

Emdebsys - The easy way to make Embedded Distributions

Wookey

wookey@aleph1.co.uk

1. Embedded Debian project - Background

Lots of people working on small systems use Debian in one way or another, particularly due to its comprehensive multiple-architecture support. That's why I use it, and of course got sucked in and becoming a developer. The Debian infrastructure is also very useful for a small company or enthused individual as it allows you to make you own specialised changes and produce your own distro. All the mechanisms and tools are there and exposed for you to use.

Frank Smith, of Amirix, was one of those people, and he decided to set up a project for like-minded types to get together, compare notes and produce useful things. It was originally envisaged as a matter of documenting what people were doing to produce a 'how to make small distributions from Debian' manual, but that bit never really took off, and in fact what it has been (so far) is a source of cross-compilers and a development project for the Emdebsys tool.

Emdebsys started life as CML2+OS, but that's a bit of a mouthful and we decided to change the name to something slightly more descriptive (but still hard type :-).

After setting things up and doing some great development on the toolchain and CML2+OS Amirix was doing stuff that didn't really coincide with Emdebian any more and things got very quiet for a while. So I volunteered to be figurehead and jump up and down about what a good idea all this stuff is in the hope that some competent people will actually make it happen. Michael Vogt (another Debian developer) got enthused towards the end of 2001 and did some work to update it to a current version of

CML2 and make it work again for i386, and then Tomek (paid for by Aleph One because we think it's a project worthy of support) made it work on ARM too.

So - that's the current state of affairs. Emdebian is kind of pumping along as a project with not much happening from month to month, although writing this talk I've done some more working on getting emdebsys up to scratch again.

It's possible that Emdebian simply isn't a popular idea, but I think it ought to be. Debian still makes sense and lots of people use it. And emdebsys is a neat idea that needs field testing to see how it works out in practice.

I hope that bringing it to a wider audience will generate some more enthusiasm from those with an interest in the development or at least the use of the tools. Any enthused or interested people would be most welcome.

2. Emdebsys Principles

CML2 is a configuration language, originally designed to replace the kernel configuration system 'CML1', but in fact it's more general than that. The idea for emdebsys grew out of trying to write some hacky little makefiles to generate a filesystem image that had the right things in it. Frank realised that much of the dependency information necessary was in the kernel make files, and that this idea could be expanded into having a set of dependency rules for both the kernel and the OS combined so you could generate both.

CML2 rulesets define relationships between components as well as menu lists and constraints. We'll look at some examples in a while.

The final result of using the tool will be a filesystem and a kernel. The kernel may actually be configured to end up inside the filesystem, in which case you only get a filesystem out the end.

The inputs to the system are:

- Kernel configuration
- OS configuration
- package archive
- kernel source
- application source

First let's go through the process of using emdebsys so we know what we're talking about.

First run **cmlcompile.py kernel+os-config.cml** this generates rules.out from top-level kernel-tree/rules.cml and os-tree/rules.cml these include other rules files. The kernel-tree rules are generated from the real kernel source, and would be in your actual source tree if using CML for normal kernel configuration too. Rules.out is a binary file in CML format.

Next do **cmlconfigure.py** this runs curses-based front-end - the equivalent of **make menu-config** The equivalents of **make config** and **make xconfig** also exist (**cmlconfigure -t** (for text) and **-x** for (X) - the latter uses the tkinter library here you can configure the kernel options in the usual way, but also the os options you need and the emdebsys options for building the system

- does the kernel go in the root fs or not
- paths to kernel, name of this config setup
- type of root filesystem
- utils wanted in the OS - busybox, uclibc, Debian binaries

All this stuff is configurable of course. Change the rules files and you get different options.

This generates the config.out file which is just like the .config files we are familiar with from kernel builds:

```
#
# Kernel+OS
#
CONFIG_OS_KERNEL_ON_ROOTFS=n
CONFIG_OS_KERNEL_PATH="/usr/src/arm/linux-2.4.16-rmk2"
CONFIG_BUILD_ROOTFS=y
CONFIG_OS_DEBUG=n

#
# Linux Kernel Configuration System
#

#
# Processor type
#
```

```
CONFIG_X86=n
CONFIG_ALPHA=n
CONFIG_SPARC32=n
CONFIG_SPARC64=n
CONFIG_ARM=y
....
#
# Text Utilities
#
CONFIG_OS_CAT=y
CONFIG_OS_CKSUM=n
CONFIG_OS_COMM=n
CONFIG_OS_CSPLIT=n
CONFIG_OS_CUT=n
#
# Configure Snarf n Pick OS generator
#
CONFIG_SNP_CONF=y
CONFIG_SNP_cache_dir="snp-cache/"
CONFIG_SNP_cdrom_dev="/dev/cdrom"
CONFIG_SNP_cdrom_dir="/cdrom"
CONFIG_SNP_config_file="config.out"
CONFIG_SNP_cross_compile="/usr/bin/arm-linux-"
CONFIG_SNP_devices_file="devices.conf"
CONFIG_SNP_objdump="/usr/bin/arm-linux-objdump"
CONFIG_SNP_rootfs_dir="rootfs/"
```

Now we install CML2 into the kernel tree with `./install.cml2 <treeroot>`

(by default it will search the locate database for kernel trees so it might find yours itself, but for non-host trees which I for example put in `usr/src/arm/linux-<version>` you'll have to tell it. now copy the generated `config.out` into your kernel tree and into the `snp` dir so that it can be used as the basis for both kernel builds and snarf and picking. (This bit is a tad naff and could use tidying up so the right config files are found automatically)

Now build the kernel in the usual way - `make dep`, `make zImage`, `make modules` then run the **snp.py** program to take all the various components (kernel, modules, source, Debian packages) and generates the finished filesystem.

3. So how does it actually work

Now that skipped over a lot of sordid details, and of course sordid details is what we are interested in here.

So - some config files:

`sources.list`

describes the set of Debian archives you have using the same syntax as an `apt/sources.list` file

```
deb http://ftp.uk.debian.org/debian testing main contrib non-free
or a local mirror:
deb file:/cdimages/mirror/debian testing main
```

`sources.conf`

maps CML2 items that should be built from source to the corresponding tarballs and their final binary name in the root filesystem, e.g. `OS_BUSYBOX_SRC busybox-0.60.2.tar.gz /bin/busybox`

`snp.conf`

contains the mapping for what each emdebsys config option generates in terms of a binary:

```
OS_CHGRP fileutils /bin/chgrp
```

which is `CONFIG` option, package name and final binary name in the root filesystem. This file is generated automatically from the Debian `Contents-i386` file and a template, so it can easily be kept up to date.

`devices.conf`

Lists the required devices and their configurations like this:

```
SOUND audio
```

or

```
SERIAL_SA1100 ttySA0 c 204 5
```

The first field is the `CONFIG_` option it depends on. The single parameter type is created using `MAKEDEV`, the second using `mknod`.

Emdebsys also lets you configure User-mode Linux options, so you have an easy way to test out the finished result on your development machine (if you are not cross-compiling).

4. CML2

Lets take a look at how CML2 works, as that underpins the whole scheme.

All the CML rules are arranged in a hierarchy with a `rules.cml` file and a `symbols.cml` file in each directory. Each `rules.cml` starts with:

```
source "symbols.cml"
```

which includes the relevant symbol file.

Arranging things in this hierarchical manner keeps things reasonably well-ordered and modular. The current tree looks like this:

```
os-tree--
```

```
  |- snp
  |
  |--busybox
  |
  |--deb
  |
  |--tinylogin
  |
  |--uclibc
```

```
kernel-tree--
```

```
  |
  |--arch
  |
  |--drivers
  |
  : (etc)
```

Each symbols file simply contains mappings between symbols and their corresponding text:

symbols

OS_NETWORKING 'networking: enable networking'

OS_HOST_IP 'Host IP Address'

OS_HOST_NAME 'Host Name' and for menu items

menus

netcfg 'Network Configuration'

and for explanations

explanations

os_uclibc_has_mmu 'BB_DPKG_DEB Uses gz_open(), which uses fork()

BB_FEATURE_TAR_GZIP Uses fork()

Which are not supported without memory management unit in uClibc.'

The rules files specify the interactions and dependencies between these symbols:

```
menu netcfg # Network Configuration
```

```
    OS_NETWORKING
```

```
{
```

```
OS_HOST_IP$
```

```
OS_HOST_NAMES$
```

```
}
```

```
unless OS_NETWORKING suppress
```

```
    OS_HOST_IP
```

```
    OS_HOST_NAME
```

```
default OS_HOST_IP from "192.168.100.100"
```

```
default OS_HOST_NAME from "emdebian"
```

This is pretty self-explanatory stuff. First a menu is defined, entitled 'Network Configuration'. The first entry is 'networking: enable networking', and if that is selected then a submenu containing 'Host IP address' and 'Host Name' is revealed. The interface lets you assign values to all these symbols.

Default initial values can be specified and complex rules to constrain symbols and values and their visibility and to generate new symbols. We won't go into this in too much detail here but the other important directives in CML2 are:

- derive
- require
- unless
- implies
- supress

5. Background and credits

This article is based on a chapter of the second edition of the Aleph One *Guide to ARMLinux for Developers* book. The first edition is available now and the second edition will be out later this year. Please refer to <http://www.aleph1.co.uk/armlinux/thebook.html> for further information.

Thanks to Tomek Religa for his advice on this article.