OpenWrt Hacking

Felix Fietkau

December 6, 2005

Contents

1 Introduction to OpenWrt **2**

2 Developer Tools **2**
2.1 Software Development Kit 2
2.2 Image Builder 3

3 Creating an OpenWrt Package Directory **3**
3.1 Config.in 3
3.2 Makefile 4
3.3 ipkg/ 6
3.4 files/ 6
3.5 patches/ 6
3.6 Kernel Module Packages 7

4 Structure of the Buildroot **7**
4.1 Build Directories 7
4.2 `toolchain/` 8
4.3 `package/` 8
4.4 `target/` 9

5 Additional Resources **11**

1 Introduction to OpenWrt

OpenWrt is a Linux distribution for wireless routers. Instead of trying to cram every possible feature into one firmware, OpenWrt provides only a minimal firmware with support for add-on packages. For users this means the ability to custom tune features, removing unwanted packages to make room for other packages and for developers this means being able to focus on packages without having to test and release an entire firmware.

OpenWrt started as a replacement firmware for the Linksys WRT54G and compatible (Broadcom BCM947xx), but currently it is being ported to other (entirely different) platforms.

In this article I want to give you an overview over using OpenWrt as a development platform, by introducing the developer tools, the package porting process and by giving a short description of the way in which the build system works.

2 Developer Tools

In order to make it easy for developers to get involved with using OpenWrt as a platform, we provide two developer packages, which are generated directly out of the build system:

2.1 Software Development Kit

The first developer tool is the *Software Development Kit* (SDK). It is a stripped-down version of the OpenWrt build system, which can build packages using the same package directory format as the full Buildroot. You can use it to maintain custom packages outside of the actual source tree, even for several different versions of OpenWrt.

The SDK contains precompiled versions of the complete toolchain and all libraries that provide development files for other packages. To use it, you can either build it yourself (by downloading the OpenWrt source and selecting it in the menuconfig system), or download it from the official download location:
http://downloads.openwrt.org/whiterussian/rc4/OpenWrt-SDK-Linux-i686-1.tar.bz2

If you want to compile packages with it, just unpack it and put your package directory inside the `package/` subdirectory of the SDK, then run `make`. If you plan on building several packages, which depend on one another, you should set the depenencies in `package/depend.mk`. The format is the same as the dependency format in the Buildroot:

```
package1-compile:  package2-compile
```

The above makes the compile step of `package2` depend on the successful build of `package1`.

2.2 Image Builder

The second developer tool is the *Image Builder*. It was designed for generating multiple firmware images from package lists and packages, without having to compile anything during the image building process. That makes it easy to maintain custom firmware builds with a specific feature set (wireless hotspot, mesh node, etc.), while staying current with the official OpenWrt releases. You can customize any part of the filesystem used in the images, either by adding or replacing packages (in the package directory or the package lists), or by adding some additional (unpackaged) files to the root filesystem.

3 Creating an OpenWrt Package Directory

3.1 Config.in

This file defines the configuration options of your package for the menuconfig system. It is required, if you want to integrate your package into the OpenWrt Buildroot. The syntax is the same as the kernel config syntax of the Linux 2.6 kernel.

Example:

```
1 config BR2_PACKAGE_STRACE
2     tristate "strace - System call tracer"
3     default m if CONFIG_DEVEL
4     help
5       A useful diagnostic, instructional, and debugging tool.
6       Allows you to track what system calls a program makes
7       while it is running.
8
9       http://sourceforge.net/projects/strace/
```

Line 1	declares the config option for the strace package. Configuration options for packages always start with `BR2_PACKAGE_`, because the package template of the common build system code will assume that it is set this way.
Line 2	defines the prompt of the config option. `tristate` means that the package can either be integrated into the firmware or only compiled as a package.
Line 3	will make sure that the package is enabled by default in developer (and release) builds.
Lines 4-9	define the help text for the current config option.

If you build multiple packages from the same source, you can add an extra config option for each of the additional packages in the same Config.in file

3.2 Makefile

This file contains all instructions that are necessary for cross-compiling your package. It is a normal makefile except for the fact that it uses a lot of shared code from the build system.

Example:

```
include $(TOPDIR)/rules.mk

PKG_NAME:=strace
PKG_VERSION:=4.5.11
PKG_RELEASE:=1
PKG_MD5SUM:=28335e15c83456a3db055a0a0efcb4fe

PKG_SOURCE_URL:=@SF/strace
PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.bz2
PKG_CAT:=bzcat

PKG_BUILD_DIR:=$(BUILD_DIR)/$(PKG_NAME)-$(PKG_VERSION)

include $(TOPDIR)/package/rules.mk

$(eval $(call PKG_template,STRACE,strace,$(PKG_VERSION)-$(PKG_RELEASE),$(ARCH)))

$(PKG_BUILD_DIR)/.configured:
        (cd $(PKG_BUILD_DIR); rm -rf config.cache; \
                $(TARGET_CONFIGURE_OPTS) \
                CFLAGS="$(TARGET_CFLAGS)" \
                CPPFLAGS="-I$(STAGING_DIR)/usr/include" \
                LDFLAGS="-L$(STAGING_DIR)/usr/lib" \
                ./configure \
                --target=$(GNU_TARGET_NAME) \
                --host=$(GNU_TARGET_NAME) \
                --build=$(GNU_HOST_NAME) \
                --program-prefix="" \
                --program-suffix="" \
                --prefix=/usr \
                --exec-prefix=/usr \
                --bindir=/usr/bin \
                --datadir=/usr/share \
                --includedir=/usr/include \
                --infodir=/usr/share/info \
                --libdir=/usr/lib \
                --libexecdir=/usr/lib \
                --localstatedir=/var \
```

```
39                  --mandir=/usr/share/man \
40                  --sbindir=/usr/sbin \
41                  --sysconfdir=/etc \
42                  $(DISABLE_NLS) \
43                  $(DISABLE_LARGEFILE) \
44          );
45          touch $@
46
47  $(PKG_BUILD_DIR)/.built:
48          $(MAKE) -C $(PKG_BUILD_DIR) \
49                  CC=$(TARGET_CC)
50          touch $@
51
52  $(IPKG_STRACE):
53          mkdir -p $(IDIR_STRACE)/usr/sbin
54          cp $(PKG_BUILD_DIR)/$(PKG_NAME) $(IDIR_STRACE)/usr/sbin/
55          $(STRIP) $(IDIR_STRACE)/usr/sbin/*
56          $(IPKG_BUILD) $(IDIR_STRACE) $(PACKAGE_DIR)
57
58  mostlyclean:
59          $(MAKE) -C $(PKG_BUILD_DIR) clean
60          rm -f $(PKG_BUILD_DIR)/.built
```

Line 1	includes the general shared makefile, which contains most of the commonly used variables, like `$(STAGING_DIR)`.
Lines 3-12	contain some information on the package, its name, source download location, etc. If you're not using the source download rules, you can omit the variables `PKG_MD5SUM`, `PKG_SOURCE_URL`, `PKG_SOURCE` and `PKG_CAT`. You will have to add a `$(PKG_BUILD_DIR)/.prepared:` rule (similar to `.configured`) in this case.
Line 14	includes some common rules for building packages.
Line 16	activates the rules for building binary packages. It must be inserted for every single binary package that you build from the source.
Lines 18-45	define the target for configuring the package. You may omit the ./configure command for packages that don't have a configure script, but you should always include the `touch $@` command to avoid unnecessary rebuilds.
Lines 47-50	define the target for compiling the source package. This does not include any ipkg package building yet. It should only run the makefile of your source package (or whatever is necessary to compile the software).
Lines 52-56	define the target for building a binary package. You start by creating the directory structure in `$(IDIR_`*`NAME`*`)` and copying all files in there. At the end you can run the build command like in line 56 to generate the package.
Lines 58-60	define the optional `mostlyclean` target, which is used for deleting binary files from the package source directory, while leaving the sources intact. For most packages it's enough to just call the `make clean` target.

If your package is a library, you may want to install the development files into the staging directory:

```
compile-targets: install-dev
install-dev:
#       install the development files into the staging dir

clean-targets: uninstall-dev
uninstall-dev:
#       remove the development files from the staging dir
```

3.3 ipkg/

This directory contains all ipkg control files (package description and install/remove scripts). The filename is always "*pkgname.type*", for example: `strace.control`. You don't need to specify these files anywhere in your makefile, as they will be automatically added to the package at build time.

3.4 files/

This optional directory may contain extra files that you either need for compiling the package or that you want to add at a later time. It has no specific structure, but you should consider using a flat hierarchy for a small number of files.

3.5 patches/

This optional directory contains patches against the original source files. All patches should have the right format so that they can be applied with `patch -p1` from the source directory, e.g. `strace-4.5.11/`. Your patches can be in a compressed form, but this is not recommended if you plan on putting them under version control (CVS, SVN, etc.).

You don't have to add any commands to your makefile to apply these patches. If this directory exists, then the build system will automatically apply all the patches that it contains, just after unpacking the source file.

3.6 Kernel Module Packages

Kernel package directories are structurally similar to normal package directories, but with some differences:

- You should construct the package version number like this:

  `$(LINUX_VERSION)+$(PKG_VERSION)-$(BOARD)-$(PKG_RELEASE)`

- You can access the path to the kernel directory through the `$(KERNEL_DIR)` variable
- The kernel modules should be installed into `$(IDIR_<pkgname>)/lib/modules/$(LINUX_VERSION)`

4 Structure of the Buildroot

When you run make on the OpenWrt build system, it will run the individual build system targets in the following order:

- `toolchain/install` builds the toolchain and installs it into the staging directory
- `target/compile` builds the linux kernel and adds the `build_<arch>/linux` symlink, then compiles the kernel modules
- `package/compile` builds all selected packages (and installs development libraries into the staging directory)
- `target/install` installs all packages, installs the kernel modules, then uses the generated root filesystem directory and the kernel image to build the firmware

4.1 Build Directories

During the build, the following directories will be created:

- `dl/` contains all downloaded source files
- `toolchain_build_<arch>/` contains the build directories of the toolchain packages (kernel headers, uClibc, binutils, gcc, gdb)
- `staging_dir_<arch>/` contains the installed toolchain, development versions of the libraries and all utilities that are needed for the compile or image building process.
- `build_<arch>` contains the build directories of the ordinary packages.

4.2 toolchain/

`toolchain/` contains all the instructions for creating a full toolchain for cross-compiling OpenWrt.

In order to build the toolchain, the build system will first extract the kernel headers, then the uClibc source. The uClibc source directory is necessary for building gcc. Then it will build and install binutils, and later generate the *initial* gcc, which is only used to build the uClibc. With uClibc fully built, it can now generate the *final* gcc, which supports dynamic linking and targets the uClibc.

As a last, optional step it can generate the gdb for cross-debugging.

4.3 package/

`package/` contains all the code for building normal (not kernel-specific) packages.

`package/Makefile` uses the menuconfig variables to determine which subdirectories it should call. A line to translate a menuconfig line into a package directory name looks like this:

```
package-$(BR2_PACKAGE_STRACE) += strace
```

For every package that you add to the Buildroot, you need to add such a line to `package/Makefile`. Dependencies are entered like this:

```
dropbear-compile: zlib-compile
```

Of the targets that a package makefile provides, only *<name>* `-prepare`, `-compile` and `-install` are used.

- *<name>*`-prepare` unpacks and patches the software
- *<name>*`-compile` builds the software and installs the development files (if it's a library)
- *<name>*`-install` installs the software

For a `-compile` call, `-prepare` is called first though a dependency. Same with `-install` and `-compile`. Stamp files are created for `-prepare` and `-compile` to avoid unnecessary rebuilds.

You can call package building targets from the top level directory by running: `make package/`*<pkgname>*`-`*<target>*

This will run the make target *<target>* in the package directory `package/`*<pkgname>* For example, if you want to clean the strace package (so that it will be rebuilt on the next make run), you just run

```
make package/strace-clean
```

4.4 target/

`target/` contains all the kernel/target specific build system code and the actual image generating part. Most of the code is in `target/linux`.

The important make directories and targets are called in this order (`target/linux-compile` means run the target `linux-compile` in `target/`):

- `target/linux/image-compile` compiles the filesystem utilities
- `target/linux-compile`
- `target/linux/linux-2.`*X*`/compile` (called from `target/linux-compile`) builds the linux kernel
- `target/linux/image-install` (called from `target/linux/linux-2.`*X*`/compile`) builds the platform-specific image building tools and creates the firmware images

The `target/linux-`*<target>* make target can be called from the top level makefile directly. This is useful to clean the whole linux kernel directory (if you've changed the config or the patches), by running:

```
make target/linux-clean
```

The Linux kernel will then be recompiled on the next make run.

To add a new platform to the build process (*OpenWrt 'Kamikaze' 2.0* only), you need to follow these steps:

- create a menuconfig entry for it in `target/linux/Config.in`
  Example:

```
config BR2_LINUX_2_4_AR7
    bool "Support for TI AR7 based devices [2.4]"
    default n
    depends BR2_mipsel
    help
      Build firmware images for TI AR7 based routers (w.g. Linksys WAG54G v2)
```

- activate the platform in `target/linux/Makefile`
  Example:

  ```
  $(eval $(call kernel_template,2.4,ar7,2_4_AR7))
  ```

- add kernel patches for the platform in `target/linux/linux-2.X/patches/<name>/`
- copy a default kernel config to `target/linux/linux-2.X/config/<name>`

You can also add additional module packages from the kernel tree to the kernel build process by adding a menuconfig option to and changing `target/linux/linux-2.X/Makefile`.

Sample menuconfig option:

```
config BR2_PACKAGE_KMOD_USB_STORAGE
    prompt "kmod-usb-storage.................. Support for USB storage devices"
    tristate
    default m
    depends BR2_PACKAGE_KMOD_USB_CONTROLLER
```

Sample makefile line (for Linux 2.4):

```
$(eval $(call KMOD_template,USB_STORAGE,usb-storage,\
    $(MODULES_DIR)/kernel/drivers/scsi/*.o \
    $(MODULES_DIR)/kernel/drivers/usb/storage/*.o \
,CONFIG_USB_STORAGE,kmod-usb-core,60,scsi_mod sd_mod usb-storage))
```

The first two parameters in Line 1 are just like the parameters for `PKG_template` in package makefiles.
Line 2-3 define all object files that are to be copied into the module directory.
The last line defines dependencies and module loading options:

- `CONFIG_USB_STORAGE` is the Linux kernel config option that this package depends on
- `kmod-usb-core` is the name of an OpenWrt package that this kernel package depends on
- `60` is the prefix for the module autoload file in `/etc/modules.d/`, which determines the module load order
- `scsi_mod sd_mod usb-storage` are names of module files that are to be loaded in `/etc/modules.d/`*`<prefix>-<name>`*

5 Additional Resources

OpenWrt Homepage:	http://openwrt.org
OpenWrt Wiki:	http://wiki.openwrt.org
OpenWrt Documentation:	http://wiki.openwrt.org/OpenWrtDocs
OpenWrt Forum:	http://forum.openwrt.org
OpenWrt Project management:	http://dev.openwrt.org
OpenWrt IRC:	#openwrt@irc.freenode.net