# OSELAS.BSP( )
# Phytec phyCORE-PXA270


## Quick Start Manual

http://www.oselas.com



$Rev : 824\ Date : 2007-12-1314:57:41+0100(Thu, 13Dec2007)$

# Contents

# Part I

# OSELAS Quickstart for Phytec phyCORE-PXA270

# 1 Getting a working Environment

## 1.1 Download Parts

In order to follow this manual, some software archives are needed. There are several possibilities how to get these: either as part of an evaulation board package, or by download from a world wide 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-phyCORE-PXA270-4, the following archives should be available on the development host:

- ptxdist-1.0.1.tgz
- ptxdist-1.0.1-patches.tgz
- OSELAS.BSP-phyCORE-PXA270-4.tar.gz
- OSELAS.Toolchain-1.1.1.tar.bz2

## 1.2 PTXdist Installation

PTXdist is shipped divided into several archives. This chapter provides information about how to install and configure PTXdist on the development host to get a working environment to build root filesystems for target systems.

### 1.2.1 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:** `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_de.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.

### 1.2.2 Extracting the Sources

To install PTXdist, at least two archives have to be extracted:

**ptxdist-1.0.1.tgz** The PTXdist software itself.

**ptxdist-1.0.1-patches.tgz** All patches against upstream software packets (known as the 'patch repository').

**ptxdist-1.0.1-projects.tgz** Generic projects (optional), can be used as a starting point for self-built projects.

The PTXdist and patches packets have 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 steps are to extract the archives:

```
~/local$ tar -zxf ptxdist-1.0.1.tgz
~/local$ tar -zxf ptxdist-1.0.1-patches.tgz
```

and if required the generic projects:

```
~/local$ tar -zxf ptxdist-1.0.1-projects.tgz
```

If everything goes well, we now have a PTXdist-1.0.1 directory, so we can change into it:

```
~/local$ cd ptxdist-1.0.1
~/local/ptxdist-1.0.1$ ls -l

total 429
drwxr-xr-x   2 jbe users   1024 2007-10-19 23:24 autoconf/
-rwxr-xr-x   1 jbe users     28 2007-10-19 23:20 autogen.sh
drwxr-xr-x   2 jbe users   1024 2007-10-19 23:24 bin/
-rw-r--r--   1 jbe users 121724 2007-10-19 23:20 ChangeLog
drwxr-xr-x   8 jbe users   1024 2007-10-19 23:24 config/
-rwxr-xr-x   1 jbe users 204310 2007-10-19 23:24 configure
-rw-r--r--   1 jbe users   9755 2007-10-19 23:20 configure.ac
-rw-r--r--   1 jbe users  18361 2007-10-19 23:20 COPYING
-rw-r--r--   1 jbe users   2792 2007-10-19 23:20 CREDITS
drwxr-xr-x   2 jbe users   1024 2007-10-19 23:24 debian/
drwxr-xr-x   2 jbe users   1024 2007-10-19 23:24 Documentation/
drwxr-xr-x   7 jbe users   1024 2007-10-19 23:24 generic/
-rw-r--r--   1 jbe users     58 2007-10-19 23:20 INSTALL
-rw-r--r--   1 jbe users   2686 2007-10-19 23:20 Makefile.in
drwxr-xr-x 132 jbe users   4096 2007-10-19 23:24 patches/
drwxr-xr-x   5 jbe users   1024 2007-10-19 23:22 projects/
-rw-r--r--   1 jbe users   3866 2007-10-19 23:20 README
-rw-r--r--   1 jbe users    691 2007-10-19 23:20 REVISION_POLICY
drwxr-xr-x   6 jbe users  24576 2007-10-19 23:24 rules/
drwxr-xr-x   6 jbe users   1024 2007-10-19 23:24 scripts/
drwxr-xr-x   2 jbe users   1024 2007-10-19 23:24 tests/
-rw-r--r--   1 jbe users  28468 2007-10-19 23:20 TODO
```

### 1.2.3 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:

```
~/local/ptxdist-1.0.1$ ./configure
```

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

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

ptxdist version 1.0.1 configured.
Using '/usr/local' for installation prefix.

Report bugs to ptxdist@pengutronix.de

~/local/ptxdist-1.0.1$
```

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-1.0.1$ 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-1.0.1$ su
[enter root password]
/home/username/local/ptxdist-1.0.1$ 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-1.0.1$ cd
~$ rm -fr local/ptxdist-1.0.1
```

### 1.2.4 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 packages 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.

#### 1.2.4.1 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>
```

#### 1.2.4.2 Source Archive Location

Whenever PTXdist downloads source archives it stores it project locally. If we are working with more than one project, every project would download its own required archives. To share all source archives between all projects PTXdist can be configured to use 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 projects.

#### 1.2.4.3 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-1.0.1/projects:/office/my_projects/ptxdist
```

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

## 1.3 Toolchains

### 1.3.1 Abstract

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.

### 1.3.2 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 menuconfig
```

and navigate to `PTXdist Config, Architecture` and `Check for specific toolchain vendor`. Clear this entry to disable the toolchain vendor check.

### 1.3.3 Building a Toolchain

PTXdist-1.0.1 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.

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-1.1.1/`.

Usually the `/opt` directory is not world writable. So in order to build our OSELAS.Toolchain into that directory we need to use a root account to change the permissions so that the user can write (`mkdir /opt/OSELAS.Toolchain-1.1.1 ; chown <username> /opt/OSELAS.Toolchain-1.1.1; chmod a+rwx /opt/OSELAS.Toolchain-1.1.1`).

#### 1.3.3.1 Building the OSELAS.Toolchain for OSELAS.BSP-phyCORE-PXA270-4

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-phyCORE-PXA270-4 board support package is

`arm-iwmmx-linux-gnueabi_gcc-4.1.2_glibc-2.5_linux-2.6.18`.

So the steps to build this toolchain are:

```
~$ tar xf OSELAS.Toolchain-1.1.1.tar.bz2
~$ cd OSELAS.Toolchain-1.1.1
~/OSELAS.Toolchain-1.1.1$ ptxdist select
  ptxconfigs/arm-iwmmx-linux-gnueabi_gcc-4.1.2_glibc-2.5_linux-2.6.18.ptxconfig
~/OSELAS.Toolchain-1.1.1$ 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
- 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

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.

### 1.3.4 Freezing the Toolchain

As we build and install this toolchain with regular user rights we should modify the permissions as a last step to avoid any later manipulation. To do so we could set all toolchain files to read only or changing recursivley the owner of the whole installation to user root.

This is an important step for reliability. Do not omit it!

#### 1.3.4.1 Building additional Toolchains

The OSELAS.Toolchain-1.1.1 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 projekt, removing the current `ptxconfig` link and creating a new one.

```
~/OSELAS.Toolchain-1.1.1$ ptxdist clean
~/OSELAS.Toolchain-1.1.1$ rm ptxconfig
~/OSELAS.Toolchain-1.1.1$ ptxdist select
       ptxconfigs/any_another_toolchain_def.ptxconfig
~/OSELAS.Toolchain-1.1.1$ ptxdist go
```

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

`/opt/OSELAS.Toolchain-1.1.1/architecture_part`.

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

`/opt/OSELAS.Toolchain-1.1.1/architecture_part/version_part`.

# 2 Building the ”light” Image for phyCORE-PXA270

## 2.1 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-PXA270-4 for the phyCORE-PXA270:

```
~$ tar -zxf OSELAS.BSP-phyCORE-PXA270-4.tar.gz
~$ cd OSELAS.BSP-phyCORE-PXA270-4
```

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-PXA270-4 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-PXA270-4$ 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-PXA270-4$ ptxdist toolchain
     /opt/OSELAS.Toolchain-1.1.1/gcc-4.1.2-glibc-2.5-kernel-2.6.18/
     arm-iwmmx-linux-gnueabi/bin
```

## 2.2 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-PXA270-4$ 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].

## 2.3 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

[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.

```
~/OSELAS.BSP-phyCORE-PXA270-4$ 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

# 3 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.

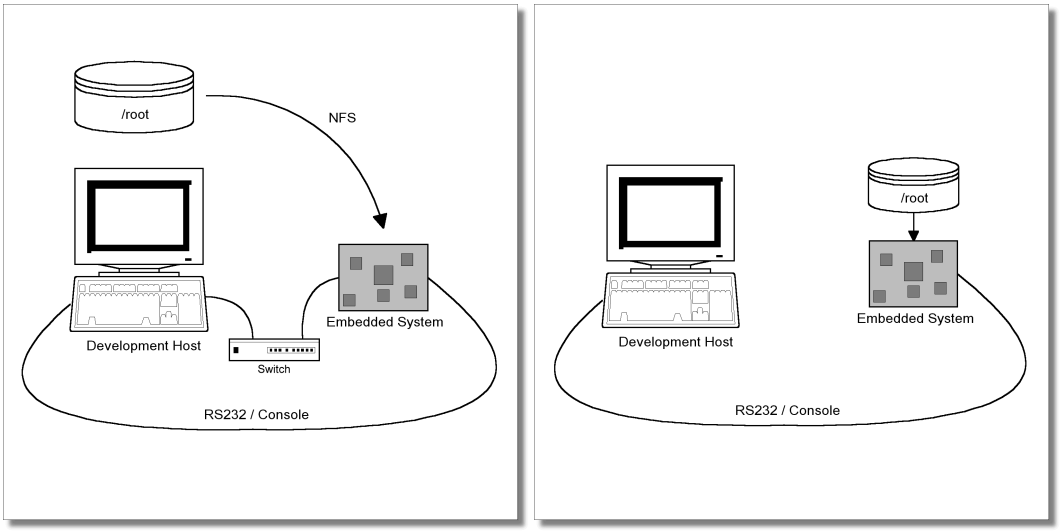


***Figure 3.1:*** *Booting the root filesystem, built with PTXdist, from the host via network and from flash.*

Figure 3.1 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 set up our target with features supported by PTXdist to simplify this challange.

## 3.1 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-PXA270-4 comes with a predefined environment setup to easily bring up the phyCORE-PXA270.

## 3.2 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` |
| mtdids | nor0=physmap-flash.0 | Where to load kernel images in the case flash is used as source (standalone mode) |
| mtdparts | mtdparts=physmap-flash.0:256k(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 for the kernel when booting remotely |
| jffs2 | root-pcm027.jffs2 | This file will be used when updating the rootfs in flash (see variable `prg_jffs2`). |
| uboot | u-boot-pcm027.bin | This file will be used when updating the U-Boot in flash (see variable `update`) |
| bargs_base | setenv bootargs console=ttyS0,115200 ip=$(ipaddr):$(serverip):$(gateway):$(netmask) $(hostname):eth0:off | This is one part of the kernel parameters used by all configurations. It defines the serial console to be used and its settings. |
| bargs_mtd | setenv bootargs $(bootargs) $(mtdparts) | This adds the flash partitioning information to the kernel parameters. |
| bargs_flash | setenv bootargs $(bootargs) root=/dev/mtdblock2 rootfstype=jffs2 | Defines the flash specific bootargs to kernel for standalone mode |
| bargs_nfs | setenv bootargs $(bootargs) root=/dev/nfs nfsroot=$(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` |
| bcmd_flash | run bargs_base bargs_mtd bargs_flash; bootm 0x40000 | This defines a command that combines the kernel parameters and boots a kernel from flash |
| bcmd_net | run bargs_base bargs_mtd bargs_nfs; tftpboot 0xa0200000 $(uimage); | This defines a command that combines the kernel parameters and boots a kernel from network |
| bootcmd | run bcmd_flash | Defines the command that should run after powerup |
| 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 |
| update | tftpboot 0xa0200000 $(uboot); erase nor0,0; cp.b 0xa0200000 0x0000 $(filesize) | This command is for convenience only and replaces the U-Boot in flash |

Note: All identifiers in U-Boot are variables. But some contain other commands and variable references. To run their contents, we can simply 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 successful:

```
uboot:  set default 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.

## 3.3 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.

### 3.3.1 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.

### 3.3.2 Preparations on the Embedded Board

We already provided the phyCORE-PXA270 with the default environment at page 13. So there is no additional preparation required here.

### 3.3.3 Booting the Embedded Board

The default environment coming with the OSELAS.BSP-phyCORE-PXA270-4 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.

## 3.4 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.

### 3.4.1 Development Host Preparations

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

If not, then a TFTP server has to be installed and configured on the development host. The exact method of doing 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.

### 3.4.2 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-PXA270-4 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 after this step.

This command will automatically write a root filesystem to the correct flash partition on the phyCORE-PXA270. It only works if we previously have set up the environment variables successfully (described at page 13).
The command should output a line like this when it was successful:

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

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

### 3.4.3 Booting the Embedded Board

To check that 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 work 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-PXA270-4 has a predefined script for booting stand-alone. To use it, we can simply enter

```
uboot> run bcmd_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 bcmd_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.

# 4 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 (PCM-990).
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 CPU 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.

## 4.1 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-PXA270-4 is being used.

From userspace the flash partitions can be accessed as

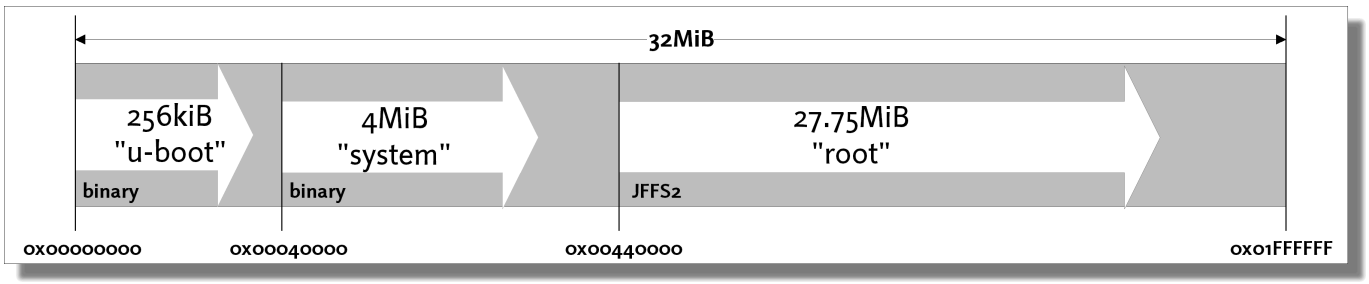


***Figure 4.1:*** *Current flash partitions*

- /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.

## 4.2 Kernel Modules

The OSELAS.BSP-phyCORE-PXA270-4 BSP uses a modularised kernel to support most of the hardware. This is the list of modules loaded at system startup as a default:

```
~# lsmod
Module                  Size  Used by
pcm027can               3232  0
sja1000dev             14152  1 pcm027can
candev                  4576  1 sja1000dev
can_raw                 7072  0
can                    24772  1 can_raw
st24cxx                 7996  0
mmc_block               9988  0
pxamci                  6624  0
mmc_core               25828  2 mmc_block,pxamci
rtc_pcf8563             6252  0
rtc_core               14896  1 rtc_pcf8563
max7301                 5536  0
pxa2xx_spi             14336  0
pxa27x_gpioevent        4576  0
pxa27x_pwm              6272  0
vfat                   10528  0
fat                    48660  1 vfat
sd_mod                 21280  0
usb_storage            36128  0
scsi_mod               89636  2 sd_mod,usb_storage
usbhid                 19396  0
hid                    33056  1 usbhid
mousedev               11076  0
ohci_hcd               18532  0
```

## 4.3 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 4.10)
- 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-PXA270-4):

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.

## 4.4 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).

Note: You cannot map a GPIO twice. This results in an error message.

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.

To unmap a mapped GPIO write the pinnumber only into the `unmap_gpio` sysfs entry.

```
~# echo 19 > /sys/class/gpio/unmap_gpio
```

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.

## 4.5 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.

## 4.6 Socket CAN

The phyCORE-PXA270 has one SJA1000 based CAN controller, which is supported by drivers using the CAN framework ”Socket-CAN”. Using this framework, CAN interfaces can be programmed with the BSD socket API.

## 4.7 About 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 (candev.ko)
- chip drivers (e. g. mscan, sja1000 etc.)

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

[1] ISO 11898/11519

### 4.7.1 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:

| 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 |

***Table 4.1:*** *CAN interface information*

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
```

If it seems the generic bit rate calculation fails to find a working nominal CAN bit time setting, we can provide our own manually calculated settings. To do so, the following values are required:

- **bitrate** nominal bit rate to be manually set [$s^{-1}$]
- **tq** CAN base time quanta [ns]
- **err** max. allowed bit rate deviation [ppm]

- **prop_seg** CAN bit timing: Prop_Seg [Tq]
- **phase_seg1** CAN bit timing: Phase_Seg1 [Tq]
- **phase_seg2** CAN bit timing: Phase_Seg2 [Tq]
- **sjw** CAN bit timing: Sync Jump Width [Tq]
- **sam** sampling count: 0 for one sample, 1 for three samples per bit

The following example shows how to set up a 125k bitrate with specific bit rate settings for `can0`:

```
~# canconfig can0 setentry bitrate 125000 tq 421 err 3000 prop_seg 7 \
             phase_seg1 8 phase_seg2 3 sjw 2 sam 0
```

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).

### 4.7.2 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.

## 4.8 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.

## 4.9 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.

## 4.10 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:

- `drivers/video/backlight/backlight.c`
- `drivers/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.

## 4.11 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`

## 4.12 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.

## 4.13 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.

### 4.13.1 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,0 20,20 unmute      # controls the built-in speaker
~# amixer sset 'Master Left Inv',0 on     # activates the built-in speaker by phase reversal
~# 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`.

Note: We also can use the command "alsamixer" to handle mixer's settings.

### 4.13.2 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'
```

Maybe the recorded sound level will be very low. To improve the volume we can enable a 20dB boost with the following command:

```
~# amixer sset 'Capture 20dB Boost',0 on
```

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.

### 4.13.3 Advanced Sound Handling

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

To see the current WM9712 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`.

## 4.14 AC97 Based Touchscreen

This device is supported through PXA2xx's standard AC97 support for the onboard Wolfson 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-PXA270-4 runs on the target to create the calibration information before you can use the X server.

To do so run the command:

```
~# ts_calibrate
```

The command uses the environment variable `TSLIB_TSDEVICE` (defined in /etc/profile) and the so called ts-lib, configured in /etc/ts.conf.

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

## 4.15 X11 Graphics

In OSELAS.BSP-phyCORE-PXA270-4'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.

## 4.16 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-PXA270-4 supports USB mice and USB Mass Storage devices (MemorySticks aso.) as default.

## 4.17 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.

### 4.17.1 I²C Realtime Clock (RTC8564)

Due to the kernel's Real Time Clock framework the RTC8564 clock chip can be accessed 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.

### 4.17.2 I²C device 24W32

This device is a 4kByte non-volatile memory. Only the upper 2kByte can be used for any purpose, due to the lower 2kBytes being used by the bootloader itself to store its environment.

This type of memory is accessible through the sysfs filesystem. To read the EEPROM contents simply `open()` the entry `/sys/bus/i2c/devices/0-0054/eeprom` and use `fseek()` and `read()` to get the values.

## 4.18 Status LEDs

These LEDs are supported to display CPU activity and heart beat. They occupy the two processor 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.

## 4.19 SD-Card and MMC Support

The phyCORE-PXA270 supports Secure Digital Cards and Multi Media Cards to be used as general purpose block-devices. They can be used in the same way as any other blockdevice or filesystem.

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

Whenever a card is plugged into the system, the udev mechanism creates device nodes in system's dev/ directory: A /dev/mmcblk0 to access the whole block device. Used for fdisk command for example. And one or more /dev/mmcblk0p? nodes, with ? starting from 1 up to the count of partitions on this card.

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.

## 4.20 Realtime Capabilities

This section should give a small summary of the realtime capabilites of the phyCORE-PXA270 without a running X system.

System without any other load:

```
root@phyCORE-PXA270:~ cyclictest -p 80 -n -l 50000 -t 5 -q
T: 0 (  475) P:80 I:   1000 C:  50000 Min:     19 Act:     29 Avg:     29 Max:    164
T: 1 (  476) P:79 I:   1500 C:  33345 Min:     20 Act:    112 Avg:     29 Max:    204
T: 2 (  477) P:78 I:   2000 C:  25009 Min:     22 Act:     68 Avg:     28 Max:    113
T: 3 (  478) P:77 I:   2500 C:  20007 Min:     21 Act:     55 Avg:     29 Max:     68
T: 4 (  479) P:76 I:   3000 C:  16672 Min:     21 Act:     24 Avg:     29 Max:    173
```

System with a running hackbench 1 as load:

```
root@phyCORE-PXA270:~ cyclictest -p 80 -n -l 50000 -t 5 -q
T: 0 ( 2318) P:80 I:   1000 C:  50000 Min:     28 Act:    163 Avg:    264 Max:    816
T: 1 ( 2319) P:79 I:   1500 C:  33341 Min:     35 Act:    293 Avg:    338 Max:    833
T: 2 ( 2320) P:78 I:   2000 C:  25006 Min:     39 Act:    307 Avg:    332 Max:    929
T: 3 ( 2321) P:77 I:   2500 C:  20005 Min:     39 Act:    398 Avg:    398 Max:    928
T: 4 ( 2322) P:76 I:   3000 C:  16670 Min:     70 Act:     70 Avg:    448 Max:    904
```

hackbench runs a series of parallel job that forces the CPU to flush its cache. That is why the maximum latency increases by factor 4...8 compared to the idle system.

# 5 Getting help

Below 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.

## 5.1 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/mailinglists/index_en.html

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*.

**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/mailinglists/index_de.html

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

## 5.2 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**

## 5.3 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.

## 5.4 Miscellaneous

**Online Linux Kernel Cross Reference**

A powerful cross reference to be used online.

http://lxr.linux.no/blurb.html

**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

## 5.5 phyCORE-PXA270 specific Maillist

**OSELAS.Phytec@pengutronix.de**

This is an english language public maillist for all BSP related questions specific to Phytec's hardware. See web site

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

how to subscribe to this list.

## 5.6 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**
**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**