An Introduction to the STAG Framework

Philippe De Swert

Date: 08/2004
Revision: 0.8

Table of Contents

History

Author Philippe De Swert. Additional OpenOffice formatting and templatizing done by Paul Schulz

Comments?

Copyright

This needs to be added. Appendix? Done

Todo

Some suggestions on what to do to improve the usefulness.

- More references. How does this document tie in with the official Debian documentation?
- Diagrams
- Remove notes! (Those things in boxes.)

1. Preface

The first chapters contain general information about which problems are specific to embedded development, with an explanation of the package management of the Debian GNU/Linux distribution, and instructions on how to set up a toolchain and a final root file system. These chapters are of interest to those people who want to find out what the Stag framework is about and why we use these tools. Experienced developers and people familiar with the Debian distribution should have little difficulty if they skip these chapters.. Later chapters explained what Stag is, how it should be used and how it can be extended.

For the moment this is still a draft. Because all details are not already sorted out, and I haven't had the time to check everything out, there could be errors in the text. Also the number of packages is very small. For the moment just enough to boot a system in a shell (busybox), but it should be enough to start with it and get the idea.

Comments are welcome.

Philippe De Swert, <philippe.deswert@pi.be>

2. Common problems in embedded (GNU/Linux) development.

First of all we have to take into account the most important limitations we have to work with when doing embedded development.

- size constraints
- processing power
- development host and target are really different
- inconsistent software configuration

First of all we have to make sure that all software needed for the functionality of the embedded systems fits in the memory space that is available. In most systems this is generally between 2 to 16 Megabytes of flash, or an even tinier ROM. Usually most software is not designed to fit in such a space. When the software is available for your target processor (like ARM processors for example) in already compiled and eventually packaged form the size is mostly not taken into account. In most cases the size of these binaries can be reduced by removing some functionality that is not needed to meet the requirements of the embedded project. In the case of packages binaries it is even worse because most of the time the program then comes with a lot of extra files, like documentation, translations,... The size can also be reduced by using libraries that are designed to be small themselves and make small linked binaries.

Because of these size constraints it is also almost impossible to do development on the system itself. The processing power of the typical embedded processor also makes it difficult to do this if there is enough room. Because these processors are usually quite weak because of power consumption and size and heat emission limits, programs do not get compiled very fast. This is also a problem because a developer always has a time-frame to work in and deadlines to meet.

This is usually solved by cross-compiling. However this approach brings along its fair share of problems. Cross-compilers are relatively difficult to build, and are more difficult to use. Also it you need to have a different cross-compiler for each architecture you are working on. You also have to be careful how your software is being linked. It's clear that linking with host specific instead of target specific software is a behaviour we do not want. Here we stumble again over the size problem because of libraries. As mentioned earlier, the choice of library used when linking and compiling your programs has an influence on the size of the final binaries. Your cross-compiling toolchain has to know about these libraries and use them correctly.

Sometimes, when building software, the generated binaries are tested or even worse used in the building process. As this happens on a different host with a different processor these tests fail because the generated binaries do not work on the development host. Because of this a developer often has to spend some time on finding workarounds to make the software actually build. Also the source code is not always easily cross-compiled. This could happen when the developer of the software added platform specific optimalisation routines. Because of this the embedded developer has to regularly patch the software, or write patches himself. This requires a good knowledge of different programming languages and architectures to recognise eventual problems in the source code.

This brings us to the inconsistent software configuration. One of the weaknesses (but also one of its strength) is the lack of standardisation. Because everybody is

free to develop his/her software as he/she likes we can have different ways how the software is build or configured. There are generally accepted ways of configuring the source code and building the software, like the autotools and ncurses interfaces, but still each program comes with it's own options. Often the developer has not the time to test all these options or to figure out what is their impact. When omitting an function or option to much he can run into dependency problems when other programs rely on that functionality. Another problem could also be leaving to much functionality which will in turn generate to much or to big code.

In a later chapter you can find an explanation about which problems the Stag framework tries to solve and how this is done.

3. The Debian package format (*.deb).

For the Debian GNU/Linux distribution a package system was developed to simplify the installation of software and the distribution itself. This package system is well thought out, defined down to the tiniest detail and is controlled by a large group of Debian developers. Debian also has a very strict policy about the contents (and structure) of the packages, like the location of the files and directories, the software licence, ...

The most remarkable thing about Debian is the easy of use of this package system and the flexibility of the installation media. Debian packages and thus also the full distribution can be installed from almost any medium you can think of. Installation over the local network through NFS, HTTP or FTP, from the hard disk, CD, DVD or floppy's,... This is one of the reasons why this type of package was chosen for this project.

Essentially there are two types of Debian packages, the binary and the source packages. In the binary packages we only find the so-called binaries, this is essentially not else than the compiled and eventually executable code and/or needed data. In a source package we can find, like its name indicates, the source code of the program. The code in this package can be compiled automatically when installing or not.

There are now also 'udeb's, (micro-debs) designed for use in installation media.

3.1. The Debian binary package.

A Debian binary package is actually an ***ar*** archive which contains some files. These files are ; ***debian-binary***, ***control.tar.gz*** and ***data.tar.gz***.

In the ***control.tar.gz*** file, which is also an archive, we can find the following files : ***shlibs***, ***postinst***, ***postrm***, ***md5sums*** and ***control***. Sometimes we can also find the files; ***conffiles***, ***preinst*** and ***prerm***.

In the ***control*** file there are a lot of variables that are used by ***dpkg***, ***dselect***, ***apt*** and other Debian package tools. These variables each have a different meaning and some of them are required, others are optional.

- Package: Defines the name of the binary package.
- Architecture: States for which processor-architecture the package has been compiled.
- Depends: Tells which other packages have to be installed before we can have a working installation of the software contained in the package we wish to install.
- Recommends:Packages that are usually used in conjunction with the package you are about to install will be mentioned here, but will not be installed.
- Suggests: Other packages that add extra functionality, but that are not needed by this package to function should be listed here.
- Pre-Depends: This will make that the selected package will only be installed when other needed packages are installed AND configured. The use of this is usually discouraged.
- Conflicts: When the selected package cannot function correctly when another package is installed this should be mentioned here. This is rather exceptional but if it is the case it must be mentioned here.

- Provides: Some packages provide a service that is defined in as a virtual package, like for example the mail program mutt provides mail-client. A full list of the virtual packages can be found in ***/usr/share/doc/debian-policy/virtual-package-names-list.txt.gz*** in your Debian installation. Use this function if your package provides a function of an existing virtual package.
- Replaces: This is used when the package replaces files from another package or even another full package.
- Maintainer: Here we can find the name and the contact information of the maintainer of this package.
- Description: A short description of the functionality and the function of the program included in the package can be found here.

In ***shlibs*** the different shared libraries that are eventually used by the software in the package are listed here.

Postinst, ***preinst***,***postrm*** and ***prerm*** are so-called maintainer scripts. These scripts are run when the package is removed, upgraded or installed.

Md5sums contains a checksum, namely the md5sum, for each file contained in the package. This way the integrity of the package can be controlled (this will be done automatically by the package management system).

In ***conffiles*** the configuration files are listed, so that when upgrading the package management system asks if the existing configuration files should be overwritten or kept in the current state.

Finally in ***data.tar.gz*** we find all the files with their exact path, that will be installed on our system.

A normal Debian package can be easily installed (if the package management system is correctly configured) in the following way:

```
$ apt-get install <package-name>
```

Apt will then automatically download the package (from the internet, your personal mirror, a CD,...) and install it on your system.

3.2. The Debian source package.

The source package is actually a collection of files. These are ***<program-name>-<version>.orig.tar.gz***, ***<program-name>-<version>.diff*** and ***<program-name>-<version>.dsc***.

In ***<program-name>-<version>.orig.tar.gz*** we find the original source code, also called upstream code by the Debian Developers. This code is decompressed and unpacked when installing the package.

After the original source has been installed ***<program-name>-<version>.diff*** is used to apply the changes that were done when “Debianizing” it. These adaptations are needed to fit in the final program in the distribution. It is essentially what we call a ***diff*** file, this is actually nothing more than a textfile which contains the changes between two versions. Of course with some extra syntax to be able to apply the changes. For example a +-sign means a line has to be added, a minus means a line has to be removed.

The ***<program-name>-<version>.dsc*** file contains a short description in ASCII text, together with an ***md5sum*** and a PGP key. As mentioned earlier the ***md5sum*** is used to control the integrity of the files, the PGP key is used to identify the maintainer of the package.

Installing a Debian source package is only slightly more difficult than installing a Debian binary package.

```
$ apt-get source <package-name>
```

This will download the source package and unpack it in your current work-directory. After this we have to compile the source code ourselves.

This can be done automatically when we use the extra -b option for *apt*.

```
$ apt-get source -b <package-name>
```

We will later see how a toolchain is being built. There already are some Debian packages to build one from source, using the following procedure.

```
$ apt-get install toolchain-source toolchain-source-gdb debhelper
$ tpkg-make <arch>  (as in tpkg-make m68k for example)
$ debuild (adapts the source and starts the build)
$ debi (installs it)
```

However this is not a real source package, other examples are the kernel-source package! This is because it only contains the source and cannot be compiled automatically. Also notice that these are installed with ***apt-get install*** instead of ***apt-get source***. Because building a toolchain this way sometimes leads to unpredictable results when the versions of the Debian tools change, this is not recommended.

3.3. Useful peculiarities of the package management system.

One of the most useful features is the management of the so-called “Orphaned” packages. The package management system knows which packages have been installed and all of their dependencies. If there are packages installed that are not needed by any other package and they have been installed because of dependencies, we call them orphaned. This could happen when the program/package that needed that orphaned package to work has been removed. To avoid such orphans and losing the space they need the package management system will ask if you want those packages removed.

3.4. Making an official Debian package.

There are two ways to make an official Debian package. We can start with already compiled code or binaries or we use the source code. Keep in mind that for obvious reasons source packages can only be made if we start with the source code.

3.4.1. Starting from source code.

First of all we have to choose the program we will package. We get the source code, which usually is a *.tar.gz or *.tar.bz2 compressed archive. Then we make a new directory in which we will do all the working. For example a new debian sub-directory in our home directory in which we will place the archive we just downloaded..

```
$ mkdir ~/debian
$ cd ~/debian
$ tar -xvjf <program-source-code>.tar.bz2
```

or

```
$ tar -xvzf <program-source-code>.tar.gz
```

In this directory we should now have the decompressed and unpacked source along the original archive. The source code then usually resides in a ***<program-***

name>-<version> directory. Enter this directory and check out the installation procedure, the documentation and the licence. Because Debian is very strict about which licence is used you should be very careful about this if you want your package accepted in the distribution. Try to build and install it in the regular way first to test the program.

This is mostly done like this (in the source-directory):

```
$ ./configure
$ make
$ make install
```

If it compiles and executes correctly you can start building the package. First of all you have to clean up the source directory:

```
$ make clean
```

or

```
$ make distclean
```

Alternatively you can start with a new source tree.

Now is the moment to choose a name for your package, this should fully be in lower-case letters and have a maximum length of 20 characters.

The source code directory has to be moved to a directory <package-name>-<version>. In this directory we will do the first “debianisation” of the source code with ***dh_make***.

```
$ dh_make -e your_email@adres.com -f ../program-source-archive
```

This brings up some questions. Like, is it a single or multiple binary package (multiple binary packages are used to package big packages or to split up different functionalities), a library package or a kernel module package? Take care not to run ***dh_make*** twice in the same directory because it could create problems, luckily newer versions refuse to be run a second time.

Most programs install by default in /usr/local/... . In Debian this directory has been reserved for the system administrator. This means you will have to adapted the source so that the program will install in /usr. This must not be done by every program, most programs which use ***./configure*** in their first compilation stage can be forced to install somewhere else through the ***rules*** file. If this is not the case you will have to edit the ***Makefile***. Also ***Makefile.am*** and ***Makefile.in*** will have to be adapted. The exact locations for the different files can be found in the File system Hierarchy Standard (more explanations about the FHS can be found in the Building root file system chapter) .

Once we know where everything will be placed and we are sure the files will indeed be installed there we need to edit or add some files in ***~/debian/<package-name>-<version>/debian*** directory. This is a sub-directory generated by ***dh_make*** in the directory where we have placed the uncompressed source code.

The most important are; ***control***, ***changelog***, ***copyright*** and ***rules***.

- The ***control*** file has been discussed in detail earlier.
- ***Copyright*** contains all information about the author, the site from where the source was downloaded, the licence and the contact information of the maker of the Debian package (in this case this should be you). The licences that Debian accepts are; GNU GPL or LGPL, BSD or Artistic.
- In the ***changelog*** file we find information about the version, eventual patches, and other important information about changes to the program in the package.

- The ***rules*** file will be called by ***dpkg-buildpackage*** when building a package, however we can also use it manually because it is executable. In reality it is a special *M**akefi**le* that has nothing to see with the standard source of the program. However this file will decide how your final package will look like.

There are also some non-mandatory files in that debian directory. For now you will probably find some files that end on .ex, and others that are ***README.Debian***, ***dirs*** and ***docs***. The files ending on .ex are examples, and when we plan to use them (after we have added the right information) we will have to rename them so that the extension is removed.

In ***README.Debian*** we have to write down what are the eventual differences between the original source code and the Debian version of it, together with comments from the packager.

In ***dirs*** we find the directories we need for a correct installation, but for whatever reason are not made by the installation-routine.

We use ***docs*** to specify which documentation will be installed.

Conffiles.ex is an example of the ***conffiles*** file. In this file you can specify the names and the path of any configuration file you may have. If you do not need it you can remove the file.

If your program has to do certain tasks at regular intervals you can configure the ***cron*** daemon with ***cron.d.ex***. If you do not use it you have to remove this file.

To be able to use the special ***Emacs*** functions of certain programs you should use the ***emacsen-*.ex*** file.

If your program relies on a startup script , like for example a daemon, you can use ***init.d.ex*** to add the correct files in ***/etc/init.d***.

To add a manual page (the so-called manpages) we can use ***manpage.1.ex*** or ***manpage.sgml.ex***. If you want to add a standard nroff formatted man-page you should use the first. Otherwise you use the latter for an sgml formatted man-page. Do not forget to replace the word manpage by the name of your program. More info about all this can be found in the standard Debian manpages.

More references should be added.

Debian uses an unified menu system that is being used by all window managers. If you want to add an entry to it when installing your program ***menu.ex*** is the file to adapt.

For use with the uscan and uupdate programs which are advanced tools from the devscripts package there is the watch.ex file. Uscan and uupdate are programs who regularly look at the sites where the original source came from to see if there are any changes. These programs are only used by very experienced Debian packagers for software that changes a lot, so this will not be necessary often.

If there is other documentation than the standard manpages this can be added by using the ***ex.package.doc-base***. Usually these are HTML, PS, PDF or text files who then will be installed in ***/usr/share/doc/<package-name>***. The name of the file will have to be changed to ***doc.base*** if used, and otherwise be deleted.

Postinst.ex, ***preinst.ex***, ***postrm.ex*** and ***prerm.ex*** are example scripts for the ***postinst***, ***preinst***,... files that were discussed earlier.

Now we have finally finished the preliminary work and we can start building the final package. We have to possibilities here.

The complete build :

Therefore we have to use the following command in the source dir;

```
$ dpkg-buildpackage -rfakeroot
```

If all goes well this should do the following:

- clean up the source tree (using ***debian/rules clean***)
- make the source package (actually ***dpkg-source -b***)
- build the program (***debian/rules build***)
- make the binary package (***debian/rules binary***)
- make a ***.dsc*** file
- make a ***.changes*** file.

The -rfakeroot flag is used to make the program build so that it fits in a normal root file system with the privileges of a normal user without the risks involved when compiling a program as the root user.

When the whole process is finished we should get these files:

- ***<package-name>-<version>.dsc***: This contains the information about the source package (see earlier) and has been generated by using the ***control*** file.
- <***package-name>-<version>.diff.gz***: Contains all your changes to the original source, but this will only be generated if the package building process has access to the original source archive that has been generated by ***dh_make*** and which is ***<package-name>-<version>.orig.tar.gz***.
- ***<package-name>-<version>.deb***: This is the final Debian binary package.
- ***<package-name>-<version>.changes***: In here all the changes relative to earlier versions of the package and any other relevant building information is listed here. This is partly generated from the changelog and ***<package-name>-<version>.dsc*** file. This file has been PGP signed and the information contained is used by the Debian archive ftp maintenance programs.

The quick rebuild:

Because everything usually does not work like it should the first time, and eventually the next time and so on..., You will have to change the ***debian/rules*** file regularly. To test the changes without having to compile the sources from scratch each time there is a faster way:

```
$ fakeroot debian/rules binary
```

Do a full rebuild once the changes have been tested to be sure they are all correct!

From existing binaries.

We create again a new directory debian (it can have any name, this is just a personal choice). In this directory we add the files with their absolute path as they would be placed in a regular root file system.

For example if you have a file ***/usr/bin/something-program*** it should be

copied in a /usr/bin sub-directory of the debian directory used here. When all files have been copied we make an extra sub-directory DEBIAN (Debian is case sensitive (as most Unices) and thus must not be confounded with the debian directory. In here we will place a ***control*** file. In ***debian/usr/share/doc/<package-name>-<version>*** we will have to place a ***manpage.1***, a ***copyright***, a ***changelog*** and a ***changelog.debian*** file.

We also have to make sure that all sub-directory's have 0755 as permissions.

Then we can build the package with these commands:

```
$ fakeroot dpkg-deb –build ./debian
$ mv debian.deb <package-name>-<version>.deb
```

Here again you have to use the fakeroot command. This time to ensure that the files in the pakage will have the user root as owner and not the current user.

3.5. Testing a Debian package.

To test a Debian package the program lintian is a very reliable and simple tool. To use it you just have to issue the following commands. Of course there are more advanced tests, but in the usual cases this is not needed.

```
$ lintian <package-name>-<version>
$ lintian <package-name>-<version>.changes
```

When testing the package ***lintian*** will show some information about the test and the results. Lines starting with an W are warnings and can eventually be ignored. Lines starting with an E are errors which, of course, have to be resolved.

4. The ipkg format.

*** Who developed the ipkg format?

4.1. Description.

The ipkg format has been derived from the Debian packages (*.deb). The main goal was to simplify the packages, and to remove some file content. This because ipkg is targeted at embedded systems which have not a lot of memory or storage at their disposal. This can essentially be met by removing documentation and other “superfluous“ files.

Essentially it is nothing more, just like a Debian package, than an ar archive which contains the following files:

Is this the different/same to 'ipk' files which are 'tar.gz' files.

- ***debian-binary*** (which contains the a version number)
- ***control.tar.gz*** (archive containing the control files)
- ***data.tar.gz*** (contains the actual binaries and other data)

Debian-binary is an ASCII text file containing the version number of the package management system.

In ***control.tar.gz*** we find all the control files. Most of them are the same as in a standard Debian package, however there are less. To sum up; ***control***, ***postinst*** and ***postrm***.

In the ***control*** file we can find the most important data. This has been split up in different fields.

- Package: The name of the program contained in the package.
- Priority: An indication of the importance of the package.
- Version: The version number of the program.
- Architecture: Indicates for which architecture the program has been compiled. (actually only arm in the case of ipkg for the moment.)
- Maintainer: Name of the person who takes care of the package.
- Section: Here we find more information about what type of program the package contains, examples are: base, optional, games, multimedia, misc, console,...
- Source: Points out where the software originally came from.
- Depends: Other packages that are needed for the current package to work.
- Description: Obviously a short description of the functionalities of the software.

The following fields are mandatory : Package, Version, Architecture, Maintainer, Section, and Description.

Non-mandatory are Priority and Depends. Problems can however arise if the package has incorrect or incomplete dependency information.

For the moment Suggests, Recommends, Provides, Conflicts and Replaces are not available.

The ***postinst*** file is an executable script with instructions that have to be executed after the installation of the package. This script is automatically called

after a successful installation.

Postrm can be compared with the ***postinst*** file, only this script will be executed when the package is removed.

4.2. Using ipkg.

```
usage: ipkg sub-command [arguments...]

Package Manipulation:
  update Update list of available packages
  upgrade Upgrade all installed packages to latest version
  install <pkg> Download and install <pkg> (and dependencies)
  install <file.ipk> Install package <file.ipk>
  remove <pkg> Remove package <pkg>

Informational Commands:
  list List available packages and descriptions
  files <pkg> List all files belonging to <pkg>
  search <file> Search for a packaging providing <file>
  info [<pkg>] [<field>] Display all/some info fields for <pkg> or all
  status [pkg] [<field>] Display all/some status fields for <pkg> or all
  depends <pkg> Print uninstalled package dependencies for <pkg>
```

The ***/etc/ipkg.conf*** file will decide from which repositories (or servers) ***ipkg-update*** and ***ipkg-get*** will get their packages. For the rest it is clear that using ipkg is very similar to ***apt***.

4.3. Building an ipkg.

Instead of two methods to build a Debian package there is only one. As such it is also much simpler to build one. You will also notice, or already did, that there is no rules file. This means that you can only build an ipkg package from binaries.

The procedure is as follows:

1. Make a directory and place all the files of the program with their correct path in here.
2. Make a sub-directory named CONTROL.
3. In the CONTROL sub-directory you have to place a control file which looks a bit like this example:

```
### Begin CONTROL/control example

Package: foo
Priority: optional
Version: 0.0
Architecture: arm
Maintainer: Familiar User
Depends: libc6, grep
Description: foo is the ever-present example program -- it does
 everything
 vfoo is not a real package. This is simply an example that you
 may modify if you wish.
 .
 When you modify this example, be sure to change the
 Package, Version, Maintainer, Depends, and Description fields.

### End CONTROL/control example
```

1. If you want to add some configuration files you also have to add a ***conffiles*** file in the CONTROL directory. This file has one line with the absolute path of that file for each configuration file .

2. If needed add also the necessary scripts, in this case this could be ***postinst*** or/and ***postrm***.
3. Now you have to use the ***ipkg-build*** command with as an argument the name of the directory you just made with the necessary software. Alternatively you can also use ***mkipkg***.

4.4. Recent improvements of ipkg.

There was a bug in ***ipkg remove*** so that dependency checking was not correctly done when removing a package. This was solved in version 0.99, however ***ipkg*** still does not complain when really important packages are removed.

Also new in the 0.99 release is the support for version depend dependencies. However Pre-depends is still not correctly handled (it is used in the same way as Depends). Also Provides, Conflicts and Replaces are now supported.

Now there is also support for the orphaned packages. These are the same as the orphaned packages in the Debian package management system. When removing software that depends on them ***ipkg*** will ask if it has to remove the packages that are not needed any more.

An important feature of this 0.99 release is the ability to install Debian packages.

4.5. Why using Debian packages instead of ipkg packages?

You will already have noticed the similarities between the Debian and the ipkg packages. This has to do with the fact that ***ipkg*** was derived from the Debian packages. But what are now the major differences that make Debian packages were chosen instead?

Well first of all ***ipkg*** was designed to be run on an embedded platform. However for the moment it is only supported officially for the arm architecture because it is relatively new. The Debian package management system however is already truly multi-platform and has also been thoroughly tested. Also we use the Stag framework on a development host, so we do not really have to pay attention to size constraints of the utilities for installing our software. Another important fact is that because of its limited size ***ipkg*** cannot contain all the functionality of the Debian package management system, so that dependency checking and other features are not so fool-proof. This is not really important on a embedded system with a small amount of packages but could be crucial on an development host.

Also ***ipkg*** recently gained support for Debian packages,, which makes it possible to use the special Stag packages on the embedded system in the same way that ***ipkg*** packages would be used.

What does this mean? It means that you can use ipkg to manage stag packages. This means you can also use stag packages in already existing ipkg based systems.

5. Toolchains

5.1. Toolchains ready to use.

On the internet there are many toolchains available that are ready to be used. They are pre-compiled for a lot of targets and platforms. It could well be that there is one in the software suite that belongs with your embedded system.

For examples look at : www.uclinux.org, www.ucdot.org, www.google.com, ...

5.2. Compiling a toolchain by hand.

Here you got quite a lot of possibilities, and this approach also gives you more control of your toolchain. However it is not the most easy nor the fastest way because the process is affected by a lot of factors. You could try a full manual build by searching all the necessary sources and patches and do the building by hand. This is however a very time-consuming task. Another way is to use or adapt an existing script. One script needs more work than another, but usually they save a lot of time and simplify the task of building a toolchain. In this chapter there are two script that will be discussed, first of all the script of Bill Gatliff (http://crossgcc.billgatliff.com/) and secondly the script (or better script-set) that Dan Kegel(http://www.kegel.com/crosstool) made, based on the former script, and which almost makes the building task full-automatic.

All these methods have the following steps in common:

1. prepare the kernel headers correctly
2. configure and compile the binutils
3. make a bootstrap ***gcc*** compiler
4. configure and compile the C-library (mostly glibc)
5. fully build the ***gcc*** compiler

Full manual build.

This is only practical for one architecture and (experienced) people with a lot of time and/or a fast computer.

Preparation.

First of all we should download the source code of the different parts. These are:

- Binutils (www.gnu.org/directory/GNU/binutils.html)
- GCC (http://gcc.gnu.org, full source)
- glibc (http://www.gnu.org/directory/GNU/glibc.html)
- glibc-linuxthreads (ftp://ftp.gnu.org/gnu/glibc)
- a Linux-kernel eventually for your platform, version not important (www.kernel.org)
- the eventual patches for your target-architecture (=target) (www.google.com)
- a working *gcc* for your current platform (=host)(usually provided in your GNU/Linux distribution)
- eventually newlib (http://sourceware.org/newlib/), uclibc (http://www.uclibc.org) or another c-library. In the example glibc is used and the differences for using uclibc and newlib will be

discussed.

We place everything in a project-dir so we can easily find everything. Personally I use ~/toolchain. For every step I use a different build-dir, to keep sources and compiled code apart. This has the advantage of always having a clean source tree when restarting a step because of errors or building for a different architecture..

In ~/toolchain we find the following directories: ***binutils-<version>.tar.bz2***, ***gcc-<version>.tar.bz2***, ***glibc-<version>.tar.bz2***, ***glibc-linuxthreads***, ***Linux-<version>.tar.bz2*** and all needed patches. We will also create the following directories: ***build-binutils***, ***build-bootstrap***, ***build-glibc*** and ***build-gcc***.

The /usr/$target-linux directory (with $target being replaced by the target architecture, like m68k, arm,...) has been chosen in concordance with the FHS (File Hierarchy Standard, see the chapter about building a root file system for more information), and thus the toolchain will fit into the strict rules for a Debian distribution. But actually the choice of directory is yours. Other useful places are the own home or the /opt directory.

The files with the tar.bz2 extension are compressed (with bzip2) tar archives. This archive form has the advantage t be smaller than the gzip-ped tar archives because of their smaller size. This has as consequences; less storage space to use, shorter downloads, less server load, ... But of course you can still use the gzip compressed files (they have tar.gz as extension).

The different versions can best be chosen based on information that can be found on the net about working combinations. Otherwise we could be spending a lot of time on trying different combinations before finding a combination that compiles into a working toolchain.

Earlier newlib and uclibc were mentioned. These are other C-libraries specially designed for embedded systems. They can compile easier than glibc in some cases and have a smaller footprint due to their design. They could be used when you fail building a correct glibc-library. More explanations will be given in this chapter and in the chapter about the root file system.

Sometimes the kernel header set-up is not needed for building some toolchains. As this step is relatively easy and not very time consuming it is recommended to do this step. If it is not needed it will not do any harm.

Generating the kernel headers.

First of all we have to decompress the source. We do this in ~/toolchain/ with:

```
$ cd ~/toolchain/
$ tar -xvjf ../linux-<version>.tar.bz2 or tar -xvzf Linux-<version>.tar.gz
```

The following is only for kernelversions smaller than 2.4.19.

The new linux directory should be renamed to linux-<version>.

```
$mv linux linux-<version>
```

Now we have some configuration work to do:

```
$ cd linux-<version>
$ make ARCH=$target-architecture CROSS_COMPILE=$target-system- menuconfig
```

Now we have some sort of menu system were we will select the correct processor ans system-type. We exit these menus and save the configuration. Alternatively we can also use ***make xconfig*** or ***make config***. However menuconfig is the best option of the three because it balances ease of use and correctness.

With $target-architecture we mean of course m68k, arm, i386, ppc,... and with target system i386-linux, arm-linux,....

The headers are now configured but not in the right place, so we copy them to their correct locations.

```
$ mkdir -p /usr/$target-linux/
$ cp -r include/linux/ /usr/$target-linux/include
$ cp -r include/asm-$target /usr/$target-linux/include/asm
$ cp -r include/asm-generic/ /usr/$target-linux/include
```

Binutils.

The binutils are a set of programs used for manipulating binary objects. The two most important ones are; ***as***, which is the GNU assembler, and ***ld***, the GNU linker. *As* supports lots of processors but with a machine independent assembler language/syntax derived from BSD 4.2 assembler.

After extracting the sources, in the same way as with the kernel sources we start with the configuration.

```
$ cd ~/toolchain/build-binutils
$ ../binutils-<version>/configure --target=$target --prefix=/usr/$target-linux
```

Do replace $target by the correct processor architecture for which we are generating this toolchain.

After we issued previous command we will see a lot of text passing on the screen. This text is generated by the autotools when configuring the binutils source. When this configuration is is successful we can start compiling with make.

```
$ make
```

The compilation of binutils should take something between 10 to 30 minutes, depending on how fast your computer is. When everything has compiled successfully we have to install the binaries into their correct locations.

```
$ make install
```

Now the binutils should be installed in /usr/$target-linux/bin.

The bootstrap compiler.

We have to compile ***gcc*** in two steps. The result of the first step is called the bootstrap compiler. Only after the second step we have the final compiler. These two steps are necessary because we need to compile the C-library which is needed for the full compiler build, for our target processor. This means that the bootstrap compiler will not be a fully fledged compiler and only be able to compile C, and not like the final ***gcc*** C and C++.

Again we start by unpacking the source and positioning ourselves in the build directory in which we will do the configuration and compilation.

```
$ cd ~/toolchain/
$ tar -xvjf gcc-<version>.tar.bz2
$ cd ~/toolchain/build-bootstrap
$ ../gcc-<version>/configure --target=$target --prefix=/usr/$target-linux \
   --without-headers \
```

```
   --with-newlib --enable-languages=c
```

The –target en –prefix options are the same as earlier. We use –without-headers and –with-newlib because we do not have all the needed headers en we do not want to use the C-library of the host. And –enable-languages=c is to tell the configuration program that we will be building a compiler which will only understand C. When the configuration process is finished we can start compiling.

```
$ make all-gcc
$ make install-gcc
```

CAREFUL! : When we want to compile gcc version 3.2 or higher we have to do an extra step. This is because the –without-headers option does not work correctly. To solve this we have to install the glibc headers before we compile the bootstrap compiler.

First we unpack the glibc source code together with the necessary glibc-add-ons.

```
$ cd ~/toolchain
$ tar -xvjf glibc-<version>.tar.bz2
$ tar -xvjf glibc-linuxthreads-<version>.tar.bz2 --directory=glibc-<version>
```

We create a new work directory build-glibc-headers and will create the headers inhere.

```
$ mkdir build-glibc-headers
$ cd build-glibc-headers
$ CC=gcc ../glibc-<version>/configure --host=$target --prefix=/usr  --without-cvs \
  --disable-sanity-checks –with-headers=/usr/$target-linux/include/
$ make sysdeps/gnu/errlist.c
$ mkdir -p stdio-common
$ touch stdio-common/errlist-compat.c
$ make cross-compiling=yes install_root=/usr/$target-linux/ prefix="" install-headers
```

Because we don't have a cross-compiler yet we set the cross-compiling flag on yes, to be sure that no parts of the library will be compiled for the host instead for the target. With install headers we only install the headers. After this we have to create a dummy ***stubs.h*** file to make sure the bootstrap compiler builds. Do not worry a correct version of this file will be generated automatically later. We also need to copy ***features.h*** to the correct location.

```
$ mkdir -p /usr/$target-linux/include/gnu
$ touch /usr/$target-linux/include/gnu/stubs.h
$ cp ../glibc-<versie>/include/features.h /usr/$target-linux/include/features.h
```

Finally we can build the bootstrap compiler, after unpacking the source (see earlier).

```
$ cd ~/toolchain/build-boot-bootstrap
$ ../gcc-<version>/configure --target=$target  --prefix=/usr/$target-linux \
  --disable-multilib --with-newlib --without-headers --disable-nls –enable-threads=no \
  --enable-symvers=gnu --enable-__cxa_atexit --enable-languages=c --disable-shared
$ make all-gcc install-gcc
```

When this is finally done we can move on to the next step, building the C-library.

The C-library.

The glibc package contains not only the standard C-library, but also a lot of other libraries, like for example encryption etc... Thus this is also the largest package we will compile when building the toolchain, so it will have the longest compilation time.

Here we also have to unpack the source first in the ~/toolchain directory. Once this is done we will position ourselves in the build-directory and do the

configuration.

```
$ cd ~/toolchain
$ tar -xvjf glibc-<versie>.tar.bz2
$ tar -xvjf glibc-linuxthreads-<versie>.tar.bz2 --directory=glibc-<versie>
$ cd build-glibc
$ CC=$target-linux-gcc ../glibc-<versie>/configure --host=$target \
  --prefix=/usr/$target-linux --without-tls --without-__thread --enable-kernel=2.4.3 \
  --without-cvs --disable-profile --disable-debug --without-gd --enable-clocale=gnu \
  --enable-add-ons=linuxthreads –with-headers=/usr/$target-linux/include \
  --disable-shared
```

With CC=$target-linux-gcc we make sure that glibc is compiled with our fresh bootstrap compiler. Now we put our target after the –host flag, this is because we will not use glibc on our development host but on the target. The –host flag in this case has to point to the system on which the library will be used. We use no installation prefix so that the correct library path, the linker expects /lib and /usr/lib, will be hard-coded in the binaries, however we will not install it in that location. To make sure the linuxthreads add-on is also compiled we add the –enable-add-ons flag and the –with-headers flag tells the autotools which headers it should use. The path of these headers is the location where we have copied the header files earlier. This is necessary so that the target headers will be used instead of those of the development host. Optionally you can also use the –without-fpu option if the processor of your target has no floating point unit nor floating point emulation. Once all this is done we start the compilation:

```
$ make
$ make install_root=/usr/$target-linux prefix="" install
```

With install_root=/usr/$target-linux prefix=”” we install the binaries where we want them to be installed, in this case /usr/$target-linux/lib (lib is the standard sub-directory).

There is still a final step before glibc is finally correctly installed. We have to adapt the ***/libc.so*** file. Because we have installed the binaries in another path we have to correct this. When we open ***/usr/$target-linux/lib/libc.so*** we should see the following.

```
/* Gnu ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.            */
GROUP (/lib/libc.so.6 /lib/libc_nonshared.a)
```

Of course we first make a backup of the file, you never know if you might need it.

```
$ cd /usr/$target-linux/lib
$ cp libc.so libc.so.bak
```

Then we edit the script and remove the absolute path. By removing the absolute path we force the linker to use the libraries that are found in the same directory as the libc.so file. This way we are sure that the libraries for development host are not used. After we have edited the file it should look like this:

```
/* Gnu ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.            */
GROUP (/libc.so.6 libc_nonshared.a)
```

An alternative; Newlib.

Earlier Newlib was mentioned as an alternative C-library. Sometimes it is useful to use another library then glibc, because it is to big, compiles badly or is not

usable for your platform (like is the case with uClinux platforms). Here we can then use Newlib. The only different step is this one, for the rest of the installation procedure is identical.

First we make a new project-directory, which we will call build-newlib in the ~/toolchain folder. We also unpack the source code and do the configuration from the build-dir.

```
$ cd ~/toolchain
$ mkdir build-newlib
$ tar -xvzf newlib-<version>.tar.gz
$ cd build-newlib
$ ../newlib-<version>/configure –target=$target –prefix=/usr/$target-linux
$ make all && make install && make info && make install-info
```

When all this has completed without errors, newlib should be ready to use.

Another alternative; uClibc.

To compile programs in a simple and reliable way with uClibc it is recommended to use a toolchain that automatically will link your executables with this library. More details about uClibc can be found in the chapter "Building a root filesystem.

The procedure is quite similar to the one we use for a Glibc toolchain. There is an option that has a huge influence on the toolchain, and this is the softfloat option. This option is necessary for cpu's without FPU. Here the floating point calculations will be replaced by integer only routines that should generate the same result as the floating point ones. Especially this option makes the procedure very different.

With soft-float

The toolchain will be build like described earlier. When there is a directory change it is assumed we start from the project directory. Configuring the kernel headers and the compilation of the binutils happens in the same way as with Glibc. The only difference is the –without-float option when configuring the binutils.

The full compiler build.

Once we have an C-library we can build the full compiler with support for C and C++. As we have unpacked the source earlier and not contamined it by building the bootstrap compiler in his own directory we can skip this . We do have to reconfigure the source, this time from the build-gcc directory were we will build the final compiler.

```
$ cd ~/toolchain/build-gcc
$ ../gcc-<version>/configure --target=$target --prefix=/usr/$target-linux --enable-
languages=c,c++
```

Now we start as usual the last compilation.

```
$ make all
$ make install
```

When all this has completed we should have a fully functional cross-compiling toolchain on our development host. Sadly enough a successful build does not

always generate a fault-less toolchain. In most cases however it will work.

5.2.1. Bill Gatliffs build-crossgcc.sh script.

Actually this script is nothing more than an automation of the manual build. However there is support for different architectures. When nothing goes wrong you should have a fully functional toolchain at the end of the process.

To start we have to download the following sources from the internet:

- a version of the binutils
- a version of gcc
- a Linux kernel
- a version of glibc and the corresponding glibc-linuxthreads
- a version of newlib

We have to pay attention to the format of the source archives. The script expects *.tar.gz archives, if you cannot find a correct archive from the version you want to use it is not really a problem. Either you edit the corresponding lines in the script or you make a *.tar.gz archive from the source. All these archives have to be placed into a sub-directory, tars, of the project directory. We can also change that location in the script if necessary. When everything has been downloaded and placed correctly we should get the following; a project-directory, for example ~/toolchain, containing the script and a sub-directory tars containing the sources.

Also notice that we have to download both glibc and Newlib. This is because this script builds with glibc for one architecture and with Newlib for another,

Now we have to edit the script a little so it suits our needs. This is because such a script cannot magically guess were it can find everything it needs or which versions have to be used. The most important parameters are well commented at the beginning of the script. However a minimal knowledge of bash scripting is welcome.

First of all we have to adapt the variable of the source directory, if we placed the archives in another location than the one mentioned earlier. Then we have to specify the prefix, which is used to place the compiled binaries if compiled correctly. We also have to indicate what the versions are of the binutils, ...

The relevant lines are:

```
DEFAULT_TARDIR=~/tars
DEFAULT_PREFIX=/opt/billgatliff/H-i686-pc-linux-gnu

BINUTILSDISTO=binutils-2.11.2
GCCDISTO=gcc-2.95.3
LINUXDISTO=linux-2.4.3
GLIBCDISTO=glibc-2.2.2
GLIBCTHREADSDISTO=glibc-linuxthreads-2.2.2
NEWLIBDISTO=newlib-1.9.0
```

Once everything has been set up correctly we start the script in the project-directory. We are offered the following choices from which we choose our target.

```
$ cd ~/toolchain
$ sh build-crossgcc.sh

Linux crosscompiler building script,
revision : build-crossgcc.sh,v 1.14 2002/03/21 03:03:32 bgat Exp $

1) arm-elf                    5) m68k-coff           9) sh-elf
```

```
2) arm-linux                  6) m68k-elf           10) sh-hms
3) h8300-coff                 7) powerpc-linux
4) h8300-hitachi-hms          8) powerpc-eabi
Please select a target:
```

After the target selection the script extracts automatically the sources, creates the necessary directories, ...

BEWARE: Do not run this script as root, to avoid overwriting system critical files, unless you really know what you are doing (or what the script does). It is best to use a normal user account en to place the created files in a folder where that user has read and write permissions. This is the function of the DEFAULT_PREFIX line in the script.

See addendum for the script and some extra comments.

addendum will also be added later.

5.2.2. Dan Kegels crosstool.

Because the script of Bill Gatliff still needs some work and does not take into account eventually needed patches, Dan Kegel has expanded it carefully. After a while it has even become a complete kit.

Again the building procedure is quite similar to the manual method. To be sure it would not become one big and unmaintainable script it has been split up in different parts, and thus in different scripts and files.

The most important are:

- demo.sh : Here we find some tested compilation parameters. This is used to start the building process and to pass the needed parameters to the all.sh script.
- all.sh : This is in fact the most important script. Inhere is defined how and which steps should be undertaken en which auxiliary scripts will be called. This is decided on basis of the parameters passed on by the demo.sh script.
- getandpatch.sh : The name is quite self-explanatory. This script fetches all necessary source archives and patches from the internet, extracts everything and applies the patches. The patches are included in the crosstool package.
- crosstool.sh : The script that does the configuration and the building is this one, in here we can find the whole building process.
- ptx.sh : When we want to do some testing this script prepares a so called userland or minimal system for your target.
- crosstest.sh : To do the actual testing you can use this script. It generates, using still other scripts, a whole set of tests, which will be run on the target platform. These tests can take several hours and can be used to see if the toolchain generates correct code.
- gcc-<version>-glibc-<version>.dat : Contains the parameters needed for different gcc-glibc combinations.
- $target.dat : Contains the parameters for the different targets, like compiler flags, ...
- $target.config : These are the kernel makefiles for the different architectures needed to build the correct kernel header files.

Now if we want to build a toolchain without to much fuss, we download the

crosstool package from Dan Kegels website (http://www.kegel.com/crosstool) and decompress it in a project-directory. Then we adapt ***demo.sh*** to our needs. It comes down to finding the line containing the options we which to use for our architecture and ***gcc*** version. Then we remove the leading "#" to uncomment it. When we now call the ***demo.sh*** script it will start the ***all.sh*** script with the correct parameters and most important the –no-test option. This to avoid it also starts testing the toolchain afterwards. Normally you should have a fully functional toolchain some quite long time later. This of course depends on the speed of your internet connection, the availability of the servers, the build system,...

Afterwards the test can always be run if needed, but as these toolchains are in wide use and often tested by other people they should be ok, be forewarned as this takes some more time though.

See the addendum for the different scripts and some extra in-line comments

6. Building a root file system.

What we call the root file system is the place where all programs, libraries, files, ... are stored. Because it contains the “root” of the basic directory tree, we call this the root file system. This file system (independent from which file system type is used, like ext2, jffs, reiserfs, ...) is mounted when the Linux kernel has booted. From this point on the system loads its configuration and gives access to the userland programs. This also applies to embedded systems independently from the boot medium, be it flash, rom, disk, network, ... The boot medium depends on the application but does not change the boot procedure significantly.

6.1. Structure.

A GNU/Linux system has a directory tree which resembles closely the one found in a mainstream (commercial) Unix system. This also applies to an embedded GNU/Linux system. To have some form of standardisation of the root file system there is an “official” standard. This standard is called the File system Hierarchy Standard of FHS in short (the document describing this can be found at http://www.pathname.com/fhs/). These rules do also apply to an embedded system. They may not be mandatory, still we use this standard to make sure our system will be interoperable, and because these rules are used by the Debian distribution on which this framework is based. Because we have an embedded system, size is an important factor and for simplicity's sake we don't use all directory's described in the FHS. This makes it simpler to build the root file system or rootfs.

We use the following structure:

Directory	***Contents***
/bin	Contains most executables for user commands
/boot	Contains the files needed by an eventual boot-loader for booting the system.
/dev	Contains the so-called device files needed to communicate with the underlying hardware.
/etc	Configuration files and some start-up files.
/home	The main directory where all main user-directory's are created.
/lib	Contains the necessary libraries (like the C-library) and the kernel modules.
/mnt	Other file systems should be mounted here.
/opt	Contains all “optional” software not needed for a functional system.
/proc	A virtual file system for kernel and process information.
/root	The root users' directory.
/sbin	Contains all the executables for the administration commands.
/tmp	Location of the temporary files.
/usr	Contains the files and documentation for the users' applications.
/var	Data- en log files of daemons or other applications

It' s even possible still, to remove a few of these directories. Because of the multi-user structure of a Unix system there are a few directories you do not need on your embedded system. These are ***/home*** and ***/opt***. Eventually you can also remove ***/root***, but it is not recommended because often a process on your system will be run as root (as this is often the default login). As a result it is recommended not to remove this directory so that there is a directory where some data can be stored, and also it may make it easier to configure some login-programs. We could also remove ***/tmp*** and ***/var*** but this could severely limit the compatibility and functionality of several programs. So it is also recommended to keep those.

Removing directory's actually does not really influence size but has a great impact on functionality. The main advantage of removing empty directories is making it easier to have a good view on the whole system.

Also there are some directories which are used by some programs and that are sub-directories of ***/var*** or ***/usr***. They are also defined by the FHS and thus are named; ***/usr/bin***, ***/usr/sbin***, ***/usr/lib***, ***/var/log***, ***/var/lock***, ***/var/tmp*** and ***/var/run***. To avoid any nasty surprises these will also be added.

6.2. Manually building a root file system.

This is done in several steps. First of all we have to make a project-directory where we will keep our root file system for the time-being. After this we will choose the library and the programs that will be used on our platform. The most important in embedded use will be discussed here.

6.2.1. Making the project-directory and the rootfs structure.

First we create the project-dir, in the example we will use ~/rootfs. This directory will be build in the user's home directory.

```
$ mkdir ~/rootfs
```

In that directory we will make the necessary sub-directories like they will appear on our system. We keep in mind that ~/rootfs will be / once it will be mountd on the system. Of course we have decided which directories we will need.

```
$ cd ~/rootfs
$ mkdir bin boot dev etc lib proc root sbin tmp usr var
$ chmod 1777 tmp
```

The last command is to give special permissions in the ***tmp*** directory. We have enabled the so-called sticky bit. This bit makes sure that files created in that directory can only be deleted by the user who made them. This is to make sure that an application that uses the ***tmp*** directory cannot erase files from another running process.

Now we continue with adding some sub-directories in ***/usr*** and in ***/var***.

```
$ mkdir usr/bin usr/sbin
$ mkdir var/log var/lib var/lock var/run var/tmp
$ chmod 1777 var/tmp
```

An attentive reader noticed we also gave the same stick-bit treatment to ***var/tmp*** as we did with ***tmp***. This is because ***tmp*** and ***var/tmp*** have more or less the same function.

Now we have build a minimal file system, that meets the demands for using “normal” standardized applications on our embedded system. It could happen

nonetheless that an exotic application needs other directories. In that case they should be added. For more information we should then look in the program specific documentation.

6.2.2. Installing the libraries.

Some libraries and the installation procedure will be explained here. The first one is the biggest and most rich. Just because of its size it is usually not used in an embedded system. That is why there are a few alternatives like newlib, dietlibc and uclibc.

Glibc.

Most of the time the cross-compiling toolchains (like ours) are build with glibc. This makes that the library is already compiled for our target platform and installed on our development system.

The library files can be found then in ***/usr/<arch>-linux/lib*** (when following the correct naming scheme and FHS rules). The most important ones are ***libc.so.6*** and ***ld-linux.so***. The first one contains most of the C routines and the other makes sure all the functions are correctly linked.

For dynamically linked programs at least these files have to be copied into the ***/lib*** directory in your root file system. You could need others like ***libcrypt.so.1***, ***libpthread.so***,... . To find out which library files you need you can use the ***readelf*** utility. When cross-compiling you have to make sure you use the ***readelf*** utility from your cross-compiling toolchain and not the native one.

Using it for cross-compiling is fairly easy because the cross-compilers use it as default library ans thus automatically link your binaries to it. When compiling with the ***-static*** flag enabled you don't have to copy the library files because the needed functions are compiled into the binaries. Be forewarned though that when using glibc this will generate quite big binaries.

uClibc.

The main advantage of uClibc is its size. It is quite a bit smaller then Glibc, to illustrate this the following quote from the uClibc says it all :

> “uClibc and glibc have different goals. glibc strives for features and performance, and is targeted for desktops and servers with (these days) lots of resources. It also strives for ABI stability.
>
> On the other hand, the goal of uClibc is to provide as much functionality as possible in a small amount of space, and it is intended primarily for embedded use. It is also highly configurable in supported features, at the cost of ABI differences for different configurations. uClibc has been designed from the ground up to be a C library for embedded Linux. We don't need to worry about things like MS-DOS support, or BeOS, or AmigaOs, or any other system. This lets us cut out a lot of complexity and very carefully optimize for Linux.
>
> In other cases, uClibc leaves certain features (such as full C99 Math library support, wordexp, IPV6, and RPC support) disabled by default. Those features can be enabled for people that need them, but are otherwise disabled to save space.
>
> Some of the space savings in uClibc are obtained at the cost of performance, and some due to sacrificing features. Much of it comes from aggressive refactoring of code to eliminate redundancy. In regard to locale

> data, elimination of redundant data storage resulted in substantial space savings. The result is a libc that currently includes the features needed by nearly all applications and yet is considerably smaller than glibc. To compare "apples to apples", if you take uClibc and compile in locale data for about 170 UTF-8 locales, then uClibc will take up about 570k. If you take glibc and add in locale data for the same 170 UTF-8 locales, you will need over 30MB!!!
>
> The end result is a C library that will compile just about everything you throw at it, that looks like glibc to application programs when you compile, and is many times smaller."

Installation is quite straight-forward except for setting the paths correctly. This is done when configuring (menu-driven configuration) before the actual building.

After extracting the source, you have to configure it with a ncurses driven menu. Here you have to select the options you want and the target platform. After this you have to build it specifying the cross-compiler on the command-line.

```
$ tar -xvjf uClibc-<version>
$ cd uClibc-<version>
$ make menuconfig
$ make CROSS=<arch>-linux-
$ make install
```

As the uClibc wrappers have been removed from uClibc, you are obliged to build a new toolchain supporting uClibc or using their buildroot environment. They have been deprecated for the following reasons :

- The linker (ld) tries to find, like it should, all necessary libraries. But if it cannot find the right uClibc library, it will use the ones present on your host. This could be a problem because they could be from another C-library or a C-library compiled for another architecture (mostly your host which is different from your target when cross-compiling) and thus generate incompatibilities or non-working binaries.
- When gcc is built, for whatever perverse and twisted reason, libgcc is linked against glibc. The wrapper is generally able to produce working binaries when using the static libgcc.a, and when your application code only needs functions that are not also using glibc symbols and structures. But increasingly, gcc is being built and released using a shared libgcc. This is a good thing for many embedded systems, but you cannot use a shared libgcc that was compiled vs glibc with uClibc binaries.

The toolchain can be easily build with their toolchain building script, the same goes for their buildroot environment. You can also download such an environment that has already been fully build from the uClibc website.

The buildroot environment provides you with a root file system which contains most needed utilities to boot a working system and to start compiling. Unfortunately it is not possible to do cross-compiling in such an environment. Because it has been compiled for a certain architecture you cannot run the programs in your buildroot if it the architecture they have been compiled for is different from your host.

The only option left for cross-compiling is building a uClibc cross-compiler. You have to get the toolchain builder script from the uClibc CVS, then you have to edit the ***Makefile*** correctly, set your target path and build it with the help of ***make***.

Once you have a working uClibc toolchain you just have to use it like any other toolchain, specifying the compiler on the command line.

```
$ CC=”<arch>-uclibc-gcc”  ./configure
```

or

```
$ make CROSS=<arch>-uclibc- ...
```

To install the uClibc shared libraries you have to use exactly the same procedure as for glibc. Take care to use the ***<arch>-uclibc-readelf*** utility and not the native one.

Newlib.

Newlib is another much used library designed for use on embedded systems. Usually it is used together with eCos. ECos is an open source, royalty-free, real-time operating system intended for embedded applications. But being POSIX compliant it can also be used for linking Linux and other Unix-like binaries.

It's most important features are POSIX compliance (as mentioned earlier), the possibility to remove floating point support in a lot of functions (like in printf and a lot of mathematical functions thus speeding up your system) and, the most important one in our case, portability. This portability is achieved because newlib is build on a set of 17 stubs which hook into the execution environment. By hooking platform dependent code onto those stubs you can quite easily use this library on your platform. For example Newlib has a stub called __fork, but you don't have to adapt or hook onto it unless your application(s) use system() or fork().

Installation is not very complex. First of all you have to extract the source archive. Then configure the source with the correct parameters.

```
$ tar -xvjf newlib-<version>.tar.bz2
$ cd newlib-<version>
$ ./configure –host=<host> --target=<arch>
$ make
$ make install
```

Newlib will the build a static library only. It also supports shared libraries but only when compiled natively (for use on the system where it has been compiled on). As such we will only use it to compile our binaries statically. This is done as follows:

```
$ gcc -nostdlib -static <target_install_dir>/lib/crt0.o <progname.c> \
    -I <target_install_dir>/include -L \ <target_install_dir>/lib -lc -lm
```

Dietlibc.

Dietlibc also has the “size matters” approach. Only this library is explicitly designed to link statically.

This means you do not have to worry about installing your libraries on the target system. And because it was designed that way it should not generate very large binaries. Nonetheless there is an ***experimental*** version of dietlibc which supports dynamic linking.

The biggest drawback of this library is the fact that is does not support all commonly used functions. However the most important are supported. To be more specific it does not support regex nor locales.

On the other hand it warns for functions that generate bloat like printf, scanf, ... or the inclusion of big header files like stdio and for some potential security risks.

Using it is also very easy. When installing dietlibc, a wrapper is also installed. When calling gcc or a cross-compiler (commonly <arch>-linux-gcc) you just have to add “diet” on the command line or use CC=”diet gcc -nostdinc”. An example

for arm when running an ***autoconf ./configure*** command would give :

```
$ CC=”diet arm-linux-gcc -nostdinc” ./configure
```

Since there have been some changes when moving over to ***gcc*** version 3 and above some options like **"*diet -Os gcc...*"** give error messages about -malign-loops, -malign-jumps and -malign-functions being obsolete. This can be fixed by creating a file ~/.diet/gcc containing this line:

```
-Os -fomit-frame-pointer -falign-functions=0 -falign-jumps=0 -falign-loops=0  \
-mpreferred-stack-boundary=2
```

If you get this options not for “***diet -Os gcc”*** but for ***“diet -Os <arch>-linux-gcc”***, put this in ***~/.diet/<arch>-linux-gcc*** instead.

6.2.3. Installing applications.

Installing applications is quite easy. Once you have installed the correct libraries because you have compiled or plan to compile your applications with a shared library. You simply need to copy the correctly compiled applications into their correct location(s). Do not forget to check them out for library dependencies. After you have copied everything you need in your root file systems temporary directory the only thing you need to do next is making an image of it an upload it to your board.

7. What is Stag and what does it solve?

The Stag framework is not another piece of software. Instead of trying to re-invent the wheel the main focus was trying to re-use the existing ones, which have proven their reliability. In fact it is a new way of working.

The idea is to use the Debian package management system to hopefully provide a wealth of full cross-compilable source packages and tested binary packages. These binary packages have also been made following specific guidelines to be sure of cross-compatibility and a focus on small size.

To be able to make those packages fully cross-compilable an adapted version of the debhelper tools is used. The dpkg-cross debhelper-add-on already provides some support, but this is not complete. The Stag framework adds the necessary changes to be able to have full cross-compiler support. The settings like cross-compiler used, the library, interactive configuration (if applicable), ... can be configured by adapting the settings in one single file. By re-using all these existing tools and others like apt, dpkg, ... the framework provides a uniform way to build from the source without having to look at program-specific installation routines. This saves the developer time because he/she does not have to spend time searching workarounds to be able to build the software.

Also the framework supplies tested cross-compiler toolchains. These have been thoroughly tested, so the developer using them can be quite sure they are fully functional and be sure they will not link to development host specific code.

Dependencies are also checked by the means of the binary packages. By selecting all the needed packages at once you can be told which eventual program dependencies have to be resolved and you even have an approximate idea of your root-file systems size. Also the binary packages can be used to test the functionality of the system, before final set-up. This can be done by installing a minimum root file system with busybox, some init scripts, device files and a kernel in a separate directory on your development host with apt, after which it can be transferred to your embedded system for first tests.

When compiling from source the Debian package management system automatically builds a binary package which in the end results in a set of Debian packages, configured the way you want tailored for your application, making reinstalls and archiving easier.

There is one big drawback. Because of the multitude of libraries and architectures one source package can generate a lot of different binary packages. This is not actually the real problem, because this is actually a behaviour we want. But all these binary packages take up a lot of space (even small packages can do that if there are enough).

8. Using the GNU/Debian tools for embedded development.

8.1. Package management.

The Debian package management system can be used to generate the toolchain and the root file system.

The embed-series of packages have two main branches. First a source branch with all the cross-compilable source code inside. These packages can be recompiled with all the configuration options you would like to use. This will be explained later. The second branch is the binary branch.

8.1.1. Building a root file system from binary packages.

This one is sorted by architecture and is found in the repositories under their respective binary-<arch> directory. These binary packages contain a set of binaries, compiled with a set of options that ***should*** work with most embedded systems. If you have a fairly standard system with MMU , FPU and the glibc c-library you should be able to quickly build a root file system to test your system. Of course you also need a working linux kernel for this. (The source for some tested kernels can be found in the kernel-source-<arch>-<version>-embed packages.) The following procedure should be quite familiar for a Debian user.

First of all you need to find you a decent mirror near you. For the embedded packages this could be ftp://ftp.stag.mind.be for example. Once you added this mirror to your sources list, which can be done by editing ***/etc/apt/source.list*** by hand or using apt-setup, you have to update the apt database for your architecture.

```
$ apt-setup
$ apt-get update -o Apt::Architecture=<arch>
```

Once this has been done, you can start installing packages containing the necessary programs. Take care to select only packages with -embed in their name, otherwise you could overwrite some of the files on your system with the ones from a debian package for another architecture, which could result in a broken system. Therefore you use the following command.

```
$ apt-get install <program-name>-embed -o Apt::Architecture=<arch> \
  -o DPkg::Options::=”--force-architecture”
```

This will get the binary package containing the required program and install it onto your development system. Normally you should find the installed files in /var/rootfs-<arch>. Once you have installed all the needed programs you can generate a rootfs-image with the tools specific to your platform.

To restore the normal apt behaviour you just run:

```
$ apt-get update
```

This will update apt's database again with packages for your host and thus you will be able to use apt again for your GNU/Linux Debian system. The described procedure has to be used each time you want to install a package or series of packages into the rootfs. Because the packages install in a directory which contains the architecture name you can have different root file systems on your development host.

The advantage of selecting all needed packages at once is that dependencies are automatically resolved, and that apt also gives an approximate size. If you

install packages one by one you only have the size of that package and its eventual dependencies.

8.1.2. Building a rootfs from source packages.

This is a little more complex, but gives you a lot more control. There are two ways to build those source packages. You can treat them like normal sources and do all settings by hand, including the compiler options etc... Or you can use ***dpkg-buildpackage*** and build them the Debian way. Dpkg-cross provides a set of wrappers around some of the debian package building scripts (that are a part of the debhelper tools). With the help of these scripts you can build your own debian packages without to much trouble. For cross-compiling we have a few extra steps. First of all we need to choose a c-library and the cross-compiler toolchain. Once we know that we can start building from the source packages, we need to tell the Debian build scripts which parameters (compiler and compiler flags) have to be used.

System set-up.

First we need to have all necessary software on our system, this can easily be installed with ***apt-get***.

```
$ apt-get update
$ apt-get install debhelper dpkg-cross fakeroot stag-addons
```

To be sure we have the latest versions we make sure that our *apt* database is up-to-date, when that is done we will install some necessary software packages. These are ***debhelper***, which contains the scripts used to build and maintain a debian package, ***dpkg-cross***, which contains the add-ons for debhelper that make cross-compiling work, fakeroot, which offers a fake root user environment to build the package correctly, and stag-addons , that contains some extras and documentation needed to make full use of the Debian package management system for cross-compilation.

With these packages installed we have all the needed files and just have to configure our system so we use the correct compiler, the correct libraries, etc...

We specify these variables in the ***/etc/dpkg/cross-compile*** file.

```
#
# /etc/dpkg/cross-compile: configuration for dpkg-cross & Co.
#

# default architecture for dpkg-cross (to avoid always typing the -a option
# if you do cross installations only for one architecture)
#default_arch = m68k

#
# general section: paths of cross compiling environment
#
# you can set the following variables here:
#  crossprefix: prefix for cross compiling binaries; default: $(ARCH)-linux-
#  crossbase  : base prefix for the following; default: /usr/local
#  crossdir   : base directory for architecture; default:
#               $(CROSSBASE)/$(ARCH)-linux
#  crossbin   : dir for binaries; default: $(CROSSDIR)/bin
#  crosslib   : dir for libraries; default: $(CROSSDIR)/lib
#  crossinc   : dir for headers; default: $(CROSSDIR)/include
#  crossinfo  : dir dpkg-cross' package info files; default:
#               $(CROSSLIB)/dpkg-cross-info
#  maintainer : maintainer name to pass to original dpkg-buildpackage
#               in -m option. If not set at all, don't pass a -m, thus
```

```
#               dpkg-buildpackage will use the name from the changelog
#               file. If set to the special string CURRENTUSER,
#               dpkg-buildpackage will use the name from the
#               changelog, too, but signing the .changes will be done
#               as the current user (default key).
#
# Usually, you need only set crossbase, or maybe also crossdir
#
crossbase = /usr
crossdir = /usr/m68k-unknown-linux-gnu
crossprefix = m68k-unknown-linux-gnu-

# A crossroot definition is for the complete-Debian-system-mounted-somewhere
# approach, mainly used for Hurd.
#crossroot-hurd-i386 = /gnu

#
# This setting for maintainer is usually right:
#
maintainer = CURRENTUSER

#
# per-package sections: additional environment variables to set
#
# If you have additions here, please tell <roman@debian.org> so they
# can be included in the package.
#

#used for selecting configuration menu for stag.
config=yes

#tells which library is used by the cross-compiler when building the package :
#glibc,uclib,newlib,..
library = glibc

#set extra compiler flags, add necessary flags, otherwhise leave empty
#example for extraflags:  -nostdlib -static $(library_install_dir)/lib/crt0.o
progname.c -I #$(library_install_dir)/#include -L $(library_install_dir)/lib -lc
-lm
extraflags =

amd:
      SYSCC = $(ARCH)-linux-gcc

gs-aladdin:
# must be a native gcc
      CCAUX = gcc

# or, depend on binutils-multiarch
bison:
      STRIPPROG = hppa-linux-strip
```

With ***default_arch*** you can set the architecture for which you are going to compile. Take care though, because this setting will affect all cross-compiling that you will do on the host using ***dpkg-cross***. It does not apply here actually because we will not use existing Debian packages as starting point for cross-compilation.

A lot of ***cross*** variables can be set, but normally you should only need to set ***crossbase, crossdir*** and ***crossprefix***. In this case they have been set to an exotically named m68k cross-compiler.

The ***crossdir*** variable defines where all the needed binaries and libraries needed for the cross-compilation can be found.

Crossprefix is used to define the prefix for the utilities which will be used for stripping the binaries, ... for example: in this case we would like to use m68k-

unknown-linux-strip to perform the stripping task and not the strip binary used for i386 binaries (if the development host is an x86). This should only be used with a compiler that does not follow the Debian naming standard (<arch>-<os> , like arm-linux,...).

When we use the ***binutils-multiarch*** package from the standard Debian package list this problem is also solved and we do not have to use this variable. When working with exotic hardware or exotic toolchains it can be that the multi-architecture binutils are not working , do not contain the architecture you use or do not have special requirements and patches.

The ***config*** variable is used when building the sources with ***dpkg-buildpackage***. The ***rules*** file which is used to build the package tests this variable for a yes or a no to know if it has to call the configuration menu (if something like that is implemented for that package) or not. This has been done to comply with the Debian policy that asks that a package should be able to compile without user intervention.

The C-library problem.

When it is necessary to use another c-library, because of size or other functional limitations and/or requirements, this can be easily solved.

Most libraries provide wrappers so that you can easily compile your programs against that c-library. A very good example is the much used ***uclibc***-library. It provides a wrapper that can be used just like a compiler. In this case you just have to set your ***crossprefix*** and ***crossdir*** variable correctly.

For uclibc you will have to use a new toolchain because they recently deprecated their wrappers.

In the other case you have to figure out the correct compiler flags needed to force gcc to compile against the library you wish to use and not to use glibc. This can be done by setting the ***CFLAGS*** correctly. An example is given, this one is for ***newlib*** but can also be used for others. The ***-static*** flag is given because ***newlib*** does not have support for shared libs on architectures different from x86. This can also be used to set extra compiler flags that may be needed for your application.

The third possibility is to have or generate a new cross-compiling toolchain that has been compiled with your target's library. This has as effect that your cross-compiler will automatically link to the library he has been compiled with. Thus making it easier for you when cross-compiling. You still have to set the correct ***crossprefix*** and ***crossdir*** variables though.

Building the software and the package.

Building the software is relatively easy with the Debian tools, if you use a Debian source package of course. You will have to watch out for some “pitfalls” however. When downloading sources (with or without unpacking or building them) ***apt*** will install the source in the current directory. So it is recommended to use a separate directory for building the packages. It also keeps your development host tidy and all your sources organized.

First of all we will get the source packages. At the same time we can pass some flags to affect the behaviour of ***apt***.

```
$ apt-get source -d <package-name>-embed
```

This just downloads the package and puts it in your working directory

```
$ apt-get source <package-name>-embed -o Apt::architecture=<arch> \
```

```
    -o DPkg::Options::=”--force-architecture”
```

Here the package is downloaded and unpacked in the current dir.

```
$ apt-get source -b <package-name>-embed
```

This is the most interesting option, not only is the source package downloaded and unpacked but the build-also starts automatically, else it will just compile for the architecture you are building on.

If you already have downloaded the sources and unpacked them, or maybe you want to rebuild them you have to enter the source directory itself and let ***dpkg-buildpackage*** do the work for you. This tool will automatically build the source and the binary package or only the binary package.

```
$ cd <package-name>-embed-<version>
$ dpkg-buildpackage -b  -a<arch> -rfakeroot
```

The -b flag to ***dpkg-buildpackage*** is used to tell it to only build the binary package (as you already have the source package). With -a<arch> you tell it for which architecture to build. This flag is absolutely necessary. If you do not use it, the package will be build for your host and not even look at the config file (***/etc/dpkg/cross-compile***) for the compiler and all the other things you specified.

Also be sure to use the right <arch> because you could have unpredictable results when trying to cross-compile for one architecture using the wrong cross-compiler. Mostly it will just fail.

We use -rfakeroot so that we can correctly build the package as a user, dpkg-buildpackage then calls a fake root environment. This is necessary because some files will have to be installed in /usr later on.

If the building finishes correctly you should get a Debian package that you will have to install by hand. The targets architecture should automatically be appended at the end of the name. Be aware of the fact that it will have exactly the same name as the binary package that can be found in the repositories. A good practice would be to rename them. Installation is easy. Just do :

```
$ dpkg -i –force-architecture <package-name>
```

The -i flag means install, --force-architecture is necessary because dpkg detects that your host architecture is different from your targets architecture. The warning can be safely ignored because the -embed series of packages install in a directory and not like standard packages in the root of the system.

This behaviour is defined in the ***rules*** file in the source, or defined in the config.in. Be careful though, when you get a configurable menu, not to the change target directory (unless you know what you are doing) because you could overwrite system files when you install the package later.

9. Building Packages for the Stag framework.

9.1. Initial configuration.

First of all you should pick a program and test it thoroughly. Then proceed to packaging. This process is greatly identical to the process of making a Debian package. There are some differences however. I will use the busybox-static-embed package as an example.

Keep in mind though that, although it seems easy as it is described, it can be a very tedious task. It needs a lot of trail and error. Although once you get the hang of it, it becomes easier.

The best way to start is from fresh sources, so contamination from previous builds can be avoided. So we create a new directory in which we will decompress the source. Once this has been done we rename the source-tree to something more appropriate like <program-name>-embed. Then we will do the initial "Debianisation" of the source tree. If there is already a debian dir we have to remove it, else it will conflict with ***dh_make***.

```
$ cd /home
$ mkdir build
$ cd build
$ tar -xvzf busybox-1.00.tar.gz
$ mv busybox-1.00 busybox-static-embed-1.00
$ cd busybox-static-embed-1.00
$ rm debian
$ dh_make -e <maintainer-e-mail-address> -f ../busybox-1.00.tar.gz
```

Dh_make will ask some questions. We will not build multiple binary packages (nor library ???) nor kernel module. After this ***dh_make*** will ask for confirmation and create a source tarball which will be a part of the source package.

After this we have to edit the files in the debian directory. The function and syntax of the files has been explained in a previous chapter; "The Debian package structure".

9.2. Editing the rules file.

This could well be the hardest part, as it is a complex task. This file will also be different for each program you would like to package. There are some things that will have to be included nonetheless.

1. When your program has a configuration menu, style ***menuconfig*** you must add a test which checks for the ***$(CONFIG)*** variable. This comes from the ***/etc/dpkg/cross-compile*** file, if ***config=yes*** is set in that file you should call the configuration menu. Else you should use some default settings that work for you (and hopefully on a lot of other platforms too).
2. You ALWAYS need to ask for the ***DEB_HOST_GNU_TYPE*** and ***DEB_HOST_ARCH***, since those are used by ***debhelper*** and ***dpkg-cross***.
3. Also you need to export the ***$(LIBC)*** variable so you can use this to set the correct library name in de control-file (for name and dependencies) and in de package-name.
4. Dh_strip does not work with the cross-compiler tools so you should use dh_stripcross.
5. Because we do not want any documentation or man files on our embedded system we do not use : ***dh_installdocs***, ***dh_installman*** and ***dh_installinfo***.
6. Dh_shlibdeps tries to find the library dependencies, but because it cannot find

them for cross-compiled packages (it could be a toolchain /library that did not come in a Debian package,...) we do not use it.

7. Make sure that your program WILL install in /var/rootfs-<arch>. Therefore you have to make sure when you build the package, that the files which are created by the building process are installed under the debian/var/rootfs-<arch> subdirectory of your current building dir. If you fail to do so you could overwrite some system-critical files on your development host when installing the program.

The rest is up to you to sort out, because every program installation has its peculiarities. The following rules file illustrates this.

```
#!/usr/bin/make -f
# -*- makefile -*-
# Sample debian/rules that uses debhelper.
# GNU copyright 1997 to 1999 by Joey Hess.

# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

export DH_OPTIONS

DEB_HOST_GNU_TYPE ?=$(shell dpkg-architecture -qDEB_HOST_GNU_TYPE) #needed by the
debhelper and
DEB_HOST_ARCH ?=$(shell dpkg-architecture -qDEB_HOST_ARCH)         # dpkg-cross
tools

CFLAGS = -Wall -g

ifneq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
      CFLAGS += -O0
else
      CFLAGS += -O2
endif
ifeq (,$(findstring nostrip,$(DEB_BUILD_OPTIONS)))
      INSTALL_PROGRAM += -s
endif

configure: configure-stamp
configure-stamp:
      dh_testdir
      # Add here commands to configure the package.
ifeq ($(CONFIG),yes)                   #configuration selection test, if yes then
do make menuconfig else
      make menuconfig           #use the standard configuration.
else
      sed -e s/CROSS_COMPILER_PREFIX=/CROSS_COMPILER_PREFIX=\"$(CROSSPREFIX)\"/
debian/config-static > debian/config.temp
      sed -e s/EXTRA_CFLAGS_OPTIONS=\"\"/EXTRA_CFLAGS_OPTIONS=\"$(EXTRA_CFLAGS)\"/
debian/config.temp > debian/config.temp2
      sed -e s/rootfs-/rootfs-$(DEB_HOST_ARCH)/ debian/config.temp2 >
debian/config.temp3
      cp debian/config.temp3 .config         # a standard configuration file is
used, we have to set some variables
endif                                  # in there like the cross-compiler prefix,
extra compiler flags, the              touch configure-stamp                    #
target architecture and the correct installation path on the target
                                # system (set here because of a busybox
peculiarity)
build: build-stamp

build-stamp: configure-stamp
      dh_testdir

      # Add here commands to compile the package.
```

```
	$(MAKE) dep
	$(MAKE)

	touch build-stamp

clean:                  # is executed as first rule, we also set the control file
correctly
	sed -e s/xxx/$(LIBC)-static/ debian/control.in > debian/control.tmp
	      # we set the name and architecture
	sed -e s/arch/$(DEB_HOST_ARCH)/ debian/control.tmp > debian/control     #
correctly in the control file
	export LIBC="$(LIBC)"

	dh_testdir
	dh_testroot
	rm -f build-stamp configure-stamp

	# Add here commands to clean up after the build process.
	$(MAKE) distclean

	dh_clean

install: build

	dh_testdir
	dh_testroot
	dh_clean -k
	dh_installdirs

	# Add here commands to install the package into debian/busybox-embed. This
is VERY important, the files dont't    # get included in the binary package
otherwise. The correct installation path has already been set earlier, but
	# usually it should happen here.
	$(MAKE) install DESTDIR=$(CURDIR)/debian/busybox-$(LIBC)-static-embed


# Build architecture-independent files here.
binary-indep: build install
# We have nothing to do by default.

# Build architecture-dependent files here.
binary-arch: build install
	dh_testdir
	dh_testroot
#	dh_installchangelogs Changelog         # we don't want a changelog on our
target system
#	dh_installdocs                         # we don't want the documentation on
our target system
#	dh_installexamples              # no examples either
#	dh_install                      # installs files specified in
debian/packages.list (not used here)
#	dh_installmenu                         # this program does not generate any
menu-entries
#	dh_installdebconf               # we do not use the debconf configuration
program here
#	dh_installlogrotate             # we do not use the logrotate program, so
we don't need support for it
#	dh_installemacsen               # we have no emacs add-ons in this package,
so not needed
#	dh_installpam                          # this program does not use pam
(plugable authentication modules)
#	dh_installmime                         # this programdoesn't have support
for certain mime-types
#	dh_installinit                         # instals init scripts, because this
program has none, not used
#	dh_installcron                         # installs cron-entries, this program
has no periodic tasks so not used
```

```
#     dh_installinfo                          # we don't want the info pages on our
target system
#     dh_installman                           # we do not want the man pages on our
target system
      dh_link                                 # creates eventual needed symlinks
      dh_stripcross                           # strips you binaries with your
cross-compilers tools
      dh_compress                       # compresses all files of your program
following the Debian Policy
      dh_fixperms                       # fixes the permissions of your files
#     dh_perl                                 # checks for dependencies on perl
parts
#     dh_python                         # checks for dependencies on python parts
#     dh_makeshlibs                           # scans for shared libraries and
makes the shlibs file
#     cp debian/copyright $(CURDIR)/debian/busybox-$(LIBC)-embed/var/rootfs-
$(DEB_HOST_ARCH)/usr/share/doc/busybox-$(LIBC)-embed/            # could be used
to be fully compliant with the debian policy
      dh_installdeb                           # installs the files in the DEBIAN
dir (postrm, postinst,...)
#     dh_shlibdeps                      # tries to resolve library dependencies (DO
NOT use with stag)
      dh_gencontrol                           # generates and installs control file
      dh_md5sums                        # calculates and stores the md5sums for the
package
      dh_builddeb                       # generates the Debian package

binary: binary-indep binary-arch
.PHONY: build clean binary-indep binary-arch binary install configure
```

9.3. Editing the control file.

In the ***control*** file we have to edit the description and the long description. Also we have to check if the package name, the branch, the section and the priority are correct. Some measures have to be taken to make it possible to compile the package with different libraries and for different architectures. Therefore you have to define some strings that you can easily replace through the ***rules*** file. Mandatory is the library in the name and the architecture. The best practice is to use a control.in file that will not change, but that will be used to generate a new control file. Also do not forget to fill in the dependencies!

```
Source: busybox-static-embed
Section: devel
Priority: optional
Maintainer: Philippe De Swert <philippedeswert@hotmail.com>
Build-Depends: debhelper (>= 4.0.0)
Standards-Version: 3.6.0

Package: busybox-xxx-embed            # the xxx should be replaced by the correct
library name (in the rules file)
Architecture: arch                    # this arch string can easily be
replaced by the correct arch using the
      # $(DEB_HOST_ARCH) variable (listed in the rules file)
Depends:                        # fill in eventual dependencies. In this case it
is  statically linked so it                                          # does not
depend on any library, nor does it depend on other programs
Description: Tiny utilities for small and embedded systems.
 BusyBox combines tiny versions of many common UNIX utilities into a single
 small executable. It provides minimalist replacements for the most common
 utilities you would usually find on your desktop system (i.e., ls, cp, mv,
 mount, tar, etc.).  The utilities in BusyBox generally have fewer options than
 their full-featured GNU cousins; however, the options that are included
 provide the expected functionality and behave very much like their GNU
```

```
counterparts.
This version is meant to test and boot your embedded system for the first time.
```

9.4. A standard configuration file.

In most cases you will need a standard configuration file because most programs rely on the autotools, while others may have a different system to include or exclude several options. It is important to use one because the Debian Policy Manual states : “Since an interactive ***debian/rules*** script makes it impossible to auto-compile that package and also makes it hard for other people to reproduce the same binary package, all required targets MUST be non-interactive. At a minimum, required targets are the ones called by ***dpkg-buildpackage***, namely, clean, binary, binary-arch, binary-indep, and build. It also follows that any target that these targets depend on must also be non-interactive.”. This way your package does surely compile in one way, and can be re-compiled for other architectures in an automated way.

You have to be careful when making your standard configuration file.

First of all you should exclude all platform-specific stuff to make sure it compiles for different architectures.

Secondly you have to make sure you can easily replace some variables like cross-compiler prefix, specific compiler flags or installation dir through the ***/debian/rules*** file. Like we did in the example earlier.

9.5. All the other files.

Still there are some more or less important files left. There are the files that contain the documentation and others that deal with eventual installation or removal problems.

Documentation:

- It is important to include a ***changelog*** file in your source-package with information about specific platform-dependent issues and or bugfixes.
- Also the ***copyright*** file should be included here.
- Any other useful information should be written down in the ***README.debian*** file.

Dealing with installation or removal specific problems:

- post-install: This is the most recurring event, deal with this by means of the post-inst script.
- post-remove: When some files have to be removed or adapted after de-installation you have to handle it by using the post-rm script.
- pre-install: This rarely occurs but if it does handle it through the pre-inst script.
- pre-remove: If something needs to be checked out before it is removed do it with the pre-rm script.

All these files already have been discussed in a previous chapter where the whole Debian package is explained.

9.6. Finally building the package itself.

Once you (think) you have correctly edited all the files in the ***debian***

subdirectory of your source tree, you can build the package. First of all you have to know which cross-compiler you are going to use, which library you are going to link to,... Once you know all this, edit the ***/etc/dpkg/cross-compile*** file accordingly. In the chapter:" Using the GNU/Debian tools for embedded development." the syntax has already been explained. As this file is going to be used to find out the specific compiling parameters it is important to do so.

The building itself should be quite easy. You have to position yourself in the root directory of your build, and issue the following commands:

```
$ cd /<compilation-dir>/<program-name>-embed-<version>
$ dpkg-buildpackage -rfakeroot -a<arch>
```

This will automatically build the source-package and the binary package. Watch out for errors here, because this could point out some faulty configuration options.

The ***<arch>*** flag should be in concordance with what you have written in your ***/etc/dpkg/cross-compile*** file. There is no check to look if your cross-compiler or cross-compiler prefix match with the architecture you specified on the command line.

When build has successfully ended it will ask you for your PGP passphrase, if you have one or specified it as an extra parameter on the command line with ***-k<keyid>***. This is not needed though, you can perfectly build Debian packages without a key, but these will not be accepted for upload on a Debian repository.

Once the build has completed you should check out your package. There are several ways, but there is one I strongly advise against. DO NOT install your package as a test unless you are sure it installs in the right place. You could overwrite system files (critical or non-critical) on your development host.

I advise to either unpack the Debian package in another directory or to check its contents with the help of a package manager (for example ***dpkg, kpackage****,...)* or others which allow you to view its contents (I like to use ***mc*** in this context).

If it compiles right, and installs correctly on your system you can try to use it on an embedded platform to check if the binary package really does work. If it does and you have build a mainstream package you could upload it to a Debian repository. To be able to do uploads to a Debian repository you have to be a Debian maintainer, and this is a whole other story. For the time being Stag packages are not admitted to Debian repository's because they are not fully (but very close to being) compliant with the Debian Policy. So find an unofficial repository where other Stag Debian packages are stored.