OSELAS.BSP( )
Phytec phyCORE-PXA270

Quick Start Manual

http://www.oselas.com

$Rev: 314 $ $Date: 2006-05-10 21:07:56 +0200 (Wed, 10 May 2006) $

Contents

1 PTXdist Installation

1.1 Building Blocks

The main tool of the OSELAS.BoardSupport() Package is PTXdist. So before starting any work we'll have to install PTXdist on the development host.

PTXdist consists of the following parts:

- **The `ptxdist` program**, which is installed on the development host during the installation process. `ptxdist` is called to trigger any actions, like building a software packet, cleaning a 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.

1.2 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.

1.3 Installation from the Sources

To install PTXdist, three archives have to be extracted:

- `ptxdist-0.10.4.tgz`, containing the software
- `ptxdist-patches-0.10.4.tar.gz`, containing all patches for upstream packets
- `OSELAS.BSP-phyCORE-PXA270-3.tar.gz`, containing the board support package (project) for the Phytec phyCORE-PXA270 board.

The PTXdist and patches packets have to be extracted into some temporary directory, 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
```

At this point we mount the CDROM with the PTXdist tar files and inflate the first two of them into the temporary directory.

```
~/local# mount /media/cdrom
~/local# tar -zxf /media/cdrom/
     OSELAS.BSP-phyCORE-PXA270-3/
     ptxdist-0.10.4.tgz
~/local# tar -zxf /media/cdrom/
     OSELAS.BSP-phyCORE-PXA270-3/
     ptxdist-0.10.4-patches.tar.gz
```

If everything goes well, we now have a PTXdist-0.10.4 directory, so we can change into it:

```
~/local# cd ptxdist-0.10.4
~/local/ptxdist-0.10.4# ls -l

total 168
-rw-r--r--   1 rsc ptx  1547 Jan 26 17:29 COMPILE-TEST
-rw-r--r--   1 rsc ptx 18361 Dec 27 12:46 COPYING
-rw-r--r--   1 rsc ptx  2084 Jan 31 08:20 CREDITS
-rw-r--r--   1 rsc ptx 41309 Feb  3 08:23 ChangeLog
drwxr-sr-x   3 rsc ptx  4096 Dec 27 12:46 Documentation
-rw-r--r--   1 rsc ptx    58 Dec 27 12:46 INSTALL
-rw-r--r--   1 rsc ptx  1275 Mar  8 18:05 Makefile
-rw-r--r--   1 rsc ptx  1216 Mar  8 18:03 Makefile.in
```

```
-rw-r--r--   1 rsc ptx  3415 Feb 19 22:58 README
-rw-r--r--   1 rsc ptx   949 Jan 26 17:29 README.Toolchains
-rw-r--r--   1 rsc ptx   835 Dec 27 12:46 SPECIFICATION
-rw-r--r--   1 rsc ptx  8927 Mar  1 07:36 TODO
-rw-r--r--   1 rsc ptx  1901 Jan 26 17:29 TOOLCHAINS
drwxr-sr-x   3 rsc ptx  4096 Mar  8 18:44 bin
drwxr-sr-x  11 rsc ptx  4096 Mar  8 19:36 config
-rwxr-xr-x   1 rsc ptx  2306 Mar  8 18:03 configure
drwxr-sr-x 106 rsc ptx  4096 Mar  5 16:12 patches
drwxr-sr-x   4 rsc ptx  4096 Dec 27 12:45 pending_patches
drwxr-sr-x  35 rsc ptx  4096 Mar  4 16:49 projects
drwxr-sr-x   4 rsc ptx 20480 Mar  8 20:03 rules
drwxr-sr-x   7 rsc ptx  4096 Mar  8 18:07 scripts
drwxr-sr-x   3 rsc ptx  4096 Feb 11 13:42 tests
```

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

```
~/local/ptxdist-0.10.4# ./configure
checking version=0.10.4
checking prefix=/usr/local
checking topdir=/home/username/tmp/ptxdist-0.10.4
checking instdir=/usr/local/lib/ptxdist-0.10.4
creating Makefile
creating rules/Kconfig
```

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.

! In this early PTXdist version not all tests are implemented in the `configure` script yet. So if something goes wrong or you don't understand some error messages send a mail to `support@pengutronix.de` and help us improve the tool.

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

```
~/local/ptxdist-0.10.4# 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 root:

```
~/local/ptxdist-0.10.4# su
[enter root password]
/home/username/local/ptxdist-0.10.4# make install
[...]
```

If we don't have root access to the machine it is also possible to install 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

```
~/local/ptxdist-0.10.4# cd
~# rm -fr local/ptxdist-0.10.4
```

2 Toolchain

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 the machine. 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 optimized 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.

2.1 Using Existing Toolchains

If a toolchain is already installed which is known to be working, the toolchain building step with PTXdist may be omitted. We have to make sure that the PATH environment variable points to the directory containing the toolchain components.

! The projects shipped with PTXdist have been tested with the 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 work properly in a cross environment.

2.2 Toolchain Building

PTXdist has several example projects included to build toolchains for different architectures. To find out which example projects are being shipped with PTXdist we use the `ptxdist projects` command.

As toolchain projects always start with `toolchain_`, we can restrict the output to only showing the toolchains:

```
~# ptxdist projects | grep toolchain_
toolchain_arm-softfloat-linux-gnu-4.0.2_glibc_2.3.6_linux_2.6.14
toolchain_i586-unknown-linux-gnu-4.0.2_glibc_2.3.6_linux_2.6.14
toolchain_powerpc-unknown-linux-gnu-4.0.2_glibc-2.3.6_linux_2.6.13
```

PTXdist toolchains, internally built with crosstool (a community provided script to build cross toolchains in a unified way), by default are being installed into the standard directory `/opt/ptxdist-0.10.4/<gcc-glibc-version>/<gnu-target>`.

So for example for the gcc-4.0.2 and glibc-2.3.6 based ARM toolchain with software floating point support mentioned above, the toolchain directory shall be `/opt/ptxdist-0.10.4/gcc-4.0.2-glibc-2.3.6/arm-softfloat-linux-gnu`.

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

!

Usually the `/opt` directory is not world writable. So in order to build our toolchain into that directory we need to use a root account to change the permissions so that the user can write (`mkdir /opt/ptxdist-0.10.4; chown <username> /opt/ptxdist- 0.10.4; chmod a+rwx /opt/ptxdist-0.10.4`).

To compile and install the toolchains we have to clone one of the predefined PTXdist toolchain projects. In this book we will build all of our stuff in $HOME/work. If this directory does not exist yet, we create it and change into it with

```
~# cd
~# mkdir work
~# cd work
```

Now we clone the ARM toolchain project for the phyCORE-PXA270. "Cloning" means that we create a local working copy of the project shipped with PTXdist:

```
~/work# ptxdist clone
     toolchain_arm-softfloat-linux-gnu-4.0.2_glibc_2.3.6_linux_2.6.14
     cross-toolchain
```

The first argument to the `ptxdist clone` command is the project to be cloned, the second one is the name of our working copy.

Now that we have changed into the toolchain project directory we can order PTXdist to build our toolchain:

```
~/work# cd cross-toolchain
~/work/cross-toolchain# 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 a cross toolchain is something like around 30 minutes up to one hour. Another possibility is to read the next chapters of this manual, to find out how to start a new project.

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

3 Building the ”light” Image for phyCORE-PXA270

3.1 Preparing a Build

After having successfully built a toolchain for the target CPU, we can proceed with building our first ”project”. Following the PTXdist nomenclature, a ”project” is a configuration that specifies which ”packets” (programs) should go into a root filesystem.

In order to build a project we have to unpack the OSELAS.BSP-phyCORE-PXA270-3 for the phyCORE-PXA270:

```
~$ tar -zxf OSELAS.BSP-phyCORE-PXA270-3.tar.gz
~$ cd OSELAS.BSP-phyCORE-PXA270-3
```

In a PTXdist project there always exists a `ptxconfig` file which defines the ”schedule”, telling the build system which packets to build and which options to use. As the phyCORE-PXA270 development kit is available in versions with and without a display, the OSELAS.BSP-phyCORE-PXA270-3 contains two ptxconfig files. So for users who happen to have a kit without a display it is not necessary to build a full blown system including an x.org server.

So what we have to do first is to select one of the ptxconfig files, `ptxconfig.light`:

```
~/OSELAS.BSP-phyCORE-PXA270-3$ ptxdist select ptxconfig.light
```

The `select` command links the `ptxconfig.light` file to `ptxconfig`, which is the default file name for PTXdist project configuration files. Now for PTXdist it looks like there is only one configuration, and that's what we want.

Before we can actually start compiling our project, we'll have to specify which toolchain shall be used:

```
~/OSELAS.BSP-phyCORE-PXA270-3$ ptxdist toolchain
     /opt/ptxdist-0.10.4/gcc-4.0.2-glibc-2.3.6/
     arm-softfloat-linux-gnu/bin
```

3.2 Compiling the Root Filesystem

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

```
~/OSELAS.BSP-phyCORE-PXA270-3$ ptxdist go
```

PTXdist does now automatically find out from the `ptxconfig` file which packages belong to the projects 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 packets and brings them into the correct order.

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

There are two things which are different between the ”final” root filesystem to be flashed into the embedded system and the `root/` tree: the device nodes are missing[1] and the access permissions are incorrect[2].

3.3 Building a Flash Image

PTXdist can build a flash image from the `root/` tree. As all necessary parameters for the phyCORE-PXA270 are configured in the `ptxconfig` file, all we need to do is to run

```
~/OSELAS.BSP-phyCORE-PXA270-3$ ptxdist images
```

Now the `images/` directory contains a JFFS2 image (`root.jffs2`).

So after running ”`ptxdist go`” and ”`ptxdist images`”, we generally find the following directories in the project workspace:

- in `root/` a complete root filesystem to run on our target
  (to be used as an NFS based filesystem)
- in `images/` everything we need on our target packetised for easy handling
  (to be used for running the target stand alone)
- in `local/` a build environment to be used for external software projects

[1]There is no way to build them as a normal user, and PTXdist should never be run with root permissions.
[2]It is not possible to chown files for example to root.

4 Booting Linux

Now that there is a root filesystem in our workspace we'll have to make it visible to the phyCORE-PXA270. There are two possibilities to do this:

1. Booting from the development host, via network.
2. Making the root filesystem persistent in the onboard flash.

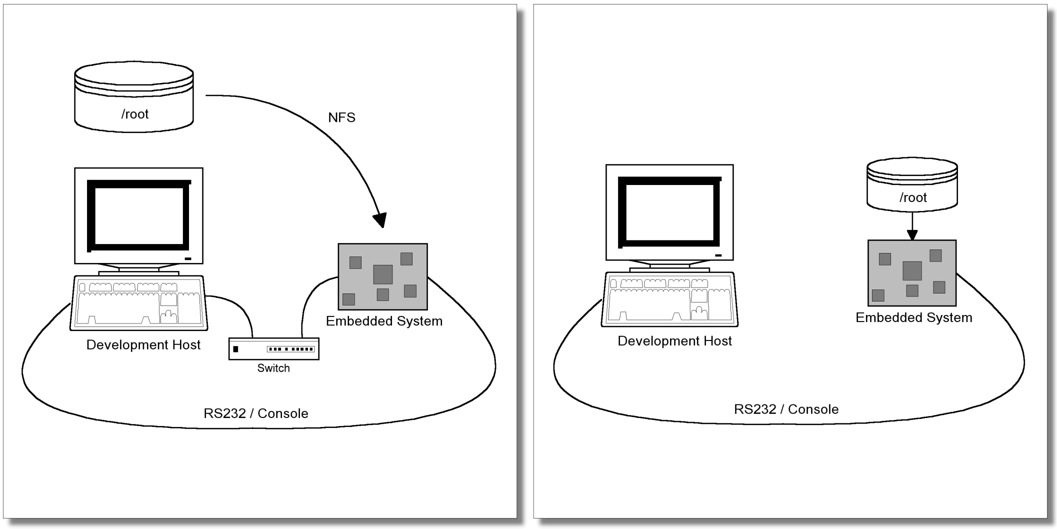

Figure 4.1: *Booting the root filesystem, built with PTXdist, from the host via network and from flash.*

Figure 4.1 shows both methods. On the left side the development host is connected to the phyCORE-PXA270 with a serial nullmodem cable and via ethernet; the embedded board boots into the bootloader, then issues a TFTP request on the network and boots the kernel from the TFTP server on the host. Then, after decompressing the kernel into the RAM and starting it, the kernel mounts it's root filesystem via NFS from the original location of the root/ directory in our PTXdist workspace.

The other way is to provide all needed components to run on the traget itself. The Linux kernel and the root filesystem is persistent in target's flash. This means the only connection needed is the nullmodem cable to see what is happen on our target.

This chapter describes how to setup our target with features supported by PTXdist to simplify this challange.

4.1 Target Side Preparation

The phyCORE-PXA270 uses U-Boot as its bootloader. U-Boot can be customised with environment variables to support any boot constellation. OSELAS.BSP-phyCORE-PXA270-3 comes with a predefined environment setup to easily bring up the phyCORE-PXA270.

4.2 Default U-Boot environment

This is the default U-Boot environment, saved onto the target when "ptxdist test setenv" was run in advance. It is a general purpose environment and works for running with network or as standalone (see next chapter for explanation).

Variable	Value	Meaning
ipaddr	ip address	IP of the target (if no DHCP is used). Will be configured with `ptxdist boardsetup`
serverip	ip address	IP of the server, needed for loading the kernel image and mounting the NFS root filesystem (if no DHCP is used). Will be configured with `ptxdist boardsetup`
gatewayip	ip address	Gateway to use (if no DHCP is used). Will be configured with `ptxdist boardsetup`
netmask	network mask	Netmask to use (if no DHCP is used). Will be configured with `ptxdist boardsetup`

continued on next page

continued from previous page

Variable	Value	Meaning
mtdids	nor0=phys_mapped_flash	Where to load kernel images in the case flash is used as source (standalone mode)
mtdparts	mtdparts=phys_mapped_flash:256k(u-boot)ro,4096k(system),-(root)	Description how the flash memory is partitioned. Note the double *mtdparts* are required!
uimage	uImage-pcm027	This filename will be used when booting remotely
jffs2	root-pcm027.jffs2	This file will be used when updating the rootfs in flash (see variable prg_jffs2).
bootargs_base	setenv bootargs console=ttyS0,115200	This is one part of the kernel parameters used by all configurations. It defines the serial console to be used and its settings.
bootargs_mtd	setenv bootargs $(bootargs) $(mtdparts)	This adds the flash partitioning information to the kernel parameters.
bootargs_flash	setenv bootargs $(bootargs) root=/dev/mtdblock2 rootfstype=jffs2	Defines the flash specific bootargs to kernel for standalone mode
bootargs_nfs	setenv bootargs $(bootargs) root=/dev/nfs ip=dhcp nfsroot=$(serverip):$(nfsrootfs),v3,tcp	Defines the NFS specific bootargs to kernel for remote mode
nfsrootfs	path to rootfs	This directory is used in remote mode. Will be configured with `ptxdist boardsetup`
bootcmd_flash	run bootargs_base bootargs_mtd bootargs_flash; bootm 0x40000	This defines a command that combines the kernel parameters and boots a kernel from flash
bootcmd_net	run bootargs_base bootargs_mtd bootargs_nfs; tftpboot 0xa0200000 $(uimage);	This defines a command that combines the kernel parameters and boots a kernel from network
bootcmd	run bootcmd_flash	Defines the command that should run after powerup

continued on next page

continued from previous page

Variable	Value	Meaning
prg_kernel	tftpboot 0xa0200000 $(uimage); erase nor0,1; cp.b 0xa0200000 0x40000 $(filesize)	This command is for convenience only and replaces the kernel image in flash
prg_jffs2	tftpboot 0xa0200000 $(jffs2); erase nor0,2; cp.b 0xa0200000 0x440000 $(filesize)	This command is for convenience only and replaces the root filesystem image in flash

Note: All identifiers in U-Boot are variables. But some contain other commands and variable references. To run their contents, we can simple enter `run variablename`. If we do so, variable references get replaced by their contents and commands are run.

Usually the environment doesn't have to be set manually on our target. PTXdist comes with an automated setup procedure to achieve a correct environment on the target.

Due to the fact some of the values of these U-Boot's environment variables must meet our local network environment and development host settings we have to define them prior to running the automated setup procedure.

Note: At this point of time it makes sense to check if the serial connection is already working, because it is essential for any further step we will do.
We can try to connect to the target with our favorite terminal application (`minicom` or `kermit` for example). With a powered target we identify the correct physical serial port and ensure that the communication is working.
Make sure to leave this terminal application to unlock the serial port prior to the next steps.

To set up development host and target specific value settings we run the command

```
~# ptxdist boardsetup
```

We navigate to "Network Configuration" and replace the default settings with our local network settings. In the next step we also should check if the "Host's Serial Configuration" entries meet our local development host settings. Especially the "serial port" must correspond to our real physical connection. At least - to make the automated setup procedure work - the "uboot prompt" entry must be `uboot>` .

"Exit" the dialouge and and save your new settings.

The command

```
~# ptxdist test setenv
```

now will automatically set up a correct default environment on the phyCORE-PXA270.
It should output a line like this when it was successfull:

```
u-boot:  flashing standard environment PASS
```

Note: If it fails, reading `test.log` will give further information about why it has failed.

We now must restart the phyCORE-PXA270 to activate the new environment settings. Then we should run the `ping` command on the target's ip address to check if the network settings are working correctly on the target.

4.3 Remote-Booting Linux

The first method we probably want to try after building a root filesystem is the network-remote boot variant. All we need is a network interface on the embedded board and a network aware bootloader which can fetch the kernel from a TFTP server.

The network boot method has the advantage that we don't have to do any flashing at all to "see" a file on the target board: All we have to do is to copy it to some location in the `root/` directory and it simply "appears" on the embedded device. This is especially helpful during the development phase of a project, where things are changing frequently.

4.3.1 Development Host Preparations

On the development host a TFTP server has to be installed and configured. The exact method to do this is distribution specific; as the TFTP server is usually started by one of the inetd servers, the manual sections describing `inetd` or `xinetd` should be consulted.

Usually TFTP servers are using the `/tftpboot` directory to fetch files from, so if we want to push kernel images to this directory we have to make sure we are able to write there. As the access permissions are normally configured in a way to let only user **root** write to `/tftpboot` we have to gain access; a safe method is to use the `sudo(8)` command to push our kernel:

```
~# sudo cp images/linuximage /tftpboot/uImage-pcm027
```

The NFS server is not restricted to a certain filesystem location, so all we have to do on most distributions is to configure `/etc/exports` and export our root filesystem to the embedded network. In this example file the whole work directory is exported, and the "lab network" between the development host is 192.168.23.0, so the IP addresses have to be adapted to the local needs:

```
/home/<user>/work 192.168.23.0/255.255.255.0(rw,no_root_squash,sync)
```

Note: Replace `<user>` with your home directory name.

4.3.2 Preparations on the Embedded Board

We already provided the phyCORE-PXA270 with the default environment at page 15. So there is no additional preparation required here.

4.3.3 Booting the Embedded Board

The default environment coming with the OSELAS.BSP-phyCORE-PXA270-3 has a predefined script for booting from NFS. To use it, we can simple enter

```
uboot> run bootcmd_net
```

This command should boot phyCORE-PXA270 into the login prompt.

As U-Boot automatically runs the `bootcmd` environment variable as a script after power-on, we set this variable to start from NFS automatically:

```
uboot> setenv bootcmd 'run bootcmd_net'
```

After the next reset or powercycle of the board it should boot the kernel from the TFTP server, start it and mount the root filesystem via NFS.

Note: The default login account is `root` with an empty password.

4.4 Stand-Alone Booting Linux

Usually, after working with the NFS-Root system for some time, the rootfs has to be made persistent in the onboard flash of the phyCORE-PXA270, without requiring the network infrastructure any more. The following sections describe the steps necessary to bring the rootfs into the onboard flash.

Only for preparation we need a network connection to the embedded board and a network aware bootloader which can fetch any data from a TFTP server.

After preparation is done, the phyCORE-PXA270 can work independently from the development host. We can "cut" the network (and serial cable) and the phyCORE-PXA270 will continue to work.

4.4.1 Development Host Preparations

If we already booted the phyCORE-PXA270 remotly (as described in the privious section) all of the development host preparations are done.

If not then on the development host has a TFTP server to be installed and configured. The exact method to do this is distribution specific; as the TFTP server is usually started by one of the inetd servers, the manual sections describing `inetd` or `xinetd` should be consulted.

Usually TFTP servers are using the `/tftpboot` directory to fetch files from, so if we want to push data files to this directory we have to make sure we are able to write there. As the access permissions are normally configured in a way to let only user **root** write to `/tftpboot` we have to gain access.

4.4.2 Preparations on the Embedded Board

To boot phyCORE-PXA270 stand-alone anything needed to run a Linux system must be locally accessible. So at this point of time we must replace any current content in phyCORE-PXA270's flash memory. To simplify this, OSELAS.BSP-phyCORE-PXA270-3 comes with an automated setup procedure for this step.

To use this procedure run the command

```
~# ptxdist test flash
```

Note: This command requires a serial and a network connection. The network connection can be cut afterwards this step.

This command will automatically write a root filesystem to the correct flash partition on the phyCORE-PXA270. It only works, if we priviously setup the environment variables successfully (described at page 15).
The command should outputs a line like this when it was successfull:

```
u-boot:  flashing root image PASS
```

Note: If it fails reading `test.log` will give further information about why it was failing.

4.4.3 Booting the Embedded Board

To check everything went successfully up to here, we can run the *boot* test.

```
~# ptxdist test boot
```

This will check if the environment settings and flash partitioning works as expected, so the target comes up in stand alone mode up to the login prompt.

To do it manually the default environment coming with the OSELAS.BSP-phyCORE-PXA270-3 has a predefined script for booting stand-alone. To use it, we can simple enter

```
uboot> run bootcmd_flash
```

This command should boot phyCORE-PXA270 into the login prompt.

As U-Boot automatically runs the `bootcmd` environment variable as a script after power-on, we set this variable to start from NFS automatically:

```
uboot> setenv bootcmd 'run bootcmd_flash'
```

After the next reset or powercycle of the board it should boot the kernel from the flash, start it and mount the root filesystem also from flash.

Note: The default login account is `root` with an empty password.

5 Accessing Peripherals

Phytec's phyCORE-PXA270 starter kit consists of the following individual boards:

1. The phyCORE-PXA270 module itself, containing the PXA270, RAM, flash, the GPIO expander chip and several other peripherals.
2. The starter kit baseboard.
3. A GPIO breakout board.

To achieve maximum software re-use, the Linux kernel offers a sophisticated infrastructure, layering software components into board specific parts. The OSELAS.BSP( ) tries to modularize the kit features as far as possible; that means that when a customized baseboards or even customer specific module is developed, most of the software support can be re-used without error prone copy-and-paste. So the kernel code corresponding to the boards above can be found in

1. `arch/arm/mach-pxa/pcm027.c` for the module
2. `arch/arm/mach-pxa/pcm027-baseboard.c` for the baseboard
3. `arch/arm/mach-pxa/pcm027-gpio-expander.c` for the breakout board.

In fact, software re-use is one of the most important features of the Linux kernel and especially of the ARM port, which always had to fight with an insane number of possibilities of the System-on-Chip CPUs.

! Note that the huge variety of possibilities offered by the phyCORE modules makes it difficult to have a completely generic implementation on the operating system side. Nevertheless, the OSELAS.BSP( ) can easily be adapted to customer specific variants. In case of interest, contact the Pengutronix support (support@pengutronix.de) and ask for a dedicated offer.

The following sections provide an overview of the supported hardware components and their operating system drivers.

5.1 NOR Flash

Linux offers the Memory Technology Devices Interface (MTD) to access low level flash chips, directly connected to a SoC CPU.

Older versions of the Linux kernel had separate mapping drivers for each board, specifying the flash layout in a driver. Modern kernels offer a method to define flash partitions on the kernel command line, using the "mtdparts" command line argument:

```
mtdparts=phys_mapped_flash:256k(u-boot)ro,4096k(system),-(root)
```

This line, for example, specifies several partitions with their size and name which can be used as /dev/mtd0, /dev/mtd1 etc. from Linux. Additionally, this argument is also understood by reasonably new U-Boot bootloaders, so if there is any need to change the partitioning layout, the U-Boot environment is the only place where the layout has to be changed. In this section we assume that the standard configuration delivered with the OSELAS.BSP-phyCORE-PXA270-3 is being used.

From userspace the flash partitions can be accessed as

- /dev/mtdblock0 (U-Boot partition)
- /dev/mtdblock1 (Linux kernel)
- /dev/mtdblock2 (Linux rootfs partition)

Only `/dev/mtdblock2` has a filesystem, so the other partition cannot be mounted into the rootfs. The only way to access them is by pushing a prepared flash image into the corresponding `/dev/mtd` device node.

5.2 PWM Units

The PXA270 has four PWM units which can be programmed individually. However, as the phyCORE-PXA270 has some hardware restrictions, not all of them can be used under all circumstances:

- PWM#0 is used for LCD Backlight brightness (see section 5.8)
- PWM#1 is used to controll the motor speed on the GPIO expander board
- PWM#2 is not available
- PWM#3 is not available

To use the PWM units we have to make sure that the PWM driver is loaded (which is automatically done when using the predefined OSELAS.BSP-phyCORE-PXA270-3):

```
~# lsmod
Module                  Size  Used by
nls_iso8859_1           3936  0
nls_cp437               5600  0
vfat                   10112  0
fat                    46620  1 vfat
usb_storage            32580  0
pcm027can               2848  0
sja1000                 6176  1 pcm027can
can_raw                 4608  0
can                     5516  2 sja1000,can_raw
rtc_pcf8563             5900  0
rtc_dev                 5160  0
rtc_core                7156  2 rtc_pcf8563,rtc_dev
max7301                 4768  0
pxa2xx_spi             13120  0
pxa27x_gpioevent        3648  0
pxa27x_pwm              5568  0
ohci_hcd               16612  0
usbhid                 32868  0
```

If the driver is not in place we have to load it with

```
~# modprobe pxa27x_pwm
```

The driver uses sysfs entries for communication with userspace; so in order to control a PWM unit we have to echo plain ASCII numbers into the corresponding sysfs entries.

For each PWM unit there are three entries:

- `/sys/class/pwm/pwm?/period`
  This entry can be used to change the period of the PWM signal. The unit of the values being written here is Microseconds, and valid numbers are 1 . . . 5000 (1 us . . . 5 ms)
- `/sys/class/pwm/pwm?/duty`
  The duty percentage is being written into this entry. The unit of the values is percent, using one position after the decimal point. Valid numbers are 0 . . . 1000 (0.0% . . . 100.0%)

- `/sys/class/pwm/pwm?/active`
  To activate the PWM, a '1' has to be written into this entry. By default the driver comes up with this value being '0', so the corresponding PWM pin is always at low level.

Note: Replace the '**?**' in each entry by the corresponding PWM unit number 0 . . . 3.

5.3 GPIO

Like most modern System-on-Chip CPUs, the PXA270 has several GPIO pins, some of which can be used for general purpose operations. If the generic gpio driver is loaded it offers a special sysfs entry that can be used to map a pin for userspace usage:

```
˜# echo 19:out:lo > /sys/class/gpio/map_gpio
```

A mapping command consists of the GPIO pin number, corresponding to the datasheet, plus the direction (out or in) and, in case of an output, the initial level (hi or lo).

To find out which GPIO pins have been mapped by which drivers we can have a look at the output of

```
˜# cat /proc/gpio
GPIO   POLICY       DIRECTION    OWNER
 19:   user space   output       kernel
```

If the breakout board is installed, GPIO19 can be used to control the motor direction of the small DC motor. In order to set the direction from the Linux command line we issue:

```
˜# echo 1 > /sys/class/gpio/gpio19/level
```

or 0 to change to the other direction. Note that this method is not very fast, so for quickly changing GPIOs it is still necessary to write a driver. The method works fine for example to influence an LED directly from userspace.

! Note that this interface is a temporary one. The Open Source Automation Development Lab (OSADL) is working on an "Industrial I/O" driver framework which will probably superseed this interface in the future.

5.4 GPIO Events

Some GPIOs are able to issue an interrupt. For example, on the breakout board the following pins offer this feature:

- GPIO14 is used as Key1 event input
- GPIO86 is used as Key2 event input
- GPIO87 is used as Key3 event input
- GPIO91 is used as light sensor event input

To read back the currently collected events simple read from
`/sys/class/gpio_events/gpio_event??/event`
(where `??` are the numbers 14, 86, 87 or 19 in the example above). It returns the number (in ASCII) of events since the last read. If no event arrieved since the last read it returns an empty file. Each read access resets the event counter.

! Note that this interface is a temporary one. The Open Source Automation Development Lab (OSADL) is working on an ”Industrial I/O” driver framework which will probably superseed this interface in the future.

5.5 CAN Bus

The phyCORE-PXA270 has one SJA1000 based CAN controller, which is supported by drivers using the (currently work-in-progress) proposed Linux standard CAN framework ”Socket-CAN”. Using this framework, CAN interfaces can be programmed with the BSD socket API.

! The Socket-CAN API is still work in progress and was not submitted to the upstream kernel maintainers yet. Although we think that the final API will be very similar to what we have now, be prepared that the API can break at any time without notice.

5.5.1 Socket-CAN

The CAN (Controller Area Network[1]) bus offers a low-bandwidth, prioritised message fieldbus for communication between microcontrollers. Unfortunately, CAN was not designed with the ISO/OSI layer model in mind, so most CAN APIs available throughout the industry don't support a clean separation between the different logical protocol layers, like for example known from ethernet.

The *Socket-CAN* framework for Linux extends the BSD socket API concept towards CAN bus. It consists of

- a core part (can.ko)
- several protocol drivers (can_raw.ko, maybe other protocols)
- chip drivers (e. g. sja1000.ko, nioscan.ko etc.)

So in order to start working with CAN interfaces we'll have to make sure all necessary drivers are loaded.

5.5.2 Starting and Configuring Interfaces from the Command Line

If all drivers are present in the kernel, "ifconfig -a" shows which network interfaces are available; as Socket-CAN chip interfaces are normal Linux network devices (with some additional features special to CAN), not only the ethernet devices can be observed but also CAN ports.

For this example, we are only interested in the first CAN port, so the information for can0 looks like

```
~# ifconfig can0
can0      Link encap:UNSPEC  HWaddr
00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          NOARP  MTU:8  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:90 Base address:0x8400
```

The output contains the usual parameters also shown for ethernet interfaces, so not all of these are necessarily relevant for CAN (for example the MAC address). These parameters contain useful information:

[1] ISO 11898/11519

Field	Description
can0	Interface Name
NOARP	CAN cannot use ARP protocol
MTU	Maximum Transfer Unit, always 8
RX packets	Number of Received Packets
TX packets	Number of Transmitted Packets
RX bytes	Number of Received Bytes
TX bytes	Number of Transmitted Bytes
errors...	Bus Error Statistics

Inferfaces shown by the ”ifconfig -a” command can be configured with canconfig. This command adds CAN specific configuration possibilities for network interfaces, similar to for example ”iwconfig” for wireless ethernet cards.

The baudrate for can0 can now be changed:

```
~# canconfig can0 baudrate 250
```

and the interface is started with

```
~# ifconfig can0 up
```

If the interface happens to fall into ”bus off” mode, `canconfig` can also be used to bring the interface back into ”normal” mode:

```
~# canconfig can0 mode start
```

More details about the `ioctl()` commands used by `canconfig` can be found in the sourcecode of the `canconfig` utility (`canutils/canconfig.c` in the canutils source code package).

5.5.3 Using the CAN Interfaces from the Command Line

After successfully configuring the local CAN interface and attaching some kind of CAN devices to this physical bus, we can test this connection with command line tools.

The tools `cansend` and `candump` are dedicated to this purpose.

To send a simple CAN message with ID 0x20 and one data byte of value 0xAA just enter:

```
~# cansend can0 --identifier=0x20 0xAA
```

To receive CAN messages run the `candump` command:

```
˜# candump can0
interface = can0, family = 29, type = 3, proto = 0
<0x020> [1] aa
```

The output of `candump` shown in this example was the result of running the `cansend` example above on a different machine.

See cansend's and candump's manual pages for further information about using and options.

5.5.4 Programming CAN Interfaces in C

With Socket-CAN we can use the standard Berkeley Socket Interface for sending and receiving CAN messages. The first step is to get a socket, using the socket() function:

```
    int socket(int domain, int type, int protocol);
```

We have to prepare the protocol family (domain), which is ”Protocol Family CAN (PF_CAN), the type (a raw socket, SOCK_RAW) and the protocol (CAN_PROTO_RAW) for the call to the socket() function:

```
    domain = PF_CAN;
    type = SOCK_RAW;
    protocol = CAN_PROTO_RAW;

    int sockdf;
    sockfd = socket(PF_CAN, SOCK_RAW, CAN_PROTO_RAW);
```

If everything succeeds, a filedescriptor for the new socket will be returned; on error, the socket() function returns -1.

Now we have to *bind* the socket to a CAN interface and specify which CAN identifiers we are interested in for reading. Binding to an interface is done with a call to the bind() function:

```
    int bind(int sockfd,
             struct sockaddr *my_addr,
             socklen_t addrlen);
```

`sockfd` is the filedescriptor of the socket we have created in the last step; `my_addr` is a pointer to a datastructure describing the interface and the CAN indentifier. The datastructure looks like this as is declared in `can.h`:

```
struct sockaddr_can {
   sa_family_t  can_family;
   int          can_ifindex;
   int          can_id;
};

struct sockaddr_can adr;
```

`can_family` is in our case again `PF_CAN`:

```
addr.can_family = PF_CAN;
```

`can_ifindex` is the index number of the interface we want to send or listen to. The decoding from the symbolic string ″can0″ to the interface index is done with an ioctl:

```
struct ifreq ifr;

ifr.ifr_name = "can0";
ioctl(sockfd, SIOCGIFINDEX, &ifr);
addr.can_ifindex = ifr.ifr_ifindex;
```

It's up to the user to specify which CAN identifier he is interested in; for unfiltered traffic, the CAN_FLAG_ALL macro can be used:

```
addr.can_id = CAN_FLAG_ALL;
```

Now that the sockaddr and addr structs are defined, we can actually call bind():

```
bind(sockfd, (struct sockaddr *)&addr, sizeof(addr));
```

When bind returns successfully with 0, the socket is open and ready to receive or send data.

5.5.5 Sending CAN Messages

To send a CAN frame, we put the `can_id`, `can_dlc` and `payload` files into a `can_frame` and call the `write` function, looking like

```
    ssize_t write(int fd, const void *buf, size_t count);
```

In our example, the call is

```
    write(sockfd, &frame, sizeof(frame));
```

5.5.6 Receiving CAN Messages

To receive messages, we call the read function:

```
    ssize_t read(int fd, void *buf, size_t count);
```

`fd` is the filedescriptor we want to read from, `buf` is a pointer to the memory block to write to and `count` is the length of this block. With one call to the read function we read one frame from the socket.

```
    struct can_frame {
        int     can_id;
        int     can_dlc;
        union {
            int64_t         data_64;
            int32_t         data_32[2];
            int16_t         data_16[4];
            int8_t          data_8[8];
            uint64_t        data_u64;
            uint32_t        data_u32[2];
            uint16_t        data_u16[4];
            uint8_t         data_u8[8];
            int8_t          data[8];     /* shortcut */
        } payload;
    };
    struct can_frame frame;

    read(sockfd, &frame, sizeof(struct can_frame));
```

The struct `can_frame` contains the CAN identifier (`can_id`) and the length of the CAN message (`can_dlc`) as well as the payload.

In case the received CAN frame had the RTR (Remote Transmission Request) bit or the Extended bit set, the corresponding flags can be read from the struct can_frame. The flags are defined like this:

```
#define CAN_FLAG_RTR      0x40000000 /* remote transmission flag*/
#define CAN_FLAG_EXTENDED 0x80000000 /* extended frame */
```

To filter out the flags, corresponding masks are defined in can.h:

```
#define CAN_ID_EXT_MASK   0x1FFFFFFF /* extended CAN id mask */
#define CAN_ID_STD_MASK   0x000007FF /* standard CAN id mask */
```

5.5.7 Closing Interfaces & Further Reading

If the userspace application is finished, the socket filedescriptors have to be closed:

```
    int close(int fd);
```

So in our example we'll have to close the socket with:

```
    close(sockfd);
```

More details about the mentioned functions can be taken from the Linux manual pages:

- man 2 socket
- man 2 bind
- man 2 write
- man 2 read
- man 2 close

5.5.8 Getting help

Community supports three CAN specific mailings lists, hosted at berlios. You can subscribe to this lists for further discussion and help.

http://lists.berlios.de/mailman/listinfo/

Search for the lists:

- Socketcan-core
  Discussion about the socket-CAN core system
- Socketcan-users
  Help and discussion about using socket-CAN
- Socketcan-commit

5.6 Network

The phyCORE-PXA270 module has an SMSC 91C111 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.

5.7 LCD Graphics

phyCORE-PXA270's LCD support uses the standard PXA2XX's framebuffer support and can be used as a regular console when also an USB keyboard is attached to the system. fb-tools can be used to manipulate the frame buffer (colour depth).

For display definitions (resolution and frequency) see source file
`arch/arm/mach-pxa/pcm027-baseboard.c`
in the kernel tree.

5.8 LCD Backlight

The LCD backlight can be controlled by using the backlight class driver. This driver offers a sysfs entry to control the brightness and a connection to the frame buffer console and to the X-server for power management.

You can find the sysfs entries in `/sys/class/backlight/pcm027-bl` and control them with plain numbers.

- `max_brightness`
  To read back the maximum value (hardware dependend). This value feeded into the brightness entry gives the maximum backlight brightness.
- `brightness`
  Set the current brightness value (0 . . . max_brightness).
- `power`
  Set or read back backlight power. 0 means backlight is off, 1 means on.
- `actual_brightness`
  To read back the current brightness setting. Its the same as brightness.

More information about the backlight driver can be found in the following files in the Linux kernel:

- `video/backlight/backlight.c`
- `video/backlight/pcm027_bl.c`

Note: On the development board, J23 must be in position **1-2** to make the PWM#0 control the inverter. See chapter "LCD interface" in the phyCORE-PXA270 manual for further details.

5.9 SPI

The phyCORE-PXA270 board supports an SPI bus, based on the PXA270'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 phyCORE-PXA270.

On the phyCORE-PXA270, channel 1 of the SPI controller is connected to the MAX7301 GPIO expander chip. The BSP currently uses the "Chip Select" alternate function of GPIO 24 to select the MAX7301; This mean the controller handles chip selection by its own in hardware. This SPI controller mode works fine if only one SPI slave device is connected (in the case of phyCORE-PXA270 it is the MAX7301, see below).
If its planned in a custome design to add more devices to this SPI channel 1 (to let it act like a bus) any chip selection has to be done in software. In this case also for the MAX7301, so GPIO 24 must be a regular GPIO without any alternate function enabled.

For a description of the SPI framework see
`Documentation/spi/spi-summary`
and for PXA2xx's SPI driver see
`Documentation/spi/pxa2xx`

5.10 GPIO Expander

This is a GPIO expander that supports 28 additional EGPIOs.
To control the direction and level of each EGPIO echo plain numbers into special sysfs entries:
`/sys/class/egpio/egpio??/level`
Replace `??` with numbers from 4 to 31 for EGPIO4 to EGPIO31.
To control each EGPIO echo one of the following numbers into its `level` entry:

- `-2` to set the corresponding EGPIO as input only
- `-1` to set the corresponding EGPIO is input with internal pullup enabled

- `0` to set the corresponding EGPIO as output and its level to low
- `1` to set the corresponding EGPIO as output and its level to high

If the EGPIO is configured as input, `cat level` will show its current level. If it is configured as output this command will read back the current output level setting. Note: The latter case uses a cached value, so no SPI transmissions will occure.
At power up all EGPIO are defined as input without internal pullup.

Note: There is an offset between the EGPIO number of the MAX7301 and the card connector's EGPIO numbers. The MAX7301 only supports EGPIO4 to EGPIO31. The EGPIO4 of the MAX7301 is at the card connector EGPIO0, the EGPIO5 of the MAX7301 is at the card connector EGPIO1 and so on. The entries in `/sys/class/egpio` correspond to the MAX7301 numbering scheme.

5.11 AC97 Based Audio

The sound features can be used through standard PXA2xx AC97 ALSA support for the onboard Wolfson WM9712 device. See sources in `sound/arm/pxa2xx.c` in the kernel source tree for further information.

5.11.1 Sound Output

To play a sound, copy your favorite mp3 file to the phyCORE-PXA270, pop up the volume and play your mp3 file.

```
~# amixer sset PCM,0 20,20 unmute
~# amixer sset Headphone,0 20,20 unmute
~# amixer sset 'Master Left Inv',0 on
~# madplay <mp3file_name>
```

If external loudspeakers are connected it is possible to mute the built in speaker with `amixer sset 'Master Left Inv',0 off`.

5.11.2 Sound Record

Note: When the Wolfson WM9712 chip comes up after power on, every sound source is muted as default. To record any sound the desired audio source must be unmuted first.

To activate sound capturing the internal ADCs have to be powered up and unmuted first:

```
˜# amixer sset ADC,0 on
˜# amixer sset Capture 15,15 unmute
```

Now its time to select the desired audio source for capturing. The following commands select the stereo line in as the source:

```
˜# amixer sset Line 30,30 unmute
˜# amixer sset 'Capture Select',0 Line
```

To select the microphone instead of the stereo line in, these commands are required:

```
˜# amixer sset 'Mic 1',0 30
˜# amixer sset Capture Select,0 'Mic 1'
```

To record any sound the command `arecord` is the recommended way to do it. This example records about 20 seconds from the desired source:

```
˜# arecord -f dat -d 20 -D hw:0,0 test.wav
```

See `arecord`'s manual for further meaning of the command line parameters.

5.11.3 Advanced Sound Handling

Note: The Wolfson WM9712 is a complex beast with many features. Sometimes its hard to understand why it works or why it fails. Armed with it's datasheet, the AC'97 specification and kernel's powerful AC97 debug feature it is much easier to use WM9712's features in the manner you like or the way the chip supports it. Not all WM9712's features are supported by the ALSA utils out of the box. Some of this features needs kernel driver patches to let the ALSA utils aware of it.

To see the current WM9712's register settings simply enter:

```
˜# cat /proc/asound/card0/codec97#0/ac97#0-0+regs
```

This is an easy way to check the results of the `amixer` command and if it supports this feature out of the box.

To change any register's value manually (without `amixer` command for test purposes only) simply enter:

```
˜# echo "1a 0404" > /proc/asound/card0/codec97#0/ac97#0-0+regs
```

This example updates WM9712's register 0x1A to the new value 0x0404. You will also need the datasheet here to know the registers, their offset and meaning.
Note: Give all values in hex but without leading `0x`.

5.12 AC97 Based Touchscreen

This device is supported through PXA2xx's standard AC97 support for the onboard Wolfsson WM9712 device driver for touchscreen. In userspace this device is supported through the tslib, so it can be used by an X server as a pointing device. See sources in
`driver/input/touchscreen/wm97xx.c`
in the kernel source tree for further information.

Modul parameters to control the driver:

- **cont_rate** Sample rate in continuous mode (Hz).
  Default is 200 samples per second.
- **pen_int** Pen down detection (1 = interrupt, 0 = polling).
  This driver can either poll or use an interrupt to indicate a pen down event. If the IRQ request fails then it will fall back to polling mode. Default is interrupt.
- **pressure** Pressure readback (1 = pressure, 0 = no pressure).
- **ac97_touch_slot** Touch screen data slot AC97 number.
  enable/disable AUX ADC sysfs, default is enabled
- **aux_sys** disable AUX ADC sysfs entries.
- **status_sys** disable codec status sysfs entries.
  enable/disable codec status sysfs, default is enabled
- These parameters are used to help the input layer discard out of range readings and reduce jitter etc.
  - min, max: indicate the min and max values our touch screen returns
  - fuzz: use a higher number to reduce jitter

  The default values correspond to Mainstone II in QVGA mode Please read Documentation/input/input-programming.txt for more details.

  - **abs_x** Touchscreen absolute X min, max, fuzz.
  - **abs_y** Touchscreen absolute Y min, max, fuzz.
  - **abs_p** Touchscreen absolute Pressure min, max, fuzz.
- **rpu** Set internal pull up resistor for pen detect.
  Pull up is in the range 1.02k (least sensitive) to 64k (most sensitive) i.e. pull up resistance = 64k Ohms / rpu.
  We adjust this value if we are having problems with pen detect not detecting any down event.

- **pil** Set current used for pressure measurement.
  Set
  - pil = 2 to use 400µA
  - pil = 1 to use 200µA and
  - pil = 0 to disable pressure measurement.

  This is used to increase the range of values returned by the ADC when measuring touchpanel pressure.
- **pressure** Set threshold for pressure measurement.
  Pen down pressure below threshold is ignored.
- **delay** Set ADC sample delay.
  For accurate touchpanel measurements, some settling time may be required between the switch matrix applying a voltage across the touchpanel plate and the ADC sampling the signal.
  This delay can be set by setting delay = n. Valid values of n can be looked up in the 'delay_table' in the driver source. Long delays >1ms are supported for completeness, but are not recommended.
- **five_wire** Set to '1' to use 5-wire touchscreen.
  NOTE: Five wire mode does not allow for readback of pressure.
- **mask** Set ADC mask function.
  Sources of glitch noise, such as signals driving an LCD display, may feed through to the touch screen plates and affect measurement accuracy. In order to minimise this, a signal may be applied to the MASK pin to delay or synchronise the sampling.
  - 0 = No delay or sync
  - 1 = High on pin stops conversions
  - 2 = Edge triggered, edge on pin delays conversion by delay param (above)
  - 3 = Edge triggered, edge on pin starts conversion after delay param

Using the touchscreen requires a calibration. This has to be done the first time a newly built OSELAS.BSP-phyCORE-PXA270-3 runs on the target to create the calibration information.

To do so run the command:

```
~# ts_calibrate
```

The command uses the environt variable /texttttTSLIB_TSDEVICE (defined in /etc/profiles) and the so called ts-lib, configured in /etc/ts.conf.

Note: When you intend to calibrate the touch stop an already running X server prior starting `ts_calibrate`. They can't share the framebuffer, so the X server gets killed and the `ts_calibrate` command hangs forever.

5.13 AC97 Based Analogue Converter

The Wolfsson WM9712L AC97 device supports up to four analogue inputs (AUX1 … AUX4) with 12 bit resolution for general purpose (in addition to the audio inputs). See WM9712L's data sheet for proper use.

1. AUX1
   Can be used for battery monitoring (analogue up to 3.3V) or dead battery detection (comparator). Read back via
   `/sys/devices/platform/pxa2xx-ac97/0-0:WM9711,WM9712/aux1`
2. AUX2
   Can be used for battery monitoring (analogue up to 3.3V) or low battery detection (comparator). Read back via
   `/sys/devices/platform/pxa2xx-ac97/0-0:WM9711,WM9712/aux2`
3. AUX3
   Can be used for battery monitoring (analogue up to 5.0V). Read back via
   `/sys/devices/platform/pxa2xx-ac97/0-0:WM9711,WM9712/aux3`
4. AUX4
   General purpose analogue input (up to 3.3V) or reference voltage for dead/low battery detection comparator. Read back via
   `/sys/devices/platform/pxa2xx-ac97/0-0:WM9711,WM9712/aux4`

5.14 AC97 Based GPIO

The Wolfsson WM9712L AC97 device supports up to five digital inputs/outputs (GPIO1 … GPIO5). While these pins shares other functions see WM9712L's data sheet for proper use. Beside the real GPIOs the WM9712L AC97 device has also five virtual GPIO. They are only internal and report status information only. All 10 bits of information can be read back from
`/sys/devices/platform/pxa2xx-ac97/0-0:WM9711,WM9712/gpio`.
See datasheet in chapter "GPIO and Interrupt Control" for bit assignments.

5.15 X11 Graphics

In OSELAS.BSP-phyCORE-PXA270-3's **full** configuration an Xorg server is supported through PXA270's framebuffer device. It supports the attached 640 x 480 TFT display with 16 bit colour depth and runs a windwo manager on top of it. To control it a USB mouse or the touchscreen can be used.

5.16 USB Host Controller

Standard OHCI Rev. 1.0a implementation.
Only channel 1 is supported, channel 2 and 3 are not available.

Make sure the required USB device module for the device to be attached is already loaded. The OSELAS.BSP-phyCORE-PXA270-3 supports USB mice and USB Mass Storage devices (MemorySticks aso.) as default.

5.17 I²C Master

The PXA270 processor based phyCORE-PXA270 supports a dedicated I²C controller on chip. 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.

5.17.1 I²C Realtime Clock (RTC8564)

Due to the kernel's Real Time Clock framework the RTC8564 clock chip can be accesses using the same tools as any other clocks.

Date and time can be manipulated with the `hwclock` tool, using the –systohc and –hctosys options. For more information about this tool refer to hwclock's manpages.

5.18 Status LEDs

These LEDs are supported to display CPU activity and heart beat. They occupy the two processor's GPIOs 90 and 91 for this purpose.

Note: These GPIOs are also used with the breakout board. So activity and heart beat function are disabled as default.

6 Some hints on using phyCORE-PXA270

6.1 Manually Setting up Flash Based Root Filesystem

6.1.1 Partitioning the Local Flash Device

phyCORE-PXA270's onboard flash is 32 MiB in size divided into 128 individual blocks each of 256kiB in size.

The first block is occupied by the U-Boot boot loader itself. From the CPU's view the base address of the whole flash is 0x00000000.

! Don't write anything to the first 256kiB block in flash (from address 0x00000000 up to 0x0003FFFF). This destroys the U-Boot binary and the board won't start anymore. In this case you need a JTAG based hardware debugger to restore the U-Boot.

We divide the flash into two independent partitions (see figure 6.1) by setting the `mtdparts`, `partition` and `mtdids` environment variable:

```
uboot> setenv mtdids nor0=phys_mapped_flash
uboot> setenv mtdparts mtdparts=phys_mapped_flash:256k(u-boot)ro,-(root)
uboot> setenv partition nor0,1
```

The first partition is 256kiB read only and protects the U-Boot image. The second partition starts at offset 0x00040000 and occupies the remaining space on this flash. The offset is important, we will need it in various later commands.

To write anything into the second partition we must erase the entire partition first.

```
uboot> erase 0x00040000 0x01FFFFFF
```

The partition is now ready to be written to.

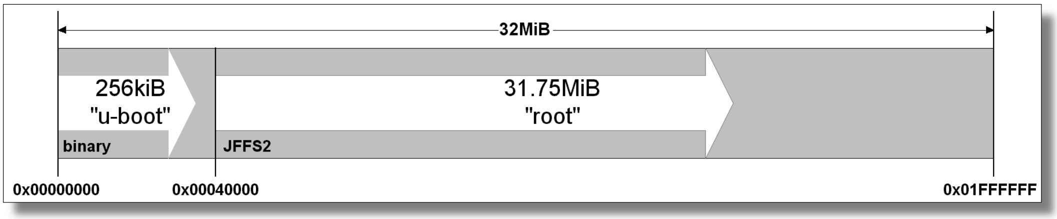

Figure 6.1: *Partitioning the flash device.*

6.1.2 Writing a Root Filesystem into Flash

PTXdist generates a root filesystem image in JFFS2 format. Here is the way to instantly use it as the root filesystem. It also embeds the kernel image.

Note: This description only replaces the root filesystem part. It does not change anything in U-Boot's part of the flash.

To be able to download, we first have to copy the `root.jffs2` file from our active project (see directory `images`) into the TFTP server's directory on our host.

```
~/work/myproject > cp images/root.jffs2 /tftpboot
```

From the target's point of view the `tftp` command downloads this image file and a `cp` command finally copies it into the flash.

Downloading the image file into phyCORE-PXA270's RAM:

```
uboot> tftp 0xA3000000 root.jffs2
```

Double check both parameters given to the `tftp` command! If a wrong address (or **no** address!) is given `tftp` destroys the U-Boot!
Avoid using the `tftp` command without any argument!

The next step is to copy the root image from RAM into flash memory:

```
uboot> cp.b 0xA3000000 0x40000 $(filesize)
```

This copying process may take a while. After it's done, this partition is ready to be used as a root filesystem.

6.1.3 Booting from Local Flash

U-Boot can handle JFFS2 based flash partitions. To load a kernel image from inside this filesystem, U-Boot supports the `fsload` command.

PTXdist generates a JFFS2 root filesystem image that embeds a kernel image in `/boot/uImage` as default. To load this kernel image we can run:

```
uboot> fsload /boot/uImage
```

U-Boot scans the JFFS2 filesystem and copies this kernel image into system RAM. After this is done we can start this kernel image with the command

```
uboot> bootm
```

To automatise this, we replace the current U-Boot `bootcmd` environment variable contents:

```
uboot> setenv bootcmd 'fsload /boot/uImage; run setup_bootargs;
     bootm'
```

! Note the two ' (single quotes)! Without them, U-Boot tries to run the commands immediately!

See next section about the contents of the `setup_bootargs` environment variable to get a correct kernel setup.

6.1.4 Using Local Flash at Runtime

When Linux kernel initialisation is finished, it automatically mounts the root filesystem.

To use the correct root filesystem we should setup a kernel parameter to define the root filesystem to use.

The Linux kernel expects the parameter `root` to define the device that contains the root filesystem, and the parameter `rootfstype` to define its filesystem type. On many systems we can omit the `rootfstype` kernel parameter, the kernel can autodetect the filesystem. But not with JFFS2!

The flash access driver comes with the MTD subsystem, so the device nodes are called `mtdblockX` with `X` as the partition number. We have more than one partition in our flash device so we have to define the right one as the root filesystem.
In this case the correct kernel parameter is:

```
root=/dev/mtdblock2 rootfstype=jffs2
```

To complete the U-Boot environment (with regard to the section before) we should also set the `setup_bootargs` variable:

```
uboot> setenv setup_bootargs 'setenv bootargs root=/dev/mtdblock2
     rootfstype=jffs2 $(mtdparts)'
```

! Note the two ' (single quotes)! Without them, U-Boot tries to run the commands immediately!

With this U-Boot environment setup we can simply enter

```
uboot> boot
```

and U-Boot automatically searches for the kernel in the flash, loads and starts it and the Linux kernel itself automatically mounts the root filesystem from the flash partition.

6.2 Decreasing Boot Time

6.2.1 Disabling Console Output During Kernel Startup

Console output during kernel startup consumes a lot of time. Most of the information we will see at this point of time are more useful when we are developing our system. If our system runs in production it's more useful to save time when booting. If we add the kernel parameter `quiet` this will suppress printk messages. Note that printk messages are still buffered in the kernel and can be retrieved after booting using the `dmesg` command.

When the `init` process starts the console is activated again.

7 Getting help

Here are a list of locations where you can get help in case of trouble or questions how to do something special within PTXdist or general questions about Linux in the embedded world.

7.1 Mailing lists

About PTXdist in special

This is an english language public mailing list for questions about PTXdist. See web site

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

how to subscribe to this list.

About embedded Linux in general

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

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

how to subscribe to this list. Note: You also can send english language mails.

7.2 News groups

About Linux in embedded environments

This is an english language news group for general questions about Linux in embedded environments.

comp.os.linux.embedded

About general Unix/Linux questions

This is a german language news group for general questions about Unix/Linux programming.

de.comp.os.unix.programming

7.3 Chat/IRC

About PTXdist in special

irc.freenode.net:6667

Create a connection to the **irc.freenode.net:6667** server and enter the chat group **#ptxdist**. This is an english language group to answer questions about PTXdist. Best time to meet somebody in there is at europeen daytime.

7.4 Miscellaneous

Online Linux Kernel Cross Reference

A powerful cross reference to be used online.

http://www.rts.uni-hannover.de/linux/lxr/source

U-Boot manual (partially)

Manual how to survive in an embedded environment and how to use the U-Boot on target's side

http://www.denx.de/wiki/DULG

8 Customer Support

8.1 Free of charge support

Everytime you can get free of charge support through mailing lists and IRC discussion groups. Up to date information about presently offered public mailing lists and IRC channels you can find at our web site:

http://www.pengutronix.de

For additional information see also chapter 7 at page 44.

8.2 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
Hannoversche Strasse 2
D-31134 Hildesheim
Germany
Phone: +49 - 51 21 / 20 69 17 - 0
Fax: +49 - 51 21 / 20 69 17 - 9

or by electronic mail:

sales@pengutronix.de