Guide to Make for ARM cross-development

*Written for CS107E by Anna Zeng, Michelle Park, and Julie Zelenski*

What is a Makefile?

Make is a tool that automates building programs; a Makefile describes the commands and options used in the build process. As you will see soon enough, using a Makefile saves you a lot of retyping and makes your life as a developer a whole lot smoother!

This guide introduces Makefiles using examples of cross-development for the ARM architecture. For further information about Makefiles in general, check out the [other resources](http://cs107e.github.io/guides/make/#resources) on the bottom of this page.

An example Makefile

The example makefile below builds the blink.bin program out of the blink.c source file. Our labs and assignments will include similar Makefiles, so you will soon become familiar with the common structure.

NAME = blink

CFLAGS = -g -Wall -Og -std=c99 -ffreestanding

LDFLAGS = -nostdlib

all: $(NAME).bin

%.bin: %.elf

arm-none-eabi-objcopy $< -O binary $@

%.elf: %.o

arm-none-eabi-gcc $(LDFLAGS) $< -o $@

%.o: %.c

arm-none-eabi-gcc $(CFLAGS) -c $< -o $@

%.list: %.o

arm-none-eabi-objdump -d $< > $@

install: $(NAME).bin

rpi-install.py $<

clean:

rm -f \*.o \*.elf \*.bin \*.list

This Makefile may look a bit cryptic at first! Let’s try breaking it down step by step.

Rules and recipes

In lecture, Pat whipped up a simple [doit script](http://cs107e.github.io/lectures/ASM/code/blink/doit) to automate retyping the commands to rebuild a program. The make tool is just a fancier version of doit. A Makefile is a text file that describes the steps needed to build a program. Here is an example of a very simple hard-coded Makefile containing three targets all, button.bin and clean:

all: button.bin

button.bin: button.c

arm-none-eabi-gcc -Og -g -Wall -std=c99 -ffreestanding -c button.c -o button.o

arm-none-eabi-gcc -nostdlib button.o -o button.elf

arm-none-eabi-objcopy button.elf -O binary button.bin

clean:

rm -f \*.bin \*.o

**Rules** are written in the following way: “the **dependencies** on the right-hand-side are required to make the **target** on the left-hand-side.” Thus the first line indicates that button.bin is required to make all. In other words, to make all, we must first make button.bin.

all: button.bin

This brings us to the next rule for button.bin:

button.bin: button.c

arm-none-eabi-gcc -Og -g -Wall -std=c99 -ffreestanding -c button.c -o button.o

arm-none-eabi-gcc -nostdlib button.o -o button.elf

arm-none-eabi-objcopy button.elf -O binary button.bin

The ingredients (dependencies on the right-hand-side) are needed as the starting point to create the desired output (target on the left-hand-side). The indented lines that follow the rule are the commands that turn the ingredients into the final product. These steps are collectively called the **recipe**. Thus, in order to make button.bin, we start with our ingredient (button.c) and then step through the commands in the recipe.

We could add a comment to explain the additional flags included when invoking the compiler. Lines starting with # are treated as comments.

# Compiler flags used:

# -std=c99 use the c99 standard

# -Og generate optimized code designed for debugging

# -g add debugging information

# -Wall give warnings about \*all\* issues

# -ffreestanding generate code assuming no operating system

The final rule indicates what should happen when we make clean; the recipe for the clean target removes any previous build products so the next compile starts fresh.

clean:

rm -f \*.bin \*.o

One particularly nifty thing make is that only rebuilds a target when one or more of the components it depends on has changed. If you attempt to re-build a target which is already up-to-date, make will tell you:

$ make

make: Nothing to be done for `all'.

Macros

After repeatedly copy-pasting the example Makefile to create a version for a new program, you can see the value in structuring it to be more general-purpose. After all, Makefiles are written for convenience!

NAME = blink

CFLAGS = -std=c99 -Og -g -Wall -ffreestanding

LDFLAGS = -nostdlib

all: $(NAME).bin

$(NAME).bin: $(NAME).c

arm-none-eabi-gcc $(CFLAGS) -c $(NAME).c -o $(NAME).o

arm-none-eabi-gcc $(LDFLAGS) $(NAME).o -o $(NAME).elf

arm-none-eabi-objcopy $(NAME).elf -O binary $(NAME).bin

clean:

rm -f \*.bin \*.o

We’ve added three **macros** up top. They’re similar to variables in that they replace instances of the macro throughout the file with their assigned text. The $(macro\_name) syntax is used to access the value of the macro. This makes it easy to change the name for a new program.

Pattern rules

We can further generalize our Makefile by using *pattern rules* that can be used to operate on any source file, without hard-coding to a particular name.

# This pattern rule compiles a C program into an object file.

# filename.o is built from filename.c

%.o: %.c

arm-none-eabi-gcc $(CFLAGS) -c $< -o $@

# This pattern rule converts assembly instructions into an object file.

# filename.o is built from filename.s

%.o: %.s

arm-none-eabi-as $(CFLAGS) $< -o $@

# This pattern rule links an object file into an executable ELF file.

# filename.elf is built from filename.o

%.elf: %.o

arm-none-eabi-gcc $(LDFLAGS) $< -o $@

# This pattern rule extract binary from an ELF executable

# filename.bin is built from filename.elf

%.bin: %.elf

arm-none-eabi-objcopy $< -O binary $@

The symbols that begin with $ and % in a pattern rule are handled by make using the following interpretations:

* % is a wildcard symbol when used in a rule; %.o for example matches any file that ends with .o
* $@ refers to the left part of the rule, before the :
* $< refers to the first element in the right part of the rule, after the :

One more special variable $^ refers to all elements in the right part of the rule, after the :, which is to say all of the dependencies.

For further convenience, we can add a rule for the install target. We use this target to invoke the command rpi-install.py blink.bin to load our newly-built program on the Pi.

# The install target uploads a freshly made binary image to rpi bootloader

install: $(NAME).bin

rpi-install.py $<

With that finishing touch, you have a general Makefile that can be easily re-purposed for any Raspberry Pi project. Now that you know that a Makefile is just a cookbook that culminates in the tasty program you wish to create, you’re ready to add your favorite recipes and bon appetit!

Going further

Some follow up references on Makefiles:

* [a Makefile tutorial](http://www.opussoftware.com/tutorial/TutMakefile.htm)
* [another Makefile tutorial](http://www.delorie.com/djgpp/doc/ug/larger/makefiles.html)
* [CS107 Guide to Makefiles](https://web.stanford.edu/class/archive/cs/cs107/cs107.1186/guide/make.html)
* An inexhaustible source of make wisdom is the full manual for [GNU make](https://www.gnu.org/software/make/manual/html_node/index.html) which will tell you more that you could ever want to know.
* Reading makefiles from real world projects is a good way to see make in action. A search on [github.com](https://github.com/search?utf8=%E2%9C%93&q=makefile&type=) will turn up a treasure trove.

**Q. Make is failing with a cryptic error about Makefile: \*\*\* missing separator. What gives?**

A. In what is widely considered one of the dumber decisions in the history of computing, a Makefile distinguishes between tabs and spaces. The recipe lines for a target must begin with a tab and an equivalent number of spaces just won’t do. Edit your makefile and replace those errant spaces with a tab to restore Makefile joy.