OSELAS.BSP( )
Phytec phyCORE-i.MX31

Quickstart Manual

http://www.oselas.com

1217 2009-03-25 11:58:31 +0100 (Mi, 25 M 2009)

Contents

Part I

OSELAS Quickstart for Phytec phyCORE-i.MX31

1 Getting a working Environment

1.1 Download Software Components

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

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

To build OSELAS.BSP-Phytec-phyCORE-11, the following archives have to be available on the development host:

- ptxdist-1.99.12.tgz
- ptxdist-1.99.12-patches.tgz
- OSELAS.BSP-Phytec-phyCORE-11.tar.gz
- OSELAS.Toolchain-1.99.3.2.tar.bz2

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

1.2 PTXdist Installation

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

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

1.2.1 Main parts of PTXdist

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

PTXdist consists of the following parts:

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

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

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

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

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

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

1.2.2 Extracting the Sources

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

ptxdist-1.99.12.tgz The PTXdist software itself.

ptxdist-1.99.12-patches.tgz All patches against upstream software packets (known as the 'patch repository').

ptxdist-1.99.12-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-1.99.12.tgz
~/local# tar -zxf ptxdist-1.99.12-patches.tgz
```

and if required the generic projects:

```
~/local# tar -zxf ptxdist-1.99.12-projects.tgz
```

If everything goes well, we now have a PTXdist-1.99.12 directory, so we can change into it:

```
~/local# cd ptxdist-1.99.12
~/local/ptxdist-1.99.12# ls -l
total 487
drwxr-xr-x  13 jb users   1024 Mar 23 13:25 ./
drwxr-xr-x  22 jb users   3072 Mar 23 13:25 ../
-rw-r--r--   1 jb users    377 Feb 23 22:23 .gitignore
-rw-r--r--   1 jb users  18361 Apr 24  2003 COPYING
-rw-r--r--   1 jb users   3731 Mar 11 18:09 CREDITS
-rw-r--r--   1 jb users 115540 Mar  7 15:25 ChangeLog
-rw-r--r--   1 jb users     58 Apr 24  2003 INSTALL
-rw-r--r--   1 jb users   2246 Feb  9 14:29 Makefile.in
```

```
-rw-r--r--   1 jb users   4196 Jan 20 22:33 README
-rw-r--r--   1 jb users    691 Apr 26  2007 REVISION_POLICY
-rw-r--r--   1 jb users  54219 Mar 23 10:51 TODO
drwxr-xr-x   2 jb users   1024 Mar 23 11:27 autoconf/
-rwxr-xr-x   1 jb users     28 Jun 20  2006 autogen.sh*
drwxr-xr-x   2 jb users   1024 Mar 23 11:27 bin/
drwxr-xr-x   6 jb users   1024 Mar 23 11:27 config/
-rwxr-xr-x   1 jb users 226185 Mar 23 11:27 configure*
-rw-r--r--   1 jb users  12390 Mar 23 11:16 configure.ac
drwxr-xr-x   2 jb users   1024 Mar 23 11:27 debian/
drwxr-xr-x   8 jb users   1024 Mar 23 11:27 generic/
drwxr-xr-x 164 jb users   4096 Mar 23 11:27 patches/
drwxr-xr-x   2 jb users   1024 Mar 23 11:27 platforms/
drwxr-xr-x   4 jb users   1024 Mar 23 11:27 plugins/
drwxr-xr-x   6 jb users  30720 Mar 23 11:27 rules/
drwxr-xr-x   7 jb users   1024 Mar 23 11:27 scripts/
drwxr-xr-x   2 jb users   1024 Mar 23 11:27 tests/
```

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-1.99.12# ./configure
```

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

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

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

ptxdist version 1.99.12 configured.
Using '/usr/local' for installation prefix.

Report bugs to ptxdist@pengutronix.de
```

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

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

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

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

```
~/local/ptxdist-1.99.12# make
```

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

```
~/local/ptxdist-1.99.12# sudo make install
[enter root password]
[...]
```

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-1.99.12# cd
~# rm -fr local
```

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.

Proxy Setup

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

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

Source Archive Location

Whenever PTXdist downloads source archives it stores these archives in a project local manner. 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.

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-1.99.12/projects:/office/my_projects/ptxdist`

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

1.3 Toolchains

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

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

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

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

```
~# gcc test.c -o test
```

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

```
~# arm-softfloat-linux-gnu-gcc test.c -o test
```

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

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

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

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

1.3.1 Using Existing Toolchains

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

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

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

Open the OSELAS.BoardSupport() Package menu with:

```
~# ptxdist platformconfig
```

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

1.3.2 Building a Toolchain

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

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-1.99.3/`.

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

```
mkdir /opt/OSELAS.Toolchain-1.99.3
chown <username> /opt/OSELAS.Toolchain-1.99.3
chmod a+rwx /opt/OSELAS.Toolchain-1.99.3.
```

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

1.3.3 Building the OSELAS.Toolchain for OSELAS.BSP-Phytec-phyCORE-11

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-Phytec-phyCORE-11 board support package is

```
arm-1136jfs-linux-gnueabi_gcc-4.3.2_glibc-2.8_binutils-2.19_kernel-2.6.27-sanitized
```

So the steps to build this toolchain are:

```
~# tar xf OSELAS.Toolchain-1.99.3.2.tar.bz2
~# cd OSELAS.Toolchain-1.99.3.2
~/OSELAS.Toolchain-1.99.3.2# ptxdist select ptxconfigs/\ Enter
> arm-1136jfs-linux-gnueabi_gcc-4.3.2_glibc-2.8_binutils-2.19_kernel-2.6.27-sanitized.ptxconf
~/OSELAS.Toolchain-1.99.3.2# ptxdist go
```

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

Measured times on different machines:

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

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 permissions 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 change recursively the owner of the whole installation to user root.

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

Building additional Toolchains

The OSELAS.Toolchain-1.99.3.2 bundle comes with various predefined toolchains. Refer the `ptxconfigs/` folder for other definitions. To build additional toolchains we only have to clean our current toolchain project, removing the current `selected_ptxconfig` link and creating a new one.

```
~/OSELAS.Toolchain-1.99.3.2# ptxdist clean
~/OSELAS.Toolchain-1.99.3.2# rm selected_ptxconfig
~/OSELAS.Toolchain-1.99.3.2# ptxdist select \ Enter
> ptxconfigs/any_other_toolchain_def.ptxconfig
~/OSELAS.Toolchain-1.99.3.2# ptxdist go
```

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

/opt/OSELAS.Toolchain-1.99.3/architecture_part.

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

/opt/OSELAS.Toolchain-1.99.3/architecture_part/version_part.

2 Building phyCORE-i.MX31's root filesystem

2.1 Extracting

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

```
~# tar -zxf OSELAS.BSP-Phytec-phyCORE-11.tar.gz
~# cd OSELAS.BSP-Phytec-phyCORE-11
```

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

```
~/OSELAS.BSP-Phytec-phyCORE-11# ls -l

total 44
-rw-r--r--  1 jb users 4078 Dec  3 18:10 ChangeLog
-rw-r--r--  1 jb users 1313 Nov  1 13:31 Kconfig
-rw-r--r--  1 jb users 1101 Nov  4 21:05 TODO
drwxr-xr-x 10 jb users 4096 Jan 14 17:33 configs/
drwxr-xr-x  3 jb users 4096 Jan 14 15:08 documentation/
drwxr-xr-x  5 jb users 4096 Nov 13 12:30 local_src/
drwxr-xr-x  5 jb users 4096 Dec 15 10:19 patches/
drwxr-xr-x  6 jb users 4096 Jun  8  2008 projectroot/
drwxr-xr-x  3 jb users 4096 Nov  1 14:18 protocols/
drwxr-xr-x  4 jb users 4096 Jan  8 16:28 rules/
drwxr-xr-x  3 jb users 4096 Jan  7 08:55 tests/
```

Notes about some of the files and directories listed above:

ChangeLog Here you can read what has changed in this release. Note: This file does not always exist.

documentation If this BSP is one of our OSELAS BSPs, this directory contains the Quickstart you are currenly reading in.

configs A multiplatform BSP contains configurations for more than one target. This directory contains the platform configuration files.

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

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

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

tests Contains test scripts for automated target setup.

2.2 Selecting a Software Platform

First of all we have to select a software platform for the userland configuration. This step defines what kind of applications will be built for the hardware platform. The OSELAS.BSP-Phytec-phyCORE-11 comes with a predefined configuration we select in the following step:

```
~/OSELAS.BSP-Phytec-phyCORE-11# ptxdist select \ Enter
> configs/ptxconfig
info: selected ptxconfig:
     'configs/ptxconfig'
```

2.3 Selecting a Hardware Platform

Before we can build this BSP, we need to select one of the possible targets to build for. In this case we want to build for the phyCORE-i.MX31:

```
~/OSELAS.BSP-Phytec-phyCORE-11# ptxdist platform \ Enter
> configs/phyCORE-i.MX31-1.99.12-2/platformconfig
info: selected platformconfig:
      'configs/phyCORE-i.MX31-1.99.12-2/platformconfig'
```

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

```
found and using toolchain:
'/opt/OSELAS.Toolchain-1.99.3/arm-1136jfs-linux-gnueabi/
      gcc-4.3.2-glibc-2.8-binutils-2.19-kernel-2.6.27-sanitized/bin'
```

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

2.4 Selecting a Toolchain

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

```
~/OSELAS.BSP-Phytec-phyCORE-11# ptxdist toolchain \ Enter
> /opt/OSELAS.Toolchain-1.99.3/arm-1136jfs-linux-gnueabi/\ Enter
> gcc-4.3.2-glibc-2.8-binutils-2.19-kernel-2.6.27-sanitized/bin
```

2.5 Building the Root Filesystem

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

```
~/OSELAS.BSP-Phytec-phyCORE-11# ptxdist go
```

PTXdist does now automatically find out from the `selected_ptxconfig` and `selected_platformconfig` files which packages belong to the project and starts compiling their *targetinstall* stages (that one that actually puts the compiled binaries into the root filesystem). While doing this, PTXdist finds out about all the dependencies between the 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 `platform-phyCORE-i.MX31/root/` directory and a bunch of **.ipk* packets in the `platform-phyCORE-i.MX31/packages/` directory, containing the single applications the root filesystem consists of.

2.6 Building an Image

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

```
~/OSELAS.BSP-Phytec-phyCORE-11# ptxdist images
```

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

- **hd.img:** contains grub bootloader, kernel and root files in a ext2 partition. Mostly used for X86 target systems.
- **root.jffs2:** root files inside a jffs2 filesystem.
- **uRamdisk:** a u-boot loadable Ramdisk
- **initrd.gz:** a traditional initrd RAM disk to be used as initrdramfs by the kernel
- **root.ext2:** root files inside a ext2 filesystem.
- **root.squashfs:** root files inside a squashfs filesystem.
- **root.tgz:** root files inside a plain gzip compressed tar ball.

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

```
~/OSELAS.BSP-Phytec-phyCORE-11# ptxdist platformconfig
```

Then select the submenu `"image creation options"`. The generated image will be placed into `platform-phyCORE-i.MX31/images/`.

Only the content of the **.ipk* packages will be used to generate the image. This means that files which are put manually into the `platform-phyCORE-i.MX31/root/` will not be enclosed in the image. If custom files are needed for the target. Install it with ptxdist.

3 phyCORE-i.MX31 preparation

This step can be omitted if the boot loader is recent enough: For this BSP at least the *U-Boot-2.0.0-rc7* is required.

3.1 Updating the Bootloader

After building the whole BSP we copy the generated file `platform-phyCORE-i.MX31/images/u-boot-v2-image` to our configured tftp exported directory.

There are some differences in the handling depending on the current boot loader we will find on our new phyCORE-i.MX31.

3.1.1 Updating from a U-Boot-v1

When switching on the phyCORE-i.MX31 a U-Boot V1 states something similar to:
`U-Boot 1.2.0-phycore_mx31-3 (May 31 2007 - 15:30:49)`

The date may differ, but the important part is the first '1' in the *1.2.0* release. It will identify this U-Boot as a *V1* type.

First of all: Write down the MAC address of your card. You will need it immediately!

Now on the target side enter:

```
uboot> tftpboot 0x80000000 u-boot-v2-image
uboot> protect off 0xa0000000 +0x20000
uboot> erase 0xa0000000 +0x20000
uboot> cp.b 0x80000000 0xa0000000 $(filesize)
uboot> reset
```

Erasing the bootloader part of the flash is a very delicate step. If anything went wrong our phyCORE-i.MX31 is clobbered and can't be used any more. Only JTAG tools are able to bring the target to life again!

Now the new U-Boot-v2 comes up and its first question is:

```
no MAC address set for eth0. please enter the one found on your board:
```

Reenter the MAC address, and after it save it to the persistant memory.

```
uboot:/ saveenv
```

3.1.2 Updating from a U-Boot-v2

This update is only required when an older U-Boot-v2 than `2.0.0-rc7` is in use. We can omit this step, if our phyCORE-i.MX31 contains such a revision.

On the target side we enter:
`uboot:/ tftp u-boot-v2-image` to load this binary image into the RAM disk of our target.

To replace the current U-Boot in our target we enter the following commands:

```
uboot:/ unprotect /dev/self0
uboot:/ erase /dev/self0
uboot:/ cp u-boot-v2-image /dev/self0
uboot:/ protect /dev/self0
```

The new U-Boot revision is now stored in the NOR flash.

4 Booting Linux

Now that there is a root filesystem in our workspace we'll have to make it visible to the phyCORE-i.MX31. There are two possibilities to do this:

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

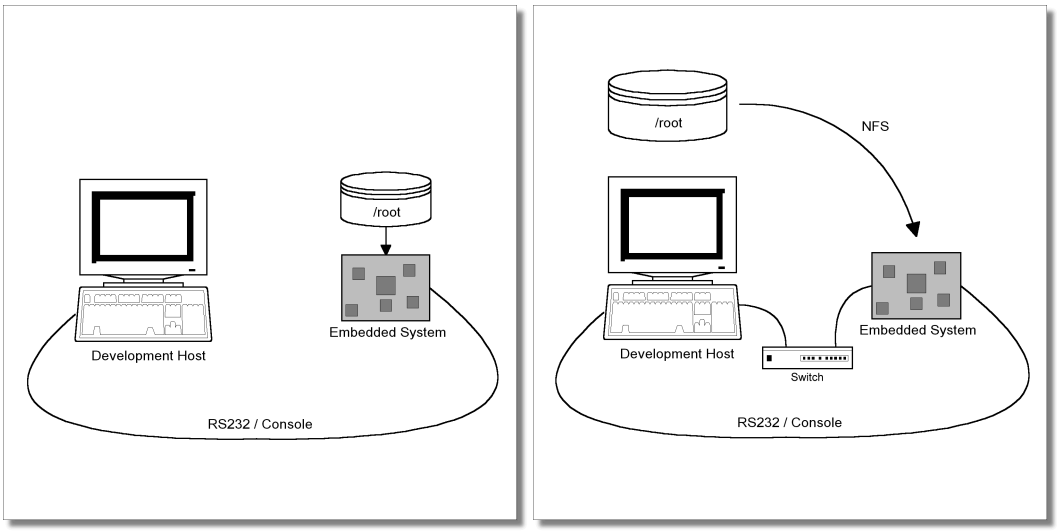

Figure 4.1: Booting the root filesystem, built with PTXdist, from the host via network and from flash.

Figure 4.1 shows both methods. The main method used in the OSELAS.BSP-Phytec-phyCORE-11 BSP is to provide all needed components to run on the target itself. The Linux kernel and the root filesystem is persistent in the media the target features. This means the only connection needed is the nullmodem cable to see what is happening on our target. We call this method *standalone*.

The other method is to provide all needed components via network. In this case the development host is connected to the phyCORE-i.MX31 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 its root filesystem via NFS (Network File System) from the original location of the `platform-phyCORE-i.MX31/root/` directory in our PTXdist workspace.

The OSELAS.BSP-Phytec-phyCORE-11 provides both methods. The latter one is especially for development purposes, as it provides a very quick turnaround while testing the kernel and the root filesystem.

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

4.1 Target Side Preparation

The phyCORE-i.MX31 uses U-Boot as its bootloader. U-Boot can be customized with environment variables and scripts to support any boot constellation. OSELAS.BSP-Phytec-phyCORE-11 comes with a predefined environment setup to easily bring up the phyCORE-i.MX31.

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 that some of the values of these U-Boot 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

```
~/OSELAS.BSP-Phytec-phyCORE-11# 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.

When everything is set up, we can "Exit" the dialog and and save our new settings.

Now the command

```
~/OSELAS.BSP-Phytec-phyCORE-11# ptxdist test setenv
```

will automatically set up a correct default environment on our phyCORE-i.MX31. We have to powercycle our target to make this step happen.
It should output lines like these when it was successful:

```
==============================
Please power on your board now!
==============================

Logging into U-Boot.....................OK
Setting new environment.................OK
Test finished successfully.
```

Note: If it fails, reading `platform-phyCORE-i.MX31/test.log` will give further information about why it has failed. Also extending the command line shown above by a `--debug` can help to see whats going wrong.

Users reported this step could fail if the Linux system running PTXdistis a virtual machine as guest in an operating system from Redmont. In this case it seems at least one of the two OSes is eating up characters sent to the serial line. Pengutronix recommends running PTXdiston a real Linux system.

4.2 Stand-Alone Booting Linux

To use the the target standalone, the rootfs has to be made persistent in one of the onboard suported media of the phyCORE-i.MX31. The following sections describe the steps necessary to bring the rootfs into the onboard NOR type 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-i.MX31 can work independently from the development host. We can "cut" the network (and serial cable) and the phyCORE-i.MX31 will continue to work.

4.2.1 Development Host Preparations

On the development host a TFTP server has to be installed and configured. The exact method to do so 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 into 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 change it. The boardsetup scripts coming with this BSP expect write permission in TFTP directory!

We can run a simple:

```
~# touch /tftpboot/my_file
```

to test if we have permissions to create files in this directory. If it fails we have to ask the administrator to grant these permissions.

Note: We must `/tftpboot` part of the command above with our local settings.

4.2.2 Preparations on the Embedded Board

To boot phyCORE-i.MX31 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-i.MX31's flash memory.

But first we must create the new root filesystem image prepared for its usage on the phyCORE-i.MX31:

```
~/OSELAS.BSP-Phytec-phyCORE-11# ptxdist images
```

To simplify this step, OSELAS.BSP-Phytec-phyCORE-11 comes with an automated setup procedure for this step. To use this procedure we run the command:

```
~/OSELAS.BSP-Phytec-phyCORE-11# 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-i.MX31. It only works if we previously have set up the environment variables successfully (described at page 19).
The command should output lines like this when it was successful:

```
==============================
Please power on your board now!
==============================

Logging into U-Boot.....................OK
Flashing kernel.........................OK
Flashing rootfs.........................OK
Flashing oftree.........................OK
Test finished successfully.
```

Note: If it fails, reading `platform-phyCORE-i.MX31/test.log` will give further information about why it has failed.

4.2.3 Booting the Embedded Board

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

```
~/OSELAS.BSP-Phytec-phyCORE-11# ptxdist test boot

==============================
Please power on your board now!
==============================

Checking for U-Boot.....................OK
Checking for Kernel.....................OK
Checking for init.......................OK
Checking for login......................OK
Test finished successfully.
```

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

Note: If it fails, reading `platform-phyCORE-i.MX31/test.log` will give further information about why it has failed.

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.

4.3 Remote-Booting Linux

The next method we want to try after building a root filesystem is the network-remote boot variant. This method is especially intended for development as everything related to the root filesystem happens on the host only. It's the fastest way in a phase of a project, where things are changing frequently. Any change made in the local `platform-phyCORE-i.MX31/root/` directory simply "appears" on the embedded device immediately.

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.

4.3.1 Development Host Preparations

If we already have booted the phyCORE-i.MX31 locally (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 change it. If we don't want to change the permission or if its disallowed to change anything, the `sudo` command may help.

```
~/OSELAS.BSP-Phytec-phyCORE-11# sudo cp platform-phyCORE-i.MX31/images/linuximage
/tftpboot/uImage-pcm037
```

The NFS server is not restricted to a certain filesystem location, so all we have to do on most distributions is to modify the file `/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-i.MX31 with the default environment at page 19. So there is no additional preparation required here.

4.3.3 Booting the Embedded Board

The default environment settings coming with the OSELAS.BSP-Phytec-phyCORE-11 has the possibility to boot from the internal flash or from the network. Configuration happens in the file `/env/config`. As U-Boot-v2 uses a full shell like console you can edit this file to configure the other scripts (the boot script for example).

To edit this configuration file we run the `edit` command on it:

```
uboot:/ edit /env/config
```

We move to the lines that define the *kernel_loc* and *rootfs_loc* variables. They can be defined to *nor*, *nand* or *net*. *nor* let the boot script load everything from the internal NOR flash memory, *nand* from the NAND flash memory.

In this example we change it to *net* to load all parts from the network. When we do that, we also have to configure the network setup a few lines above in this file. We setup these values to the network we want to run the phyCORE-i.MX31.

Leaving this editor with saving the changes happens with *CTRL-D*. Leaving it without saving the changes happens with *CTRL-C*.

Note: Saving here means the changes will be saved to the RAM disks U-Boot-v2 uses for the environment. To store it to the persistent memory, an additional `saveenv` command is required.

Now its time to boot the phyCORE-i.MX31. To do so, simply run:

```
uboot:/ boot
```

This command should boot phyCORE-i.MX31 into the login prompt.

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

5 Accessing Peripherals

The following sections provide an overview of the supported hardware components and their corresponding operating system drivers. Further changes can be ported on demand of the customer.

Phytec's phyCORE-i.MX31 starter kit consists of the following individual boards:

1. The phyCORE-i.MX31 module itself (PCM-037), containing the i.MX31, RAM, flash and several other peripherals.
2. The starter kit baseboard (PCM970).

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-mx3/pcm037.c` for the CPU module

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.

Modern kernels offer a method to define flash partitions on the kernel command line, using the `mtdparts` command line argument:

```
physmap-flash.0:256k(uboot)ro,128k(ubootenv),2M(kernel),-(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.

From userspace the NOR flash partitions can be accessed as

- /dev/mtdblock0 (e.g. U-Boot partition)
- /dev/mtdblock1 (e.g. U-Boot environment partition)
- /dev/mtdblock2 (e.g. Kernel partition)
- /dev/mtdblock3 (e.g. Linux rootfs partition)

Note: This is an example only. The partitioning on our phyCORE-i.MX31 target can differ from this layout.

Only the `/dev/mtdblock3` on the phyCORE-i.MX31 has a filesystem, so the other partitions 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 NAND Flash

The phyCORE-i.MX31 module comes with a 64MiB NAND memory to be used as an additional media to store applications and their data files. This type of media will be managed by the JFFS2 filesystem. This filesystem uses compression and decompression on the fly, so there is a change to bring more than 64MiB of data into this device.

NOTE: JFFS2 has a disadvantage: We cannot use the mmap()-API to manipulate data on a mapped file.

5.2.1 NAND Usage

From userspace the NAND flash partitions can be accessed as

- /dev/mtdblock4 (U-Boot partition)
- /dev/mtdblock5 (U-Boot environment partition)
- /dev/mtdblock6 (Kernel partition)
- /dev/mtdblock7 (Linux rootfs partition)

5.2.2 NAND Preparation

On a fresh phyCORE-i.MX31 the NAND memory can be unformatted. In this case the mount will fail. To prepare it for later usage, we must prepare it by erasing the whole memory.

```
~# flash_eraseall /dev/mtd7
```

After erasing, mounting is possible via:

```
~# mount /media/nand
```

Note: Mounting this memory the first time after erasing it can take a few seconds. In this case the JFFS2 filesystem does its own preparation of the memory in the background.

Now this filesystem is available through `/media/nand` and can be used like any other filesystem.

5.3 SRAM Memory

The phyCORE-i.MX31 is shipped with a 512 kiB SRAM. We can use it as plain memory or we can put a filesystem on top of it and use it as a regular part of the root filesystem.

- /dev/mtdblock8 is the full SRAM partition.

To create a filesystem we type:

```
~# mkfs.minix -n 30 /dev/mtdblock8
704 inodes
2048 blocks
Firstdatazone=26 (26)
Zonesize=1024
Maxsize=268966912
```

Note: The output of the `mkfs.minix` my differ due to different SRAM sizes.

And to mount it:

```
~# mount -t minix /dev/mtdblock8 /mnt
```

5.4 Serial TTYs

The i.MX31 SoC supports up to 4 so called UART units. On the phyCORE-i.MX31 two UARTs are routed to the connectors and can be used in user's application.

- `ttymxc0` at connector P1 (bottom connector) used as the main kernel and control console.
- `ttymxc2` at connector P1 (top connector). Unused in this BSP

5.5 Network

The phyCORE-i.MX31 module has an SMSC SMC911x 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.

5.6 SPI Master

The phyCORE-i.MX31 board supports an SPI bus, based on the i.MX31'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-i.MX31.

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

This BSP currently supports one dedicated SPI bus. Its used to control the external so called PMIC, the main peripheral controller.

5.7 Touch

A simple test of this feature can be run with:

```
~# ts_calibrate
```

to calibrate the touch and with:

```
~# ts_test
```

to do a simple application using this feature.

5.8 I²C Master

The i.MX31 processor based phyCORE-i.MX31 supports a dedicated I²C controller onchip. The kernel supports this controller as a master controller.

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

5.8.1 I²C Realtime Clock RTC8564

Due to the Real Time Clock framework of the kernel the RTC8564 clock chip can be accessed using the same tools as for any other real time clock.

Date and time can be manipulated with the `hwclock` tool, using the `-w` (systohc) and `-s` (hctosys) options. For more information about this tool refer to the manpage of hwclock.

OSELAS.BSP-Phytec-phyCORE-11 tries to set up the date at system startup. If there was a powerfail `hwclock` will state:

```
pcf8564 1-0051: low voltage detected, date/time is not reliable.
pcf8564 1-0051: retrieved date/time is not valid.
```

In this case set the date manually (see `man date`) and run `hwclock -w -u` to store the new date into the RTC8564.

5.8.2 I²C device 24W32

This device is a 4 kiB non-volatile memory for general purpose usage.

This type of memory is accessible through the sysfs filesystem. To read the EEPROM content simply `open()` the entry `/sys/bus/i2c/devices/1-0052/eeprom` and use `fseek()` and `read()` to get the values.

5.9 Framebuffer

This driver gains access to the display via device node `/dev/fb0`. For this BSP the Hitachi TX09D70VM1CCA display with a resolution of 240x320 is supported.

A simple test of this feature can be run with:

```
~# fbtest
```

This will show various pictures on the display.

You can check your framebuffer resolution with the command

```
~# fbset
```

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

Earlier system revisions are using the SHARP LQ035Q7DH06 display instead. To continue using this display U-Boot's environment or the BSP must be changed.

In order to continue using the predecessor display the kernel must be setup with the `video=mx3fb:Sharp-LQ035Q7` parameter.

If the target system is already configured, this parameter can be changed from inside the running U-Boot: Modify the file `/env/config` and replace the current `video=mx3fb:TX090` with the string shown above.

To modify the BSP to make this change persistant we can simply modify the `configs/phyCORE-i.MX31-1.99.12-2/u-boot` and also replace the `video=mx3fb:TX090` by the `video=mx3fb:Sharp-LQ035Q7` string. With this modification targe's auto setup will do the right thing.

5.10 USB Host Controller

The i.MX31 CPU embedds a USB 2.0 EHCI controller that is also able to handle low and full speed devices (USB 1.1).

The OSELAS.BSP-Phytec-phyCORE-11 includes support for mass storage devices and keyboards. Other USB related device drivers must be enabled in the kernel configuration on demand.

Due to `udev`, connecting various mass storage devices get unique IDs and can be found in `/dev/disks/by-id`. These IDs can be used in `/etc/fstab` to mount different USB memory devices in a different way.

5.11 OneWire Interface

"There is something like 1W existing in this universe."

As the real support for this kind of devices is currently very broken, only a direct access is provided. Any detected 1W device will be mapped to the sysfs filesystem.

For example a connected temperature sensor could be accessed via this entry:

`/sys/bus/w1/devices/10-000801018ed7/w1_slave`

A simple `cat` command can give you the follwing output:

```
root@phyCORE:~ cat /sys/bus/w1/devices/10-000801018ed7/w1_slave
2e 00 4b 46 ff ff 0e 10 91 : crc=91 YES
2e 00 4b 46 ff ff 0e 10 91 t=22875
```

5.12 MMC/SD Card

The phyCORE-i.MX31 supports *Secure Digital Cards* and *Multi Media Cards* in conjunction with its PCM970 to be used as general purpose blockdevices. These devices can be used in the same way as any other blockdevice.

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

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

`/dev/mmcblk0pY`

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

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

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

5.13 CAN Bus

The phyCORE-i.MX31 provides a CAN feature, 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.

Configuration happens within the script `/etc/network/can-pre-up`. This script will be called when `/etc/init.d/networking` is running at system start up. To change default used bitrates on the target change the variables **CAN_0_BITRATE** and/or **CAN_1_BITRATE** in `/etc/network/can-pre-up`.

For a persistent change of the default bitrates change the local `projectroot/etc/network/can-pre-up` instead and rebuild the BSP.

The Socket-CAN API is still work in progress and was submitted to the upstream kernel maintainers in part only.

5.13.1 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.

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
     inet addr:127.42.23.180  Mask:255.255.255.0
     UP RUNNING NOARP  MTU:16  Metric:1
     RX packets:35948 errors:0 dropped:0 overruns:0 frame:0
     TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
     collisions:0 txqueuelen:10000
     RX bytes:243744 (238.0 KiB)  TX bytes:2 (2.0 B)
     Interrupt:145 Base address:0x900
```

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
cano	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 5.1: CAN interface information

[1] ISO 11898/11519

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 cano can now be changed:

```
~# canconfig can0 bitrate 250000
```

and the interface is started with

```
~# ifconfig can0 up
```

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.

6 Special Notes

6.1 Analysing the CAN Bus Data Transfer

The OSELAS.BSP-Phytec-phyCORE-11 BSP comes with the standard *pcap* library and `tcpdump` tool. Both are capable of analyzing CAN data transfer wich includes time stamping.

We set up the CAN interface(s) as usual and use it in our application. With `tcpdump` we can sniff at any point of time the data transferred on the CAN line.

To do so, we simply start tcpdump:

```
~# tcpdump -i can0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on can0, link-type LINUX_CAN (Linux CAN), capture size 68 bytes
```

Whenever there is any traffic on the line, `tcpdump` will log it to stdout. We will generate some traffic by using the `cansend` command:

```
~# cansend can0 -i 0x12 0x0f 0xf0 0x10 0x01
```

For this data, `tcpdump` will output:

```
00:15:52.482066 CAN Out ID:00000012 PL_LEN:4 PAYLOAD: 0x0f 0xf0 0x10 0x01
```

The log `tcpdump` generates consist of six fields:

1. `00:15:52.482066` is the timestamp this data was on the line. Its format is HH:MM:SS:TTTTTT, with TTTTTT as second's fraction
2. `CAN` interface type
3. `Out` message'S data direction on this interface
4. `ID:00000012` CAN message ID
5. `PL_LEN:4` byte count of message data
6. `PAYLOAD: 0x0f 0xf0 0x10 0x01` the payload data

Some notes:

- The data direction field could be `Out` or `In`
- The CAN message ID encodes some additional info into higher bit values:

  - Bit 31 encodes an extended frame. If this bit is set, an extended message frame was on the line
  - Bit 30 encodes an RTR frame. If this bit is set, a remote transmission message frame was on the line

  The message ID resides in the lower bits of this field
- The PAYLOAD field could be empty, when there were no data elements in the message

6.2 Using the NAND Flash for Root Filesystem

To use the phyCORE-i.MX31 NAND memory for root filesystem usage, some preparations are required.

1. Partitioning the NAND memory (optional step)
2. Erasing that part of the NAND memory that should act as the root file partition
3. Generating the root filesystem image for writing into the NAND memory

6.2.1 Partitioning

If the NAND memory is the sole flash memory on the phyCORE-i.MX31, we must provide a so called `mtdparts` kernel parameter to describe the partition layout of the NAND memory. This parameter will be forwarded by U-Boot to the starting kernel.

All we need in this case is a kernel parameter like that:
`mtdparts=\"NAND 64MiB 3,3V 8-bit:128k(uboot)ro,128k(ubootenv),2M(kernel),-(root)\"`

If there is an additional flash memory on the phyCORE-i.MX31 in use (and also partitioned), this kernel parameter may already exist. In this case we must extend the line with the new NAND partition description:
`mtdparts=<old-description>;`

`\"NAND 64MiB 3,3V 8-bit:128k(uboot)ro,128k(ubootenv),2M(kernel),-(root)\"`

Note: Before continuing with the next step, the system must be booted with the new or extended `mtdparts` kernel command line.

6.2.2 Erasing the Root Partition

If not already done the dedicated root partition must be erased before any new data can be written. This will be done at target side by

```
~# flash_eraseall /dev/mtd7
```

Note: Do not use the `flash_eraseall` command with the `-j` command line option for NAND memories. `-j` is intended only to be used for NOR memories!

Now this partition is ready for use. The JFFS2 filesystem is able to use this erased part of the NAND memory. So we could mount this partition and store some data on it. But we want to use it as a root filesystem the next time we boot this system. So we need many files to store on this partition. It's not a good idea to do it manually. We let PTXdist generate an image that includes all required directories and files.

6.2.3 Writing a Root Filesystem Image

Due to the JFFS2 filesystem usage and different NANDs the image must meet some physical restrictions. For example the *erase block size* of the NAND must be known while generating the image. Some parameters must be set up in the PTXdist menus, before the image can be generated.

We run `ptxdist platformconfig`, navigate to `image creation options`, `Generate images/root.jffs2` and change the `Erase Block Size` to the sector size of the used NAND memory. In the case of phyCORE-i.MX31 this sector size is 16384 (=16 kiB). We also must enter a `-n` entry into `extra arguments passed to mkfs.jffs2` to generate a proper image for NAND memory usage.

After changing the menu settings a `ptxdist images` will generate the new root filesystem image. We have to transfer this image now to the target via our favoured method (NFS, FTP or something else). Writing this image to the NAND memory is easily done by:

```
~# nandwrite -j /dev/mtd7 <root-filesystem-image>
```

6.2.4 Booting into the NAND based Root Filesystem

To make the starting kernel use the new NAND partition for its root filesystem, we only have to change the `root` kernel parameter to:

```
root=/dev/mtdblock7
```

To be successful with this step, the kernel itself must be prepared:

- the driver to access the NAND memory must be statically linked
- the JFFS2 filesystem support must be statically linked
- a second root filesystem related kernel parameter must be also present: `rootfstype=jffs2`

7 Getting help

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

7.1 Mailing Lists

7.1.1 About PTXdist in particular

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

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

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

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

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

7.1.2 About embedded Linux in general

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

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

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

7.2 News Groups

7.2.1 About Linux in embedded environments

This is an English newsgroup for general questions about Linux in embedded environments.

comp.os.linux.embedded

7.2.2 About general Unix/Linux questions

This is a German newsgroup for general questions about Unix/Linux programming.

de.comp.os.unix.programming

7.3 Chat/IRC

About PTXdist in particular

irc.freenode.net:6667

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

7.4 phyCORE-i.MX31 Support 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

7.5 Commercial Support

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

Contact us at:

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

or by electronic mail:

sales@pengutronix.de