OSELAS.BSP( )
Phytec phyCORE-MPC5200

Quick Start Manual

http://www.oselas.com

$Rev: 460 $ $Date: 2006-06-27 13:08:36 +0200 (Tue, 27 Jun 2006) $

Contents

1 PTXdist Installation

1.1 Building Blocks

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

PTXdist consists of the following parts:

- **The `ptxdist` program**, which is installed on the development host during the installation process. `ptxdist` is called to trigger any actions, like building a software packet, cleaning a tree etc. Usually the `ptxdist` program is used in a *workspace* directory, which contains all project relevant files.
- **A configuration system.** The config system is used to customize a *configuration*, which contains information about which packages have to be built and which options are selected.
- **Patches.** Due to the fact that some upstream packages are not bug free – especially with regard to cross compilation – it is often necessary to patch the original software. PTXdist contains a mechanism to automatically apply patches to packages. The patches are bundled into a separate archive. Nevertheless, they are necessary to build a working system.
- **Package descriptions.** For each software component there is a "recipe" file, specifying which actions have to be done to prepare and compile the software. Additionally, packages contain their configuration sniplet for the config system.

1.2 Prerequisites

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

1.3 Installation from the Sources

To install PTXdist, three archives have to be extracted:

- `ptxdist-0.10.4.tgz`, containing the software
- `ptxdist-patches-0.10.4.tar.gz`, containing all patches for upstream packets
- `OSELAS.BSP-phyCORE-MPC5200B-1.tar.gz`, containing the board support package (project) for the Phytec phyCORE-MPC5200B board.

The PTXdist and patches packets have to be extracted into some temporary directory, for example the `local` directory in the user's home. If this directory does not exist, we have to create it and change into it

```
~# cd
~# mkdir local
~# cd local
```

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

```
~/local# mount /media/cdrom
~/local# tar -zxf /media/cdrom/
     OSELAS.BSP-phyCORE-MPC5200B-1/
     ptxdist-0.10.4.tgz
~/local# tar -zxf /media/cdrom/
     OSELAS.BSP-phyCORE-MPC5200B-1/
     ptxdist-0.10.4-patches.tar.gz
```

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

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

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

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

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

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

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

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

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

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

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

```
~/local/ptxdist-0.10.4# make
```

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

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

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

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

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

2 Toolchain

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

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

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

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

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

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

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

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

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

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

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

2.1 Using Existing Toolchains

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

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

2.2 Toolchain Building

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

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

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

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

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

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

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

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

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

Now we clone the PowerPC toolchain project for the phyCORE-MPC5200B. "Cloning" means that we create a local working copy of the project shipped with PTXdist:

```
~/work# ptxdist clone
    toolchain_powerpc-unknown-linux-gnu-4.0.2_glibc-2.3.6_linux_2.6.13
    cross-toolchain
```

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

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

```
~/work# cd cross-toolchain
~/work/cross-toolchain# ptxdist go
```

At this stage we have to go to our boss and tell him that it's probably time to go home for the day. Even on reasonably fast machines the time to build a cross toolchain is something like around 30 minutes up to one hour. Another possibility is to read the next chapters of this manual, to find out how to start a new project.

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

3 Building phyCORE-MPC5200B's root filesystem

3.1 Our First Project

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-MPC5200B-1 for the phyCORE-MPC5200B:

```
~$ tar -zxf OSELAS.BSP-phyCORE-MPC5200B-1.tar.gz
~$ cd OSELAS.BSP-phyCORE-MPC5200B-1
```

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

```
~/OSELAS.BSP-phyCORE-MPC5200B-1$ ptxdist toolchain
     toolchain
     /opt/ptxdist-0.10.4/gcc-4.0.2-glibc-2.3.6/
     powerpc-unknown-linux-gnu/bin
```

3.2 Compiling the Root Filesystem

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

```
~/OSELAS.BSP-phyCORE-MPC5200B-1$ ptxdist go
```

PTXdist does now automatically find out from the `ptxconfig` file which packages belong to the projects and starts compiling their "targetinstall" stages (that one that actually puts the compiled binaries into the root filesystem). While doing this, PTXdist finds out about all the dependencies between the packets and brings them into the correct order.

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

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

3.3 Building a Flash Image

PTXdist can build a flash image from the `root/` tree. As all necessary parameters for the phyCORE-MPC5200B are configured in the `ptxconfig` file, all we need to do is to run

```
~/OSELAS.BSP-phyCORE-MPC5200B-1$ ptxdist images
```

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

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

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

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

4 Booting Linux

Now that there is a root filesystem in our workspace we'll have to make it visible to the phyCORE-MPC5200B. There are two possibilities to do this:

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

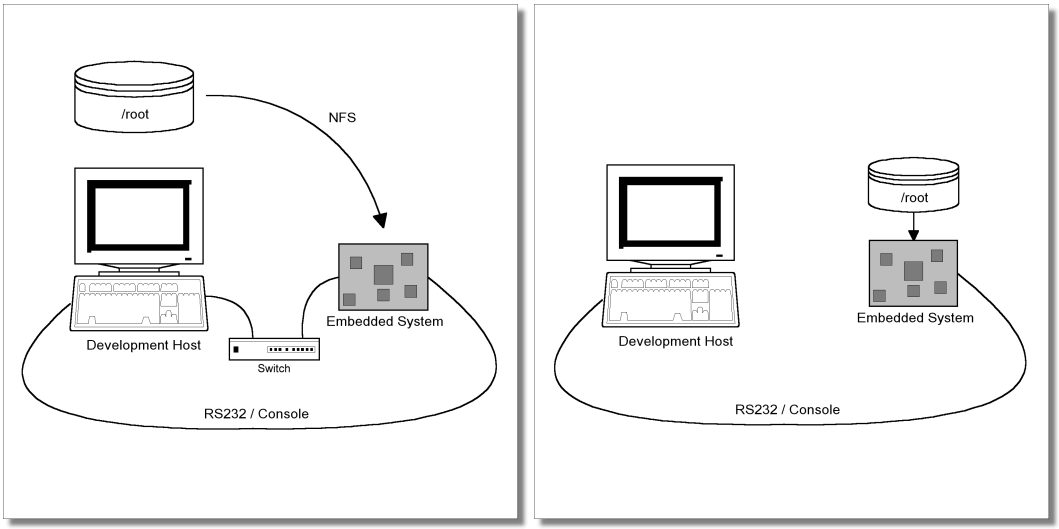

Figure 4.1: *Booting the root filesystem, built with PTXdist, from the host via network and from flash.*

Figure 4.1 shows both methods. On the left side the development host is connected to the phyCORE-MPC5200B with a serial nullmodem cable and via ethernet; the embedded board boots into the bootloader, then issues a TFTP request on the network and boots the kernel from the TFTP server on the host. Then, after decompressing the kernel into the RAM and starting it, the kernel mounts it's root filesystem via NFS from the original location of the root/ directory in our PTXdist workspace.

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

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

4.1 Target Side Preparation

The phyCORE-MPC5200B uses U-Boot as its bootloader. U-Boot can be customised with environment variables to support any boot constellation. OSELAS.BSP-phyCORE-MPC5200B-1 comes with a predefined environment setup to easily bring up the phyCORE-MPC5200B.

4.2 Default U-Boot environment

4.2.1 Stand alone

This could be the default U-Boot environment, if we boot stand alone from flash.

Variable	Value	Meaning
autostart	yes	after timeout U-Boot runs anything in `bootcmd`
baudrate	115200	baudrate to use in the serial console
ethaddr	00:50:C2:3B:A8:AA	unique hardware address
bootdelay	2	time to wait for any key until auto boot starts
bootcmd	run setup_bootargs; fsload /boot/uImage; bootm	commands to run after timeout or when `boot` is entered
setup_bootargs	setenv bootargs root=/dev/mtdblock0 rootfstype=jffs2 console=ttyPSC2,$(baudrate)n8 $(mtdparts)	`bootargs` contents is forwarded to the starting kernel
mtdparts	mtdparts=phys_mapped_flash:15m (jffs2),256k(uboot)ro,-(space)	Partition definition defined here for easier modification, used in `bootargs`
mtdids	nor0=phys_mapped_flash	Reference U-Boot's names with Linux's names
partition	nor0,0	define the partition where all file operations should work on

4.2.2 Network based

This could be the default U-Boot environment, if we boot all things needed (kernel image and root filesystem) from our host.

Variable	Value	Meaning
autostart	yes	after timeout U-Boot runs anything in `bootcmd`
baudrate	115200	baudrate to use in the serial console
ethaddr	00:50:C2:3B:A8:AA	unique hardware address
bootdelay	2	time to wait for any key until auto boot starts
hostname	ppc5200	used for kernel's network setup
bootcmd	run setup_bootargs; tftpboot	commands to run after timeout or when `boot` is entered
setup_bootargs	setenv bootargs root=/dev/nfs rw nfsroot=$(serverip):$(nfsrootfs) ip=$(ipaddr):$(serverip):$(gatewayip) ::$(hostname)::off console=ttyPSC2,$(baudrate)n8 $(mtdparts)	`bootargs` contents is forwarded to the starting kernel
mtdparts	mtdparts=phys_mapped_flash:15m (jffs2),256k(uboot)ro,-(space)	Partition definition defined here for easier modification, used in `bootargs`
netmask	255.255.255.0	netmask to use
serverip	192.168.23.1	IP of the server to load the kernel image and to mount the NFS root filesystem (if no DHCP is used)
ipaddr	192.168.23.108	Targets own IP (if no DHCP is used)
gatewayip	192.168.23.1	Gateway to use (if no DHCP is used)
nfsrootfs	/ptx/work/jbe/rootnptl	Path to mount as root filesystem on server
bootfile	uImage	Name of the file to load as kernel image from server

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 `PCM026>` .

"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-MPC5200B. It should output a line like this when it was successfull:

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

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

We now must restart the phyCORE-MPC5200B to activate the new environment settings. Then we should run the `ping` command on the target's ip address to check if the network settings are working correctly on the target.

4.3 Remote-Booting Linux

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

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

4.3.1 Development Host Preparations

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

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

```
~# sudo cp images/linuximage /tftpboot/uImage-pcm026
```

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

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

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

4.3.2 Preparations on the Embedded Board

We already provided the phyCORE-MPC5200B with the default environment at page 15. So there is no additional preparation required here.

4.3.3 Booting the Embedded Board

The default environment coming with the OSELAS.BSP-phyCORE-MPC5200B-1 has a predefined script for booting from NFS. To use it, we can simple enter

```
PCM026> run bootcmd_net
```

This command should boot phyCORE-MPC5200B 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:

```
PCM026> setenv bootcmd 'run bootcmd_net'
```

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

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

4.4 Stand-Alone Booting Linux

Usually, after working with the NFS-Root system for some time, the rootfs has to be made persistent in the onboard flash of the phyCORE-MPC5200B, 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-MPC5200B can work independently from the development host. We can ”cut” the network (and serial cable) and the phyCORE-MPC5200B will continue to work.

4.4.1 Development Host Preparations

If we already booted the phyCORE-MPC5200B remotly (as described in the privious section) all of the development host preparations are done.

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

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

4.4.2 Preparations on the Embedded Board

To boot phyCORE-MPC5200B 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-MPC5200B’s flash memory. To simplify this, OSELAS.BSP-phyCORE-MPC5200B-1 comes with an automated setup procedure for this step.

To use this procedure run the command

```
~# ptxdist test flash
```

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

This command will automatically write a root filesystem to the correct flash partition on the phyCORE-MPC5200B. It only works, if we priviously setup the environment variables

successfully (described at page 15).
The command should outputs a line like this when it was successfull:

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

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

4.4.3 Booting the Embedded Board

The default environment coming with the OSELAS.BSP-phyCORE-MPC5200B-1 has a predefined script for booting stand-alone. To use it, we can simple enter

```
PCM026> run bootcmd_flash
```

This command should boot phyCORE-MPC5200B 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:

```
PCM026> setenv bootcmd 'run bootcmd_flash'
```

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

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

5 Acessing Peripherals

Phytec's phyCORE-MPC5200B starter kit consists of the following individual boards:

1. The phyCORE-MPC5200B module itself, containing the MPC5200B processor, RAM, flash and several other peripherals.
2. The starter kit baseboard.

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 the kernel source tree at:

1. `arch/ppc/platforms/pcm026.c` for the processor module

In fact, software re-use is one of the most important features of the Linux kernel and especially of the PowerPC 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 Memory

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:15m(jffs2),256k(uboot)ro,-(space)
```

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-MPC5200B-1 is being used.

From userspace the flash partitions can be accessed as

- /dev/mtdblock0 (Linux rootfs partition)
- /dev/mtdblock1 (U-Boot partition)
- /dev/mtdblock2 (spare due to reset vector restrictions)

Only `/dev/mtdblock0` 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 CAN Bus

The MPC5200B has an OnChip based CAN controller with two channels, which is supported by drivers using the (currently work-in-progress) proposed Linux standard CAN framework "Socket-CAN". Using this framework, CAN interfaces can be programmed with the BSD socket API.

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

5.2.1 Socket-CAN

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

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

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

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

5.2.2 Starting and Configuring Interfaces from the Command Line

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

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

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

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

[1] ISO 11898/11519

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

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

The baudrate for can0 can now be changed:

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

and the interface is started with

```
~# ifconfig can0 up
```

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

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

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

5.2.3 Using the CAN Interfaces from the Command Line

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

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

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

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

To receive CAN messages run the `candump` command:

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

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

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

5.2.4 Programming CAN Interfaces in C

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

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

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

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

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

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

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

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

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

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

struct sockaddr_can adr;
```

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

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

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

```
struct ifreq ifr;

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

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

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

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

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

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

5.2.5 Sending CAN Messages

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

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

In our example, the call is

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

5.2.6 Receiving CAN Messages

To receive messages, we call the read function:

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

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

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

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

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

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

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

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

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

5.2.7 Closing Interfaces & Further Reading

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

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

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

```
close(sockfd);
```

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

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

5.2.8 Getting help

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

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

Search for the lists:

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

5.3 Network

The phyCORE-MPC5200B module has an OnChip ethernet chip, 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.

5.4 FPGA support

The phyCORE-MPC5200B is shipped with an Altera FPGA of type Cyclone II EP2C8F256C8N. This FPGA is a general purpose device with no special dedication when shipped. Its up to you to blow life in it with your own firmware, for example to do some high speed signal processing. The OSELAS.BSP-phyCORE-MPC5200B-1 provides a mechanism to load the FPGA firmware while the whole system is already running.

5.4.1 General

The source archive contains the files `fpga.c` and `Makefile`. During the build process, a module named `fpga.ko` is created. Here's how to achieve this:
We locate the makefile after unpacking and change the KDIR variable in this makefile to the location of our own kernel source directory and then start compiling. We make sure to include the path to the binary of the crosscompiler in our searchpath.

Putting the mechanism to work is as easy as loading the module with parameter `firmware=<filename.rbf>`.
The firmware must be in RBF (Altera raw binary format) and stored in the place where the firmware agent of our embedded system will expect it. (It depends on the configuration of our udev daemon. Usually it should be `/lib/firmware/`)

In practice, the module is loaded like in this example:

```
~# insmod fpga.ko firmware=my_own_firmware.rbf
```

The module assumes specific hardware connections between processor and FPGA which are shown in the following table. The user does not need to take care about these when using the standard hardware.

See phyCORE-MPC5200B's datasheet for FPGA's pin assignment.

FPGA Pin	MPC5200B Pin	Pin Loc.	Pin Name	Pin Conf.
nCONFIG	UART6_CTS_TTL	PSC6_1	GPIO_WKUP_5	out
CONFIG_DONE	UART6_RXD_TTL	PSC6_0	GPIO_WKUP_4	in
nSTATUS	GPIO7		GPIO_WKUP_7	in
DCLK	UART6_RTS_TTL	PSC6_3	GPIO_IRDA_1	out
DATA0	UART6_TXD_TTL	PSC6_2	GPIO_IRDA_0	out

Table 5.1: *FPGA pin connections*

5.4.2 Demo

The BSP is shipped with a simple demo. To download it into the FPGA you should enter:

```
~# insmod /home/fpga.ko firmware=MPC_to_WBSlave.rbf
```

It supports five 32 bit registers at the baseaddess CS1 (chip select) is using.

- Register 0 at CS1 baseaddress + 0x00
- Register 1 at CS1 baseaddress + 0x04
- Register 2 at CS1 baseaddress + 0x08
- Register 3 at CS1 baseaddress + 0x0C
- Register 4 at CS1 baseaddress + 0x10 (mirrored up to the end of CS1 address space)

In the first step to access the FPGA registers we do not need a device driver. Due to PowerPC architecture maps devices in memory we can use a small tool called `memedit` for testing.

The most important part is, where the physical baseaddress of processor's CS1 is located. Usually we can skip this part, the setup is always the same: Physical address starts at `0xe0000000` and ends at `0xe1ff0000`. But in case of trouble, it could be helpfull to check if the CS1 is on its right location:

To gain access to processor's control registers we use the `memedit` tool on the target:

```
~# memedit /dev/mem
```

This tool works interactive, so the next step is to map some physical memory space into the memory of the `memedit` process.

```
-> map 0xf0000000 0x1000
```

`0xf0000000` is the base address of the MBAR area, where all processor and its chipset internal control registers are mapped. Next we read back the CS1 configuration register (see processor's manual for further descriptions about location and meaning of these type of registers).

```
-> md 0x0c 0x10
0000000c:  0000e000 0000e1ff 0000fde0 0000fde7 ..............
```

`0x0c` is an offset to the MBAR (=`0xf0000000`) and is the start address of CS1. We read back values `0x0000e000` (=start) and `0x0000e1ff` (=end). The values have to be shifted by 16 to get the real physical address. In this case it starts at `0xe0000000` and ends at `0xe1ff0000`.

Unmap the MBAR area first, to do the next step (do not leave `memedit`):

```
-> unmap
```

After ensuring the physical address area of CS1 we now can map this area to gain access to the FPGA itself. When our mapping of CS1 differs we have to enter other values here. The default should be:

```
-> map 0xe0000000 0x1ff0000
```

Now we can read and write into this area and - if the FPGA is ready to work - we can access its registers.

```
-> md 0x0 0x20
00000000:  00000000 00000000 00000000 00000000 ................
00000010:  00000000 00000000 00000000 00000000 ................
```

After system reset everything should be zero. We do some writings to check the registers:

```
-> mm 0x0 0x10
new values:
0x00000000:  00000010
-> mm 0x4 0x20
new values:
0x00000004:  00000020
-> mm 0x8 0x30
new values:
0x00000008:  00000030
-> mm 0xC 0x40
new values:
0x0000000c:  00000040
-> mm 0x10 0x50
```

```
new values:
0x00000010:  00000050
```

And try to read back the written values:

```
-> md 0x0 0x20
00000000:  00000010 00000020 00000030 00000040 .......  ...0...@
00000010:  00000050 00000050 00000050 00000050 ...P...P...P...P
```

We see the described behaviour of FPGA's default firmware. Four single registers at offset 0x0,0x4,0x8 and 0xC, and a mirrored register until the end of FPGA's area starting at offset 0x10.

6 Some hints on using phyCORE-MPC5200B

6.1 Decreasing boot time

6.1.1 Easier and faster kernel load

Whenever we load the kernel image from a JFFS2 filesystem, U-Boot must scan the whole flash partition to synchronise with the chains JFFS2 uses to handle the memory. This may consume many seconds during which nothing else happens.

After the Linux kernel is up and running it has to scan the JFFS2 filesystem again. This also takes some time during which the user of this system has to wait.

To avoid the first scan, we can create four partitions instead of three. The last two partitions remain unchanged, these are U-Boot and the sparse space. We create an additial partition located right before the second partition, with at least the size of one kernel image. For example 2048k. The first partition is still the root filesystem.

The new `mtdparts` variable looks like this (see figure 6.1):

```
mtdparts=phys_mapped_flash:13m(jffs2),2048k(kernel),
     256k(uboot)ro,-(sparse)
```

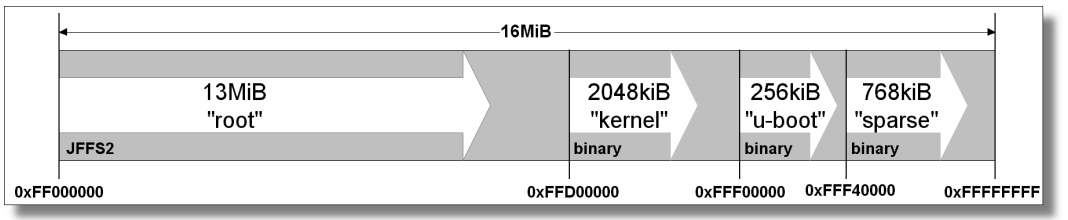

Figure 6.1: *The flash device divided into four partitions.*

Now we can write a kernel image into the second partition. We don't need a filesystem in this partition, because it's only a placeholder. Due to this, we can later boot this kernel image with a simple command

`PCM026> bootm 0xFFD00000`.

Note: 0xFFD00000 is the starting offset of the second partition.

It's much faster, because there is no previous scan required.

6.1.2 Disable Console output while kernel startup

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

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