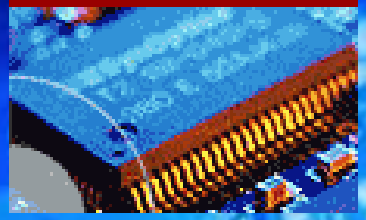
EEE3096S



prac lecture

Embedded Systems II

Makefiles

Lecture P2

Embedded Systems II

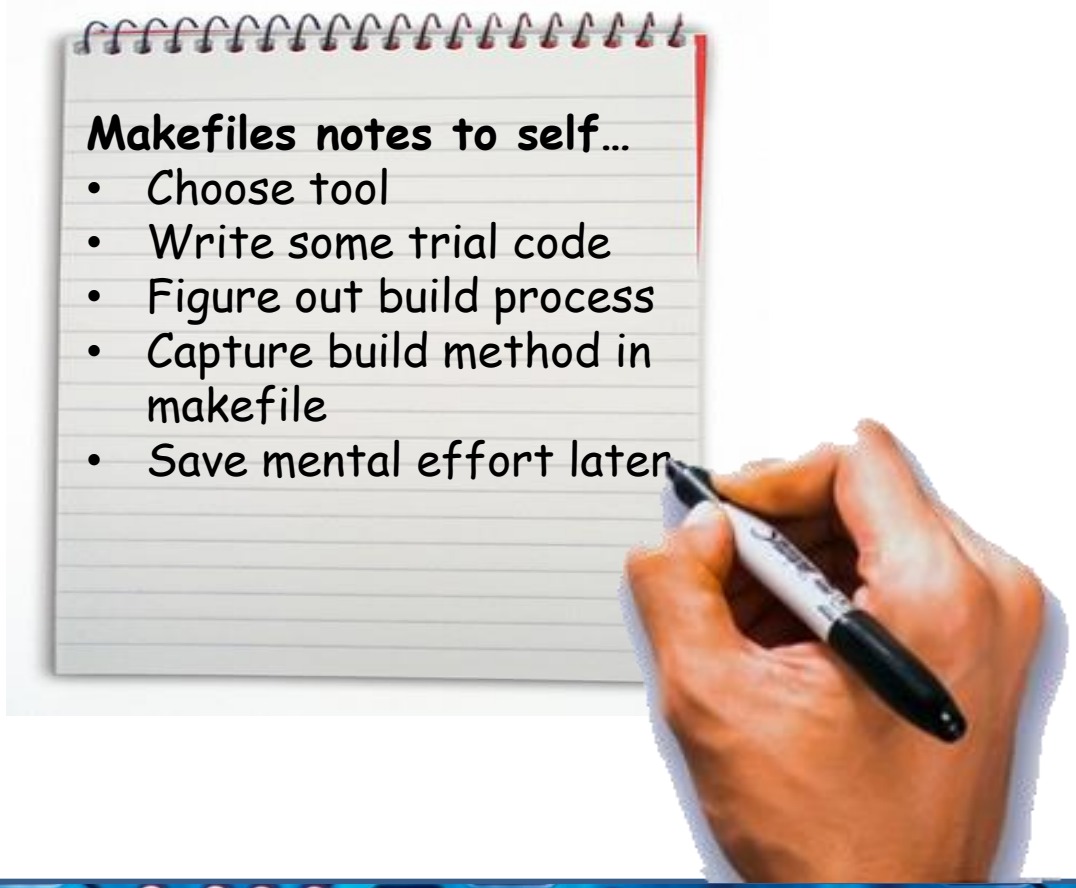
Dr Simon Winberg

Electrical Engineering
University of Cape Town

Outline

- Introducing makefiles
- Planning a makefile
- Makefile example

The delights of writing Makefiles …



Makefile

See tutorials (related to prac 1, GDB) for getting properly into using makefiles. These slides cover just some essentials…

- The first thing, before you attempt writing a makefile is to know how the application you want to make is generated. Knowing of whatever compilers, assemblers, linkers and other tools to go from the code to an executable*.
- When talking about C and using GCC, the essential thing you should hopefully know already is basics of the gcc command, for instance the following compilers and links code files in one go:

```
$ gcc -o myfile main.c util.c
```

* Note that makefiles are not limited to just making executables from (e.g.) C or compliable code. They can be used for generating other kinds of execution objects which might involve generating some sort of runtime environment in which the exucatable, or executables, run within.

Makefile

- Before making a makefile, you should know what tools to use, and the sequencing of these tools, for building your executable.
- There may be various other commonly used operations related to the build process, that might not specifically be part of the build process but needed in starting or setting up the process.
  - An example of such a process is the 'clean' process, which deletes any intermediary files (e.g. object files) and the previously compiled executable. This generally forces the build processes to remake the executable from scratch the next time it is run.
  - Other common process are:
    - 'mark': which is not really what tutors use for marking your code (it could be!) but rather some companies use it to make a significant state of the project e.g. mark a build in some way, possibly sending it to a repository.
    - 'backup': obvious one this. Apply some backup operation, maybe zipping up the project files and uploading it to some file store.
    - 'test': either generating testing code (sometimes one generates automated testing code, e.g. a separate project that will test your project) and/or runs tests through your already compile application.

Now, I'm hoping you are more appreciating the Power of Makefiles!

Makefile example

- To illustrate the process, let us consider we have the following files:
  - test1.c with H-file test1.h
  - We want to compile test1.c with gcc and we want to generate a GDB debugger compatible executable, which needs the -g flag to be specified. We want to call the generated executable test1 (an elf file, not needing extension if done on Linux)
- We could do this easily with just gcc:

  gcc -g -o test1 test1.c
- But this isn't scalable. And we haven't capture information (e.g. to future developers or users) of how to build the program. If we were going to make a more complex application needing possibly hundreds of files and different tools and compiler flags to generate some of them, doing it in this way would be very clunky if even possible.
- Thus, lets convert this process into a makefile…

Planning makefile example

- Let us plan this simple build method.
- It would be useful these two processes available:
  - A clean process which deletes all intermediary files and the executable.
  - A build process (the default process) that will generate the executable if any of its dependencies (as in the .c or .h files) have changed since the last attempt at building
- I've explained a basic attempt in the next side

A super simple makefile

We first define some aliases (or variables), referring to command we might change later. Typically CC is used to refer to the C compiler. If we wanted to use a different compiler, e.g. xc8, to compile this for an 8-bit PIC processor, you could just set CC to xc8 instead of gcc. Assuming you have the needed tools installed.

```
CC=gcc
# note the -g below for debugging
CFLAGS=-I. -g

test1.o: test1.c test1.h
	$(CC) -c -o test.o $< $(CFLAGS)

test1: test1.o
	$(CC) -o $@ $^ $(CFLAGS)

clean:
	rm -f -r test1 test1.o
```

Here's how to do a comment. You can use a hash #.

I've defined a variable for the compiler flags I want to use. Thus I don't need to copy them each time the compiler is called. It also allows easy changes to compiler flags to use to do a build.

Dependency definition. Notation for this line is: *output_file* : *list of dependencies*

This is the command to generate the *output_file* (i.e. test.o)

This is the command that will generate test1, which make will assume is our target because it is the root of the tree. Clean has no dependencies listed so will be ignored, only active if put as the command line argument to make.

Key to special variables:

$@ : the name of the output to generate, the LHS of the dependency rule.

$< : filename of first prerequisite in the dependency definition. For the first dependency definition, i.e. in the entry "test1.o: test1.c …" the first prerequisite corresponds to test.c. So to make test1.o it will call the command "gcc -c -o test.o test.c -I. -g".

$^ : This corresponds to the filenames of all the prerequisites, separated by spaces. So, for the case "test1: test1." the command that will be invoked is "gcc -o test1 test1.o -I. -g".

What is the ultimate target or root target? i.e. final executable? Answer: the makefile will automatically decide test1 is the root target, at the base of the dependencies (there are no entries dependent on test1… And no, clean is not dependent on test1 as test1 is not in its dependency definition. So, test1 will be decided as the file you want to make by default, when calling make with no arguments).

Power Up the Make game!

- Great. You have passed level 1 of how to make a makefile!

While I'm not going to get into much more depth of makefiles in this course, the next slide nevertheless takes things a little further…
Click to continue.

Makefile demonstrating wildcards and macros

```
CC=gcc
# note the -g below for debugging
CFLAGS=-I. -g
DEPS = test1.h
OBJ = test1.o

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

test1: $(OBJ)
	$(CC) -o $@ $^ $(CFLAGS)

clean:
	rm -f -r test1 *.o
```

You can do some fairly fancy things in makefiles, which can save a lot of writing. One such is the use of macros and **special variables** (which were mentioned before and copied below). This is essentially defining a dependency rule that says that each .o file (call it %) depends on a file of that name with .c extension (i.e. a file called %.c). In addition to those files listed in DEPS. I'm use of DEPS here is potentially inefficient, it a shortcut I open use to just list all .h files in a project, stating (possibly inaccurately) that every compile done depends on all h files in the project.

% will be tested for both LHS and RHS. So if you have a test.c but no test.o it will see the test.c and generate a dependency test.o : test.c even if the .o does not exist. If it sees a duplicate dependency rule it ignores it and carries on.

$@ indicates the the LHS (dependency rule's target). So if it puts test.o here.

$^ is substituted by all files o the RHS of the dependency rule.

Key to special variables:

$@ : the name of the output to generate, the LHS of the dependency rule.

$< : filename of first prerequisite in the dependency definition. For the first dependency definition, i.e. in the entry "test1.o: test1.c …" the first prerequisite corresponds to test.c. So to make test1.o it will call the command "gcc -c -o test.o test.c -I. -g".

$^ : This corresponds to the filenames of all the prerequisites, separated by spaces. So, for the case "test1: test1." the command that will be invoked is "gcc -o test1 test1.o -I. -g".

What is the ultimate target or root target? i.e. final executable? Answer: the makefile will automatically decide test1 is the root target, at the base of the dependencies (there are no entries dependent on test1… And no, clean is not dependent on test1 as test1 is not in its dependency definition. So, test1 will be decided as the file you want to make by default, when calling make with no arguments).

Makefile – Game Over!

- Hope that has helped you gain some understanding of makefile syntax and the power of this tool.
- I realize that may have been rather dense information, and you may well feel it's Game Over for Makefiles. But do give them a try as they are used extensively by embedded developers, so it is worth knowing a bit about them!

Recommended further reading / experimenting:

https://makefiletutorial.com : a thorough makefile coding tutorial. Try this if these slides gave you too little info and you are struggling to figure out coding of makefiles.

(optional reading, not examined)

CMake

If you happen to still have some Makefile lives left, you may want to explore cmake. CMake takes Make further, as per the Cmake website it is defined as:

> "CMake is an open-source, cross-platform family of tools designed to build, test and package software."
> https://cmake.org/

CMake can be considered an abbreviation for "cross-platform make". The C *does not* stand for C or C++, as this tool can be used for many different languages and tools, besides C.

For some quick information, links to online tutorials and manual pages for cmake, see: http://wiki.ee.uct.ac.za/Cmake

End of Lecture