Contents

Chapter 1

The Router

1.1 Getting started

1.1.1 Installation

1.1.2 Initial configuration

1.1.3 Failsafe mode

1.2 Configuring OpenWrt

1.2.1 Network

The network configuration in Kamikaze is stored in `/etc/config/network` and is divided into interface configurations. Each interface configuration either refers directly to an ethernet/wifi interface (`eth0`, `wl0`, ..) or to a bridge containing multiple interfaces. It looks like this:

```
config interface     "lan"
    option ifname    "eth0"
    option proto     "static"
    option ipaddr    "192.168.1.1"
    option netmask   "255.255.255.0"
    option gateway   "192.168.1.254"
    option dns       "192.168.1.254"
```

`ifname` specifies the Linux interface name. If you want to use bridging on one or more interfaces, set `ifname` to a list of interfaces and add:

```
    option type     "bridge"
```

It is possible to use VLAN tagging on an interface simply by adding the VLAN IDs to it, e.g. `eth0.1`. These can be nested as well.

This sets up a simple static configuration for `eth0`. `proto` specifies the protocol used for the interface. The default image usually provides `'none' 'static'`, `'dhcp'` and `'pppoe'`. Others can be added by installing additional packages.

When using the `'static'` method like in the example, the options `ipaddr` and `netmask` are mandatory, while `gateway` and `dns` are optional. DHCP currently only accepts `ipaddr` (IP address to request from the server) and `hostname` (client hostname identify as) - both are optional.

PPP based protocols (`pppoe`, `pptp`, ...) accept these options:

- username
  The PPP username (usually with PAP authentication)
- password
  The PPP password
- keepalive
  Ping the PPP server (using LCP). The value of this option specifies the maximum number of failed pings before reconnecting. The ping interval defaults to 5, but can be changed by appending ",<interval>" to the keepalive value
- demand
  Use Dial on Demand (value specifies the maximum idle time.
- server: (pptp)
  The remote pptp server IP

For all protocol types, you can also specify the MTU by using the `mtu` option.

Setting up static routes

You can set up static routes for a specific interface that will be brought up after the interface is configured.

Simply add a config section like this:

```
config route foo
        option interface lan
        option target 1.1.1.0
        option netmask 255.255.255.0
        option gateway 192.168.1.1
```

The name for the route section is optional, the `interface`, `target` and `gateway` options are mandatory. Leaving out the `netmask` option will turn the route into a host route.

Setting up the switch (currently broadcom only)

The switch configuration is set by adding a `'switch'` config section. Example:

```
config switch       "eth0"
    option vlan0    "1 2 3 4 5*"
    option vlan1    "0 5"
```

On Broadcom hardware the section name needs to be eth0, as the switch driver does not detect the switch on any other physical device. Every vlan option needs to have the name vlan<n> where <n> is the VLAN number as used in the switch driver. As value it takes a list of ports with these optional suffixes:

- `'*'`: Set the default VLAN (PVID) of the Port to the current VLAN
- `'u'`: Force the port to be untagged
- `'t'`: Force the port to be tagged

The CPU port defaults to tagged, all other ports to untagged. On Broadcom hardware the CPU port is always 5. The other ports may vary with different hardware.

1.2.2 Wireless

The WiFi settings are configured in the file `/etc/config/wireless` (currently supported on Broadcom and Atheros). When booting the router for the first time it should detect your card and create a sample configuration file. By default '`option network lan`' is commented. This prevents unsecured sharing of the network over the wireless interface.

Generic Broadcom wireless config:

```
config wifi-device      "wl0"
    option type         "broadcom"
    option channel      "5"

config wifi-iface
    option device       "wl0"
#   option network  lan
    option mode         "ap"
    option ssid         "OpenWrt"
    option hidden       "0"
    option encryption   "none"
```

Generic Atheros wireless config:

```
config wifi-device      "wifi0"
    option type         "atheros"
    option channel      "5"
    option mode          "11g"

config wifi-iface
    option device       "wifi0"
#   option network  lan
    option mode         "ap"
    option ssid         "OpenWrt"
    option hidden       "0"
    option encryption   "none"
```

Generic multi-radio Atheros wireless conifg:

```
config wifi-device  wifi0
    option type     atheros
    option channel  1

config wifi-iface
    option device   wifi0
#   option network  lan
    option mode     ap
    option ssid     OpenWrt_private
    option hidden   0
    option encryption none

config wifi-device  wifi1
    option type     atheros
    option channel  11

config wifi-iface
    option device   wifi1
#   option network  lan
    option mode     ap
    option ssid     OpenWrt_public
    option hidden   1
    option encryption none
```

There are two types of config sections in this file. The '`wifi-device`' refers to the physical wifi interface and '`wifi-iface`' configures a virtual interface on top of that (if supported by the driver).

A full outline of the wireless configuration file with description of each field:

```
config wifi-device     wifi device name
    option type     broadcom, atheros
```

```
    option country  us, uk, fr, de, etc.
    option channel  1-14
    option maxassoc 1-128 (broadcom only)
    option distance 1-n
    option mode     11b, 11g, 11a, 11bg (atheros only)

config wifi-iface
    option network  the interface you want wifi to bridge with
    option device   wifi0, wifi1, wifi2, wifiN
    option mode     ap, sta, adhoc, or wds
    option ssid     ssid name
    option bssid    bssid address
    option encryption none, wep, psk, psk2, wpa, wpa2
    option key      encryption key
    option key1     key 1
    option key2     key 2
    option key3     key 3
    option key4     key 4
    option server   ip address
    option port     port
    option hidden   0,1
    option isolate  0,1
```

Options for the `wifi-device`:

- `type`
  The driver to use for this interface.
- `country`
  The country code used to determine the regulatory settings.
- `channel`
  The wifi channel (e.g. 1-14, depending on your country setting).
- `maxassoc`
  Optional: Maximum number of associated clients. This feature is supported only on the broadcom chipset.
- `distance`
  Optional: Distance between the ap and the furthest client in meters. This feature is supported only on the atheros chipset.
- `mode`
  The frequency band (`b`, `g`, `bg`, `a`). This feature is only supported on the atheros chipset.

Options for the `wifi-iface`:

- `network`
  Selects the interface section from `/etc/config/network` to be used with this interface

- `device`
  Set the wifi device name.

- `mode`
  Operating mode:

  - `ap`
    Access point mode
  - `sta`
    Client mode
  - `adhoc`
    Ad-Hoc mode
  - `wds`
    WDS point-to-point link

- `ssid` Set the SSID to be used on the wifi device.

- `bssid` Set the BSSID address to be used for wds to set the mac address of the other wds unit.

- `encryption`
  Encryption setting. Accepts the following values:

  - `none`
  - `wep`
  - `psk`, `psk2`
    WPA(2) Pre-shared Key
  - `wpa`, `wpa2`
    WPA(2) RADIUS

- `key, key1, key2, key3, key4` (wep, wpa and psk)
  WEP key, WPA key (PSK mode) or the RADIUS shared secret (WPA RADIUS mode)

- `server` (wpa)
  The RADIUS server ip address

- `port` (wpa)
  The RADIUS server port

- `hidden`
  0 broadcasts the ssid; 1 disables broadcasting of the ssid

- `isolate`
  Optional: Isolation is a mode usually set on hotspots that limits the clients to communicate only with the AP and not with other wireless clients. 0 disables ap isolation (default); 1 enables ap isolation.

Limitations: There are certain limitations when combining modes. Only the following mode combinations are supported:

- **Broadcom**:
  - 1x `sta`, 0-3x `ap`
  - 1-4x `ap`
  - 1x `adhoc`

  WDS links can only be used in pure AP mode and cannot use WEP (except when sharing the settings with the master interface, which is done automatically).
- **Atheros**:
  - 1x `sta`, 0-4x `ap`
  - 1-4x `ap`
  - 1x `adhoc`

1.3 Advanced configuration

Structure of the configuration files

The config files are divided into sections and options/values.

Every section has a type, but does not necessarily have a name. Every option has a name and a value and is assigned to the section it was written under.

Syntax:

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

Every parameter needs to be a single string and is formatted exactly like a parameter for a shell function. The same rules for Quoting and special characters also apply, as it is parsed by the shell.

Parsing configuration files in custom scripts

To be able to load configuration files, you need to include the common functions with:

```
. /etc/functions.sh
```

Then you can use `config_load <name>` to load config files. The function first checks for <*name*> as absolute filename and falls back to loading it from `/etc/config` (which is the most common way of using it).

If you want to use special callbacks for sections and/or options, you need to define the following shell functions before running `config_load` (after including `/etc/functions.sh`):

```
config_cb() {
    local type="$1"
    local name="$2"
    # commands to be run for every section
}

option_cb() {
    # commands to be run for every option
}
```

You can also alter `option_cb` from `config_cb` based on the section type. This allows you to process every single config section based on its type individually.

`config_cb` is run every time a new section starts (before options are being processed). You can access the last section through the `CONFIG_SECTION` variable. Also an extra call to `config_cb` (without a new section) is generated after `config_load` is done. That allows you to process sections both before and after all options were processed.

You can access already processed options with the `config_get` command Syntax:

```
config_get <section> <option>            # prints the value of the option
config_get <variable> <section> <option> # stores the value inside the variable
```

In busybox ash the three-option `config_get` is faster, because it does not result in an extra fork, so it is the preferred way.

Additionally you can also modify or add options to sections by using the `config_set` command.

Syntax:

```
config_set <section> <option> <value>
```

1.3.1 Hotplug

1.3.2 Init scripts

Because OpenWrt uses its own init script system, all init scripts must be installed as `/etc/init.d/`*name* use `/etc/rc.common` as a wrapper.

Example: `/etc/init.d/httpd`

```
#!/bin/sh /etc/rc.common
# Copyright (C) 2006 OpenWrt.org

START=50
start() {
    [ -d /www ] && httpd -p 80 -h /www -r OpenWrt
}

stop() {
    killall httpd
}
```

as you can see, the script does not actually parse the command line arguments itself. This is done by the wrapper script `/etc/rc.common`.

`start()` and `stop()` are the basic functions, which almost any init script should provide. `start()` is called when the user runs `/etc/init.d/httpd start` or (if the script is enabled and does not override this behavior) at system boot time.

Enabling and disabling init scripts is done by running `/etc/init.d/`*name* `enable` or `/etc/init.d/`*name* `disable`. This creates or removes symbolic links to the init script in `/etc/rc.d`, which is processed by `/etc/init.d/rcS` at boot time.

The order in which these scripts are run is defined in the variable `START` in the init script, which is optional and defaults to `50`. Changing it requires running `/etc/init.d/`*name* `enable` again.

You can also override these standard init script functions:

- `boot()`
  Commands to be run at boot time. Defaults to `start()`
- `restart()`
  Restart your service. Defaults to `stop(); start()`
- `reload()`
  Reload the configuration files for your service. Defaults to `restart()`

You can also add custom commands by creating the appropriate functions and referencing them in the `EXTRA_COMMANDS` variable. Helptext is added in `EXTRA_HELP`.

Example:

```
status() {
    # print the status info
}

EXTRA_COMMANDS="status"
EXTRA_HELP="        status  Print the status of the service"
```

1.3.3 Network scripts

Using the network scripts

To be able to access the network functions, you need to include the necessary shell scripts by running:

```
. /etc/functions.sh      # common functions
include /lib/network     # include /lib/network/*.sh
scan_interfaces          # read and parse the network config
```

Some protocols, such as PPP might change the configured interface names at run time (e.g. `eth0` => `ppp0` for PPPoE). That's why you have to run `scan_interfaces` instead of reading the values from the config directly. After running `scan_interfaces`, the `'ifname'` option will always contain the effective interface name (which is used for IP traffic) and if the physical device name differs from it, it will be stored in the `'device'` option. That means that running `config_get lan ifname` after `scan_interfaces` might not return the same result as running it before.

After running `scan_interfaces`, the following functions are available:

- `find_config` *`interface`*
  looks for a network configuration that includes the specified network interface.
- `setup_interface` *`interface [config] [protocol]`*
  will set up the specified interface, optionally overriding the network configuration name or the protocol that it uses.

Writing protocol handlers

You can add custom protocol handlers by adding shell scripts to `/lib/network`. They provide the following two shell functions:

```
scan_<protocolname>() {
    local config="$1"
    # change the interface names if necessary
}

setup_interface_<protocolname>() {
    local interface="$1"
    local config="$2"
    # set up the interface
}
```

`scan_`*`protocolname`* is optional and only necessary if your protocol uses a custom device, e.g. a tunnel or a PPP device.

Chapter 2

Development issues

2.1 The build system

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, which is most of the time incompatible with the hardware on your development machine, 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, it is not something that is regularly attempted and so there is a certain amount of mystery and black magic associated with it. 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 – it is a time saving step but at the same time often means you will be using a rather dated set of tools. 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.

2.1.1 Building an image

OpenWrt takes a different approach to building a firmware; downloading, patching and compiling everything from scratch, including the cross compiler. To put it in simpler terms, OpenWrt does not contain any executables or even 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 template, you can change any step in the process.

As an example, if a new kernel is released, a simple change to one of the Makefiles will download the latest kernel, patch it to run on the embedded platform and

produce a new firmware image – there is no work to be done trying to track down an unmodified copy of the existing kernel to see what changes had been made, the patches are already provided and the process ends up almost completely transparent. This does not just apply to the kernel, but to anything included with OpenWrt – It is this one simple understated concept which is what allows OpenWrt to stay on the bleeding edge with the latest compilers, latest kernels and latest applications.

So let's take a look at OpenWrt and see how this all works.

Download OpenWrt

This article refers to the "Kamikaze" branch of OpenWrt, which can be downloaded via subversion using the following command:

```
$ svn checkout https://svn.openwrt.org/openwrt/trunk kamikaze
```

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

The 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, `tool_build`, which is a temporary directory for building the target independent tools, `toolchain_build_<arch>` which is used for building the toolchain for a specific architecture, and `staging_dir_<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.

- `tool_build`
- `toolchain_build_<arch>`

`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 at the following location:

```
$ svn checkout https://svn.openwrt.org/openwrt/packages packages
```

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:

```
$ ls
kamikaze  packages
$ ln -s packages/net/nmap kamikaze/package/nmap
```

To include all packages, issue the following command:

```
$ ln -s packages/*/* kamikaze/package/
```

`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_<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_<arch>`
- `dl`

Building OpenWrt

While the OpenWrt build environment was intended mostly for developers, it also has to be simple enough that an inexperienced end user can easily build his or her own customized firmware.

Running the command "`make menuconfig`" will bring up OpenWrt's configuration menu screen, through this menu you can select which platform you're targeting, which versions of the toolchain you want to use to build and what packages you want to install into the firmware image. Note that it will also check to make sure you have the basic dependencies for it to run correctly. If that fails, you will need to install some more tools in your local environment before you can begin.

Similar to the linux kernel config, almost every option has three choices, `y/m/n` which are represented as follows:

- `<*>` (pressing y)
  This will be included in the firmware image

- <M> (pressing m)
  This will be compiled but not included (for later install)
- < > (pressing n)
  This will not be compiled

After you've finished with the menu configuration, exit and when prompted, save your configuration changes.

If you want, you can also modify the kernel config for the selected target system. simply run "`make kernel_menuconfig`" and the build system will unpack the kernel sources (if necessary), run menuconfig inside of the kernel tree, and then copy the kernel config to `target/linux/<platform>/config` so that it is preserved over "`make clean`" calls.

To begin compiling the firmware, type "`make`". By default OpenWrt will only display a high level overview of the compile process and not each individual command.

Example:

```
make[2] toolchain/install
make[3] -C toolchain install
make[2] target/compile
make[3] -C target compile
make[4] -C target/utils prepare

[...]
```

This makes it easier to monitor which step it's actually compiling and reduces the amount of noise caused by the compile output. To see the full output, run the command "`make V=99`".

During the build process, buildroot will download all sources to the "`dl`" directory and will start patching and compiling them in the "`build_<arch>`" directory. When finished, the resulting firmware will be in the "`bin`" directory and packages will be in the "`bin/packages`" directory.

2.1.2 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 typicall contains package specific startup scripts or default configuration files that can be used out of the box with OpenWrt.

Looking at one of the package makefiles, you'd hardly recognize it as a makefile. Through what can only be described as blatant disregard and abuse of the traditional make format, the makefile has been transformed into an object oriented template which simplifies the entire ordeal.

Here for example, is `package/bridge/Makefile`:

```
1  #
2  # Copyright (C) 2006 OpenWrt.org
3  #
4  # This is free software, licensed under the GNU General Public License v2.
5  # See /LICENSE for more information.
6  #
7  # $Id: Makefile 5624 2006-11-23 00:29:07Z nbd $
8
9  include $(TOPDIR)/rules.mk
10
11 PKG_NAME:=bridge
12 PKG_VERSION:=1.0.6
13 PKG_RELEASE:=1
14
15 PKG_SOURCE:=bridge-utils-$(PKG_VERSION).tar.gz
16 PKG_SOURCE_URL:=@SF/bridge
17 PKG_MD5SUM:=9b7dc52656f5cbec846a7ba3299f73bd
18 PKG_CAT:=zcat
19
20 PKG_BUILD_DIR:=$(BUILD_DIR)/bridge-utils-$(PKG_VERSION)
21
22 include $(INCLUDE_DIR)/package.mk
23
24 define Package/bridge
25   SECTION:=net
26   CATEGORY:=Base system
27   TITLE:=Ethernet bridging configuration utility
28   DESCRIPTION:=\
29     Manage ethernet bridging: a way to connect networks together to \\\
30     form a larger network.
31   URL:=http://bridge.sourceforge.net/
32 endef
33
34 define Build/Configure
35     $(call Build/Configure/Default, \
36         --with-linux-headers="$(LINUX_DIR)" \
```

```
37      )
38  endef
39
40  define Package/bridge/install
41      $(INSTALL_DIR) $(1)/usr/sbin
42      $(INSTALL_BIN) $(PKG_BUILD_DIR)/brctl/brctl $(1)/usr/sbin/
43  endef
44
45  $(eval $(call BuildPackage,bridge))
```

As you can see, there's not much work to be done; everything is hidden in other makefiles and abstracted to the point where you only need to specify a few variables.

- `PKG_NAME`
  The name of the package, as seen via menuconfig and ipkg
- `PKG_VERSION`
  The upstream version number that we are downloading
- `PKG_RELEASE`
  The version of this package Makefile
- `PKG_SOURCE`
  The filename of the original sources
- `PKG_SOURCE_URL`
  Where to download the sources from (no trailing slash), you can add multiple download sources by separating them with a
  and a carriage return.
- `PKG_MD5SUM`
  A checksum to validate the download
- `PKG_CAT`
  How to decompress the sources (zcat, bzcat, unzip)
- `PKG_BUILD_DIR`
  Where to compile the package

The `PKG_*` variables define where to download the package from; `@SF` is a special keyword for downloading packages from sourceforge. There is also another keyword of `@GNU` for grabbing GNU source releases. If any of the above mentionned download source fails, the OpenWrt mirrors will be used as source.

The md5sum (if present) is used to verify the package was downloaded correctly and `PKG_BUILD_DIR` defines where to find the package after the sources are uncompressed into `$(BUILD_DIR)`.

At the bottom of the file is where the real magic happens, "BuildPackage" is a macro set up by the earlier include statements. BuildPackage only takes one argument directly – the name of the package to be built, in this case "`bridge`".

All other information is taken from the define blocks. This is a way of providing a level of verbosity, it's inherently clear what the contents of the `description` template in `Package/bridge` is, which wouldn't be the case if we passed this information directly as the Nth argument to `BuildPackage`.

`BuildPackage` uses the following defines:

`Package/<name>:`
<name> matches the argument passed to buildroot, this describes the package the menuconfig and ipkg entries. Within `Package/`*<name>* you can define the following variables:

- `SECTION`
  The type of package (currently unused)
- `CATEGORY`
  Which menu it appears in menuconfig: Network, Sound, Utilities, Multimedia ...
- `TITLE`
  A short description of the package
- `URL`
  Where to find the original software
- `MAINTAINER` (optional)
  Who to contact concerning the package
- `DEPENDS` (optional)
  Which packages must be built/installed before this package. To reference a dependency defined in the same Makefile, use <*dependency name*>. If defined as an external package, use +<*dependency name*>. For a kernel version dependency use: *@LINUX_2_<minor version>*

`Package/<name>/conffiles` **(optional):**
A list of config files installed by this package, one file per line.

`Build/Prepare` **(optional):**
A set of commands to unpack and patch the sources. You may safely leave this undefined.

`Build/Configure` **(optional):**
You can leave this undefined if the source doesn't use configure or has a normal config script, otherwise you can put your own commands here or use "`$(call Build/Configure/Default,`*<first list of arguments, second list>*`)`" as above to pass in additional arguments for a standard configure script. The first list of arguments will be passed to the configure script like that: `-arg 1 -arg 2`. The second list contains arguments that should be defined before running the configure script such as autoconf or compiler specific variables.

To make it easier to modify the configure command line, you can either extend or completely override the following variables:

- `CONFIGURE_ARGS`
  Contains all command line arguments (format: `-arg 1 -arg 2`)

- `CONFIGURE_VARS`
  Contains all environment variables that are passed to ./configure (format: `NAME="value"`)

`Build/Compile` **(optional):**
How to compile the source; in most cases you should leave this undefined.

As with `Build/Configure` there are two variables that allow you to override the make command line environment variables and flags:

- `MAKE_FLAGS`
  Contains all command line arguments (typically variable overrides like `NAME="value"`

- `MAKE_VARS`
  Contains all environment variables that are passed to the make command

`Package/<name>/install:`
A set of commands to copy files out of the compiled source and into the ipkg which is represented by the `$(1)` directory. Note that there are currently 4 defined install macros:

- `INSTALL_DIR`
  install -d -m0755

- `INSTALL_BIN`
  install -m0755

- `INSTALL_DATA`
  install -m0644

- `INSTALL_CONF`
  install -m0600

The reason that some of the defines are prefixed by "`Package/<name>`" and others are simply "`Build`" is because of the possibility of generating multiple packages from a single source. OpenWrt works under the assumption of one source per package Makefile, but you can split that source into as many packages as desired. Since you only need to compile the sources once, there's one global set of "`Build`" defines, but you can add as many "Package/<name>" defines as you want by adding extra calls to `BuildPackage` – see the dropbear package for an example.

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.

2.1.3 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.

Here for instance the Makefile for the I2C subsytem kernel modules :

```
1  #
2  # Copyright (C) 2006 OpenWrt.org
3  #
4  # This is free software, licensed under the GNU General Public License v2.
5  # See /LICENSE for more information.
6  #
7  # $Id $
8
9  I2CMENU:=I2C Bus
10
11 define KernelPackage/i2c-core
12   TITLE:=I2C support
13   DESCRIPTION:=Kernel modules for i2c support
14   SUBMENU:=$(I2CMENU)
15   KCONFIG:=$(CONFIG_I2C_CORE) \
16         $(CONFIG_I2C_DEV)
17   FILES:=$(MODULES_DIR)/kernel/drivers/i2c/*.$(LINUX_KMOD_SUFFIX)
18   AUTOLOAD:=$(call AutoLoad,50,i2c-core i2c-dev)
19 endef
20 $(eval $(call KernelPackage,i2c-core))
```

To group kernel modules under a common description in menuconfig, you might want to define a *<description>MENU* variable on top of the kernel modules makefile.

- `TITLE`
  The name of the module as seen via menuconfig
- `DESCRIPTION`
  The description as seen via help in menuconfig
- `SUBMENU`
  The sub menu under which this package will be seen

- KCONFIG
  Kernel configuration option dependency. For external modules, remove it.

- FILES
  Files you want to inlude to this kernel module package, separate with spaces.

- AUTOLOAD
  Modules that will be loaded automatically on boot, the order you write them is the order they would be loaded.

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.

2.1.4 Conventions

There are a couple conventions to follow regarding packages:

- files
  1. configuration files follow the convention
     <*name*>.conf
  2. init files follow the convention
     <*name*>.init

- patches
  1. patches are numerically prefixed and named related to what they do

2.1.5 Troubleshooting

If you find your package doesn't show up in menuconfig, try the following command to see if you get the correct description:

```
TOPDIR=$PWD make -C package/<name> DUMP=1 V=99
```

If you're just having trouble getting your package to compile, there's a few shortcuts you can take. Instead of waiting for make to get to your package, you can run one of the following:

- make package/<*name*>-clean V=99

- make package/<*name*>-install V=99

Another nice trick is that if the source directory under `build_<arch>` is newer than the package directory, it won't clobber it by unpacking the sources again. If you were working on a patch you could simply edit the sources under the `build_<arch>/<source>` directory and run the install command above, when satisfied, copy the patched sources elsewhere and diff them with the unpatched sources. A warning though - if you go modify anything under `package/<name>` it will remove the old sources and unpack a fresh copy.

Other useful targets include:

- `make package/<name>-prepare V=99`
- `make package/<name>-compile V=99`
- `make package/<name>-configure V=99`

2.2 Extra tools

2.2.1 Image Builder

2.2.2 SDK

2.3 Adding platform support

Linux is now one of the most widespread operating system for embedded devices due to its openess as well as the wide variety of platforms it can run on. Many manufacturer actually use it in firmware you can find on many devices: DVB-T decoders, routers, print servers, DVD players ... Most of the time the stock firmware is not really open to the consumer, even if it uses open source software.

You might be interested in running a Linux based firmware for your router for various reasons: extending the use of a network protocol (such as IPv6), having new features, new piece of software inside, or for security reasons. A fully open-source firmware is de-facto needed for such applications, since you want to be free to use this or that version of a particular reason, be able to correct a particular bug. Few manufacturers do ship their routers with a Sample Development Kit, that would allow you to create your own and custom firmware and most of the time, when they do, you will most likely not be able to complete the firmware creation process.

This is one of the reasons why OpenWrt and other firmware exists: providing a version independent, and tools independent firmware, that can be run on various platforms, known to be running Linux originaly.

2.3.1 Which Operating System does this device run?

There is a lot of methods to ensure your device is running Linux. Some of them do need your router to be unscrewed and open, some can be done by probing the device using its external network interfaces.

Operating System fingerprinting and port scanning

A large bunch of tools over the Internet exists in order to let you do OS fingerprinting, we will show here an example using **nmap**:

```
nmap -P0 -O <IP address>
Starting Nmap 4.20 ( http://insecure.org ) at 2007-01-08 11:05 CET
Interesting ports on 192.168.2.1:
Not shown: 1693 closed ports
PORT   STATE SERVICE
22/tcp open  ssh
23/tcp open  telnet
53/tcp open  domain
80/tcp open  http
MAC Address: 00:13:xx:xx:xx:xx (Cisco-Linksys)
Device type: broadband router
Running: Linksys embedded
OS details: Linksys WRT54GS v4 running OpenWrt w/Linux kernel 2.4.30
Network Distance: 1 hop
```

nmap is able to report whether your device uses a Linux TCP/IP stack, and if so, will show you which Linux kernel version is probably runs. This report is quite reliable and it can make the distinction between BSD and Linux TCP/IP stacks and others.

Using the same tool, you can also do port scanning and service version discovery. For instance, the following command will report which IP-based services are running on the device, and which version of the service is being used:

```
nmap -P0 -sV <IP address>
Starting Nmap 4.20 ( http://insecure.org ) at 2007-01-08 11:06 CET
Interesting ports on 192.168.2.1:
Not shown: 1693 closed ports
PORT   STATE SERVICE VERSION
22/tcp open  ssh     Dropbear sshd 0.48 (protocol 2.0)
23/tcp open  telnet  Busybox telnetd
53/tcp open  domain  ISC Bind dnsmasq-2.35
80/tcp open  http    OpenWrt BusyBox httpd
MAC Address: 00:13:xx:xx:xx:xx (Cisco-Linksys)
Service Info: Device: WAP
```

The web server version, if identified, can be determining in knowing the Operating System. For instance, the **BOA** web server is typical from devices running an open-source Unix or Unix-like.

Wireless Communications Fingerprinting

Although this method is not really known and widespread, using a wireless scanner to discover which OS your router or Access Point run can be used. We

do not have a clear example of how this could be achieved, but you will have to monitor raw 802.11 frames and compare them to a very similar device running a Linux based firmware.

Web server security exploits

The Linksys WRT54G was originally hacked by using a "ping bug" discovered in the web interface. This tip has not been fixed for months by Linksys, allowing people to enable the "boot_wait" helper process via the web interface. Many web servers used in firmwares are open source web server, thus allowing the code to be audited to find an exploit. Once you know the web server version that runs on your device, by using **nmap -sV** or so, you might be interested in using exploits to reach shell access on your device.

Native Telnet/SSH access

Some firmwares might have restricted or unrestricted Telnet/SSH access, if so, try to log in with the web interface login/password and see if you can type in some commands. This is actually the case for some Broadcom BCM963xx based firmwares such as the one in Neuf/Cegetel ISP routers, Club-Internet ISP CI-Box and many others. Some commands, like **cat** might be left here and be used to determine the Linux kernel version.

Analysing a binary firmware image

You are very likely to find a firmware binary image on the manufacturer website, even if your device runs a proprietary operating system. If so, you can download it and use an hexadecimal editor to find printable words such as **vmlinux**, **linux**, **ramdisk**, **mtd** and others.

Some Unix tools like **hexdump** or **strings** can be used to analyse the firmware. Below there is an example with a binary firmware found other the Internet:

```
hexdump -C <binary image.extension> | less (more)
00000000  46 49 52 45 32 2e 35 2e  30 00 00 00 00 00 00 00  |FIRE2.5.0.......|
00000010  00 00 00 00 31 2e 30 2e  30 00 00 00 00 00 00 00  |....1.0.0.......|
00000020  00 00 00 00 00 00 00 38  00 43 36 29 00 0a e6 dc  |.......8.C6)..??|
00000030  54 49 44 45 92 89 54 66  1f 8b 08 08 f8 10 68 42  |TIDE..Tf....?.hB|
00000040  02 03 72 61 6d 64 69 73  6b 00 ec 7d 09 bc d5 d3  |..ramdisk.?}.???|
00000050  da ff f3 9b f7 39 7b ef  73 f6 19 3b 53 67 ea 44  |???.?9{?s?.;Sg?D|
```

Scroll over the firmware to find printable words that can be significant.

Amount of flash memory

Linux can hardly fit in a 2MB flash device, once you have opened the device and located the flash chip, try to find its characteristics on the Internet. If your flash

chip is a 2MB or less device, your device is most likely to run a proprietary OS such as WindRiver VxWorks, or a custom manufacturer OS like Zyxel ZynOS.

OpenWrt does not currently run on devices which have 2MB or less of flash memory. This limitation will probably not be worked around since those devices are most of the time micro-routers, or Wireless Access Points, which are not the main OpenWrt target.

Pluging a serial port

By using a serial port and a level shifter, you may reach the console that is being shown by the device for debugging or flashing purposes. By analysing the output of this device, you can easily notice if the device uses a Linux kenrel or something different.

2.3.2 Finding and using the manufacturer SDK

Once you are sure your device run a Linux based firmware, you will be able to start hacking on it. If the manufacturer respected the GPL, it will have released a Sample Development Kit with the device.

GPL violations

Some manufacturers do release a Linux based binary firmware, with no sources at all. The first step before doing anything is to read the license coming with your device, then write them about this lack of Open Source code. If the manufacturer answers you they do not have to release a SDK containing Open Source software, then we recommend you get in touch with the gpl-violations.org community.

You will find below a sample letter that can be sent to the manufacturer:

> Miss, Mister,
>
> I am using a <device name>, and I cannot find neither on your website nor on the CD-ROM the open source software used to build or modify the firmware.
>
> In conformance to the GPL license, you have to release the following sources:
>
> - complete toolchain that made the kernel and applications be compiled (gcc, binutils, libc)
> - tools to build a custom firmware (mksquashfs, mkcramfs ...)
> - kernel sources with patches to make it run on this specific hardware, this does not include binary drivers
>
> Thank you very much in advance for your answer.
>
> Best regards, <your name>

Using the SDK

Once the SDK is available, you are most likely not to be able to build a complete or functional firmware using it, but parts of it, like only the kernel, or only the root filesystem. Most manufacturers do not really care releasing a tool that do work every time you uncompress and use it.

You should anyway be able to use the following components:

- kernel sources with more or less functional patches for your hardware
- binary drivers linked or to be linked with the shipped kernel version
- packages of the toolchain used to compile the whole firmware: gcc, binutils, libc or uClibc
- binary tools to create a valid firmware image

Your work can be divided into the following tasks:

- create a clean patch of the hardware specific part of the linux kernel
- spot potential kernel GPL violations especially on netfilter and USB stack stuff
- make the binary drivers work, until there are open source drivers
- use standard a GNU toolchain to make working executables
- understand and write open source tools to generate a valid firmware image

Creating a hardware specific kernel patch

Most of the time, the kernel source that comes along with the SDK is not really clean, and is not a standard Linux version, it also has architecture specific fixes backported from the **CVS** or the **git** repository of the kernel development trees. Anyway, some parts can be easily isolated and used as a good start to make a vanilla kernel work your hardware.

Some directories are very likely to have local modifications needed to make your hardware be recognized and used under Linux. First of all, you need to find out the linux kernel version that is used by your hardware, this can be found by editing the **linux/Makefile** file.

```
head -5 linux-2.x.x/Makefile
VERSION = 2
PATCHLEVEL = x
SUBLEVEL = y
EXTRAVERSION = z
NAME=A fancy name
```

So now, you know that you have to download a standard kernel tarball at **kernel.org** that matches the version being used by your hardware.

Then you can create a **diff** file between the two trees, especially for the following directories:

```
diff -urN linux-2.x.x/arch/<sub architecture> linux-2.x.x-modified/arch/<sub architec
diff -urN linux-2.x.x/include/ linux-2.x.x-modified/include > 02-includes.patch
diff -urN linux-2.x.x/drivers/ linux-2.x.x-modified/drivers > 03-drivers.patch
```

This will constitute a basic set of three patches that are very likely to contain any needed modifications that has been made to the stock Linux kernel to run on your specific device. Of course, the content produced by the **diff -urN** may not always be relevant, so that you have to clean up those patches to only let the ”must have” code into them.

The fist patch will contain all the code that is needed by the board to be initialized at startup, as well as processor detection and other boot time specific fixes.

The second patch will contain all useful definitions for that board: adresses, kernel granularity, redefinitions, processor family and features ...

The third patch may contain drivers for: serial console, ethernet NIC, wireless NIC, USB NIC ... Most of the time this patch contains nothing else than ”glue” code that has been added to make the binary driver work with the Linux kernel. This code might not be useful if you plan on writing drivers from scratch for this hardware.

Using the device bootloader

The bootloader is the first program that is started right after your device has been powered on. This program, can be more or less sophisticated, some do let you do network booting, USB mass storage booting ... The bootloader is device and architeture specific, some bootloaders were designed to be universal such as RedBoot or U-Boot so that you can meet those loaders on totally different platforms and expect them to behave the same way.

If your device runs a proprietary operating system, you are very likely to deal with a proprietary boot loader as well. This may not always be a limitation, some proprietary bootloaders can even have source code available (i.e : Broadcom CFE).

According to the bootloader features, hacking on the device will be more or less easier. It is very probable that the bootloader, even exotic and rare, has a documentation somewhere over the Internet. In order to know what will be possible with your bootloader and the way you are going to hack the device, look over the following features :

- does the bootloader allow net booting via bootp/DHCP/NFS or tftp
- does the bootloader accept loading ELF binaries ?

- does the bootloader have a kernel/firmware size limitation ?
- does the bootloader expect a firmware format to be loaded with ?
- are the loaded files executed from RAM or flash ?

Net booting is something very convenient, because you will only have to set up network booting servers on your development station, and keep the original firmware on the device till you are sure you can replace it. This also prevents your device from being flashed, and potentially bricked every time you want to test a modification on the kernel/filesystem.

If your device needs to be flashed every time you load a firmware, the bootlader might only accept a specific firmware format to be loaded, so that you will have to understand the firmware format as well.

Making binary drivers work

As we have explained before, manufacturers do release binary drivers in their GPL tarball. When those drivers are statically linked into the kernel, they become GPL as well, fortunately or unfortunately, most of the drivers are not statically linked. This anyway lets you a chance to dynamically link the driver with the current kernel version, and try to make them work together.

This is one of the most tricky and grey part of the fully open source projects. Some drivers require few modifications to be working with your custom kernel, because they worked with an earlier kernel, and few modifications have been made to the kernel in-between those versions. This is for instance the case with the binary driver of the Broadcom BCM43xx Wireless Chipsets, where only few differences were made to the network interface structures.

Some general principles can be applied no matter which kernel version is used in order to make binary drivers work with your custom kernel:

- turn on kernel debugging features such as:
  - CONFIG_DEBUG_KERNEL
  - CONFIG_DETECT_SOFTLOCKUP
  - CONFIG_DEBUG_KOBJECT
  - CONFIG_KALLSYMS
  - CONFIG_KALLSYMS_ALL
- link binary drivers when possible to the current kernel version
- try to load those binary drivers
- catch the lockups and understand them

Most of the time, loading binary drivers will fail, and generate a kernel oops. You can know the last symbol the binary drivers attempted to use, and see in the kernel headers file, if you do not have to move some structures field before or after that symbol in order to keep compatibily with both the binary driver and the stock kernel drivers.

Understanding the firmware format

You might want to understand the firmware format, even if you are not yet capable of running a custom firmware on your device, because this is sometimes a blocking part of the flashing process.

A firmare format is most of the time composed of the following fields:

- header, containing a firmare version and additional fields: Vendor, Hardware version ...
- CRC32 checksum on either the whole file or just part of it
- Binary and/or compressed kernel image
- Binary and/or compressed root filesystem image
- potential garbage

Once you have figured out how the firmware format is partitioned, you will have to write your own tool that produces valid firmare binaries. One thing to be very careful here is the endianness of either the machine that produces the binary firmware and the device that will be flashed using this binary firmware.

Writing a flash map driver

The flash map driver has an important role in making your custom firmware work because it is responsible of mapping the correct flash regions and associated rights to specific parts of the system such as: bootloader, kernel, user filesystem.

Writing your own flash map driver is not really a hard task once you know how your firmware image and flash is structured. You will find below a commented example that covers the case of the device where the bootloader can pass to the kernel its partition plan.

First of all, you need to make your flash map driver be visible in the kernel configuration options, this can be done by editing the file **linux/drivers/mtd/maps/Kconfig**:

```
config MTD_DEVICE_FLASH
        tristate "Device Flash device"
        depends on ARCHITECTURE && DEVICE
        help
         Flash memory access on DEVICE boards. Currently only works with
         Bootloader Foo and Bootloader Bar.
```

Then add your source file to the **linux/drivers/mtd/maps/Makefile**, so that it will be compiled along with the kernel.

```
obj-\$(CONFIG_MTD_DEVICE_FLASH)      += device-flash.o
```

You can then write the kernel driver itself, by creating a **linux/drivers/mtd/maps/device-flash.c** C source file.

```
// Includes that are required for the flash map driver to know of the prototypes:
#include <asm/io.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/mtd/map.h>
#include <linux/mtd/mtd.h>
#include <linux/mtd/partitions.h>
#include <linux/vmalloc.h>

// Put some flash map definitions here:
#define WINDOW_ADDR 0x1FC00000         /* Real address of the flash */
#define WINDOW_SIZE 0x400000                /* Size of flash */
#define BUSWIDTH 2                                  /* Buswidth */

static void __exit device_mtd_cleanup(void);

static struct mtd_info *device_mtd_info;

static struct map_info devicd_map = {
       .name = "device",
       .size = WINDOW_SIZE,
       .bankwidth = BUSWIDTH,
       .phys = WINDOW_ADDR,
};

static int __init device_mtd_init(void)
{
  // Display that we found a flash map device
       printk("device: 0x\%08x at 0x\%08x\n", WINDOW_SIZE, WINDOW_ADDR);
   // Remap the device address to a kernel address
       device_map.virt = ioremap(WINDOW_ADDR, WINDOW_SIZE);

       // If impossible to remap, exit with the EIO error
       if (!device_map.virt) {
               printk("device: Failed to ioremap\n");
               return -EIO;
       }

   // Initlialise the device map
       simple_map_init(&device_map);

   /* MTD informations are closely linked to the flash map device
       you might also use "jedec_probe" "amd_probe" or "intel_probe" */
       device_mtd_info = do_map_probe("cfi_probe", &device_map);

  if (device_mtd_info) {
               device_mtd_info->owner = THIS_MODULE;
```

```
int parsed_nr_parts = 0;

// We try here to use the partition schema provided by the bootloader specific code
                        if (parsed_nr_parts == 0) {
                                int ret = parse_bootloader_partitions(device_mtd_info,
                                if (ret > 0) {
                                        part_type = "BootLoader";
                                        parsed_nr_parts = ret;
                                }
                        }

                        add_mtd_partitions(devicd_mtd_info, parsed_parts, parsed_nr_pa

                        return 0;
                }
        iounmap(device_map.virt);

        return -ENXIO;
}

// This function will make the driver clean up the MTD device mapping
static void __exit device_mtd_cleanup(void)
{
  // If we found a MTD device before
        if (device_mtd_info) {
   // Delete every partitions
                del_mtd_partitions(device_mtd_info);
   // Delete the associated map
                map_destroy(device_mtd_info);
        }

// If the virtual address is already in use
        if (device_map.virt) {
// Unmap the physical address to a kernel space address
                iounmap(device_map.virt);
// Reset the structure field
               device_map.virt = 0;
        }
}


// Macros that indicate which function is called on loading/unloading the module
module_init(device_mtd_init);
module_exit(device_mtd_cleanup);


// Macros defining licence and author, parameters can be defined here too.
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Me, myself and I <memyselfandi@domain.tld");
```

2.4 Debugging and debricking

2.4.1 Adding a serial port

2.4.2 JTAG

2.5 Reporting bugs

2.5.1 Using the Trac ticket system

OpenWrt as an open source software opens its development to the community by having a publicly browseable subversion repository. The Trac software which comes along with a Subversion frontend, a Wiki and a ticket reporting system is used as an interface between developers, users and contributors in order to make the whole development process much easier and efficient.

We make distinction between two kinds of people within the Trac system:

- developers, able to report, close and fix tickets
- reporters, able to add a comment, patch, or request ticket status

Opening a ticket

A reporter might want to open a ticket for the following reasons:

- a bug affects a specific hardware and/or software and needs to be fixed
- a specific software package would be seen as part of the official OpenWrt repository
- a feature should be added or removed from OpenWrt

Regarding the kind of ticket that is open, a patch is welcome in those cases:

- new package to be included in OpenWrt
- fix for a bug that works for the reporter and has no known side effect
- new features that can be added by modifying existing OpenWrt files

Once the ticket is open, a developer will take care of it, if so, the ticket is marked as ”accepted” with the developer name. You can add comments at any time to the ticket, even when it is closed.

Submitting patches

In order to include a patch to a ticket, you need to output it, this can be done by using the **svn diff** command which generates the differences between your local copy (modified) and the version on the OpenWrt repository (unmodified yet). Then attach the patch with a description, using the "Attach" button.

Your patch must respect the following conventions :

- it has to work, with no side effect on other platforms, distributions, packages ...
- it must have a reason to be included in OpenWrt : bug fix, enhancement, feature adding/removing
- the patch name should be named like that : <index number>-this_fixes_bug_foo_and_bar.patch
- if several, they have to be indexed with an integer number : 100-patch1, 200-patch2 ...

Your patch will be read and most likely be used as-is by the developpers if it is clean and working. If not, the patch will be accepted anyway and modified to be OpenWrt-rules compliant

Closing a ticket

A ticket might be closed by a developer because:

- the problem is already fixed (wontfix)
- the problem described is not judged as valid, and comes along with an explanation why (invalid)
- the developers know that this bug will be fixed upstream (wontfix)
- the problem is very similar to something that has already been reported (duplicate)
- the problem cannot be reproduced by the developers (worksforme)

A the same time, the reporter may want to get the ticket closed since he is not longer able to trigger the bug, or found it invalid by himself.

When a ticket is closed by a developer and marked as "fixed", the comment contains the subversion changeset which corrects the bug.