The OpenWrt embedded development framework

Florian Fainelli - florian@openwrt.org

January 22, 2008

Abstract

One of the biggest challenges to getting started with embedded devices is that you cannot just install a copy of Linux and expect to be able to compile a firmware. Even if you did remember to install a compiler and every development tool offered, you still would not have the basic set of tools needed to produce a firmware image. The embedded device represents an entirely new hardware platform, so in a process called cross compiling you need to produce a new compiler capable of generating code for your embedded platform, and then use it to compile a basic Linux distribution to run on your device.

The process of creating a cross compiler can be tricky and in many cases when you are dealing with embedded devices you will be provided with a binary copy of a compiler and basic libraries rather than instructions for creating your own. Likewise, it is also common to be provided with a patched copy of the Linux kernel from the board or chip vendor, but this is also dated and it can be difficult to spot exactly what has been modified to make the kernel run on the embedded platform.

OpenWrt has different approach to building a firmware; downloading, patching and compiling everything from scratch, including the cross compiler. OpenWrt does not contain any executables and only very few sources, it is an automated system for downloading the sources, patching them to work with the given platform and compiling them correctly for that platform. What this means is that just by changing the templates, you can change any step in the process.

Contents

1 Design

1.1 Directory structure

There are four key directories in the base:

- `tools`
- `toolchain`
- `package`
- `target`

`tools` and `toolchain` refer to common tools which will be used to build the firmware image, the compiler, and the C library. The result of this is three new directories, `build_dir/host`, which is a temporary directory for building the target independent tools, `build_dir/toolchain-<arch>*` which is used for building the toolchain for a specific architecture, and `staging_dir/toolchain-<arch>*` where the resulting toolchain is installed. You will not need to do anything with the toolchain directory unless you intend to add a new version of one of the components above.

- `build_dir/host`
- `build_dir/toolchain-<arch>*`

1.2 Packages and external repositories

`package` is for exactly that – packages. In an OpenWrt firmware, almost everything is an `.ipk`, a software package which can be added to the firmware to provide new features or removed to save space. Note that packages are also maintained outside of the main trunk and can be obtained from subversion using the package feeds system:

```
$ ./scripts/feeds update
```

Those packages can be used to extend the functionality of the build system and need to be symlinked into the main trunk. Once you do that, the packages will show up in the menu for configuration. From kamikaze you would do something like this:

```
$ ./scripts/feeds search nmap
Search results in feed 'packages':
nmap       Network exploration and/or security auditing utility

$ ./scripts/feeds install nmap
```

To include all packages, issue the following command:

```
$ make package/symlinks
```

`target` refers to the embedded platform, this contains items which are specific to a specific embedded platform. Of particular interest here is the "`target/linux`" directory which is broken down by platform <*arch*> and contains the patches to the kernel, profile config, for a particular platform. There's also the "`target/image`" directory which describes how to package a firmware for a specific platform.

Both the target and package steps will use the directory "`build_dir/<arch>`" as a temporary directory for compiling. Additionally, anything downloaded by the toolchain, target or package steps will be placed in the "`dl`" directory.

- `build_dir/<arch>`
- `dl`

1.3 Toolchain

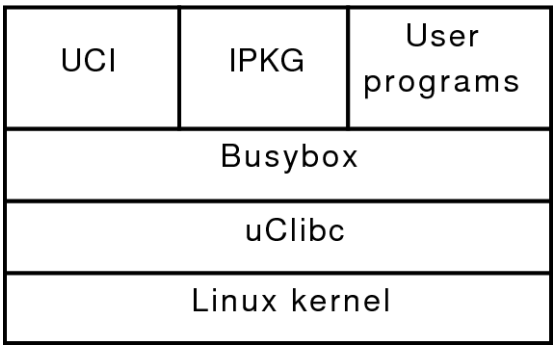
OpenWrt automates the toolchain creation for your particular architecture by passing the right arguments to the compiler during the cross-compilation process and using patches known to be working for your target architecture.

It also allows you to switch between different combinations of gcc, binutils, kernel headers and uClibc so that you can easily check regressions and programs compilation. You can even customize the compiler flags passed to your cross-compiler so you can test optimizations features of your cross-compiler or different locations for your cross-compiled libraries.

If any of the toolchain component is not present adding a new version is easy and can be easily done by adding an entry in the toolchain/Config.in and creating a subdirectory for it to include specific patches if any.

1.4 Software architecture

OpenWrt uses the common embedded Linux tools such as uClibc, busybox, shell interpreter and provides a hardware abstraction layer and package manager. Here is the software stack that OpenWrt uses:



Every architecture uses a different Linux kernel allowing the user-space environnment to be shared and consistent acrross devices. Therefore you only need to recompile the uClibc and packages to match your target architecture to get the same programs running on a totally different embedded device.

1.5 System and package configuration

UCI which stands for Unified Configuration Interface is a C library which provides configuration context for user space and system configuration and management. UCI was adopted with the extent of OpenWrt to other devices which did not have the NVRAM to store their settings into a separate flash partition.

Since UCI is a C library, it can be easily integrated into an existing user-space application or to develop a configuration storage that is OpenWrt compatible for your new application.

Further developments for UCI include a web interface that uses UCI as a configuration file format as well as SNMP plugins to easily change the configuration and take actions on the embedded device.

For instance adding a new configuration file is as simple as creating a new file in **/etc/config/package** which should contain the following lines:

```
config      <type> ["<name>"]      # Section
    option  <name> "<value>"       # Option
```

Later on, the system scripts and the UCI library allows you to parse this configuration context from either an init script or directly an user-space program.

2 Developing with OpenWrt

Developing with OpenWrt is easy and allows you to be really flexible with kernel, C library and user-space programs development.

2.1 Creating packages

One of the things that we've attempted to do with OpenWrt's template system is make it incredibly easy to port software to OpenWrt. If you look at a typical package directory in OpenWrt you'll find two things:

- `package/<name>/Makefile`
- `package/<name>/patches`
- `package/<name>/files`

The patches directory is optional and typically contains bug fixes or optimizations to reduce the size of the executable. The package makefile is the important item, provides the steps actually needed to download and compile the package.

The files directory is also optional and typically contains package specific startup scripts or default configuration files that can be used out of the box with OpenWrt.

After you have created your `package/<name>/Makefile`, the new package will automatically show in the menu the next time you run "make menuconfig" and if selected will be built automatically the next time "`make`" is run.

2.1.1 Package source download

One of the cool things with OpenWrt is that it supports different fetching methods, allowing you to retrieve the source of a package from various fetching methods such as:

- GIT
- Subversion
- CVS
- HTTP
- local source

For instance, if you wish to checkout a particular revision of a package using Subversion, just define the following download method in your package Makefile:

```
PKG_VER:=963
PKG_BRANCH:=batman-adv-userspace
PKG_VERSION:=r\$(PKG\_REV)

PKG_SOURCE_PROTO:=svn
PKG_SOURCE_URL:=http://downloads.open-mesh.net/svn/batman/trunk/
```

It then becomes very easy to test development snapshots and branches of your application.

2.1.2 Using another package manager

OpenWrt assumes you use IPKG as the default package manager, but extending the build system to generate RPMs, DEBs or other package system could be easily achieved by tweaking the package templates in `include/package*.mk`.

2.2 Creating kernel modules packages

The OpenWrt distribution makes the distinction between two kind of kernel modules, those coming along with the mainline kernel, and the others available as a separate project. We will see later that a common template is used for both of them.

For kernel modules that are part of the mainline kernel source, the makefiles are located in *package/kernel/modules/*.mk* and they appear under the section "Kernel modules"

For external kernel modules, you can add them to the build system just like if they were software packages by defining a KernelPackage section in the package makefile.

After you have created your `package/kernel/modules/<name>.mk`, the new kernel modules package will automatically show in the menu under "Kernel modules" next time you run "make menuconfig" and if selected will be built automatically the next time "`make`" is run.

Provided that you gave the right KCONFIG variables, your kernel configuration will be updated accordingly and the kernel module will be built only if the corresponding module is selected. It is also possible to make the packaged modules be loaded at boot time simply by defining it the Makefile.

2.3 Adding support for a new target

OpenWrt is very convenient when it comes to adding support for a new target and requires few steps to be quickly operationnal. You will have to create a new directory under `target/linux/<my target>` which will contain a Makefile defining its build system features and the kernel version to use.

Adding custom patches against a particular kernel version can be done by putting the patches into the `target/linux/<my target>//patches` directory wich an index number to let them be applied into the right order.

One of the interesting feature is that you can also put files that will be copied to the Linux kernel build directory. For instance, if you are developing a new file which should be into `arch/mips/<my target>/` in the Linux kernel directory, just place this file in `target/linux/<my target>//files/arch/mips/<my target>/file.c` and it will be copied to the Linux kernel.

This feature is really interesting when you are developing because you prefer editing C files instead of patches directly that will be creating C files.

2.3.1 Using quilt

OpenWrt natively supports quilt so that you can easily create and rebase your existing patches without the need to develop scsript to patch the sources up to a certain patch number, and then let you in with an editor.

Using quilt is very easy and allows you to edit the source files and generate the differences to produce a new patch with the right patch level. This patch is then automatically placed into the right location under the OpenWrt build

tree, such that you only need to issue the `make component/subcomponent/compile` command to get it recompiled with your previously edited patch.

2.3.2 Building an external kernel tree

OpenWrt recently added support for building an external kernel build tree wich allows you to use the cross-compiler created by the OpenWrt framework with, for instance a GIT snapshot of a kernel tree. The only thing that you need to configure is the path poinntig to your kernel directory.

This option is partcularly interesting for embedded developers who wish to focus on testing their modifications against a git snapshot, will it be for later submission into the mainline kernel or for internal use. Of course, quilt can be used inside the external kernel tree along with the source content manager tools.

3 Deploying OpenWrt

3.1 Supported root filesystems

Testing OpenWrt is really easy since you can flash your embedded device using JFFS2, SquashFS, ext2/3, cpio images, and even run ramdisk enabled kernel just by selecting an option in the configuration menu.

If the expected root filesystem does not exist, you can easily add it by tweaking the image generation process of your particular target.

3.2 The Image builder

The Image builder is a deployment tool that already contains a compiled toolchain and kernel such that you choose which packages and additionnal files you want to include.

The packages should be provided under the IPKG format and the files can simply be copied into a files/ directoy in the image builder root directory to be get copied to the root filesystem.

This tool is really useful for people who wish to include binary files and packages into an OpenWrt image.

3.3 The SDK

The Software Development kits also contains a binary toolchain, that allows you to compile packages without the need to build your toolchain from scratch.

This tool is particularly useful to test new package versions that are not yet packaged by OpenWrt or any other external repository and upload them to your embedded device for testing.

Getting further

You can reach the OpenWrt team using the following communication means.

IRC

You can chat with us on IRC using the Freenode Network and joining the `#openwrt` or `#openwrt-devel` channels.

Mailing-lists

You can get access to the list of mailing-lists at https://lists.openwrt.org/. To ask questions about the OpenWrt development please use the `openwrt-devel` mailing list.

Web site

The OpenWrt website is at http://openwrt.org/ and has a news engine and a forum to get in touch with other OpenWrts users and developers.

There is a trac interface on https://dev.openwrt.org/ which can be used to monitor svn commits and browse the source repository.