OSELAS.Support
OSELAS.Training
OSELAS.Development
OSELAS.Services

Quickstart Manual
OSELAS.BSP( )
FriendlyARM Mini2440

Pengutronix e. K.
Peiner Straße 6–8
31137 Hildesheim

+49 (0)51 21 / 20 69 17 – 0 (Fon)
+49 (0)51 21 / 20 69 17 – 55 55 (Fax)

info@pengutronix.de

Cut Here and Stick on your Monitor

Don't Panic

Contents

Part I

OSELAS Quickstart for FriendlyARM Mini2440

1 You have been warned

The Barebox bootloader and the Linux kernel contained in this board support package will modify your NAND memory. This is important to know if you want to keep a way back to the previous usage. At least the bad block marker maybe lost if you try to switch back to the old behaviour.

If you already used another recent kernel on your Mini2440, you can ignore this warning.

A word about using NAND memory for the bootloader and the filesystem:

NAND memory can be forgetful. That is why some kind of redundancy information is always required. This board support package uses ECC (error-correcting code) checksums as redundancy information when the bootloader and the Linux kernel are up and running.

This kind of redundancy information can repair one bit errors and detect two bit errors in a page of data. Its very important to use ECC at least for the bootloader to ensure to bring up the Mini2440 successfully. But its currently not done in the bootloader while bootstrapping. So, there is still a risk for long term use to fail booting the Mini2440 from NAND. In this case the bootloader must be re-written making the Mini2440 booting again from NAND.

In one of the next releases, ECC check and correction will be done while bootstrapping as well, to make the system more reliable for long term use. But then one question will be still open: Does the hardware of the S3C2440 CPU ECC check and correction for the very first page? I guess no, because the hardware has no idea, where the ECC checksum is stored. So, maybe there is no 100 % reliable solution for long term users.

2 First steps with PTXdist

In the next sections you will work with PTXdist to get everything you need to get your Mini2440 up and working. To give you a quick idea what PTXdist is, you should read this section.

PTXdist works as a console command tool. Everything we want PTXdist to do, we have to enter as a command. But it's always the same base command:

```
$ ptxdist <parameter>
```

To run different functions, this command must be extended by parameters to define the function we want to run.

If we are unsure what parameter must be given to obtain a special function, we run it with the parameter *help*.

```
$ ptxdist help
```

This will output all possible parameters and subcommands and their meaning.

As the list we see is very long, let's explain the major parameters usually needed for daily usage:

menu This starts a dialog based frontend for those who do not like typing commands. It will gain us access to the most common parameters to configure and build a PTXdist project. Note: it needs 'dialog' to be installed to make is work. It will fail if this tool is not installed on your host. `menuconfig` can be used instead in this case.

menuconfig Starts the Kconfig based project configurator for the current selected userland configuration. This menu will give us access to various userland components the root filesystem of our target should consist of.

platformconfig Starts the Kconfig based platform configurator. This menu lets us set up all target specific settings. Major parts are:

- Toolchain (architecture and revision)
- boot loader
- root filesystem image type
- Linux kernel (revision)

Note: A PTXdist project can consist of more than one platform configuration at the same time.

kernelconfig Runs the standard Linux kernel Kconfig to configure the kernel for the current selected platform. To run this feature, the kernel must be already set up for this platform.

menuconfig barebox Runs the standard Barebox's Kconfig to configure the bootloader. To run this feature, Barebox must be already set up for this platform.

toolchain Sets up the path to the toolchain used to compile the current selected platform. Without an additional parameter, PTXdist tries to guess the toolchain from platform settings. To be successful, PTXdist depends on the OSELAS.Toolchains installed to the `/opt` directory.
If PTXdist wasn't able to autodetect the toolchain, an additional parameter can be given to provide the path to the compiler, assembler, linker and so on.

select Used to select the current userland configuration, which is only required if there is no `selected_ptxconfig` in the project's main directory. This parameter needs the path to a valid `ptxconfig`. It will generate a soft link called `selected_ptxconfig` in the project's main directory.

platform Used to select the current platform configuration, which is only required if there is no `selected_platformconfig` in the project's main directory. This parameter needs the path to a valid `platformconfig`. It will generate a soft link called `selected_platformconfig` in the project's main directory.

go The mostly used command. This will start to build everything to get all the project defined software parts. Also used to rebuild a part after its configuration was changed.

images Used at the end of a build to create an image from all userland packages to deploy the target (its flash for example or its hard disk).

setup Mostly run once per PTXdist revision to set up global paths and the PTXdist behavior.

All these commands depending on various files a PTXdist based project provides. So, running the commands make only sense in directories that contain a PTXdist based project. Otherwise, PTXdist gets confused and then it tries to confuse the user with funny error messages.

3 Getting a working Environment

3.1 Download Software Components

In order to follow this manual, some software archives are needed. There are several possibilities how to get these: either as part of an evaluation board package or by downloading them from the Pengutronix web site.

The central place for OSELAS related documentation is http://www.oselas.com. This website provides all required packages and documentation (at least for software components which are available to the public).

To build OSELAS.BSP-Pengutronix-Mini2440-2012.06.0, the following archives have to be available on the development host:

- ptxdist-2012.06.0.tar.bz2
- OSELAS.BSP-Pengutronix-Mini2440-2012.06.0.tar.gz
- ptxdist-2011.11.0.tar.bz2
- OSELAS.Toolchain-2011.11.1.tar.bz2

If they are not available on the development system yet, it is necessary to get them.

The PTXdist- 2011.11.0 is only required to build the toolchain. All PTXdist revisions can co-exist.

3.2 PTXdist Installation

The PTXdist build system can be used to create a root filesystem for embedded Linux devices. In order to start development with PTXdist it is necessary to install the software on the development system.

This chapter provides information about how to install and configure PTXdist on the development host.

3.2.1 Main Parts of PTXdist

The most important software component which is necessary to build an OSELAS.BSP( ) board support package is the `ptxdist` tool. So before starting any work we'll have to install PTXdist on the development host.

PTXdist consists of the following parts:

The `ptxdist` Program: `ptxdist` is installed on the development host during the installation process. `ptxdist` is called to trigger any action, like building a software packet, cleaning up the tree etc. Usually the `ptxdist` program is used in a *workspace* directory, which contains all project relevant files.

A Configuration System: The config system is used to customize a *configuration*, which contains information about which packages have to be built and which options are selected.

Patches: Due to the fact that some upstream packages are not bug free – especially with regard to cross compilation – it is often necessary to patch the original software. PTXdist contains a mechanism to automatically apply patches to packages. The patches are bundled into a separate archive. Nevertheless, they are necessary to build a working system.

Package Descriptions: For each software component there is a "recipe" file, specifying which actions have to be done to prepare and compile the software. Additionally, packages contain their configuration sniplet for the config system.

Toolchains: PTXdist does not come with a pre-built binary toolchain. Nevertheless, PTXdist itself is able to build toolchains, which are provided by the OSELAS.Toolchain() project. More in-deep information about the OSELAS.Toolchain() project can be found here: http://www.pengutronix.de/oselas/toolchain/index_en.html

Board Support Package This is an optional component, mostly shipped aside with a piece of hardware. There are various BSP available, some are generic, some are intended for a specific hardware.

3.2.2 Extracting the Sources

Do the following steps at best in your own home directory ($HOME). You need root permissions only in the `make install` step, and **nowhere** else.

To install PTXdist, the archive Pengutronix provides has to be extracted:

ptxdist-2012.06.0.tar.bz2 The PTXdist software itself

The PTXdist packet has to be extracted into some temporary directory in order to be built before the installation, for example the `local/` directory in the user's home. If this directory does not exist, we have to create it and change into it:

```
$ cd
$ mkdir local
$ cd local
```

Next step is to extract the archive:

```
$ tar -xjf ptxdist-2012.06.0.tar.bz2
```

If everything goes well, we now have a PTXdist-2012.06.0 directory, so we can change into it:

```
$ cd ptxdist-2012.06.0
$ ls -lF
total 531
-rw-r--r--   1 jb user  18361 Jun 11 18:09 COPYING
-rw-r--r--   1 jb user   4030 Jun 11 18:09 CREDITS
-rw-r--r--   1 jb user 115540 Jun 11 18:09 ChangeLog
-rw-r--r--   1 jb user     57 Jun 11 18:09 INSTALL
-rw-r--r--   1 jb user   3505 Jun 11 18:09 Makefile.in
-rw-r--r--   1 jb user   4268 Jun 11 18:09 README
```

```
-rw-r--r--   1 jb user  63516 Jun 11 18:09 TODO
-rwxr-xr-x   1 jb user     28 Jun 11 18:09 autogen.sh*
drwxr-xr-x   2 jb user     72 Jun 11 18:09 bin
drwxr-xr-x  10 jb user    296 Jun 11 18:09 config
-rwxr-xr-x   1 jb user 230810 Jun 12 09:44 configure*
-rw-r--r--   1 jb user  13319 Jun 11 18:09 configure.ac
drwxr-xr-x  10 jb user    248 Jun 11 18:09 generic
drwxr-xr-x 236 jb user   7968 Jun 11 18:09 patches
drwxr-xr-x   2 jb user   1376 Jun 11 18:09 platforms
drwxr-xr-x   4 jb user    112 Jun 11 18:09 plugins
drwxr-xr-x   6 jb user  55008 Jun 11 18:09 rules
drwxr-xr-x   8 jb user    912 Jun 11 18:09 scripts
drwxr-xr-x   2 jb user    512 Jun 11 18:09 tests
```

3.2.3 Prerequisites

Before PTXdist can be installed it has to be checked if all necessary programs are installed on the development host. The configure script will stop if it discovers that something is missing.

The PTXdist installation is based on GNU autotools, so the first thing to be done now is to configure the packet:

```
$ ./configure
```

This will check your system for required components PTXdist relies on. If all required components are found the output ends with:

```
[...]
checking whether /usr/bin/patch will work... yes

configure: creating ./config.status
config.status: creating Makefile
config.status: creating scripts/ptxdist_version.sh
config.status: creating rules/ptxdist-version.in

ptxdist version 2012.06.0 configured.
Using '/usr/local' for installation prefix.

Report bugs to ptxdist@pengutronix.de
```

Without further arguments PTXdist is configured to be installed into `/usr/local`, which is the standard location for user installed programs. To change the installation path to anything non-standard, we use the `--prefix` argument to the `configure` script. The `--help` option offers more information about what else can be changed for the installation process.

The installation paths are configured in a way that several PTXdist versions can be installed in parallel. So if an old version of PTXdist is already installed there is no need to remove it.

One of the most important tasks for the `configure` script is to find out if all the programs PTXdist depends on are already present on the development host. The script will stop with an error message in case something is missing. If this happens, the missing tools have to be installed from the distribution befor re-running the `configure` script.

When the `configure` script is finished successfully, we can now run

```
$ make
```

All program parts are being compiled, and if there are no errors we can now install PTXdist into it's final location. In order to write to `/usr/local`, this step has to be performed as user *root*:

```
$ sudo make install
[enter password]
[...]
```

If we don't have root access to the machine it is also possible to install PTXdist into some other directory with the `--prefix` option. We need to take care that the `bin/` directory below the new installation dir is added to our `$PATH` environment variable (for example by exporting it in `~/.bashrc`).

The installation is now done, so the temporary folder may now be removed:

```
$ cd ../../
$ rm -fr local
```

3.2.4 Configuring PTXdist

When using PTXdist for the first time, some setup properties have to be configured. Two settings are the most important ones: Where to store the source archives and if a proxy must be used to gain access to the world wide web.

Run PTXdist's setup:

```
$ ptxdist setup
```

Due to PTXdist is working with sources only, it needs various source archives from the world wide web. If these archives are not present on our host, PTXdist starts the `wget` command to download them on demand.

Proxy Setup

To do so, an internet access is required. If this access is managed by a proxy `wget` command must be adviced to use it. PTXdist can be configured to advice the `wget` command automatically: Navigate to entry *Proxies* and enter the required addresses and ports to access the proxy in the form:

`<protocol>://<address>:<port>`

Source Archive Location

Whenever PTXdist downloads source archives it stores these archives in a project local manner. This is the default behaviour. If we are working with more than one PTXdist based project, every project would download its own required archives in this case. To share all source archives between all projects, PTXdist can be configured to share only one archive directory for all projects it handles: Navigate to menu entry *Source Directory* and enter the path to the directory where PTXdist should store archives to share between its projects.

Generic Project Location

If we already installed the generic projects we should also configure PTXdist to know this location. If we already did so, we can use the command `ptxdist projects` to get a list of available projects and `ptxdist clone` to get a local working copy of a shared generic project.

Navigate to menu entry *Project Searchpath* and enter the path to projects that can be used in such a way. Here we can configure more than one path, each part can be delemited by a colon. For example for PTXdist's generic projects and our own previous projects like this:

```
/usr/local/lib/ptxdist-2012.06.0/projects:/office/my_projects/ptxdist
```

Leave the menu and store the configuration. PTXdist is now ready for use.

3.3 Toolchains

Before we can start building our first userland we need a cross toolchain. On Linux, toolchains are no monolithic beasts. Most parts of what we need to cross compile code for the embedded target comes from the *GNU Compiler Collection*, gcc. The gcc packet includes the compiler frontend, gcc, plus several backend tools (cc1, g++, ld etc.) which actually perform the different stages of the compile process. gcc does not contain the assembler, so we also need the *GNU Binutils package* which provides lowlevel stuff.

Cross compilers and tools are usually named like the corresponding host tool, but with a prefix – the *GNU target*. For example, the cross compilers for ARM and powerpc may look like

- arm-softfloat-linux-gnu-gcc
- powerpc-unknown-linux-gnu-gcc

With these compiler frontends we can convert e.g. a C program into binary code for specific machines. So for example if a C program is to be compiled natively, it works like this:

```
$ gcc test.c -o test
```

To build the same binary for the ARM architecture we have to use the cross compiler instead of the native one:

```
$ arm-softfloat-linux-gnu-gcc test.c -o test
```

Also part of what we consider to be the "toolchain" is the runtime library (libc, dynamic linker). All programs running on the embedded system are linked against the libc, which also offers the interface from user space functions to the kernel.

The compiler and libc are very tightly coupled components: the second stage compiler, which is used to build normal user space code, is being built against the libc itself. For example, if the target does not contain a hardware floating point unit, but the toolchain generates floating point code, it will fail. This is also the case when the toolchain builds code for i686 CPUs, whereas the target is i586.

So in order to make things working consistently it is necessary that the runtime libc is identical with the libc the compiler was built against.

PTXdist doesn't contain a pre-built binary toolchain. Remember that it's not a distribution but a development tool. But it can be used to build a toolchain for our target. Building the toolchain usually has only to be done once. It may be a good idea to do that over night, because it may take several hours, depending on the target architecture and development host power.

3.3.1 Using Existing Toolchains

If a toolchain is already installed which is known to be working, the toolchain building step with PTXdist may be omitted.

The OSELAS.BoardSupport() Packages shipped for PTXdist have been tested with the OSELAS.Toolchains() built with the same PTXdist version. So if an external toolchain is being used which isn't known to be stable, a target may fail. Note that not all compiler versions and combinations work properly in a cross environment.

Every OSELAS.BoardSupport() Package checks for its OSELAS.Toolchain it's tested against, so using a different toolchain vendor requires an additional step:

Open the OSELAS.BoardSupport() Package menu with:

```
$ ptxdist platformconfig
```

and navigate to `architecture ---> toolchain` and `check for specific toolchain vendor`. Clear this entry to disable the toolchain vendor check.

Preconditions an external toolchain must meet:

- it shall be built with the configure option `--with-sysroot` pointing to its own C libraries.
- it should not support the *multilib* feature as this may confuse PTXdist which libraries are to select for the root filesystem

If we want to check if our toolchain was built with the `--with-sysroot` option, we just run this simple command:

```
$ mytoolchain-gcc -v 2>&1  grep with-sysroot
```

If this command **does not** output anything, this toolchain was not built with the `--with-sysroot` option and cannot be used with PTXdist.

3.3.2 Building a Toolchain

PTXdist handles toolchain building as a simple project, like all other projects, too. So we can download the OSELAS.Toolchain bundle and build the required toolchain for the OSELAS.BoardSupport() Package.

Building any toolchain of the OSELAS.Toolchain-2011.11.1 is tested with PTXdist-2011.11.0. Pengutronix recommends to use this specific PTXdist to build the toolchain. So, it might be essential to install more than one PTXdist revision to build the toolchain and later on the Board Support Package if the latter one is made for a different PTXdist revision.

A PTXdist project generally allows to build into some project defined directory; all OSELAS.Toolchain projects that come with PTXdist are configured to use the standard installation paths mentioned below.

All OSELAS.Toolchain projects install their result into `/opt/OSELAS.Toolchain-2011.11.1/`.

Usually the `/opt` directory is not world writeable. So in order to build our OSELAS.Toolchain into that directory we need to use a root account to change the permissions. PTXdist detects this case and asks if we want to run `sudo` to do the job for us. Alternatively we can enter:
`mkdir /opt/OSELAS.Toolchain-2011.11.1`
`chown <username> /opt/OSELAS.Toolchain-2011.11.1`
`chmod a+rwx /opt/OSELAS.Toolchain-2011.11.1`.

We recommend to keep this installation path as PTXdist expects the toolchains at `/opt`. Whenever we go to select a platform in a project, PTXdist tries to find the right toolchain from data read from the platform configuration settings and a toolchain at `/opt` that matches to these settings. But that's for our convenience only. If we decide to install the toolchains at a different location, we still can use the *toolchain* parameter to define the toolchain to be used on a per project base.

3.3.3 Building the OSELAS.Toolchain for OSELAS.BSP-Pengutronix-Mini2440-2012.06.0

Do the following steps in your own home directory ($HOME). The final OSELAS.Toolchain gets installed to `opt/`, but must **never** be compiled in the **opt/** directory. You will get many funny error messages, if you try to compile the OSELAS-Toolchain in **opt/**.

To compile and install an OSELAS.Toolchain we have to extract the OSELAS.Toolchain archive, change into the new folder, configure the compiler in question and start the build.

The required compiler to build the OSELAS.BSP-Pengutronix-Mini2440-2012.06.0 board support package is

`arm-v4t-linux-gnueabi_gcc-4.6.2_glibc-2.14.1_binutils-2.21.1a_kernel-2.6.39-sanitized`

So the steps to build this toolchain are:

In order to build any of the OSELAS.Toolchains, the host must provide the tool *fakeroot*. Otherwise the message `bash: fakeroot: command not found` will occur and the build stops.

Please ensure the 'current directory' (the `.` entry) is not part of your PATH environment variable. PTXdist tries to sort out this entry, but might not be successfully doing so. Check by running `ptxdist print PATH` if the output still contains any kind of 'current directory' as a component. If yes, remove it first.

```
$ tar xf OSELAS.Toolchain-2011.11.1.tar.bz2
$ cd OSELAS.Toolchain-2011.11.1
$ ptxdist select ptxconfigs/↵
        arm-v4t-linux-gnueabi_gcc-4.6.2_glibc-2.14.1_binutils-2.21.1a_kernel-2.6.39-sanitized.ptxconfig
$ ptxdist go
```

At this stage we have to go to our boss and tell him that it's probably time to go home for the day. Even on reasonably fast machines the time to build an OSELAS.Toolchain is something like around 30 minutes up to a few hours.

Measured times on different machines:

- Single Pentium 2.5 GHz, 2 GiB RAM: about 2 hours
- Turion ML-34, 2 GiB RAM: about 1 hour 30 minutes
- Dual Athlon 2.1 GHz, 2 GiB RAM: about 1 hour 20 minutes
- Dual Quad-Core-Pentium 1.8 GHz, 8 GiB RAM: about 25 minutes

- 24 Xeon cores 2.54 GHz, 96 GiB RAM: about 22 minutes

Another possibility is to read the next chapters of this manual, to find out how to start a new project.

When the OSELAS.Toolchain project build is finished, PTXdist is ready for prime time and we can continue with our first project.

3.3.4 Protecting the Toolchain

All toolchain components are built with regular user permissions. In order to avoid accidential changes in the toolchain, the files should be set to read-only permissions after the installation has finished successfully. It is also possible to set the file ownership to root. This is an important step for reliability, so it is highly recommended.

3.3.5 Building Additional Toolchains

The OSELAS.Toolchain-2011.11.1 bundle comes with various predefined toolchains. Refer the `ptxconfigs/` folder for other definitions. To build additional toolchains we only have to clean our current toolchain project, removing the current `selected_ptxconfig` link and creating a new one.

```
$ ptxdist clean
$ rm selected_ptxconfig
$ ptxdist select ptxconfigs/any_other_toolchain_def.ptxconfig
$ ptxdist go
```

All toolchains will be installed side by side architecture dependent into directory

`/opt/OSELAS.Toolchain-2011.11.1/architecture_part`.

Different toolchains for the same architecture will be installed side by side version dependent into directory

`/opt/OSELAS.Toolchain-2011.11.1/architecture_part/version_part`.

4 Building a root filesystem for the Mini2440

4.1 Extracting the Board Support Package

In order to work with a PTXdist based project we have to extract the archive first.

```
$ tar -zxf OSELAS.BSP-Pengutronix-Mini2440-2012.06.0.tar.gz
$ cd OSELAS.BSP-Pengutronix-Mini2440-2012.06.0
```

PTXdist is project centric, so now after changing into the new directory we have access to all valid components.

```
total 68
-rw-r--r-- 1 jb users  5378 Jun 24 22:19 Changelog
drwxr-xr-x 3 jb users  4096 Jun 24 22:19 configs
-rw-r--r-- 1 jb users   408 Dec 23  2011 CONTRIBUTORS
-rw-r--r-- 1 jb users 18002 Dec 23  2011 COPYING
drwxr-xr-x 3 jb users  4096 Oct 21  2011 documentation
-rw-r--r-- 1 jb users  4350 Jun 24 22:19 FAQ
drwxr-xr-x 4 jb users  4096 Dec 23  2011 local_src
drwxr-xr-x 3 jb users  4096 Oct 21  2011 projectroot
drwxr-xr-x 2 jb users  4096 Jun 24 22:19 protocol
-rw-r--r-- 1 jb users   177 Oct 21  2011 README
drwxr-xr-x 2 jb users  4096 Jun 24 22:19 rules
```

Notes about some of the files and directories listed above:

ChangeLog Here you can read what has changed in this release.

FAQ Some questions and some answers

documentation This directory contains the Quickstart you are currenly reading in.

configs This directory contains the platform specific configuration files for the Mini2440.

projectroot Contains files and configuration for the target's runtime. A running GNU/Linux system uses many text files for runtime configuration. Most of the time the generic files from the PTXdist installation will fit the needs. But if not, customized files are located in this directory.

local_src Application sources especially related to the Mini2440.

rules If something special is required to build the BSP for the target it is intended for, then this directory contains these additional rules.

patches If some special patches are required to build the BSP for this target, then this directory contains these patches on a per package basis.

protocol Contains the test protocol made for the release. Also known open issues if any.

4.2 Feature Dependend Configurations

The FriendlyARM Mini2440 comes in various incarnations. Mostly they differ in the NAND memory size, but also other features may be present or not. Read the following sub-sections to adapt this board support package to meet exactly your Mini2440 requirements.

Note: In this documentation the FriendlyARM Mini2440 with 64 MiB of NAND memory is the reference platform. However, everything mentioned herein is also valid for Mini2440s shipped with more than 64 MiB of NAND.

4.2.1 Identify Your Mini2440

The FriendlyARM Mini2440s are shipped with various NAND memory sizes. The smallest is a 64 MiB unit, the largest one comes with 1 GiB of NAND memory.

As this kind of memory needs some special treatment depending on its internal layout, we must distinguish between them prior to generating any images. This board support package comes with two configurations:

- `platformconfig-NAND-64M` for the Mini2440 with 64 MiB of NAND memory
  - the NAND device is marked with the text `K9F1208`
- `platformconfig-NAND-128M` for the Mini2440 with 128 MiB of NAND memory or more
  - these NAND devices are marked with the text `K9F1G08`, `K9F2G08` or `K9K8G08`.

This is important while performing the platform selection step in section 4.4. As this section references the 64 MiB NAND configuration (`platformconfig-NAND-64M`), we must select the 128 MiB configuration (`platformconfig-NAND-128M`) instead, if we are using a NAND memory larger than 64 MiB.

Running a 64 MiB configuration on a 128 MiB (or above) Mini2440 will give us many confusing error messages (the same the other way around).

What differs in both configurations:

- erase block size (16 kiB versus 128 kiB)
- JFFS2 root filesystem creation (needs different parameters)
- count of spare blocks (important for NAND memory usage)
- partition sizes (due to different spare block counts)

4.2.2 Network Adaptions

The default network configurations for the bootloader and the Linux kernel are located in different files in this board support package. These files must be changed in order to meet the local network requirements, to enable the bootloader and the Linux kernel to communicate via network.

The network configuration can still be changed later on when the Mini2440 is up and running. Changing it prior the build is more for convenience.

4.2.2.1 Bootloader Barebox

As there is no generic network setting available, some changes to our own network should be done prior building the board support package.

To do so, we should open one of the following files with our favourite editor:

- `configs/platform-friendlyarm-mini2440/barebox-64m-env/config` if we are using a Mini2440 with 64 MiB NAND
- `configs/platform-friendlyarm-mini2440/barebox-128m-env/config` if we are using a Mini2440 with 128 MiB or larger NAND

These settings are relevant only for the bootloader. The file content will be the default settings later on, when we are using the Mini2440. Default settings mean they can be permanently changed at run-time later on. But, whenever the bootloader loses its environment it will fall back to the settings in this file. So, to avoid making more changes at run-time than required, we should do the settings carefully here.

We can keep the `ip=dhcp` option enabled. This requires a DHCP server in the network to be able to update the NAND memory content or to use NFS root filesystem while developing our application. In this case at least the `eth0.ethaddr` must be set, to give our network device a unique MAC address.

If no DHCP server is available, a static network setting can be used instead. We just comment out the `ip=dhcp` option and enable all the `eth0.*` lines and give them appropriate values.

If we want to use an NFS based root filesystem, we also should adapt the `nfsroot` setting.

Don't forget the setting of a unique MAC address. At least the entry `eth0.ethaddr` must be set.

4.2.2.2 Linux Kernel

To define network settings used at the run-time of the Linux kernel, we must adapt the file `configs/platform-friendlyarm-mini2440/projectroot/etc/network/interfaces` instead.

4.2.3 Feature Adaptions

Other features of the Mini2440 are:

- the attached LCD
- if the touch facility is used
- if a camera is present

These features can be enabled/disabled or configured at run-time with the kernel parameter `mini2440=`. The content of this parameter is also configured in the `config` files mentioned above.

It is important that the LCD is configured correctly, so that it works at run-time. Here is a list of currently known LCDs:

- **o N35**: 3.5" TFT + touchscreen (NEC NL2432HC22-23B/LCDN3502-23B)

- **1 A70**: 7” TFT + touchscreen (Innolux AT070TN83)
- **2**: VGA shield
- **3 t35**: 3.5” TFT + touchscreen (TD035STED4)
- **4**: 5.6” TFT + touchscreen (Innolux AT056TN52)
- **5 x35**: 3.5” TFT + touchscreen (Sony ACX502BMU-7)
- **6 w35i**: 3.5” TFT + touchscreen (Sharp LQ035Q1DG06)
- **7 N43**: 4.3” TFT + touchscreen (NEC NL4827HC19-01B / Sharp LQ043T3DX02)

The list above corresponds to the number (beginning with 0) given to the `mini2440=` kernel parameter to define the LCD in use.

When starting the kernel later on, it will output the list of supported displays with the currently selected one embraced.

```
MINI2440: LCD [0:240x320] 1:800x480 2:1024x768 3:240x320 4:640x480 5:240x320 6:320x240 7:480x272
```

As all of these existing LCDs differ in size and resolution, also userland may need more information than only their resolution. If we run Qt based applications the Qt library must know some additional data about the display as well. At least the physical size of the visible display area is an important value, as Qt uses this information to calculate the font’s scale.

The following changes are not required if you are using the 3.x kernel of this BSP. This kernel provides the visible size of the attached screen to userland. If you are using a different kernel, or the *VGA shield* the following changes are still to be done.

The BSP comes with a pre-configuration for the portrait *N35* 240 x 320 display. Its visible display area size is: width 53 mm, height 71 mm.

To forward this additional information to Qt, the file `configs/platform-friendlyarm-mini2440/projectroot/etc/profile.envir` exists in the BSP. We can edit it prior the build and change the size settings according to our own display if it differs from the default one. This file will be part of the root filesystem and used at run-time.

After these changes are made, we can continue building the board support package.

4.3 Selecting a Userland Configuration

First of all we have to select a userland configuration. This step defines what kind of applications will be built for the hardware platform. The OSELAS.BSP-Pengutronix-Mini2440-2012.06.0 comes with a predefined configuration we select in the following step:

```
$ ptxdist select configs/ptxconfig
info: selected ptxconfig:
      'configs/ptxconfig'
```

4.4 Selecting a Hardware Platform

Before we can build this BSP, we need to select one of the possible platforms to build for. In this case we want to build for the Mini2440:

```
$ ptxdist platform configs/platform-friendlyarm-mini2440/platformconfig-NAND-64M
info: selected platformconfig:
       'configs/platform-friendlyarm-mini2440/platformconfig-NAND-64M'
```

Note: If you have installed the OSELAS.Toolchain() at its default location, PTXdist should already have detected the proper toolchain while selecting the platform. In this case it will output:

```
found and using toolchain:
'/opt/OSELAS.Toolchain-2011.11/arm-v4t-linux-gnueabi/↵
    gcc-4.6.2-glibc-2.14.1-binutils-2.21.1a-kernel-2.6.39-sanitized/bin'
```

If it fails you can continue to select the toolchain manually as mentioned in the next section. If this autodetection was successful, you can omit the step of the next section and continue to build the BSP.

4.5 Selecting a Toolchain

If not automatically detected, the last step in selecting various configurations is to select the toolchain to be used to build everything for the target.

```
$ ptxdist toolchain /opt/OSELAS.Toolchain-2011.11/arm-v4t-linux-gnueabi/↵
      gcc-4.6.2-glibc-2.14.1-binutils-2.21.1a-kernel-2.6.39-sanitized/bin
```

4.6 Building the Root Filesystem

Now everything is prepared for PTXdist to compile the BSP. Starting the engines is simply done with:

```
$ ptxdist go
```

PTXdist does now automatically find out from the `selected_ptxconfig` and `selected_platformconfig` files which packages belong to the project and starts compiling their *targetinstall* stages (that one that actually puts the compiled binaries into the root filesystem). While doing this, PTXdist finds out about all the dependencies between the packages and builds them in the correct order.

While the command `ptxdist go` is running we can watch it building all the different stages of a package. In the end the final root filesystem for the target board can be found in the `platform-mini2440/root/` directory and a bunch of **.ipk* packets in the `platform-mini2440/packages/` directory, containing the single applications the root filesystem consists of.

4.7 Building an Image

After we have built a root filesystem, we can make an image, which can be flashed to the target device. To do this call

```
$ ptxdist images
```

PTXdist will then extract the content of priorly created **.ipk* packages to a temporary directory and generate an image out of it. PTXdist supports following image types:

- **hd.img:** contains grub bootloader, kernel and root files in a ext2 partition. Mostly used for x86 target systems.
- **root.jffs2:** root files inside a jffs2 filesystem.
- **uRamdisk:** a barebox/u-boot loadable Ramdisk
- **initrd.gz:** a traditional initrd RAM disk to be used as initrdramfs by the kernel
- **root.ext2:** root files inside a ext2 filesystem.
- **root.squashfs:** root files inside a squashfs filesystem.
- **root.tgz:** root files inside a plain gzip compressed tar ball.
- **root.ubi:** root files inside a ubi volume.

The to be generated image types and addtional options can be defined with

```
$ ptxdist platformconfig
```

Then select the submenu "image creation options". The generated image will be placed into `platform-mini2440/images/`.

Only the content of the **.ipk* packages will be used to generate the image. This means that files which are put manually into the `platform-mini2440/root/` will not be enclosed in the image. If custom files are needed for the target, install them with PTXdist.

5 Bring in the Bootloader Barebox

In order to make use of all possible boot sources the bootloader Barebox must be installed into the Mini2440. This bootloader enables the following boot sources:

- NAND flash memory: This enables the Mini2440 to boot standalone and very fast
- SD/MMC card: This enables the user to change all relevant software parts very easy by changing the SD/MMC card
- Network: This is intended for easy development

What we need for this step:

- a working network infrastructure
- host with network and USB capabilities
- a working TFTP server on our host
- some cables
  - network
  - USB-A to USB-B
  - RS232
- serial terminal running on our host

We assume here:

- the directory of the TFTP server is `/tftpboot`
- network is already configured for the host and the target (refer section 4.2.2)
- all connections are done (network, USB, serial)
- the serial terminal is able to handle 8 bits at 115200 Bd

To load the kernel and rootfs images, we first must load the new bootloader. This is the trickiest part, as we need special tools on our host and the target. Also, we may have to deal with confusing error messages.

First of all, we must change the **S2** switch on our Mini2440 to the NOR position to start the internal *vivi* bootloader. After switching on the Mini2440, the vivi bootloader will greet us with:

```
##### FriendlyARM BIOS for 2440 #####
[x] bon part 0 320k 2368k
[v] Download vivi
[k] Download linux kernel
[y] Download root_yaffs image
[a] Absolute User Application
[n] Download Nboot
[l] Download WinCE boot-logo
```

```
[w] Download WinCE NK.bin
[d] Download & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the nand flash
[b] Boot the system
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[i] Version: 1026-12
Enter your selection:
```

We want to use the *vivi* bootloader to load the new Barebox bootloader into the Mini2440's RAM. In order to do so, we need the size in bytes of Barebox's binary:

```
$ ls -l platform-mini2440/images/barebox-image
-rw-r--r-- 1 jb user 147844 Jun 04 23:07 platform-mini2440/images/barebox-image
```

The size of this binary may differ in your case. In our case here it is *147844*.

With this size we instruct the *vivi* bootloader to expect this number of bytes from the USB and store it to the internal RAM. To do so, we enter '*q*' to enter vivi's shell. Then we start the download command.

```
Enter your selection: q
Supervivi> load ram 0x31000000 147844 u
```

Please consider the *147844* number here. This number must be the same as size of your Barebox image.

At this point of time many error messages can happen. The Mini2440 may output `USB host is not connected yet`. In this case disconnect the USB cable again, powercycle the Mini2440 and try again.

At the host side, the system may not be able to enumerate the Mini2440 correctly. In this case also disconnect the Mini2440, powercycle it and connect it again.

You can check that the host was able to enumerate the Mini2440 successfully by issuing the `lsusb` command. If the following line occurs in the list, the Mini2440 is successfully enumerated:

```
Bus 001 Device 023: ID 5345:1234 Owon PDS6062T Oscilloscope
```

Note: The bus and device number may differ in your case.

If the USB connection is up and working on both sides, we can start to push the new bootloader into the target. This BSP comes with the required tool to do so.

```
$ sudo platform-mini2440/sysroot-host/bin/usbpush platform-mini2440/images/barebox-image
```

If the transfer was successful, the `usbpush` host tool will output:

```
csum = 0x74f9
send_file: addr = 0x30000000, len = 0x00024124
```

At the target side we will see:

```
Now, Downloading [ADDRESS:31000000h,TOTAL:147854]
RECEIVED FILE SIZE:  147854 (144KB/S, 1S)
Downloaded file at 0x31000000, size = 147844 bytes
```

Note: The numbers shown above may be different then what you see.

After a successful transfer, we can now run the downloaded bootloader:

```
Supervivi>  go 0x31000000
go to 0x31000000
  argument 0 = 0x00000000
  argument 1 = 0x00000000
  argument 2 = 0x00000000
  argument 3 = 0x00000000
```

This will start the Barebox bootloader on the Mini2440, which will greet us with:

```
barebox 2011.05.0-mini2440-ptx-2012.06.0 (Jun 24 2012 - 22:16:54)

Board: Mini 2440
NAND device: Manufacturer ID: 0xec, Chip ID: 0x76 (Samsung NAND 64MiB 3,3V 8-bit)
Bad block table found at page 131040, version 0x01
Bad block table found at page 131008, version 0x01
dm9000 i/o: 0x20000300, id: 0x90000a46
eth@eth0: got MAC address from EEPROM: FF:FF:FF:FF:FF:FF
refclk:    12000 kHz
mpll:     405000 kHz
upll:      48000 kHz
fclk:     405000 kHz
hclk:     101250 kHz
pclk:      50625 kHz
SDRAM1:   CL4@101MHz
SDRAM2:   CL4@101MHz
Malloc space: 0x33a00000 -> 0x33e00000 (size  4 MB)
Stack space : 0x339f8000 -> 0x33a00000 (size 32 kB)
envfs: wrong magic on /dev/env0
no valid environment found on /dev/env0. Using default environment
running /env/bin/init...

Hit any key to stop autoboot:  3
```

Stop the autoboot timeout by hitting any key. We are now in the shell environment of Barebox. To update the NAND content in the next step we need a working network first. One check is to show the current setting of the network interface. You should see your own settings here, done in section 4.2.2. Here is an example for a static network configuration:

```
mini2440:/ devinfo eth0
base  : 0x00000000
size  : 0x00000000
driver: none

Parameters:
          ipaddr = 192.168.1.240
         ethaddr = 00:04:f3:00:06:35
         gateway = 192.168.1.2
         netmask = 255.255.255.0
        serverip = 192.168.1.7
```

If you do not see appropriate values here and you are using the DHCP option, run the dhcp command first:

```
mini2440:/ dhcp
DHCP client bound to address 192.168.1.27
```

If you do not use DHCP for network configuration, you must edit the file `/env/config` in the Barebox environment.

```
mini2440:/ edit /env/config
```

Note: Barebox supports auto completion of commands, paths and filenames. Use the well known TAB key.

Edit the lines beginning with `eth0.*` and give them appropriate values. We can leave the editor by hitting CTRL-D to save our changes, or CTRL-C to discard any change. If we want these new settings to be persistent, we can save them now to NAND:

```
mini2440:/ saveenv
```

For further explanation about the default (compiled-in) Barebox environment and the "user defined" Barebox environment in NAND, see section 8.21.1.

To make the new static network configuration work, we must execute the `config` file again:

```
mini2440:/ . /env/config
```

Running the `devinfo eth0` command again should now show the values for the network interface that you put in earlier. To check if it is really working, try pinging other hosts:

```
mini2440:/ ping 192.168.1.7
host 192.168.1.7 is alive
```

In order to store the bootloader Barebox into the NAND, we can now use some of the builtin features in Barebox.

First we must copy - at the host side - the Barebox image from the board support package to the directory used by the TFTP server:

```
$ cp platform-mini2440/images/barebox-image /tftpboot/barebox-mini2440
```

Due to some scripts from the compiled-in environment and already setting up the network, storing Barebox into the NAND is now very easy:

```
mini2440:/ update -t barebox -d nand
```

That's all. To boot using the new bootloader, we must now change the switch **S2** back to the NAND position. Power cycle the Mini2440 or press its reset button and the new bootloader will start.

After installing Barebox on the Mini2440 the hardest part is done. Now it's time to decide about how to boot the Mini2440.

6 How to Boot the Mini2440

Various methods are exist to bring up the Mini2440. The main difference is if the Mini2440 can boot in a standalone manner or if it depends on some services from another host via network.

To start the Mini2440 in a standalone manner it provides some local memory types to store the relevant software parts.

- `NOR type flash memory` can be used to bring up the Mini2440. In this BSP NOR is used for the initial bootload of Barebox and as a fall-back or rescue bootloader. This allows the user to easily restore a "bricked" Mini2440.
- `NAND type flash memory` provides enough memory space to hold all run-time relevant software parts. This board support package supports this memory type as one way to start the Mini2440 standalone.
- `SD/MMC card memory` is a nice and easy way to deploy the run-time relevant software parts at the development host and simply booting the Mini2440 with it.

Note: All the mentioned boot methods below require Barebox as their bootloader. The easiest way to get Barebox into the Mini2440 is via its internal NAND. So, at least Barebox and its persistent environment has to be stored into the NAND flash memory. The other parts (kernel and root filesystem) can be loaded from all other supported media.

6.1 NOR Type Flash Memory

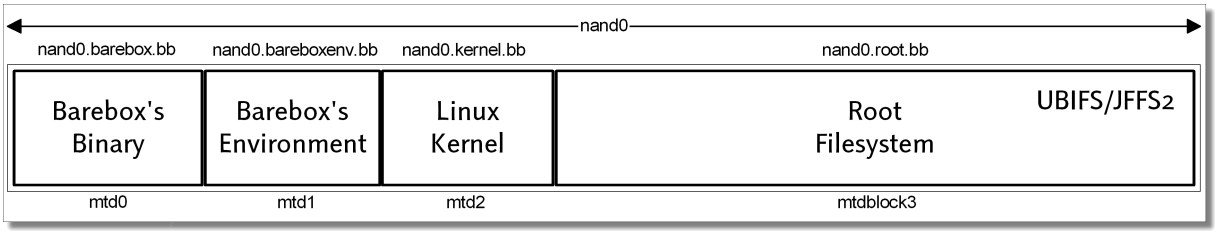
As the NOR type flash memory is not supported in this board support package, we skip its explanation here.

6.2 NAND Type Flash Memory

This is the most common method to boot the Mini2440. It does not depend on any external media or network connection. And it is also a fast method to boot it.

To use the NAND flash as the boot media, this board support package creates two files when running the last `ptxdist images` step:

- `platform-mini2440/images/linuximage` is the Linux kernel and must be stored in the corresponding NAND partition
- `platform-mini2440/images/root.jffs2` contains the root filesystem and must be stored 'as is' in the corresponding NAND partition



At our host's side:

```
$ cp platform-mini2440/images/root.jffs2 /tftpboot/root-mini2440.jffs2
$ cp platform-mini2440/images/linuximage /tftpboot/uImage-mini2440
```

Then we can run a script at the Mini2440's side to bring in the files into the NAND memory:

```
mini2440:/ update -t rootfs -d nand
mini2440:/ update -t kernel -d nand
```

Corresponding setup in the Barebox's `/env/config`:

```
kernel_loc=nand
rootfs_loc=nand
```

Specific settings in `/env/config` for the NAND case:

```
rootfs_type=jffs2
rootfs_mtdblock_nand=3
kernelimage_type=uimage
```

With these settings Barebox will automatically load the kernel from the NAND memory and instruct the kernel to also use the NAND memory for its root filesystem. `rootfs_mtdblock_nand` defines the partition number in the NAND memory the kernel should mount as its root filesystem. `rootfs_type` defines the used filesystem. `kernelimage_type` defines the type of the kernel image, to make Barebox aware how to extract it.

Start manually:

```
mini2440:/ boot nand
```

The manual start from the NAND memory requires the following specific settings in the `/env/config`:

```
rootfs_mtdblock_type=jffs2
rootfs_mtdblock_nand=3
kernelimage_type=uimage
```

6.3 SD/MMC Card Memory

To have all run-time relevant parts on an SD/MMC card, some help from the NAND type flash memory is required: the Barebox bootloader must be present to be able to load the Linux kernel from the SD/MMC media.

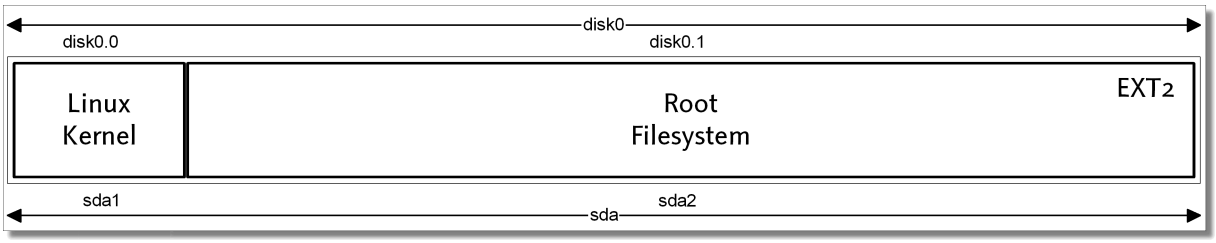
To deploy the card we just partitionate it and create a filesystem on the second partition. As this kind of media behaves like a regular hard disk, we can use any filesystem we like. But there is still a risk: this kind of media uses flash memory internally. And writing it often will destroy it over the time. That's why at least a journaling filesystem might not be a good idea, as they tend to write more data and more often as non journaling filesystems.

The generic Barebox's environment is prepared for the kernel in the first partition, the root filesystem in the second partition and *ext2* as the selected filesystem. There is no policy to use it in this way, but any variation would require some changes in the Barebox's default environment.

To use an SD/MMC card as the boot media, this board support package comes with two files after the last `ptxdist images` step:

- `platform-mini2440/images/linuximage` is the Linux kernel and must be copied plainly into the first partition of the SD/MMC card

- platform-mini2440/images/root.tgz contains the root filesystem and must be untared to the formatted second partition of the SD/MMC card



Note: The partition which contains the kernel image does not use any kind of filesystem. So, the kernel's image is copied to the *device*, not to its *filesystem*.

Persistent setup in the Barebox's /env/config.

```
kernel_loc=mmc
rootfs_loc=mmc
```

Specific settings in /env/config for the SD/MMC case:

```
rootfs_type=ext2
rootfs_mmc_part=2
kernel_mmc_part=0
kernelimage_type=uimage
```

With these settings Barebox will automatically load the kernel from the SD card and instruct the kernel to also use the SD card as its root filesystem. kernel_mmc_part defines the SD/MMC card's partition number to load the kernel from. rootfs_mmc_part defines the partition on the SD/MMC card the kernel should use as its root filesystem and rootfs_type defines the used filesystem. kernelimage_type defines the type of the kernel image, to make Barebox aware how to extract it.

Start manually:

```
mini2440:/ boot mmc
```

The manual start from the SD/MMC card requires the following specific settings in the /env/config:

```
rootfs_type=ext2
rootfs_mmc_part=2
kernel_mmc_part=0
kernelimage_type=uimage
```

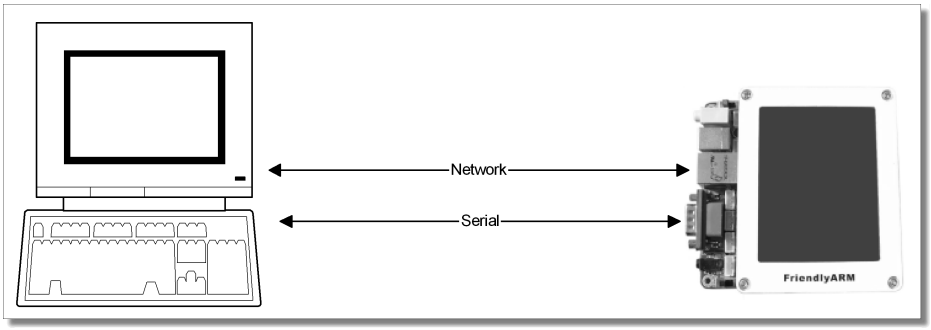
6.4 Network Memory

Using the network based method to boot the Mini2440 is more dedicated for development. It simplifies replacing software in the kernel and root filesystem. Also, the development cycle time can be reduced, with fewer opportunities for making errors. All run-time relevant software parts are still part of the development host in this case and can be easily changed on the host and are instantly visible at the target's side.

To use the network based method this board support package creates the following parts:

- platform-mini2440/images/linuximage is the Linux kernel and must be copied to the NFS or TFTP exported directory with a name the target expect it (in our case here for example uImage-mini2440)

- `platform-mini2440/root/` contains the root filesystem to be exported via NFS



Persistent setup in the Barebox's `/env/config`.

Using the TFTP protocol for downloading the kernel:

```
kernel_loc=tftp
rootfs_loc=net
```

Specific settings in `/env/config` for the TFTP case:

```
kernelimage=uImage-mini2440
kernelimage_type=uimage
nfsroot="/path/to/nfs/root"
```

With these settings Barebox will automatically load the kernel with the name `kernelimage` via network from a TFTP server and instruct it to use the path given in `nfsroot` as its root filesystem. `kernelimage_type` defines the type of the kernel image, to make Barebox aware how to extract it.

Using the NFS protocol for downloading the kernel:

```
kernel_loc=nfs
rootfs_loc=net
```

Specific settings in `/env/config` for the NFS case:

```
kernelimage=uImage-mini2440
kernelimage_type=uimage
nfsroot="/path/to/nfs/root"
```

With these settings Barebox will automatically load the kernel with the name `kernelimage` via network from a NFS server and instruct it to use the path given in `nfsroot` as its root filesystem. `kernelimage_type` defines the type of the kernel image, to make Barebox aware how to extract it.

Start manually and download the kernel via TFTP:

```
mini2440:/ boot tftp
```

The manual start from a TFTP server requires the following specific settings in the `/env/config`:

```
kernelimage=uImage-mini2440
kernelimage_type=uimage
nfsroot="/path/to/nfs/root"
```

Start manually and download the kernel via NFS:

```
mini2440:/ boot nfs
```

The manual start from an NFS server requires the following specific settings in the /env/config:

```
kernelimage=uImage-mini2440
kernelimage_type=uimage
nfsroot="/path/to/nfs/root"
```

6.5 Adaptions

As mentioned above all settings are based on changes in a simple text file. Whenever we run the 'boot' command, the script in /env/bin/boot is called. So, adaptions can be made in /env/config to use one of the supported boot methods in /env/bin/boot. For more complicated scenarious also the /env/bin/boot can be changed. As it's a simple shell script, it's very easy to adapt it to different requirements.

7 Updating the Mini2440

At any time it's possible to update any of the software components running on the Mini2440.

- the bootloader Barebox
- Barebox's persistent environment
- the Linux kernel
- the kernel's root filesystem

This chapter explains how to update these parts for the various boot scenarious. Note: This chapter is not finished yet.

7.1 NOR flash memory case

Not yet supported.

7.2 NAND flash memory case

If the Mini2440 uses the NAND flash memory to boot standalone, we can update its components in the following way:

7.2.1 Updating the Bootloader

Most of the time there is no further need to re-flash the bootloader and its persistent environment. After it was setup once, it does its work "in the background". But nevertheless there could be the need to update the bootloader due to feature additions or bug fixes. If the current Barebox bootloader is still working, its replacement can be done by using the existing bootloader. This assumes that the network is still functioning. In this case, a simple

```
$ cp platform-mini2440/images/barebox-image /tftpboot/barebox-mini2440
```

provides the updated bootloader binary via TFTP and a

```
mini2440:/ update -t barebox -d nand
```

will do the job at the target side. After starting this command, do not disturb! This is a critical update process. Because, for a short period of time the NAND flash is erased, with no bootloader present. But don't panic: Unless a power fail or a target reset happens, this command can be repeated if the first run failed.

7.2.2 Updating the Persistent Environment

Updating the persistent environment is also possible. A simple

```
$ cp platform-mini2440/images/barebox-default-environment /tftpboot/barebox-default-environment-mini2440
```

provides the updated environment via TFTP and a

```
mini2440:/ update -t bareboxenv -d nand
```

will do the job. Note: This new persistent environment will be used at the next system start.

If the persistent environment is broken, there is a second method to restore a working environment: that is, using the compiled in default environment which comes with Barebox. To force the usage of the compiled in default environment, just erase the current one in the NAND flash memory and reset the target (or run the `reset` command).

```
mini2440:/ erase /dev/bareboxenv.bb
mini2440:/ reset
```

Now, Barebox will stumble about the empty partition and then fall back to its compiled-in environment version. This can now be changed by editing the files in `env/` and then saved back to the NAND flash memory with the command

```
mini2440:/ saveenv
```

7.2.3 Updating the Linux Kernel

Changing the Linux kernel configuration can be quite dynamic, especially while the developer is trying different kernel configurations. Updating this part happens in the same way like the other parts. Providing the Linux kernel via TFTP:

```
$ cp platform-mini2440/images/linuximage /tftpboot/uImage-mini2440
```

and running the update script at the target's side:

```
mini2440:/ update -t kernel -d nand
```

7.2.4 Updating the Root Filesystem

And last, but not least, updating the root filesystem. Same procedure:

```
$ cp platform-mini2440/images/root.jffs2 /tftpboot/root-mini2440.jffs2
```

and running the update script at the target's side:

```
mini2440:/ update -t rootfs -d nand
```

7.3 SD/MMC card memory case

We assume here the partitioning is done as mentioned in section 6.3: two partitions on the SD/MMC card, first contains the kernel image and no filesystem, second contains the root filesystem content with a filesystem we like. We also assume here, updating the card content is not done at the target's side, it is still done at our host. This means, we are using some kind of USB based card reader and these kind of devices show us the attached cards as SCSI devices.

7.3.1 Updating the Linux Kernel

Changing the kernel is rather simple:

```
$ sudo cp platform-mini2440/images/linuximage /dev/sdd1
```

Note: Replace the `/dev/sdd1` here with the device node name the card reader is visible in your system.

7.3.2 Updating the Root Filesystem

Changing the root filesystem content mostly means we must remove its old content. This can be done quickly by re-formatting it.

```
$ sudo mkfs.ext2 /dev/sdd2
$ sudo mount /dev/sdd2 /mnt
$ sudo tar -xvzf platform-mini2440/images/root.tgz -C /mnt
$ sudo umount /mnt
```

7.4 Network memory case

As this kind of memory is on our host, everything we change will be also changed instantly at the Mini2440's side (as long we are talking about the kernel and the root filesystem).

By the way: the `update` command is not a real command built into Barebox. Its a simple shell script coming from the persistent environment. If one has different update scenarios she/he can change or adapt this script. Changing this behaviour can be done without touching Barebox's source code.

8 Special Notes

8.1 Available Kernel Releases

The predifined Mini2440 platform configuration always uses the latest Linux kernel release. If users want to stay with an older Linux kernel release, they are also available. Here is a list of currently available Linux kernel releases in the OSELAS.BSP-Pengutronix-Mini2440-2012.06.0:

- 3.4, patch level 4 (default)
- 3.3, stable patch level 8
- 3.2, stable patch level 21
- 3.1, stable patch level 10
- 3.0, stable patch level 36
- 2.6.39, stable patch level 4
- 2.6.38, stable patch level 8

If we want to build the BSP with a non-default kernel release, we just run `ptxdist platformconfig` and change the kernel setting prior to building. As PTXdist checks the MD5 sums of the archives, we also have to change the MD5 sum of the corresponding kernel archive.

Note: The MD5 sums for the kernels are (used by PTXdist):

- 3.4: `967f72983655e2479f951195953e8480`
- 3.3: `7133f5a2086a7d7ef97abac610c094f5`
- 3.2: `364066fa18767ec0ae5f4e4abcf9dc51`
- 3.1: `edbdc798f23ae0f8045c82f6fa22c536`
- 3.0: `ecf932280e2441bdd992423ef3d55f8f`
- 2.6.39: `25cd73d797a49ad5b4119b67f1caf2cc`
- 2.6.38: `0f28e3a47495ede6ff8b5d5c97680fe5`

For older PTXdist revisions than 2012.05.0 the kernel's archive format is *bz2* instead of the newer *xz*. Here are the MD5 sums for reference for the *bz2* format.

- 3.3: `98a6cdd7d082b7ea72df9c89842bac74`
- 3.2: `7ceb61f87c097fc17509844b71268935`
- 3.1: `8d43453f8159b2332ad410b19d86a931`
- 3.0: `398e95866794def22b12dfbc15ce89c0`
- 2.6.39: `1aab7a741abe08d42e8eccf20de61e05`
- 2.6.38: `7d471477bfa67546f902da62227fa976`

8.2 Available Userland Configuration

The Mini2440 BSP comes with two different predefined userland configurations:

- **configs/ptxconfig**: it is the standard one to get a small running embedded system. It can be used as a base for your own development running the Mini2440 headless.
- **configs/ptxconfig.qt**: this configuration is intended for graphical usage of the Mini2440. It has the Qt library enabled and brings in two small Qt based application examples. These applications will be started automatically at system's startup, to show how to get a graphical system up and running.

It's up to you and your needs which configuration you may choose in section 4.3.

8.2.1 Some details about the `configs/ptxconfig.qt`

The mentioned small Qt based applications we can find in `local_src/qt4-demo-2011.12.0/` and `local_src/qml-demo-2011.12.0/`. They can act as a template for our own Qt development.

As only one of these Qt applications can run at system start, we have to select which one we want to run prior the build of the BSP. To do so, after selecting the `configs/ptxconfig.qt` configuration, we run menuconfig and select one of the available demos. The Qt4/QWT demo is the default one.

```
$ ptxdist menuconfig
  Graphics & Multimedia  --->
     qt --->
        <*> Mini2440 related Qt demos --->
           Demo type (Qt4/QWT demo) --->
              (X) Qt4/QWT demo
              ( ) Qt4/QML demo
```

The "secrets" how to build and install these applications we can find in

- `mini2440-demo.in`
- `rules/qt4-demo.make`
- `rules/qml-demo.make`

The "magic" behind the autostart of these small Qt based applications at system startup can be found in `local_src/qt4-demo-2011.12.0/init/sysv/startup` and `local_src/qml-demo-2011.12.0/init/sysv/startup`.

Note: The Qt4/QWT demo is prepared to run on a portrait 240 x 320 screen. If your screen differs from this setup, don't expect a correct image. The Qt4/QML demo adapts itself to the available screen size and format.

8.3 Framebuffer

This driver gains access to the display via device node `/dev/fb0`. For this BSP the LCDN3502-23B (N35) display with a resolution of 240x320 is supported.

A simple test of this feature can be run with:

```
# fbtest
```

This will show various pictures on the display.

You can check your framebuffer resolution with the command

```
# fbset
```

NOTE: `fbset` cannot be used to change display resolution or colour depth. Depending on the framebuffer device different kernel command line may be needed to do this. Please refer to your display driver manual for details.

8.4 GPIO

Like most modern System-on-Chip CPUs, the S3C2440 has numerous GPIO pins. Some of them are inaccessible for the userspace, as Linux drivers use them internally. Others are also used by drivers but are exposed to userspace via sysfs. Finally, the remaining GPIOs can be requested for custom use by userspace, also via sysfs.

Refer to the in-kernel documentation `Documentation/gpio.txt` for complete details how to use the sysfs-interface for manually exporting GPIOs.

8.4.1 GPIO Usage Example

When generic architecture GPIO support is enabled in the kernel, some new entries appear in sysfs. Everything is controlled via read and writable files to generate events on the digital lines.

We find all the control files in `/sys/class/gpio`. In that path, there are a number of `gpiochipXXX` entries, with `XXX` being a decimal number. Each of these folders provide information about a single GPIO controller registered on the Mini2440 board, for example with `gpiochip192`:

```
# ls /sys/class/gpio/gpiochip192
base       label      ngpio      subsystem  uevent
```

The entry `base` contains information about the base GPIO number and `ngpio` contains all GPIOs provided by this GPIO controller.
We use `GPIO193` as an example to show the usage of a single GPIO pin.

```
# echo 193 > /sys/class/gpio/export
```

This way we export `gpio193` for userspace usage. If the export was successful, we will find a new directory named `/sys/class/gpio/gpio193` afterwards. Within this directory we will be able to find the entries to access the functions of this GPIO. If we wish to set the direction and initial level of the GPIO, we can use the command:

```
# echo high > /sys/class/gpio193/direction
```

This way we export `GPIO193` for userspace usage and define our GPIO's direction attribute to an output with initially high level. We can change the value or direction of this GPIO by using the entries `direction` or `value`.

Note: This method is not very fast, so for quickly changing GPIOs it is still necessary to write a kernel driver. The method shown works well, for example to influence an LED directly from userspace.

To unexport an already exported GPIO, write the corresponding gpio-number into `/sys/class/gpio/export`.

```
# echo 193 > /sys/class/gpio/unexport
```

Now the directory `/sys/class/gpio/gpio193` will disappear.

Note: The GPIO193 is available at connector 4, pin 17 for measurement.

8.5 I²C Master

The S3C2440 processor based Mini2440 supports a dedicated I²C controller onchip. The kernel supports this controller as a master controller.

Additional I²C device drivers can use the standard I²C device API to gain access to their devices through this master controller. For further information about the I²C framework see `Documentation/i2c` in the kernel source tree.

8.5.1 I²C Device AT24c08

This device is a 1024 bytes non-volatile memory for general purpose usage.

This type of memory is accessible through the sysfs filesystem. To read the EEPROM content simply `open()` the entry `/sys/bus/i2c/devices/0-0050/eeprom` and use `fseek()` and `read()` to get the values.

8.6 LEDs

The LEDs on the Mini2440 can be controlled via the LED-subsystem of the Linux kernel. It provides methods for switching them on and off as well as using so-called triggers. For example, you could trigger the LED using a timer. That enables us to make it blink with any frequency we want.

All LEDs can be found in the directory `/sys/class/leds`. Each one has its own subdirectory. We will use `led1` for the following examples.

For each directory, you have a file named `brightness` which can be read and written with a decimal value between 0 and 255. The first one means LED off, the latter maximum brightness. Inbetween values scale the brightness if the LED supports that. If not, non-zero means just LED on.

```
/sys/class/leds/led1# echo 255 > brightness; # LED on
/sys/class/leds/led1# echo 128 > brightness; # LED at 50% (if supported)
/sys/class/leds/led1# echo 0 > brightness; # LED off
```

LEDs can be connected to triggers. A list of available triggers we can get from the `trigger` entry

```
/sys/class/leds/led1# cat trigger
[none] nand-disk mmc0 timer backlight
```

The embraced entry is the currectly connected trigger to this LED.

To change the trigger source to the *timer*, just run a:

```
/sys/class/leds/led1# echo timer > trigger
```

If the timer-trigger is activated you should see two additional files in the current directory, namely `delay_on` and `delay_off`. You can read and write decimal values there, which will set the corresponding delay in milliseconds. As an example:

```
/sys/class/leds/led1# echo 250 > delay_on
/sys/class/leds/led1# echo 750 > delay_off
```

will blink the LED being on for 250ms and off for 750 ms.

Replace `timer` with `none` to disable the trigger again. Or select a different one from the list read from the `trigger` entry.

Refer to `Documentation/leds/leds-class.txt` in-kernel documentation for further details about this subsystem.

8.7 MMC/SD Card

The Mini2440 supports *Secure Digital Cards* and *Multi Media Cards* to be used as general purpose blockdevices. These devices can be used in the same way as any other blockdevice.

These kind of devices are hot pluggable, so you must pay attention not to unplug the device while its still mounted. This may result in data loss.

After inserting an MMC/SD card, the kernel will generate new device nodes in `dev/`. The full device can be reached via its `/dev/mmcblk0` device node, MMC/SD card partitions will occur in the following way:

```
/dev/mmcblk0pY
```

`Y` counts as the partition number starting from 1 to the max count of partitions on this device.

Note: These partition device nodes will only occur if the card contains a valid partition table (”harddisk” like handling). If it does not contain one, the whole device can be used for a filesystem (”floppy” like handling). In this case `/dev/mmcblk0` must be used for formatting and mounting.

The partitions can be formatted with any kind of filesystem and also handled in a standard manner, e.g. the `mount` and `umount` command work as expected.

8.8 Network

The Mini2440 module has a DM9000 ethernet chip onboard, which is being used to provide the `eth0` network interface. The interface offers a standard Linux network port which can be programmed using the BSD socket interface.

8.9 SPI Master

The Mini2440 board supports an SPI bus, based on the S3C2440’s integrated SPI controller. It is connected to the onboard devices using the standard kernel method, so all methods described here are not special to the Mini2440.

Connected devices can be found in the sysfs at the path `/sys/bus/spi/devices`. It depends on the corresponding SPI slave device driver providing access to the SPI slave device through this way (sysfs), or any different kind of API.

Currently no SPI slave devices are registered, so the `/sys/bus/spi/devices` directory is empty.

8.10 Touchscreen

A simple test of this feature can be run with:

```
# ts_calibrate
```

to calibrate the touch and with:

```
# ts_test
```

to run a simple application using this feature.

To see the exact events the touch generates, we can also use the `evtest` tool.

```
# evtest /dev/input/touchscreen
Input driver version is 1.0.1
Input device ID: bus 0x19 vendor 0xdead product 0xbeef version 0x102
Input device name: "S3C24XX TouchScreen"
Supported events:
  Event type 0 (Sync)
  Event type 1 (Key)
    Event code 330 (Touch)
  Event type 3 (Absolute)
    Event code 0 (X)
     Value      0
     Min        0
     Max     1023
  Event code 1 (Y)
     Value      0
     Min        0
     Max     1023
Testing ... (interrupt to exit)
```

Whenever we touch the screen this tool lists the values the driver reports.

Don't rely on the event node the kernel creates in the `/dev/input/` direcory. Over the time the index can change. This board support package comes with a special *udev* rule, which creates a link called `touchscreen` always pointing to the correct event node. Use the `touchscreen` in your setup instead of the plain event node.

8.10.1 If the Touchscreen does not work

A functional touchscreen depends on some external configurations and parameters. Firstly, the touchscreen driver for the S3C2440 CPU must be enabled in the kernel. If it is supported, it can be checked at run-time with the following command:

```
# ls /sys/bus/platform/drivers
```

A `samsung-ts` must be listed in this directory. If not, the kernel must be re-configured to support this device.

Secondly, a functional touchscreen depends on is the registered touchscreen device. If it is registered, this can be checked at run-time with this command:

```
# ls /sys/bus/platform/devices
```

A s3c2440-ts must be listed in this directory. If not, something is preventing the kernel from registering this device. The touchscreen on this platform is an optional part, so it must be enabled on demand to make it work. The touchscreen is enabled by the mini2440= kernel parameter. If the running kernel receives the correct parameter this setting can be checked with:

```
# cat /proc/cmdline
console=ttySAC0,115200 mini2440=0tb mtdparts=nand:256k(barebox),64k(bareboxenv),2048k(kernel),-(root)
```

Referring to the mini2440=0tb parameter, specifically to the 't'. If the 't' is present the touchscreen gets registered at run-time and can be used. If the 't' is missing the touchscreen will NOT be registered.

To add a missing 't', restart the target, stop Barebox from booting and edit the bootparameter in the /env/config file. Save the new settings and boot again.

8.10.2 If the Touchscreen does not work as expected

It's not easy to create a touchscreen driver that works with all kinds of touchscreens. They differ in their hardware parameters, so most of the time some adaptions must be done to get better results.

Two locations exists where parameters can be changed:

- in the kernel driver
- in the *tslib* (**t**ouch**s**creen **lib**rary)

The *tslib* is a userland component and can be changed at any time. The kernel driver is a compiled in component, so the kernel must be re-built and re-started to make any change visible.

Lets start with the kernel driver: It uses three parameters to support the physical behaviour of the touchscreen.

- **.delay** a delay counted in clocks of 3.68 MHz between the measurement of the X and Y coordinate. If the touchscreen lines are filtered with a low-pass it could be useful to increase this value. Max value is oxffff.
- **.presc** clock prescaler for the AD converter. The larger the value is, the lower the AD converter works (FIXME: Seems not be used)
- **.oversampling_shift** defines the samples to be measured and to be averaged before reporting a coordinate. '0' means one sample per report, '1' means two samples per report, '2' means 4 samples and so on.

To modify the setting, open the file platform-mini2440//build-target/linux-3.4/arch/arm/mach-s3c2440/mach-mini2440.c and search for the mini2440_ts_cfg structure. It looks like this:

```
static struct s3c2410_ts_mach_info mini2440_ts_cfg __initdata = {
        .delay = 10000,
        .presc = 0xff,
        .oversampling_shift = 0,
};
```

After modifying, the kernel must be re-built:

```
$ ptxdist drop kernel compile
$ ptxdist go
```

These steps ensure the modified sources are re-compiled. Use this new kernel and do the tests with the touchscreen again.

To change the userland *tslib* this can be done at run-time of the Mini2440. Just modify the `/etc/ts.conf`.

- **module_raw input** means the tslib uses the raw data from the Linux's input system
- **module pthres pmin=1** means the minimal pressure must be '1' to count as a touchscreen event. Other values do not make sense yet, as the driver does not support pressure measurement
- **module variance delta=30** FIXME
- **module dejitter delta=10** FIXME
- **module linear** FIXME

After changing one of these entries a run of `ts_test` can show if the new settings are better than the previous ones.

8.11 LCD Backlight

The backlight of the LCD can be controlled via the sysfs entry in:

```
/sys/class/leds/backlight/
```

To switch it *off*, just write a '0' into its `brightness` entry:

```
# echo 0 > /sys/class/leds/backlight/brightness
```

and a '1' to switch it *on* again:

```
# echo 1 > /sys/class/leds/backlight/brightness
```

8.12 USB Host Controller Unit

The Mini2440 supports a standard OHCI Rev. 1.0a compliant host controller onboard for low and full speed connections.

8.13 Watchdog

The internal watchdog will be activated when an application opens the device `/dev/watchdog`. Default timeout is 15 seconds. An application must periodically write to this device. It does not matter what is written. Just the interval between these writes should not exceed the timeout value, otherwise the system will be reset.

For testing the hardware, there is also a shell command which can do the triggering:

```
# watchdog -t <trigger-time-in-seconds> /dev/watchdog
```

This command is part of the busybox shell environment. Keep in mind, that it should only be used for testing. If the watchdog gets fed by it, a crash of the real application will go unnoticed.

For the Mini2440 the default 60 seconds interval period the tool is using is too long. The driver for the S3C2440 only supports up to a 40 seconds interval. So, the additional parameter `-T 40` must be given.

8.14 ADC

Getting the digital equivalent of one of the analogue input channels can be done by reading the corresponding entries in the sysfs.

For example the analogue input channel 0 on the Mini2440 is connected to the potentiometer W1. By reading the entry `/sys/devices/platform/s3c24xx-adc/s3c-hwmon/in0_input` we can watch the different digital values while turning the potentiometer W1.

Note: The analogue input channels 4 ... 7 are occupied by the touchscreen feature and can only be used as simple analogue inputs if the touchscreen feature is disabled.

8.15 Keypad

Using the up to 6 available key buttons on the Mini2440 in a regular manner requires a working console in the kernel. Here the list of the current key codes they generate when pressed:

- K1, code 'F1'
- K2, code 'F2'
- K3, code 'F3'
- K4, code 'Power'
- K5, code 'F5'
- K6, code 'F6'

If one wants to change the generated codes, she/he can change it in the platform code found in `arch/arm/mach-s3c2440/mach-mini2440.c`, specially in the array `mini2440_buttons`.

If the key buttons are working as expected, can also be checked without a working console with the following command:

```
# evtest /dev/input/buttons
Input driver version is 1.0.1
Input device ID: bus 0x19 vendor 0x1 product 0x1 version 0x100
Input device name: "gpio-keys"
Supported events:
  Event type 0 (Sync)
  Event type 1 (Key)
    Event code 59 (F1)
    Event code 60 (F2)
    Event code 61 (F3)
    Event code 63 (F5)
    Event code 64 (F6)
    Event code 116 (Power)
Testing ... (interrupt to exit)
```

Don't rely on the event node the kernel creates in the `/dev/input/` direcory. Over the time the index can change. This board support package comes with a special *udev* rule, which creates a link called `buttons` always pointing to the correct event node. Use the `buttons` in your setup instead of the plain event node.

8.16 Audio

This kernel supports the audio capabilities of the Mini2440 via a standard ALSA device. So, most of the available tools to play or record sounds should work out of the box.

To control the audio mixer run the tool `alsamixer`, to play a simple sound file `aplay` can be used and for MP3 files, textttmadplay is the correct tool.

8.17 USB Device

The S3C2440 processor in the Mini2440 comes with a USB device unit. This is the physical interface to let the Mini2440 act in some roles in the USB world. For example the Mini2440 can act as a printer or a simple serial adapter. There are also drivers to act as a mass storage device, but its setup is more complicated. So, this section describes the printer case.

To prepare the Mini2440 to act as a printer just load the printer gadget driver.

```
# modprobe g_printer
Printer Gadget: Printer Gadget, version: 2007 OCT 06
Printer Gadget: using s3c2410_udc, OUT ep2-bulk IN ep1-bulk
```

Starting this driver will create a `/dev/g_printer` device node. This device node can be opened for reading and writing. It's the end of two "pipes" for data to and from a connected host.

Now, the Mini2440 is ready for connection to a host via its USB B plug. If it works, the kernel at the host side will detect a new device:

```
usb 1-1: new full speed USB device number 2 using s3c2410-ohci
usblp0: USB Bidirectional printer dev 7 if 0 alt 0 proto 2 vid 0x0525 pid 0xA4A8
```

Note: At the host side the `usblp` module is required to make this new USB hotplug device visible as a printer.

At the host side now a `/dev/usblp0` device node will be created. Also this device node can be opened for reading and writing. And also this node is the end of two "pipes" for data to and from the "printer".

Everything we 'echo' into `/dev/usblp0` at the host side, we can 'cat' from `/dev/g_printer` at the Mini2440 side. And vice versa.

And a real funny game is to connect Mini2440's USB A to its own USB B. Then the Mini2440 can talk to itself.

8.18 Buzzer

The buzzer on the Mini2440 will be triggered by a regular alert in a console. So, a simple

```
# echo -e "\a"
```

can make a noise. But this only works if local console support is enabled.

If no local console is available the small `ring-bell` tool will help.

```
# ring-bell /dev/input/beeper
```

will make some noise for you.

8.19 Get the latest BSP Release for the Mini2440

Information and the latest release of the Mini2440 BSP can be found on our website at:

http://www.oselas.org/oselas/bsp/index_en.html

8.20 Be Part of the Mini2440 BSP Development

If you want to use the latest and greatest board support package for the Mini2440 you can use the git repository as your working source, instead of a release archive.

The git repository can be found here:

```
http://git-public.pengutronix.de/git-public/OSELAS.BSP-Pengutronix-Mini2440.git
```

If you want to contribute to this project by sending patches, these patches should always be based on the **master** branch of this repository.

8.21 Notes About the Bootloader Barebox

Everything mentioned here (variable names and file names) in the run-time environment that Barebox uses, is for convenience only. The developers of Barebox decided to provide a generic run-time environment that satisfies the most common requirements. All descriptions below will refer to this generic run-time environment and it's behaviour.

There are no restrictions in how to adapt this environment for one's own needs. How Barebox enters it's shell is compiled-in. Changing the `/env/bin/init`, allows one to modify Barebox's behaviour.

8.21.1 Run-Time Environment

The Barebox binary handles only target initialization and provides device drivers and various commands to do things after the initialization. It is up to the user to use these features to make her/his target work. This works on a shell code base. For example, Barebox tries to run the `/env/bin/init` script right after the initialization is finished. This file is expected as a part of the environment.

From the technical point of view, the Barebox environment is a simple archive which contains files and directories. At startup, this archive will be extracted to the `env/` directory and can be used afterwards on a regular file base. Note: the / directory in Barebox is a RAM filesystem.

Barebox always tries to load the user environment archive from the registered persistent media first (aka "user env"). If this fails, Barebox defaults to the default compiled-in environment archive.

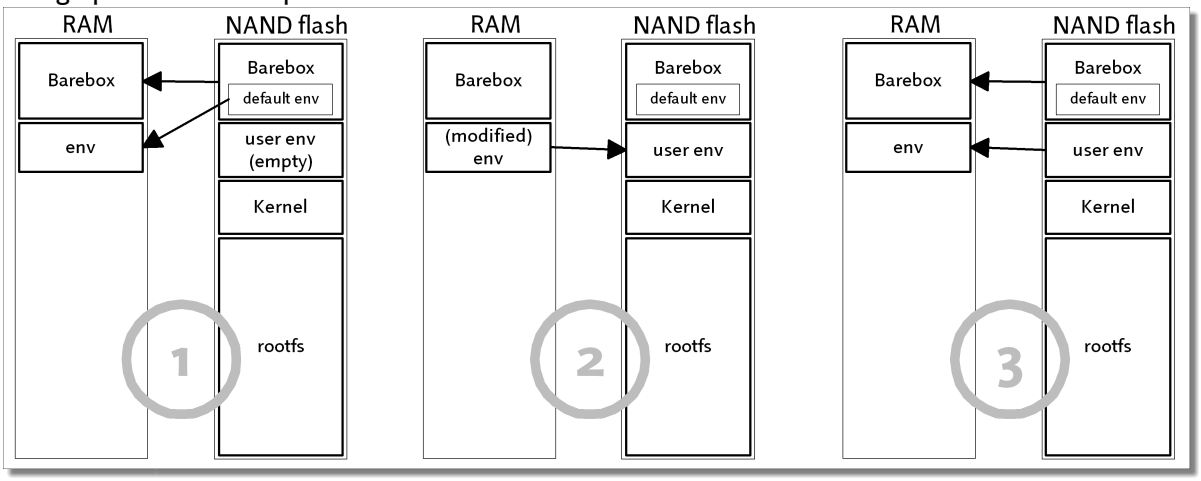
The compiled-in default environment archive is a read only archive defined at compile-time of Barebox.

The *user env* can be changed at any point of time.

8.21.1.1 Run-Time Environment (env) and how it is controlled by Barebox

When starting from a totally clean system (e.g. NAND flash is empty) and if you do an `update -t barebox -d nand` to bring in the Barebox bootloader into the NAND flash and then reset the mini2440, Barebox will use its default compiled-in environment because the *user env* is still empty. To change the run-time Barebox environment, on the target, one can do `edit /env/config` (to make the changes), *CTRL D* (to store the changes to the RAM disk and leave the editor) and a `saveenv` (to save the changes to the *user env*). The next time we boot the mini2440, the *user env* takes priority. Note: the default Barebox environment has not been changed.

Here a graphic to show the process:



- state 1: the 1st boot. Barebox runs in RAM and loads the *env* from the default compiled-in environment due to the *user env* is still empty
- state 2: the user changes the *env* and saves it. Now, the *user env* is no longer empty
- state 3: At the next system start Barebox now ignores the default compiled-in environment and loads its *env* from the *user env*

The above method is appropriate for minor changes in the Barebox environment. For major env changes it is suggested that we modify the environment files in the board support package instead of on the target. On our host it's easier to change these files (for example with our favourite editor instead of the simple editor embedded in Barebox) and these changes remain, should we want to re-build everything again.

The environment is a collection of files from different locations in the board support package. One location is the `configs/platform-friendlyarm-mini2440/barebox-64m-env/`. Another location is `platform-mini2440//build-target/barebox-2011.05.0/defaultenv/`. The second location contains the more generic shell scripts, while the first location contains user specific settings. If we intend to change the generic shell scripts, it's easier to copy them to the `configs/platform-friendlyarm-mini2440/barebox-64m-env/` location instead of modifying them at their original path (as this modification would be lost, if we run `ptxdist clean`). The files at `configs/platform-friendlyarm-mini2440/barebox-64m-env/` takes priority over the one from the `platform-mini2440//build-target/barebox-2011.05.0/defaultenv/` if files with the same name exits.

After such a change we should run:

```
$ ptxdist clean barebox
$ ptxdist go
$ ptxdist images
```

This rebuilds Barebox and includes the changed default compiled-in environment.

Next, make these files available for download via network/TFTP:

```
$ cp platform-mini2440/images/barebox-image /tftpboot/barebox-mini2440
```

Now,

```
mini2440:/ update -t barebox -d nand
```

together with its new compiled-in default environment. But at the next system start also this new Barebox would use the *user env* if it is still present. To make it use the new compiled-in default environment we must either remove the *user env* with:

```
mini2440:/ erase /dev/nand0.bareboxenv.bb
```

or update the *user env* with the prepared archive from the last build:

```
$ cp platform-mini2440/images/barebox-default-environment /tftpboot/barebox-default-environment-mini2440
```

```
mini2440:/ update -t bareboxenv -d nand
```

8.21.2 How does the Partitioning Work in Barebox

Partitioning is a way to handle large media in smaller logical units. This simplifies updates of different components and leaves others untouched. For example, one can update the kernel to fix a bug in a driver but keep the root filesystem unchanged. Also, redundant boot can be realized with more than one partition per component.

Barebox uses partitioning of the available persistent media (for example, NOR or NAND flash, but also harddisks or SD cards) to handle and store the required parts to make a target work.

Some of the available persistent media can store it's partition information on the media itself. For example hard disks, compact flash cards or SD cards can provide their own partition table.

In this case, Barebox can read back this table from the media and handle these partition's sizes and locations in a correct manner.

But, there are still some media that do not provide this kind of partition table. The well known plain flash devices (of type NOR or NAND) are such candidates. These devices need slightly different handling. The most common method the kernel uses is the *Command line partition table parsing* for the MTD (Memory Technology Devices) devices. A user gives a kernel parameter with the list of names and sizes that describes the partition layout of the corressponding flash memory.

Barebox uses the same syntax to describe the partition and kernel layout. So, a user only has to define the layout once. It will be shared between Barebox and the Linux kernel. If one doesn't use consistent layout, one could destroy the data in one partition by changes in another parition.

This partition layout string is defined to:

```
<size>(<name>)[,<size>(<name>)[,<size>(<name>)]...]
```

`<size>` is a number followed by its unit. The unit can be `k` for *kilobyte*, `M` for *megabyte* and `G` for *gigabyte*. For `<size>` also the special letter `-` can be given. This means, fill the remaining space up to the end of the media. The `<name>` can be anything one likes, but must not contain any spaces!

Here is the most common partition layout configuration:

In Barebox's run-time environment it looks like:

```
256k(barebox),64k(bareboxenv),2048k(kernel),-(root)
```

- bootloader itself (*barebox*): this binary brings up the target after power on or reset
- persistent environment (*bareboxenv*): used by Barebox to bring up the whole system in the way that the user has configured it
- operating system (*kernel*): the kernel image, Barebox will load and run it
- root filesystem (*root*): used by the kernel as the root filesystem

The size and location of some of these partitions can be modified at run-time via the variable `nand_parts` in the `env/config` file. Here the user can increase the kernel partition, or add more partitions to the *free part* of the list.

However, two of the listed partitions are special: the location and size of the bootloader (*barebox*) partition and of the run-time environment (*bareboxenv*) partition.

These must be known soon after reset. So, we have a chicken/egg problem: to read the persistent environment, Barebox must know where the persistent environment is located. To do so, Barebox initially creates the *barebox* and *bareboxenv* partitions and after loading the persistent environment Barebox then adds the **remaining** partitions based on the `nand_parts` variable.

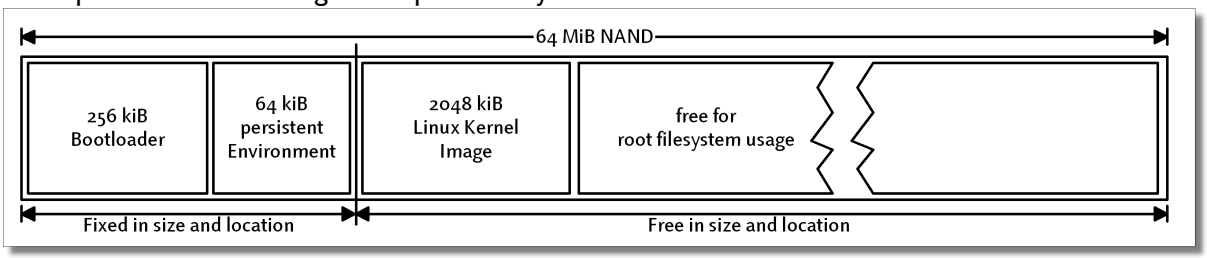
This handling implies the internally registered partitions for Barebox and the persistent environment must be the same in size and location as the partitions described in the `nand_parts` variable.

This then means, if one would like to change the size of the *barebox* or the *bareboxenv* partitions, she/he must change the platform source code **and** the `nand_parts` variable.

Here an example for a partition setup in the run-time environment:

```
nand_parts="256k(barebox),64k(bareboxenv),2048k(kernel),-(root)"
```

It corresponds to the following NAND partition layout:



This setup defines

- 256 kiB for the bootloader (*barebox*) at the beginning of the persistent media.
- 64 kiB for the persistent environment (*bareboxenv*) following the bootloader partition
- 2 MiB for the kernel (*kernel*)
- the remaining space on the persistent media for the root filesystem (*root*)

For the Mini2440 the platform source code is located in:

`platform-mini2440/build-target/barebox-<version>/arch/arm/boards/mini2440/mini2440.c`

and looks like this:

```
[...]
      /* ----------- add some vital partitions -------- */
   devfs_del_partition("self_raw");
   devfs_add_partition("nand0", 0x00000, 0x40000, PARTITION_FIXED, "self_raw");
```

```
   dev_add_bb_dev("self_raw", "self0");

   devfs_del_partition("env_raw");
   devfs_add_partition("nand0", 0x40000, 0x10000, PARTITION_FIXED, "env_raw");
   dev_add_bb_dev("env_raw", "env0");
[...]
```

Please ensure after changing any of the "Fixed in size and location" partitions that Barebox is re-compiled and re-flashed to keep the compiled-in environment in sync with the platform source code.

Also consider: for the partitions that are free in size and location, you can change these settings at run-time and store it to the persistent environment. But, if this persistent environment gets lost Barebox will default to the compiled-in environment. If this compiled-in environment has different partition sizes and locations, error messages will occur. This is because reading from partitions with wrong settings in size and location will fail.

So, the best procedure is to change the compiled-in environment to ensure the partition layout is always consistent, even if the modified persistent environment gets lost.

8.22 NFS with PTXdist and Barebox

NFS stands for **N***etwork* **F***ile***S***ystem*. It is one way to export a local filesystem content via network to another host. In our case we can use it to simulate a root filesystem for our target. This simplifies application development for the Mini2440. We do not have to change anything locally on the Mini2440. As our target is using our local host's filesystem, we can also change everything locally and it will be visible at target's side instantly.

What we need at our host's side:

- NFS server support in our Linux kernel
- NFS utils
- the `portmap` tool

It depends on the distribution we are using, how to get and enable this feature on our host.

To get these components on a Ubuntu based system we just have to enter:

```
$ sudo apt-get install nfs-kernel-server nfs-common portmap
```

The last two may not be required for a NFS server, but if you decide to install a NFS client on the Ubuntu box or need to do any troubleshooting later, you will need them.

First thing is to enable the export of specific parts of the host's local filesystem. This should be done **very** carefully, because:

- we must export the required filesystem parts in a very unsafe manner
- typos in the export's controling file may results into very funny and confusing error messages

The "unsafe manner" means, we allow any *root* user on the remote system using our local filesystem also to be the root user on our host! This is a huge security hole, if more parts of the local filesystem gets exported than intended.

PTXdist creates one directory intended for NFS root filesystem export when building the board support package:

```
$  ls platform-mini2440/root
bin  dev  etc  home  lib  mnt  proc  root  run  sbin  sys  tmp  usr  var
```

If we are a little bit paranoid we should only export exactly this one directory. The best way to know what directory path has to be used to export exactly this directory we can follow these steps:

```
$ cd platform-mini2440/root
platform-mini2440/root$ pwd -P
/home/me/OSELAS.BSP-Pengutronix-Mini2440-2012.06.0/platform-mini2440/root
```

The path, the `pwd` command outputs is the one we should export at our local host and our Mini2440 must mount at runtime to get access to this directory.

Exporting this specific directory presumes the directory must already exist when the export is enabled. Exporting first and then building the board support package (which creates the directory at this point of time) does not work as expected. Every `ptxdist clean; ptxdist go` requires to re-export this directory.
To be less dependend on the build state of the board support package, the BSP's directory could be exported instead: `/home/me/OSELAS.BSP-Pengutronix-Mini2440-2012.06.0`

But step by step. *Export* first:

The file to be modified is the `/etc/exports`. This file is used by the NFS related tools to know what parts of the filesystem should be exported and how it should be exported.

In our case we add the following line to this file:

```
/home/me/OSELAS.BSP-Pengutronix-Mini2440-2012.06.0/platform-mini2440/root *(rw,no_root_squash,sync, ↵
    no_subtree_check)
```

What does it mean: First part of the line is the to be exported directory. Second part is, who is allowed to use this directory. In the line mentioned above the asterisk ('*') means *everybody*. Here we can use an IP range instead of an asterisk to make the security hole a little bit smaller. Or, even smaller by only allowing one IP address. The third part are the permissions the remote host has on our localy exported path. `rw` means read write permission. The opposite is `ro` which means read only. But in our case we need the `rw` in order to use the exported directory for our root filesystem requirements.

Note: Do not add any white space between the asterisk/IP range and the left parenthesis starting the permission or you will get confusing error messages.

Most of the time there is no user management at our targets. Or to be more precise: most of the time only one user is required to make the target work as expected. This is always the user *root*. To allow this root user at the target side to touch all files in our exported directory the parameter `no_root_squash` must be given. But be aware: In this case the remote root user is also the root user on our host! Do you feel the risk? ;-) That's why only to export the required directory and nothing more.

If the NFS service is not running on our host yet, it's now time to run it:

```
$ sudo /etc/init.d/nfs-kernel-server start
```

Note: This step can differ on other distributions.

If the NFS service is already running, we can force to export all directories listed in `/dev/exports` by running:

```
$ sudo exportfs -rv
```

To check the current export state we can simply run a:

```
$ sudo exportfs -v
/home/me/OSELAS.BSP-Pengutronix-Mini2440-2012.06.0/platform-mini2440/root
        <world>(rw,wdelay,no_root_squash,no_subtree_check)
```

Note: Sometimes the list of options differ from the options we give in `/etc/exports`. These options are the default settings the NFS service is used if not otherwise set.

<world> means here the asterisk ('*').

Now is everything prepared at the host's side. Next step is to instruct our Mini2440 to use the new host's feature. Refer section 6.4 how to do so. The variable `nfsroot` must be setup in accordance to the exported filesystem on our host.

Even if we export

```
/home/me/OSELAS.BSP-Pengutronix-Mini2440-2012.06.0
```

on our host (to avoid the `ptxdist clean; ptxdist go` issue mentioned in the WARNING above), our Mini2440 must always use the full path:

```
nfsroot=/home/me/OSELAS.BSP-Pengutronix-Mini2440-2012.06.0/platform-mini2440/root
```

After we have a working NFS system all that is required is a power-up on the Mini2440, connected to our host, to enable sharing of the root file system.

After login to our target we can now look at any of the files in `platform-mini2440/root`, for example:

On the server (host):

```
/home/me/OSELAS.BSP-Pengutronix-Mini2440-2012.06.0/platform-mini2440/root/etc/fstab
```

and on the target (Mini2440)

```
/etc/fstab
```

We should see that the contents of these files are the same. Also, we can change this file's contents from our target or from our host AND see the changes appear on the other device.

8.22.1 Troubleshooting

If towards the end of the kernel boot process you get this message:

```
VFS:  Unable to mount root via NFS, trying floppy
VFS:  Cannot open root device “”nfs or unknown-block(2,0)
```

Don't panic!

Have a look in `/var/log/syslog` on the server and if you see a message like this:

```
Jun 15 21:30:09 my-desktop mountd[1716]: refused mount request from 192.168.1.230 for
/home/me/OSELAS.BSP-Pengutronix-Mini2440-2012.06.0/platform-mini2440/root (/): not exported
```

Have a **real close** look and see that you haven't made a mistake. Check again the directory the server exports and also check the content of the `nfsroot` variable. The start of the path must match exactly. Any typo will result into this kind of failure.

8.23 Using a Foreign Toolchain

It is possible to use a different toolchain than the recommended OSELAS.Toolchain. But to make sure the result will work on the Mini2440, some preconditions must be met.

The Mini2440 uses a Samsung S3C2440 processor. This processor comes with a so called *ARMv4 core* (also called *the architecture*). This means some internal features this processor supports and - much more important - the command set it understands.

So, an important precondition to use a foreign toolchain is, it must generate code only for this *ARMv4 core*. There are always two ways to ensure the compiler generates code that matches the *architecture*:

- the default settings are matching the *ARMv4 core architecure*
- a compile parameter temporarely switches to this *ARMv4 core architecure*

The first way was set, when the toolchain was built. In the case of our OSELAS.Toolchain we configure the compiler with:

`--with-arch=armv4t`

which results into the default compiler option for code generation if not otherwise specified later on.

The second way can be archived by using the compiler parameter `-march=armv4t`. This will switch the code generation to the *ARMv4 core architecure* for the run of the compiler.

8.23.1 Discovering Toolchain's Compiler Defaults

Two ways do exist to get the default settings out of the toolchain's compiler.

8.23.1.1 By Preprocessor Macros

Run the following command and examine the result:

```
$ echo "" | arm-linux-gcc -E -dM - | grep __ARM_ARCH_
#define __ARM_ARCH_4T__ 1
```

If the listed **__ARM_ARCH_** macro contains higher architecture numbers than the shown **4** this toolchain cannot be used for the Mini2440.

Currently known **not** working architectures are:

- __ARM_ARCH_5__
- __ARM_ARCH_5E__
- __ARM_ARCH_5T__
- __ARM_ARCH_5TE__
- __ARM_ARCH_5TEJ__
- __ARM_ARCH_6__
- __ARM_ARCH_6J__
- __ARM_ARCH_6K__

- __ARM_ARCH_6Z__
- __ARM_ARCH_6ZK__
- __ARM_ARCH_6T2__
- __ARM_ARCH_7M__
- __ARM_ARCH_7A__

8.23.1.2 By Parameter Listing

More default options can be read back from the toolchain's compiler in this way:

```
$ touch test.c
$ arm-linux-gcc -S --verbose-asm
```

The result in `test.s` will look like this:

```
   .arch armv4t
   .fpu softvfp
   .eabi_attribute 20, 1
   .eabi_attribute 21, 1
   .eabi_attribute 23, 3
   .eabi_attribute 24, 1
   .eabi_attribute 25, 1
   .eabi_attribute 26, 2
   .eabi_attribute 30, 6
   .eabi_attribute 18, 4
   .file "test.c"
@ GNU C (ctng-1.6.1) version 4.4.3 (arm-none-linux-gnueabi)
@  compiled by GNU C version 4.3.0 20080428 (Red Hat 4.3.0-8), GMP version 4.3.1, MPFR version 2.4.2-p2.
@ GGC heuristics: --param ggc-min-expand=98 --param ggc-min-heapsize=128213
@ options passed:  test.c -march=armv4t -mtune=arm920t -mfloat-abi=soft
@ -fverbose-asm
@ options enabled:  -falign-loops -fargument-alias -fauto-inc-dec
@ -fbranch-count-reg -fcommon -fearly-inlining
@ -feliminate-unused-debug-types -ffunction-cse -fgcse-lm -fident
@ -finline-functions-called-once -fira-share-save-slots
@ -fira-share-spill-slots -fivopts -fkeep-static-consts
@ -fleading-underscore -fmath-errno -fmerge-debug-strings
@ -fmove-loop-invariants -fpeephole -freg-struct-return -fsched-interblock
@ -fsched-spec -fsched-stalled-insns-dep -fsigned-zeros
@ -fsplit-ivs-in-unroller -ftrapping-math -ftree-cselim -ftree-loop-im
@ -ftree-loop-ivcanon -ftree-loop-optimize -ftree-parallelize-loops=
@ -ftree-reassoc -ftree-scev-cprop -ftree-switch-conversion
@ -ftree-vect-loop-version -funit-at-a-time -fverbose-asm
@ -fzero-initialized-in-bss -mglibc -mlittle-endian -msched-prolog
@ -mthumb-interwork

@ Compiler executable checksum: cd60c209f399d02b2e94fe81b4cfa8ba

   .ident   "GCC: (ctng-1.6.1) 4.4.3"
   .section .note.GNU-stack,"",%progbits
```

The important line here is the one with the `-march=armv4t`. It shows the default setting for code generation.

8.23.2 Discovering Toolchain's Library Optimization

Beside the compiler defaults, also the basic C library coming with the toolchain is an important part. As it comes with the toolchain in a binary data format, the way it was compiled must also match our architecture.

```
$ readelf -A <toolchain-install-directory>/arm-none-linux-gnueabi/lib/libc-2.9.so
Attribute Section: aeabi
File Attributes
  Tag_CPU_name: "4T"
  Tag_CPU_arch: v4T
  Tag_ARM_ISA_use: Yes
  Tag_THUMB_ISA_use: Thumb-1
  Tag_ABI_PCS_wchar_t: 4
  Tag_ABI_FP_denormal: Needed
  Tag_ABI_FP_exceptions: Needed
  Tag_ABI_FP_number_model: IEEE 754
  Tag_ABI_align8_needed: Yes
  Tag_ABI_enum_size: int
```

The important information is the `Tag_CPU_arch: v4T` line which matches our architecture.

Note: running the command on the `libc-2.0.so` library is an example only. You can run it on any target library you find in your toolchain.

It is always possible to overwrite the architecture the compiler creates code for by command line parameters. But it only works for newly compiled code. It does not help for the basic C library when this component was compiled for a different architecture. In this case there is no way to use this toolchain.

8.23.3 BSP changes to use a foreign Toolchain

Each PTXdist based board support package is configured for a specific toolchain vendor, a specific compiler version, toolchain name and basic C library version. This is to ensure a user uses the same environment the release tests where made of and to get a reliable build.

Using another toolchain than the already defined one means, all these specified values must be changed. To do so, we run:

```
$ ptxdist platformconfig
 architecture --->
   toolchain  --->
      ()  check for specific toolchain vendor
      (4.4.3) check for specific gcc version
      (2.9) check for specific glibc version
      (arm-linux) gnu target
```

The settings above will switch off the vendor check and configures the board support package for a `gcc-4.4.3` with a `glibc-2.9` and a toolchain/compiler name of `arm-linux`.

After this change PTXdist is unable to find the toolchain by its own. In this case an additional step is required to prepare the board support package for the build:

```
$ ptxdist toolchain <toolchain-install-directory>/bin
```

After that, a regular build can happen.

Don't expect a successfull build after changing to a foreign toolchain. It is impossible to provide code that builds on any compiler and compiler releases and any C library in the wild.

8.24 Using automagically the correct PTXdist Revision

After following the releases of this Mini2440 BSP our harddisk may contain various Mini2440 BSP releases, various corresonding OSELAS.Toolchains revisions and PTXdist revisions. How to avoid to get confused in this situation?

One solution can be to do it in the same way as PTXdist itself remembers the platform, the current configuration and the toolchain to be used to build the project: by symbolic filesystem links.

Lets take a look into the project, after it is set up:

```
$ ls -lF
total 80
-rw-r--r-- 1 jb user   408 Dec 23 14:00 CONTRIBUTORS
-rw-r--r-- 1 jb user 18002 Dec 23 14:00 COPYING
-rw-r--r-- 1 jb user  5100 Feb 21 21:27 Changelog
-rw-r--r-- 1 jb user  4133 Feb 21 21:27 FAQ
-rw-r--r-- 1 jb user   177 Oct 21 12:35 README
drwxr-xr-x 3 jb user  4096 Feb 21 21:27 configs/
drwxr-xr-x 3 jb user  4096 Oct 21 12:35 documentation/
drwxr-xr-x 4 jb user  4096 Dec 23 14:03 local_src/
drwxr-xr-x 2 jb user  4096 Feb 21 22:01 platform-mini2440/
drwxr-xr-x 3 jb user  4096 Oct 21 12:35 projectroot/
drwxr-xr-x 2 jb user  4096 Feb 21 21:27 protocol/
drwxr-xr-x 2 jb user  4096 Feb 21 21:27 rules/
lrwxrwxrwx 1 jb user    62 Feb 21 22:01 selected_platformconfig -> configs/platform-friendlyarm-mini2440 ↵
    /platformconfig-NAND-128M
lrwxrwxrwx 1 jb user    17 Feb 21 22:01 selected_ptxconfig -> configs/ptxconfig
lrwxrwxrwx 1 jb user   121 Feb 21 22:01 selected_toolchain -> /opt/OSELAS.Toolchain-2011.11.1/arm-v4t- ↵
    linux-gnueabi/gcc-4.6.2-glibc-2.14.1-binutils-2.21.1a-kernel-2.6.39-sanitized/bin/
```

Note the three symbolic links `selected_platformconfig`, `selected_ptxconfig` and `selected_toolchain`. They point to the correct files and locations to build this project.

Why not do the same with the PTXdist tool itself?

```
$ ln -s /usr/local/lib/ptxdist-2012.06.0/bin/ptxdist .
```

This command will add a new link with the name `ptxdist`, pointing to the PTXdist revision to be used in this project. All we now need to do is to call the local link instead of the global PTXdist. This can be achived by using the trailing `./`

```
$ ./ptxdist do
$ ./ptxdist menuconfig
$ ./ptxdist images
```

You can imagine this will free us from thinking about the correct PTXdist revision on each build or configure step in each PTXdist project. Now, all projects and PTXdist revisions can co-exist and we can use them all at the same time.

And as we do not want to type longer commands than absolute required, we should shorten the symbolic link to p

```
$ ln -s /usr/local/lib/ptxdist-2012.06.0/bin/ptxdist p
```

Now, typing is less time-consuming:

```
$ ./p do
$ ./p menuconfig
$ ./p images
```

And if we also do not want to enter the trailing ./ again and again (we are so lazy, aren't we?) we could add the following function to our own environment in /.bashrc:

```
function p ()
{
    if [ -h ./p ]; then
        ./p ${@};
    else
        echo "No local PTXdist symlink found. Create the correct one first"
    fi
}
```

Now it looks like a regular command again:

```
$ p do
$ p menuconfig
$ p images
```

To avoid the trailing ./ we could also add the current directory to the search PATH. But to do so is a really bad idea. At least compiling the OSELAS.Toolchain fails with the current directory in the PATH. Use the function shown above. It also gives us a helpful error message instead silently searching for 'p' in our filesystem with different funny and confusing error messages at the end.

8.25 Thanks

A thank you goes to Dave Festing for fixing my English spelling and discussions about what to add to this manual. Many details are still missing.

Another thank you goes to Josef Holzmayr for the QML demo included herein.

9 Document Revisions

2011/05/07	Initial Revision
2011/05/10	Path to the public GIT repository fixed
2011/05/21	Add info about available kernel releases
2011/06/03	Spelling fixes all over the place
2011/06/05	Add info about ADC, key buttons and audio usage
2011/06/17	Add info how to use the Mini2440 as a USB gadget
2011/06/17	Add info how to update individual software parts
2011/06/19	Add info how to control the LCD backlight
2011/07/02	Add info about the predefined Qt enabled configuration
2011/07/02	Add info about environment variables used by Qt at runtime
2011/07/24	Add generic info about PTXdist itself
2011/08/02	Add more info how to adapt the touchscreen to make it work as expected
2011/08/12	Add info how to setup an NFS service on the development host
2011/08/13	Add info how Barebox handles the various environment sources
2011/08/13	Rename "Deploying the Mini2440" section to "Bring in the Bootloader Barebox" due to content change
2011/08/13	Add info about the various boot methods the Mini2440 supports and how to make use of them
2011/09/03	Add a hint about display sizes are now handled in the kernel
2011/12/23	Add information about the QML demo which is now present in the BSP
2012/02/21	Add how to use the FriendyARM toolchain to build the Mini2440 BSP
2012/02/21	Add some ideas how to handle the different revisions of PTXdist
	×

10 Getting help

Below is a list of locations where you can get help in case of trouble. For questions how to do something special with PTXdist or general questions about Linux in the embedded world, try these.

10.1 Mailing Lists

10.1.1 About PTXdist in Particular

This is an English language public mailing list for questions about PTXdist. See

http://www.pengutronix.de/mailinglists/index_en.html

on how to subscribe to this list. If you want to search through the mailing list archive, visit

http://www.mail-archive.com/

and search for the list *ptxdist*. Please note again that this mailing list is just related to the PTXdist as a software. For questions regarding your specific BSP, see the following items.

10.1.2 About Embedded Linux in General

This is a German language public mailing list for general questions about Linux in embedded environments. See

http://www.pengutronix.de/mailinglists/index_de.html

on how to subscribe to this list. Note: You can also send mails in English.

10.2 About Working on the Linux Kernel

The book *Linux Kernel in a Nutshell* from *Greg Kroah-Hartman*. Its online version can be read here:

http://www.kroah.com/lkn/

10.3 Chat/IRC

About PTXdist in particular

irc.freenode.net:6667

Create a connection to the **irc.freenode.net:6667** server and enter the chatroom **#ptxdist**. This is an English room to answer questions about PTXdist. Best time to meet somebody there is at European daytime.

10.4 The Web

Wilson Wingston Sharon made a tutorial for the Mini2440. To be found here:

http://wingston.workshopindia.com/wingz/embedded-programming-on-the-mini2440-using-ptxdist/

10.5 FriendlyARM Mini2440 specific Mailing List

oselas@community.pengutronix.de

This is a community mailing list open for everyone for all Mini2440's board support package related questions. Refer our page at

http://www.pengutronix.de/mailinglists/index_en.html

to subscribe to this mailing list.

Note: Please be aware that we cannot answer hardware only related questions on this list.

10.6 Commercial Support

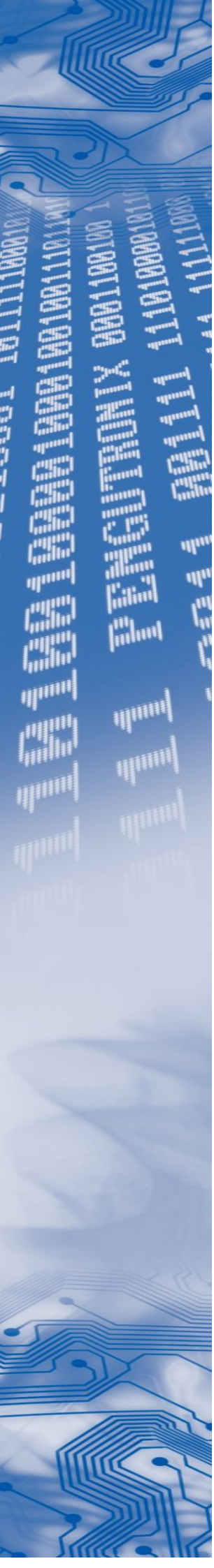
You can order immediate support through customer specific mailing lists, by telephone or also on site. Ask our sales representative for a price quotation for your special requirements.

Contact us at:

Pengutronix
Peiner Str. 6-8
31137 Hildesheim
Germany
Phone: +49 - 51 21 / 20 69 17 - 0
Fax: +49 - 51 21 / 20 69 17 - 55 55

or by electronic mail:

sales@pengutronix.de



This is a Pengutronix Quickstart Manual

Pengutronix e.K.
Peiner Str. 6-8
31137 Hildesheim
Germany
Phone: +49 - 51 21 / 20 69 17 - 0
Fax: +49 - 51 21 / 20 69 17 - 55 55