bytes transferred = 308278 (4b436 hex)
=> bmp i 100000
Image size    : 640 x 480
Bits per pixel: 8
Compression   : 0
=> bmp d 100000
```

5.14.3. Splash Screen Support

Even if you manage to boot U-Boot and Linux into a graphical user application within 5 or 6 seconds of power-on (which is not difficult), many customers expect to see "something" immediately. U-Boot supports the concept of a *splash screen* for such purposes.

To enable splash screen support, you have to add a **"#define CONFIG_SPLASH_SCREEN"** to your board configuration file. This will also implicitly enable U-Boot Bitmap Support.

After power-on, U-Boot will test if the environment variable "`splashimage`" is defined, and if it contains the address of a valid bitmap image. If this is the case, the normal startup messages will be suppressed and the defined **splash screen** will be displayed instead. Also, all output (devices `stdout` and `stderr`) will be suppressed (redirected to the "`nulldev`" device).

For example, to install this feature on a system, proceed as follows:

```
=> tftp 100000 /tftpboot/denx_startup.bmp
TFTP from server 192.168.3.1; our IP address is 192.168.3.74
Filename '/tftpboot/denx_startup.bmp'.
Load address: 0x100000
Loading: #############################################################
done
Bytes transferred = 308278 (4b436 hex)
=> cp.b 100000 41F80000 $filesize
```

```
Copy to Flash... done
=> setenv splashimage 41F80000
=> saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
=> bmp info $splashimage
Image size    : 640 x 480
Bits per pixel: 8
Compression   : 0
```

Note that, for perfect operation, this option has to be complemented by matching Splash Screen Support in Linux.

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://www.denx.de/cgi-bin/gitweb.cgi. To checkout the module for the first time, proceed as follows:

```
bash$ cd /opt/eldk/usr/src
bash$ git clone git://www.denx.de/git/linuxppc_2_4_devel.git linuxppc_2_4_devel
bash$ cd linuxppc_2_4_devel
```

6.2. Kernel Configuration and Compilation

The TQM8xxL 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 "powerpc" 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 TQM8xxL board that has been extensively tested. It is recommended to use this as a starting point for other, customized configurations:

```
bash$ make tqm8xxl_config
```

The TQM8xxL boards are available in many configurations (different CPUs, with or without LCD display, with or without Fast Ethernet interface). Depending on the board configuration chose one of the following `make` targets:

TQM823L_config
TQM823L_LCD_config
TQM850L_config
TQM860L_config

Please use the TQM860L configuration for TQM855L boards.

Note: When you type "`make XXX_config`" this means that a default configuration file for the board named `XXX` gets selected. The name of this default configuration file is **`arch/""/configs/XXX_defconfig`** . By listing the contents of the **`arch/""/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 oldconfig
bash$ make dep
bash$ make uImage
```

Otherwise you can modify the kernel configuration as follows:

```
bash$ make config
```

or

```
bash$ make menuconfig
```

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

The **make** target **uImage** uses the tool **mkimage** (from the U-Boot package) to create a Linux kernel image in *arch/ppc/boot/images/uImage*

which is immediately usable for download and booting with U-Boot.

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

```
bash$ make modules
```

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

```
bash$ make INSTALL_MOD_PATH=/opt/eldk/ppc_8xx modules_install
```

If your host computer is not the same architecture as the target system, and if you got your kernel tree from kernel.org or other "official" sources, then you may have to supply an architecture override and a cross compiler definition. The most reliable way to do this is to specify them on the make command line as part of the `make` command. If this is the case, use for example:

```
bash$ make ARCH=ppc CROSS_COMPILE=ppc_8xx-
```

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/ppc/boot/images/uImage /tftpboot/uImage
```

7. Booting Embedded Linux

7.1. Introduction

In principle, if you have a Linux kernel image 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 0x40080000 - then you can boot this image with the following command:

```
=> bootm 40080000
```

7.2. 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 0x200000 using the `initrd` ramdisk image at address 0x400000 as root filesystem, you can use the following commands:

```
=> setenv bootargs root=/dev/ram rw
=> bootm 200000 400000
```

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 "10.0.0.2" and exports the directory "/opt/eldk/ppc_8xx" as root filesystem for the target. The target has been assigned the IP address "10.0.0.99" and the hostname "tqm". A netmask of "255.0.0.0" is used:

```
=> setenv bootargs root=/dev/nfs rw nfsroot=10.0.0.2:/opt/eldk/ppc_8xx ip=10.0.0.99:10.0.0.2:10.0
=> bootm 200000
```

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
```

7.3. 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 10.0.0.99
=> setenv serverip 10.0.0.2
=> setenv netmask 255.0.0.0
=> setenv hostname tqm
=> setenv rootpath /opt/eldk/ppc_8xx
=> 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}:${netmask}:${h
=> setenv ram_root 'run ramargs addip;bootm ${kernel_addr} ${ramdisk_addr}'
=> setenv nfs_root 'run nfsargs addip;bootm ${kernel_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, respecively. 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 two resp. one address argument(s) to boot the Linux kernel image with resp. without a ramdisk image. (We assume here that the variables `kernel_addr` and `ramdisk_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 will be explained in detail in the examples below.

7.4. Networked Operation with Root Filesystem over NFS

You can use the `printenv` command on the **Target** to find out which commands get executed by U-Boot to load and boot the Linux kernel:

```
=> printenv
bootcmd=bootp; setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath} ip=${ipaddr}:${se
bootdelay=5
baudrate=115200
stdin=serial
stdout=serial
stderr=serial
...
```

After Power-On or reset the system will initialize and then wait for a key-press on the console port. The duration of this countdown is determined by the contents of the `bootdelay` environment variable (default: 5 seconds).

If no key is pressed, the command (or the list of commands) stored in the environment variable `bootcmd` is executed. If you press a key, you get a prompt at the console port which allows for interactive command input.

In the example above the following commands are executed sequentially:

```
bootp
setenv bootargs root=/dev/nfs nfsroot=${serverip}:${rootpath} ip=${ipaddr}:${serverip}:${gatewayi
bootm
```

These commands take the following effect (pay attention for the modification of environment variables by these commands):

- `bootp`: This command uses the **BOOTP** protocol to ask a boot server for information about our system and to load a boot image (which will usually be a Linux kernel image). Since no aguments are passed to this command, it will use a default address to load the kernel image (0x100000 or the last address used by other operations).

```
=> bootp
BOOTP broadcast 1
ARP broadcast 0
TFTP from server 10.0.0.2; our IP address is 10.0.0.99
Filename '/tftpboot/TQM8xxL/uImage'.
Load address: 0x100000

Loading: #########################################################################################
done

=> printenv
bootcmd=bootp; setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath} ip=${ipaddr}:${se
bootdelay=5

baudrate=115200
stdin=serial
stdout=serial
stderr=serial
bootfile=/tftpboot/TQM8xxL/uImage
gatewayip=10.0.0.2
netmask=255.0.0.0
hostname=tqm
rootpath=/opt/eldk/ppc_8xx
ipaddr=10.0.0.99
serverip=10.0.0.2
dnsip=10.0.0.2
...
```

The **Target** sends a **BOOTP** request on the network, and (assuming there is a BOOTP server available) receives a reply that contains the IP address (`ipaddr=10.0.0.99`) and other network information for the target (`hostname=tqm`, `serverip=10.0.0.2`, `gatewayip=10.0.0.2`, `netmask=255.0.0.0`).

Also, the name of the boot image (`bootfile= /tftpboot/TQM8xxL/uImage`) and the root directory on a NFS server (`rootpath=/opt/eldk/ppc_8xx`) was transmitted.

U-Boot then automatically downloaded the bootimage from the server using **TFTP**.

You can use the command `iminfo` (Image Info, or short `imi`) to verify the contents of the loaded image:

```
=> imi 100000

## Checking Image at 00100000 ...
   Image Name:   Linux-2.4.4
   Created:      2002-04-07  21:31:59 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    605429 Bytes = 591 kB = 0 MB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
=>
```

This tells you that we loaded a compressed Linux kernel image, and that the file was not corrupted, since the CRC32 checksum is OK.

```
setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath} \
ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}::off
```

This command defines the environment variable `bootargs`. (If an old definition exists, it is deleted first). The contents of this variable is passed as command line to the LInux kernel when it is booted (hence the name). Note how U-Boot uses variable substitution to dynamically modify the boot arguments depending on the information we got from the **BOOTP** server.

To verify, you can run this command manually:

```
=> setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath} ip=${ipaddr}:${serverip}:${ga

=> printenv
...
bootargs=root=/dev/nfs rw nfsroot=10.0.0.2:/opt/eldk/ppc_8xx ip=10.0.0.99:10.0.0.2:10.0.0.2:255.0
...
```

This command line passes the following information to the Linux kernel:

- `root=/dev/nfs rw`: the root filesystem will be mounted using NFS, and it will be writable.
- `nfsroot=10.0.0.2:/opt/eldk/ppc_8xx`: the NFS server has the IP address 10.0.0.2, and exports the directory */opt/eldk/ppc_8xx* for our system to use as root filesystem.
- `ip=10.0.0.99:10.0.0.2:10.0.0.2:255.0.0.0:tqm::off`: the target has the IP address `10.0.0.99`; the NFS server is `10.0.0.2`; there is a gateway at IP address `10.0.0.2`; the netmask is `255.0.0.0` and our hostname is `tqm`. The first ethernet interface (`eth0`) willbe 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.

- `bootm`: This command boots an operating system image that resides somewhere in the system memory (RAM or flash - the `m` in the name is for memory). In this case we do not pass any memory address for the image, so the load address 0x100000 from the previous TFTP transfer is used:

```
=> run flash_nfs
## Booting image at 40040000 ...
   Image Name:   Linux-2.4.4
   Created:      2002-04-07  21:31:59 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    605429 Bytes = 591 kB = 0 MB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
Linux version 2.4.4 (wd@larry.denx.de) (gcc version 2.95.3 20010111 (prerelease/franzo/20010111))
On node 0 totalpages: 16384
zone(0): 16384 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/nfs rw nfsroot=10.0.0.2:/opt/hardhat/devkit/ppc/8xx/target ip=10.0
Decrementer Frequency: 3125000
Calibrating delay loop... 49.86 BogoMIPS
Memory: 62580k available (1164k kernel code, 564k data, 52k init, 0k highmem)
Dentry-cache hash table entries: 8192 (order: 4, 65536 bytes)
Buffer-cache hash table entries: 4096 (order: 2, 16384 bytes)
Page-cache hash table entries: 16384 (order: 4, 65536 bytes)
Inode-cache hash table entries: 4096 (order: 3, 32768 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Starting kswapd v1.8
CPM UART driver version 0.03
ttyS0 on SMC1 at 0x0280, BRG1
ttyS1 on SMC2 at 0x0380, BRG2
pty: 256 Unix98 ptys configured
block: queued sectors max/low 41520kB/13840kB, 128 slots per queue
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
```

```
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)
No card in slot B: PIPR=ff00ff00
eth0: CPM ENET Version 0.2 on SCC1, 00:d0:93:00:28:81
JFFS version 1.0, (C) 1999, 2000  Axis Communications AB
JFFS2 version 2.1. (C) 2001 Red Hat, Inc., designed by Axis Communications AB.^M Amd/Fujitsu Exte
number of JEDEC chips: 1
0: offset=0x0,size=0x8000,blocks=1
1: offset=0x8000,size=0x4000,blocks=2
2: offset=0x10000,size=0x10000,blocks=1
3: offset=0x20000,size=0x20000,blocks=31
 Amd/Fujitsu Extended Query Table v1.0 at 0x0040
number of JEDEC chips: 1
0: offset=0x0,size=0x8000,blocks=1
1: offset=0x8000,size=0x4000,blocks=2
2: offset=0x10000,size=0x10000,blocks=1
3: offset=0x20000,size=0x20000,blocks=31
TQM flash bank 0: Using static image partition definition
Creating 4 MTD partitions on "TQM8xxL Bank 0":
0x00000000-0x00040000 : "ppcboot"
0x00040000-0x00100000 : "kernel"
0x00100000-0x00200000 : "user"
0x00200000-0x00400000 : "initrd"
TQM flash bank 1: Using static file system partition definition
Creating 2 MTD partitions on "TQM8xxL Bank 1":
0x00000000-0x00200000 : "cramfs"
0x00200000-0x00400000 : "jffs"
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 4096 bind 4096)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
Looking up port of RPC 100003/2 on 10.0.0.2
Looking up port of RPC 100005/2 on 10.0.0.2
VFS: Mounted root (nfs filesystem).
Freeing unused kernel memory: 52k init
modprobe: modprobe: Can't locate module char-major-4
INIT: version 2.78 booting
Activating swap...
Checking all file systems...
Parallelizing fsck version 1.19 (13-Jul-2000)
Mounting local filesystems...
not mounted anything
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 2
Starting internet superserver: inetd.

MontaVista Software's Hard Hat Linux 2.0

tqm login: root
PAM-securetty[76]: Couldn't open /etc/securetty
PAM_unix[76]: (login) session opened for user root by LOGIN(uid=0)
Last login: Fri Feb  1 02:30:32 2030 on console
Linux tqm 2.4.4 #1 Sun Apr 7 23:28:08 MEST 2002 ppc unknown
login[76]: ROOT LOGIN on `console'

root@tqm:~#
```

7.5. Boot from 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 flash memory of your TQM8xxL board.

The U-Boot command `flinfo` can be used to display information about the available on-board flash on your system:

```
=> fli

Bank # 1: FUJITSU 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

Bank # 2: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
  Size: 4 MB in 35 Sectors
  Sector Start Addresses:
    40400000      40408000      4040C000      40410000      40420000
    40440000      40460000      40480000      404A0000      404C0000
    404E0000      40500000      40520000      40540000      40560000
    40580000      405A0000      405C0000      405E0000      40600000
    40620000      40640000      40660000      40680000      406A0000
    406C0000      406E0000      40700000      40720000      40740000
    40760000      40780000      407A0000      407C0000      407E0000
=>
```

From this output you can see the total amount of flash memory, and how it is divided in blocks (*Erase Units* or *Sectors*). The `RO` markers show blocks of flash memory that are write protected (by software) - this is the area where U-Boot is stored. The remaining flash memory is available for other use.

For instance, we can store the Linux kernel image in flash starting at the start address of the next free flash sector. Before we can do this we must make sure that the flash memory in that region is empty - a Linux kernel image is typically around 600...700 kB, so to be on the safe side we dedicate the whole area from 0x40080000 to 0x4027FFFF for the kernel image. Keep in mind that with flash memory only whole erase units can be cleared.

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.

Note: Included topic DULGData.tqm8xxlInstallKernelTftp does not exist yet

Note how the `filesize` variable (which gets set by the TFTP transfer) is used to automatically adjust for the actual image size.

Now we can boot directly from flash. All we need to do is passing the in-flash address of the image (40080000) with the `bootm` command; we also make the definition of the `bootargs` variable permanent now:

```
=> setenv bootcmd bootm 40080000
=> setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath} ip=${ipaddr}:${serverip}:${ga
```

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 40080000
bootargs=root=/dev/nfs rw nfsroot=10.0.0.2:/opt/eldk/ppc_8xx
ip=10.0.0.99:10.0.0.2:10.0.0.2:255.0.0.0:tqm::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:

Note: Included topic DULGData.tqm8xxlLinuxBootSelf does not exist yet

7.6. 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.

Ramdisk images for tests can be found in the ftp://ftp.denx.de/pub/LinuxPPC/usr/src/SELF/images/ directories.

Load the ramdisk image into RAM and write it to flash as follows:

Note: Included topic DULGData.tqm8xxlUBootInstallRamdisk does not exist yet

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:

Note: Included topic DULGData.tqm8xxlLinuxBootSelf does not exist yet

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_tqm8xxl=y
```

Note: this configuration uses **CFI** conformant AMD flash chips; you may need to adjust these settings on other boards.

The layout of your flash devices ("partitioning") is defined by the mapping routines for your board in the Linux MTD sources (see *drivers/mtd/maps/*). The configuration for the TQM8xxL looks like this:

```
/* partition definition for first flash bank
 * also ref. to "drivers\char\flash_config.c"
 */
static struct mtd_partition tqm8xxl_partitions[] = {
```

```
        {
          name: "ppcboot",
          offset: 0x00000000,
          size: 0x00020000,           /* 128KB           */
          mask_flags: MTD_WRITEABLE,  /* force read-only */
        },
        {
          name: "kernel",             /* default kernel image */
          offset: 0x00020000,
          size: 0x000e0000,
          mask_flags: MTD_WRITEABLE,  /* force read-only */
        },
        {
          name: "user",
          offset: 0x00100000,
          size: 0x00100000,
        },
        {
          name: "initrd",
          offset: 0x00200000,
          size: 0x00200000,
        }
};
/* partition definition for second flahs bank */
static struct mtd_partition tqm8xxl_fs_partitions[] = {
        {
          name: "cramfs",
          offset: 0x00000000,
          size: 0x00200000,
        },
        {
          name: "jffs",
          offset: 0x00200000,
          size: 0x00200000,
          //size: MTDPART_SIZ_FULL,
        }
};
```

This splits the available flash memory (8 MB in this case) into 6 separate "partitions":

- uboot: size: 128 kB; used to store the U-Boot firmware
- kernel: size: 896kB; used to store the (compressed) Linux kernel image
- user: size: 1 MB; not used
- initrd: size: 2 MB; used to store a (compressed) ramdisk image
- cramfs: size: 2 MB; used for a compressed ROM filesystem (read-only)
- jffs: size: 2 MB; used for a flash filesystem (using **JFFS**)

When you boot a system with this configuration you will see the following kernel messages on the console:

Note: Included topic DULGData.tqm8xxlLinuxMtdBoot does not exist yet

Another way to check this information when the system is running is using the `proc` filesystem:

Note: Included topic DULGData.tqm8xxlLinuxProcMtd does not exist yet

Now we can run some basic tests to verify that the flash driver routines and the partitioning works as expected:

```
# xd /dev/mtd0 | head -4
       0  27051956 7fe5f641  3be91e9d 0008061f  |'  V   A;        |
      10  00000000 00000000  7667315e 05070201  |        vg1^     |
      20  4c696e75 782d322e  342e3400 00000000  |Linux-2.4.4      |
```

```
      30  00000000 00000000  00000000 00000000  |                |
# xd /dev/mtd1 | head -4
       0  27051956 6735cb88  3be79508 000d11bf  |'  Vg5  ;       |
      10  00000000 00000000  7d5cbfc8 05070301  |        }\      |
      20  4170706c 69636174  696f6e20 72616d64  |Application ramd|
      30  69736b20 696d6167  65000000 00000000  |isk image       |
# xd /dev/mtd6 | head -10
       0  6a0358f7 626f6f74  64656c61 793d3500  |j X bootdelay=5 |
      10  62617564 72617465  3d393630 30006c6f  |baudrate=9600 lo|
      20  6164735f 6563686f  3d310063 6c6f636b  |ads_echo=1 clock|
      30  735f696e 5f6d687a  3d310065 74686164  |s_in_mhz=1 ethad|
      40  64723d30 303a6362  3a62643a 30303a30  |dr=00:cb:bd:00:0|
      50  303a3131 006e6673  61726773 3d736574  |0:11 nfsargs=set|
      60  656e7620 626f6f74  61726773 20726f6f  |env bootargs roo|
      70  743d2f64 65762f6e  66732072 77206e66  |t=/dev/nfs rw nf|
      80  73726f6f 743d2428  73657276 65726970  |sroot=$(serverip|
      90  293a2428 726f6f74  70617468 29007261  |):$(rootpath) ra|
# xd /dev/mtd7
       0  ffffffff ffffffff  ffffffff ffffffff  |                |
                    *** same ***
   80000
```

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 wih.

The last output shows the partition to be empty. We can try write some data into it:

```
# date >/dev/mtd7
# xd /dev/mtd7
       0  57656420 4e6f7620  20372031 353a3339  |Wed Nov  7 15:39|
      10  3a313220 4d455420  32303031 0affffff  |:12 MET 2001    |
      20  ffffffff ffffffff  ffffffff ffffffff  |                |
                    *** same ***
   80000                                        |                |
# sleep 10 ; date >/dev/mtd7
Last[3] is 3aa73020, datum is 3a343020
date: write error: Input/output error
```

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:

```
# xd /dev/mtd7
       0  57656420 4e6f7620  20372031 353a3339  |Wed Nov  7 15:39|
      10  3a303020 4d455420  32303031 0affffff  |:00 MET 2001    |
      20  ffffffff ffffffff  ffffffff ffffffff  |                |
                    *** same ***
   80000                                        |                |
# eraseall /dev/mtd7
Erased 512 Kibyte @ 0 -- 100% complete.
# xd /dev/mtd7
       0  ffffffff ffffffff  ffffffff ffffffff  |                |
                    *** same ***
   80000                                        |                |
# date >/dev/mtd7
# xd /dev/mtd7
       0  57656420 4e6f7620  20372031 353a3432  |Wed Nov  7 15:42|
      10  3a313920 4d455420  32303031 0affffff  |:19 MET 2001    |
      20  ffffffff ffffffff  ffffffff ffffffff  |                |
                    *** same ***
   80000
```

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,wsize=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 sufficent 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 PowerPC 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,wsize=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 filesyste 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
```

9.2. 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/fileysystems*:

```
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 PC Cards for Flash Disks, CompactFlash, and IDE Harddisks

If your board is equipped with a **PC-Card** adapter (also known as **PCMCIA** adapter) you can use this for miscellaneous types of mass storage devices like Flash Disks, CompactFlash, and IDE Harddisks.

Please note that there are other options to operate such devices on Embedded PowerPC Systems (for instace you can use the PCMCIA controller builtin to the MPC8xx CPUs to build a direct IDE interface, or you can use some external controller to provide such an interface). The following description does not cover such configurations. Only the solution which uses a standard PC Card Slot is described here.

9.3.1. PC Card Support in U-Boot

When PC Card support is enabled in your U-Boot configuration the target will try to detect any PC Cards in the slot when booting. If no card is present you will see a message like this:

```
PPCBoot 1.1.1 (Nov 11 2001 - 18:06:06)

CPU:   XPC862PZPnn0 at 48 MHz: 16 kB I-Cache 8 kB D-Cache FEC present
Board: ICU862 Board
DRAM:  32 MB
FLASH: 16 MB
In:    serial
Out:   serial
Err:   serial
PCMCIA:   No Card found
```

Depending on the type of PC Card inserted the boot messages vary; for instance with a Flash Disk card you would see:

```
...
PCMCIA: 3.3V card found: SunDisk SDP 5/3 0.6
            Fixed Disk Card
            IDE interface
            [silicon] [unique] [single] [sleep] [standby] [idle] [low power]
Bus 0: OK
  Device 0: Model: SanDisk SDP3B-8 Firm: Vdd 1.02 Ser#: fq9bu499900
            Type: Removable Hard Disk
            Capacity: 7.7 MB = 0.0 GB (15680 x 512)
...
```

With a CompactFlash Card you get:

```
...
PCMCIA: 3.3V card found:   CF 128MB CH
            Fixed Disk Card
            IDE interface
            [silicon] [unique] [single] [sleep] [standby] [idle] [low power]
Bus 0: OK
  Device 0: Model: CF 128MB Firm: Rev 1.01 Ser#: 1969C32AA0210002
            Type: Removable Hard Disk
            Capacity: 122.3 MB = 0.1 GB (250368 x 512)
```

```
...
```

Even more exotic memory devices (like the "MemoryStick as used in some Digital Cameras") will usually work without problems:

```
...
PCMCIA: 3.3V card found: SONY MEMORYSTICK(128M) 1.0
            Fixed Disk Card
            IDE interface
            [silicon] [unique] [single] [sleep] [standby] [idle] [low power]
Bus 0: .OK
  Device 0: Model: MEMORYSTICK 128M 16K                    Firm: SONY1.00` Ser#:
            Type: Removable Hard Disk
            Capacity: 123.8 MB = 0.1 GB (253696 x 512)
...
```

And with a harddisk adapter you would see:

```
...
PCMCIA: 5.0V card found: ARGOSY PnPIDE D5
Bus 0: OK
  Device 0: Model: IBM-DKLA-24320 Firm: KL4AA43A Ser#: YD2YD246800
            Type: Hard Disk
            Capacity: 4126.10 MB = 4.0 GB (8452080 x 512)
...
```

Note that most other cards will be detected by U-Boot, but not supported otherwise, for instance:

```
...
PCMCIA: 5.0V card found: ELSA AirLancer MC-11 Version 01.01
            Network Adapter Card
...
```

or

```
...
PCMCIA: 5.0V card found: Elsa MicroLink 56k MC Internet 021 A
            Serial Port Card
...
```

9.3.2. PC Card Support in Linux

The standard way to use PC Cards in a Linux system is to install the "PCMCIA Card Services" package. This is a quite complex set of kernel modules and tools that take care of things like automatic detection and handling of "card insert" or "remove" events, identification of the inserted cards, loading the necessary device drivers, etc. This is a very powerful package, but for embedded applications it has several serious disadvantages:

- Memory footprint - the package consists of a lot of tools and modules that take a lot of space both in the root filesystem and in system RAM when running
- Chicken and Egg Problem - the package loads the needed device drivers as kernel modules, so it needs a root filesystem on another device; that means that you cannot easily put the root filesystem on a PC Card.

For "disk" type PC Cards (FlashDisks, CompactFlash, Hard Disk Adapters - basicly 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 willnot 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 PowerPC systems, there are two commonly used options:

9.3.2.1. Using a MacOS Partition Table

A MacOS partition table is the "native" partition table format on PowerPC systems; most desktop PowerPC systems use it, so you may prefer it when you have PowerPC 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 &lt;nth&gt;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 &lt;nth&gt;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 chosing a partition size of 2 MB is more than generous. 2 MB coresponds 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*1024*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
```

9.3.2.2. 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
```

9.3.3. Using PC Card "disks" with U-Boot and Linux

U-Boot provides only basic functionality to access PC Card based "disks": you can print the partition table and read and write blocks (addressed by absolute block number), but there is no support to create new partitions or to read files from any type of filesystem.

[Such features could be easily added as U-Boot extensions aka "standalone programs", but so far it has not been implemented yet.]

As usual, you can get some information about the available IDE commands using the `help` command in U-Boot:

```
=> help ide
ide reset - reset IDE controller
ide info  - show available IDE devices
ide device [dev] - show or set current device
ide part [dev] - print partition table of one or all IDE devices
ide read  addr blk# cnt
ide write addr blk# cnt - read/write `cnt' blocks starting at block `blk#'
    to/from memory address `addr'
```

That means you will have to partition the "disk" on your host system; U-Boot can be configured for DOS and MacOS type partition tables. Since U-Boot cannot read files from a filesystem you should create one (or more) small partitions (maybe 1 MB or so) if you want to boot from the "disk".

For example on a 128 MB CompactFlash card we could create the following partiton table under Linux:

```
# fdisk /dev/hda
 hda: hda1 hda2 hda3 hda4

Command (m for help): p

Disk /dev/hda: 8 heads, 32 sectors, 978 cylinders
Units = cylinders of 256 * 512 bytes

   Device Boot    Start       End    Blocks   Id  System
/dev/hda1             1        17      2160   83  Linux
/dev/hda2            18        34      2176   83  Linux
/dev/hda3            35       803     98432   83  Linux
/dev/hda4           804       978     22400   82  Linux swap

Command (m for help): q

# mkswap /dev/hda4
Setting up swapspace version 1, size = 22933504 bytes
```

Here we have two small boot partitions (*/dev/hda1* and */dev/hda2*, 2 MB each), one big partition to hold a filesystem (*/dev/hda3*, 99 MB), and a swap partition (*/dev/hda4*, 22 MB). We also initialized */dev/hda4* as swap space.

U-Boot will recognize this partition table as follows:

```
=> ide part

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

Partition     Start Sector     Num Sectors     Type
    1                   32            4320      83
    2                 4352            4352      83
    3                 8704          196864      83
    4               205568           44800      82
```

We can now load a Linux kernel image over ethernet and store it both of the boot partitions:

```
=> tftp 100000 /tftpboot/uImage
ARP broadcast 1
TFTP from server 10.0.0.2; our IP address is 10.0.0.99
Filename '/tftpboot/uImage'.
Load address: 0x100000
Loading: #################################################################
         ##############################################
done
Bytes transferred = 566888 (8a668 hex)
=> ide write 100000 0x20 0x800

IDE write: device 0 block # 32, count 2048 ... 2048 blocks written: OK
=> ide write 100000 0x1100 0x800

IDE write: device 0 block # 4352, count 2048 ... 2048 blocks written: OK
```

This requires a little more explanation: as you can see from the output of the `help ide` command, the `write` subcommand takes 3 arguments: a memory address from where the data are read, an (absolute) block number on the disk where the writing starts, and a number of disk blocks.

Since U-Boot expects all input in hex notation we have to perform some calculation: partition 1 starts at block (or sector) number 32, which is 0x20; partition 2 starts at block number 4352 = 0x1100.

We used a block count of 0x800 = 2048 in both cases - this means we wrote 2048 block of 512 bytes each, or a 1024 kB - much more than the actual size of the LInux kernel image - but the partition is big enough and we are on the safe side, so we didn't bother to calculate the exact block count.

To boot from a disk you can use the `diskboot` command:

```
=> help diskboot
diskboot loadAddr dev:part
```

The `diskboot` command (or short `disk`) expects a load address in RAM, and a combination of device and partition numbers, separated by a colon. It then reads the image from disk and stores it in memory. We can now boot it using the `bootm` command [to automatically boot the image define the U-Boot environment `autostart` with the value =yes=].

```
=> disk 400000 0:1

Loading from IDE device 0, partition 1: Name: hda1
  Type: PPCBoot
   Image Name:   Linux-2.4.4
   Created:      2001-11-11  18:11:11 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    566824 Bytes = 553 kB = 0 MB
   Load Address: 00000000
   Entry Point:  00000000
=> bootm 400000
## Booting image at 00400000 ...
   Image Name:   Linux-2.4.4
   Created:      2001-11-11  18:11:11 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    566824 Bytes = 553 kB = 0 MB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
Linux version 2.4.4 (wd@denx.denx.de) (gcc version 2.95.2 19991024 (release)) #1 Sun Nov 11 19:05
On node 0 totalpages: 8192
...
```

We can use the same method that we used to store a Linux kernel image to a disk partition to load a filesystem image into another partiton - as long as the image fits into physical RAM - but usually it's easier to initialize the filesystem either on the host system (swapping the PC Card between host and target is easy enough), or you can use the configuration with root filesystem over NFS to populate the filesystem on the target.

You only have to set the `bootargs` variable to boot Linux with root filesystem on disk, for instance:

```
=> setenv bootargs root=/dev/hda3
=> setenv autostart yes
=> disk 400000 0:1

Loading from IDE device 0, partition 1: Name: hda1
  Type: PPCBoot
   Image Name:   Linux-2.4.4
   Created:      2001-11-11  18:11:11 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    566824 Bytes = 553 kB = 0 MB
   Load Address: 00000000
   Entry Point:  00000000
Automatic boot of image at addr 0x00400000 ...
## Booting image at 00400000 ...
   Image Name:   Linux-2.4.4
   Created:      2001-11-11  18:11:11 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    566824 Bytes = 553 kB = 0 MB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
Linux version 2.4.4 (wd@denx.denx.de) (gcc version 2.95.2 19991024 (release)) #1 Sun Nov 11 19:05
On node 0 totalpages: 8192
zone(0): 8192 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/hda3 ip=10.0.0.99:10.0.0.2::255.0.0.0:tqm::off panic=1
Decrementer Frequency: 3000000
Calibrating delay loop... 47.82 BogoMIPS
Memory: 30548k available (1088k kernel code, 488k data, 48k init, 0k highmem)
Dentry-cache hash table entries: 4096 (order: 3, 32768 bytes)
Buffer-cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 2048 (order: 2, 16384 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Starting kswapd v1.8
CPM UART driver version 0.03
ttyS0 on SMC1 at 0x0280, BRG1
ttyS1 on SMC2 at 0x0380, BRG2
pty: 256 Unix98 ptys configured
block: queued sectors max/low 20226kB/6742kB, 64 slots per queue
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
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
```

```
eth0: FEC ENET Version 0.2, FEC irq 3, MII irq 4, addr 00:cb:bd:00:00:11
JFFS version 1.0, (C) 1999, 2000  Axis Communications AB
 Amd/Fujitsu Extended Query Table v1.1 at 0x0040
number of JEDEC chips: 1
ICU862 flash bank 0: Using static image partition definition
Creating 8 MTD partitions on "ICU862 Bank 0":
0x00000000-0x00100000 : "kernel"
0x00100000-0x00400000 : "initrd"
0x00400000-0x00800000 : "jffs"
0x00800000-0x00c00000 : "cramfs"
0x00c00000-0x00f00000 : "jffs2"
0x00f00000-0x00f40000 : "ppcboot"
0x00f40000-0x00f80000 : "environment"
0x00f80000-0x01000000 : "spare"
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP, IGMP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 2048 bind 2048)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
 hda: hda1 hda2 hda3 hda4
 hda: hda1 hda2 hda3 hda4
VFS: Mounted root (ext2 filesystem) readonly.
Freeing unused kernel memory: 48k init
init started:  BusyBox v0.51 (2001.11.06-02:06+0000) multi-call binary

BusyBox v0.51 (2001.11.06-02:06+0000) Built-in shell (lash)
Enter 'help' for a list of built-in commands.

#
```

9.4. Adding Swap Space

If you are running out of system RAM, you can add virtual memory by using *swap space*. If you reserved a swap partition on your disk drive, you have to initialize it once using the `mkswap` command:

```
# fdisk -l /dev/hda

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
# mkswap /dev/hda3
Setting up swapspace version 1, size = 67604480 bytes
```

Then, to activate it, you use the `swapon` command like this:

```
# free
             total       used       free     shared    buffers     cached
Mem:         14628      14060        568       8056        100      11664
-/+ buffers/cache:       2296      12332
Swap:            0          0          0
# free
             total       used       free     shared    buffers     cached
Mem:         14628      14060        568       8056        100      11664
-/+ buffers/cache:       2296      12332
Swap:            0          0          0
# swapon /dev/hda3
Adding Swap: 66016k swap-space (priority -2)
# free
```

```
             total       used       free     shared    buffers     cached
Mem:         14628      14084        544       8056        100      11648
-/+ buffers/cache:       2336      12292
Swap:        66016          0      66016
```

If you forgot to reserve (sufficient) space in a separate partition on your disk, you can still use an ordinary file for swap space. You only have to create a file of appropriate size, and initialize it as follows:

```
# mount /dev/hda4 /mnt
# df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/root              2087212   1378824    708388  67% /
/dev/hda4               711352        20    675196   1% /mnt
# dd if=/dev/zero of=/mnt/swapfile bs=1024k count=64
64+0 records in
64+0 records out
# mkswap /mnt/swapfile
Setting up swapspace version 1, size = 67104768 bytes
```

Then activate it:

```
# free
             total       used       free     shared    buffers     cached
Mem:         14628      14084        544       6200         96      11788
-/+ buffers/cache:       2200      12428
Swap:            0          0          0
# swapon /mnt/swapfile
Adding Swap: 65528k swap-space (priority -3)
# free
             total       used       free     shared    buffers     cached
Mem:         14628      14084        544       6200         96      11752
-/+ buffers/cache:       2236      12392
Swap:        65528          0      65528
```

9.5. Splash Screen Support in Linux

To complement the U-Boot Splash Screen feature the new configuration option "`CONFIG_8xx_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...255 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.

9.6. 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 aready 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.

So our focus here is on the second item: the options we have for chosing 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.

9.6.1. 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 zxf /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/mtdchar     c  640  0       0       90       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/ptypf      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/ttyp       c  640  0       0        3      0       0       1       10
/dev/ttypa      c  640  0       0        3      10      -       -       -
/dev/ttypb      c  640  0       0        3      11      -       -       -
/dev/ttypc      c  640  0       0        3      12      -       -       -
/dev/ttypd      c  640  0       0        3      13      -       -       -
/dev/ttype      c  640  0       0        3      14      -       -       -
/dev/ttypf      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.

9.6.2. 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 unorderly 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 rebootet 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 unorderly shutdown
- 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 zxf /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}:${netmask}:${hostn
   => 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 1
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::
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% /
```

9.6.3. 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/dhcp /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) endianess
$ 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"`

9.6.4. 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 falsh 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 levelling which is needed for writable file systems in flash memory, nor is it robust against unorderly 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 -zxf /tmp/rootfs.tar.gz
```

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
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/dhcp /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
```

2. __EOD__
   chmod 0666 /tmpfs/dev/*
3. Like we did for the ramdisk, we 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=ext2.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. We can again use the same setup as before for the JFFS2 filesystem, just changing the bootargument to "`rootfstype=ext2`" (or simply omit it completely as this is the default anyway), and we must change the "`rw`" argument into "`ro`" to mount our root file system really read-only:

```
...
Linux version 2.4.25 (wd@xpert) (gcc version 3.3.3 (DENX ELDK 3.1.1 3.3.3-9)) #1 Sun Jun 1
On node 0 totalpages: 4096
zone(0): 4096 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/mtdblock6 ro rootfstype=ext2 ip=192.168.3.80:192.168.3.1::2
Decrementer Frequency = 187500000/60
Calibrating delay loop... 49.86 BogoMIPS
...
```

9.6.5. Root File System on a Flash Card

Using an `ext2` file system on a flash memory card (like CompactFlash, SD, MMC or a USB memory stick) is standard technology. To avoid unnecessary flash wear it is a good idea to mount the root file system read-only, or at least using the "`noatime`" mount option.

For our test we can use the "`ext2.img`" file from the previous step without changes:

1. In this test we use a standard CompactFlash card which comes with a single partition on it. We use U-Boot to copy the `ext2` file system image into this partition:

```
=> tftp 100000 /tftpboot/TQM860L/ext2.img
Using FEC ETHERNET device
TFTP from server 192.168.3.1; our IP address is 192.168.3.80
Filename '/tftpboot/TQM860L/ext2.img'.
Load address: 0x100000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
```

```
        ################################################################
        #########################
done
Bytes transferred = 3788800 (39d000 hex)
=> 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 1ce8

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.

2. We now prepare the Linux boot arguments to take this partition as read-only root device:

```
=> setenv cf_args setenv bootargs root=/dev/hda1 ro
=> setenv flash_cf 'run cf_args addip;bootm ${kernel_addr}'
=> setenv bootcmd run flash_cf
```

3. ...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 1
On node 0 totalpages: 4096
zone(0): 4096 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/hda1 ro ip=192.168.3.80:192.168.3.1::255.255.255.0:tqm860l:
Decrementer Frequency = 187500000/60
Calibrating delay loop... 49.86 BogoMIPS
...
```

9.6.6. 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 chosing a root file system for your embedded system exist.

The szenario 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 consistes 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:

1. The root file system image is easy: as mantioned 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.
2. 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, that 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
   ```
3. 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       -       -       -
   ```
4. 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.

5. 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 though 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 basicly 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 inital 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?

1. 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
```
2. 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
```
3. And finally we try it out:

```
U-Boot 1.1.3 (Jun 13 2005 - 02:24:00)

CPU:   XPC86xxxZPnnD4 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 1
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:tqm860l:eth1:off panic=
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 ...
```

9.7. 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. -

9.8. 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 modifed 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 *writeable*

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 desireable 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 developement 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/Know/MiniFOHome.

9.9. 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`.

9.9.1. 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.

9.9.2. 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

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
0xfffffffc in ?? ()
(gdb) b cpu_init_f
Breakpoint 1 at 0xfffd3310: 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>` - `CFG_MONITOR_LEN`. In our example with 16MB RAM and `CFG_MONITOR_LEN` = 192KB this yields the address `0x1000000` - `0x30000` = `0xFD0000`. 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
.this           00000060  cf030000  2**2
.text           000002f4  cf030060  2**2
.rodata         00000134  cf030354  2**2
.data           00000000  cf030488  2**0
.sdata          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_sect .rodata`\
 `add_sect .data`\
 `add_sect .sdata`\
 `add_sect .bss`\
 `add_sect .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:

12.1. Application Notes

A collection of Application Notes relevant for embedded computing can be found on the DENX web server.

12.2. Books

12.2.1. Linux kernel

- **Karim Yaghmour**: "Building Embedded Linux Systems", Paperback: 400 pages, O'Reilly & Associates; (May 2003); 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
- 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.
- **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
- **LWN: Porting device drivers to the 2.6 kernel** - Series of articles (37) in Linux Weekly News: http://lwn.net/Articles/driver-porting/

12.2.2. General Linux / Unix programming

- **W. Richard Stevens**: "Advanced Programming in the UNIX Environment", Addision Wesley, ISBN 0-201-56317-7

- **Eric S. Raymond**: "The Art of Unix Programming", Addision Wesley, ISBN 0131429019 => Read online
- **David R. Butenhof**: "Programming with POSIX Threads", Addision Wesley, ISBN 0-201-63392-2.
- **Bradford Nichols**, **Dick Buttlar** and **Jacqueline Proulx Farrell**: "Pthreads Programming", O'Reilly & Associates

12.2.3. Network Programming

- **W. Richard Stevens**: "TCP/IP Illustrated, Volume 1 - The Protocols", Addision Wesley, ISBN 0-201-63346-9
- **Gary R. Wright, W. Richard Stevens**: "TCP/IP Illustrated, Volume 2 - The Implementation", Addision Wesley, ISBN 0-201-63354-X
- **W. Richard Stevens**: "TCP/IP Illustrated, Volume 3 - TCP for Transactions", Addision 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

12.2.4. PowerPC Programming

- Introduction to Assembly on the PowerPC: http://www-106.ibm.com/developerworks/library/l-ppc/?t=gr,lnxw09=PowPC
- IBM PDF file (600+ page book) on PowerPC assembly language: http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2
- IBM PDF compiler writers guide on PPC asm tuning etc.: http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF7785256996007558C6
- A developer's guide to the POWER architecture: http://www-128.ibm.com/developerworks/linux/library/l-powarch/index.html
- PowerPC EABI Calling Sequence: ftp://sourceware.redhat.com/pub/binutils/ppc-docs/ppc-eabi-calling-sequence
- PowerPC Embedded Application Binary Interface (32-Bit Implementation): ftp://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

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.

- linuxppc-embedded - Communications among developers and users of Linux on embedded PowerPC boards
- linuxppc-dev - Communications among active developers of Linux on 32 bit PowerPC plattforms. Not intended for user support.
- linuxppc64-dev - Communications among active developers of Linux on 64 bit PowerPC plattforms. Not intended for user support.
- u-boot-users - Support for "U-Boot" Universal Bootloader
- u-boot-cvs - This mailing list tracks CVS commits. Not intended for discussions.

12.4. Links

Linux Kernel Resources:

- The Linux Documentation Project : http://www.tldp.org/
- Generic ("official") Linux Kernel sources: ftp://ftp.kernel.org/pub/linux/kernel/v2.4/
- Generic kernel sources for PowerPC systems: http://penguinppc.org/dev/kernel.shtml
- MIPS Linux Porting Guide: http://linux.junsun.net/porting-howto/porting-howto.html
- DENX kernel source trees: http://www.denx.de/re/linux.html
- Cross-Referencing the Linx Kernel: http://lxr.linux.no/source/?a=ppc
- Linux for PowerPC Embedded Systems HOWTO (old): http://penguinppc.org/embedded/howto/PowerPC-Embedded-HOWTO.html
- Linux for PowerPC Embedded Systems HOWTO (new): http://www.denx.de/twiki/bin/view/PPCEmbedded
- Linux Networking topics (like NAPI, GSO, VLAN, IPsec etc.): http://linux-net.osdl.org/index.php/Main_Page

RTAI:

- RTAI Home Page: http://www.rtai.org/
- DENX RTAI Patches: ftp://ftp.denx.de/pub/RTAI/

U-Boot:

- U-Boot Project Page: http://sourceforge.net/projects/u-boot
- 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

Programming:

- 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
- Position Independent Binaries: Ulrich Drepper: "Text Relocations"
- Shared Libraries: Drepper: "Good Practice in Library Design, Implementation, and Maintenance"
- More Ulrich Drepper stuff: http://people.redhat.com/drepper/

Standards:

- Linux Standard Base: http://refspecs.freestandards.org/lsb.shtml
- Single UNIX Specification, Version 2
- PCI Bus Bindings - Standard for Boot Firmware: http://playground.sun.com/1275/bindings/pci/pci2_1.pdf

12.5. More Links

- Starting point for Linux based asm (mostly x86): http://linuxassembly.org/
- Andries Brouwers remarks to the linux kernel: http://www.win.tue.nl/~aeb/linux/lk/lk.html
- A quite complete history of the UNIX family can be found here: http://www.levenez.com/unix/
- Unix Manual, first edition, 3 November 1971
- 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 disassember, and "SmartDMA Hand-Assembly Guides" document.

12.6. 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 automagically omits "unwanted" files like CVS repository data, etc.)

13. Appendix

13.1. BDI2000 Configuration file

```
; bdiGDB configuration file for TQM8xxL Module
; --------------------------------------------
;
[INIT]
; init core register
WREG    MSR             0x00001002      ;MSR  : ME,RI
WSPR    27              0x00001002      ;SRR1 : ME,RI
WSPR    149             0x2002000F      ;DER  : set debug enable register
;;WSPR  149             0x2006000F      ;DER  : enable SYSIE for BDI flash progr.
WSPR    638             0xFFF00000      ;IMMR : internal memory at 0xFFF00000
WSPR    158             0x00000007      ;ICTRL:

; init SIU register
;;;WM32 0xFFF00000      0x00610400      ;SIUMCR
WM32    0xFFF00000      0x00010400      ;SIUMCR - for use with PCMCIA
WM32    0xFFF00004      0xFFFFFF89      ;SYPCR

WSPR    796             0x00000000      ;M_TWB: invalidate TWB

[TARGET]
MMU         XLAT        ; support virtual addresses (for Linux!)
PTBASE      0x000000F0  ; ptr to page table pointers
CPUCLOCK    45000000    ;the CPU clock rate after processing the init list
BDIMODE     AGENT       ;the BDI working mode (LOADONLY | AGENT)
BREAKMODE   HARD        ;SOFT or HARD, HARD uses PPC hardware breakpoints

[HOST]
IP          192.168.3.1
```

```
FILE        /tftpboot/TQM8xxL/u-boot.bin
FORMAT      BIN
LOAD        MANUAL      ;load code MANUAL or AUTO after reset
DEBUGPORT   2001
START       0x0100

[FLASH]
CHIPTYPE    AM29BX16    ;Flash type (AM29LV160B)
CHIPSIZE    0x200000    ;The size of one flash chip in bytes
BUSWIDTH    32          ;The width of the flash memory bus in bits (8 | 16 | 32)
WORKSPACE   0xFFF02000          ; RAM buffer for fast flash programming
FILE        /tftpboot/TQM8xxL/u-boot.bin        ;The file to program
FORMAT      BIN  0x00000000
ERASE       0x00000000 BLOCK
ERASE       0x00008000 BLOCK
ERASE       0x0000C000 BLOCK
ERASE       0x00010000 BLOCK
ERASE       0x00020000 BLOCK

[REGS]
DMM1    0xFFF00000
FILE    /tftpboot/reg860.def
```

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 UBoot 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 3.4.3. Initial Installation
4. Set envrionment 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:

   · GNU `make` has to be used.
   · 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}.`
   · 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
```

14.1.2. ELDK 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.

14.1.3. 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 partititons:

```
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 RCmountfs
-#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
```

14.1.4. 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.

14.2. U-Boot

14.2.1. Can UBoot 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.

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 to your board, burn the image to flash, and debug it in it's 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 adaptions 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. 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.

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.

14.2.4. 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 envrionment variables using the **`saveenv`** command.

14.2.5. 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.

14.2.6. Linux hangs after uncompressing the kernel

Question:

I am using U-Boot with a Linux kernel version `Y` (`Y` < 2.4.5-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 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 corrct 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.

14.2.7. 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).

14.2.8. Ethernet Does Not Work

Question:
: Ethernet does not work on my board. 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.

14.2.9. 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 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.

14.2.10. 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 ###########
       ###########
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 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 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).

14.2.11. 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:

14.2.11.1. Old, simple command line parser

- supports environment variables (through `setenv` / `saveenv` commands)
- several commands on one line, separated by `';'`
- variable substitution using `"... ${_variablename_} ..."` syntax
  NOTE: Older versions of U-Boot used `"$(...)"` 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}:${
  ```

14.2.11.2. 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:$netmask:$hostnam
```

14.2.11.3. 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.

14.2.11.4. 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.

14.2.12. 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`...

14.2.13. 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 00000000b000
```

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.

14.2.14. 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`.

14.2.15. My standalone program does not work

Question:

I tried adding some new code to the `hellow_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 calles 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, but at 0x40058. Just start your standalone program at this address, and everything should work well.

14.2.16. 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 figuring out where. Device handling and board-specific code is the place to start.

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 PowerPC processor). Why?

Answer:
: Quite likely your SDRAM initialization is bad. See UBootCrashAfterRelocation for more information.

  On a PowerPC, the instructions beginning with 0xFF are floating point instructions. When your memory subsystem fails, the PowerPC 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 PowerPC 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 PowerPC 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 PowerPC 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.

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 3.6. Mounting Target Components via NFS.

14.3.9. 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 deamon is running, which is causing your problems. There are two solutions for the problem:

1. Add the portmap deamon (`/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`" deamon 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.

14.3.10. 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

14.3.11. 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.

14.3.12. 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.

14.3.13. 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

14.3.14. 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 **Bogo**MIPS.

On PowerPC, 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.

14.3.15. Linux Kernel crashes when using a ramdisk image

Question:
: I have a PowerPC 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
```

14.3.16. 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".)

14.3.17. 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.

14.3.18. Adding Files to Ramdisk is Non Persistent

Quetsion:

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.

14.3.19. 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 bootup 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).

14.3.20. 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
```

14.3.21. 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 PowerPC systems, there are two commonly used options:

14.3.21.1. Using a MacOS Partition Table

A MacOS partition table is the "native" partition table format on PowerPC systems; most desktop PowerPC systems use it, so you may prefer it when you have PowerPC 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 chosing a partition size of 2 MB is more than generous. 2 MB coresponds 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*1024*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
```

14.3.21.2. 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
```

14.3.22. 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>  := <mtd-id>:<partdef>[,<partdef>]
<partdef> := <size>[@offset][<name>][ro]
<mtd-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)
```

14.3.23. 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.

14.3.24. 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.)

14.3.24.1. 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.

14.3.24.2. 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 "+t" 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.

14.3.24.3. 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 PowerPC 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 availible XIP extension sources and highlighting how to put all the pieces together.

14.3.24.4. 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 1.1MB + 1.5M = 2.6 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

14.3.25. 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).

14.3.26. 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
```

14.3.27. 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`.

14.3.28. 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*.

14.4. 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 eaiser 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 -b
```

- 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 -VBpdum /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" menue), 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://h623653.serverkompetenz.net/wiki/pub/DULG/ConflictsWithAsmClobberList/patch

14.6. BDI2000

14.6.1. Where can I find BDI2000 Configuration Files?

A collection of configuration files for the BDI2000 BDM/JTAG debugger by Abatron can be found at ftp://ftp.denx.de/pub/BDI2000/

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.

14.7. 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`

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.

14.8. TQM Boards

14.8.1. Using a PCMCIA WLAN Card with a TQM8xxL Board

Question:
: What is needed to get a PCMCIA WLAN card running on a TQM8xxL system?

Answer:
: You need ELDK version 2.0.2 or later; this includes (1) the Linux kernel source with the required extensions, the PCMCIA Card Service package with extensions for MPC8xx systems, and the wireless tools package to control the PCMCIA devices.

  To bring up the WLAN card for network operations, the following actions should be performed (the example output shows card configuration for a WLAN network controlled by the Access Point ("managed" mode):

  1. Starting CardServices on the target:

     ```
     bash# /etc/rc.d/init.d/pcmcia start
     ```
  2. Assign the IP address of the WLAN network segment to the WLAN interface:

     ```
     bash# ifconfig eth1 192.168.2.3
     ```
  3. Assign the Network (or Domain) Name to the WLAN interface:

     ```
     bash# iwconfig eth1 essid "DENX"
     ```
  4. At this point the Acess Point station MAC address should appear on the `iwconfig` output:

     ```
     bash# iwconfig eth1
     eth1      IEEE 802.11-DS  ESSID:"DENX"  Nickname:"Prism  I"
               Mode:Managed  Frequency:2.462GHz  Access Point: 00:02:2D:03:A5:15
               Bit Rate:2Mb/s   Tx-Power=15 dBm   Sensitivity:1/3
               Retry min limit:8   RTS thr:off   Fragment thr:off
               Encryption key:off
               Power Management:off
               Link Quality:28/92  Signal level:151/153  Noise level:107/153
               Rx invalid nwid:0  invalid crypt:0  invalid misc:0
     ```

  The card is now ready for normal network operations.

14.8.2. Ethernet Problems on TQM8xxL boards

Question:
: I am using a TQM8xxL module on a STK8xxL Starter Kit board. Everything is fine, but Ethernet does not work - neither in U-Boot nor in Linux.

Answer:
: The TQM855L/M, TQM860L/M and TQM862L/M modules use SCC1 for the Ethernet interface. Make sure that jumpers are set on connectors labeled X.12, X.13 and X.14 on the STK8xxL board on the positions 1-3 and 2-4; also make sure to remove the jumpers from positions 7-8, 9-10 and 11-12 on X.30.

  For the TQM823L and TQM850L modules SCC2 is used for Ethernet. Here jumpers must be set on connectors X.12, X.13 and X.14 on the positions 3-5 and 4-6; X.30 is used for USB configuration on these boards - if you don't use USB it's safe to remove the jumpers from positions 7-8, 9-10 and 11-12 on X.30.

15. Glossary

ABI

- Application Binary Interface

The convention for register usage and C linkage commonly used on desktop PowerPC 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 eliminationg 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 PowerPC core itself, or the physical processor device (including CPM, SIU, packaging etc) as a single unit.

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.). Sucessor 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 PowerPC 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 eliminationg 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-independend 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 MII.

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 resoucres 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 PowerPC systems can be found at http://www.yellowdoglinux.com.