Philippe De Swert, <philippe.deswert@student.denayer.wenk.be>
version 1.2

Stag framework quick start guide.

Getting started with Stag is quite easy. You just have to follow the following steps. Making new Stag packages is the most difficult, and as such is only briefly described in this document. However experienced Debian packagers should have no problem creating them. More documentation can be found in the full doumentation.

Installation of the Stag framework tools.

It is very easy when you have a Debian distribution, however the package management tools are also ported to “normal” rpm based distributions. I do not have any experience with this so you will have to try by yourself and look into your distributions documentation.
Mostly these are the normal Debian tools. However as they are not complete there are some changes. Maybe they will be admitted in the standard Debian packages, but for the moment we have a separate stagg-addons package. This package depends on dpkg-cross, debhelper and fakeroot, but normally they should be installed together by the package management system, once there is a working repository. Then this should be sufficient:

```
$ dpkg -i –force-all stag-addons-1.0.2.deb
```

Take care to do this when using dpkg-cross 1.14. The 1.16 version has the Stag/Emdebian changes added but this is still in development and not fully functional.

Using the Stag framework.

Using Stag binary packages.

When you plan to only use binary packages from the Stag framework you can skip the installation.
Installation from a repository should be easy using apt. It comes down to this:

```
$ stag-get <arch> update
$ stag-get <arch> install <package-name>
```

If you downloaded the package you can install it with:

```
$ dpkg-stag arm -i
```

Unfortunately there are no repositories available for the moment.

Using Stag/Emdebia source packages.

This is somewhat more complex but still quite easy. First thing to to is edit the /etc/dpkg/cross-compile file. You will have to change the following:

- You have to set the crossbase, which is the prefix of the directory where your toolchain has been installed. In Debian this should usually be /usr, but can also be /usr/local as most toolchain scripts place the compiled toolchains in there.
- You also have to set the crossdir, which is the root directory of your toolchain. It should be something like /usr/<arch>-linux.
- Crossprefix has only to be set when using a compiler which does not use the standard <Arch>-linux- naming convention. An example could be arm-uclibc-linux-.
- If you want to do some configuration be hand you have to set config to yes. If set to yes the rules in the source package are so defined it will call the configuration menu if there is one.
- You will have to set the library you use. This is for naming conventions and dependencies only, but is nonetheless quite useful.
- The extraflags are usually not needed, but if you want to link to another library than the library of your toolchain. Or use other optimalisation levels or compiler flags you should specify this here.

You should end up with something like this:

```
#
# /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 = /home/fille/toolchain/gcc-3.3.x/toolchain_arm_nofpu
#crossbase = /usr
crossdir = /usr/arm-linux

#crossprefix = When the prefix is standard <arch>-<os> (like arm-linux-) DO NOT USE THIS!!!
#crossprefix = arm-linux-uclibc-

# 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.
#

#sets manual or automatic configuration when compiling source: yes/no
config = no

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

#set extra compiler flags, add necessary flags, otherwhise leave empty
extraflags =
#-msoft-float


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

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

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

Once this is done you can download the source packages, decompress and build it.

```
$ dpkg-source -x <package-name>.dsc
```

```
$ cd <package-name>
$ dpkg-buildpackage -b -a<arch> -Emdb -D
```

When building is complete you get a binary package you will have to installation like a normal binary package with dpkg-stag.

```
dpkg-stag <archi> -i <package-name>
```

For the moment only the -i and -r (to remove) are fully supported!

What has been installed and where?

When installing the binary packages a directory is created in your /var dir. Inhere you will find a rootfs-<arch> dir. (Unless you specify another location by changing the ROOT= option in the dpkg-stag wrapper)This directory contains all the files you installed, and this is also your root file system.

To create a minimal root file system you should at least install init-dev-files and a busybox package (static or with its library). To all this you should add a working kernel.
Because this is quite architecture depend I cannot give more details here. I used it this way on “my” cerfboard (strongarm cpu):

I install some Stag/Emdebian packages and make an image of the rootfs-arm directory.

```
$ genromfs -d /var/rootfs-arm -f /tftboot/romfs.img
```

I start the board which starts with redboot. I load over tftp the romfs.img file at an memory address I specified in the kernel set-up (initrd), I then also load the kernel in memory. Then I start the kernel which will mount the image in memory as root file system and boot from there.

Keep in mind that your approach may be totally different!!!!

Building Stag/Emdebian packages.

If it does not already exists of course. This is only a limited introduction. It is possible you will not find everything in here.

Get a regular Debian package.

```
$ apt-get update
```

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

This will extract the source package in your CURRENT working directory.
We make a new emdebian directory in the source directory. In directory we will place all files needed by the Debian package management system. The contents of this directory will be quite similar to the normal Debian sub-directory.

Editing the *emdebian/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. Luckiley most of the time it is almost similar to the *debian/rules* file.

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_embed*.
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. All other debhelper scripts should be replaced by their *_embed* version.
8. If *install -s* is used, remove the -s option and add *dh_stripcross_embed* at the end of the build. This to strip the binaries at the end of the build with the correct strip binary instead of the one of the development system.
9. The compiler flags should be set to -Wall -Os (to optimize for size).

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 -Os

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_embed
	dh_testroot_embed
#	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_embed				# creates eventual needed symlinks
	dh_stripcross_embed			# strips you binaries with your cross-compilers tools
	dh_compress_embed			# compresses all files of your program following the Debian Policy
	dh_fixperms_embed			# 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
	dh_installdeb_embed			# installs the files in the DEBIAN dir (postrm, postinst,...)
#	dh_shlibdeps				# tries to resolve library dependencies (DO NOT use with stag)
	dh_gencontrol_embed			# generates and installs control file
	dh_md5sums_embed			# calculates and stores the md5sums for the package
	dh_builddeb_embed			# generates the Debian package

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

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.
```

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.

All the other files.

These should be symlinks to the existing files in the debian subdirectory. How we will treat postinst, postrm and other scripts should still has to be decided, so we do not use them for the time being.
