EMBEDDED SYSTEMS TRAINING

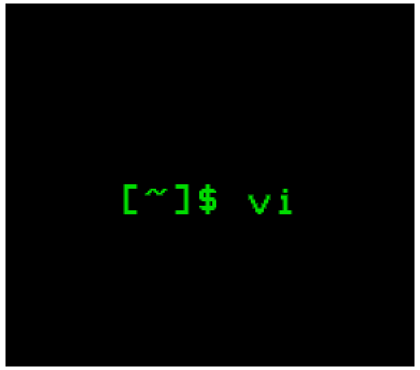
Khóa học : “Basic Embedded Linux”

- **Email** : *training.laptrinhnhung@gmail.com*
- **Website** : *http://hethongnhung.com* – *http://laptrinhnhung.com*

Text editor & Compilation

Vi Editor

- ✓One of the most popular text editor in UNIX
  - ✓O'Reilly says that its Vim tutorial sells twice as many copies as for Emacs
- ✓Ubiquitous
  - ✓Runs on all UNIX machines
  - ✓Essentially all unix and unix-like systems come with vi built-in
- ✓vim : extended version of vi



```
[~]$ vi
```

Text Editor in UNIX : vi

- To invoke vi, type “vi” in shell prompt

```
$vi [file [file …]]
```

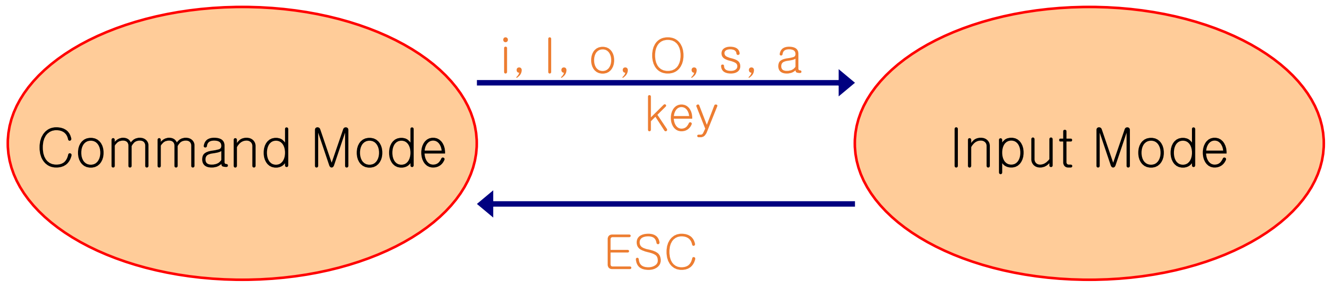
- Filename is often specified.
- It is not required to specifying filename

vi mode

- Input (insertion) mode
  - Actually edit the text

- Command (escape) mode
  - The keyboard input is interpreted as a command

- Mode switch



Quit vi

- :wq
  - Save current file and quit
- zz
  - Same as :wp

- :q
  - Just quit
- :q!
  - Quit without save

Cursor movement

h	: move backward by letter
j	: move down to the next line
k	: move up to the previous line
l	: move forward by letter
b	: move backward by word
w	: move forward by word
$	: move to the end of the current line
^	: move to the beginning of the current line
Ctrl-F	: move forward to next screen
Ctrl-B	: move backward to previous screen

Edit Mode Commands

- x
  - delete the letter beneath the cursor
- r
  - Replace current character
- u
  - undo
- cw
  - Change current word
- dw
  - Delete current word

- Cut, Copy, Paste
  - dd
    - cut (delete) current line
  - yy
    - Copy current line
  - p
    - paste

Pattern search

- In command mode, press ‘/’ or ‘?’ and search pattern (a word or words)
  - /search_pattern
    - Search after cursor
  - ?search_pattern
    - Search before cursor

Line number

- Display line number
  - :set number
  - :set nu
- Remove line number
  - :set nonumber
  - :set nonu

- :1
  - Go to line 1
- :25
  - Go to line 25

Switch to insertion mode

- i
  - Insert characters
- a
  - Append characters
- o
  - Open a new line below the current line and add text
- I
  - Insert text at the beginning of the current line
- A
  - Add text to the end of the current line
- O
  - Open a new line above the current line and add text

replace

- :s/pattern/replace
  - Replace in current line

- :%s/pattern/replace
  - Globally replace

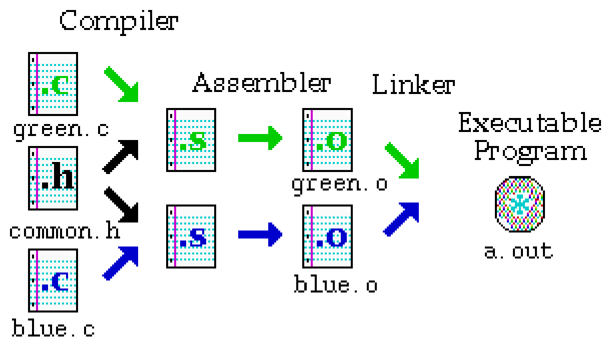
- Example) :s/from/FROM

File Save

- :w
- :w filename
- :wq

C/C++ compile

- UNIX itself was written in C
- Most UNIX utilities and applications were written in C/C++
- It is hard to understand UNIX without understanding C/C++
- Some UNIX has its own C/C++ compiler and some other UNIX do not.
- GNU C/C++ compiler is commonly used
- C compiler
  - cc [-option] C-files
  - gcc [-option] C-files



C/C++ compile

```
$ gcc hello.c
$ g++ hello.cpp
```

Source codes (hello.c , hello.cpp) are compiled

executable file (a.out) is created

Compile option

- `-c`

```
$ gcc -c hello.c
```

  - only object code hello.o is created.

- `-o`
  - Specify the name of executable file

```
$ gcc -o hello hello.c
```

or

```
$ gcc -o hello hello.o    (when hello.o already exists)
```

- `-g` : debug mode

Makefile

Why Makefile?

Compiling the source code files can be tiring, especially when you have to include several source files and type the compiling command everytime you need to compile. Makefiles are the solution to simplify this task. Makefiles are special format files that help build and manage the projects automatically.

For example, let's assume we have the following source files.

- main.cpp
- hello.cpp
- factorial.cpp
- functions.h

Makefile

main.cpp:

```
#include <iostream.h>
#include "functions.h"
int main(){
    print_hello();
    cout << endl;
    cout << "The factorial of 5 is " << factorial(5) << endl;
    return 0;
}
```

Makefile

hello.cpp:

```
#include <iostream.h>
#include "functions.h"
void print_hello(){
        cout << "Hello World!";
}
```

Makefile

factorial.cpp:

```
#include "functions.h"
int factorial(int n){
        if(n!=1){
                return(n * factorial(n-1));
        }
        else return 1;
}
```

Makefile

functions.h:

```
void print_hello();
int factorial(int n);
```

Makefile

- The trivial way to compile the files and obtain an executable is by running the command:

```
CC main.cpp hello.cpp factorial.cpp -o hello
```

- This command generates hello binary.
- In this example, we have only four files and we know the sequence of the function calls.
- Hence, it is feasible to type the above command and prepare a final binary.
- However, for a large project where we have thousands of source code files, it becomes difficult to maintain the binary builds

Makefile

- The make command allows you to manage large programs or groups of programs.
- As you begin to write large programs, you notice that re-compiling large programs takes longer time than re-compiling short programs.
- Moreover, you notice that you usually only work on a small section of the program such as a single function, and much of the remaining program is unchanged.
- In the subsequent section, we see how to prepare a makefile for our project.

Makefile

- Makefile specifies dependency of source files for correct and effective compilation

```
$make [-f makefile_name]
```

Default : `makefile` or `Makefile` as input

Makefile

- To manage executable file using “make”, Makefile must be created first
- Makefile includes interdependency list of source files
- Makefile sometimes has .mk suffix
- Makefile format

```
targetList: dependencyList
    commandList
```

  - `targetList` :object file (or executable file) list
  - `dependencyList` : or files that are dependent on the files in targetList
  - `commandList` : command that generates object file from files in dependencyList

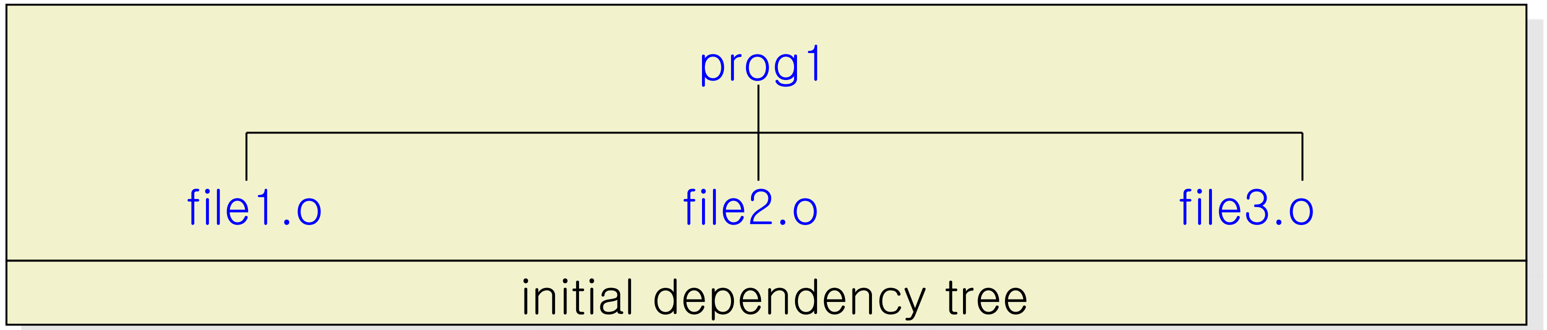
`commandList` should start with “tab”

Makefile Example

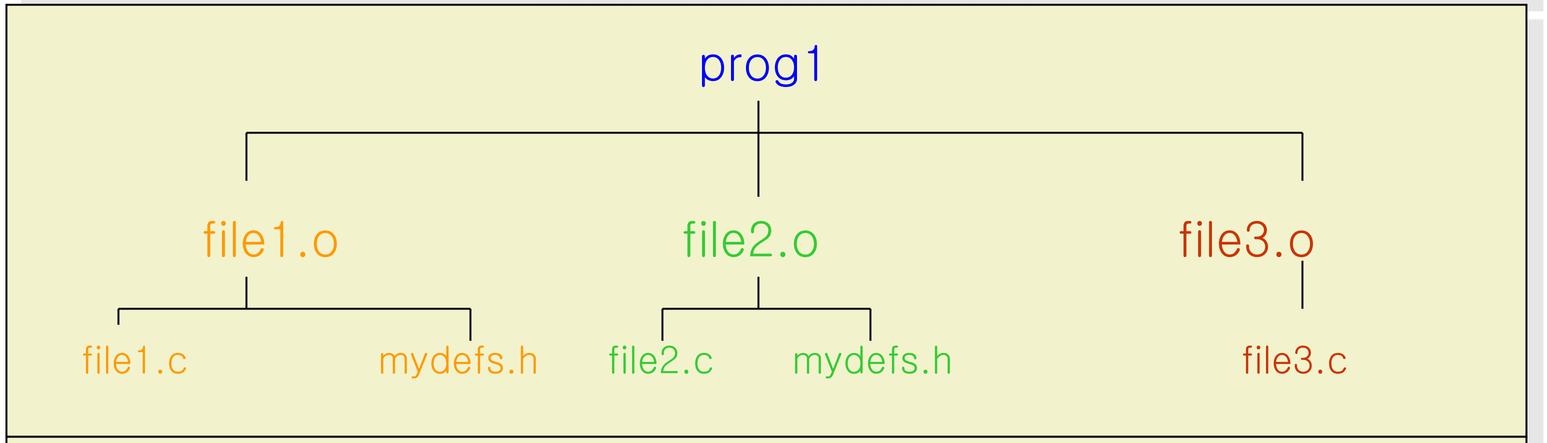
```
prog1: file1.o file2.o file3.o
        gcc -o prog1 file1.o file2.o file3.o
# if one of file1.o, file2.o, and file3.o is modified,
#   , above linking is performed to (re)generate “prog1”

file1.o: file1.c mydefs.h
  gcc -c file1.c
# if one of file1.c and mydefs.h is modified,
#  above compilation is performed to generate file1.o

file2.o: file2.c mydefs.h
  gcc -c file2.c
file3.o: file3.c
  gcc -c file3.c

clean :
  rm file1.o file2.o file3.o
```

Make : dependency tree

- Executable file becomes root of the tree
- Target becomes parent node, and dependency files become children nodes.



initial dependency tree



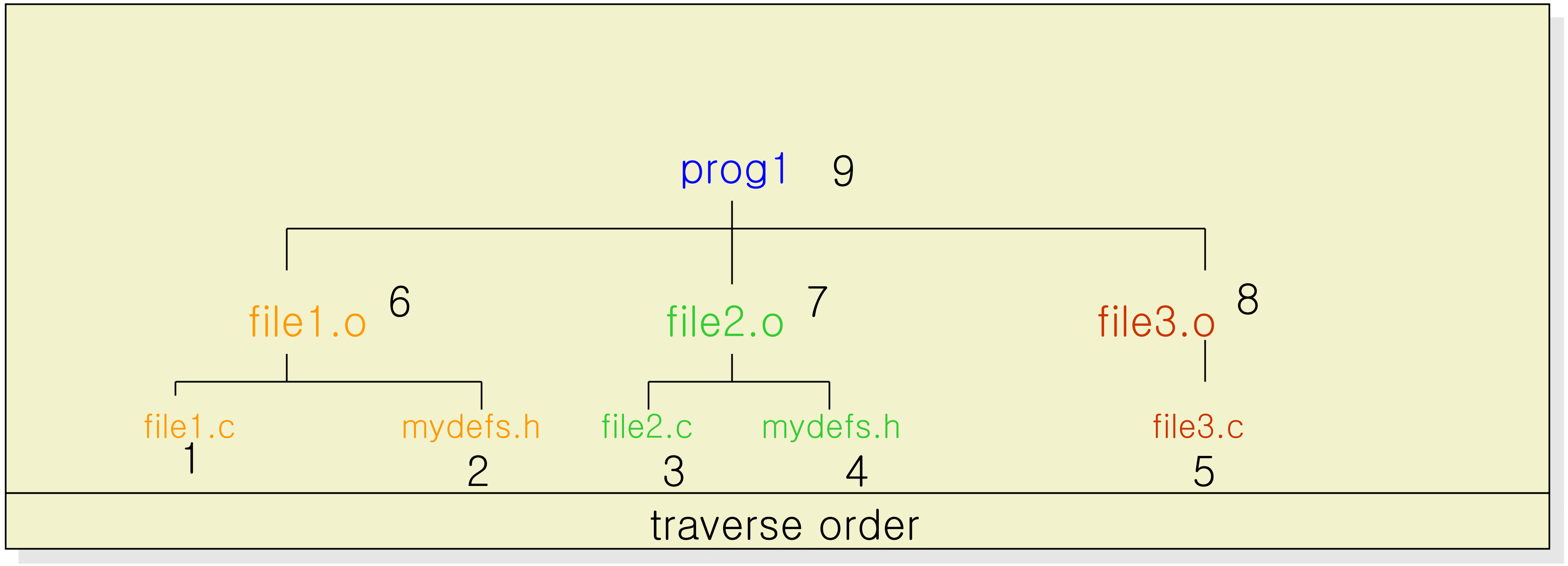
final dependency tree

Compilation Steps

- make finds the target prog1 and sees that it depends on the object files file1.o file2.o file3.o
- make next looks to see if any of the three object files are listed as targets. They are so make looks at each target to see what it depends on. make sees that file1.o depends on the files file1.cc and mydefs.h.
- Now make looks to see if either of these files are listed as targets and since they aren't it executes the commands given in file1.o's rule and compiles file1.cc to get the object file.
- make looks at the targets file2.o and file3.o and compiles these object files in a similar fashion.
- make now has all the object files required to make prog1 and does so by executing the commands in its rule.

Traverse Dependency Tree During Compilation

- Traverse from leaf nodes to root node
- Check if modification time of parent node is earlier than modification time of child node.
- If so, perform the compilation for updating the parent node



Phony Target

- Target that is not a name of a file
- Just have a list of commands to execute
- Common example : removing all object files

```
clean :
    rm *.o
```

```
> make clean
```

Macro

- Example

```
OBJS = file1.o file2.o file3.o
CC = gcc
prog1 : ${OBJS}
   ${CC} -o prog1 ${OBJS}
file1.o : file1.c mydefs.h
   ${CC} -c file1.c
file2.o : file2.c mydefs.h
   ${CC} -c file2.c
file3.o : file3.c
   ${CC} -c file3.c

clean :
   rm ${OBJS}
```

- Command line macro

```
% make prog1 DEBUG_FLAG=-g
```

Internal Macro

- Internally defined macros are ones that are predefined in make
- make –p

```
CXX = CC
CXXFLAGS = -O
GFLAGS =
CFLAGS = -O
CC = cc
LDFLAGS =
LD = ld
LFLAGS =
MAKE = make
MAKEFLAGS = b
```

Suffix rule

- “make” has a set of default rules to build a program
- (ex) for c program, .o object files are made from .c source files

- The suffix rule that make uses for a C program is

```
.c.o :
    ${CC} -c $<
```

  - $< means the source file of the current (single) dependency.

Shortened Makefile

```
OBJS = file1.o file2.o file3.o

prog1 : ${OBJS}
  ${CC} -o $@ ${OBJS}   # CC is internally defined
                                  # $@ means current target filename

file1.o file2.o : mydefs.h

clean : rm ${OBJS}
```

Cross Compiler ToolChain

Setup Cross Compiler Toolchain

- Step 1 : Download source for toolchain

```
wget -c https://releases.linaro.org/components/toolchain/binaries/5.3-2016.02/arm-linux-gnueabihf/gcc-linaro-5.3-2016.02-x86_64_arm-linux-gnueabihf.tar.xz
```

- Step 2 : Extract

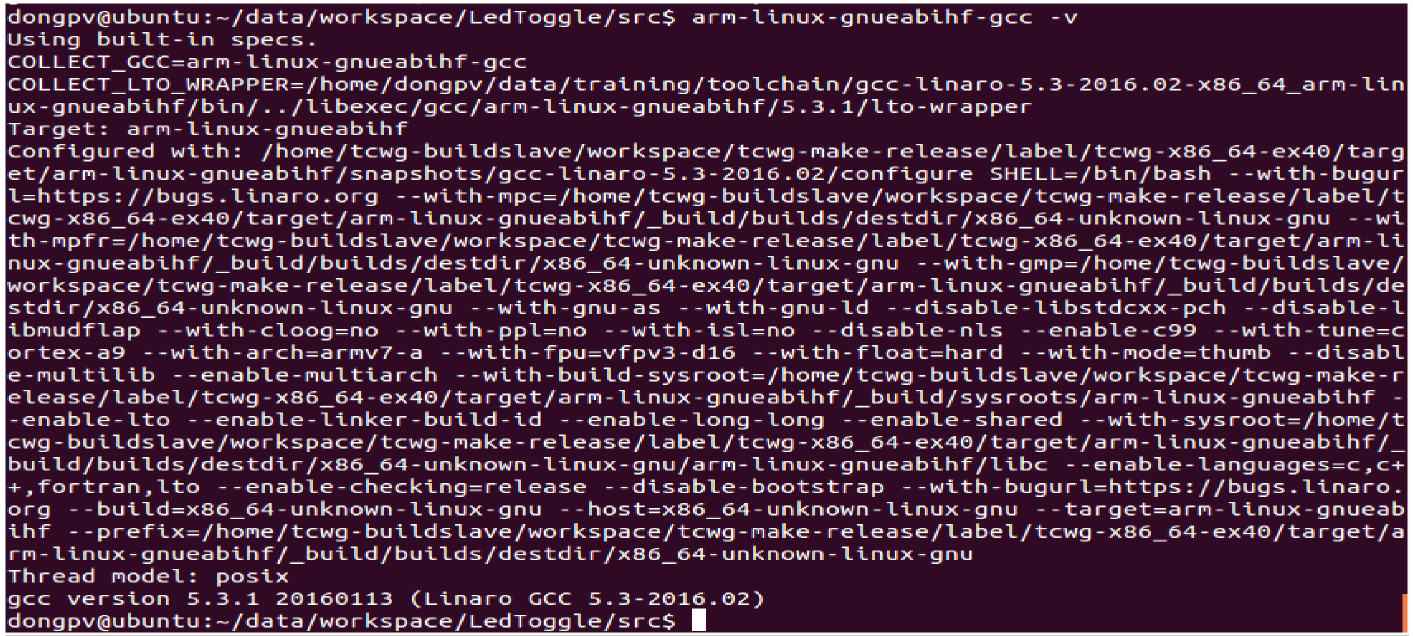
```
tar xf gcc-linaro-5.3-2016.02-x86_64_arm-linux-gnueabihf.tar.xz
```

- Step 3 : Update PATH environment variable

```
export PATH=$PATH:/home/dongpv/data/training/toolchain/gcc-linaro-5.3-2016.02-x86_64_arm-linux-gnueabihf/bin
```

- Step 4 : Check information of cross compiler toolchain

```
arm-linux-gnueabihf-gcc -v
```



```
dongpv@ubuntu:~/data/workspace/LedToggle/src$ arm-linux-gnueabihf-gcc -v
Using built-in specs.
COLLECT_GCC=arm-linux-gnueabihf-gcc
COLLECT_LTO_WRAPPER=/home/dongpv/data/training/toolchain/gcc-linaro-5.3-2016.02-x86_64_arm-lin
ux-gnueabihf/bin/../libexec/gcc/arm-linux-gnueabihf/5.3.1/lto-wrapper
Target: arm-linux-gnueabihf
Configured with: /home/tcwg-buildslave/workspace/tcwg-make-release/label/tcwg-x86_64-ex40/targ
et/arm-linux-gnueabihf/snapshots/gcc-linaro-5.3-2016.02/configure SHELL=/bin/bash --with-bugur
l=https://bugs.linaro.org --with-mpc=/home/tcwg-buildslave/workspace/tcwg-make-release/label/t
cwg-x86_64-ex40/target/arm-linux-gnueabihf/_build/builds/destdir/x86_64-unknown-linux-gnu --wi
th-mpfr=/home/tcwg-buildslave/workspace/tcwg-make-release/label/tcwg-x86_64-ex40/target/arm-li
nux-gnueabihf/_build/builds/destdir/x86_64-unknown-linux-gnu --with-gmp=/home/tcwg-buildslave/
workspace/tcwg-make-release/label/tcwg-x86_64-ex40/target/arm-linux-gnueabihf/_build/builds/de
stdir/x86_64-unknown-linux-gnu --with-gnu-as --with-gnu-ld --disable-libstdcxx-pch --disable-l
ibmudflap --with-cloog=no --with-ppl=no --with-isl=no --disable-nls --enable-c99 --with-tune=c
ortex-a9 --with-arch=armv7-a --with-fpu=vfpv3-d16 --with-float=hard --with-mode=thumb --disabl
e-multilib --enable-multiarch --with-build-sysroot=/home/tcwg-buildslave/workspace/tcwg-make-r
elease/label/tcwg-x86_64-ex40/target/arm-linux-gnueabihf/_build/sysroots/arm-linux-gnueabihf -
-enable-lto --enable-linker-build-id --enable-long-long --enable-shared --with-sysroot=/home/t
cwg-buildslave/workspace/tcwg-make-release/label/tcwg-x86_64-ex40/target/arm-linux-gnueabihf/_
build/builds/destdir/x86_64-unknown-linux-gnu/arm-linux-gnueabihf/libc --enable-languages=c,c+
+,fortran,lto --enable-checking=release --disable-bootstrap --with-bugurl=https://bugs.linaro.
org --build=x86_64-unknown-linux-gnu --host=x86_64-unknown-linux-gnu --target=arm-linux-gnueab
ihf --prefix=/home/tcwg-buildslave/workspace/tcwg-make-release/label/tcwg-x86_64-ex40/target/a
rm-linux-gnueabihf/_build/builds/destdir/x86_64-unknown-linux-gnu
Thread model: posix
gcc version 5.3.1 20160113 (Linaro GCC 5.3-2016.02)
dongpv@ubuntu:~/data/workspace/LedToggle/src$
```

Makefile for ARM example 1

```
CC=arm-linux-gnueabihf-gcc
all:  hello.c
        $(CC) -g -o hello hello.c
clear:
        rm hello
```

- Compile application : make all
- Clean : make clean

Makefile for ARM example 2

```
CC=arm-linux-gnueabihf-g++
OUTPUT=hello
all: main.o hello.o factorial.o
	$(CC) main.o hello.o factorial.o -o $(OUTPUT)
main.o : main.cpp functions.h
	$(CC) -c main.cpp
factorial.o : factorial.cpp functions.h
	$(CC) -c factorial.cpp
hello.o : hello.cpp functions.h
	$(CC) -c hello.cpp
```

Questions?