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

How to become a PTXdist Guru

Based on the OSELAS.BSP( )
Pengutronix Generic-arm

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

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

info@pengutronix.de

Cut Here and Stick on your Monitor

Don't Panic

Contents

1 Welcome to the Embedded World

1.1 First Steps in the Embedded World

Once upon in time, programming embedded systems was easy: all a developer needed when he wanted to start a new product was a good toolchain, consisting of

- a compiler
- maybe an assembler
- probably an EPROM burning device

and things could start. After some more or less short time, every register of the CPU was known, a variety of library routines had been developed and our brave developer was able to do his project with the more and more well-known system. The controllers had legacy interfaces like RS232, i2c or SPI which connected them to the outside world and the main difference between the controllers available on the market was the number of GPIO pins, UARTS and memory ressources.

Things have changed. Hardware manufacturers have weakened the border between deeply embedded microcontrollers – headless devices with just a few pins and very limited computing power – and full blown microprocessors. System structures became much more complicated: where our good old controllers have had just some interrupts with some small interrupt service routines, we today need complicated generic interrupt infrastructures, suitable for generic software frameworks. Where we've had some linearly mapped flash ROM and some data RAM we today have multi-stage-pipeline architectures, memory management units, virtual address spaces, on-chip-memory, caches and other complicated units, which is not exactly what the embedded system developer wants program every other day.

Entering embedded operating systems. Although there are still some processors out there (like the popular ARM7TDMI based SoCs) which can be programmed the good old non-operating-system way with reasonable effort, it in fact is becoming more and more difficult. On the other hand, legacy I/O interfaces like RS232 are increasingly often replaced by modern plug-and-play aware communication channels: USB, FireWire (IEEE1394), Ethernet & friends are more and more directly being integrated into today's microcontroller hardware. Whereas some of these interfaces can "'somehow"' be handled the old controller-style way of writing software, the developer following this way will not be able to address the security and performance issues which come up with the modern network accessible devices.

During the last years, more and more of the small-scale companies which developed little embedded operating systems have been pushed out of the market. Nearly no small company is able to support all the different interfaces, communication stacks, development tools and security issues out there. New interfaces and -variants (like USB On-the-Go) are developed faster than operating system developers can supply the software for them. The result is a consolidation of the market: today we see that, besides niche products, probably only the largest commercial embedded operating system suppliers will survive that development.

Only the largest commercial...? There is one exception: when the same situation came up in the "mainstream" computer market at the beginning of the 1990ies, people started to develop an alternative to the large commercial operating systems: Linux. Linux did never start with a ready-to-use solution: people had a problem, searched for a solution but didn't find one. Then they started to develop one themselves, often several people did this

in parallel, and in a huge community based evolution mechanism the best solutions found their way into the Linux kernel, which over the time formed one of the most reliable and performant kernels available today. This "develop-and-evolute" mechanism has shown its effectiveness over and over again in the server and desktop market of today.

1.2 From Server to Embedded

The fact that for most technical problems that might occur it may be possible to find somebody on the internet who has already worked on the same or another very similar problem, was one of the major forces behind the success story of Embedded Linux.

Studies have shown that more than 70% of the embedded developers are not satisfied with a black-box operating system: they want to adapt it to their needs, to their special hardware situation (which most times is Just Different than anything available). Embedded projects are even more variegated than desktop- or server projects, due to the fact that there exist so many different embedded processors with lots of peripherals out there.

Linux has evolved from an i386 only operating system to a kernel running on nearly every modern 32 bit processor available today: x86, PowerPC, ARM, MIPS, m68k, cris, Super-H etc. The kernel supplies a hardware abstraction layer which lets our brave embedded developer once again concentrate on his very special problem, not on handling negligibilities like memory management.

But Linux is only half of the story. Besides the kernel, a Linux based embedded system consists of a "userland": a filesystem, containing all the small tools which form a small Unix system. Only the combination of the kernel and a Userland let's the developer run "normal" processes on his x86 development machine as well as on his embedded target.

1.3 Linux = Embedded Linux

Whereas the mainstream developers were always able to use normal Linux distributions like SuSE, RedHat, Mandrake or Debian as a base for their applications, things are different for embedded systems.

Due to the restricted ressources these systems normally have, distributions have to be small and should only contain those things that are needed for the application. Today's mainstream distributions cannot be installed in less than 100 MiB without major loss of functionality. Even Debian, probably today the most customizable mainstream distribution, cannot be shrunk below this mark without for example losing the packet management, which is an essential feature of using a distribution at all.

Additionally, source code for industrial systems has to be

- auditable and
- reproducable.

Embedded developers usually want to know what's in their systems – be it that they have to support their software for a long time span (something like 10-15 years are usual product lifetimes in automation applications) or that they have such a special scenario that they have to maintain their integrated source of their Userland themselves.

Entering PTXdist.

2 Getting a working Environment

2.1 Download Software Components

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

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

To build OSELAS.BSP-Pengutronix-Generic-2010.01.0, the following archives have to be available on the development host:

- ptxdist-2010.04.0.tgz
- ptxdist-2010.04.0-patches.tgz
- OSELAS.BSP-Pengutronix-Generic-2010.01.0.tar.gz
- OSELAS.Toolchain-1.99.3.6.tar.bz2

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

2.2 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.

2.2.1 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.

2.2.2 Extracting the Sources

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

ptxdist-2010.04.0.tgz The PTXdist software itself.

ptxdist-2010.04.0-patches.tgz All patches against upstream software packets (known as the ’patch repository’).

ptxdist-2010.04.0-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:

```
$ tar -zxf ptxdist-2010.04.0.tgz
$ tar -zxf ptxdist-2010.04.0-patches.tgz
```

and if required the generic projects:

```
$ tar -zxf ptxdist-2010.04.0-projects.tgz
```

If everything goes well, we now have a PTXdist-2010.04.0 directory, so we can change into it:

```
$ cd ptxdist-2010.04.0
$ ls -lF
total 532
-rw-r--r--   1 jb users  18361 Jan  6 14:50 COPYING
-rw-r--r--   1 jb users   3865 Jan  6 14:50 CREDITS
-rw-r--r--   1 jb users 115540 Jan  6 14:50 ChangeLog
-rw-r--r--   1 jb users     57 Jan  6 14:50 INSTALL
-rw-r--r--   1 jb users   2228 Jan  6 14:50 Makefile.in
-rw-r--r--   1 jb users   4196 Jan  6 14:50 README
-rw-r--r--   1 jb users    691 Jan  6 14:50 REVISION_POLICY
-rw-r--r--   1 jb users  63019 Jan  6 14:50 TODO
-rwxr-xr-x   1 jb users     28 Jan  6 14:50 autogen.sh
drwxr-xr-x   2 jb users   4096 Jan  6 14:50 bin
```

```
drwxr-xr-x   9 jb users   4096 Jan  6 14:50 config
-rwxr-xr-x   1 jb users 213205 Jan  6 16:29 configure
-rw-r--r--   1 jb users  12539 Jan  6 14:50 configure.ac
drwxr-xr-x  10 jb users   4096 Jan  6 14:50 generic
drwxr-xr-x 162 jb users   4096 Jan  6 14:50 patches
drwxr-xr-x   2 jb users   4096 Jan  6 14:50 platforms
drwxr-xr-x   4 jb users   4096 Jan  6 14:50 plugins
drwxr-xr-x   6 jb users  32768 Jan  6 14:50 rules
drwxr-xr-x   8 jb users   4096 Jan  6 14:50 scripts
drwxr-xr-x   2 jb users   4096 Jan  6 14:50 tests
```

2.2.3 Prerequisites

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

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

```
$ ./configure
```

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

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

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

ptxdist version 2010.04.0 configured.
Using '/usr/local' for installation prefix.

Report bugs to ptxdist@pengutronix.de
```

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

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

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

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

```
$ make
```

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

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

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

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

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

2.2.4 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-2010.04.0/projects:/office/my_projects/ptxdist
```

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

2.3 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.

2.3.1 Using Existing Toolchains

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

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

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

Open the OSELAS.BoardSupport() Package menu with:

```
$ ptxdist platformconfig
```

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

Preconditions an external toolchain must meet:

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

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

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

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

2.3.2 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.

2.3.3 Building the OSELAS.Toolchain for OSELAS.BSP-Pengutronix-Generic-2010.01.0

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

The required compiler to build the OSELAS.BSP-Pengutronix-Generic-2010.01.0 board support package is

`arm-v4t-linux-gnueabi_gcc-4.3.2_glibc-2.8_binutils-2.18_kernel-2.6.27-sanitized`

So the steps to build this toolchain are:

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

```
$ tar xf OSELAS.Toolchain-1.99.3.6.tar.bz2
$ cd OSELAS.Toolchain-1.99.3.6
$ ptxdist select ptxconfigs/↵
       arm-v4t-linux-gnueabi_gcc-4.3.2_glibc-2.8_binutils-2.18_kernel-2.6.27-sanitized.ptxconfig
$ ptxdist go
```

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

Measured times on different machines:

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

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.

2.3.4 Protecting the Toolchain

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

2.3.5 Building Additional Toolchains

The OSELAS.Toolchain-1.99.3.6 bundle comes with various predefined toolchains. Refer the `ptxconfigs/` folder for other definitions. To build additional toolchains we only have to clean our current toolchain project, removing the current `selected_ptxconfig` link and creating a new one.

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

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

`/opt/OSELAS.Toolchain-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`.

3 PTXdist User's Manual

This chapter should give any newbie the information he/she needs to be able to handle any embedded Linux projects based on PTXdist. Also the advanced user may find new valueable information.

3.1 How does it work?

PTXdist supports various aspects of the daily work to develop, deploy and maintain an embedded Linux based project.

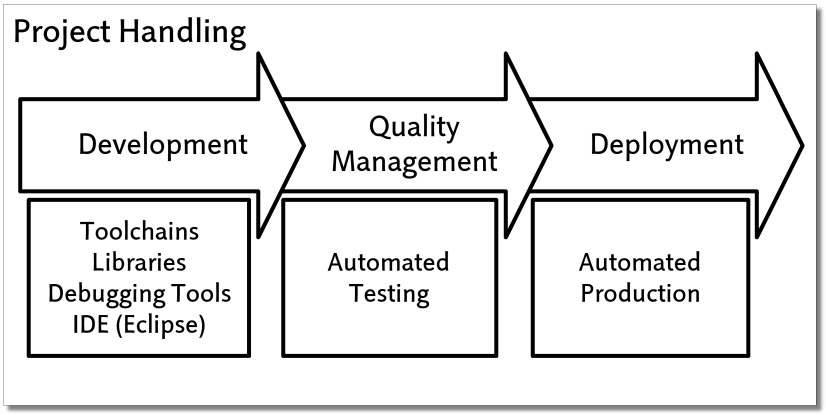

Figure 3.1: Objectives in a project

The most important part is the development. For this project phase, PTXdist provides features to ensure reproducibility and verifiability.

3.1.1 PTXdist's perception of the world

PTXdist works project centric. A PTXdist project contains all information and files to populate any kind of target system with all required software components.

- Specific configuration for
  - Bootloader
  - Kernel
  - Userland (root filesystem)
- Adapted files (or generic ones) for runtime configuration
- Patches for all kind of components (to fix bugs or improve features)

Some of these information or files are coming from the PTXdist base installation (patches for example), but also can be part of the project itself. By this way, PTXdist can be adapted to any kind of requirement.

Most users are fine with the information and files the PTXdist base installation provides. Development of PTXdist is done in a way to find default settings most user can work with. But advanced users can still adapt to their special needs.

As stated above, a PTXdist project consists of all required parts, some of these parts are separated by design: PTXdist separates a platform configuration from userland configuration (root filesystem). So, platforms can share a common userland configuration, but use a specific kernel configuration in their own platform configuration.

Collecting various platforms into one single project should help to maintain such projects. But some platforms do need special userland (think about graphic/non graphic platforms). To be able to also collect this requirement into one single project, so called *collections* are supported. With this feature, a user can configure a full featured main userland, reduced via a collection by some components for a specific platform where it makes no sense to build and ship them.

PTXdist can handle the following project variations:

- one hardware platform, one userland configuration (common case)
- one hardware platform, various userland configurations
- various hardware platforms, one userland configuration (common case)
- various hardware platforms, one userland configuration, various collections
- various hardware platforms, various userland configuration
- various hardware platforms, various userland configuration, various collections

3.1.2 PTXdist's build process

When PTXdist is building one part (we call it a *package*)of the whole project, it is divided into up to six stages:

get The package will be obtained from its source (downloaded from the web for example)

extract The package archive gets extracted and patched if a patch set for this package exists

prepare Many packages can be configured in various ways. If supported, this stage does the configuration in a way defined in the menu (project specific)

compile The package gets built.

install The package installs itself into a project local directory. This step is important at least for libraries (other packages may depend on)

targetinstall Relevant parts of the package will be used to build an IPKG archive and the root filesystem

For each single package, one so called *rule file* exists, describing the steps to be done in each stage shown above (refer section 5.3 for further details).

Due to the *get* stage, PTXdist needs a working internet connection to download an archive currently not existing on the development host. But there are ways to prevent PTXdist from doing so (refer to section 2.2.4).

3.2 First steps with PTXdist

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

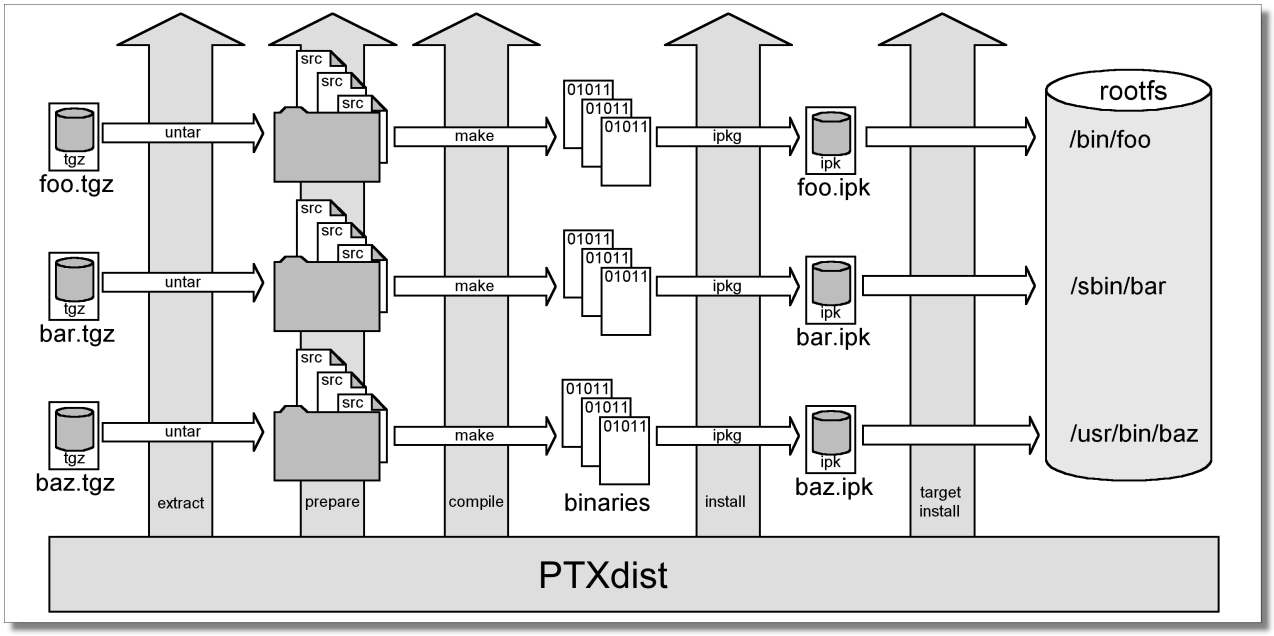

Figure 3.2: The build process

```
$ ptxdist <parameter>
```

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

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

```
$ ptxdist help
```

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

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

menu This starts a dialog based frontend for those who do not like typing commands. It will gain us access to the most common parameters to configure and build a PTXdist project.

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

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

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

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

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

menuconfig collection If multiple platforms are sharing one userland configuration, collections can define a subset of all selected packages for specific platforms. This is an advanced feature, rarely used.

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

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

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

collection Used to select the current collection configuration, which is only required in special cases. This parameter needs the path to a valid `collection`. It will generate a soft link called `selected_collection` in the project's main directory. This is an advanced feature, rarely used.

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

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

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

All these commands depending on various files a PTXdist based project provides. So, running the commands make only sense in directorys that contains a PTXdist based project. Otherwise PTXdist gets confused and confuses the user with funny error messages.

To show the usage of some listed major subcommands, we are using a generic PTXdist based project.

3.2.1 Extracting the Board Support Package

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

```
$ tar -zxf OSELAS.BSP-Pengutronix-Generic-2010.01.0.tar.gz
$ cd OSELAS.BSP-Pengutronix-Generic-2010.01.0
```

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

```
total 32
-rw-r--r-- 1 jb users  252 Jan  5 18:18 ChangeLog
-rw-r--r-- 1 jb users  741 Jan  5 18:18 README
drwxr-xr-x 5 jb users 4096 Jan  5 18:15 configs/
drwxr-xr-x 3 jb users 4096 Jan  5 18:15 documentation/
drwxr-xr-x 4 jb users 4096 Jan  5 18:15 patches/
drwxr-xr-x 5 jb users 4096 Jan  5 18:15 projectroot/
drwxr-xr-x 3 jb users 4096 Jan  5 18:15 protocols/
drwxr-xr-x 3 jb users 4096 Jan  5 18:15 rules/
```

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 respective 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.

Next we will build the OSELAS.BSP-Pengutronix-Generic-2010.01.0 to show some of PTXdist's main features.

3.2.2 Selecting a Userland Configuration

First of all we have to select a userland configuration. This step defines what kind of applications will be built for the hardware platform. The OSELAS.BSP-Pengutronix-Generic-2010.01.0 comes with a predefined configuration we select in the following step:

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

3.2.3 Selecting a Hardware Platform

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

```
$ ptxdist platform configs/arm-qemu-3/platformconfig
info: selected platformconfig:
       'configs/arm-qemu-3/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-v4t-linux-gnueabi/↵
    gcc-4.3.2-glibc-2.8-binutils-2.18-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.

3.2.4 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.

```
$ ptxdist toolchain /opt/OSELAS.Toolchain-1.99.3/arm-v4t-linux-gnueabi/↵
      gcc-4.3.2-glibc-2.8-binutils-2.18-kernel-2.6.27-sanitized/bin
```

3.2.5 Building the Root Filesystem Content

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

```
$ ptxdist go
```

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

3.2.6 What we Got Now

After building the project, we find even more sub directories in our project.

platform-versatilepb/build-cross Contains all packages sources compiled to run on the host and handle target architecture dependend things.

platform-versatilepb/build-host Contains all packages sources compiled to run on the host and handle architecture independend things.

platform-versatilepb/build-target Contains all package sources compiled for the target architecure.

platform-versatilepb/images Generated files for the target can be found here: Kernel image and root filesystem image.

platform-versatilepb/packages Location for alle individual packages in ipk format.

platform-versatilepb/sysroot-target Contains everything target architecture dependend (libraries, header files and so on).

platform-versatilepb/sysroot-cross Contains everything that is host specific but must handle target architecture data.

platform-versatilepb/sysroot-host Contains everything that is only host specific.

platform-versatilepb/root Target's root filesystem image. This directory can be mounted as an NFS root for example.
Note: Due to only root can create device nodes and the build system must be run as a non root user, there is no device node present in this image. At least `/dev/console`, `/dev/null` and `/dev/zero` should exist. Create them as user root manually in order to use this directory as an NFS root.

platform-versatilepb/root-debug Target's root filesystem image. The difference to `root/` is, all programs and libraries in this directory still have their debug information present. This directory is intended to be used as system root for a debugger. To be used by the debugger, you should setup your debugger with
`set solib-absolute-prefix </path/to/workspace>/root-debug`

platform-versatilepb/state Building every package is divided onto stages. And stages of one package can depend on stages of other packages. In order to handle this correctly, this directory contains timestamp files about finished stages.

This are the generated files:

platform-versatilepb/logfile Every run of PTXdist will add its output to this file. If something fails, this file can help to find the cause.

3.2.7 Creating a Root Filesystem Image

After we have built the root filesystem content, we can make an image, which can be flashed to the target system or copied on some kind of disk media. To do so, we just run

```
$ ptxdist images
```

PTXdist now extracts the content of priorly created **.ipk* packages to a temporary directory and generates an image out of it. PTXdist supports following image types:

- **hd.img:** contains bootloader, kernel and root files in an 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 an ext2 filesystem.
- **root.squashfs:** root files inside a squashfs filesystem.
- **root.tgz:** root files inside a plain gzip compressed tar ball.

All these files can be found in `platform-versatilepb/images`.

3.2.8 Running all Parts in an emulated Environment (QEMU)

The OSELAS.BSP-Pengutronix-Generic-2010.01.0 is prepared to give every user a chance to run the results of the previous steps even in the absense of real hardware. All we need is a working QEMU on our development host.

Simply run

```
$ ./configs/arm-qemu-3/run
```

This will start QEMU in full system emulation mode and runs the previously built kernel which then uses the generated disk image to bring up a full Linux based system.

The running system uses a serial device for its communication. QEMU forwards this emulated device to the current development host console. So, we can watch the starting kernel's output and log in on this system.

Note: Log in as user '`root`' with no password (just enter).

Leaving this emulated environment happens by entering the key sequence *CTRL-A X*.

3.3 Adapting the OSELAS.BSP-Pengutronix-Generic-2010.01.0 Project

Handling a fully prepared PTXdist project is easy. But everything is fixed to the settings the developer selected. We now want to adapt the OSELAS.BSP-Pengutronix-Generic-2010.01.0 project in a few simple settings.

3.3.1 Working with Kconfig

Whenever we modify our project, PTXdist is using *Kconfig* to manipulate the settings. *Kconfig* means *kernel configurator* and was mainly developed to configure the Linux kernel itself. But it is easy to adapt, to use and so popular that more and more projects are using *Kconfig* for their purposes. PTXdist is one of them.

What is Kconfig

It is a user interface to select given resources in a convenient way. The resources that we can select are given in simple text files. It uses a powerful "language" in these text files to organize them in a hierarchical manner, solves challenges like resource dependencies, supports help and search features. PTXdist uses all of these features. *Kconfig* supports a text based user interface by using the *ncurses* library to manipulate the screen content and should work on nearly all host systems.

For example running PTXdist's `menuconfig` subcommand in this way

```
$ ptxdist menuconfig
```

will show the following console output

Figure 3.3: Main userland configuration menu

Navigate in Kconfig menu (select, search, ...)

To navigate through the configuration tree, we are using the arrow keys. Up and down navigates vertically in the menu entries. Right and left navigates between *Select*, *Exit* and *Help* (in the bottom part of our visual screen).

To enter one of the menus, we navigate to this entry to highlight it and press the *Enter* key. To leave it, we select *Exit* and press the *Enter* key again. There are shortcuts available, instead of pressing the *Enter* key to enter a menu we also can press *alt-s* and to leave a menu *alt-e*. Also an ESC double hit leaves any menu we are in.

To select a menu entry, we use the *Space* key. This will toggle the selection. Or, to be more precise and faster, we use the key *y* to select an entry, and key *n* to deselect it.

To get help for a specific menu topic, we navigate vertically to highlight it and horizontally to select the *Help* entry. Then we can press *Enter* to see the help.

To search for specific keywords, we press the / key and enter a word. Kconfig then lists all occurences of this word in all menus.

Meaning of visual feedbacks in Kconfig

- Submenus to enter are marked with a trailing `>`
  Note: Some submenus are also marked with a leading bracket `[ ]`. To enter them we first must select/enable them `[*]`
- Entries with a list of selectable alternatives are also marked with a trailing `>`
- Entries we can select are marked with a leading empty bracket `[ ]`
- Entries that are already selected are marked with a leading filled bracket `[*]`
- Entries that are selected due to dependencies into other selected entries are marked with a leading
- Some entries need a free text to enter, they are marked with leading brackets `()` and the free text in it

Menus and submenus in Kconfig (sectioning)

There are dozens of entries in the PTXdist configuring menus. To handle them, they are divided and separated into logical units.

The main building blocks in the *userland configuration* menu are:

- Host Options: Some parts of the project are build host relevant only. For example PTXdist can build the DDD debugger to debug applications running on the target.
- Root Filesystem: Settings to arrange target's root filesystem and to select the main C runtime library
- Applications: Everything we like to run on your target.

The main building blocks in the *platform configuration* menu are:

- Architecture: Basic settings, like the main and sub architecture the target system uses, the toolchain to be used to build everything and some other architecture dependent settings.
- Linux kernel: Which kernel revision and kernel configuration should be used
- Bootloader: Which bootloader (if any) should be built in the project
- The kind of image to populate a root filesystem into the target system

The main building blocks in the *board setup configuration* menu are:

- Network: Network settings for the target
- Host: Host setup to be able to reach the target system

At this point it could be useful to walk to the whole menus and their submenus to get an idea about the amount of features and applications PTXdist currently supports.

3.3.2 Adapting Platform Settings

Some parts of the OSELAS.BSP-Pengutronix-Generic-2010.01.0 project are platform specific (in contrast to the userland configuration that could be shared between platforms). We now want to change the used Linux kernel of our current arm-qemu-3 platform. It comes with a default linux-2.6.31 and we want to change it to a more recent linux-2.6.32.

To do so, we run:

```
$ ptxdist menuconfig platform
```

In this Kconfig dialogue we navigate to the entry:

```
[*] Linux kernel  --->
        (2.6.31) kernel version
```

and replace the 2.6.31 value by the 2.6.32 value. Now we leave the menu and save the new settings.

A Linux kernel needs a configuration for being built correctly. The OSELAS.BSP-Pengutronix-Generic-2010.01.0 project comes with a prepared configuration in the file `configs/arm-qemu-3/kernelconfig` for the 2.6.31 kernel.

It is always a good idea to start with a known-to-work kernel configuration. So, in this case, we continue using the `configs/arm-qemu-3/kernelconfig` file even for our new 2.6.32 kernel.

But due to the update to a more recent kernel revision, we should check in the next step if the new kernel supports more or different features for our platform.

3.3.3 Adapting Linux Kernel Settings

In this section we want to show how to change some Linux kernel settings of our OSELAS.BSP-Pengutronix-Generic-2010.01.0project.

First of all, we run

```
$ ptxdist menuconfig kernel
```

This command will start the kernel's Kconfig. For this example we want to enable USB host support in the kernel. To do so, we navigate to:

```
Device Drivers  --->
        [ ] USB support  --->
                < > Support for Host-side USB
                        < > OHCI HCD support
```

Note: All the listed empty [ ] and < > above must be activated to get all submenu entries.

We leave the menu and save the new kernel configuration.

To start building a new kernel with the new configuration, we again run:

```
$ ptxdist go
```

This builds or re-builds the kernel, because we changed its settings.

Note: If nothing was changed, `ptxdist go` also will do nothing.

When PTXdist has finished its job, the new bootable kernel can be found at `platform-versatilepb/images/linuximage`. To boot it again in the QEMU emulation, the hard disk image must be re-created with:

```
$ ptxdist images
$ ./configs/arm-qemu-3/run
```

The emulated system should now start with a 2.6.32 based kernel with USB support.

3.3.4 Adapting Userland Settings

After changing some platform and kernel settings, we are now reaching the most interesting area: Userland.

In the userland area we can enable and use all the applications and services PTXdist provides. Many of them are working out of the box when enabled and executed on the target side. Some need additional runtime configuration, but PTXdist comes with most common configurations for such packages.

In this simple example, we want to add the missing head command to our target's shell. First we run:

```
$ ptxdist menuconfig
```

The additional command we want to enable is provided by the *Busybox* package. So we navigate to:

```
Shell & Console Tools --->
       -*- busybox  --->
              Coreutils  --->
                     [ ] head
```

After activating the [ ] head entry, we leave the menu and save the new configuration.

Once again, a

```
$ ptxdist go
```

will build or re-build the busybox package due to its configuration change.

And also once again, after finishing its job, the following commands let us test the new command:

```
$ ptxdist images
$ ./configs/arm-qemu-3/run
```

Log in on the emulated system and simply check with a:

```
ptx login: root
login[xxx]: root login on 'ttyS0'
root@ptx:~ head /etc/fstab
#
# /etc/fstab
#

# special filesystems
proc    /proc                   proc    defaults                        0 0
debugfs /sys/kernel/debug       debugfs defaults,noauto                 0 0
devpts  /dev/pts                devpts  defaults                        0 0
none    /tmp                    tmpfs   defaults,mode=1777,uid=0,gid=0  0 0
none    /sys                    sysfs   defaults                        0 0
```

We are done now. These simple examples should give the users a quick feeling how things are working in PTXdist and how to modify them. Adapting this generic BSP to a different platform with nearly the same features as our reference platforms is possible with this knowledge.

But most of the time, a user needs more detailed adaptions to be able to fit all requirements of the new platform. At this point of time we are no longer ordinary users of PTXdist, we become developers now.

So, right now its time to read the *PTXdist Developer's Manual*

4 PTXdist Developer's Manual

This chapter will show all (or most) of the details of how PTXdist works.

- where are the files stored that PTXdist uses when building packages
- how patching works
- where is PTXdist fetching a package's runtime configuration files from
- how to control a package's build stages
- how to add new packages

4.1 PTXdist's directory hierarchy

Note: Referenced directories are meant relative to the PTXdist main installation location (if not otherwise stated). If not configured differently, this main path is `/usr/local/lib/ptxdist-2010.04.0`

4.1.1 Rule Files

When building a single package, PTXdist needs the information on how to handle the package, i.e. on how to get it from the source up to what the target needs at runtime. This information is provided by a rule file per package.

PTXdist collects all rule files in its `rules/` directory. Whenever PTXdist builds something, all these rule files are scanned at once. These rule files are global rule files, valid for all projects. PTXdist uses a mechanism to be able to add or replace specific rule files on a per project base. If a `rules/` directory exists in the current project, its content is scanned too. These project local rule files are used in addition to the global rule files or – if they are using the same name as a global rule file – **replacing** the global rule file.

The replacing mechanism can be used to extend or adapt packages for specific project requirements. Or it can be used for bug fixing by backporting rule files from more recent PTXdist revisions to projects that are stuck to an older PTXdist revision for maintenance only.

4.1.2 Patch Series

There are many packages in the wild that are not cross build aware. They fail compiling some files, use wrong include paths or try to link against host libraries. To be sucessful in the embedded world, these types of failures must be fixed. If required, PTXdist provides such fixes per package. They are organized in *patch series* and can be found in the `patches/` directory within a subdirectory using the same name as the package itself.

PTXdist uses the utility `patch` or `quilt` to apply an existing patch series after extracting the archive. So, every patch series contains a set of patches and one `series` file to define the order in which the patches must be applied.

Note: Patches can be compressed.

Besides the patches/ directory at the main installation location, PTXdist searches two additional locations for a patch series for the package in question.

One location is the project's currently used platform directory. If the currently used platform is located in configs/arm-qemu-3, PTXdist searches in ./configs/arm-qemu-3/patches/<package name>.

If no patch series was found in the platform directory, the next location PTXdist it searches for a patch series is the main project directory in ./patches/<package name>.

If both project local locations do not provide a patch series for the specific package, PTXdist falls back to the patches/ directory at its main installation location.

This search order can be used to use specific patch series for specific cases.

- platfom specific
- project specific
- common case
- bug fixing

The *bug fixing* case is used in accordance to a replacement of a rule file. If this was done due to a backport, and the more recent PTXdist revision does not only exchange the rule file but also the patch series, this mechanism ensures that both relevant parts can be updated in the project.

4.1.3 Runtime Configuration

Many packages are using runtime configuration files along with their executables and libraries. PTXdist provides default configuration files for the most common cases. These files can be found in the generic/etc directory and they are using the same names as the ones at runtime (and their install directory on the target side will also be /etc).

But some of these default configuration files are empty, due to the absence of a common case. The project must provide replacements of these files with a more useful content in every case where the (empty) default one does not meet the target's requirements.

PTXdist first searches the project local ./projectroot/etc directory for a specific configuration file and falls back to use the default one if none exists locally.

A popular example is the configuration file /etc/fstab. The default one coming with PTXdist works for the most common cases. But if our project requires a special setup, we can just copy the default one to the local ./projectroot/etc/fstab, modify it and we are done. The next time PTXdist builds the root filesystem it will use the local fstab instead of the global (default) one.

4.2 Adding new Packages

PTXdist provides a huge amount of applications sufficient for the most embedded use cases. But there is still need for some fancy new packages. This section describes the steps and the background on how to intergrate new packages into the project.

At first a summary about possible application types which PTXdist can handle:

- **host type**: This kind of package is built to run on the build host. Most of the time such a package is needed if another target relevant package needs to generate some data. For example the *glib* package depends on its own to create some data. But if it is compiled for the target, it can't do so. That's why a host glib package is required to provide these utilities runnable on the build host. It sounds strange to build a host package, even if on the build host such utilities are already installed. But this way ensures that there are no dependencies regarding the build host system.
- **target type**: This kind of package is built for the target.
- **cross type**: This kind of package is built for the build host, but creates architecture specific data for the target.
- **klibc type**: This kind of package is built against klibc to be part of an initramfs for the target.
- **src-autoconf-prog**: This kind of package is built for the target. It is intended for development, as it does not handle a released archive but a plain source project instead. Creating such a package will also on demand create a small autotools based source template project to give the developer an easy point to start. This template is prepared to build a single executable program.
- **src-autoconf-lib**: This kind of package is built for the target. It is intended for development, as it does not handle a released archive but a plain source project instead. Creating such a package will also on demand create a small autotools/libtool based source template project to give the developer an easy point to start. This template is prepared to build a single shared library.
- **src-autoconf-proglib**: This kind of package is built for the target. It is intended for development, as it does not handle a released archive but a plain source project instead. Creating such a package will also on demand create a small autotools/libtool based template project to give the developer an easy point to start. This template is prepared to build a single shared library and a single executable program. The program will be linked against the shared library.
- **simple**: This kind of package is intended to add a few simple files into the build process. We assume these files do not need any processing, they are ready to use and only must present in the build process or at runtime (HTML files for example). Refer the section 6.8 for further details on how to use it.
- **src-make-prog**: This kind of package is built for the target. Its intended for development, as it does not handle a released archive but a plain source project instead. Creating such a package will also create a simple makefile based template project the developen can use as a starting point for development.
- **src-cmake-prog**: This kind of package is built for the target. Its intended for developments based on the *cmake* buildsystem. If the developen is going to develop a QT based application, this rule is prepared to compile sources in accordance to the target libraries and their settings. Creating such a package will also create a simple template project to be used as a starting point for development.
- **font**: This package is a helper to add X font files to the root filesystem. This package does not create an additional IPKG, instead it adds the font to the existing font IPGK. This includes the generation of the directory index files, required by the Xorg framework to recognize the font file.
- **src-linux-driver**: This kind of package builds an out of tree kernel driver. It also creates a driver template to give the developer an easy point to start.
- **src-stellaris**: This package is intended for development of firmware for standalone Cortex-M3 Stellaris microcontrollers. These kind of microcontrollers are running without any operating systems most of the time. It also creates a small firmware template to give the developer an easy point to start.

4.2.1 Rule File Creation

To create such a new package, we create a project local `rules/` directory first. Then we run

```
$ ptxdist newpackage <package type>
```

If we omit the <package type>, PTXdist will list all available package types.

In our first example, we want to add a new target type archive package. When running the

```
$ ptxdist newpackage target
```

command, PTXdist asks a few questions about this package. This information is the basic data PTXdist must know about the package.

```
ptxdist: creating a new 'target' package:

ptxdist: enter package name......: foo
ptxdist: enter version number....: 1.1.0
ptxdist: enter URL of basedir....: http://www.foo.com/download/src
ptxdist: enter suffix............: tar.gz
ptxdist: enter package author....: My Name <me@my-org.com>
```

What we have to answer:

- **package name**: As this kind of package handles a source archive, the correct answer here is the basename of the archive's file name. If its full name is `foo-1.1.0.tar.gz`, then `foo` is the basename to enter here.
- **version number**: Most source archives are using a release or version number in their file name. If its full name is `foo-1.1.0.tar.gz`, then `1.1.0` is the version number to enter here.
- **URL of basedir**: This URL tells PTXdist where to download the source archive from the web (if not already done). If the full URL to download the archive is `http://www.foo.com/download/src/foo-1.1.0.tar.gz`, the basedir part `http://www.foo.com/download/src` is to enter here.
- **suffix**: Archives are using various formats for distribution. PTXdist uses the *suffix* entry to select the matching extraction tool. If the archive's full name is `foo-1.1.0.tar.gz`, then `tar.gz` is the suffix to enter here.
- **package author**: If we intend to contribute this new package to PTXdist mainline, we should add our name here. This name will be used in the copyright note of the rule file and will also be added to the generated ipkg.

4.2.2 Make it Work

Generating the rule file is only one of the required steps to get a new package. The next steps to make it work are to check if all stages are working as expected and to select the required parts to get them installed in the target root filesystem. Also we must find a reasonable location where to add our new menu entry to configure the package.

The generated skeleton starts to add the new menu entry in the main configure menu. Running `ptxdist menuconfig` will show it on top of all other menus entries. To be able to implement all the other required steps for adding a new package, we first must enable the package for building (fine tuning the menu can be the last step).

Lets start checking the *get* and *extract* stage, calling them manually one after another.

```
$ ptxdist get foo

---------------------------
target: foo-1.1.0.tar.gz
---------------------------
```

```
--2009-12-21 10:54:45--  http://www.foo.com/download/src/foo-1.1.0.tar.gz
Length: 291190 (284K) [application/x-gzip]
Saving to: '/global_src/foo-1.1.0.tar.gz.XXXXOGncZA'

100%[======================================>] 291,190      170K/s   in 1.7s

2009-12-21 10:54:48 (170 KB/s) - '/global_src/foo-1.1.0.tar.gz' saved [291190/291190]
```

This command should start to download the source archive. If it fails, we should check our network connection and if the URL in use is correct.

```
$ ptxdist extract foo

-----------------------
target: foo.extract
-----------------------

extract: archive=/global_src/foo-1.1.0.tar.gz
extract: dest=/home/jbe/my_new_prj/build-target
PATCHIN: packet=foo-1.1.0
PATCHIN: dir=/home/jbe/my_new_prj/build-target/foo-1.1.0
PATCHIN: no patches for foo-1.1.0 available
Fixing up /home/jbe/my_new_prj/build-target/foo-1.1.0/configure
finished target foo.extract
```

In this example we expect an autotoolized source package. E.g. to prepare the build, the archive comes with a `configure` script. This is the default case for PTXdist. So, there is no need to modify the rule file and we can simply run:

```
$ ptxdist prepare foo

-----------------------
target: foo.prepare
-----------------------

[...]

checking build system type... i686-host-linux-gnu
checking host system type... i586-unknown-linux-gnu
checking whether to enable maintainer-specific portions of Makefiles... no
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for i586-unknown-linux-gnu-strip... i586-unknown-linux-gnu-strip
checking for i586-unknown-linux-gnu-gcc... i586-unknown-linux-gnu-gcc
checking for C compiler default output file name... a.out

[...]

configure: creating ./config.status
config.status: creating Makefile
config.status: creating ppa_protocol/Makefile
config.status: creating config.h
```

```
config.status: executing depfiles commands
finished target foo.prepare
```

At this stage things can fail:

- The `configure` script is not cross compile aware
- The package depends on external components (libraries for example)

If the `configure` script is not cross compile aware, we are out of luck. We must patch the source archive in this case to make it work. Refer to section 4.2.6 on how to use PTXdist's features to simplify this task.
If the package depends on external components, these components might be already part of PTXdist. In this case we just have to add this dependency into the menu file and we are done. But if PTXdist cannot fulfill this dependency, we also must add it as a separate package first.

If the *prepare* stage has finished successfully, the next step is to compile the package.

```
$ ptxdist compile foo

-----------------------
target: foo.compile
-----------------------

make[1]: Entering directory '/home/jbe/my_new_prj/build-target/foo-1.1.0'
make  all-recursive
make[2]: Entering directory '/home/jbe/my_new_prj/build-target/foo-1.1.0'
make[3]: Entering directory '/home/jbe/my_new_prj/build-target/foo-1.1.0'

[...]

make[3]: Leaving directory '/home/jbe/my_new_prj/build-target/foo-1.1.0'
make[2]: Leaving directory '/home/jbe/my_new_prj/build-target/foo-1.1.0'
make[1]: Leaving directory '/home/jbe/my_new_prj/build-target/foo-1.1.0'
finished target foo.compile
```

At this stage things can fail:

- The build system is not cross compile aware (it tries to execute just created target binaries for example)
- The package depends on external components (libraries for example) not detected by `configure`
- Sources are ignoring the endianess of some architectures or using header files from the build host system (from `/usr/include` for example)
- The linker uses libraries from the build host system (from `/usr/lib` for example) by accident

In all of these cases we must patch the sources to make them work. Refer to section 4.2.4 on how to use PTXdist's features to simplify this task.

In this example we expect the best case: Everything went fine, even for cross compiling. So, we can continue with the next stage: *install*

```
$ ptxdist install foo

-----------------------
target: foo.install
-----------------------
```

```
make[1]: Entering directory ‘/home/jbe/my_new_prj/build-target/foo-1.1.0’
make[2]: Entering directory ‘/home/jbe/my_new_prj/build-target/foo-1.1.0’
make[3]: Entering directory ‘/home/jbe/my_new_prj/build-target/foo-1.1.0’
test -z ”/usr/bin”  /bin/mkdir -p ”/home/jbe/my_new_prj/build-target/foo-1.1.0/usr/bin”
  /usr/bin/install -c ’foo’ ’/home/jbe/my_new_prj/build-target/foo-1.1.0/usr/bin/foo’
make[3]: Leaving directory ‘/home/jbe/my_new_prj/build-target/foo-1.1.0’
make[2]: Leaving directory ‘/home/jbe/my_new_prj/build-target/foo-1.1.0’
make[1]: Leaving directory ‘/home/jbe/my_new_prj/build-target/foo-1.1.0’
finished target foo.install

----------------------------
target: foo.install.post
----------------------------

finished target foo.install.post
```

This *install* stage does not install anything to the target root filesystem. It is mostly intended to install libraries that other programs should link against later on.

The last stage – *targetinstall* – is the one that defines the package's components to be forwarded to the target's root filesystem. Due to the absence of a generic way, this is the task of the developer. So, at this point of time we must run our favourite editor and modify our new rule file `./rules/foo.make`.

The skeleton for the *targetinstall* stage looks like this:

```
# ----------------------------------------------------------------------------
# Target-Install
# ----------------------------------------------------------------------------

$(STATEDIR)/foo.targetinstall:
        @$(call targetinfo)

        @$(call install_init,  foo)
        @$(call install_fixup, foo,PACKAGE,foo)
        @$(call install_fixup, foo,PRIORITY,optional)
        @$(call install_fixup, foo,VERSION,$(FOO_VERSION))
        @$(call install_fixup, foo,SECTION,base)
        @$(call install_fixup, foo,AUTHOR,”My Name <me@my-org.com>”)
        @$(call install_fixup, foo,DEPENDS,)
        @$(call install_fixup, foo,DESCRIPTION,missing)

        @$call install_copy, foo, 0, 0, 0755, $(FOO_DIR)/foobar, /dev/null)

        @$(call install_finish, foo)
        @$(call touch)
```

The ”header” of this stage defines some information IPKG needs. The important part that we must modify is the call to the `install_copy` macro (refer to section 5.2 for more details about this kind of macros). This call instructs PTXdist to include the given file (with PID, UID and permissions) into the IPKG, which means to install this file to the target's root filesystem.

From the previous *install* stage we know this package installs an executable called `foo` to location `/usr/bin`. We can do the same for our target by changing the *install_copy* line to:

```
        @$(call install_copy, foo, 0, 0, 0755, $(FOO_DIR)/foo, /usr/bin/foo)
```

To check it, we just run:

```
$ ptxdist targetinstall foo

-----------------------------
target: foo.targetinstall
-----------------------------

install_init:   preparing for image creation...
install_init:   @ARCH@ -> i386 ... done
install_init:   preinst not available
install_init:   postinst not available
install_init:   prerm not available
install_init:   postrm not available
install_fixup:  @PACKAGE@ -> foo ... done.
install_fixup:  @PRIORITY@ -> optional ... done.
install_fixup:  @VERSION@ -> 1.1.0 ... done.
install_fixup:  @SECTION@ -> base ... done.
install_fixup:  @AUTHOR@ -> "My Name <me\@my-org.com>" ... done.
install_fixup:  @DESCRIPTION@ -> missing ... done.
install_copy:
  src=/home/jbe/my_new_prj/build-target/foo-1.1.0/foo
  dst=/usr/bin/foo
  owner=0
  group=0
  permissions=0755
xpkg_finish:    collecting license (unknown) ... done.
xpkg_finish:    creating ipkg package ... done.
finished target foo.targetinstall

----------------------------------
target: foo.targetinstall.post
----------------------------------

finished target foo.targetinstall.post
```

After this command, the target's root filesystem contains a file called `/usr/bin/foo` owned by root, its group is also root and everyone has execution permissions, but only the user root has write permissions.

One last task of this port is still open: A reasonable location for the new menu entry in PTXdist's menu hierarchy. PTXdist arranges its menus on the meaning of each package. Is it a network related tool? Or a scripting language? Or a graphical application?
Each of these global meanings have their own submenu, where we can add our new entry to. We just have to edit the head of our new menu file to add it to a specific global menu. If our new package is a network related tool, the head of the menu file should look like:

```
## SECTION=networking
```

We can grep through the other menu files from the PTXdist main installation `rules/` directory to get an idea what section names are available:

```
rules/ $ find . -name \*.in | xargs grep "## SECTION"
./acpid.in:## SECTION=shell_and_console
./alsa-lib.in:## SECTION=system_libraries
./alsa-utils.in:## SECTION=multimedia_sound
./apache2.in:## SECTION=networking
./apache2_mod_python.in:## SECTION=networking
```

```
[...]
./klibc-module-init-tools.in:## SECTION=initramfs
./xkeyboard-config.in:## SECTION=multimedia_xorg_data
./xorg-app-xev.in:## SECTION=multimedia_xorg_app
./xorg-app-xrandr.in:## SECTION=multimedia_xorg_app
./host-eggdbus.in:## SECTION=hosttools_noprompt
./libssh2.in:## SECTION=networking
```

Porting a new package to PTXdist is finished now.

To check it right away, we simply run these two commands:

```
$ ptxdist clean foo
rm -rf /home/jbe/my_new_prj/state/foo.*
rm -rf /home/jbe/my_new_prj/packages/foo_*
rm -rf /home/jbe/my_new_prj/build-target/foo-1.1.0
$ ptxdist targetinstall foo

[...]
```

4.2.3 Advanced Rule Files

The previous example on how to create a rule file sometimes works as shown above. But most of the time source archives are not that simple. In this section we want to give the user a more detailed selection how the package will be built.

Adding Static Configure Parameters

The `configure` scripts of various source archives provide additional parameters to enable or disable features, or to configure them in a specific way.

We assume the `configure` script of our `foo` example (refer to section 4.2.1) supports two additional parameters:

- **--enable-debug**: Make the program more noisy. It's disabled by default.
- **--with-bar**: Also build the special executable **bar**. Building this executable is also disabled by default.

We now want to forward these options to the `configure` script when it runs in the *prepare* stage. To do so, we must again open the rule file with our favourite editor and navigate to the *prepare* stage entry.

PTXdist uses the variable `FOO_AUTOCONF` as the list of parameters to be given to `configure`.

Currently this variable is defined to:

```
FOO_AUTOCONF := $(CROSS_AUTOCONF_USR)
```

The variable `CROSS_AUTOCONF_USR` is predefined by PTXdist and contains all basic parameters to instruct `configure` to prepare for a cross compile environment.

To use the two additional mentioned `configure` parameters, we supplement this expression as follows:

```
FOO_AUTOCONF := $(CROSS_AUTOCONF_USR) \
        --enable-debug \
        --with-bar
```

Note: We recommend to use this format with each parameter on a line of its own. That is easier to read and a diff shows more exactly any change.

To do a fast check if this addition was successful, we run:

```
$ ptxdist print FOO_AUTOCONF
--prefix=/usr --sysconfdir=/etc --host=i586-unknown-linux-gnu --build=i686-host-linux-gnu --enable-debug ↵
     --with-bar
```

Or re-build the package with the new settings:

```
$ ptxdist drop foo prepare
$ ptxdist targetinstall foo
```

Adding Dynamic Configure Parameters

Sometimes it makes sense to add this kind of parameters on demand only; especially a parameter like `--enable-debug`. To let the user decide if this parameter is to be used or not, we must add a menu entry. So, let's expand our menu. Here is its current content:

```
## SECTION=fixme

config FOO
        tristate
        prompt "foo"
        help
          FIXME
```

We'll add two menu entries, one for each optional parameter we want to add on demand to the `configure` parameters:

```
## SECTION=fixme

config FOO
       tristate
       prompt "foo"
       help
         FIXME

if FOO
config FOO_DEBUG
       bool
       prompt "add debug noise"

config FOO_BAR
       bool
       prompt "build bar"

endif
```

Note: To extend the base name by a suboption name as a trailing component gives PTXdist the ability to detect a change in the package's settings to force its rebuild.

To make usage of the new menu entries, we must check them in the rule file and add the correct parameters:

```
#
# autoconf
#
FOO_AUTOCONF := $(CROSS_AUTOCONF_USR)

ifdef PTXCONF_FOO_DEBUG
FOO_AUTOCONF += --enable-debug
else
FOO_AUTOCONF += --disable-debug
endif

ifdef PTXCONF_FOO_BAR
FOO_AUTOCONF += --with-bar
else
FOO_AUTOCONF += --without-bar
endif
```

It is a good practice to add both settings, e.g. `--disable-debug` even if this is the default case. Sometimes `configure` tries to guess something and the binary result might differ depending on the build order. For example some kind of package would also build some X related tools, if X libraries are found. In this case it depends on the build order, if the X related tools are built or not. All the autocheck features are problematic here. So, if we do not want `configure` to guess its settings we **must disable everything we do not want**

If some parts of a package are built on demand only, they must also be installed on demand only. Besides the *prepare* stage, we also must modify our *targetinstall* stage:

```
[...]

        @$(call install_copy, foo, 0, 0, 0755, $(FOO_DIR)/foo, /usr/bin/foo)

ifdef PTXCONF_FOO_BAR
        @$(call install_copy, foo, 0, 0, 0755, $(FOO_DIR)/bar, /usr/bin/bar)
endif

        @$(call install_finish, foo)
        @$(call touch)

[...]
```

Now we can play with our new menu entries and check if they are working as expected:

```
$ ptxdist menuconfig
$ ptxdist targetinstall foo
```

Whenever we change a *FOO* related menu entry, PTXdist should detect it and re-build the package when a new build is started.

Managing External Compile Time Dependencies

While running the prepare stage, it could happen that it fails due to a missing external dependency.

For example:

```
[...]
checking whether zlib exists....failed
```

In this example, our new package depends on the compression library *zlib*. PTXdist comes with a target *zlib*. All we need to do in this case is to declare that our new package *foo* depends on *zlib*. This kind of dependencies is managed in the menu file of our new package by simply adding the `select ZLIB` line. After this addition our menu file looks like:

```
## SECTION=fixme

config FOO
       tristate
       select ZLIB
       prompt "foo"
       help
         FIXME

if FOO
config FOO_DEBUG
       bool
       prompt "add debug noise"

config FOO_BAR
       bool
       prompt "build bar"

endif
```

PTXdist now builds the *zlib* first and our new package thereafter.

Managing External Compile Time Dependencies on Demand

It is good practice to add only those dependecies that are really required for the current configuration of the package. If the package provides the features *foo* and *bar* and its `configure` provides switches to enable/disable them independently, we can also add dependencies on demand. Let's assume feature *foo* needs the compression library *libz* and *bar* needs the XML2 library *libxml2*. These libraries are only required at runtime if the correspondig feature is enabled. To add these dependencies on demand, the menu file looks like:

```
## SECTION=fixme

config FOO
       tristate
       select ZLIB if FOO_FOO
       select LIBXML2 if FOO_BAR
       prompt "foo"
       help
         FIXME

if FOO
config FOO_DEBUG
       bool
       prompt "add debug noise"

config FOO_FOO
       bool
       prompt "build foo"

config FOO_BAR
```

```
	bool
	prompt "build bar"

endif
```

Note: Do not add these `select` statements to the correspondig menu entry. They must belong to the main menu entry of the package to ensure that the calculation of the dependencies between the packages is done in a correct manner.

Managing External Runtime Dependencies

Some packages are building all of their components and also installing them into the target's sysroot. But only their *targetinstall* stage decides which parts are copied to the root filesystem. So, compiling and linking of our package will work, because everything required is found in the target's sysroot.

In our example there is a hidden dependency to the math library `libm`. Our new package was built successfully, because the linker was able to link our binaries against the `libm` from the toolchain. But in this case the `libm` must also be available in the target's root filesystem to fulfil the runtime dependency: We have to force PTXdist to install `libm`. `libm` is part of the *glibc* package, but is not installed by default (to keep the root filesystem small). So, it **does not** help to select the `GLIBC` symbol, to get a `libm` at runtime.

The correct solution here is to add a `select LIBC_M` to our menu file. With all the additions above it now looks like:

```
## SECTION=fixme

config FOO
	tristate
	select ZLIB if FOO_FOO
	select LIBXML2 if FOO_BAR
	select LIBC_M
	prompt "foo"
	help
	  FIXME

if FOO
config FOO_DEBUG
	bool
	prompt "add debug noise"

config FOO_FOO
	bool
	prompt "build foo"

config FOO_BAR
	bool
	prompt "build bar"

endif
```

Note: There are other packages around, that do not install everything by default. If our new package needs something special, we must take a look into the menu of the other package how to force the required components to be installed and add the corresponding `selects` to our own menu file. In this case it does not help to enable the required parts in our project configuration, because this has no effect on the build order!

Managing Non Autotool Packages

Many packages are still coming with a plain `Makefile`. The user has to adapt it to make it work in a cross compile environment as well. PTXdist can also handle this kind of packages. We only have to specifiy a special *prepare* and *compile* stage.

Such packages often have no special need for any kind of preparation. We can omit this stage by defining this empty rule:

```
$(STATEDIR)/foo.prepare:
      @$(call targetinfo)
      @$(call touch)
```

To compile the package, we can use `make`'s feature to overwrite variables used in the `Makefile`. With this feature we can still use the original `Makefile` but with our own (cross compile) settings.

Most of the time the generic compile rule can be used, only a few settings are required.

`make` will be called in this case with:

`cd $(FOO_DIR) && $(FOO_MAKE_ENV) $(MAKE) $(FOO_MAKE_OPT)`

So, in the rule file only the two variables `FOO_MAKE_ENV` and `FOO_MAKE_OPT` must be set, to forward the required settings to the package's buildsystem. If the package cannot be built in parallel, we can also add the `FOO_MAKE_PAR := NO`. `YES` is the default.

Note: *FOO* is still the name of our example package. It must be replaced by the real package name.

4.2.4 Patching Packages

There can be various reasons why a package must be patched:

- Package is broken for cross compile environments
- Package is broken within a specific feature
- Package is vulnerable and needs some fixes
- or anything else

PTXdist handles patching automatically. After extracting the archive, PTXdist checks for the existence of a patch directory with the same name as the package. If our package's name is `foo-1.1.0`, PTXdist searches for patches in:

1. platform (`./configs/arm-qemu-3/patches/foo-1.1.0`)
2. project (`./patches/foo-1.1.0`)
3. ptxdist (`<ptxdist installation>/path/patches/foo-1.1.0`)

The patches from the first location found are used. Note: Due to this search order, a PTXdist project can replace global patches from the PTXdist installation. This can be useful if a project sticks to a specific PTXdist revision but fixes from a more recent revision of PTXdist should be used.

4.2.5 Creating Patches for a Package

PTXdist uses the utilities *patch* or *quilt* to work with patches or patch series. We recommend *quilt*, as it can manage patch series in a very easy way. For this manual we assume *quilt* is installed on the build host.

Creating a Patch Series for a Package

To create a patch series for the first time, we can run the following steps. We are still using our *foo-1.1.0* example package here:

We create a special directory for the patch series in the local project directory:

```
$ mkdir -p patches/foo-1.1.0
```

PTXdist expects a `series` file in the patch directory. Otherwise it fails. Due to the fact that we do not have any patch yet, we'll start with an empty `series` file.

```
$ touch patches/foo-1.1.0/series
```

Next is to extract the package (if already done, we must remove it first):

```
$ ptxdist extract foo
```

This will extract the archive and create a symbolic link in the build directory pointing to our local patch directory. Working this way will ensure that we do not lose our created patches if we enter `ptxdist clean foo` by accident. In our case the patches are still present in `patches/foo-1.1.0` and can be used the next time we extract the package again.

All we have to do now is to do the modification we need to make the package work. We change into the build directory and use *quilt* to create new patches, add files to respective patches, modify these files and refresh the patches to save our changes.

We recommend this way when modifying source files. But this way is improper when an autotools based buildsystem itself needs modifications. Refer to section 4.2.6 on how PTXdist can handle this special task.

Adding more Patches to a Package

If we want to add more patches to an already patched package, we can use nearly the same way as creating patches for the first time. But if the patch series comes from the PTXdist main installation, we do not have write permissions to these directories (do NEVER work on the main installation directories, NEVER, NEVER, NEVER). Due to the search order in which PTXdist searches for patches for a specific package, we can copy the global patch series to our local project directory. Now we have the permissions to add more patches or modify the existing ones. Also *quilt* is our friend here to manage the patch series.

If we think that our new patches are valuable also for others, or they fix an error, it could be a good idea to send these patches to PTXdist mainline.

4.2.6 Modifying Autotoolized Packages

Autotoolized packages are very picky when automatically generated files get patched. The patch order is very important in this case and sometimes it even fails and nowbody knows why.

To improve a package's autotools-based build system, PTXdist comes with its own project local autotools to regenerate the autotools template files, instead of patching them. With this feature, only the template files must be patched, the required `configure` script and the `Makefile.in` files are regenerated in the final stages of the *prepare* step.

This feature works like the regular patching mechanism. The only difference is the additional `autogen.sh` file in the patch directory. If it exists and has execution permissions, it will be called after the package was patched (while the *extract* stage is running).

Its content depends on developer needs; for the most simple case the content can be:

```
#!/bin/bash

aclocal $ACLOCAL_FLAGS

libtoolize \
        --force \
        --copy

autoreconf \
        --force \
        --install \
        --warnings=cross \
        --warnings=syntax \
        --warnings=obsolete \
        --warnings=unsupported
```

Note: This way also still not autotoolized package can be autotoolized. We just have to add the common autotool template files (`configure.ac` and `Makefile.am` for example) via a patch series to the package source and the `autogen.sh` to the patch directory.

5 PTXdist Reference

5.1 Variables Reference

The following variables are provided by PTXdist to simplify creating rule files. Every developer should use these variables in every single line in the rule file to avoid any further adaption when external paths are changed.

To get their content related to the current project, we can simply run a:

```
$ ptxdist print PTXDIST_TOPDIR
/usr/local/lib/ptxdist-2010.01.0
```

Replace the PTXDIST_TOPDIR with one of the other generic variables PTXdist provides.

5.1.1 PTXDIST_TOPDIR

Points always to the installation directory of PTXdist.

5.1.2 PTXDIST_WORKSPACE

Everything that references PTXDIST_WORKSPACE will use the active projects's folder.

5.1.3 PTXDIST_SYSROOT_CROSS

PTXDIST_SYSROOT_CROSS points to a directory tree all cross relevant executables, libraries and header files are installed to in the current project. All project's packages built for the host to create data for the target are searching in this directory tree for their dependencies (executables, header and library files). Use $(PTXDIST_SYSROOT_CROSS)/bin to install executables, $(PTXDIST_SYSROOT_CROSS)/include for header files and $(PTXDIST_SYSROOT_CROSS)/lib for libraries.

5.1.4 PTXDIST_SYSROOT_HOST

PTXDIST_SYSROOT_HOST points to a directory tree all host relevant executables, libraries and header files are installed to. All project's packages built for the host are searching in this directory tree for their dependencies (executables, header and library files). Use $(PTXDIST_SYSROOT_HOST)/bin to install executables, $(PTXDIST_SYSROOT_HOST)/include for header files and $(PTXDIST_SYSROOT_HOST)/lib for libraries.

5.1.5 PTXDIST_SYSROOT_TARGET

PTXDIST_SYSROOT_TARGET points to a directory tree all target relevant libraries and header files are installed to. All project's packages built for the target are searching in this directory tree for their dependencies (header and library files). These files are for compile time only (for example to link a target executable against a target library), not for runtime! Use $(PTXDIST_SYSROOT_TARGET)/include for header files and $(PTXDIST_SYSROOT_TARGET)/lib for libraries.

Other useful variables:

5.1.6 CROSS_PATH

Use to find cross tools. This path must be used to create anything that depends on the target's architecture, but needs something running on the host to do the job. Examples:

Creating a jffs2 image from the target's root filesystem This will need a tool running on the host, but it will create data or code that runs on or is used on the target

Building a library for the target If this library needs other resources to be built (other libraries) its configure finds the right information in this path.

5.1.7 HOST_PATH

Used to find host tools. This path must be used to create anything that not depends on the architecture.

5.1.8 ROOTDIR

ROOTDIR points to the root of the target's root filesystem in the current project. Used in very rare cases (to create strange packages based on data in target's root filesystem for example).

5.2 Rule File Macro Reference

Rules files in PTXdist are using macros to get things work. Its highly recommended to use these macros instead to do something by ourself. Using the macros is portable and such easier to maintain in the case a project should be upgraded to a more recent PTXdist version.

This chapter describes the predefined macros in PTXdist and their usage.

Note: This list is not complete yet.

5.2.1 targetinfo

Usage:

```
$(call targetinfo)
```

Gives a feedback, what build *stage* is just started. Thats why it should always be the first call for each *stage*. For the package *foo* and the *compile stage* it will output:

```
--------------------
target: foo.compile
--------------------
```

5.2.2 touch

Usage:

```
$(call touch)
```

Gives a feedback, what build *stage* is just finished. Thats why it should always be the last call for each *stage*. For the package *foo* and the *compile stage* it will output:

```
finished target foo.compile
```

5.2.3 clean

Usage:

```
$(call clean, <directory path>)
```

Removes the given directory <directory path>.

5.2.4 install_copy

Usage:

```
$(call install_copy, <package>, <UID>, <GID>, <permission>, <source> [, <destination>])
```

Installs given file or directory into:

- the project's platform-versatilepb/root/
- the project's platform-versatilepb/root-debug/
- an ipkg packet in the project's platform-versatilepb/packages/

Some of the parameters have fixed meanings:

<package> Name of the IPKG, the macro should work on

<UID> User ID (in a numerical value) the file should use in target's root filesystem

<GID> Group ID (in a numerical value) the file should use in target's root filesystem

<permission> Permission (in an octal value) the file should use in target's root filesystem

The remaining parameters vary with the use case:

The <source> parameter can be:

- a directory path that should be created in target's root filesystem. In this case the <destination> must be omitted. The given path must always start with a / and means the root of target's filesystem.
- an absolute path to a file that should be copied to target's root filesystem. To avoid fixed pathes all packages are providing the <package>_DIR variable. So, this parameter in our *foo* example package can be a $(FOO_DIR)/foo.
- a minus sign (-). PTXdist uses the <destination> parameter in this case to locate the file to copy from. This only works if the package uses the default *install* stage. Only in this case an additional folder in platform-versatilepb/packages will be created for the package and its files. For our *foo* example package this directory is platform-versatilepb/packages/foo-1.1.0.

The <destination> parameter can be:

- omitted if a directory in target's root filesystem should be created. For this case the directory to be created is in the <source> parameter.
- an absolute path and filename with its root in target's root filesysem. It must start with a slash (/). If also the <source> parameter was given, the file can be renamed while copying.
  If the <source> parameter was given as a minus sign (-) the <destination> is also used to locate the source. For our *foo* example package if we give <destination> as `/usr/bin/foo`, PTXdist copies the file `platform-versatilepb/packages/foo-1.1.0/usr/bin/foo`

Due to the complexity of this macro, here are some usage examples:

Create a directory in the root filesystem:

```
$(call install_copy, foo, 0, 0, 0755, /home/user-foo)
```

Copy a file from the package build directory to the root filesystem:

```
$(call install_copy, foo, 0, 0, 0755, $(FOO_DIR)/foo, /usr/bin/foo)
```

Copy a file from the package build directory to the root filesystem and rename it:

```
$(call install_copy, foo, 0, 0, 0755, $(FOO_DIR)/foo, /usr/bin/bar)
```

Copy a file from the package install directory to the root filesystem:

```
$(call install_copy, foo, 0, 0, 0755, -, /usr/bin/foo)
```

5.2.5 install_alternative

Usage:

```
$(call install_alternative, <package>, <UID>, <GID>, <permission>, <destination>)
```

Installs given files or directories into:

- the project's `platform-versatilepb/root/`
- the project's `platform-versatilepb/root-debug/`
- an ipkg packet in the project's `platform-versatilepb/packages/`

The base parameters and their meanings:

<package> Name of the IPKG, the macro should work on

<UID> User ID (in a numerical value) the file should use in target's root filesystem

<GID> Group ID (in a numerical value) the file should use in target's root filesystem

<permission> Permission (in an octal value) the file should use in target's root filesystem

The parameter <destination> is meant as an absolute path and filename in target's root filesystem. PTXdist searches for the source of this file in:

- the local project
- PTXdist's install path

If the file exists in the local project it will be used for the target root filesystem. If the local project does not contain this file, the generic file from the PTXdist installation directory will be used.

If our <destination> is `/etc/foo.conf`, PTXdist checks first, if `./projectroot/etc/foo.conf` exists. Else it uses the file `<ptxdist install path>/generic/etc/foo.conf` instead.

In a multiplatform project PTXdist checks first for the file `./projectroot/etc/foo.conf.<platform name>`, and then for `./projectroot/etc/foo.conf` and falls back to the generic one if noone of these files exist.

5.2.6 install_link

Usage:

```
$(call install_link, <package>, <point to>, <where>)
```

Installs a symbolic link into:

- the project's `platform-versatilepb/root/`
- the project's `platform-versatilepb/root-debug/`
- an ipkg packet in the project's `platform-versatilepb/packages/`

The parameters and their meanings:

<package> Name of the IPKG, the macro should work on

<point to> Path and name the link should point to. Note: This macro rejects absolute paths. If needed use relative paths instead.

<where> Path and name of the symbolic link.

A few usage examples.

Create a symbolic link as `/usr/lib/libfoo.so` pointing to `libfoo.so.1.1.0` in the same directory:

```
$(call install_link, foo, libfoo.so.1.1.0, /usr/lib/libfoo.so)
```

Create a symbolic link as `/usr/bin/foo` pointing to `/bin/bar`:

```
$(call install_link, foo, ../../bin/bar, /usr/bin/foo)
```

5.3 Rule file layout

Each rule file provides PTXdist with the required steps to be done on a per package base:

- get
- extract
- prepare
- compile
- install
- targetinstall

5.3.1 Default stage rules

As for most packages these steps can be done in a default way, PTXdist provides generic rules for each package. If a package's rule file does not provide a specific stage rule, the default stage rule will be used instead.

Omitting one of the stage rules **does not mean** that PTXdist skips this stage!
In this case the default stage rule is used instead.

get Stage Default Rule

If the *get* stage is omitted, PTXdist runs instead:

```
$(STATEDIR)/@package@.get:
                @$(call targetinfo, $@)
                @$(call touch, $@)
```

Which means this step is skipped.

If the package is an archive that must be downloaded from the web, the following rule must exist in this case:

```
$(@package@_SOURCE):
        @$(call targetinfo, $@)
        @$(call get, @package@)
```

extract Stage Default Rule

If the *extract* stage is omitted, PTXdist runs instead:

```
$(STATEDIR)/@package@.extract:
                @$(call targetinfo, $@)
                @$(call clean, $(@package@_DIR))
                @$(call extract, @package@)
                @$(call patchin, @package@)
                @$(call touch, $@)
```

prepare Stage Default Rule

If the *prepare* stage is omitted, PTXdist runs a default stage rule depending on some variable settings.

If the package's rule file defines a `@package@_AUTOCONF` variable (`FOO_AUTOCONF` for our *foo* example), PTXdist treats this package as an autotoolized package and runs:

```
$(STATEDIR)/@package@.prepare:
                @$(call targetinfo)
                @$(call clean, $(@package@_DIR)/config.cache)
                cd $(@package@_DIR)/$(@package@_SUBDIR) && \
                        $(@package@_PATH) $(@package@_ENV) \
                        ./configure $(@package@_AUTOCONF)
                @$(call touch)
```

If the package's rule file defines a @package@_CMAKE variable (FOO_CMAKE for our *foo* example), PTXdist treats this package as a *cmake* based package and runs:

```
$(STATEDIR)/@package@.prepare:
                @$(call targetinfo)
                @$(call clean, $(@package@_DIR)/config.cache)
                cd $(@package@_DIR) && \
                        $(@package@_PATH) $(@package@_ENV) \
                        cmake $(@package@_CMAKE)
                @$(call touch)
```

compile Stage Default Rule

If the *compile* stage is omitted, PTXdist runs instead:

```
$(STATEDIR)/@package@.compile:
                @$(call targetinfo)
                cd $(@package@_DIR) && \
                        $(@package@_PATH) $(@package@_MAKE_ENV) \
                        $(MAKE) $(@package@_MAKE_OPT) $(@package@_MAKE_PAR)
                @$(call touch)
```

Note: @package@_MAKE_PAR can be defined to YES or NO to control if the package can be built in parallel.

install Stage Default Rule

If the *install* stage is omitted, PTXdist runs instead:

```
$(STATEDIR)/@package@.install:
                @$(call targetinfo)
                cd $(@package@_DIR) && \
                        $(@package@_PATH) $(@package@_MAKE_ENV) \
                        $(MAKE) $(@package@_INSTALL_OPT)
                @$(call touch)
```

Note: @package@_INSTALL_OPT is always defined to `install` if not otherwise specified. This value can be replaced by a package's rule file definition.

targetinstall Stage Default Rule

There is no default rule for a package's *targetinstall* state. PTXdist has no idea what is required on the target at runtime. This stage is up to the developer only. Refer to section 5.2 for further info on how to select files to be included in the target's root filesystem.

5.3.2 Skipping a Stage

For the case that a specific stage should be skipped, an empty rule must be provided:

```
$(STATEDIR)/@package@.<stage_to_skip>:
                @$(call targetinfo)
                @$(call touch)
```

Replace the <stage_to_skip> by `get`, *extract*, *prepare*, *compile*, *install* or *targetinstall*.

5.4 PTXdist parameter reference

PTXdist is a command line tool, which is basicly called as:

```
$ ptxdist <action [args]> [options]
```

5.4.1 Setup and Project Actions

menu: This will start a menu frontend to control some of PTXdist's features in a menu based convenient way. This menu handles the actions *menuconfig*, *platformconfig*, *kernel* config, *select*, *platform*, *boardsetup*, *setup*, *go* and *images*.

select <config>: This action will select a userland configuration. This step is only required in projects, where no selected_ptxconfig file is present. The <config> argument must point to a valid userland configuration file. PTXdist provides this feature to enable the user to maintain more than one userland configuration in the same project.

platform <config>: This action will select a platform configuration. This step is only required in projects, where no selected_platform file is present. The <config> argument must point to a valid platform configuration file. PTXdist provides this feature to enable the user to maintain more than one platform in one project.

setup: PTXdist uses some global settings, independent from the project it is working on. These settings belong to users preferences or simply some network settings to permit PTXdist to download required packages.

boardsetup: PTXdist based projects can provide information to setup and configure the target automatically. This action let the user setup the environment specific settings like the network IP address and so on.

projects: If the generic projects coming in a separate archive are installed, this actions lists the projects a user can clone for its own work.

clone <from> <to>: This action clones an existing project from the projects list into a new directory. The <from>argument must be a name gotten from ptxdist projects command, the <to>argument is the new project (and directory) name, creeated in the current directory.

menuconfig: Start the menu to configure the project's root filesystem. This is in respect to userland only. Its the main menu to select applications and libraries, the root filesystem of the target should consist of.

menuconfig platform: This action starts the menu to configure platform's settings. As these are architecture and target specific settings it configures the toolchain, the kernel and a bootloader (but no userland components). Due to a project can support more than one platform, this will configure the currently selected platform. The shortform for this action is platformconfig.

menuconfig kernel: Start the menu to configure the platform's kernel. As a project can support more than one platform, this will configure the currently selected platform. The shortform for this action is kernelconfig.

menuconfig u-boot-v2 | barebox: This action starts the configure menu for the selected bootloader. It depends on the platform settings which bootloader is enabled and to be used as an argument to the menuconfig action parameter. Due to a project can support more than one platform, this will configure the bootloader of the currently selected platform. The shortform for this action is platformconfig.

5.4.2 Build Actions

go: This action will build all enabled packages in the current project configurations (platform and userland). It will also rebuild reconfigured packages if any or build additional packages if they where enabled meanwhile. If enables this step also builds the kernel and bootloader image.

`images`: Most of the time this is the last step to get the required files and/or images for the target. It creates filesystems or device images to be used in conjunction with the target's filesystem media. The result can be found in the `images/` directory of the project or the platform directory.

5.4.3 Clean Actions

`clean`: The `clean` action will remove all generated files while the last `go` run: All build, packages and root filesystem directories. Only the selected configuration files are left untouched. This is a way to start a fresh build cycle.

`clean root`: This action will only clean the root filesystem directories. All the build directories are left untouched. Using this action will regenerate all ipkg packets from the already built packages and also the root filesystem directories in the next `go` action. The `clean root` and `go` action is usefull, if the *targetinstall* stage for all packages should run again.

`distclean`: The `distclean` action will remove all files that are not part of the main project. It removes all generated files and directories like the `clean` action and also the created links in any `platform` and/or `select` action.

project to <target dir>

6 Various Aspects of Daily Work

6.1 Using an External Kernel Source Tree

This application note describes how to use an external kernel source tree within a PTXdist project. In this case the external kernel source tree is managed by GIT.

6.1.1 Cloning the Linux Kernel Source Tree

In this example we are using the officiall Linux kernel developmen tree.

```
jbe@octopus:~$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6
[...]
jbe@octopus:~$ ls -l
[...]
drwxr-xr-x 38 jbe  ptx 4096 2009-03-17 10:21 myprj
drwxr-xr-x 25 jbe  ptx 4096 2009-03-17 10:42 linux-2.6
[...]
```

6.1.2 Configuring PTXdist

To make PTXdist use of this kernel source tree, we run

```
jbe@octopus:~$ ptxdist setup
```

and navigate to **Source Directories**, **Prefix for kernel trees** and enter the base path to the kernel source tree into this menu entry (omit the kernel source tree directory name itself). The kernel source tree directory to be used in this base path will be setup on a per project base. So it will be possible to place more than one kernel source tree in the base path.

6.1.3 Configuring the PTXdist Project

Now we can setup this kernel source tree to be used in our project. PTXdist will handle it in the same way as a kernel part of the project. Due to this, we must setup:

- Some kind of kernel version
- Kernel configuration
- Image type used on our target architecture
- If we want to build modules
- Patches to be used (or not)

Let's setup these topics. Assumption is here, the directory `/ myprj` contains a valid PTXdist project. We just add the kernel component to it.

```
jbe@octopus:~$ cd myprj
jbe@octopus:~/myprj$ ptxdist platformconfig
```

We must enable the **Linux kernel** entry first, to enable kernel building as part of the project. After enabling this entry, we must enter this entry, and:

- Setting up the **kernel version** with the name of kernel source tree directory. In our case shown above it would be `linux-2.6`.
- Selecting the correct image type in the entry **Image Type**.
- Configuring the kernel within the menu entry **patching & configuration**.
  - If no patches should be used on top of the selected kernel source tree, we keep the **patch series file** entry empty. As GIT should help us to create these patches for deployment, it should be kept empty on default in this first step.
  - Select a name for the kernel configuration file and enter it into the **kernel config file** entry.
- As we want to use an existing kernel source tree we also must enable the **Local kernel tree** menu entry.

Now we can leave the menu and store the new setup. The only still missing component is a valid kernel config file now. We can use one of the default config files the Linux kernel supports as a starting point. To do so, we copy one to the location, where PTXdist expects it in the current project. In a multi platform project this location is the platform directory usally in `configs/<platform-directory>`. We must store the file with a name selected in the platform setup menu (**kernel config file**).

6.1.4 Work Flow

Now its up to ourself working on the GIT based kernel source tree and using PTXdist to include the kernel into the root filesystem.

To configure the kernel source tree, we simply run:

```
jbe@octopus:~/myprj$ ptxdist kernelconfig
```

To build the kernel:

```
jbe@octopus:~/myprj$ ptxdist targetinstall kernel
```

To rebuild the kernel:

```
jbe@octopus:~/myprj$ ptxdist drop kernel targetinstall
jbe@octopus:~/myprj$ ptxdist targetinstall kernel
```

6.2 Discovering Runtime Dependencies

Often it happens that an application on the target fails to run, because one of its dependencies is not fulfilled. This section should give some hints on how to discover these dependencies.

6.2.1 Dependencies on Shared Libraries

Getting the missed shared library for example at runtime is something easily done: The dynamic linker prints the missing library to the console.

To check at build time if all other dependencies are present is easy, too. The architecture specific `readelf` tool can help us here. It comes with the OSELAS.Toolchain and is called via `arm-v4t-linux-gnueabi-readelf` (this example uses the one coming with the *arm-v4t-linux-gnueabi* toolchain).

To test the `foo` binary from our new package `FOO`, we simply run:

```
$ arm-v4t-linux-gnueabi-readelf -d platform-versatilepb/ root/usr/bin/foo | grep NEEDED
 0x00000001 (NEEDED)                     Shared library: [libm.so.6]
 0x00000001 (NEEDED)                     Shared library: [libz.so.1]
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
```

We now can check if all of the listed libraries are present in the root filesystem. This works for shared libraries, too. It is also a way to check if various configurations of our package are working as expected (e.g. disabling a feature should also remove the required dependency of this feature).

6.2.2 Dependencies on other Resources

Sometimes a binary fails to run due to missing files, directories or device nodes. Often the error message (if any) that the binary creates in this case is ambiguous. Here the `strace` tool can help us, namely to observe the binary at runtime. `strace` shows all the system calls that the binary or its shared libraries are performing.

`strace` is one of the target debugging tools which PTXdist provides in its `Debug Tools` menu.

After adding strace to the root filesystem, we can use it and observe our `foo` binary:

```
$ strace usr/bin/foo
execve("/usr/bin/foo", ["/usr/bin/foo"], [/* 41 vars */]) = 0
brk(0)                                  = 0x8e4b000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=77488, ...}) = 0
mmap2(NULL, 77488, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f87000
close(3)                                = 0
open("/lib//lib/libm-2.5.1.so", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0p%\0\000"..., 512) = 512
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f86000
fstat64(3, {st_mode=S_IFREG|0555, st_size=48272, ...}) = 0
mmap2(NULL, 124824, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7f67000
mmap2(0xb7f72000, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xb) = 0xb7f72000
mmap2(0xb7f73000, 75672, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7f73000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\332X\1"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1405859, ...}) = 0
[...]
```

Occasionally the output of `strace` can be very long and the interesting parts are lost. So, if we assume the binary tries to open a nonexisting file, we can limit the output to all `open` system calls:

```
$ strace -e open usr/bin/foo
open("/etc/ld.so.cache", O_RDONLY)      = 3
open("/lib/libm-2.5.1.so", O_RDONLY) = 3
open("/lib/libz.so.1.2.3", O_RDONLY) = 3
open("/lib/libc.so.6", O_RDONLY)        = 3
[...]
open("/etc/foo.conf", O_RDONLY) = -1 ENOENT (No such file or directory)
```

The binary may fail due to a missing `/etc/foo.conf`. This could be a hint on what is going wrong (it might not be the final solution).

6.3 Debugging with CPU emulation

If we do not need some target related feature to run our application, we can also debug it through a simple CPU emulation. Thanks to QEMU we can run ELF binaries for other architectures than our build host is.

6.3.1 Running an Application made for a different Architecture

PTXdist creates a fully working root filesystem with all runtime components in `platform-versatilepb/root/`. Lets assume we made a PTXdist based project for a arm926 CPU. Part of this project is our application `myapp` we are currently working on. PTXdist builds the root filesystem and also compiles our application. It also installs it to `usr/bin/myapp` in the root filesystem.

With this preparation we can run it on our build host:

```
$ cd platform-versatilepb/root
platform-versatilepb/root $ qemu-arm -cpu arm926 -L . usr/bin/myapp
```

This command will run the application `usr/bin/myapp` built for an arm926 CPU on the build host and is using all library compontents from the current directory.

For the stdin and -out QEMU uses the regular mechanism of the build host's operating system. Using QEMU in this way let us simply check our programs. There are also QEMU environments for other architectures available.

6.3.2 Debugging an Application made for a different Architecture

Debugging our application is also possible with QEMU. All we need are a root filesystem with debug symbols available, QEMU and an architecture aware debugger.

The root filesystem with debug symbols will be provided by PTXdist, the architecture aware debugger comes with the OSELAS.Toolchain. Two consoles are required for this debug session in this example. We start the QEMU in the first console as:

```
$ cd platform-versatilepb/root-debug
platform-versatilepb/root-debug $ qemu-arm -g 1234 -cpu arm926 -L . usr/bin/myapp
```

Note: PTXdist always builds two root filesystems. `platform-versatilepb/root/` and `platform-versatilepb/root-debug/`. `platform-versatilepb/root/` contains all components without debug information (all binaries are in the same size as used later on on the real target), while all components in `platform-versatilepb/root-debug/` still containing the debug symbols and are much bigger in size.

The added *-g 1234* parameter lets QEMU wait for a GDB connection to run the application.

In the second console we start GDB with the correct architecture support. This GDB comes with the same OSELAS.Toolchain that was also used to build the project. In our example we are using the *arm-v4t-linux-gnueabi* toolchain for our qemu-arm CPU:

```
$ arm-v4t-linux-gnueabi-gdbtui root-debug/usr/bin/myapp
```

This will run a *curses* based GDB. Not so easy to handle (we must enter all the commands and cannot click with a mouse!), but very fast to take a quick look at our application.

At first we tell GDB where to look for debug symbols. The correct directory here is `platform-versatilepb/root-debug/`.

```
(gdb) set solib-absolute-prefix platform-versatilepb/root-debug
```

Next we connect this GDB to the waiting QEMU:

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
[New Thread 1]
0x40096a7c in _start () from root-debug/lib/ld.so.1
```

As our application is already started, we can't use the GDB command `start` to run it until it reaches `main()`. We set a breakpoint instead at `main()` and *continue* the application:

```
(gdb) break main
Breakpoint 1 at 0x100024e8: file myapp.c, line 644.
(gdb) continue
Continuing.
Breakpoint 1, main (argc=1, argv=0x4007f03c) at myapp.c:644
```

The top part of the running gdbtui console will always show us the current source line. Due to the `root-debug/` directory usage all debug information for GDB is available.

Now we can step through our application by using the commands *step*, *next*, *stepi*, *nexti*, *until* and so on.

Note: It might be impossible for GDB to find debug symbols for components like the main C runtime library. In this case they where stripped while building the toolchain. There is a switch in the OSELAS.Toolchain menu to keep the debug symbols also for the C runtime library. But be warned: This will enlarge the OSELAS.Toolchain installation on your harddisk! When the toolchain was built with the debug symbols kept, it will be also possible for GDB to debug C library functions our application calls (so it might worth the disk space).

6.4 Migration between Minor Releases

6.4.1 Simple Upgrade

To migrate an existing project from within one minor release of the second main release to the next one (e.g. 2.0.0 to 2.0.x), we do the following steps:

```
~/my_bsp# ptxdist --force migrate
```

6.5 Migration between Major Releases

6.5.1 Basic Conversion

To migrate an existing PTXdist-1 project to the new PTXdist-2 the following steps are to be done:

To convert a PTXdist-1 project into a PTXdist-2 project simply copy your old `ptxconfig` file into the files `selected_ptxconfig` **and** `selected_platform`.

After that, run

```
~/my_bsp# ptxdist --force platformconfig
```

to configure the platform specific part and check if everything is set up correctly and save these settings. All in the platform unused symbols will now be discarded and the remaining required symbols and their values are saved to the `selected_platform` file.

Repeat the same step with

```
~/my_bsp# ptxdist --force menuconfig
```

In this case all for userland configuration unused symbols are discarded from the original `ptxconfig` file settings and the remaining are saved to the `selected_ptxconfig` file. Check carefully if everything is set up correctly[1]

Now you have successfully separated *platform* and *userland* configuration. This is the main configuration ptxdist-2 uses in a single platform project.

6.5.2 Adaption of Rules Files

A few adaptions in the project local rules files are required, to also migrate them to the new PTXdist infrastructure.

Variables

The names of a few variables have changed.

- `PTXCONF_HOST_PREFIX` must be replaced by `PTXCONF_SYSROOT_HOST`
- `PTXCONF_ARCH` must be replaced by `PTXCONF_ARCH_STRING`
- `IMAGEDIR` must be replaced by `PKGDIR`

Local Source Handling

To retain the old behaviour of ptxdist-1 in handling local sources simply keep the <packetname>_SOURCE variable empty in the corresponding local source rule file.

[1] At least if you are using Busybox as your shell environment you must check carefully if all settings in the previous Busybox version are still present. Each version of Busybox changes various used symbolnames, so it could happen that some settings are lost during the transition.

6.5.3 PTXdist-1 Look and Feel

To keep the PTXdist-1 feeling use the following rules:

- Omit the *platform name* (e.g. keep this menu entry empty) when running `pxtdist platformconfig`
- Open the `selected_platformconfig` file with an editor and search for *SYSROOT_TARGET*, *SYSROOT_HOST* and *SYSROOT_CROSS*. Replace their default settings with:
  - For *SYSROOT_TARGET*: `$PTXDIST_WORKSPACE/local/$PTXCONF_ARCH_STRING`
  - For *SYSROOT_HOST*: `$PTXDIST_WORKSPACE/local-host`
  - For *SYSROOT_CROSS*: `$PTXDIST_WORKSPACE/local-cross`

With these settings you get the same directory structure in a ptxdist-2 project as in ptxdist-1.

6.6 Software Installation and Upgrade

Root filesystems for Linux are usually built as a flash image and pushed into the respective root medium. However, when working with Embedded Linux systems, developers often have the need to

- install new packages
- remove packages
- update packages
- add configuration

Installation of new packages may for example happen if a developer works on a new piece of application code, or if a new library is being written for the embedded system. Package updating may be a requirement even during the system's life cycle, for example for updating a customer application in the field.

Conventional Linux distributions like Debian, SuSE or Fedora use package systems like RPM or DEB to organize their software packages. Unfortunately, these methods require huge packet databases in the root system, which is bad for space constrained embedded systems. So what we need is a packet system that

- offers installation/removement of packages
- has no big database but a very low overhead
- allows packet management features like pre/post scripts (i.e. shutdown a web server, then upgrade it and start it again)

`ipkg` is such a packet system and it is being used in ptxdist. Originally developed for the IBM Itsy, `ipkg` is meanwhile being used on all kinds of embedded Linux projects. The concept of `ipkg` archives is based on the well known Debian packet management format: `ipkg` archives are "ar" archives, containing a tarball with the binary files for the target box, plus management scripts which can be run on pre-install, post-install, pre-rm and post-rm. So even if no `ipkg` management utilities are available, developers can modify the archives with standard shell and console tools.

6.6.1 ipkg Usage in PTXdist

PTXdist end users and packet developers don't have to care directly about `ipkg`. Packages are being created during the *targetinstall* stage, then put into the `platform-versatilepb/packages/` directory. After the targetinstall

stage of a packet was done, this directory contains the `ipkg` packet itself plus, for most packages, a directory with the file content.

The `ipkg` packets contain the binaries for the root filesystem as well as start/stop scripts and meta data about the Unix filesystem permissions; when PTXdist creates the root filesystem which is later flashed into the onboard flash of an embedded system, it takes the information from the `ipkg` packets as the source, in order to make sure that the image is consistent to what the packages contain.

Internally, PTXdist always uses `ipkg` packets to store it's target data. However, the `ipkg` functionality is not always exposed to the target itself. So in order to use packets, navigate to *Disk and File Utilities* and enable `ipkg`. In the `ipkg` sub menu, make sure that the *install /etc/ipkg.conf* switch is active. This config file is necessary to make `ipkg` work at runtime system.

The *ipkg* tool can only be used in images created by `ptxdist images`. It's not fully working within the `platform-versatilepb/root/` subdirectory used as NFS root filesystem.

6.6.2 Packet Installation

A common use case for `ipkg` packets is to push new software to an already deployed target. There must be a communication channel to transfer the packet itself to the embedded system, i.e. by using FTP or a secure connection via SFTP or SSH, so it has to be made sure that such a service is installed and configured on the target. It is necessary that enough free space is available, in order to store the packet itself. A good rule of thumb is to have twice the size of the installed package: while the packet is being installed, the archive as well as it's contents must fit into the system storage. This may be a problem on space constrained systems.

If the packet was transferred, it is necessary to have remote shell access to the box, either via telnet or, if security is an issue, by using SSH. It is also possible to automate this process by using an intelligent update mechanism. The shell is being used to start the necessary commands. If the packet was installed, the `ipkg` archive can be removed again.

6.6.3 Automatic Packet Download

It is also possible to let the embedded system download `ipkg` packets automatically from a network source, without pushing the packets from the outside. In order to do so, a valid URL must be written into the `/etc/ipkg.conf` file. In this case one of the `wget` implementations in PTXdist must be selected, either the one in busybox (**Shell & Console Tools**, **BusyBox**, **Networking Utilities**) or the native implementation (**Networking Tools**).

6.6.4 The ipkg Command

The following sections describe the `ipkg` features.

What's Installed on the System?

To get a list of installed packages, use *list_installed*:

```
# ipkg list_installed
busybox - 1.1.3 -
figlet - 222 -
gcclibs - 4.1.1 -
gdbserver - 6.4 -
glib - 2.8.6 -
glibc - 2.5 -
ipkg - 0.99.163 -
ixp-firmware - 1 -
kernel-modules - 2.6.18 -
libxml2 - 2.6.27 -
mc - 4.6.1 -
memedit - 0.7 -
ncurses - 5.5 -
pciutils - 2.2.1 -
pureftpd - 1.0.21 -
readline - 5.0 -
rootfs - 1.0.0 -
strace - 4.5.14-20061101 -
udev - 088 -
zlib - 1.2.3 -
Successfully terminated.
```

Content of a Package

To see what files are in an installed package, use *files*:

```
# ipkg files udev
Package udev (106) is installed on root and has the following files:
/etc/init.d/udev
/sbin/udevtrigger
/etc/udev/udev.conf
/etc/rc.d/S00_udev
/sbin/udevd
/sbin/udevsettle

Successfully terminated.
```

Adding a Package

Adding a new packet or replacing an already installed one is done by

```
# ipkg install <package-name>.ipk
```

Note the trailing **.ipk**. This extension must be given if the package file is already part of the filesystem. Otherwise `ipkg` tries to download it from the URL configured in `/etc/ipkg.conf`.

Removing a Package

To remove the contents of a package from the running system, ensure that nothing from the package is currently in use. Find out the precise packet name with

```
# ipkg list
```

and remove it's contents from the runtime system with

```
# ipkg remove <package-name>
```

Upgrading a Package

To upgrade a package, first remove it's current contents from the runtime system. In a second step, install the contents of the new `ipkg` package.

```
# ipkg list
# ipkg remove <package-name>
# ipkg <package-name>[.ipk]
```

6.7 Downloading Packages from the Web

Sometimes it makes sense to get all required source archives at once. For example prior to a shipment we want to also include all source archives, to free the user from downloading it by him/herself.

PTXdist supports this requirement by the `export_src` parameter. It collects all required source archives into one given single directory. To do so, the current project must be set up correctly, e.g. the `select` and `platform` commands must be ran prior the `export_src` step.

If everything is set up correctly we can run the following commands to get the full list of required archives to build the project again without a internet connection.

```
$ mkdir my_archives
$ ptxdist export_src my_archives
```

PTXdist will now collect all source archives to the `my_archives/` directory.

Note: If PTXdist is configured to share one source archive directory for all projects, this step will simply copy the source archives from the shared source archive directory. Otherwise PTXdist will start to download them from the world wide web.

6.8 Adding Files into the Build

Most projects handled by PTXdist are needing more than a simple root filesystem. So there is a cause for additional files in every project to be automatically installed when the `ptxdist go` command is running. These files could be HTML files to be provided by a web server running on the target or some special kind of configuration files that PTXdist does not support natively.

Basically we must add two files for this task:

rules/adding_file.in This file describes the corresponding menu for this package. It is called a *menu file*.

rules/adding_file.make This file describes what to do while the `ptxdist go` command is running. It is called a *rule file*.

The name *adding_file* was arbitrarily selected. We can name these files with any name we like.

6.8.1 Creating the Rule and Menu File

To do the installation we need the additional rule file `rules/adding_file.make` and `rules/adding_file.in` mentioned above. These are more or less large files, so we should use PTXdist's template mechanism to create these files, instead of writing our own. The following command must run in a PTXdist project directory:

```
$ ptxdist newpacket simple

ptxdist: creating a new 'simple' package:

ptxdist: enter packet name.......: adding-file
ptxdist: enter version number....:
ptxdist: enter packet author.....: Juergen Beisert
```

Only the *packet name* is required, as it also defines the file name for the menu and rule file. The other two questions we can answer or left them empty.

This command creates the new menu and rule file in the project local `rule/` directory.

6.8.2 Adapting the Generic Rule File

As this rule file is only a generic one, we now must add the instructions to do the real job. We open `rules/adding_file.make` with our favoured editor and navigate to line 56 and/or 101.

It depends on the use of our files where they should get installed. If the files are used at the host side to build other packets the *install* stage in line 56 and below is the right one. Follow the *TODO* text in the *install* stage to do the modification.

If the files are required by the target the *targetinstall* stage is the correct one. For the target we cannot use any simple native copy command, we must create an ipkg package instead. Uncomment all macro calls here and call the *install_copy* macro for each file to be installed on the target root filesystem.

Note:
There is no restriction where the files are come from to insert them into target's root filesystem. We can give any source path and filename. If we are using a special directory not already existing on target's root filesystem we must also create it.

Refer section 5.1 for further details about helpful variables PTXdist provides for the *install* stage and section 5.2 for further details about IPKG related macros to be used in the *targetinstall* stage.

6.8.3 Adapting the Menu File

Refer the section 4.2.1 on how to add more menuentries (if required) and how to locate this new menu in PTXdist's menu hierarchy.

7 Document Revisions

2010/01/08	Initial Revision
2010/01/11	Section ”Adding simple Files to the Build Process” added
2010/01/11	Section ”Variables reference” added
2010/01/20	Section ”Debugging with CPU emulation” added
2010/01/21	Section ”PTXdist parameter reference” added
2010/04/15	Section ”Adding new Packages” extended
2010/04/16	Section ”Using Existing Toolchains” extended
	×

8 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.

8.1 Mailing Lists

8.1.1 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.

8.1.2 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.

8.2 News Groups

8.2.1 About Linux in Embedded Environments

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

comp.os.linux.embedded

8.2.2 About General Unix/Linux Questions

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

de.comp.os.unix.programming

8.3 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.

8.4 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