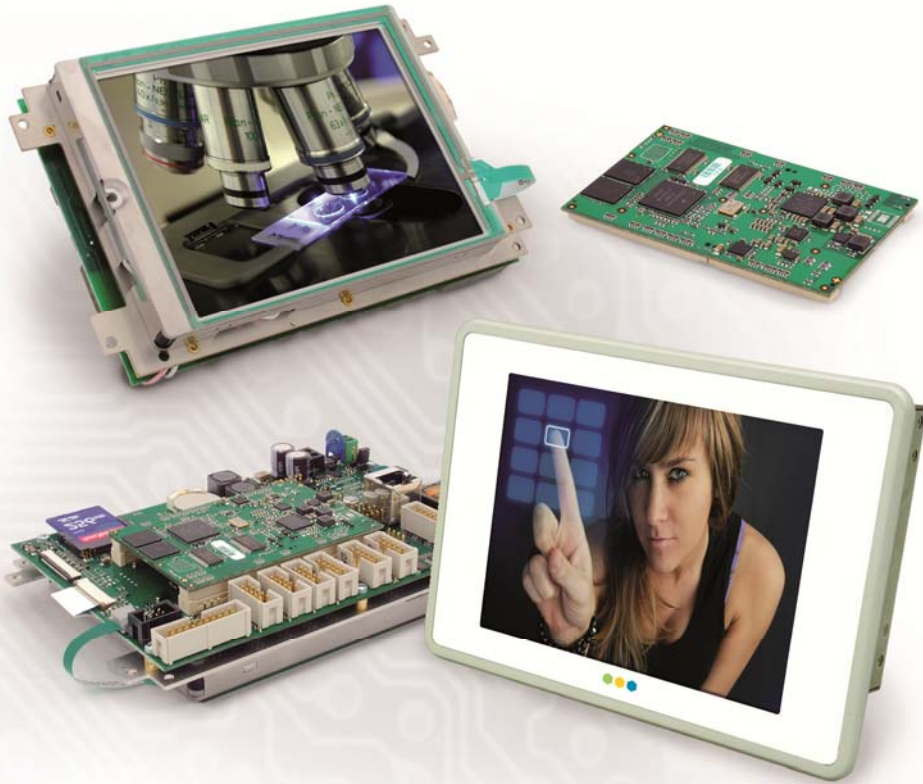


**Garz & Fricke**

**Embedded Computer Systems**

Embedded Computer Systems

# Preliminary



**Linux - Manual**



Zuverlässige  
Qualität  
Made in Germany

## 1 Important hints

Thank you very much for purchasing a Garz & Fricke product. Our products are dedicated to industrial use and therefore we suppose extended technical knowledge and practice in working with such products.



### Notes:

- Microsoft, Windows, CE, Windows NT, Visual Studio, Visual C++, MFC and embedded Visual C++ are registered trademarks, trademarks or products of Microsoft Corp. Redmond, USA.
- Sharp is a registered trademark of Sharp Electronics Europe GmbH
- Freescale and i.MX31 are registered trademarks of Freescale Semiconductor, Inc.
- ARM, ARM9, ARM926, ARM11, ARM1136 are registered trademarks of ARM Ltd.
- eCosCentric and eCos are registered trademarks of eCosCentric Ltd.
- RedBoot is a registered trademark of Red Hat Inc.
- Linux is a registered trademark of Linus Torvalds
- Flash'nGo, Auckland and Auckland Starter Kit are registered trademarks or products of Garz & Fricke GmbH, Hamburg.
- CUPID, CUPID Starter Kit, NESO, NESO Starter Kit, ADELAIDE and ADELAIDE Starter Kit are registered trademarks or products of Garz & Fricke GmbH, Hamburg.
- JUPITER Display-on-Module is a registered trademark of Garz & Fricke GmbH, Hamburg
- OSELAS and PTXdist are registered trademarks of Pengutronix e.K., Hildesheim

Their use is subject to national and international laws and agreements. Every use of these names in this documentation occurs subject to the legal regulations. While trademark symbols may be omitted for the purpose of simplification, they are implied when the names of the trademarks are used in the remainder of this document and should be interpreted as present.



### Disclaimer:

Although Garz & Fricke makes every effort to keep the information in this manual correct and up to date, Garz & Fricke gives no guarantees for the currency, correctness, completeness or quality of the information provided. No responsibility is accepted by Garz & Fricke, the author or translator for material or other damages caused by the use or non-use of erroneous or incomplete information, as long as no intent or gross negligence on our part or the part of the author or translator is proven. All offers are without obligation. Garz & Fricke reserves the right to change, extend or delete parts of the manual without notice, or to remove the publication temporarily or permanently.

### Copyrights:

© 2010

This document is issued under copyright by  
Garz & Fricke GmbH, Tempowerkring 2, 21079 Hamburg, Germany.



### Support contact:

Phone +49 (0) 40 / 791 899 - 30  
Fax +49 (0) 40 / 791 899 - 39  
Email ► [support@garz-fricke.com](mailto:support@garz-fricke.com)  
URL ► [www.garz-fricke.com](http://www.garz-fricke.com)

### Document Information:

Garz & Fricke  
Embedded Systems · Linux Manual  
Revision: 0.1  
Date of issue: 29.03.2010

## Content

<b>1</b>	<b>Important hints</b>	<b>2</b>
<b>2</b>	<b>Introduction</b>	<b>4</b>
<b>3</b>	<b>Overview</b>	<b>4</b>
<b>4</b>	<b>Construction of Garz &amp; Fricke embedded Linux systems</b>	<b>6</b>
<b>5</b>	<b>Deploying the Linux system to the target</b>	<b>9</b>
<b>6</b>	<b>Access to / from the target system</b>	<b>13</b>
<b>7</b>	<b>Building user applications (“Hello World!” applications) for the target systems</b>	<b>13</b>
<b>Annex A:</b>	<b>Document History</b>	<b>24</b>

## 2 Introduction

In addition to Windows CE Garz & Fricke systems can run with a Linux operating system. Linux is a royalty-free open-source operating system. There are various possibilities to build a Linux operating system. Garz & Fricke uses Pengutronix PTXdist for building embedded Linux systems for Garz & Fricke target platforms by providing a Board Support Package (BSP).

PTXdist is a build system specializing in building embedded Linux systems. In addition to the build system itself a GNU cross tool chain is needed to produce machine specific executable opcode for the target system. Tool chains for Garz & Fricke BSPs are built Pengutronix OSELAS.Toolchain(), a PTXdist Project for building GNU tool chains for various target platforms.

This manual contains information about the handling of Linux with PTXdist and OSELAS.Toolchain() for Garz & Fricke systems. For further documentation on PTXdist please refer to the official PTXdist web-sites:

- [▶ http://www.ptxdist.org/software/ptxdist/index\\_en.html](http://www.ptxdist.org/software/ptxdist/index_en.html)

Information regarding OSELAS.Toolchain() can be found at:

- [▶ http://www.pengutronix.de/oselas/toolchain/index\\_en.html](http://www.pengutronix.de/oselas/toolchain/index_en.html)

## 3 Overview

Generally a Linux System consists of a boot loader, a Linux kernel, and a root file system.

There are several boot loaders for the various Linux platforms. For desktop PC Linux systems GRUB or LILO is commonly used. Those boot loaders are started by hardwired PC-BIOS.

Embedded Systems do not have a PC like BIOS. In most cases they are started from a FLASH device. For this purpose there are special boot loaders like RedBoot, U-Boot or Barebox. Garz & Fricke embedded Linux systems are started by RedBoot.

The kernel includes the micro kernel specific parts of the Linux OS and if configured some internal device and subsystem drivers.

The root file system is simply a file system. It contains the Linux file system hierarchy. Depending of the system configuration the root file system contains:

- System configuration files
- Shared runtime libraries
- Dynamic device and subsystem drivers so called “loadable kernel modules” in contrast to kernel-included device and subsystem drivers.
- Executable programs for system handling
- Fonts
- ...

There is usually a special standard set of runtime libraries found in almost every Linux system including standard C/C++ runtime, math support, threading support etc..

The main difference between most embedded Linux systems is the way of graphics handling. The following examples are commonly used in embedded Linux systems:

- Qt-Embedded on top of a Linux frame buffer device
- Qt-Embedded on top of DirectFB graphics acceleration library
- Qt-Embedded on top of a X-Server
- GTK+ on top of DirectFB graphics acceleration library
- GTK+ on top of a X-Server
- Nano-X / Microwindows on top of a Linux frame buffer device
- (GTK+ and Qt-Embedded may also run on top of this)

Sometimes a window manager with small footprint may additionally be used.

Garz & Fricke embedded Linux systems are using Qt-Embedded on top of a Linux frame buffer device.

In addition to these three parts of a Linux system most embedded boot loaders need some setup parameters passed to the OS. The RedBoot boot loader of Garz & Fricke embedded systems manages these parameters as an XML text string.

If desired there will also be a boot logo shown on a display after system start up.

The previous stated parts are written as images into flash partitions in the system.

Garz & Fricke embedded Linux systems have per default the following partition scheme:

- RedBoot: the boot loader binary image
- FIS directory: the boot loader XML parameters
- Redundant FIS: the mirrored boot loader XML parameters
- logo.png: the boot logo
- kernel: the Linux kernel binary image
- rootfs: the root file system image

For the Linux kernel image there are two image types that RedBoot can start:

- zImage: This image is compressed and includes a decompressor.
- Image: This image is uncompressed and does not use a decompressor

Normally the uncompressed image is twice the size of a compressed image but faster in execution time. Overall, it is faster to use an uncompressed image if a NAND flash is used. Likewise, the Linux kernel can handle several file system types for the root file system. In Garz & Fricke embedded Linux systems an “Unsorted Block Image File System” (UBIFS) is used.

It is a very complex task to build a Linux system. In addition to a GNU tool chain consisting of a compiler (gcc), a set of binary utilities (as, readelf, strip...), a set of basic runtime libraries (libc, libm, ...). a debugger (gdb) some tools like GNU autotools, pkg-config, make, qmake, cmake, sed, ... are needed to satisfy this task. For this reason the complexity of the build process is hidden in the Linux build system PTXdist.

In contrast to a desktop Linux system which is completely built with a native GNU tool chain, an embedded Linux system is built with GNU a cross tool chain. As stated before a GNU cross tool chain must have the ability to produce target specific opcode while running on a host system.

To distinguish between the native GNU tool chain and the GNU cross tool chain, the GNU cross tools are prefixed with a triplet. E.g. if the tool chain produces opcode for an armv5te core having library routines that can deal with Linux system calls satisfying the GNU EABI, the Compiler is named “arm-v5te-linux-gnueabi-gcc”, the assembler is named “arm-v5te-linux-gnueabi-as”, .... Sometimes a tool chain prefix is only named “arm-linux-“ or something else. This depends on the tool chain vendor. Garz & Fricke uses the naming convention stated before.

Information regarding embedded Linux systems can be found at:

- “Building Embedded Linux systems 2<sup>nd</sup> Edition”, Karim Yaghmour, John Masters, Gilad Ben-Yossef, Philippe Gerum, O’Reilly, 2008, ISBN: 978-0-596-52968-0.

Information regarding Linux structure issues in general can be found at:

- <http://tldp.org/LDP/Pocket-Linux-Guide/html/>

and

- <http://www.linuxfromscratch.org/>

Information about Qt/Embedded for Linux can be found at:

- <http://qt.nokia.com/products/platform/qt-for-embedded-linux>

## 4 Construction of Garz & Fricke embedded Linux systems

The construction of the embedded Linux system is divided in three parts:

- Installing PTXdist
- Installing the GNU cross tool chain for the target architecture
- Building the BSP for the target platform

The necessary steps are described in the following sections.

### 4.1 Installing PTXdist

PTXdist only supports Linux as host systems.

To install PTXdist the following files from the CD / USB stick shipped with the starter kit for the platform have to be extracted:

- Tools/ptxdist-<version>.tgz
- Tools/ptxdist-<version>-patches.tgz

The PTXdist and patches packets have to be extracted into a 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
```

The next step is to extract the archives:

```
# tar -zxf ptxdist-<version>.tgz
# tar -zxf ptxdist-<version>-patches.tgz
```

If everything went well, we now have a PTXdist-<version> directory, so we can change into it:

```
$ cd ptxdist-<version>
```

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 <version> 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 before re-running the configuration script.

When the configuration-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 its 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
```

## 4.2 Installing the GNU cross tool chain for the target architecture

There are two possible ways to install the tool chain:

- Short way: Install a pre-compiled tool chain
- Build the tool chain with PTXdist

There are different tool chains needed depending on the platform the software will be built for:

- For a NESO platform a “arm-v5te-linux-gnueabi” tool chain is needed because the iMX27 MCU includes an arm v5te processor core.
- For a CUPID platform a “arm-1136jfs-linux-gnueabi” tool chain is needed because the iMX27 MCU includes an arm 1136jfs processor core.

For the following chapters, please replace “<arm-core>” with “v5te” and “<PLATFORM>” with “NESO” if the software will be built for a NESO platform. Likewise, replace “<arm-core>” with “1136jfs” and “<PLATFORM>” with “CUPID” if the software will be built for a CUPID platform.

### 4.2.1 Short way: Install a pre-compiled tool chain

Extract the pre-compiled tool chain from on the host system from the CD / USB stick folder Tools as the user `root`. (The following example refers to an arm-<arm-core>-linux-gnueabi tool chain):

```
$ sudo tar -xf arm-<arm-core>-linux-gnueabi.tar.bz2 -C /
```

The tool chain bin directory is now located at

```
/opt/OSELAS.Toolchain-<version>/arm-<arm-core>-linux-gnueabi/gcc-<version>-glibc-<version>-binutils-<version>-<version>-sanitized/bin.
```

### 4.2.2 Build the tool chain with PTXdist

PTXdist handles tool chain building as a simple project, like all other projects, too. So we can download the OSELAS.Toolchain bundle and build the required tool chain for the OSELAS.BSP() 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-<version>/`. 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-<version>
chown <username> /opt/OSELAS.Toolchain-<version>
```

```
chmod a+rwX /opt/OSELAS.Toolchain-<version>
```

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

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-GUF-Linux-<version> for e.g. an arm-<arm-core>-linux-gnueabi based board support package is

```
arm-<arm-core>-linux-gnueabi_gcc-<version>_glibc-<version>_binutils-<version>_kernel-<version>-sanitized
```

So the steps to build this tool chain are:

```
$ tar xf OSELAS.Toolchain-<version>.tar.bz2
$ cd OSELAS.Toolchain-<version>
$ ptxdist select ptxconfigs/ ↵
  arm-<arm-core>-linux-gnueabi_gcc-<version>_glibc-<version>_binutils- ↵
  <version>_kernel-<version>-
  sanitized.ptxconfig
$ ptxdist go
```

The build will take between 20 min and 2 hours depending on the host system.

The tool chain bin directory is now located at

```
/opt/OSELAS.Toolchain-<version>/arm-<arm-core>-linux-gnueabi/gcc-<version>-glibc-<version>-binutils-
<version>-<version>-sanitized/bin.
```

It is strongly recommended, that the newly tool chain path will be write protected after the build.

### 4.3 Building the BSP for the target platform

The following steps describe the way how to build a Linux BSP for a Garz & Fricke platform. In this step the Linux kernel and the root file system is build. The boot loader RedBoot is already pre-installed on the target.

The file OSELAS-BSP-GUF-Linux-<version>.tar.bz must be copied from the CD / USB stick shipped with the starter kit to the host system. The file can be found in the subfolder "BSP".

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

```
$ tar -zxf OSELAS.BSP-GUF-Linux-<Version>.tar.bz2
$ cd OSELAS.BSP-GUF-Linux-<Version>.tar.bz2
```

The next step is to choose the hardware platform:

```
$ ptxdist platform configs/<PLATFORM>/platformconfig
```

The toolchain is found automatically if it is installed in the default location:

```
found and using toolchain:
'/opt/OSELAS.Toolchain-<version>/arm-<arm-core>-linux-gnueabi/ ↵
gcc-<version>-glibc-<version>-binutils-<version>-kernel-<version>-sanitized/bin'
```

Otherwise you have to select the toolchain manually:

```
$ ptxdist toolchain /opt/OSELAS.Toolchain-<version>/arm-<arm-core>-linux-gnueabi/ ↵
gcc-<version>-glibc-<version>-binutils-<version>-kernel-<version>-sanitized/bin
```

The next step is to choose the system configuration:

```
$ ptxdist select configs/ptxconfig.guf
```

Now, the BSP can be built:

```
$ ptxdist go
```

Depending on your host system this step will take between 1 and 3 hours. The main time is required by building the Qt packet.

At the end of the build process you will see the following output on the build console:

```
-----
For a proper NFS-root environment, some device nodes are essential.
In order to create them root privileges are required.
-----

(Please press enter to start 'sudo' to gain root privileges.)

WARNING: NFS-root might not be working correctly!
```

At this moment you have missed the creation of the essential device nodes for the target system. Device nodes can (and should ever be) created only with root privileges. But don't worry about that. Simply, type again:

```
$ ptxdist go
```

After a few seconds you have once again the possibility to create the device nodes:

```
-----
For a proper NFS-root environment, some device nodes are essential.
In order to create them root privileges are required.
-----

(Please press enter to start 'sudo' to gain root privileges.)
```

Press enter and you get the following output:

```
root's password:
[enter password]
Creating device node: platform-<PLATFORM>/root/dev/null
Creating device node: platform-<PLATFORM>/root/dev/zero
Creating device node: platform-<PLATFORM>/root/dev/console
Creating device node: platform-<PLATFORM>/root-debug/dev/null
Creating device node: platform-<PLATFORM>/root-debug/dev/zero
Creating device node: platform-<PLATFORM>/root-debug/dev/console
```

The root file system is now located at "OSELAS.BSP-GUF-Linux-<version>/platform-<PLATFORM>/root". The kernel is located at "OSELAS.BSP-GUF-Linux-<version>/platform-<PLATFORM>/images/linuximage".

In order to build an UBIFS from the root file system we have to type:

```
$ ptxdist images
```

If you have built the BSP for <PLATFORM>, the UBIFS image is now located at "OSELAS.BSP-GUF-Linux-<version>/platform-<PLATFORM>/root.ubi".

## 5 Deploying the Linux system to the target

The deployment of the Linux system has to be distinguished into two cases:

- Deployment in the development phase
- Release deployment

### 5.1 Deployment in the development phase

It is common to embedded Linux developers to use a technique called "root over NFS" and loading the Linux kernel from a TFTP server during the development phase. In this manner the FLASH stays unused and many write cycles to the FLASH are saved.

In order to make the host ready for this technique there must be a TFTP server and a NFS server installed on the host system. Usually, the packages "tftp" and "nfs-utils" can be installed on every Linux

distribution.

In the usual case the TFTP server must be configured as follows in the “/etc/xinetd.d/tftpd” file on the host system to provide the directory “/tftpboot” as TFTP directory:

```
service tftp
{
    socket_type          = dgram
    protocol             = udp
    wait                = yes
    user                 = root
    server               = /usr/bin/in.tftpd
    server_args          = -s /tftpboot
    disable              = no
}
```

The NFS server must be configured as follows in the “/etc/exports” file on the host system to provide the directory “/rootfs” as NFS share for the target with e.g. the IP address 192.168.1.1:

```
/rootfs 192.168.1.1(rw,no_root_squash)
```

Most Linux distributions provide tools (e.g. YAST on SuSE) to perform those settings. Consult your Linux distribution documentation for further information.

If the host setup stated before is done successfully you can copy the kernel from “OSELAS.BSP-GUF-Linux-<version>/platform-<PLATFORM>/images” to the TFTP directory “/tftpboot” and the contents of the root file system from “OSELAS.BSP-GUF-Linux-<version>/platform-<PLATFORM>/root to the NFS share “/rootfs”:

```
$ cp platform-<PLATFORM>/images/linuximage /tftpboot/linuximage
$ rm -Rf /rootfs/*
$ cp -R platform-<PLATFORM>/root/* /rootfs/
```

Now, we are ready to connect the target to the host system. Be sure that the target is connected to the host with a null modem cable for a serial terminal connection and with an Ethernet connection for networking.

Start your favourite terminal program on your host system (e.g. minicom) with the following settings:

- Baud rate: 115200
- 8 data bits
- No parity
- 1 stop bit
- No hardware flow control
- No software flow control

Power on your target and interrupt the boot loader RedBoot with “STRG+C” to have access to its console:

```
RedBoot>
```

Setup RedBoot to load the kernel from the TFTP server and mount the root file system from NFS share. (In this example it is assumed that the IP address of the host is 192.168.1.111)

```
RedBoot> fc
Run script at boot: true
Boot script:
.. fi lo kernel
.. exec -c "console=ttymx0, 115200 mem=124M ubi.mtd=5 root=ubi0_1 rootfstype=ubifs"
Enter script, terminate with empty line
>> net
>> load -r -v -b 0x80100000 linuximage
>> exec -c "console=ttymx0, 115200 mem=124M root=/dev/nfs rw ↵
nfsroot=192.168.1.111:/rootfs"
>> [ENTER]
[...]
```

You will be asked further for additional settings. Change the gateway IP address, the local IP address and the local IP address mask according to your network configuration.

If all settings are done, you have to confirm the update of the configuration:

```
Update RedBoot non-volatile configuration - continue (y/n) y
```

Now re-power your target.

The first time the newly installed root file system is mounted by the target the boot process is delayed by prelink while pre-linking the shared libraries and the SSH daemon while generating the RSA and DSA keys. This is done only once by the first system start up:

```
...
running prelink...
generating rsa key...done
generating dsa key...done
...
```

If the system boot has finished correctly you will see the Linux login shell in your terminal program:

```
ptx login: [Enter a password]
```

You can login as user “root” and you do not need a password by default.

For security reasons you should set a root password by typing:

```
root@ptx:~ passwd root
New password: [Enter a password]
Retype password: [Retype password]
```

Furthermore you should see the Garz & Fricke demo application on the display.

## 5.2 Release deployment

When the development phase has finished the kernel and the root file system can be written into the system FLASH and executed from there.

In chapter 4.3 we created two image files:

- linuximage – the Linux kernel
- root.ubi – the root file system in form of an UBIFS

If you have done all steps before the “linuximage” file is already copied to the TFTP directory.

The same must be done with the “root.ubi” file:

```
$ cp platform-<PLATFORM>/images/root.ubi /tftpboot/
```

To write these images to the system flash re-power your target and interrupt the boot loader RedBoot with “STRG+C “ to have access to its console:

```
RedBoot>
```

By typing

```
RedBoot> fis list
... Read from 0x000d04e0-0x000e04e0 at 0xe00c0000:
.
Name           FLASH addr  Mem addr    Length      Entry point
RedBoot        0xE0000000  0xE0000000  0x00080000  0x00000000
Redundant FIS  0xE0080000  0xE0080000  0x00040000  0x00000000 backup
FIS directory  0xE00C0000  0xE00C0000  0x00040000  0x00000000 current
logo.png       0xE0300000  0x80100000  0x00020000  0x80100000
kernel         0xE0320000  0x80100000  0x00400000  0x80100000
rootfs         0xE0720000  0x80100000  0x0BA00000  0x80100000
```

the current partition scheme is shown.

Remove the old kernel and root file system partition:

```
RedBoot> fis delete kernel
...
RedBoot> fis delete rootfs
...
```

Now, we can load the kernel image from the TFTP server to the target RAM:

```
RedBoot> load -r -v -b 0x80100000 linuximage
```

If the image file is successfully loaded we can create a kernel FLASH partition with it:

```
RedBoot> fis create -b 0x80100000 -l 0x400000 -e 0x80100000 -r 0x80100000 kernel
```

The same can be done with the root file system image:

```
RedBoot> load -r -v -b 0x80100000 root.ubi
```

The partition is created a little different. After loading the file to the target RAM you have to recognize the file size of the image. The address range is shown in the console output e.g.:

```
...
Raw file loaded
Entry point: 0x80100000, address range: 0x80100000-0x81ebffff
...
```

With this Information the image size for the above example can be calculated:  
 $0x81ebffff - 0x80100000 = 0x1dbffff$

Calculate the current file size of the image likewise. The calculated image size must be set with the parameter “-s” for the partition creation command:

```
RedBoot> fis create -b 0x80100000 -l 0xba00000 -s 0x1dbffff -e 0x80100000 rootfs
```

Finally, RedBoot must be configured to execute the images from FLASH.

Change the RedBoot boot script as follows. You do not have to enter all the other settings again. Type a “.” After the timeout question and confirm the update of the configuration:

```
RedBoot> fc
Run script at boot: true
Boot script:
.. net
.. load -r -v -b 0x80100000 linuximage
.. exec -c "console=ttymxc0, 115200 mem=124M root=/dev/nfs rw ↵
  nfsroot=192.168.1.111:/rootfs"
.. fi lo kernel
.. exec -c "console=ttymxc0,115200 mem=124M ubi.mtd=5 root=ubi0_1 rootfstype=ubifs"
Enter script, terminate with empty line
>> fi lo kernel
>> exec -c "console=ttymxc0,115200 mem=124M ubi.mtd=5 root=ubi0_1 ↵
  rootfstype=ubifs"
>> [ENTER]
Boot script timeout (1000ms resolution): 1.
Update RedBoot non-volatile configuration - continue (y/n) y
```

If the images are written correctly into the FLASH memory and the configuration is changed accordingly the system should now boot from the FLASH memory.

The first time the newly installed root file system is mounted by the target the boot process is delayed by prelink while pre-linking the shared libraries and the SSH daemon while generating the RSA and DSA keys. This is done only once by the first system start up:

```
...
running prelink...
generating rsa key...done
generating dsa key...done
...
```

If the system has booted correctly you will see the Linux login shell in your terminal program:

```
ptx login: [Enter a password]
```

You can login as user “root” and you do not need a password by default.

For security reasons you should set a root password by typing:

```
root@ptx:~ passwd root
New password: [Enter a password]
Retype password: [Retype password]
```

Furthermore you should see the Garz & Fricke demo application on the display.

There is a general touch screen calibration preinstalled on the system. To get an exact touch screen calibration for the current <PLATFORM> sample you have to run “ts\_calibrate” on the target system.

Before you can calibrate the touch screen, you have to stop the demo application on the target which is started automatically after system boot:

```
root@ptx:~ /etc/init.d/qt4-guf-demo stop
```

Now, the “ts\_calibrate” tool can be executed:

```
root@ptx:~ /ts_calibrate
```

Calibrate the touch screen by touching the 5 crosshairs on the display.

After you have finished the touch screen calibration you can reboot the system. And the new touch screen calibration is used.

## 6 Access to / from the target system

In addition to the terminal console on the serial port the target system can be accessed with SSH.

The first time you access the target system from the host system, the target is added to the list of known hosts. To establish the connection you have to confirm this step.

To login with SSH type on the host system (SSH must be installed on the host system):

```
$ ssh root@<IP-address-target>
root@<IP-address-target>'s password: [Enter root password]
```

To copy e.g. the file “myapp” from the hosts current working directory to the targets “/usr/bin” directory type:

```
scp ./myapp root@<IP-Adresse-Target>:/usr/bin/myapp
root@<IP-address-target>'s password: [Enter root password]
```

To copy e.g. the targets “/usr/bin/myapp” file back to the hosts current working directory type:

```
scp root@<IP-Adresse-Target>:/usr/bin/myapp ./myapp
root@<IP-address-target>'s password: [Enter root password]
```

Another possibility to access a shell on the target is to use TELNET (TELNET must be installed on the host system):

```
$ telnet <IP-Adresse-Target>
Trying <IP-Adresse-Target>...
Connected to <IP-Adresse-Target>.
Escape character is '^]'.

Ptx login: root
Password: [Enter password]
root@ptx:~
```

You can also copy files from the host system on the target system with the TFTP client.

On the host copy e.g. the file “myapp” to your TFTP directory:

```
$ cp ./myapp /tftpboot/
```

On the target system (e.g. logged in on a serial console) you can type

```
root@ptx:~ cd /usr/bin
root@ptx:/usr/bin tftp -g -r myapp <IP-Adresse-Target>
```

to cp “myapp” to the targets “/usr/bin” directory.

If you do not want to access the target system as superuser root feel free to use the “adduser” and “addgroup” commands on the target system.

## 7 Building user applications (“Hello World!” applications) for the target systems

The options are supported to build user applications on the target systems:

- Building non-GUI user applications (without Qt) outside from PTXdist
- Building non-GUI user applications (without Qt) integrated into PTXdist
- Building GUI user applications (with Qt) outside from PTXdist
- Building GUI user applications (with Qt) integrated into PTXdist

The following sections describe how to build a “Hello Wold!” application for each option.

## 7.1 Building non-GUI user applications (without Qt) outside from PTXdist

Create a directory in your home directory on the host system and change to it:

```
$ cd ~
$ mkdir myapp
$ cd myapp
```

Touch the empty files “main.cpp” and “Makefile” into this directory:

```
$ touch main.cpp Makefile
```

With your favourite editor edit the contents of the “main.cpp” file as follows:

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    cout << "Hello World !" << endl;
    return 0;
}
```

With your favourite editor edit the contents of the “Makefile” as follows:

```
CROSS_COMPILE=/opt/OSELAS.Toolchain-<version>/arm-<arm-core>-linux-gnueabi/ ↵
gcc-<version>-glibc-<version>-binutils-<version>-kernel-<version>-sanitized/bin/arm-<arm-
core>-linux-gnueabi-

myapp: main.cpp
$(CROSS_COMPILE)g++ -o $@ $<
$(CROSS_COMPILE)strip $@

clean:
rm -f myapp *.o *~ *.bak
```

If the toolchain is installed in the default directory, this example compiles for the target system by typing

```
$ make
```

in the “myapp” directory. Otherwise the CROSS\_COMPILE variable must be set according to the toolchain installation.

Now there is the “myapp” executable build in the “myapp” directory. You can transfer this application to the target systems “/usr/bin” directory in one of the ways described in chapter 6.

In order to let the target system start this application automatically instead of the demo application you have to copy the file “/etc/init.d/qt4-guf-demo” to “/etc/init.d/myapp” on the target system. (You can do this by login e.g. on a serial console with a terminal program, TELNET or SSH as described above):

```
root@ptx:~ cp /etc/init.d/qt4-guf-demo /etc/init.d/myapp
```

Edit “/etc/init.d/myapp” with the editor “nano” on the target system by typing:

```
root@ptx:~ nano /etc/init.d/myapp
```

Change the contents of this file as follows:

```
#!/bin/sh

./etc/profile

case "$1" in
start)
    start-stop-daemon -m -p /var/run/myapp.pid -b -a /usr/bin/myapp -S
    ;;
stop)
    start-stop-daemon -p /var/run/myapp.pid -K
    ;;
*)
    echo "Usage: /etc/init.d/myapp {start|stop}" >&2
```

```

        exit 1
        ;;
    esac

```

Save the changes by pressing “STRG+O” and accept the target file name as suggested by pressing “[ENTER]”. Leave the “nano” editor by pressing “STRG+X”

Finally, the current start up link to “/etc/init.d/qt4-guf-demo” “/etc/rc.d/S95qt4-guf-demo” must be deleted and a new start up link “/etc/rc.d/S95myapp” to “/etc/init.d/myapp” must be created:

```

root@ptx:~ rm -f /etc/rc.d/S95qt4-guf-demo
root@ptx:~ ln -s /etc/init.d/myapp /etc/rc.d/S95myapp

```

After system reboot your application starts automatically and “Hello World!” is written to the console.

## 7.2 Building non-GUI user applications (without Qt) integrated into PTXdist

Create a directory in “OSELAS.BSP-GUF-Linux-<version>/local\_src” on the host system and change to it:

```

$ cd local_src
$ mkdir myapp
$ cd myapp

```

Touch the empty files “main.cpp” and “Makefile.am” into this directory:

```

$ touch main.cpp Makefile.am

```

With your favourite editor edit the contents of the “main.cpp” file as follows:

```

#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    cout << "Hello World !" << endl;
    return 0;
}

```

With your favourite editor edit the contents of the “Makefile.am” file as follows:

```

bin_PROGRAMS = \
    myapp

myapp_SOURCES = \
    main.cpp

```

Now run “autoscan” and move the created “configure.scan” file to “configure.ac”:

```

$ autoscan
$ mv configure.scan configure.ac

```

With your favourite editor change the contents of the “configure.ac” file as follows. (You should use your email address instead of the stated one):

```

#                                     -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ([2.63])
AC_INIT([myapp], [1.0], [carsten.behling@garz-fricke.com])
AC_CONFIG_SRCDIR([main.cpp])
AC_CONFIG_HEADERS([config.h])
AM_INIT_AUTOMAKE([-Wall -Werror foreign])

# Checks for programs.
AC_PROG_CXX

# Checks for libraries.

# Checks for header files.

```

```
# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Now, run “autoreconf”:

```
$ autoreconf --install
```

Change to the BSP base directory and create a new PTXdist package (Again, you should use your email address instead of the stated one):

```
$ cd ...
$ ptxdist newpackage target
ptxdist: creating a new 'target' package:
ptxdist: enter packet name.....: myapp
ptxdist: enter version number....: trunk
ptxdist: enter URL of basedir....: file://$(PTXDIST_WORKSPACE)/local_src
ptxdist: enter suffix.....:
ptxdist: enter packet author.....: Carsten Behling <carsten.behling@garz-fricke.com>
```

With your favourite editor change the contents of the created “OSELAS.BSP-GUF-Linux-<version>/rules/myapp”.in file as follows:

```
## SECTION=fixme

config MYAPP
    tristate
    prompt "myapp"
    help
    The myapp application.
```

With your favourite editor change the contents of the created “OSELAS.BSP-GUF-Linux-<version>/rules/myapp.make” file as follows:

```
# -*-makefile-*-
#
# Copyright (C) 2010 by Carsten Behling <carsten.behling@garz-fricke.com>
#
# See CREDITS for details about who has contributed to this project.
#
# For further information about the PTXdist project and license conditions
# see the README file.
#
#
# We provide this package
#
PACKAGES-$(PTXCONF_MYAPP) += myapp

#
# Paths and names
#
MYAPP_VERSION      := trunk
MYAPP              := myapp
MYAPP_SUFFIX       :=
MYAPP_URL          := file://$(PTXDIST_WORKSPACE)/local_src/$(MYAPP)
MYAPP_SOURCE       := $(SRCDIR)/$(MYAPP)
MYAPP_DIR          := $(BUILDDIR)/$(MYAPP)
MYAPP_LICENSE      := unknown

# -----
# Get
# -----

$(MYAPP_SOURCE):
    @$(call targetinfo)
    @$(call get, MYAPP)

# -----
# Prepare
```

```

# -----
#MYAPP_CONF_ENV      := $(CROSS_ENV)

#
# autoconf
#
MYAPP_CONF_TOOL      := autoconf
#MYAPP_CONF_OPT      := $(CROSS_AUTOCONF_USR)

#$(STATEDIR)/myapp.prepare:
#   @$(call targetinfo)
#   @$(call clean, $(MYAPP_DIR)/config.cache)
#   cd $(MYAPP_DIR) && \
#       $(MYAPP_PATH) $(MYAPP_ENV) \
#       ./configure $(MYAPP_CONF_OPT)
#   @$(call touch)

# -----
# Compile
# -----

#$(STATEDIR)/myapp.compile:
#   @$(call targetinfo)
#   @$(call world/compile, MYAPP)
#   @$(call touch)

# -----
# Install
# -----

#$(STATEDIR)/myapp.install:
#   @$(call targetinfo)
#   @$(call world/install, MYAPP)
#   @$(call touch)

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

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

    @$(call install_init, myapp)
    @$(call install_fixup, myapp,PACKAGE,myapp)
    @$(call install_fixup, myapp,PRIORITY,optional)
    @$(call install_fixup, myapp,VERSION,$(MYAPP_VERSION))
    @$(call install_fixup, myapp,SECTION,base)
    @$(call install_fixup, myapp,AUTHOR,"Carsten Behling <carsten.behling@garz- ←
        fricke.com>")
    @$(call install_fixup, myapp,DEPENDS,)
    @$(call install_fixup, myapp,DESCRIPTION,missing)

    @$(call install_copy, myapp, 0, 0, 0755, $(MYAPP_DIR)/myapp, /usr/bin/myapp)
    @$(call install_alternative, myapp, 0, 0, 0755, \
        /etc/init.d/myapp)
    @$(call install_link, myapp, ../init.d/myapp, \
        /etc/rc.d/S95myapp)

    @$(call install_finish, myapp)

    @$(call touch)

# -----
# Clean
# -----

#$(STATEDIR)/myapp.clean:
#   @$(call targetinfo)
#   @$(call clean_pkg, MYAPP)

# vim: syntax=make

```

Finally, the start up behaviour of the system must be changed in that manner that the “myapp” application runs automatically after system boot instead of the Qt4 demo application. In the “Target-Install” section of the file above a symbolic link “/etc/rc.d/S95myapp” to the start up script “./init.d/myapp” is created. We have to create this start up script.

Copy the file “OSELAS.BSP-GUF-Linux-<version>/projectroot/etc/init.d/qt4-guf-demo” to “OSELAS.BSP-GUF-Linux-<version>/projectroot/etc/init.d/myapp”. Change the contents of this file with your favourite editor as follows:

```
#!/bin/sh

/etc/profile

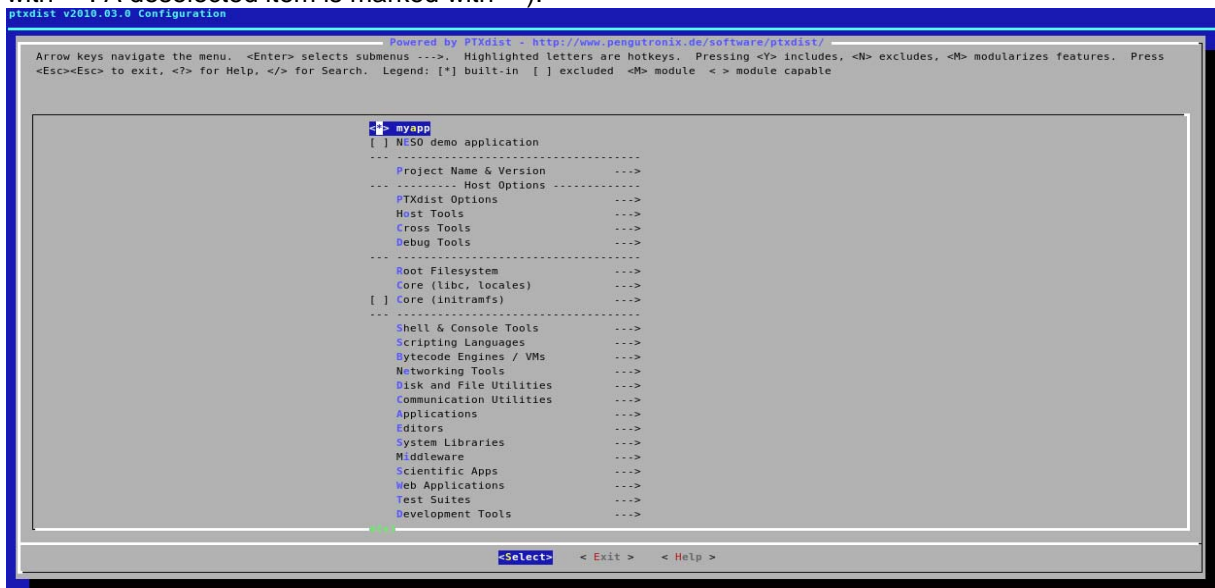
case "$1" in
start)
    start-stop-daemon -m -p /var/run/myapp.pid -b -a /usr/bin/myapp -S
    ;;
stop)
    start-stop-daemon -p /var/run/myapp.pid -K
    ;;
*)
    echo "Usage: /etc/init.d/myapp {start|stop}" >&2
    exit 1
    ;;
esac
```

Now, we are ready to build our newly created application.

First, run activate the integration of your package with PTXdist:

```
$ ptxdist menuconfig
```

Unselect the item “Garz & Fricke demo application” and select the item “myapp” (You can navigate between the items with “↑” and “↓” and select/deselect an item with “[SPACE]”). A selected item is marked with “\*”. A deselected item is marked with “ ”):



To rebuild the target system with the new application in the BSP directory “OSELAS.BSP-GUF-Linux-<version>” remove the start up links for the Garz & Fricke demo application “OSELAS.BSP-GUF-Linux-<version>/platform-<PLATFORM>/root/rc.d/S95qt4-guf-demo” and “OSELAS.BSP-GUF-Linux-<version>/platform-<PLATFORM>/root-debug/etc/rc.d/S95qt4-guf-demo” and rebuild the system with PTXdist:

```
$ rm -f platform-<PLATFORM>/root/etc/rc.S/S95qt4-guf-demo
$ rm -f platform-<PLATFORM>/root-debug/etc/rc.S/S95qt4-guf-demo
$ ptxdist go
$ ptxdist images
```

Now, you can deploy the new target system like described in chapter 5.

If you want to modify your application e.g. change the “main.cpp” file you can rebuild your application “maypp” by removing the state files for the “myapp” package and rebuild the system with PTXdist:

```
[Modify main.cpp]
$ rm -f platform-<PLATFORM>/myapp.*
$ ptxdist go
$ ptxdist images
```

Additional information of handling packages with PTXdist can be found in the PTXdist documentation from the CD / USB stick shipped with the starter kit in the Documentation folder (“OSELAS.BSP-Pengutronix-Generic-arm-Quickstart.pdf”).

### 7.3 Building GUI user applications (with Qt) outside from PTXdist

Create a directory in your home directory on the host system and change to it:

```
$ cd ~
$ mkdir qt4-myapp
$ cd qt4-myapp
```

Touch the empty files “main.cpp” and “build.sh” into this directory and make “build.sh” executable:

```
$ touch main.cpp build.sh
$ chmod a+x ./build.sh
```

With your favourite editor edit the contents of the “main.cpp” file as follows:

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    app.setOverrideCursor(Qt::BlankCursor);
    QPushButton hello("Hello World!");
    hello.setWindowFlags(Qt::FramelessWindowHint);
    hello.resize(800, 480);
    hello.show();
    return app.exec();
}
```

With your favourite editor edit the contents of the “build.sh” file as follows:

```
#!/bin/sh

TOOLCHAINDIR=/opt/OSELAS.Toolchain-<version>/arm-<arm-core>-linux-gnueabi/ ↵
gcc-<version>-glibc-<version>-binutils-<version>-kernel-<version>-sanitized
BSP_DIR=/home/behlingc/release/OSELAS.BSP-GUF-Linux-<version>
BSP_PLATFORM=<PLATFORM>
QT4_DIR=$BSP_DIR/platform-$BSP_PLATFORM/build-target/ ↵
qt-everywhere-opensource-src-4.6.2-build
CROSS_DIR=$BSP_DIR/platform-$BSP_PLATFORM/sysroot-cross

export PATH=$TOOLCHAINDIR/bin:$CROSS_DIR/bin:/bin:/usr/bin

#
# Create Makefile by running qmake
#
QMAKEPATH=/home/behlingc/release/OSELAS.BSP-GUF-Linux-<version>/ ↵
platform-<PLATFORM>/build-target/qt-everywhere-opensource-src-4.6.2 \
INSTALL_ROOT=/home/behlingc/release/OSELAS.BSP-GUF-Linux-<version>/ ↵
platform-<PLATFORM>/sysroot-target \
QMAKESPEC=/home/behlingc/release/OSELAS.BSP-GUF-Linux-<version>/ ↵
platform-<PLATFORM>/build-target/ ↵
qt-everywhere-opensource-src-4.6.2/mkspecs/qws/linux-ptx-g++ \
qmake qt4-myapp.pro

#
# Run make
#
make
```

```
#
# Strip down the binary
#
arm-<arm-core>-linux-gnueabi-strip qt4-myapp
exit 0
```

Execute “build.sh”

```
$ ./build.sh
```

in the “qt4-myapp” directory.

Now there is the “qt4-myapp” executable build in the “qt4-myapp” directory. You can transfer this application to the target systems “/usr/bin” directory in one of the ways described in chapter 6.

In order to let the target system start this application automatically instead of the demo application you have to copy the file “/etc/init.d/qt4-guf-demo” to “/etc/init.d/qt4-myapp” on the target system. (You can do this by login e.g. on a serial console with a terminal program, TELNET or SSH as described above):

```
root@ptx:~ cp /etc/init.d/qt4-guf-demo /etc/init.d/qt4-myapp
```

Edit “/etc/init.d/myapp” with the editor “nano” on the target system by typing:

```
root@ptx:~ nano /etc/init.d/qt4-myapp
```

Change the contents of this file as follows:

```
#!/bin/sh

./etc/profile

case "$1" in
start)
    start-stop-daemon -m -p /var/run/qt4-myapp.pid -b -a /usr/bin/qt4-myapp -S ↵
    -- -qws
    ;;
stop)
    start-stop-daemon -p /var/run/qt4-myapp.pid -K
    ;;
*)
    echo "Usage: /etc/init.d/qt4-myapp {start|stop}" >&2
    exit 1
    ;;
esac
```

Save the changes by pressing “STRG+O” and accept the target file name as suggested by pressing “[ENTER]”. Leave the “nano” editor by pressing “STRG+X”

Finally, the current start up link to “/etc/init.d/qt4-guf-demo” “/etc/rc.d/S95qt4-guf-demo” must be deleted and a new start up link “/etc/rc.d/S95qt4-myapp” to “/etc/init.d/qt4-myapp” must be created:

```
root@ptx:~ rm -f /etc/rc.d/S95qt4-guf-demo
root@ptx:~ ln -s /etc/init.d/myapp /etc/rc.d/S95qt4-myapp
```

After system reboot your application starts automatically and “Hello World!” is written to the display.

## 7.4 Building GUI user applications (with Qt) integrated into PTXdist

Create a directory in the “OSELAS.BSP-GUF-Linux-<version>/local\_src” on the host system and change to it:

```
$ cd local_src
$ mkdir qt4-myapp
$ cd qt4-myapp
```

Touch the empty files “main.cpp” and “qt4-myapp.pro” into this directory:

```
$ touch main.cpp qt4-myapp.pro
```

With your favourite editor edit the contents of the “main.cpp” file as follows:

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    app.setOverrideCursor(Qt::BlankCursor);
    QPushButton hello("Hello World!");
    hello.setWindowFlags(Qt::FramelessWindowHint);
    hello.resize(800, 480);
    hello.show();
    return app.exec();
}
```

With your favourite editor edit the contents of the “qt4-myapp.pro” file as follows:

```
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
SOURCES += main.cpp
```

Change to the BSP base directory “OSELAS.BSP-GUF-Linux-<version>” and create a new PTXdist package (You should use your email address instead of the stated one):

```
$ cd ...
$ ptxdist newpackage target
ptxdist: creating a new 'target' package:
ptxdist: enter packet name.....: qt4-myapp
ptxdist: enter version number....: trunk
ptxdist: enter URL of basedir....: file://$(PTXDIST_WORKSPACE)/local_src
ptxdist: enter suffix.....:
ptxdist: enter packet author.....: Carsten Behling <carsten.behling@garz-fricke.com>
```

With your favourite editor change the contents of the created “OSELAS.BSP-GUF-Linux-<version>/rules/myapp.in” file as follows:

```
## SECTION=fixme

config QT4_MYAPP
    tristate
    prompt "qt4-myapp"
    help
    The qt4-myapp application.
```

With your favourite editor change the contents of the created “OSELAS.BSP-GUF-Linux-<version>/rules/qt4-myapp.make” file as follows:

```
# -*-makefile-*-
#
# Copyright (C) 2010 by Carsten Behling <carsten.behling@garz-fricke.com>
#
# See CREDITS for details about who has contributed to this project.
#
# For further information about the PTXdist project and license conditions
# see the README file.
#
#
# We provide this package
#
PACKAGES-$(PTXCONF_QT4_MYAPP) += qt4-myapp

#
# Paths and names
#
QT4_MYAPP_VERSION := trunk
QT4_MYAPP         := qt4-myapp
QT4_MYAPP_SUFFIX :=
QT4_MYAPP_URL     := file://$(PTXDIST_WORKSPACE)/local_src/$(QT4_MYAPP)
```

```

QT4_MYAPP_SOURCE := $(SRCDIR)/$(QT4_MYAPP)
QT4_MYAPP_DIR    := $(BUILDDIR)/$(QT4_MYAPP)
QT4_MYAPP_LICENSE := unknown

# -----
# Get
# -----

$(QT4_MYAPP_SOURCE):
    @$(call targetinfo)
    @$(call get, QT4_MYAPP)

# -----
# Prepare
# -----

QT4_MYAPP_PATH := PATH=$(CROSS_PATH)

QT4_MYAPP_ENV = \
    $(CROSS_ENV) \
    QMAKEPATH=$(QT4_DIR) \
    INSTALL_ROOT=$(SYSROOT) \
    QMAKESPEC=$(QT4_DIR)/mkspecs/qws/linux-ptx-g++

$(STATEDIR)/qt4-myapp.prepare:
    @$(call targetinfo)
    cd $(QT4_MYAPP_DIR) && \
        $(QT4_MYAPP_PATH) $(QT4_MYAPP_ENV) qmake qt4-myapp.pro
    @$(call touch)

# -----
# Compile
# -----

$(STATEDIR)/qt4-myapp.compile:
    @$(call targetinfo)
    cd $(QT4_MYAPP_DIR) && $(QT4_MYAPP_PATH) $(MAKE) $(PARALLELMFLAGS)
    @$(call touch)

# -----
# Install
# -----

#$(STATEDIR)/qt4-myapp.install:
#    @$(call targetinfo)
#    @$(call world/install, QT4_MYAPP)
#    @$(call touch)

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

$(STATEDIR)/qt4-myapp.targetinstall:
    @$(call targetinfo)

    @$(call install_init, qt4-myapp)
    @$(call install_fixup, qt4-myapp, PACKAGE, qt4-myapp)
    @$(call install_fixup, qt4-myapp, PRIORITY, optional)
    @$(call install_fixup, qt4-myapp, VERSION, $(QT4_MYAPP_VERSION))
    @$(call install_fixup, qt4-myapp, SECTION, base)
    @$(call install_fixup, qt4-myapp, AUTHOR, "Carsten Behling ↵
    <carsten.behling@garz-fricke.com>")
    @$(call install_fixup, qt4-myapp, DEPENDS, )
    @$(call install_fixup, qt4-myapp, DESCRIPTION, missing)

    @$(call install_copy, qt4-myapp, 0, 0, 0755, $(QT4_MYAPP_DIR)/qt4-myapp, ↵
    /usr/bin/qt4-myapp)
    @$(call install_alternative, qt4-myapp, 0, 0, 0755, \
    /etc/init.d/qt4-myapp)
    @$(call install_link, qt4-myapp, ../init.d/qt4-myapp, \
    /etc/rc.d/S95qt4-myapp)

```

```

    @$(call install_finish, qt4-myapp)

    @$(call touch)

# -----
# Clean
# -----

#$(STATEDIR)/qt4-myapp.clean:
#   @$(call targetinfo)
#   @$(call clean_pkg, QT4_MYAPP)

# vim: syntax=make

```

Finally, the start up behaviour of the system the system must be changed in that manner that the “qt4-myapp” application runs automatically after system boot instead of the Garz & Fricke demo application. In the “Target-Install” section of the file above a symbolic link “/etc/rc.d/S95qt-4myapp” to the start up script “./init.d/qt4-myapp” is created. We have to create this start up script.

In the BSP directory “OSELAS.BSP-GUF-Linux-<version>” copy the file “OSELAS.BSP-GUF-Linux-<version>/projectroot/etc/init.d/qt4-guf-demo” to “OSELAS.BSP-GUF-Linux-<version>/projectroot/etc/init.d/qt-4myapp”. Change the contents of this file with your favourite editor as follows:

```

#!/bin/sh

. /etc/profile

case "$1" in
start)
    start-stop-daemon -m -p /var/run/qt4-myapp.pid -b -a /usr/bin/qt4-myapp ↵
    -S -- -qws
    ;;
stop)
    start-stop-daemon -p /var/run/qt4-myapp.pid -K
    ;;
*)
    echo "Usage: /etc/init.d/qt4-myapp {start|stop}" >&2
    exit 1
    ;;
Esac

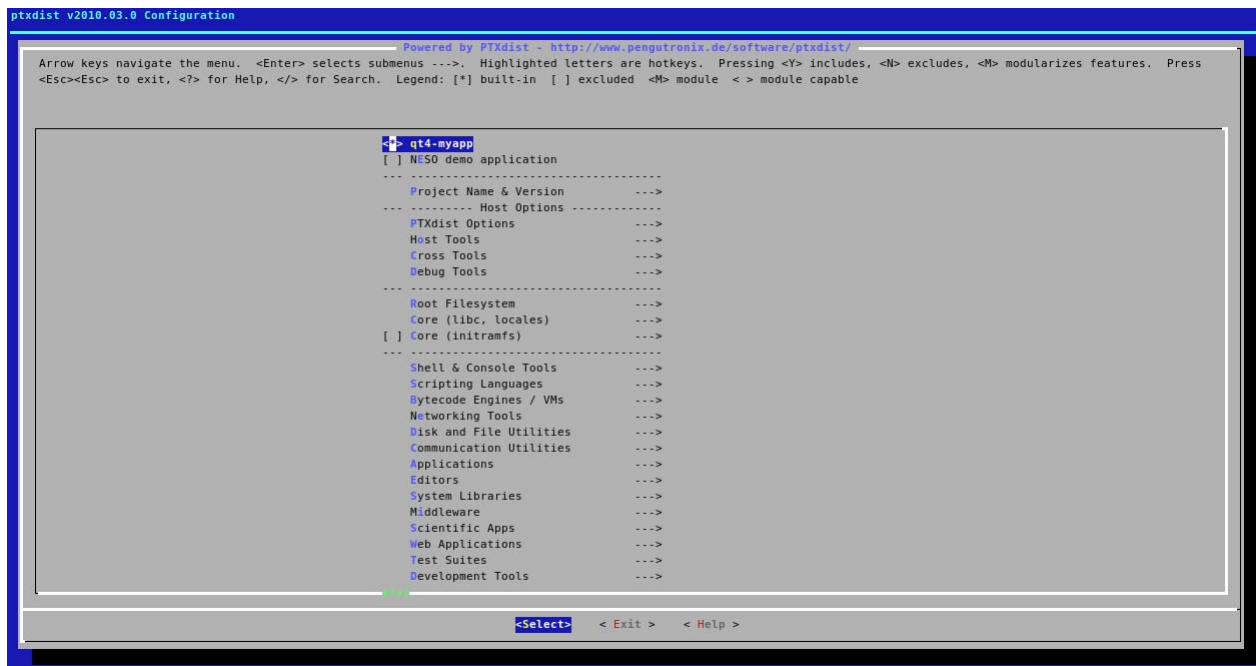
```

Now, we are ready to build our newly created application.

First, run activate the integration of your package with PTXdist:

```
$ ptxdist menuconfig
```

Unselect the item “Garz & Fricke demo application” and select the item “myapp” (You can navigate between the items with “↑” and “↓” and select/deselect an item with “[SPACE]”. A selected item is marked with “\*”. A deselected item is marked with “ ”):



To rebuild the target system with the new application in the BSP directory remove the start up links for the Garz & Fricke demo application “OSELAS.BSP-GUF-Linux-<version>/platform-<PLATFORM>/root/rc.d/S95qt4-guf-demo” and “OSELAS.BSP-GUF-Linux-<version>/platform-<PLATFORM>/root-debug/etc/rc.d/S95qt4-guf-demo” and rebuild the system with PTXdist:

```
$ rm -f platform-<PLATFORM>/root/etc/rc.S/S95qt4-<PLATFORM>-demo
$ rm -f platform-<PLATFORM>/root-debug/etc/rc.S/S95qt4-<PLATFORM>-demo
$ ptxdist go
$ ptxdist images
```

Now, you can deploy the new target system like described in chapter 5.

After system reboot your application starts automatically and “Hello World!” is written to the display.

If you want to modify your application e.g. modify “main.cpp” you can rebuild your application “maypp” by removing the state files for the “myapp” package and rebuild the system with PTXdist:

```
[Modify main.cpp]
$ rm -f platform-<PLATFORM>/myapp.*
$ ptxdist go
$ ptxdist images
```

Additional information of handling packages with PTXdist can be found in the PTXdist documentation from the CD / USB stick shipped with the starter kit in the Documentation folder (“OSELAS.BSP-Pengutronix-Generic-arm-Quickstart.pdf”).

## Annex A: Document History

Release/Date	Title	Description
Preliminary, 31.03.2010	Initial document release	