

Linux Guide

Linux Guide

Copyright © 2011 taskit GmbH

All rights to this documentation and to the product(s) described herein are reserved by taskit GmbH.

This document was written with care, but errors cannot be excluded. Neither the company named above nor the seller assumes legal liability for mistakes, resulting operational errors or the consequences thereof. Trademarks, company names and product names may be protected by law. This document may not be reproduced, edited, copied or distributed in part or in whole without written permission.

This document was generated on 2012-01-06T15:36:58+01:00.

Table of Contents

1. Introduction	1
2. Supported products	2
3. Configuring the system	3
3.1. Using the package manager	3
3.1.1. Updating the package database	3
3.1.2. Upgrade installed software	3
3.1.3. Installing new software	3
3.1.4. Removing installed packages	3
3.1.5. Listing available packages	4
3.1.6. Search for a package in the package list	4
3.1.7. Listing installed packages	4
3.2. Configuring network interfaces	5
3.2.1. Automatic configuration (DHCP)	5
3.2.2. Static configuration	5
4. Setting up a development system	6
4.1. Installing the toolchain	6
4.2. Setting up an NFS Server	7
4.3. Setting up a TFTP server	7
5. Compiling and debugging applications	9
5.1. Compiling the application sample	9
5.1.1. Stamp9G20/PortuxG20	9
5.1.2. Stamp9G45	9
5.2. Starting the sample	9
5.3. Debugging the sample	10
5.4. Downsizing the binary	10
6. Compiling a new Linux kernel	11
6.1. Unpacking the kernel sources	11
6.2. Configuring the kernel	11
6.3. Compiling the kernel	12
6.4. Preparing the kernel image	13
6.5. Installing the kernel	13
7. Creating a new root filesystem	14
7.1. Setting up the OpenEmbedded build system	14
7.1.1. Prerequisites	14
7.1.2. Getting the needed metadata	14
7.1.3. Configuring bitbake	15
7.1.4. Setting up the environment	17
7.1.5. Keeping the metadata up to date	18
7.2. Building and installing images	19
7.2.1. Building images	19
7.2.2. Installing images	20
7.3. Customizing images	22
7.3.1. Creating your own machine	23
7.3.2. Creating your own image	26
7.3.3. Customizing existing packages	28
7.3.4. Adding own packages	29
8. Using and configuring the bootloader	31

8.1. Concepts	31
8.1.1. Components	31
8.1.2. Boot procedure	31
8.2. Reference	32
8.2.1. Environment variables	32

List of Figures

6.1. Kernel configuration dialog	12
8.1. Boots flow-chart	32

List of Tables

3.1. Elements of /etc/network/interfaces 5

List of Examples

3.1. DHCP network configuration	5
3.2. Static network configuration with all options	5
7.1. Bitbake local configuration	16
7.2. Bitbake layer configuration	17
7.3.	17
7.4. Custom machine definition before modification	24
7.5. Custom machine definition after modification	25
7.6. Customized kernel recipe	26
7.7. custom-image.bb: Extending taskit-image	26
7.8. taskit-image.bb	27
7.9. custom-image.bb: Complete new image	28
7.10. Makefile	29
7.11. app_1.0.bb	30

1. Introduction

Your product is delivered with a customized Linux and the boot loader U-Boot. This document will describe how to install and customize the operation system.

Furthermore it will describe how to setup a development system and you will be given small examples that demonstrate how to compile your own applications.

Because of the wide variety of existing operating systems taskit can only give support for the **Debian GNU/Linux**[™] operating system. Taskit is utilising the Linux-based operating system **Debian**[™] (<http://www.debian.org>) as development system because it is one of the most reliable operating systems. Furthermore it is easy to install additional software on Debian because you only need the tool **apt-get** to automatically download software packages that are installed and configured automatically. Debian can be downloaded free from the internet and the installation is also very easy because you only need to download a portion (<http://www.debian.org/distrib/netinst>) and the remaining parts will be automatically downloaded and installed from the internet.

A cross-platform toolchain for cross compiling on **Debian**[™] can be found on the Starterkit-CD. Developing on **MS Windows**[™] is not supported by taskit.

Instructions for the first start-up are located in the *QuickStartGuide*. If you want to develop your own drivers or hardware extensions you will have to work with the appropriate *Technical Reference* and *Atmel manual* for your product.

The newest revision of this document can always be found on <http://armbedded.eu/documentation>.

2. Supported products

The specifications in this document apply to the following products:

- Stamp9G20
- Stamp9G45
- PortuxG20

3. Configuring the system

This chapter describes, how you can configure the running system. Be aware that if you are doing a mass deployment, it might make more sense to create complete images instead of installing standard images and configuring them afterwards. Please consult Chapter 7, *Creating a new root filesystem* if you want to learn more about creating images.

3.1. Using the package manager

The products covered by this manual use a package manager called **opkg**. It enables you to install additional software without the need to (cross) compile it yourself. This section contains information on common tasks when using **opkg**.

3.1.1. Updating the package database

Before you can install additional packages or update them you have to get the current list of available packages:

```
opkg update
```

You can repeat this command as often as you want to ensure, that the package database is always up to date. This way, the package manager always knows if there are updates or new packages.

3.1.2. Upgrade installed software

From time to time, there might be updates to some packages, mostly because of bug fixes. These updates can easily be installed after updating the packages database:

```
opkg upgrade
```

Do not worry about the download error at the end. It is there because the Ångström distribution's website does not host packages specific to taskit boards. If the error bothers you, you can remove the opkg config file specific to your board, e.g. `/etc/opkg/stamp9g20evb-feed.conf`.

3.1.3. Installing new software

You can install software, that is currently not installed with the following command:

```
opkg install package
```

3.1.4. Removing installed packages

If you do not need a package anymore and want to get rid of it, you can remove it with opkg:

```
opkg remove package
```

Configuring the system

When installing a package, `opkg` might also have installed additional packages, e.g. libraries. You can instruct `opkg` to remove these software packages automatically when removing packages:

```
opkg -autoremove remove package
```

There are cases where you might be trying to remove packages that are needed by other packages. If this happens `opkg` will list all packages that depend on the package to be removed. You now have three choices:

1. Leave the package in the system

This might be your only choice, if you need the depending packages.

2. Remove the package and all packages depending on this package

If you do not need the other packages, you can let `opkg` remove them, too:

```
opkg -recursive remove package
```

You can also use the option `-autoremove` here.

3. Remove only the package

This is not advised because it is very likely that the dependent packages are broken afterwards, but if you really want to do that, you can use the following command:

```
opkg -force-depends remove package
```

3.1.5. Listing available packages

Before installing a package you certainly want to know, which packages are available. Be aware, that the following command can produce a very long list:

```
opkg list
```

3.1.6. Search for a package in the package list

As `opkg` has no native ability to search in the package list you have to use tools like **grep** to search the package list, e.g.

```
opkg list|grep mysql
```

to find all packages containing `mysql` in their package name oder description.

3.1.7. Listing installed packages

If you want to know, which packages are currently installed, run

```
opkg list_installed
```

3.2. Configuring network interfaces

To configure network interfaces persistently, you have to edit the file `/etc/network/interfaces`. Network interfaces configured in this file can be brought up and down with the tools **ifup** and **ifdown**.

element	description
auto <i>interface</i>	<i>interface</i> should be brought up when using ifup -a , i.e. when booting the system
iface <i>interface</i> ...	configure <i>interface</i>
lines starting with #	ignored as comments

Table 3.1. Elements of `/etc/network/interfaces`

The full syntax of the `iface` directive is:

```
iface interface address_family method
  option1 value1
  option2 value2
  ...
```

3.2.1. Automatic configuration (DHCP)

To use DHCP on `eth0`, you would enter the following into the configuration file.

```
auto eth0
iface eth0 inet dhcp
```

Example 3.1. DHCP network configuration

As this is the default, you normally do not have to do this.

3.2.2. Static configuration

In Example 3.2, “Static network configuration with all options” you can see a static configuration for `eth0`. The IP address `192.168.1.100` is assigned and `192.168.1.1` is used as the default gateway. The options `network` and `broadcast` are optional.

```
iface eth0 inet static
  address 192.168.1.100
  netmask 255.255.255.0
  gateway 192.168.1.1
  network 192.168.1.0
  broadcast 192.168.1.255
```

Example 3.2. Static network configuration with all options (network and broadcast are optional)

4. Setting up a development system

The development system described here assumes that your device is connected to a separate development computer, using either Ethernet or a serial cable. All transfers between the two systems occur exclusively over this connection. The development system does not have any particular hardware demands; a standard PC is in most cases sufficient. A Linux workstation is normally used as a development computer for an embedded Linux device. A network card and serial interface are required for the connection.

As a basis for such a host system, taskit recommends and supports the freely available Debian Linux distribution for development. Debian stands out for its stability and good packet management. Several ways to acquire Debian are described at <http://www.debian.org/distrib/>. For complete installation instructions for the i386 architecture, see <http://www.debian.org/releases/stable/i386/install>. The following descriptions relate to such a Debian system.

You could also run a Linux system in a virtual environment using a virtual machine such as VMWare, VirtualPC or VirtualBox. This solution, however, severely limits performance and usability.

The following sections assume that you do all your development in the path `/develop`. You can of course use a different path, but you have to adjust all the paths accordingly.

To create it just issue the following command with root rights:

```
mkdir /develop
```

Now change the permissions to be able to use it with your normal user account, e.g.

```
chown developer /develop
```

if your user account is called “developer”.

If you want to share this directory with other users, change the group of it:

```
chgrp users /develop  
chmod 775 /develop
```

This allows all users in the group “users” (which are all users on Debian by default) to access the directory with read and write permissions.

4.1. Installing the toolchain

A toolchain for cross compiling is the most important element of the development system. Precompiled binaries for the i386 architecture are on the Starterkit CD. You can find it in the `toolchain` directory, e.g. `angstrom-2009.X-stable-armv5te-linux-gnueabi-toolchain.tar.bz2`. To install it, mount the CD and enter the following commands:

```
tar -xvjf angstrom-2009.X-stable-armv5te-linux-gnueabi-toolchain.tar.bz2 -C /
```

Setting up a development system

For the Stamp9G45, the toolchain is available for 32-bit and 64-bit x86 systems. Choose the corresponding file from the Stamp9G45 toolchain folder and extract it like in the previous example.

The toolchain will be installed into the directory `/usr/local/angstrom` or similar depending on the product and architecture.

The compilation of a toolchain itself is labour intensive and will not be described here. The toolchain was made with OpenEmbedded (using the Ångström distribution), which simplifies the compilation considerably. For more information on OpenEmbedded consult Chapter 7, *Creating a new root filesystem*.

Additionally, further tools might be needed (e.g. **make**). To get the basic tools install the package **build-essential**.

```
apt-get install build-essential
```

4.2. Setting up an NFS Server

After installing the toolchain, you can compile your own software for the arm processor. In the early stages of development, it is convenient to mount the working directory on the development system with NFS (network file system), in order to make changes available quickly.

Installation of the NFS-Server:

```
apt-get install nfs-kernel-server
```

If an NFS server is already set up on the development system, you only need to add one line to the `/etc/exports` file:

```
/develop 192.168.1.*(ro)
```

This gives every host with an IP in the range from 192.168.1.1 to 192.168.1.254 read-only access to the directory `/develop`. For further options consult the `exports(5)` manual page. Adjust this line according to your network setup and preferences.

The exported directory can then be mounted to a directory on the target board with the `mount` command:

```
mkdir /mnt/develop  
mount -t nfs -o nolock,tcp servername:/develop /mnt/develop
```

4.3. Setting up a TFTP server

To transfer customized firmware to the target board in the boot loader (U-Boot), TFTP (Trivial File Transfer Protocol) is needed. Additionally, it is also possible to transfer files while running the Linux system on the target board via TFTP. For this purpose a corresponding TFTP server must be set up on the development system.

Use `apt-get` to install the required `tftpd` demon under Debian:

Setting up a development system

```
apt-get install tftpd
```

Usually tftpd is not started directly, but rather via the inetd Internet demon. An entry for TFTP must be added in the inetd configuration file after installation. Under Debian, the following line is automatically added to the configuration file `/etc/inetd.conf` during packet installation:

```
tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd /tftpboot
```

As you can see, `/tftpboot` is the default directory for TFTP. If it does not exist you have to create. Follow the same steps as for the `/develop` creation.

5. Compiling and debugging applications

If everything is setup according to Chapter 4, *Setting up a development system*, you can start to develop programs.

5.1. Compiling the application sample

In the `/examples` directory on the Starterkit CD you will find the `example1.c` file, which contains C source code for a simple program for entering and printing text. For editing, first copy the file to the `/develop` directory on the development computer.

5.1.1. Stamp9G20/PortuxG20

Before running any compiler command, you have to source the environment setup in your shell:

```
. /usr/local/angstrom/arm/environment-setup
```

You can now compile the sample with the following command:

```
arm-angstrom-linux-gnueabi-gcc example1.c -o example1
```

5.1.2. Stamp9G45

Before running any compiler command, you have to source the environment setup in your shell:

```
. /usr/local/angstrom-eglibc-i686-armv5te/environment-setup-armv5te-angstrom-linux-gnu
```

or

```
. /usr/local/angstrom-eglibc-x86_64-armv5te/environment-setup-armv5te-angstrom-linux-g
```

on a 64-bit system.

You can now compile the sample with the following command:

```
arm-angstrom-linux-gnueabi-gcc $CFLAGS $LDFLAGS example1.c -o example1
```

The following works also, as the `CC` environment variable is set:

```
$CC $CFLAGS $LDFLAGS example1.c -o example1
```

The usage of the `CFLAGS` and `LDFLAGS` variables is essential, as without them, the compiler will not find all header files or libraries.

5.2. Starting the sample

If the execution rights for the newly created binary are set correctly, the program can now be started on the target board:

```
cd /mnt/develop
./example1
```

5.3. Debugging the sample

The GNU debugger (GDB) is one of the most important debugging tools for Linux. The `gdbserver` itself is a small application that carries out commands from the `gdb`, which runs on the development system. You will find the **`gdbserver`** in the Linux Starterkit's root file system, in the `/usr/bin` directory.

Before debugging a program, you must compile it with the appropriate flags (`-g` or `-ggdb` for more information).

```
arm-angstrom-linux-gnueabi-gcc -g example1.c -o example1_debug
```

If you include in debugging information, the binary created is much larger. As long as you have the original version with the debugging information on the development system, however, you can simply copy the smaller, stripped-down version to the target system. You can strip down the debugger using the **`arm-angstrom-linux-gnueabi-strip`** tool.

For remote debugging, you can set up communication between the `gdbserver` on the taskit device and the `gdb` on the development system either over a serial null modem cable or over a TCP/IP connection. The connection via TCP/IP is described below. First you need to start the `gdbserver` on the taskit device, and then create the connection from the `gdb` on the development computer:

```
gdbserver host:port example1_debug
```

Host should be replaced with the host, where **`gdb`** will be started, but it is currently ignored by **`gdbserver`**. As *port*, choose any available port. All command line parameters for the program (if you later need some) must be given in this call. Then you can start the `gdb` on the other system and create the connection to your taskit device:

```
arm-angstrom-linux-gnueabi-gdb example1_debug
(GDB) target remote remote_ip:port
```

Replace *remote_ip* with the ip of the target board and *port* with the port given to **`gdbserver`**. Now you are ready to start debugging with the usual `gdb` commands.

5.4. Downsizing the binary

After compiling the example the file size of the binary can be notably reduced by removing unneeded informations generated by the compiler as well as debug informations

```
arm-angstrom-linux-gnueabi-strip example1
```

6. Compiling a new Linux kernel

If you work with Embedded Linux regularly, you will often face the need to create your own kernel. In most cases, this involves integrating new drivers, e.g. for USB devices, or additional file systems. Because memory space is limited on an embedded board, it does not make sense to set up a large number of drivers to start with (as is common for desktop PCs) unless you know for sure that you actually need them.

The kernel binaries and sources delivered with the product are made up of a standard kernel with patches or drivers from taskit. The process for creating your own kernel is broken down into three steps: configuring, compiling and installing.

6.1. Unpacking the kernel sources

Before you can configure and compile the kernel, you need the kernel source code. It can be found on the Starterkit-CD in gzip- oder bzip2-compressed tar archive (tarball), e.g. `linux-2.6.29-stamp9g20.tar.bz2`. It may also be possible, that there are updated versions available for download, see <http://armbedded.eu/downloads>. Then the archive has to be extracted to your development directory, e.g.:

```
tar -xvjf path/linux-2.6.29-stamp9g20.tar.bz2
```

Replace *path* with the path, where the Linux tarball can be found.

If you decompress a gzipped archive, replace **-xvjf** with **-xvzf**. Bzip2 tarballs commonly have an extension of `.tar.bz2` or `.tbz2`. Gzip tarballs use `.tar.gz` or `.tgz`.

6.2. Configuring the kernel

To configure the kernel, enter the just created directory. Because configuring the whole kernel from scratch, taskit provides a default configuration for each product. To use the it, enter the following command:

```
make ARCH=arm product_defconfig
```

Replace *product* with the name of the product, e.g. `stamp9g20evb`.

You can now change the kernel configuration. There are multiple interfaces available for this purpose. They are provided via the **make** targets `config`, `menuconfig`, `xconfig` and `gconfig`. We will use `menuconfig`.

In order to use `menuconfig`, curses headers have to be installed on your system. If they are not at the moment, issue the following command to get them:

```
apt-get install ncurses-dev
```

You can now configure the kernel:

```
make ARCH=arm menuconfig
```

Compiling a new Linux kernel

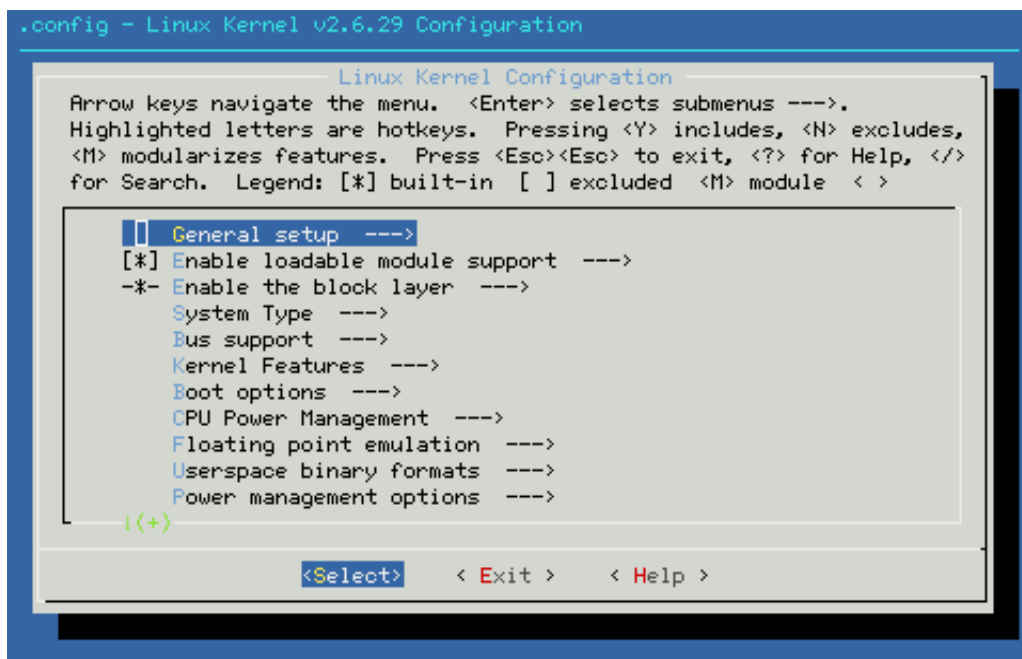


Figure 6.1. Kernel configuration dialog

Figure 6.1, “Kernel configuration dialog” shows a screenshot of the menuconfig utility. You can now enable and disable the options as you like. Each option can have up to three states: “ ”, “M” and “*”. “ ” denotes disabled options, “M” means, they are compiled as a module and can later be loaded with the help of **insmod** or **modprobe**, and “*” selects an option to be built into the kernel image. If the option belongs to a driver built as a module, “*” means, that this options is built into the module, not the kernel image.

When you are finished with configuring the kernel, exit and save the new configuration. You are now ready to compile the kernel.

6.3. Compiling the kernel

Compiling is simple:

```
make ARCH=arm CROSS_COMPILE=arm-angstrom-linux-gnueabi-
```

This builds the kernel and all selected kernel modules. If you want to build the kernel and the modules in two steps, use the following to commands:

```
make ARCH=arm CROSS_COMPILE=arm-angstrom-linux-gnueabi- zImage
make ARCH=arm CROSS_COMPILE=arm-angstrom-linux-gnueabi- modules
```

The kernel make file provides a target for installing the modules: `modules_install`. By default, the modules are installed in `/lib/modules`. For cross-environment development, the modules must be installed in a different directory. We will install it into `/develop/modules`. When entering the path of the module directory, ensure that no relative paths are given; since the script traverses the kernel directories, relative paths can change.

```
make ARCH=arm INSTALL_MOD_PATH=/develop/modules modules_install
```

6.4. Preparing the kernel image

To be able to start the kernel with U-Boot, the image has to be wrapped in an uImage. It adds a header containing important information for U-Boot. To create the uImage you need the **mkimage** tool. You can find it in the `/scripts` directory on the Starterkit-CD. Enter the following command to create the image:

```
mkimage -A arm -T kernel -O linux -C none -a 21000000 -e 21000000 -n linux \
-d arch/arm/boot/zImage uImage
```

6.5. Installing the kernel

Finally the image has to be programmed into the flash memory. We will do the procedure in linux.

First, we have to identify the flash partition, where the image has to be programmed to. You can get a list of all partitions with the following command:

```
cat /proc/mtd
```

You should get something like this:

```
dev:    size  erasesize  name
mtd0:  00020000  00020000  "bootstrap"
mtd1:  00040000  00020000  "uboot"
mtd2:  00020000  00020000  "env1"
mtd3:  00020000  00020000  "env2"
mtd4:  00200000  00020000  "linux"
mtd5:  1fd60000  00020000  "root"
```

The image must be written to the partition with the name "linux", in this case `mtd4`. We will use this name in the following descriptions, replace all occurrences of `mtd4` in the next steps with right one for your system.

Now you have to erase the flash partition:

```
flash_eraseall /dev/mtd4
```

Now you can write the image to the flash. It is assumed, that your development directory is mounted on the target board and you changed your current working directory to the kernel source tree.

```
flashcp -v uImage /dev/mtd4
```

If your board uses NAND flash, use these commands:

```
flash_eraseall /dev/mtd4
nandwrite -p /dev/mtd4 uImage
```

7. Creating a new root filesystem

The root file system is the place where system applications and libraries are stored. Most probably you want to change certain files or add additional software. If these changes are minimal and only for a small number of devices, you can make these by hand on the device. Changes will be preserved between reboots. Only certain directories (like /tmp) will be wiped after each boot.

If you want to make bigger changes and deploy them on a lot of devices, it is advised to create a new image, that can be flashed to all targets. The root file system provided by taskit was made with OpenEmbedded (using the Ångström distribution). OpenEmbedded is a build system using so-called recipes as build instructions used by the build tool **bitbake**. This chapter will explain, how to set up your system to use OpenEmbedded to create your root file system.

You may also consult the OpenEmbedded wiki for additional information: http://wiki.openembedded.net/index.php/Main_Page

It is assumed, that all OpenEmbedded related work is done in the directory /develop/oe. Adjust all paths, if you use a different directory.

Furthermore, all instructions are done exemplary for the **Stamp9G20™** evaluation board. If you use a different machine, replace all occurrences of the machine name with the correct one.

7.1. Setting up the OpenEmbedded build system

7.1.1. Prerequisites

Before you can start to use the OpenEmbedded build system, you need to install some software first. On Debian everything should be installed after the following command entered as root:

```
apt-get install ccache sed wget cvs subversion git-core bzip2 \
  coreutils unzip texi2html texinfo libstdc++2.9-dev docbook-utils \
  gawk python-pysqlite2 diffstat help2man make gcc build-essential
```

If some packages are missing, **bitbake** will complain and tell you what is missing. Just install the packages.

You can find additional information on http://wiki.openembedded.net/index.php/Required_software and <http://wiki.openembedded.net/index.php/OEandYourDistro>.

7.1.2. Getting the needed metadata

First we create our working directory and enter it:

```
mkdir /develop/oe
cd /develop/oe
```

7.1.2.1. Stamp9G20/PortuxG20

Now we get the OpenEmbedded metadata. It is stored in a **git** revision control repository:

```
git clone git://git.openembedded.org/openembedded
```

This will take a while. After the command is finished, you will have a new directory called `openembedded`. It will contain all the OpenEmbedded metadata for the development branch (`org.openembedded.dev`). As we want a more stable environment, we checkout the stable branch:

```
cd openembedded
git checkout origin/stable/2009 -b stable/2009
cd ..
```

The stable branch also contains the build tool **bitbake**, which the development branch does not. If you later decide to try out the development branch, you will have the additional install step of obtaining **bitbake** which will not be discussed here. Refer to the OpenEmbedded wiki.

Now you should get the taskit overlay for OpenEmbedded. Overlays contain additional metadata, in this case metadata specific to taskit products.

```
git clone git://gitorious.org/taskit/taskit-overlay.git
```

7.1.2.2. Stamp9G45

For the Stamp9G45 OpenEmbedded Core was used instead of the classic OpenEmbedded, so the setup is a bit different. OpenEmbedded Core uses a much more layered approach regarding the metadata. Therefore we need to clone multiple repositories:

```
git clone git://git.openembedded.org/openembedded-core
git clone git://git.openembedded.org/meta-openembedded
git clone git://git.angstrom-distribution.org/meta-angstrom
```

Additionally clone the taskit overlay for OpenEmbedded Core:

```
git clone git://gitorious.org/taskit/taskit-overlay-oe-core.git
```

Finally, you need a current copy of bitbake in the `openembedded-core` directory. Use either a snapshot (<http://git.openembedded.org/bitbake/>) or clone the bitbake repository:

```
git clone git://git.openembedded.org/bitbake
```

You can directly do this inside the `openembedded-core` directory or just create a symlink there.

7.1.3. Configuring bitbake

7.1.3.1. Stamp9G20/PortuxG20

Now it is time to create the configuration. The configuration will be put into the file `build/conf/local.conf`.

```
mkdir -p build/conf
vi build/conf/local.conf
```

You can use any other editor, if you are not comfortable with **vi**.

The file contents should look like in Example 7.1, “Bitbake local configuration”

```
DL_DIR = "/develop/oe/sources" ❶
TMPDIR = /develop/oe/tmp ❷
BBFILES = " \ ❸
    /develop/oe/taskit-overlay/recipes/*/*.bb \
    /develop/oe/openembedded/recipes/*/*.bb \
"
MACHINE ?= "stamp9g20evb" ❹
DISTRO = "angstrom-2008.1" ❺
ENABLE_BINARY_LOCALE_GENERATION = "1" ❻
GLIBC_GENERATE_LOCALES = "en_GB.UTF-8 de_DE.UTF-8 fr_FR.UTF-8" ❼
IMAGE_FSTYPES = "jffs2 tar" ❽
```

Example 7.1. Bitbake local configuration

- ❶ Source packages are downloaded to this directory
- ❷ All compilation and packaging will be done in this directory. It will also contain a subdirectory called **deploy**, where packages and images will be put.
- ❸ This variable tells **bitbake** where packages recipes can be found. In this case we told it to look into the OpenEmbedded repository and the taskit overlay.
- ❹ Define the machine to build. It determines which kernel will be built and can select machine specific package overrides. This variable can be overridden on the console, because it is defined with “?=”.
- ❺ Define the distribution to build. We use Ångström, as it is most tested distribution in the OpenEmbedded repository.
- ❻ This enables the binary locale generation. It speeds up the first boot of the target board, because the locales are built on the development machine instead of the target.
- ❼ Select only certain locales to speed up the build process. Add needed locales or leave this variable out to build all locales, although this will increase the build time considerably.
- ❽ We create jffs2-images and tar-archives. See http://docs.openembedded.org/usermanual/usermanual.html#image_types for all available image types. As you can see, you can also specify multiple images types to create them all at once.

The jffs2-images will be used to deploy images in the flash memory. The tar-archives can be used to write the rootfs to SD cards.

7.1.3.2. Stamp9G45

The basic configuration is automatically done by a script provided in the **openembedded-core** directory called **oe-init-build-env**. If you source it with no parameters, it will create and enter the directory called **build** in the current directory. This will be used for the subsequent builds and contains the configuration files in the subdirectory **conf**.

The command would look like

```
. openembedded-core/oe-init-build-env
```

or

```
source openembedded-core/oe-init-build-env
```

You can also append another name to the command if you would like to use a different build directory.

Two configuration files will be created by default, `bblayers.conf` and `local.conf`. The first one will by default look like in Example 7.2, “Bitbake layer configuration”

```
# LAYER_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
LCONF_VERSION = "4"

BBFILES ?= ""
BBLAYERS = " \
  /develop/oe/openembedded-core/meta \
"
```

Example 7.2. Bitbake layer configuration

This has to be extended to look like in Example 7.3, “”.

```
# LAYER_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
LCONF_VERSION = "4"

BBFILES ?= ""
BBLAYERS = " \
  /develop/oe/openembedded-core/meta \
  /develop/oe/meta-openembedded/meta-oe \
  /develop/oe/meta-openembedded/meta-gnome \
  /develop/oe/meta-openembedded/meta-efl \
  /develop/oe/meta-angstrom \
  /develop/oe/taskit-overlay-oe-core \
"
```

Example 7.3.

The `local.conf` is basically fine as it is, but adjusting the `MACHINE` variable should make it more convenient to invoke bitbake later on (set it to "stamp9g45"). Additionally, the following line should be added, to use the Ångström distribution for building:

```
DISTRO = "angstrom-2010.x"
```

Read the comments in the file as some of the options can improve the build performance on multicore systems.

7.1.4. Setting up the environment



Note

This section only applies to Stamp9G20 and PortuxG20. For the Stamp9G45, the script from the previous section also sets up the environment, so just source it again before you use bitbake in a new shell.

Creating a new root filesystem

The last thing to do to get a working OpenEmbedded build environment is to set some environment variables.

```
export BBPATH=/develop/oe/build:/develop/oe/taskit-overlay:\
/develop/oe/openembedded
export PATH=/develop/oe/openembedded/bitbake/bin:$PATH
```

The first line sets the variables BBPATH. It is used by bitbake to search for configuration files. Here, `build` contains your `local.conf`, and `taskit-overlay` and `openembedded` contain distribution and machine configurations.

The second line adjusts the PATH variable so that you do not have to type the full path every time you want to start bitbake.



Note

This value only works with the stable branch, if you want to use the development branch you have to obtain bitbake separately and adjust the path accordingly.



Tip

It is advised to put both lines into a file, e.g. `sourceme`, so that you do not have to type them every time you want to build something. Now you can enter the following line before you start bitbake the first time in your shell session:

```
. sourceme
```

7.1.5. Keeping the metadata up to date

The metadata from the OpenEmbedded repository can change each day, so can the taskit overlay. These changes can be bug fixes or feature additions.

To update all the metadata, just enter the corresponding directory (`/develop/oe/taskit-overlay` and `/develop/oe/openembedded`) and use the following command:

```
git pull
```

It is advised to remember the current revision before updating in case the update includes some bugs, so that you can go back easily. Just enter

```
git show
```

and you see the last change in the current branch. The first line includes the commit hash.

If you notice that builds do not work anymore after updating or the result has some errors use

```
git checkout hash
```

to go back to the commit (*hash*) remembered. In this state, you cannot use **git pull**. You have to checkout “stable/2009” again to update it:

```
git checkout stable/2009
```

To list the changes between *hash* and the current checked out revision, use

```
git log hash..
```

To learn more about **git**, consult <http://git-scm.com>.

7.2. Building and installing images

7.2.1. Building images

7.2.1.1. Stamp9G20/PortuxG20

After setting up the build environment, you can start to build images. The smallest image you can build is the *helloworld-image*. It contains just a statically linked *helloworld* application, which is started on boot and then hangs forever. This is a good test to see, if your environment is setup correctly.

```
bitbake helloworld-image
```

Although the image is simple, it will take a while until it is ready, because the toolchain will be built first. This can take some hours if you have a slow system. The following builds will not take so long, as the toolchain will then be built already (unless you wipe the *tmp* directory).

As a next step you could built the *base-image*. It contains everything needed to boot the target board.

```
bitbake base-image
```

When the command is finished, you can find the image in the directory */develop/oe/tmp/deploy/glibc/images/machine*. Replace *machine* with the machine you set in the *local.conf*.

If everything worked as expected, you can try to build default image, that came preloaded with your product. It is called *taskit-image*. Before you try to build it, you have to add some lines to your *local.conf*:

```
PREFERRED_PROVIDER_virtual/javac-native = "ecj-bootstrap-native"
PREFERRED_PROVIDER_virtual/java-native = "cacao-native"
PREFERRED_PROVIDER_virtual/java-initial = "cacao-initial"
PREFERRED_PROVIDER_classpath = "classpath"

PREFERRED_VERSION_cacao-initial = "0.98"
PREFERRED_VERSION_cacao-native = "0.99.3"
PREFERRED_VERSION_jamvm = "1.5.0"

PREFERRED_VERSION_classpath = "0.97.2"
PREFERRED_VERSION_classpath-minimal = "0.97.2"
PREFERRED_VERSION_classpath-native = "0.97.2"
PREFERRED_VERSION_classpath-initial = "0.93"
```

Creating a new root filesystem

They are needed, because the taskit-image contains **jamvm** and **classpath** and without the correct version numbers, the build might fail. These are the settings we used for building the image. If they do not work for you, see http://wiki.openembedded.net/index.php/Java#Version_suggestions for further information.

If you only use devices without a screen, you can replace

```
PREFERRED_PROVIDER_classpath = "classpath"
```

with

```
PREFERRED_PROVIDER_classpath = "classpath-minimal"
```

This leaves out all graphical components of **classpath** and thereby improves build times.

7.2.1.2. Stamp9G45

A small image to try at first, is the core-image-minimal. If it builds without errors, you can be sure everything is set up correctly:

```
bitbake core-image-minimal
```

Now you could also try to build the taskit-demo-image but this will take some more time as it includes QtEmbedded:

```
bitbake taskit-demo-image
```

The bootloader can also be built using OpenEmbedded Core. In case of the Stamp9G45 which uses a Linux based loader with an initramfs, you have to switch to uclibc mode so that the initramfs is small compared to an glibc version:

```
TCLIBC=uclibc bitbake virtual/bootloader
```

7.2.2. Installing images

After finishing the build, you will get a standard kernel image and two file system images (jffs2 and tar) in the directory `/develop/oe/tmp/deploy/glibc/images/stamp9g20evb`. There will also be symlinks to each file with a shorter name, e.g.

- `uImage-stamp9g20evb.bin`
- `taskit-image-stamp9g20evb.jffs2`
- `taskit-image-stamp9g20evb.tar`

For the Stamp9G45, the corresponding images will appear in the `tmp-eglibc/deploy/images/stamp9g45/` subdirectory.

7.2.2.1. Installing on an SD card

Before flashing the rootfs into the integrated flash, we will test the system on an SD card. For this, you need an SD card with at least one partition. We will assume, that your SD card

Creating a new root filesystem

is accessible as device `/dev/sdb` and you have a directory `/media/card`. Additionally you have to have root rights to proceed with the following steps.



Caution

Make sure, that you use the correct device file or you will lose data.

The first thing to do is to format the first partition of the SD card with the ext3 file system:

```
mkfs.ext3 /dev/sdb1
```

All data on this partition will be lost.

Now it should be mounted to `/media/card`:

```
mount /dev/sdb1 /media/card
```

You can now extract the files to SD card. Enter the directory `/develop/oe/tmp/deploy/glibc/images/stamp9g20evb` and use the following command:

```
tar -xvf taskit-image-stamp9g20evb.tar -C /media/card
```

Finally, unmount the SD card:

```
umount /media/card
```

If you remove the SD card now and place it into the SD card slot of the target board, you can boot the system from the U-Boot prompt with the following command:

```
run sdboot
```



Important

Do not simply switch off the board when the SD card is mounted read-write (default). The ext3 file system is not made for power loss situations. Use the **halt** command and wait until

```
Power down.
```

can be read on the serial console.

7.2.2.2. Installing on internal flash

When you have tested the root file system on the SD card, you can write the jffs2 version to the internal flash. Because the jffs2 is mounted during the time, Linux is booted, you cannot easily replace it from within Linux. There are ways, but there are a bit complicated and unsafe. Therefore it is better to flash it from U-Boot or the newly created SD card.

In this section we will use the SD card to write the rootfs to the flash memory.

To write the jffs2 image to the flash memory, you obviously need the image accessible on the target board. Either copy it to SD card or mount your development directory via NFS.

Creating a new root filesystem

Before you flash the image, you have to identify the correct MTD-Partition like in Chapter 6, *Compiling a new Linux kernel*

```
cat /proc/mtd
```

You should get something like this:

```
dev:   size  erasesize  name
mtd0: 00020000 00020000 "bootstrap"
mtd1: 00040000 00020000 "uboot"
mtd2: 00020000 00020000 "env1"
mtd3: 00020000 00020000 "env2"
mtd4: 00200000 00020000 "linux"
mtd5: 1fd60000 00020000 "root"
```

The image must be written to the partition with the name “root”, in this case it is `/dev/mtd5`.

First, erase the partition completely.

```
flash_eraseall -j /dev/mtd5
```

The `-j` is used to indicate, that a jffs2 image will be written to the partition. It will put a so-called CLEANMARKER into each flash block. Jffs2 uses them to test, if the last erase operation was correctly completed. If it is missing, jffs2 will erase the block again on the first mount resulting in a unneeded double erase.

Now enter the directory containing the jffs2 image. Depending on the type of flash, use either

```
nandwrite /dev/mtd5 taskit-image-stamp9g20evb.jffs2
```

for NAND flash or

```
flashcp -v taskit-image-stamp9261evb.jffs2 /dev/mtd5
```

for NOR flash.

The jffs2 image is now written to the flash and can be used on next boot with the U-Boot command

```
run flashboot
```

7.3. Customizing images

There are multiple ways to manipulate the contents of the file system image. The following sections describe some of the ways. Some ways need an overlay of your own. So the first thing we do is to create this overlay.

We kind of created a half overlay already in Section 7.1.4, “Setting up the environment”. By specifying `/develop/oe/build` in `BBPATH`, **bitbake** is already instructed to look for

configuration files in this directory. This includes machine definitions. To make it a full overlay, just replace

```
BBFILES = " \
  /develop/oe/taskit-overlay/recipes/*/*.bb \
  /develop/oe/openembedded/recipes/*/*.bb \
"
```

with

```
BBFILES = " \
  /develop/oe/build/recipes/*/*.bb \
  /develop/oe/taskit-overlay/recipes/*/*.bb \
  /develop/oe/openembedded/recipes/*/*.bb \
"
```

in your `/develop/oe/conf/local.conf`

Now, **bitbake** will also look for recipes (package descriptions) in this directory.

7.3.1. Creating your own machine

This section overlaps in parts with Section 7.3.3, “Customizing existing packages”, because you need to customize the kernel package to add a machine.

By adding your board as a new machine, you can add board specific customizations to packages. You could of course do that with the machine, you base your work on, e.g. stamp9g20evb, but than you will not be able to use the original customizations as a reference anymore.

The only disadvantage of creating your own machine is, that you have to replicate customizations you need/want, that have already been done for other machines.

Adding a machine is most of the time a two step process, see Procedure 7.1, “Adding a machine”

Procedure 7.1. Adding a machine

1. Add a machine definition
2. Add a kernel recipe/customize an existing kernel recipe

As you create a machine based on a taskit product, you will most probably just alter the kernel configuration used to build the kernel.

7.3.1.1. Adding a machine definition

Creating a new machine definition most of time boils down to copying an existing machine definition and modifying it. As your are using a taskit product, copy the corresponding file from `/develop/taskit-overlay/conf/machine/` to `/develop/build/conf/machine/`. Create the directory beforehand.

Creating a new root filesystem

As an example, we create a new machine called “custommachine”, based on the **Stamp9G20 EVB™**. We copy the file `stamp9g20evb.conf` to your overlay and call it `custommachine.conf`:

```
cp /develop/oe/taskit-overlay/conf/machine/stamp9g2evb.conf \
  /develop/oe/build/conf/machine/custommachine.conf
```

The contents of the file should now look like Example 7.4, “Custom machine definition before modification”

```
#@TYPE: Machine
#@Name: taskit Stamp9G20 Evaluation Board ❶
#@DESCRIPTION: Machine configuration for the Stamp9G20 Evaluation Board ❷

TARGET_ARCH = "arm"

PREFERRED_PROVIDER_virtual/kernel = "linux"

KERNEL_IMAGETYPE = "uImage"

#don't try to access tty1
USE_VT = "0"

MACHINE_FEATURES = "kernel26 ext2 vfat usbhost usbgadget" ❸

# used by sysvinit_2
SERIAL_CONSOLE = "115200 ttyS0"
IMAGE_FSTYPES ?= "jffs2" ❹
EXTRA_IMAGECMD_jffs2 = "--little-endian --eraseblock=0x20000 -n" ❺

require conf/machine/include/tune-arm926ejs.inc
```

Example 7.4. Custom machine definition before modification

- ❶❷ These are the name and description of your machine. Replace it with appropriate text.
- ❸ These features will be used in task based images or in tasks in general. Tasks are packages that do not contain any files but just dependencies. They will add extra dependencies to some general tasks (e.g. `task-base`). So this is a place, where you can influence to some degree, what packages will be added to the file system image. For available machine features, look into `recipes/tasks/task-base.bb` in the OpenEmbedded repository.

As an example, let us say you do not need any usb support but want to to have **pppd** in all images (works only for task based images or images explicitly using these variables). To achieve that, remove “usbhost” and “usb gadget” from `MACHINE_FEATURES` and add “ppp”.

- ❹ These are the fstypes normally build for this machines. We have overridden this value in Example 7.1, “Bitbake local configuration”.
- ❺ These are extra parameters for the **mkfs.jffs2** command and are essential to build correct jffs2 images for the device. Do not change unless you know what you are doing.

You can also add two further variables `MACHINE_ESSENTIAL_EXTRA_RDEPENDS` and `MACHINE_EXTRA_RDEPENDS`. You can add package names to both variables. The former

Creating a new root filesystem

is used for packages essential to boot and will be added as dependencies for **task-boot**. These should land in almost all images (provided they use **task-boot**). Packages mentioned in the second will be added as dependencies for **task-base**.

As a last resort, you can also add the variable `IMAGE_EXTRA_INSTALL` and list packages that should end in all images. It is highly discouraged to use this method. It is better to create your own image if the other described methods are not enough to customize the image. See Section 7.3.2, “Creating your own image”.

After the mentioned modifications, your machine definition could look like in Example 7.5, “Custom machine definition after modification”.

```
#@TYPE: Machine
#@Name: Custom Machine
#@DESCRIPTION: My first custom machine

TARGET_ARCH = "arm"

PREFERRED_PROVIDER_virtual/kernel = "linux"

KERNEL_IMAGETYPE = "uImage"

#don't try to access tty1
USE_VT = "0"

MACHINE_FEATURES = "kernel26 ext2 vfat ppp"

# used by sysvinit_2
SERIAL_CONSOLE = "115200 ttyS0"
IMAGE_FSTYPES ?= "jffs2"
EXTRA_IMAGECMD_jffs2 = "--little-endian --eraseblock=0x20000 -n"

require conf/machine/include/tune-arm926ejs.inc
```

Example 7.5. Custom machine definition after modification

7.3.1.2. Customizing the kernel recipe

Now it is time to customize the kernel image so that you can let bitbake build the kernel and the file system. This is especially useful if you want to include kernel modules in the file system.

The first step is to create a kernel config as described in Chapter 6, *Compiling a new Linux kernel*.

After doing so, we copy the kernel recipe and corresponding files used for the **Stamp9G20 Evaluation Board**TM to your overlay.

```
cp -r /develop/oe/taskit-overlay/recipes/linux /develop/oe/build/recipes
```

Now create the directory `/develop/oe/build/recipes/linux/linux-2.6.29/custommachine`. In this directory, place the file `.config` from the kernel configuration process renamed to `defconfig`.

Creating a new root filesystem

Finally edit the kernel recipe itself (`/develop/oe/build/recipes/linux/linux_2.6.29.bb`). You should duplicate all Stamp9G20 specific lines, in this case `DEFAULT_PREFERENCE` and `SRC_URI_append`. `DEFAULT_PREFERENCE` tells bitbake, which recipe it should build. The recipe with the highest `DEFAULT_PREFERENCE` for a given machine is built (if the recipe version is not fixed with `PREFERRED_VERSION_recipename` in one of the config files). See Example 7.6, “Customized kernel recipe” for reference.

```
require recipes/linux/linux.inc

S = "${WORKDIR}/linux-2.6.29"

# Mark archs/machines that this kernel supports
DEFAULT_PREFERENCE = "-1"
DEFAULT_PREFERENCE_stamp9g20evb = "1"
DEFAULT_PREFERENCE_custommachine = "1"

SRC_URI = "${KERNELORG_MIRROR}/pub/linux/kernel/v2.6/linux-2.6.29.tar.bz2 \
file://defconfig"

SRC_URI_append_stamp9g20evb = " \
file://stamp9g20.patch;patch=1 \
"
SRC_URI_append_custommachine = " \
file://stamp9g20.patch;patch=1 \
"
```

Example 7.6. Customized kernel recipe

7.3.2. Creating your own image

Creating your own image is relatively easy and the most straightforward way to get exactly what you want into the root file system. You can do it by either extending an existing image or creating a completely new one.

New image recipes should be created in your overlay in the `recipes/images` directory. We will call our image `custom-image` so create the file `custom-image.bb` in this directory.

As a first example, we extend `taskit-image` to get additional packages into it. See Example 7.7, “`custom-image.bb`: Extending `taskit-image`” for reference.

```
require recipes/images/taskit-image.bb ❶

IMAGE_INSTALL += " \
iptables \ ❷
"

export IMAGE_BASENAME="custom-image" ❸
```

Example 7.7. `custom-image.bb`: Extending `taskit-image`

- ❶ This includes the contents of the `taskit-image`. The full path ensures, that it is found although it is in another overlay.

Creating a new root filesystem

- ❷ This adds **iptables** to the contents of your image.
- ❸ This tells the build system how to name the file system image. If you do not set this variable, the image will be called the same as the extended image.

Now imagine you want to base your image on `taskit-image`, but it does contain stuff you do not want, e.g. the java stack, and you still want to add **iptables**. This can only be solved by copying, renaming and editing the image file. Example 7.8, “`taskit-image.bb`” shows you the contents of the `taskit-image` (at the time of this writing).

```

IMAGE_PREPROCESS_COMMAND = "create_etc_timestamp" ❶

DISTRO_SSH_DAEMON ?= "dropbear"
DISTRO_PACKAGE_MANAGER ?= "opkg-nogpg opkg-collateral"

IMAGE_LINGUAS = "en-gb de-de fr-fr" ❷

IMAGE_INSTALL += " \
    busybox \
    modutils-initscripts \
    netbase \
    base-files \
    base-passwd \
    update-alternatives \
    ${MACHINE_ESSENTIAL_EXTRA_RDEPENDS} \
    \
    ${DISTRO_PACKAGE_MANAGER} \
    ${DISTRO_SSH_DAEMON} \
    mtd-utils \
    u-boot-utils \
    jamvm \
    librxtx-java \
    librxtx-jni \
    gdbserver \
    strace \
    libstdc++ \
    ${@base_contains('MACHINE_FEATURES', 'ext2', 'task-base-ext2', '', d)} \
    ${@base_contains('MACHINE_FEATURES', 'vfat', 'dosfstools', '', d)} \
    ${@base_contains('MACHINE_FEATURES', 'ppp', 'task-base-ppp', '', d)} \
"

export IMAGE_BASENAME="taskit-image"

inherit image ❸

```

Example 7.8. `taskit-image.bb`

- ❶ This instructs the build system to create a timestamp in `/etc` in the root file system, so that you always know, when the file system image was created.
- ❷ This variable contains all locales, for which the binary versions should be installed. If you do not need localization in your application, you can leave this variable empty.
- ❸ This statement has to be in all images, that do not derive from other images. It inherits the image bitbake class, which includes all code needed for building images.

Now let us say, you want to get rid of the java stack, do not need the flexibility of `MACHINE_FEATURES`, don't use localization and want to include **iptables**. Additionally, you do not need the C++ library (it would be added anyway, if it was needed by a package).

The resulting image recipe would look like in Example 7.9, “custom-image.bb: Complete new image”

```
IMAGE_PREPROCESS_COMMAND = "create_etc_timestamp"

DISTRO_SSH_DAEMON ?= "dropbear"
DISTRO_PACKAGE_MANAGER ?= "opkg-nogpg opkg-collateral"

IMAGE_LINGUAS = ""

IMAGE_INSTALL += " \
    busybox \
    modutils-initscripts \
    netbase \
    base-files \
    base-passwd \
    update-alternatives \
    ${MACHINE_ESSENTIAL_EXTRA_RDEPENDS} \
    \
    ${DISTRO_PACKAGE_MANAGER} \
    ${DISTRO_SSH_DAEMON} \
    mtd-utils \
    u-boot-utils \
    gdbserver \
    strace \
    iptables \
"

export IMAGE_BASENAME="custom-image"

inherit image
```

Example 7.9. custom-image.bb: Complete new image

7.3.3. Customizing existing packages

This section will only discuss customization for a specific device. You can of course make modifications to all recipes as you like without keeping them specific to one machine.

Machine specific customization of a recipe can be done in two ways:

- overriding variables and functions in the recipe
- overriding files (given in the `SRC_URI` variable) in the recipe subdirectories

Overriding of variables and functions is easy. You just take the variable/function name, add the machine name separated with an underscore to the end. We have already used this in Example 7.6, “Customized kernel recipe”.

In this example, we have also overridden one file. To do that, there can be multiple directories, where you can place the file. If you have a bitbake called `foo_1.0.bb` (meaning it is package **foo**, version 1.0), your file to override can be placed in the following subdirectories in the recipe directory (ordered by priority):

1. `foo-1.0`: Files in this directory are only used for the recipe **foo**, version 1.0.

2. **foo**: Files in this directory are used for all recipes called **foo** without looking at the version.
3. **files**: Files in this directory are used for all recipes in the directory, without looking at the name or version (You can have multiple completely different recipes in each directory. Directories are just used for categorizing recipes.).

So to actually override one of the files, create a directory `custommachine` in one of these directories and place a file with the name you want to override there. It will be used instead of the default ones.

The only thing you have to keep in mind when you customize a package by copying it to your overlay is, that you also have to copy the default files, or at least the ones, you do not override.

7.3.4. Adding own packages

Sooner or later, you want to add your own application to the root file system. This section will tell you how to do it for a simple application (some source files resulting in one standalone binary, no data files, no extra libraries needed).

The sample project contains three files:

- `main.c`
- `util1.c`
- `util2.c`

To build this project we use the Makefile shown in Example 7.10, “Makefile”. It will create a binary called “app” from the sources.

```
app: main.o util1.o util2.o
    ${LINK.c} $? -o $@

clean:
    rm -r app *.o

install:
    install -d ${DESTDIR}/bin
    install -m755 app ${DESTDIR}/bin/app
```

Example 7.10. Makefile

To build the binary with this make file, just enter the following command:

```
make CC=arm-angstrom-linux-gnueabi-gcc
```

To install the binary, the make file includes an install rule. This rule is basically there to ease the writing of the bitbake recipe later. The destination directory will be given in the `DESTDIR` variable, e.g.:

```
make DESTDIR=/develop/appinstall install
```

Creating a new root filesystem

To make the recipe even more easy to write, put the sources and the make file in a directory called `app-1.0` (we assume the project is called `app` and is the first release version).

Now we create a tarball from this directory.

```
tar -cvjf app-1.0.tar.bz2 app-1.0
```

It is now time to create the recipe. To do that, create a directory called `recipes/app` in your overlay and put a file called `app_1.0.bb` in there, see Example 7.11, “`app_1.0.bb`” for reference. Additionally create a `files` directory and put the tarball there.

```
SRC_URI="file://${PN}-${PV}.tar.bz2" ❶
PR="r0" ❷

do_install () {
    oe_runmake install DESTDIR=${D} ❸
}
```

Example 7.11. `app_1.0.bb`

- ❶ This line tells bitbake where it can find the sources. The variables `PN` and `PV` are automatically expanded to the package name and package version. To use these variable has the advantage, that you can update the recipe to a new version by just renaming it.

If you do not want to copy the tarball to the overlay or have it on a webserver, you can of course replace the path with either an absolute path or the URL, e.g.:

```
SRC_URI="file:///develop/${PN}-${PV}.tar.bz2"
```

OR

```
SRC_URI="http://hostname/${PN}-${PV}.tar.bz2"
```

- ❷ This is the package revision. If you leave it out, it defaults to “`r0`”. The value should be incremented every time you modify this recipe so that **bitbake** knows it has to rebuild it and the package manager **opkg** knows it needs to update the package in the root file system after it is build.
- ❸ To install any files, you have to implement the install task (`do_install`). We use the function `oe_runmake` to call the install target of our make file. The variable `D` holds the temporary directory, where all files, that should go into packages, should be installed, so we pass the contents of it to the `DESTDIR` variable.

Apart from that, we do not need anything else in the recipe, as the defaults of OpenEmbedded handle the rest for us.

8. Using and configuring the bootloader

The Stamp9G20 and PortuxG20 come U-Boot preinstalled. This is an open-source bootloader for embedded systems developed and maintained by Denx Software Engineering. As it is already well documented, it is not described here. See the original documentation for further information: <http://www.denx.de/wiki/view/DULG/UBoot>.

The Stamp9G45 on the other hand comes with a Linux-based bootloader named Boots. It uses a standard Linux kernel for hardware support, busybox for scripting and kexec to load the real kernel to be used. There are also some helper scripts. To permanently store settings and boot scripts it uses the U-Boot environment variable tools.

This chapter will describe this bootloader.

8.1. Concepts

8.1.1. Components

The Boots bootloader itself is just a couple of small programs and shell scripts. Combined with the Linux kernel, a small initramfs containing kexec and busybox and U-Boot style environment variables this results in a flexible boot mechanism.

- **Linux kernel.** The Linux kernel is the center of the bootloader and responsible for hardware access. Using it instead of dedicated bootloader drivers gives you the same maturity and stability as the real system.
- **kexec.** Without kexec, this software would not be able to function as a bootloader. Kexec is a mechanism provided by the Linux kernel, to start another kernel without rebooting.
- **Busybox.** Busybox contains a lot of small userspace tools and a shell to allow flexible scripting for the boot process.
- **U-Boot style environment variables.** U-Boot uses a very simple but reliable way to store boot variables and scripts. This method is also used here.

8.1.2. Boot procedure

Figure 8.1, “Boots flow-chart” shows the general boot procedure. As you can see, at first, the script code in the variable `preboot` is executed. After that, the boot is delayed for `bootdelay` tenths of a second. Depending on whether this delay was aborted or not, either `bootcmd` or `abortcmd` is executed. Normally, `bootcmd` does not finish, as it is used to boot into the real system, but if it does or `abortcmd` finishes, either a shell will be launched for you to interact with the system, or `loopcmd` will be executed if it exists. Both will be restarted a second after they finish.

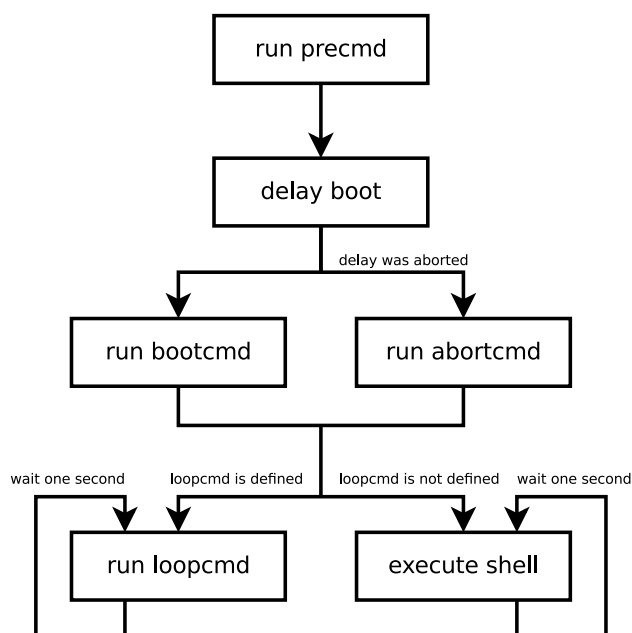


Figure 8.1. Boots flow-chart

8.2. Reference

8.2.1. Environment variables

abortcmd	command (list) that is executed when the boot delay is aborted
abortparams	contains additional parameters to control the abort of the boot delay, can be either
	key {key-nr}
	or
	gpio {gpio-nr} [high-active]
	arguments:
key-nr	key number as specified in <code>linux/input.h</code>
gpio-nr	GPIO number as described here: http://www.armbedded.eu/node/258
high-active	specifies if the GPIO should be high active, every number greater 0 means high active, everything else (even leaving it out) means low active
bootcmd	command (list) that is executed after the boot delay
bootdelay	delay between the execution of <code>precmd</code> and <code>bootcmd</code> , can be aborted by a key press
bootdevices	white space separated list of devices that a search for bootable system by the autoboot command

Using and configuring the bootloader

<code>delay_device</code>	delay in seconds that the autoboot command should wait before trying to look for a bootable system on <i>device</i> , can be needed for devices that are enumerated asynchronously
<code>fstype_device</code>	filesystem type used by the autoboot when mounting <i>device</i> , can be empty most of the time, when not using a flash filesystems
<code>kexecoptions</code>	extra options used by the autoboot when invoking kexec
<code>loopcmd</code>	command (list) that, if defined, gets executed in a loop after either <code>bootcmd</code> or <code>abortcmd</code> finishes
<code>options_device</code>	mount options used by the autoboot when mounting <i>device</i> , default to "ro"
<code>precmd</code>	command (list) that is executed on every boot before the boot delay
<code>stdin, stdout, stderr</code>	contain the device file, which is used to get input or print output and error messages, defaults to <code>/dev/ttyS0</code>