BCM963XX

Linux Software Build Guide

Software User Manual

Table of Contents

Chapter 1: About This Document

1.1 Purpose and Audience

This document provides detailed information on the software build framework for the BCM96xx Broadband Gateway reference design platforms. This document is meant for software design engineers.

1.2 References

The references in this section may be used in conjunction with this document.

Document (or Item) Name	Number	Source
[1] Configuration Management System (CMS) Developer's Guide	CPE-SWUM1xx-R	CSP
[2] Software Release Notes	96XXX-SWRNXXX-R	docSAFE

Chapter 2: Introduction

This document provides detailed information on the software build framework for the BCM96xx Broadband Gateway reference design platforms. This build framework is used to build a software image, which includes both a Linux® kernel and userspace applications. Under the framework, developers have the flexibility to choose a variety of pre-defined configurations or customize those configurations for their platform. For example, if a developer does not need firewall or NAT, they can be excluded from the build configurations to obtain a smaller image size. The file specifying build configurations for a hardware platform is called a platform build profile. This framework is generic and designed with scalability, so that future platforms can be added with little effort.

2.1 Toolchain

In order to build the software image, the latest toolchain must be installed. Refer to the Software Release Notes (Reference [2] on page 4) for detailed information about the latest toolchain.

2.1.1 Typical Installation

The typical Installation is to the /opt/toolchains. To install in any other directory, see the section “Non-typical Installation”. Refer to the Software Release Notes (Reference [2] on page 4) for installation instructions.

2.1.2 Non-typical Installation

Non-typical installation assumes that you need to install in a directory other than /opt/toolchains. To install the toolchain in a different location, it is important that you rebuild the toolchain with the correct directory specified – do not attempt to copy the toolchain from the default installed directory to another directory, as this is known to cause problems.

To build the toolchain:

1. Obtain the toolchain source files from Broadcom. Then build using make TOOLCHAIN_TOP=</new/toolchain/directory> where </new/toolchain/directory> is where you want the toolchain to be installed, e.g., `/usr/local/toolchain`.
2. Once the make is complete, add export TOOLCHAIN_TOP=</new/toolchain/directory> to your .bashrc file, or include TOOLCHAIN_TOP=</new/toolchain/directory> to your build command line (e.g., `make PROFILE=96328GW TOOLCHAIN_TOP=/usr/local/toolchain)/newtoolchain/directory`).

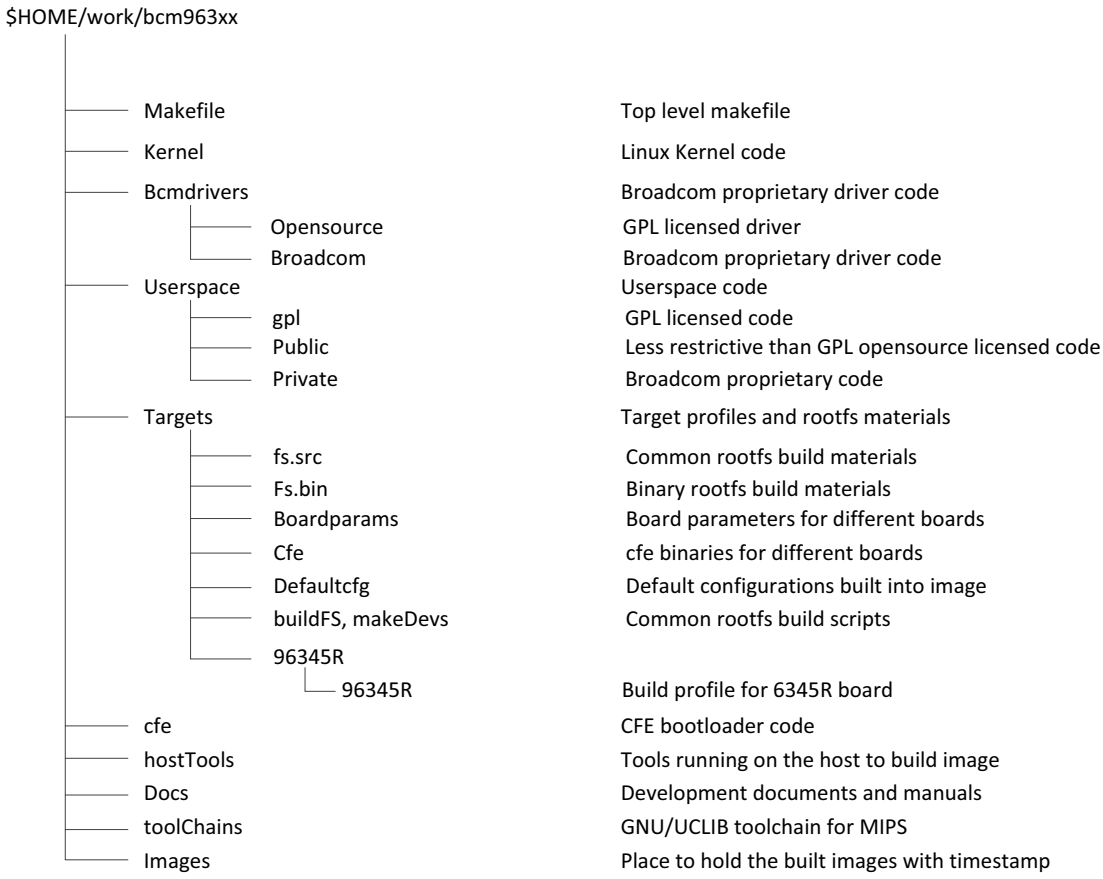
Chapter 3: The Architecture Of Build Framework

In the build framework, kernel, drivers, user applications and platform build profiles are separated into different directories. The top-level Makefile controls the whole build process according to the chosen platform build profile. Figure 1 shows the complete source code structure.

- The *kernel* directory is the Linux kernel source directory as well as board support package (BSP) for BCM9XXX boards.
- The *bcmdrivers* directory includes both Broadcom proprietary and GPL licensed driver code. o*pensource* sub-directory contains GPL licensed code. *broadcom* sub-directory contains the proprietary code.
- The *userspace* directory is the user space application source directory. The *gpl* sub-directory contains code which has the GPL license. The public sub-directory contains code from Broadcom and other sources which has a less strict license than the GPL. See the licensing text in these directories for specific licensing terms. The private sub-directory contains the proprietary code developed by Broadcom.
- The *targets* directory is the place to hold different build profiles for different hardware platforms. Several default profiles are included in this directory, and developers can create their own profiles in this directory using the ncurses-based GUI tool (see "Using the Build GUI" on page 9). Each profile corresponds to a sub-directory in the *targets* directory. The sub-directory must have the same name as that of the profile. The final target root file system and flash images generated by the build process for a chosen profile are also included in the sub-directory.

NOTE: The CFE bootloader is not built as part of the normal build process. See the bcm63xx_bootloader_appnote included in the release tarball for more details about building the CFE bootloader.

Figure 1: Directory System



Chapter 4: Platform Build Profile

A platform build profile specifies various configuration items when building a software image for a board. It is a central file that specifies the configurations of a build. When developers need a customized image, they can modify an existing profile in the software release (recommended) or create a new build profile and specify the configurations. Note that a hardware board may have any number of build profiles, with different configurations specified.

When you receive a software release from Broadcom, it contains one or more build profiles for each of the supported reference boards. It is important that you review the profile or profiles for the board that you are developing and verify that the settings are appropriate for your intended use of the software image. For example, the profile that you receive in the software release may have many useful debugging features disabled. If you will be doing more software development on this profile, you should enable these debugging features but then disable them during the final test phase before releasing the software to your customers. Alternatively, the profile(s) in the software release may have many debugging features enabled which are appropriate for software development but not for the final customer image. ***Some of these debug settings may create a security risk***. You ***must*** review your profile before final software release to verify that unnecessary options are disabled.

4.1 Supported Configuration Items

Some of the configuration items supported by the build profiles are described below (not a complete list). Note that if a configuration item is not supported for the given chip, it will not be shown as an option.

- Chip and Board Configuration selection
- Toolchain selection
- Root File system Selection
- Kernel Configuration Selection
- Major Feature Selection
  - Use Broadcom Configuration Management System (CMS)
  - Busybox
- DSL/XTM Selection
- Ethernet and VLAN Selection
- USB and Storage Options Selection
- WLAN Selection
- Voice/Phone Selection
- Broadcom Execution Environment Platform (BEEP)
- WAN Protocols and VPN
- Firewalls, ALG’s, and Networking Features
- Packet Acceleration
- Other Features
- Management Protocols and User Interface Selection
  - Consoled: Required for login on the serial console
  - Support Menu Driven Interface: Allows users to select configuration options from a numbered menu of options.
  - Support Command Line Interface: Requires users to type commands from a pre-defined set of commands.
  - httpd: allow configuration via Web UI.
  - Telnet: Allows login to the system using the Telnet protocol.
  - TR69 Management Protocol.
  - SSHD: Allows login to the system using the SSH protocol. Only one login using the SSH protocol is allowed at a time.
  - Maximum Sessions for Telnet and SSH: The total number of Telnet and SSH logins allowed at a time.

NOTE: Only one SSH login is allowed. So if this number is five, then a maximum of five Telnets and no SSH logins, or four Telnets and one SSH logins will be allowed.

- Debug Selection Section
  - Enable KALLSYMS in kernel: This allows function names to be printed in oops and other stack backtrace messages. Uses more kernel memory.
  - Enable CONFIG_PRINTK and CONFIG_BUG: Enables printk and BUG functionality. Disable this only if you really need to save kernel memory.
  - Enable Colorized Prints: Some messages have color encoding, disable this if you find the color coding distracting.
  - Enable Asserts: This should be enabled during development phase. If disabled, all BCM assert code is compiled out. This saves a little code space and may improve performance slightly.
  - Enable Fatal Asserts: Cause a fatal error when BCM assert fails. You may want to keep asserts enabled but disable fatal asserts in the final production build. The software will still log an assert failure but try to keep running.
  - Enable Debug Tools: Enables various debug code. This should be disabled in production builds since some debug tools may open security holes.
  - Bypass CMS Login: Allows to log into the CLI without typing username and password. Useful during development phase but definitely should be disabled for production code.

The options in make menuconfig should provide sufficient customization for most customers. However, customers who need even more detailed customization can add additional scripts, similar to the existing scripts, to hostTools/scripts/gendefconfig.d/ (for kernel settings) and userspace/gpl/apps/busybox/config.d/ (for busybox settings).

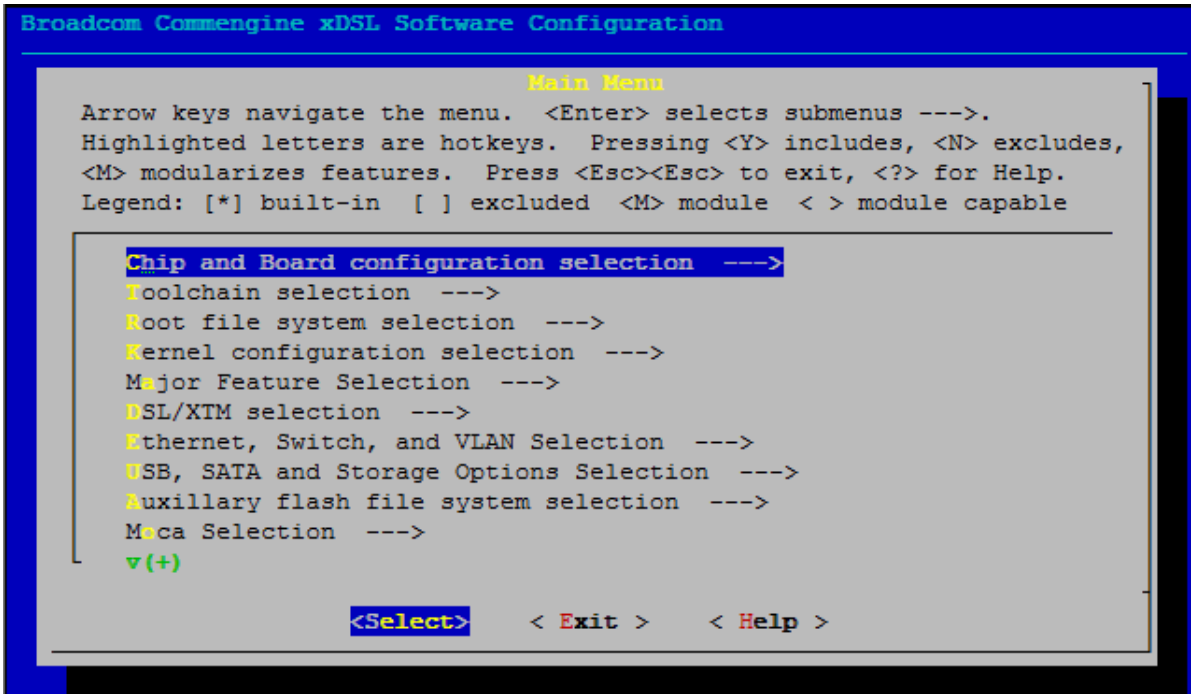
4.2 Using the Build GUI

The build framework provides an ncurses-based GUI interface to make the configuration process easier.

To start the GUI interface, go to the top-level directory of the source code and type **make menuconfig.** Within the make menuconfig interface, different configuration options accept different types of entries. Use the arrow keys and the return key to navigate through the different menu levels and use the space bar to cycle through the different options.

- For kernel drivers and modules:
  - "m" means build the driver as kernel module
  - “*” means statically link with kernel
  - "" (blank) means do not build the driver
- For some user applications:
  - "dynamic" means that the user application will be built as a standalone executable (as opposed to being linked into a giant single executable. Linking all applications into a single giant executable is not supported.)
  - "static" is a legacy setting. It means the same as dynamic.
  - "" (blank) means do not include the application in the build.
- For other configuration options:
  - “*” means that the option is enabled.
  - "" (blank) means the option is disabled.

Figure 2: Main Console Screen



4.3 Create A Build Profile via the GUI

To create a build profile follow these steps:

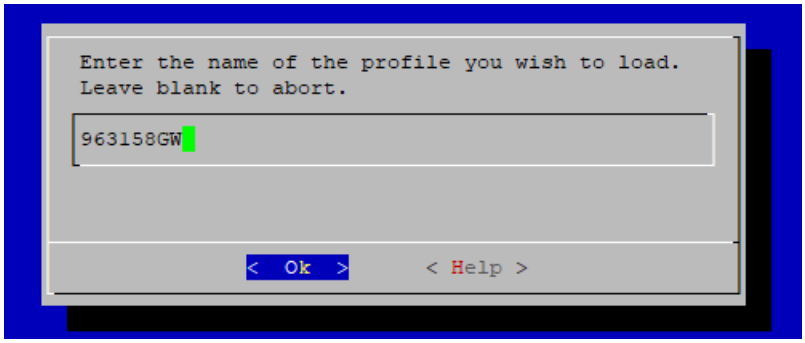
1. Open the GUI: type **make menuconfig** in the top-level directory of the source code.
2. To load an existing profile: scroll down in the main menu, or type **L** to **Load software build profile**. You can also specify the profile on the command line, e.g., **make menuconfig PROFILE=963158GW**.
3. Modify the configuration as necessary.
4. Save the changes to a different profile by selecting **Save Software Build Profile As …** to create your own platform build profile. If you do not Save Software Build Profile As..., then when you exit, you will be prompted to save your settings to the current profile.

4.3.1 Typical Example

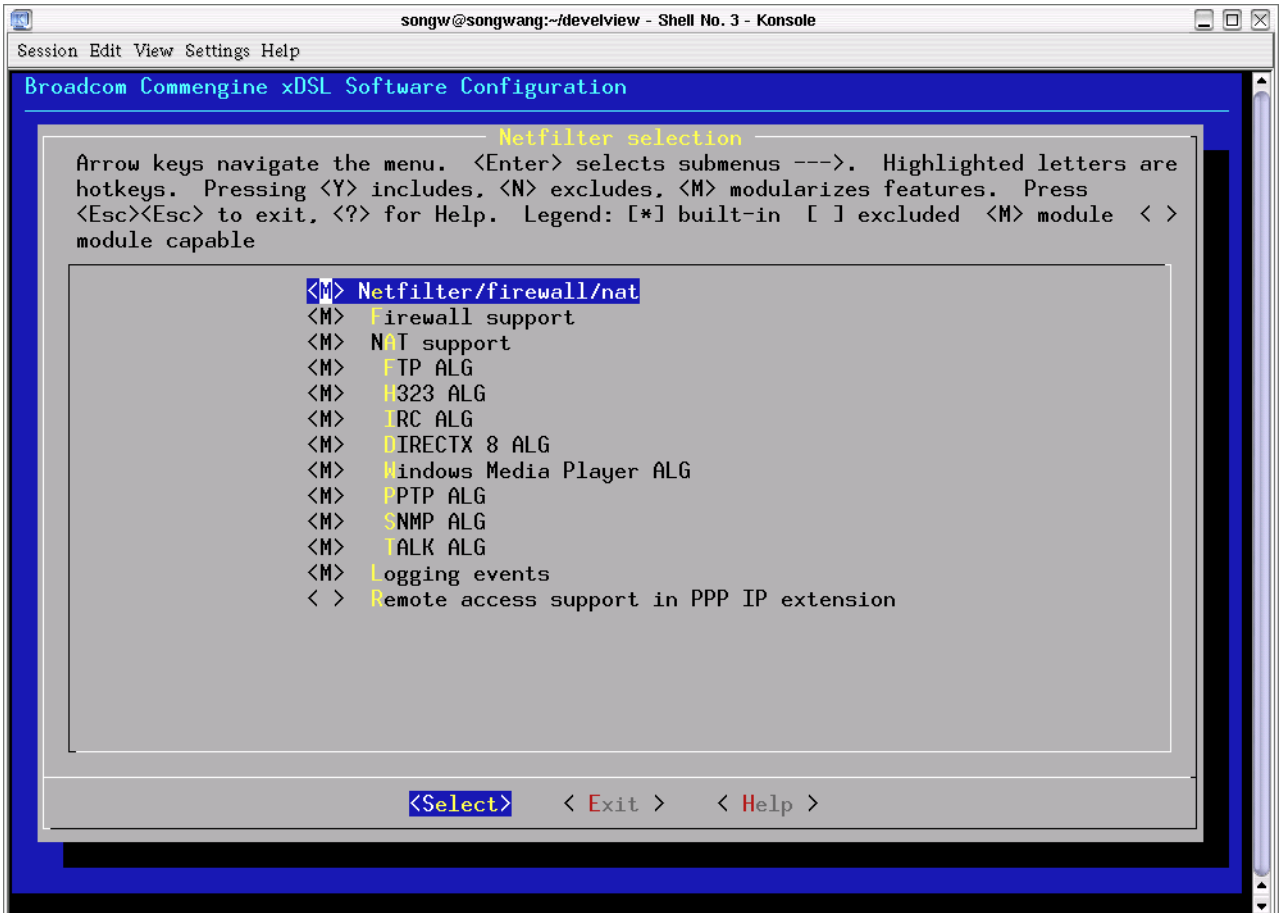
The following steps show how to create a customized profile based upon the 963158GW profile. By default, the 963158GW profile supports all the Netfilter features including NAT and firewall.

In this example, we create a profile that does not support NAT and firewall, but it needs remote access in PPP IP extension mode. Consequently, the user applications, upnp and reaim (transparent proxy of AOL/MSN messenger) are not needed, and are excluded from the new build profile. In the example, a non-valid profile is used, the 963158GW, and the feature options are not current, but the build example is still applicable to current profiles.

1. Load the 963158GW profile by selecting **Load software build profile** from the main screen.



2. Select "**Firewall, ALGs, and Networking Features**," then "**Netfilter and firewall selection**." NAT, firewall, and ALGs are all supported by the default BCM963158GW profile.

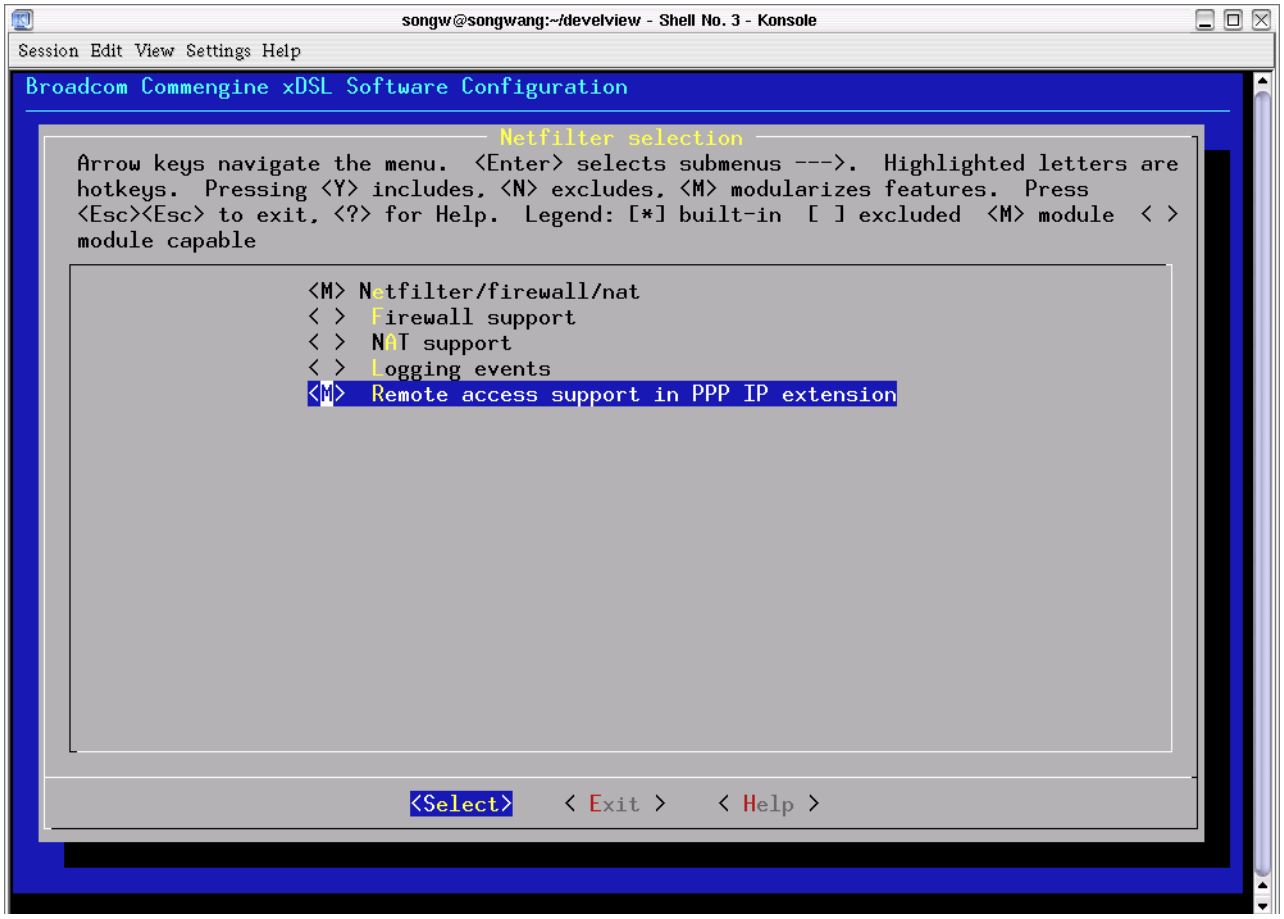


3. Clear **Firewall support**, **NAT support** and **Logging events** by pressing space or "n"
4. Select **Remote access support in PPP IP extension** by pressing "m" if built as kernel modules, or "y" if built as statically linked with the kernel.

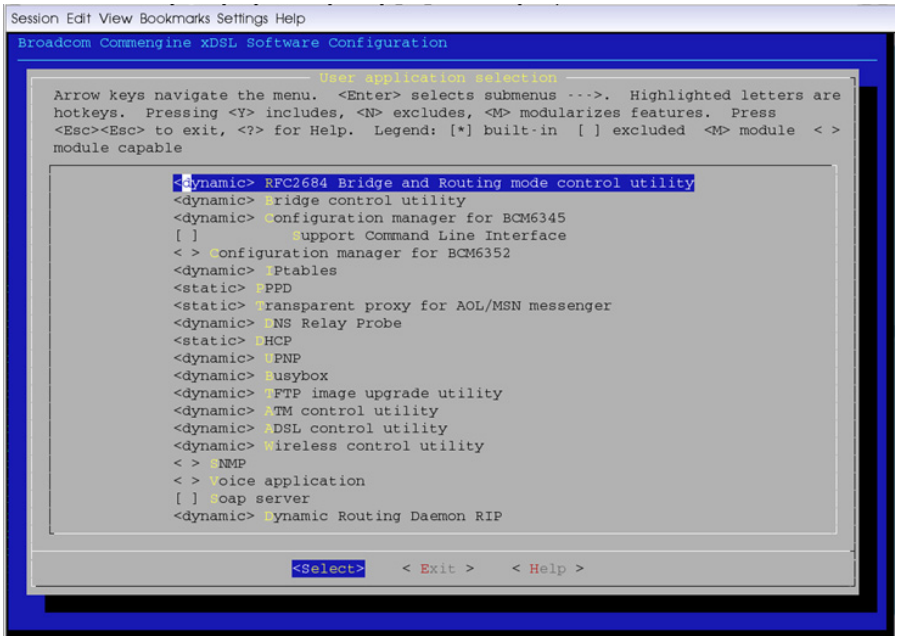
NOTE: When Firewall support is enabled, several Netfilter kernel modules will be built into the root file system including ip_tables.o, ipt_state.o, iptable_filter.o, ipt_TCPMSS.o.

NOTE: When NAT support is enabled, several netfilter kernel modules will be automatically built into the root file system including ip_tables.o, ip_conntrack.o, iptable_nat.o, ipt_MASQUERADE.o, ipt_REDIRECT.o. You can enable those ALG modules as well, such as H.323 (ip_conntrack_h323.o, ip_nat_h323.o).

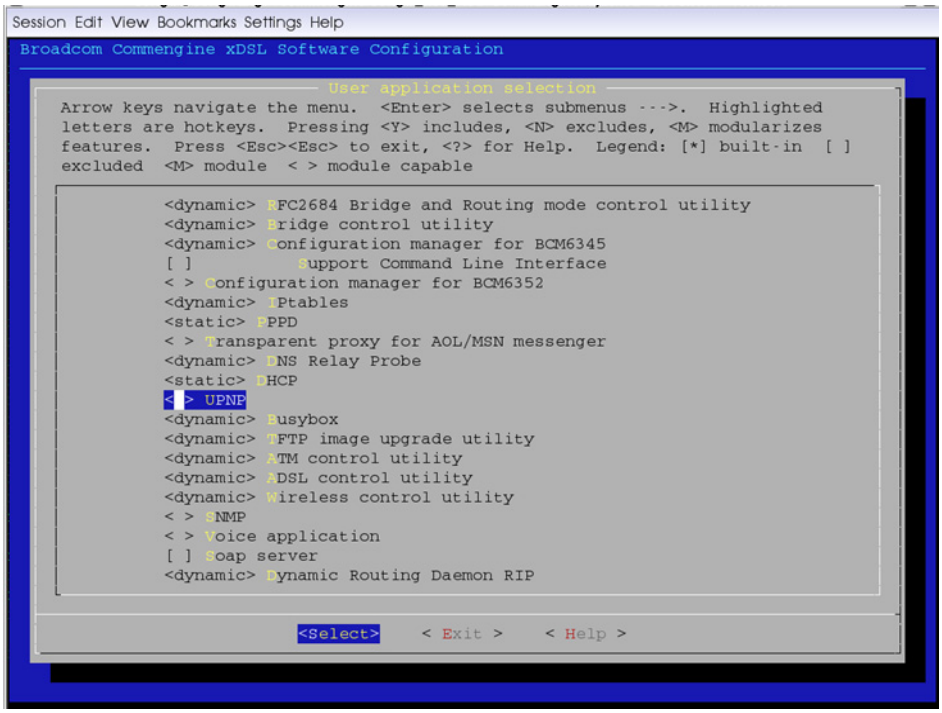
NOTE: When Remote access support in PPP IP extension is enabled, several Netfilter kernel modules will be built into the root file system including ip_tables.o, ip_conntrack.o, iptable_nat.o, iptable_filter.o and ipt_TCPMSS.



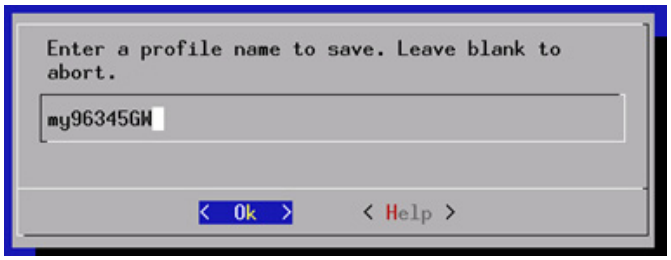
5. Select **Exit** and go back to the main menu.
6. Select **Major Feature Selection**. All the included applications for the BCM963158GW board are listed.



7. Clear **Transparent Proxy For AOL/MSN Messenger** and **UPNP** by pressing space or “n”.



8. Select **Exit** and go back to the main menu.
9. Select **Save software build profile as…** and then type <**my963158GW**>.



After the profile <my963158GW> is saved, a new directory <my963158GW> is created under the targets directory and the profile <my963158GW> is also created. You can use the new profile to build your customized image by following the commands described in the next section.

Chapter 5: Build Images With Profile

The image created by the build process is located in the targets/<profile_name> directory. The same image is copied to the images directory, but with a build timestamp and version tag attached to the file name for archive purposes.

When building for the first time, specify the build profile name (examples given below). After the first build, the build system remembers the last build profile name in a file called .last_profile at the top level directory. So the build profile name does not have to be specified in subsequent builds. .last_profile is removed by make clean, so after you do a make clean, then you must specify the profile in the make command line again.

To build an image using a different profile name, you must first do a **make clean** followed by a **make** with the new profile name. If make clean encounters an error, the .last_profile file may not be removed. In this case, simply use the **rm** command to remove the .last_profile file.

If you modify a setting in the build profile and then build the image, the dependency rules in the build system may not notice the new setting in the profile. In this case, it is safest to do a make clean and then make again.

Most modern systems contain more than one CPU. The build system now takes advantage of multi-processor systems by spawning one make thread per CPU during some parts of the build. You can control the number of threads used by the build system by specifying BRCM_MAX_JOBS=n on the make command line. However, if you encounter compile errors, it is better to use a single thread to compile so that the build stops immediately when encountering an error and the error message does not scroll off the screen. In this case, you can type:

```
make PROFILE=963268GW BRCM_MAX_JOBS=1
```

During some parts of the build, a sub-system or directory is not able to use multi-threaded builds. So when building that sub-system or directory, multi-threaded make is disabled. You will see the following message, which can be safely ignored.

```
warning: -jN forced in submake: disabling jobserver mode
```

5.1 Image Build Commands

The following commands list shows how to build the image under different circumstances.

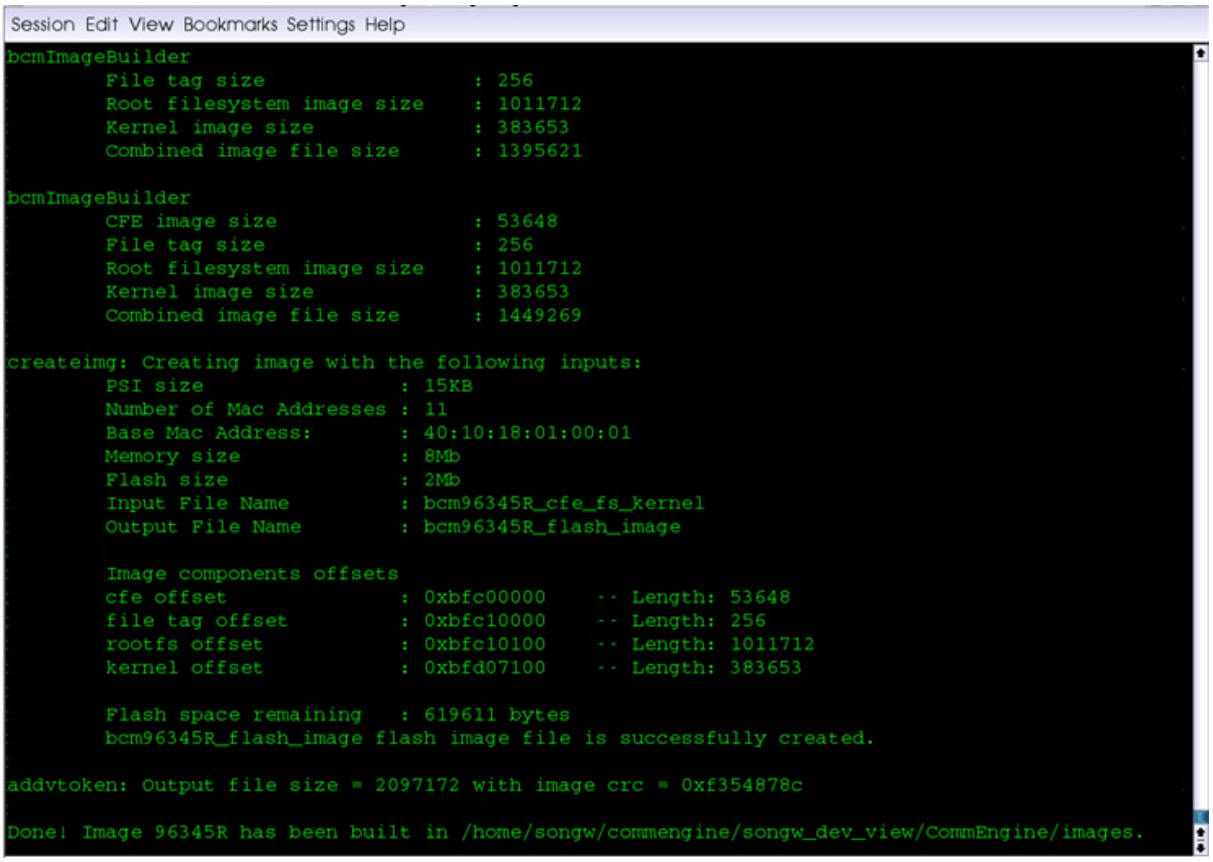
Table 1: Image Build Commands

Command	Description	Notes
`$make PROFILE=<profile name>`	Build image	–
`$make PROFILE=<profile name> kernel`	Build kernel and final image only	If you only change a kernel file and then build an image, use this command, which saves build time.
`$make PROFILE=<profile name> modules`	Build kernel modules and final image only	If you only change kernel module code such as Broadcom proprietary drivers in the bcmdrivers/broadcom directory and then build an image, use this command.
`$make PROFILE=<profile name> userspace`	Build userspace and final image only	If you change a user application file and then build an image, use this command.
`$make PROFILE=<profile name> buildimage`	Build final image only	If you only change a configuration file such as a file in the fs.src directory, you should use this command.
`$make PROFILE=<profile name> clean`	Clean up all directories	–
`$make PROFILE=<profile name> kernel_clean`	Clean up kernel and modules only	–
`$make PROFILE=<profile name> userspace_clean`	Clean up applications and target only	–
`$make PROFILE=<profile name> TOOLCHAIN=</new/toolchain/directory>`	Make using a toolchain outside of the /opt/toolchains/ crosstools-mips-gcc-4.6-linux-3.4-uclibc-0.9.32-binutils-2.21 directory	–
`$make BRCM_MAX_JOBS=4`	Make using four make threads, regardless of the actual number of CPUs on the build system	–

5.2 Flash Layout and Image Size Report

After the profile is specified and the image built, the layout and size information of all the components in your target board flash memory are reported. This includes the CFE bootloader, root file system, kernel, and remaining free space.

Figure 3: Layout And Size Information View



5.3 Other Image Build Materials

In the *targets* directory, the *fs.src* sub-directory has configuration files and scripts that must be copied into the target root file system when the image is built. The *fs.bin* sub-directory may contain binaries to be included in the root file system. In this case, the binaries (kernel modules or user applications) must be organized in the same way as the root file system is organized.

The buildFS script specifies how to build the final root file system, i.e., how to pull out the necessary files in fs.src into the fs directory.

For each profile, its directory, *targets/<profile name>/fs*, which is generated by the build process, contains the final target root file system.

Chapter 6: Adding a New Userspace Application

Follow these steps to add a new application to the build system. In this section, an application called spictl is used as an example. The steps for adding a library is similar to this method. See the existing libraries for examples.

Step 1: Allow for conditional build (optional)

If your application will always be compiled, you can skip this step. Otherwise, this step shows you how to add an option to the make menuconfig system so that users can choose whether they want to compile the new application in their build profile.

The file which controls the menu selection items in make menuconfig is located at **targets/config.in**.

Find an appropriate section for your application, and then add a line like this:

```
bool  'Enable spictl SPI SLAVE DEVICE TEST'  BUILD_SPICTL
```

The bool option allows the user to either choose “*”, which means the option is selected, or “” (blank), which means the option is not selected.

Look in config.in for examples of other ways to present a menu choice.

If the user selects the option, then after changes are saved, the build profile file contains `BUILD_SPICTL=y`. Otherwise, the saved build profile file contains `# BUILD_SPICTL is not set`.

Step 2: Define global flag in make.common (optional)

If the presence of the new application will not affect the compilation of anything else in the system, then you can skip this step. However, if it is often necessary to make other compilation units aware of the presence of the new application. This is achieved by modifying the *make.common* file at the top level.

In the CMS build system, a flag called CMS_DMP_FLAGS is passed to all standard CMS Makefiles. How to write a standard CMS Makefile is described in a later step. Note that some Makefiles, specifically those from open source packages imported from the Internet, may not have been modified to use a standard CMS Makefile. So if those applications need to use the CMS_DMP_FLAGS (which is rare), you will have to make specific modifications in those Makefiles.

To make other compilation units aware of the presence of the new application, add the following lines to *make.common* where flags are added to the CMS_COMPILE_FLAGS symbol (between lines 800 and 1700):

```
ifneq ($(strip $(BUILD_SPICTL)),)
CMS_DMP_FLAGS += -DSUPPORT_SPICTL -DDMP_X_BROADCOM_COM_SPICTL
endif
```

When you build, notice that `-DSUPPORT_SPICTL` and `-DDMP_X_BROADCOM_COM_SPICTL` is now on the compile line of the applications which use a standard CMS Makefile.

- SUPPORT_SPICTL is used to generally indicate that the SPICTL application is in the build.
- DMP_X_BROADCOM_COM_SPICTL is used to include SPICTL entries in the CMS Data Model. If the SPICTL application does not have any associated entries in the CMS Data Model, then it is not necessary to define this symbol. See the CMS Developer’s Guide (Reference [1] on page 4) for more details about the Data Model.

Step 3: Determine the location of the application

The discussion in this section should be considered as general guidelines. Consult your legal department for specific instructions.

All applications that are obtained with the GNU Public License (GPL) should be placed under the gpl/apps/ directory. All libraries obtained with the GPL should be placed under the gpl/libs/ directory. If the GPL application or library has any header files which are intended to be shared (included) by other applications and libraries, the header file should go into gpl/include. Note that header files which are only used by the application or library itself should remain in the directory of that application or library.

To comply with the terms of the GPL, all source code under the gpl directory should be released to the end consumers.

All applications that have a proprietary license should be placed under the private/apps directory. All libraries that have a proprietary license should be placed under the private/libs directory. If the proprietary application or library has any header files which are intended to be shared (included) by other applications and libraries, the header file should go into private/include. Note that header files which are only used by the application or library itself should remain in the directory of that application or library.

Applications and libraries that have a proprietary license are typically developed internally, and their source code is not released to end consumers. To avoid issues with the GPL, proprietary applications and libraries must not link (either dynamically or statically) with any GPL applications and libraries.

The following types of applications and libraries should be placed under the public/apps and public/libs directories, respectively:

- Applications or libraries obtained with an open source license which is not GPL. Examples of these types of license are LGPL, BSD, Apache, and OpenSSL.
- Applications or libraries which are developed internally, but are intended to be used by GPL applications and libraries. Examples are the cms_util and cms_msg libraries. These libraries contain code which enable GPL applications to work with the proprietary CMS system, but still keeps the GPL applications separated from the core proprietary code to avoid GPL contamination.

If the public application or library has any header files which are intended to be shared (included) by other applications and libraries, the header file should go into public/include. Note that header files which are only used by the application or library itself should remain in the directory of that application or library.

Source code under the public directory which are obtained under an open source license are released to end consumers under the same license. Source code under the public directory which were developed internally but links with GPL applications or libraries are released with a “GPL with link exception” license to the end consumer. See userspace/public/libs/cms_msg/msg.c for the text of the GPL With Link Exception license.

Step 4: Put your source code in the selected directory

Now add your .c and .h files to the source tree according to the decisions made in the previous sub-section.

Step 5: Adding your application to the build system

You can add a userspace directory to the build system without having to modify the Makefile in the level above your selected directory. Simply create an empty file called autodetect in your selected directory. The upper level Makefile will automatically detect your directory based on this autodetect file and call the Makefile in your directory.

Note that if your application is conditionally compiled based on the selection in make menuconfig and the build profile, the Makefile in your directory must implement the conditional compilation logic. This is described in Step 6: Creating a New Makefile.

The autodetect file contains rules to control the order in which the userspace makefiles are given the opportunity to build. As a result, if an application's autodetect file includes the line

```
dependson: public/libs/openssl public/libs/something
```

Then, both of the other makefiles will be called (if they exist) BEFORE the application whose autodetect includes this line. Those makefiles, in turn, may elect to return without actually building anything.

Step 6: Creating a New Makefile

CMS Makefiles

A standard CMS Makefile for an application begins with the following (using userspace/private/apps/spictl):

```
EXE = spictl
OBJS = ddrinit_6368.o spictl.o
#LIBS = -lcms_dal -lcms_cli $(CMS_COMMON_LIBS) $(CMS_CORE_LIBS)
all dynamic install: build
```

- The EXE line specifies the name of the application, which should match the name of the directory.
- The OBJS line specifies the objects which make up the application.
- spictl does not need to link against any other CMS libraries, but many applications typically will link against some set of CMS libraries. If so, the libraries can be added by setting the LIBS variable. CMS_COMMON_LIBS and CMS_CORE_LIBS are set in make.common. Remember that proprietary applications must not link against any GPL libraries.
- The fourth line directs *make all*, *make dynamic*, *make install* and *make* to a conditional target called *build*.

A standard CMS Makefile for a library begins with the following (using userspace/public/lib/cms_msg as an example):

```
LIB = libcms_msg.so
OBJS = msg.o
all: sanity_check $(LIB) generic_public_lib_install
```

In case of a library, the name of the directory is the name of the library with the lib and .so parts removed. Currently there are no examples of a library Makefile which does conditional compilation. However, by understanding the way conditional compilation works for applications, it is easy to apply the same concepts to the library Makefile.

This section is similar for applications and libraries:

```
CURR_DIR := $(shell pwd)
BUILD_DIR:=$(subst /userspace, /userspace,$(CURR_DIR))
BUILD_DIR:=$(word 1, $(BUILD_DIR))
include $(BUILD_DIR)/make.common
ALLOWED_INCLUDE_PATHS := -I.\
                                                    -I$(BUILD_DIR)/userspace/private/include
# applications may need this next line
#ALLOWED_LIB_DIRS := /lib:/lib/private:/lib/public
CUSTOM_CFLAGS += -Werror -Wfatal-errors
```

Spacing is important for the first three lines. It is recommended to use cut-and-paste on an existing Makefile to prevent white space differences.

You can customize ALLOWED_INCLUDE_PATHS and ALLOWED_LIB_DIRS to include directories that your application or libraries need. Remember that proprietary applications and libraries must not link with anything, and therefore should not include header files from gpl directories.

The following code is the conditional compilation rule for applications:

```
ifneq ($(strip $(BUILD_SPICTL)),)
build: $(EXE) generic_exe_install
else
build:
    @echo "skipping $@ (not configured)"
endif

$(EXE): $(OBJS)
    $(CC) -o $@ $(OBJS)
# if the exe needs to link against additional libraries, the $(CC) line would look like
#   $(CC) -o $@ $(OBJS) -Wl,-rpath $(CMS_LIB_RPATH) $(CMS_LIB_PATH) $(LIBS)
clean: generic_clean
    rm -f $(INSTALL_DIR)/bin/$(EXE)
```

The following code is the compilation rule for libraries. Note this is not a conditional compile, but it is easy to apply the same conditional compilation logic from the application Makefile here. Also, to keep this example clear, the oaldir part of the real Makefile is omitted.

```
$(LIB): $(OBJS)
    $(CC) -shared -Wl,--whole-archive,-soname,$@ -o $@ $(OBJS) -Wl,--no-whole-archive
clean: generic_clean
    rm -f $(INSTALL_DIR)/lib/public/$(LIB)
```

Finally, include this code for both application and library Makefiles.

```
include $(BUILD_DIR)/make.deprules
-include $(OBJS:.o=.d)
```

See existing Makefiles for more examples and additional comments/explanations about each of these sections.

Typical Linux Userspace Makefiles

When importing an external package into the build system, the Makefile that sits alongside the autodetect file should be the Makefile responsible for interfacing to the Broadcom system; and the packages files and makefiles should be in a subdirectory below that.

In the case of userspace/public/libs/libpcap/Makefile:

```
# Default rule -- call conditional build
libpcap: conditional_build

# include common variables for build environment
CURR_DIR := $(shell pwd)
BUILD_DIR:=$(subst /userspace, /userspace,$(CURR_DIR))
BUILD_DIR:=$(word 1, $(BUILD_DIR))
include $(BUILD_DIR)/make.common

# libpcap's configure requires a specific variable exported
# for cross-compile
export LINUX_VER_STR TOOLCHAIN_PREFIX

# either build or don't
ifneq ($(strip $(BUILD_LIBPCAP)),)
conditional_build: all
else
conditional_build:
    @echo "skipping libpcap (not configured)"
endif
```

If the build is enabled, the "upstream" tarball is untarred (if needed), configure is run (if needed — preferably performing the configure and build in a distinct directory), and then the code is built. When packages are installed, it is best to have them install themselves including header files and documentation and static and dynamic libraries into directories under $(BCM_FSBUILD_DIR)/ and then copy only the executables and shared libraries to the flash image under $(INSTALL_DIR)

```
libpcap/configure: libpcap.tar.gz
    mkdir -p libpcap
    cd libpcap ; tar --strip-components=1 -xzf ../libpcap.tar.gz
    touch -c libpcap/configure
check_config: objs/$(PROFILE_ARCH)/Makefile
objs/$(PROFILE_ARCH)/Makefile: libpcap/configure
    mkdir -p objs/$(PROFILE_ARCH)
    cd objs/$(PROFILE_ARCH) ; ac_cv_linux_vers=$(LINUX_VER_STR)  ../../libpcap/configure \
           --host=$(TOOLCHAIN_PREFIX) --with-pcap=linux --prefix=$(BCM_FSBUILD_DIR)/public/
all: check_config
    mkdir -p $(INSTALL_DIR)/lib/public/
    cd objs/$(PROFILE_ARCH) ; make
    cd objs/$(PROFILE_ARCH) ; make install
    mkdir -p  $(INSTALL_DIR)/lib$(BCM_INSTALL_SUFFIX_DIR)
    cp -d $(BCM_FSBUILD_DIR)/public/lib/libpcap.so* $(INSTALL_DIR)/lib$(BCM_INSTALL_SUFFIX_DIR)

clean:
     rm -rf objs/*
```

Finally, if a package needs startup scripts in the flash file system's /etc/init.d/ or /etc/rc3.d/ directories, this makefile should install the scripts as appropriate.

Step 7: Checking the build

You can verify that your application is correctly included by cd'ing into your application directory and typing **make**. If the compilation of your application is controlled by the build profile, make sure it is enabled first.

Finally, if you type **make** from the top level, your application should be built.

Chapter 7: Adding a New Kernel Driver To The Build Framework

Drivers are added in a top level directory called bcmdrivers. The advantage of maintaining drivers in a directory outside of the Linux kernel tree is that the Linux kernel can be upgraded without causing major disruptions to the structure and source control system of the bcmdrivers.

Follow these steps to add a new driver. In the steps below, a new hypothetical driver called *gmp* is used as an example.

Step 1: Make the kernel aware of the need for the driver

To add config elements in Kconfig, the following two lines need to be added to **kernel/<linux>/arch/mips/bcm963xx/ Kconfig,** where <linux> is the Linux version currently in use.

```
config BCM_GMP
    tristate "Support for the Broadcom GMP driver"
    depends on BCM96816 || BCM96818 || BCM968500
config BCM_GMP_IMPL
    int "Implementation index for the Broadcom GMP driver"
    depends on BCM96816 || BCM96818 || BCM968500
```

If the driver is associated with some hardware subsystem which is only found on some chips, then the *depends* line is used to restrict that driver to those chips. However, if the driver can be used on any chip platform, then the depends line may be omitted.

See the existing Kconfig file for other examples.

Step 2: Add defaults in defconfig-bcm.template

Default settings for the drivers are in **hostTools/scripts/defconfig-bcm.template**. By convention, the default value for a driver is *not set* and the default implementation index for all chips is 1. Basically, you need to add one line for the driver and one line per chip for the implementation index. Again, look at the defconfig-bcm.template for examples.

```
# CONFIG_BCM_GMP is not set

# CONFIG_BCM96816_GMP_IMPL=1
# CONFIG_BCM96818_GMP_IMPL=1
# CONFIG_BCM968500_GMP_IMPL=1
```

Not all chip implementation indices are shown in the above example. But in general, you should add an implementation index for all supported chips. In some rare cases, a driver will have different implementation numbers for different chips. For example, CONFIG_BCM96362_XTMRT_IMPL=3 but CONFIG_BCM963268_XTMRT_IMPL=4.

Step 3: Determine the location of the driver

Determine the location of the driver under bcmdrivers. If the driver is a proprietary driver (not GPL), then put it under broadcom. If the driver was obtained from an external source under the GPL or was written by Broadcom but can be released to the general public as GPL, then put it under opensource. In general, try to release a driver as GPL to avoid licensing problems in the future. Also note that many of the new kernel APIs are accessible only by GPL drivers. However, if a driver truly contains proprietary code that must be kept a trade secret, put it in the broadcom directory.

Under both broadcom and opensource, there are “char” and “net”. Select the appropriate sub-directory for your driver. It does not matter to Linux which directory you choose.

Step 4: Add links in bcmdrivers/Makefile

This Makefile creates the symlink between the chip number and the implementation index. It also adds the driver to the list of drivers that Linux will build during the kernelbuild process. The example below is for the hypothetical driver *gmp*, which will be added to bcmdrivers/broadcom/char/gmp.

```
ifneq ($(CONFIG_BCM_GMP),)
    LN_DRIVER_DIRS +=ln -sn impl$(CONFIG_BCM_GMP_IMPL) broadcom/char/gmp/$(LN_NAME);
    obj-$(CONFIG_BCM_GMP) += broadcom/char/gmp/$(LN_NAME)/
endif
```

Look at the existing examples in the Makefile for up-to-date instructions. If done correctly, early in the build process, a symlink should be created from the chip number to the implementation index:

```
$ cd bcmdrivers/broadcom/char/gmp
$ ls -l
bcm968500 -> impl1
impl1
```

Step 5: Allow for conditional build (optional)

If the driver should always be included in the build then this step can be skipped.

For drivers which are optional to the build, a selection item must be added to the make menuconfig system so the user can choose.

The file which controls the menu selection items in make menuconfig is located at **targets/config.in**

Find an appropriate section for your driver. Then add lines such as:

```
if [ "$BRCM_6816" = "y" -o "$BRCM_6818" = "y" -o "$BRCM_68500" = "y" ]; then
    tristate "GMP" BRCM_DRIVER_GMP
fi
```

Again, you are strongly encouraged to look at the existing file for other variations and examples. The above is only a very simple example of what can be done inside config.in.

Note that our hypothetical *gmp* driver should only be used on the 6816, 6818, and 68500 chips. So there is an if statement to check for that condition. Note these chips should match the depends line in step 1. The tristate keyword allows users to chose among three possible values: “” (empty string), “m” (dynamically loadable module), “y” (statically compiled module). The user selection is stored in the BRCM_DRIVER_GMP variable.

When the user exists from make menuconfig, there is a prompt to save the settings to the profile file, e.g., targets/96818GW/96818GW. Check that the expected settings are in this file.

NOTE: Most drivers do not allow the user to choose the implementation index. The implementation index for a specific chip is usually specified in the defconfig-bcm.template file.

Step 6: Modify gendefconfig

The gendefconfig script merges the settings in the profile file, which was customized by make menuconfig, with the settings in defconfig-bcm.template to produce a .config file which can be used by the kernel. The gendefconfig scripts can be found in **hostTools/scripts/gendefconfig.d/.** Gendefconfig uses a series of commands to transform the lines in defconfig-bcm.template.

If the driver should always be included in the build and if the implementation index is known, you can simply add a few sed commands to this script to change the default settings in defconfig-bcm.template to the desired setting. For example, if the hypothetical driver *gmp* should be compiled as a dynamically loadable driver for the 6816 and 6818 chips, and there is only implementation index 1, then the following code should be added:

```
if ( $chip =~ /^(6816|6818)$/ ) {    $c->driver_setup( "GMP", "GMP" );
fi
```

Note the driver_setup function does not actually change the implementation numbers. It depends on the defconfig-bcm.template file to specify the correct implementation number for the given chip.

In both of the examples above, the BCM9$(CHIP) is stripped out of the IMPL variable, such that,

```
# CONFIG_BCM968500_GMP_IMPL=1
```

is transformed to:

```
CONFIG_GMP_IMPL=1
```

Look at the various examples in gendefconfig. There are many different ways and styles to achieve similar results. Currently, the only driver that allows the user to select the implementation index from make menuconfig is the WLAN driver. Search for BCM_WLIMPL in gendefconfig for an example of how that is done.

Step 7: Adding your source code in the directory

Continuing with the hypothetical *gmp* driver, in step 4, we decided it would be placed into bcmdrivers/broadcom/char/impl1. So place all the source files (e.g., gmp.c, gmp_common.c and gmp_util.c) into this directory.

If the driver is a proprietary driver, i.e., under the bcmdrivers/broadcom directory, one of the .c files must declare itself as a proprietary driver using this line:

```
MODULE_LICENSE("Proprietary");
```

If the driver is a GPL driver, one of the .c files should declare itself as a GPL driver using this line:

```
MODULE_LICENSE("GPL");
```

How to write a Linux device driver is outside of the scope of this document. Linux Device Drivers, 3rd edition by O'Reilly is an excellent source of information on this subject.

Step 8: Creating a Makefile

The Makefile for the drivers under bcmdrivers must follow the format below:

```
gmp-objs := gmp.o gmp_common.o gmp_util.o
obj-$(CONFIG_BCM_GMP) += gmp.o

EXTRA_CFLAGS += -Werror -Wall

clean:
rm -f core *.o *.a *.s .*.cmd *.ko;
```

The word in bold and red is the actual name of the driver. The two instances of the driver name on the first two lines must match. Add additional EXTRA_CFLAGS if required for include paths, compile options, etc.

Chapter 8: Customizing Kernel Configuration

CAUTION! Changes to the kernel configuration can create situations that are inappropriate for the product. It is essential that all of the impacts of a kernel change are understood before making any updates.

To change a config flag, you can ADD a new executable file to hostTools/scripts/gendefconfig.d/ (see hostTools/scripts/gendefconfig.d/README.txt) such as hostTools/scripts/gendefconfig.d/30acme.conf.

In the example below, the setting is unconditional, but you will see examples in the other files that depend on profile variables. This is used to create the "defconfig" file. As long as that is not in conflict with any Kconfig rules, that will propagate to .config.

```
#!/usr/bin/perl
use strict;
use warnings;
use FindBin qw($Bin);
use lib "$Bin/../../PerlLib";
use BRCM::GenConfig;

# $p will allow us to GET values from the PROFILE
# $c will allow us to SET (and GET and DRIVER_SETUP) on the config
# file as we transform it from a template to the final config

# arguments
# * profile file
# * config  file
my $p    = new BRCM::GenConfig(shift);
my $chip = $p->get('BRCM_CHIP');
my $c    = new BRCM::GenConfig( shift, Chip => $chip, Profile => $p );

# your changes here

$c->set('CONFIG_NETFILTER_XT_MATCH_IPRANGE', 'y');
# your changes end here
$c->write();
```

Revision History

963XX-SWUM203-R; March 26, 2018

- Updated "Supported Configuration Items" on page 7
- Updated Chapter 6, Adding a New Userspace Application
- Added "Typical Linux Userspace Makefiles" on page 22

Previous Release History

Revision	Date	Change Description
963XX-SWUM202-R	02/18/14	**Updated:** ■ "Toolchain" on page 8 **Added:** ■ "Customizing Kernel Configuration" on page 29
963XX-SWUM201-R	01/26/14	Updated and reformatted. Added description of two pass build of userspace libraries.
963XX-SWUM200-R	12/25/04	Initial release


BROADCOM®