How to compile programs on the router with an SDK?

Table of contents

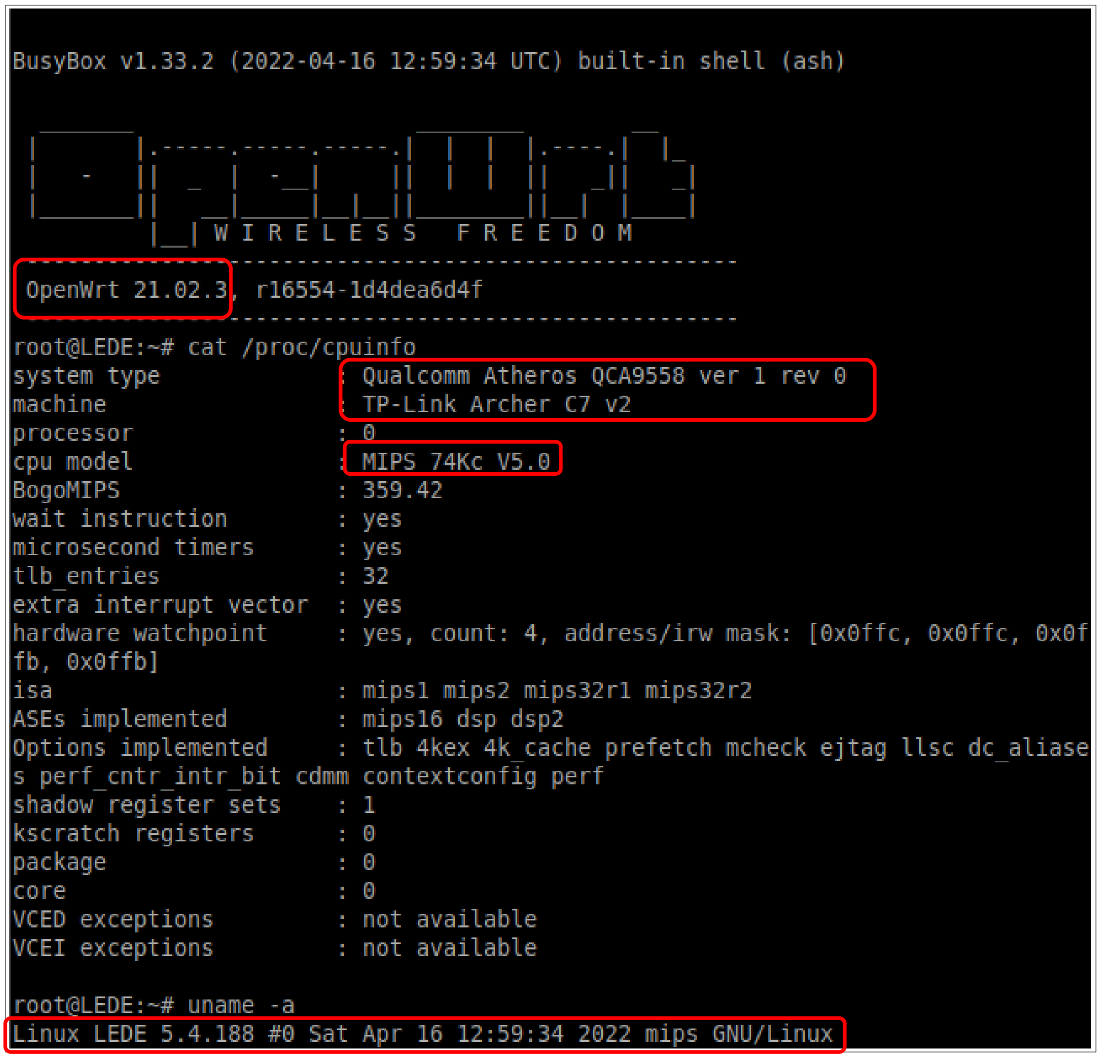
The documentation as well as the program code files are available on GitHub at the following link :

https://github.com/vinjour/UpLink

To compile a C program on the router with OpenWRT, we need an SDK to cross-compile because we don't have enough memory on the router.
The SDK will allow us to compile our programs in C to obtain an executable that we can use on our router.

1. Know our architecture

Firstly, we need to choose the correct SDK that matches the OpenWRT version of our router.
To do this, we ssh into our router to see the distribution version of our OpenWRT. Then we display the processor architecture with the command: cat /proc/cpuinfo.
We also display our linux kernel version with the command: uname -a



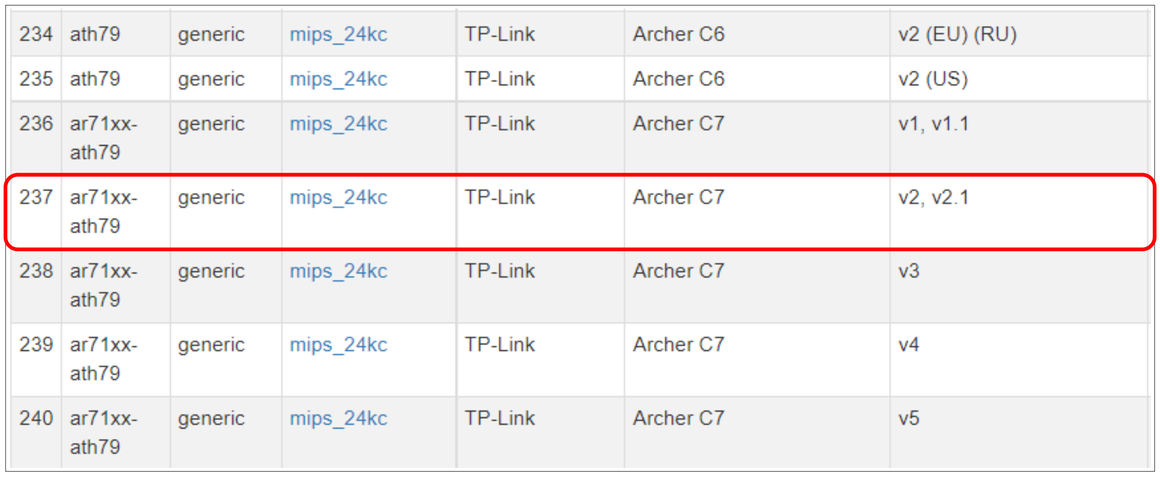
```
BusyBox v1.33.2 (2022-04-16 12:59:34 UTC) built-in shell (ash)

  _______                     ________        __
 |       |.-----.-----.-----.|  |  |  |.----.|  |_
 |   -   ||  _  |  -__|     ||  |  |  ||   _||   _|
 |_______||   __|_____|__|__||________||__|  |____|
          |__| W I R E L E S S   F R E E D O M
 -----------------------------------------------------
 OpenWrt 21.02.3, r16554-1d4dea6d4f
 -----------------------------------------------------
root@LEDE:~# cat /proc/cpuinfo
system type             : Qualcomm Atheros QCA9558 ver 1 rev 0
machine                 : TP-Link Archer C7 v2
processor               : 0
cpu model               : MIPS 74Kc V5.0
BogoMIPS                : 359.42
wait instruction        : yes
microsecond timers      : yes
tlb_entries             : 32
extra interrupt vector  : yes
hardware watchpoint     : yes, count: 4, address/irw mask: [0x0ffc, 0x0ffc, 0x0f
fb, 0x0ffb]
isa                     : mips1 mips2 mips32r1 mips32r2
ASEs implemented        : mips16 dsp dsp2
Options implemented     : tlb 4kex 4k_cache prefetch mcheck ejtag llsc dc_aliase
s perf_cntr_intr_bit cdmm contextconfig perf
shadow register sets    : 1
kscratch registers      : 0
package                 : 0
core                    : 0
VCED exceptions         : not available
VCEI exceptions         : not available

root@LEDE:~# uname -a
Linux LEDE 5.4.188 #0 Sat Apr 16 12:59:34 2022 mips GNU/Linux
```

MIPS means that we are in big endian.

2. Find the right SDK

Now, we have to find the right SDK corresponding with our architecture.
To do this, we have to find the correct target of the architecture of our router. Check that here :
[OpenWrt Wiki] Table of Hardware: Firmware downloads

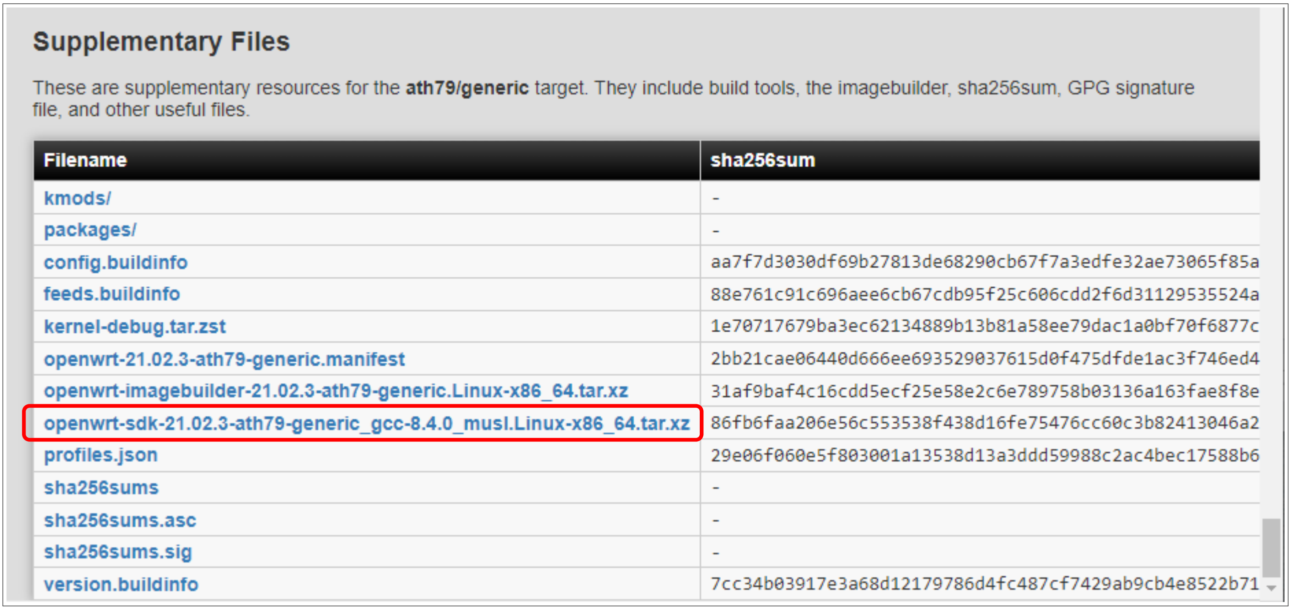
Our TP-LINK C7 AC1750 Archer C7 v2 router has as target : ath79.
We can also check this here : https://openwrt.org/docs/techref/targets/ath79



234	ath79	generic	mips_24kc	TP-Link	Archer C6	v2 (EU) (RU)
235	ath79	generic	mips_24kc	TP-Link	Archer C6	v2 (US)
236	ar71xx-ath79	generic	mips_24kc	TP-Link	Archer C7	v1, v1.1
237	ar71xx-ath79	generic	mips_24kc	TP-Link	Archer C7	v2, v2.1
238	ar71xx-ath79	generic	mips_24kc	TP-Link	Archer C7	v3
239	ar71xx-ath79	generic	mips_24kc	TP-Link	Archer C7	v4
240	ar71xx-ath79	generic	mips_24kc	TP-Link	Archer C7	v5

We have to visit : http://downloads.openwrt.org/ and navigate to the SDK corresponding to our version and to our target.
For us, this is the one : https://downloads.openwrt.org/releases/21.02.3/targets/ath79/generic/

We need to go to the bottom of the page to the Supplementary files section to download our SDK :
openwrt-sdk-21.02.3-ath79-generic_gcc-8.4.0_musl.Linux-x86_64.tar.xz



Supplementary Files

These are supplementary resources for the **ath79/generic** target. They include build tools, the imagebuilder, sha256sum, GPG signature file, and other useful files.

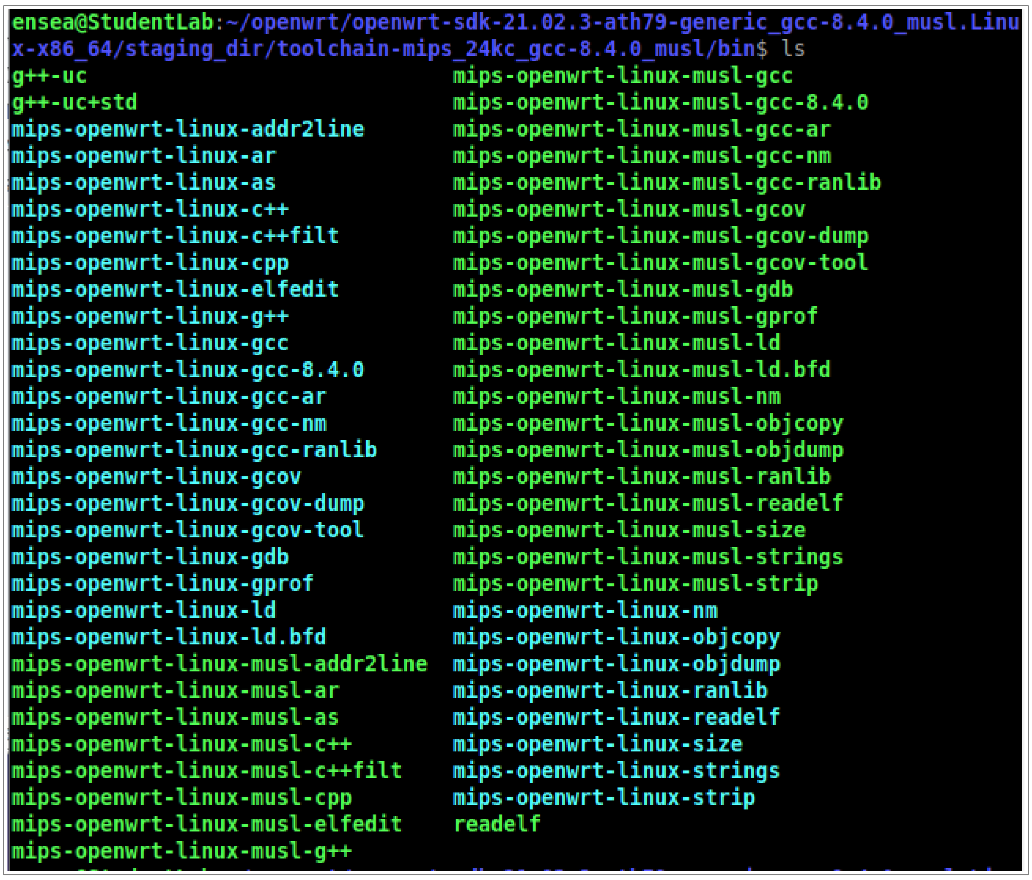
Filename	sha256sum
kmods/	-
packages/	-
config.buildinfo	aa7f7d3030df69b27813de68290cb67f7a3edfe32ae73065f85a
feeds.buildinfo	88e761c91c696aee6cb67cdb95f25c606cdd2f6d31129535524a
kernel-debug.tar.zst	1e70717679ba3ec62134889b13b81a58ee79dac1a0bf70f6877c
openwrt-21.02.3-ath79-generic.manifest	2bb21cae06440d666ee693529037615d0f475dfde1ac3f746ed4
openwrt-imagebuilder-21.02.3-ath79-generic.Linux-x86_64.tar.xz	31af9baf4c16cdd5ecf25e58e2c6e789758b03136a163fae8f8e
openwrt-sdk-21.02.3-ath79-generic_gcc-8.4.0_musl.Linux-x86_64.tar.xz	86fb6faa206e56c553538f438d16fe75476cc60c3b82413046a2
profiles.json	29e06f060e5f803001a13538d13a3ddd59988c2ac4bec17588b6
sha256sums	-
sha256sums.asc	-
sha256sums.sig	-
version.buildinfo	7cc34b03917e3a68d12179786d4fc487cf7429ab9cb4e8522b71

3. Prepare the environment

Once the compressed file is downloaded, we extract it into a new folder : « openwrt » in our linux system.

3. 1. Find the compilers

Then we have to look for where the compilers (gcc) and executables are.
For us, they are here :



```
ensea@StudentLab:~/openwrt/openwrt-sdk-21.02.3-ath79-generic_gcc-8.4.0_musl.Linu
x-x86_64/staging_dir/toolchain-mips_24kc_gcc-8.4.0_musl/bin$ ls
g++-uc                             mips-openwrt-linux-musl-gcc
g++-uc+std                         mips-openwrt-linux-musl-gcc-8.4.0
mips-openwrt-linux-addr2line       mips-openwrt-linux-musl-gcc-ar
mips-openwrt-linux-ar              mips-openwrt-linux-musl-gcc-nm
mips-openwrt-linux-as              mips-openwrt-linux-musl-gcc-ranlib
mips-openwrt-linux-c++             mips-openwrt-linux-musl-gcov
mips-openwrt-linux-c++filt         mips-openwrt-linux-musl-gcov-dump
mips-openwrt-linux-cpp             mips-openwrt-linux-musl-gcov-tool
mips-openwrt-linux-elfedit         mips-openwrt-linux-musl-gdb
mips-openwrt-linux-g++             mips-openwrt-linux-musl-gprof
mips-openwrt-linux-gcc             mips-openwrt-linux-musl-ld
mips-openwrt-linux-gcc-8.4.0       mips-openwrt-linux-musl-ld.bfd
mips-openwrt-linux-gcc-ar          mips-openwrt-linux-musl-nm
mips-openwrt-linux-gcc-nm          mips-openwrt-linux-musl-objcopy
mips-openwrt-linux-gcc-ranlib      mips-openwrt-linux-musl-objdump
mips-openwrt-linux-gcov            mips-openwrt-linux-musl-ranlib
mips-openwrt-linux-gcov-dump       mips-openwrt-linux-musl-readelf
mips-openwrt-linux-gcov-tool       mips-openwrt-linux-musl-size
mips-openwrt-linux-gdb             mips-openwrt-linux-musl-strings
mips-openwrt-linux-gprof           mips-openwrt-linux-musl-strip
mips-openwrt-linux-ld              mips-openwrt-linux-nm
mips-openwrt-linux-ld.bfd          mips-openwrt-linux-objcopy
mips-openwrt-linux-musl-addr2line  mips-openwrt-linux-objdump
mips-openwrt-linux-musl-ar         mips-openwrt-linux-ranlib
mips-openwrt-linux-musl-as         mips-openwrt-linux-readelf
mips-openwrt-linux-musl-c++        mips-openwrt-linux-size
mips-openwrt-linux-musl-c++filt    mips-openwrt-linux-strings
mips-openwrt-linux-musl-cpp        mips-openwrt-linux-strip
mips-openwrt-linux-musl-elfedit    readelf
mips-openwrt-linux-musl-g++
```

3. 2. Define new system variables

Finally, We need to define 2 new system variables PATH and STAGING_DIR to use the compilers.

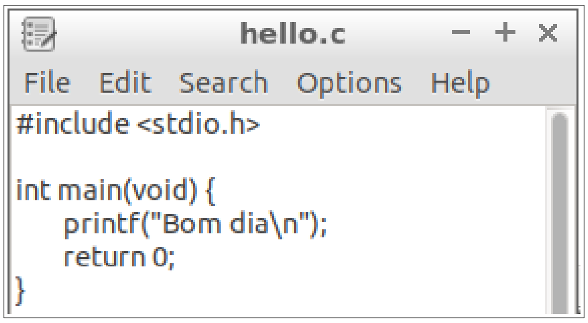
- export PATH=~/openwrt/openwrt-sdk-21.02.3-ath79-generic_gcc-8.4.0_musl.Linux-x86_64/staging_dir/toolchain-mips_24kc_gcc-8.4.0_musl/bin:$PATH

- export STAGING_DIR=~/openwrt/openwrt-sdk-21.02.3-ath79-generic_gcc-8.4.0_musl.Linux-x86_64/staging_dir

4. Compile « Hello World » program and export it to the router

4. 1. Create our program

Now we can create our program.

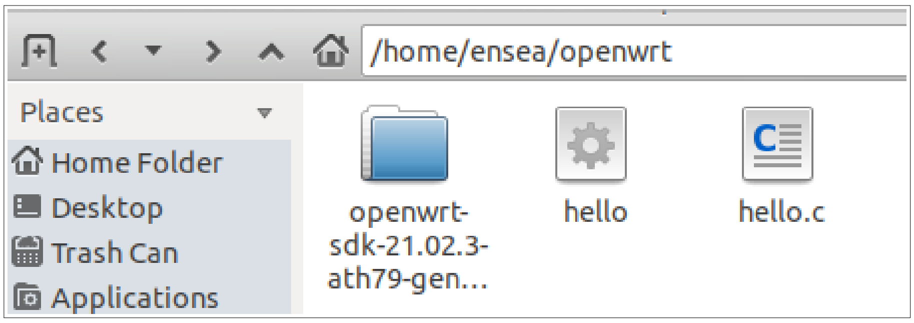


```
#include <stdio.h>

int main(void) {
    printf("Bom dia\n");
    return 0;
}
```

4. 2. Compile our program

And we can compile it with the command : mips-openwrt-linux-gcc hello.c -o hello
(For C++, the command is : mips-openwrt-linux-c++ hello.cpp -o hello)

Now, we can see the executable created.

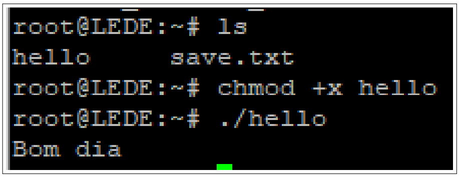


4. 3. Export executable to router

And we can finally transfer the executable to our router.
To do this, we transfer the executable to Windows using a folder shared with the Linux virtual machine. Then we transfer it with WinSCP to our router.

We change the permission of the "hello" file so that we can run it with the command :
chmod +x hello

And we run our program with the command : ./hello

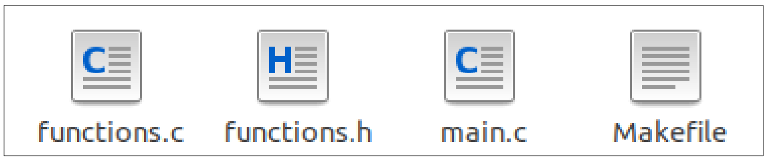


```
root@LEDE:~# ls
hello      save.txt
root@LEDE:~# chmod +x hello
root@LEDE:~# ./hello
Bom dia
```

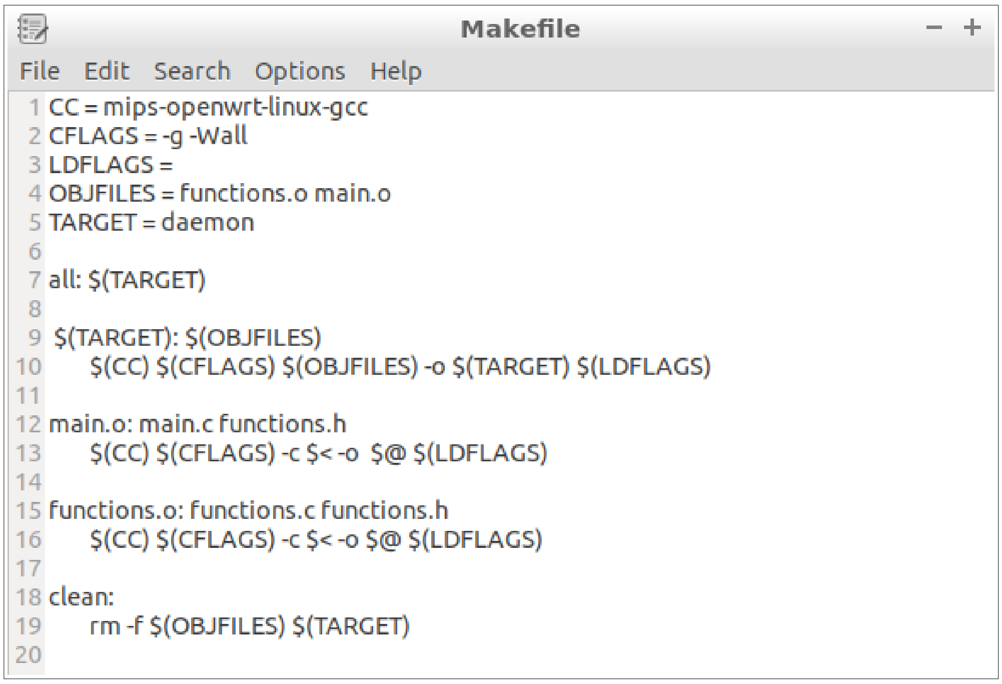
5. Compile a program with multi files using a Makefile

In order to compile a program with several C files, we use a Makefile.

Here are our different code files needed to compile our program.



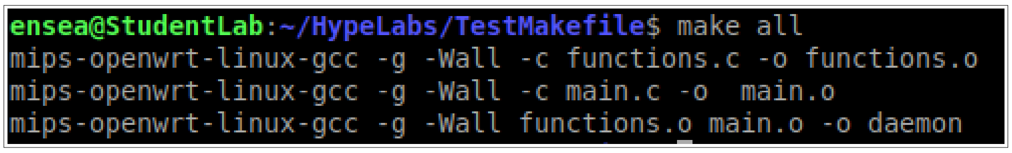
Here is an example of a Makefile.



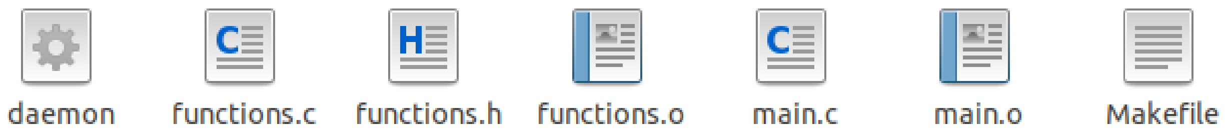
```
CC = mips-openwrt-linux-gcc
CFLAGS = -g -Wall
LDFLAGS =
OBJFILES = functions.o main.o
TARGET = daemon

all: $(TARGET)

 $(TARGET): $(OBJFILES)
	$(CC) $(CFLAGS) $(OBJFILES) -o $(TARGET) $(LDFLAGS)

main.o: main.c functions.h
	$(CC) $(CFLAGS) -c $< -o  $@ $(LDFLAGS)

functions.o: functions.c functions.h
	$(CC) $(CFLAGS) -c $< -o $@ $(LDFLAGS)

clean:
	rm -f $(OBJFILES) $(TARGET)
```

Note the CC compiler which is the one of our SDK.

After running the « make all » command, we get our object files and our executable.
All that remains is to transfer our executable to our router and run it.



```
ensea@StudentLab:~/HypeLabs/TestMakefile$ make all
mips-openwrt-linux-gcc -g -Wall -c functions.c -o functions.o
mips-openwrt-linux-gcc -g -Wall -c main.c -o  main.o
mips-openwrt-linux-gcc -g -Wall functions.o main.o -o daemon
```



With the « make clean » command, we delete the object files and the executable.

6. Compile a program with external libraries

6. 1. Installing some prerequisites packages

In order to use properly a SDK, we need to install some packages on our machine.
The packages vary depending on the operating system used. The example given here will be for a Linux/Ubuntu operating system.

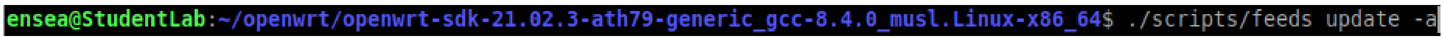
You can check the prerequisites packages depending with your OS here : [OpenWrt Wiki] Build system setup

For an Ubuntu system, you must intall those packages :

```
sudo apt update
sudo apt install build-essential gawk gcc-multilib flex git gettext l
ibncurses5-dev libssl-dev python3-distutils zlib1g-dev
```

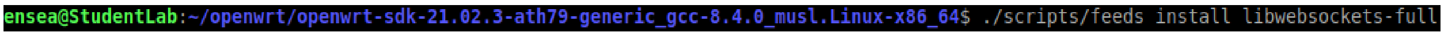
6. 2. Update and download libraries on our SDK

In order to download the library we want to compile our program, we must update our sdk like if we do an « opkg update »
We do this in our SDK directory.



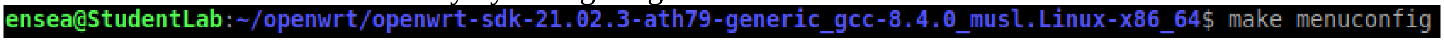
```
ensea@StudentLab:~/openwrt/openwrt-sdk-21.02.3-ath79-generic_gcc-8.4.0_musl.Linux-x86_64$ ./scripts/feeds update -a
```

After this is done, we can download the library we want among those available on OpenWRT.



```
ensea@StudentLab:~/openwrt/openwrt-sdk-21.02.3-ath79-generic_gcc-8.4.0_musl.Linux-x86_64$ ./scripts/feeds install libwebsockets-full
```

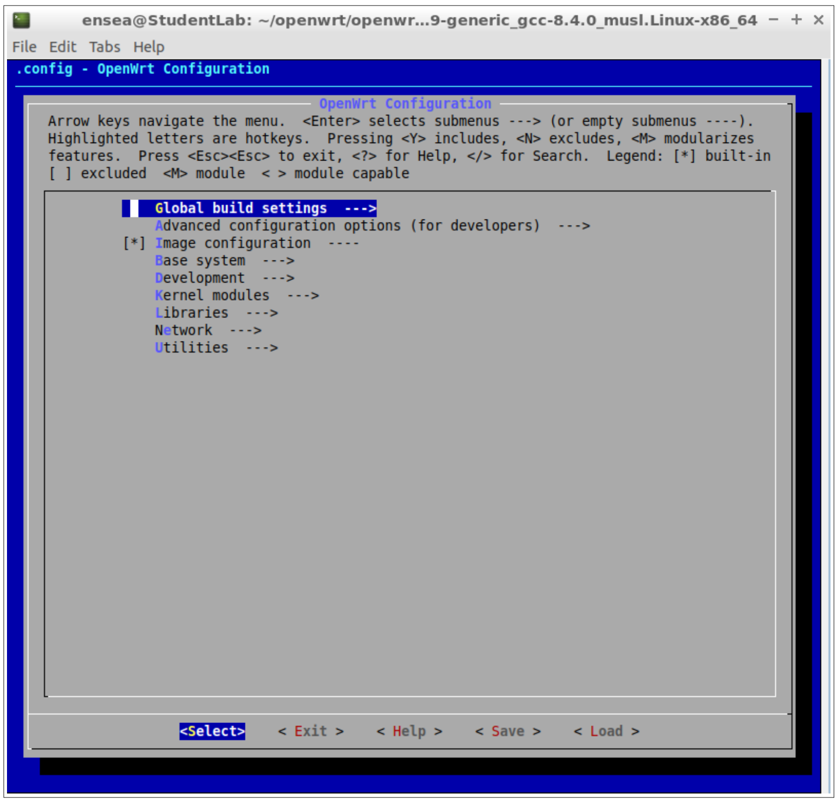
Now we can install the library by configuring the menu in our SDK



```
ensea@StudentLab:~/openwrt/openwrt-sdk-21.02.3-ath79-generic_gcc-8.4.0_musl.Linux-x86_64$ make menuconfig
```

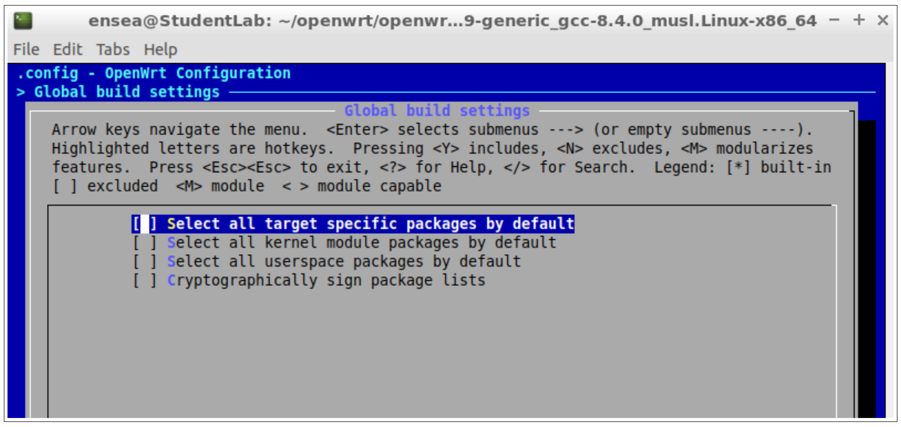
6. 3. Configure and rebuild our SDK

After this, you will get the main page :



```
ensea@StudentLab: ~/openwrt/openwr...9-generic_gcc-8.4.0_musl.Linux-x86_64
File  Edit  Tabs  Help
.config - OpenWrt Configuration
                          OpenWrt Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
  features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in
  [ ] excluded  <M> module  < > module capable

            Global build settings  --->
            Advanced configuration options (for developers)  --->
        [*] Image configuration  ----
            Base system  --->
            Development  --->
            Kernel modules  --->
            Libraries  --->
            Network  --->
            Utilities  --->

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

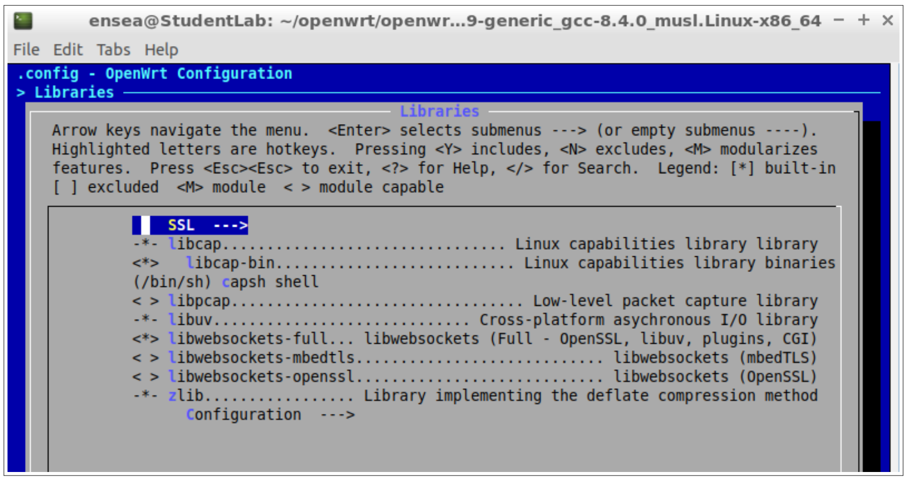
It’s useless to rebuild the OpenWRT kernel and modules and it will be a terrible loss of time. That is why we enter the « Global build settings » submenu and we deselect all the four items by pressing the letter N when we select the item.



```
ensea@StudentLab: ~/openwrt/openwr...9-generic_gcc-8.4.0_musl.Linux-x86_64
File  Edit  Tabs  Help
.config - OpenWrt Configuration
> Global build settings
                          Global build settings
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
  features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in
  [ ] excluded  <M> module  < > module capable

        [ ] Select all target specific packages by default
        [ ] Select all kernel module packages by default
        [ ] Select all userspace packages by default
        [ ] Cryptographically sign package lists
```

Back to the main menu, we enter the « Libraries » submenu and we can see the libraries we downloaded earlier (/scripts/feeds intall ...)

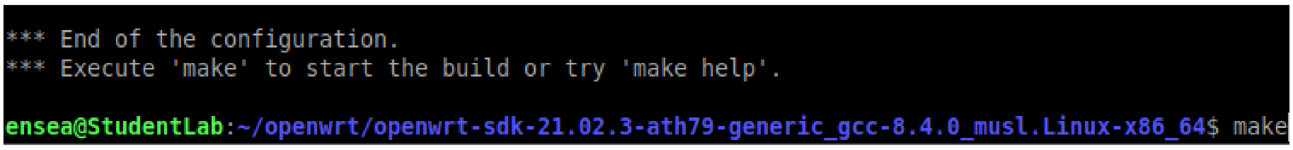
We select the libraries we want (libwebsocket-full and its dependancies in my case) by pressing the letter Y.



```
ensea@StudentLab: ~/openwrt/openwr...9-generic_gcc-8.4.0_musl.Linux-x86_64
File  Edit  Tabs  Help
.config - OpenWrt Configuration
> Libraries ──────────────────────────────────────────────────────
                          Libraries
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
  features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in
  [ ] excluded  <M> module  < > module capable

          SSL  --->
      -*- libcap.............................. Linux capabilities library library
      <*>   libcap-bin......................... Linux capabilities library binaries
      (/bin/sh) capsh shell
      < > libpcap.............................. Low-level packet capture library
      -*- libuv............................ Cross-platform asychronous I/O library
      <*> libwebsockets-full... libwebsockets (Full - OpenSSL, libuv, plugins, CGI)
      < > libwebsockets-mbedtls........................... libwebsockets (mbedTLS)
      < > libwebsockets-openssl........................... libwebsockets (OpenSSL)
      -*- zlib................. Library implementing the deflate compression method
            Configuration  --->
```

Then we confirm ".config" as the filename to save, and then simply exit until getting back to prompt.

As soon as you will get back to prompt, you'll see something similar to this and so you can rebuild the SDK with the new libraries by typing the command « make ».



```
*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.

ensea@StudentLab:~/openwrt/openwrt-sdk-21.02.3-ath79-generic_gcc-8.4.0_musl.Linux-x86_64$ make
```

It can take several minutes to rebuild the SDK.

6. 4. Find the libraries and link them with the compiler

After you have finished rebuilding the SDK, you can find the library header files in the following path : /staging_dir/toolchain-mips_24kc_gcc-8.4.0_musl/include

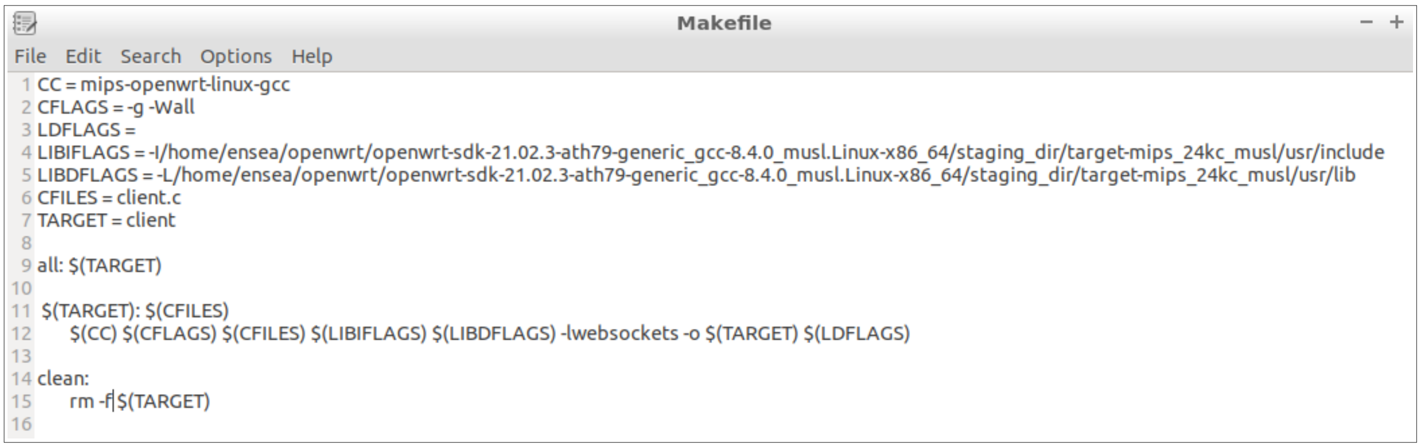
and the library files (.a, .so) in the following path : /staging_dir/toolchain-mips_24kc_gcc-8.4.0_musl/lib

My advice is to copy all the files in these directories and copy them to the following directories to link them with the SDK compiler.

So you copy the headers in « /staging_dir/toolchain-mips_24kc_gcc-8.4.0_musl/include » and copy them in « /staging_dir/target-mips_24kc_musl/usr/include »

And copy the library files in « /staging_dir/toolchain-mips_24kc_gcc-8.4.0_musl/lib » and copy them in « /staging_dir/target-mips_24kc_musl/usr/lib »

After that, we are ready to compile with our new library. We just need to compile by linking the libraries with the compiler.
To do this, here is an example of a Makefile to compile with the websockets library.



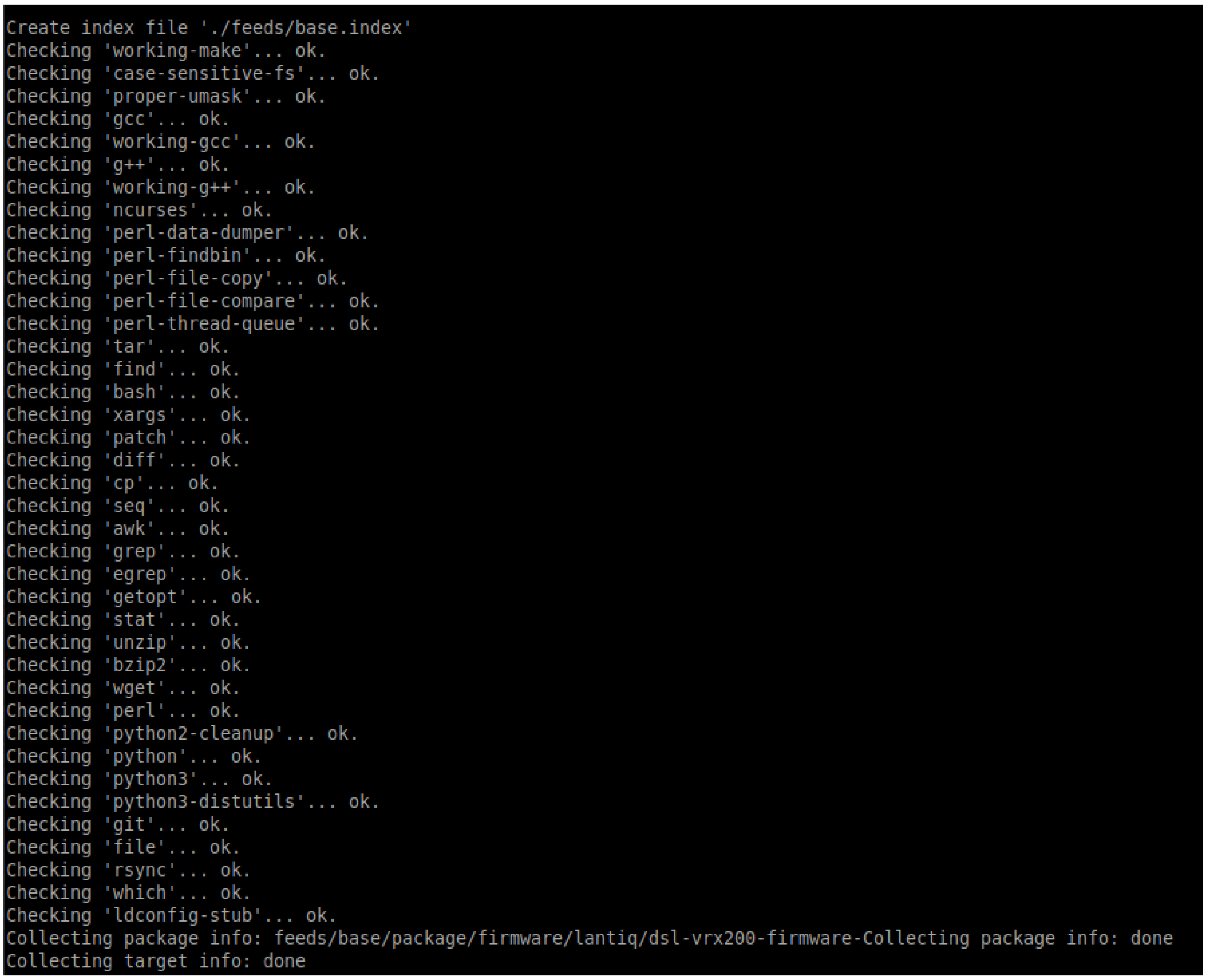
```
CC = mips-openwrt-linux-gcc
CFLAGS = -g -Wall
LDFLAGS =
LIBIFLAGS = -I/home/ensea/openwrt/openwrt-sdk-21.02.3-ath79-generic_gcc-8.4.0_musl.Linux-x86_64/staging_dir/target-mips_24kc_musl/usr/include
LIBDFLAGS = -L/home/ensea/openwrt/openwrt-sdk-21.02.3-ath79-generic_gcc-8.4.0_musl.Linux-x86_64/staging_dir/target-mips_24kc_musl/usr/lib
CFILES = client.c
TARGET = client

all: $(TARGET)

 $(TARGET): $(CFILES)
	$(CC) $(CFLAGS) $(CFILES) $(LIBIFLAGS) $(LIBDFLAGS) -lwebsockets -o $(TARGET) $(LDFLAGS)

clean:
	rm -f $(TARGET)

```

7. Potential problems encountered

7. 1. Problems with prerequisites packages

If you can’t update your SDK with the command « ./scripts/feeds update -a », you have probably an error with one of your packages or you are missing a package.

To check this, you can see on the prompt the error you have when you type the first command to update the SDK : « ./scripts/feeds update -a » and see if all the prerequisites packages are working correctly as on the image below :



```
Create index file './feeds/base.index'
Checking 'working-make'... ok.
Checking 'case-sensitive-fs'... ok.
Checking 'proper-umask'... ok.
Checking 'gcc'... ok.
Checking 'working-gcc'... ok.
Checking 'g++'... ok.
Checking 'working-g++'... ok.
Checking 'ncurses'... ok.
Checking 'perl-data-dumper'... ok.
Checking 'perl-findbin'... ok.
Checking 'perl-file-copy'... ok.
Checking 'perl-file-compare'... ok.
Checking 'perl-thread-queue'... ok.
Checking 'tar'... ok.
Checking 'find'... ok.
Checking 'bash'... ok.
Checking 'xargs'... ok.
Checking 'patch'... ok.
Checking 'diff'... ok.
Checking 'cp'... ok.
Checking 'seq'... ok.
Checking 'awk'... ok.
Checking 'grep'... ok.
Checking 'egrep'... ok.
Checking 'getopt'... ok.
Checking 'stat'... ok.
Checking 'unzip'... ok.
Checking 'bzip2'... ok.
Checking 'wget'... ok.
Checking 'perl'... ok.
Checking 'python2-cleanup'... ok.
Checking 'python'... ok.
Checking 'python3'... ok.
Checking 'python3-distutils'... ok.
Checking 'git'... ok.
Checking 'file'... ok.
Checking 'rsync'... ok.
Checking 'which'... ok.
Checking 'ldconfig-stub'... ok.
Collecting package info: feeds/base/package/firmware/lantiq/dsl-vrx200-firmware-Collecting package info: done
Collecting target info: done
```

All packages must be « ok » otherwise check the package that does not work. Either you didn't install it or it doesn't work properly and you have to reinstall it or update it.