Make and Makefiles

Abstract:

"make" is besides the compiler one of the most important software development tool. This article explains what the magic behind "make" is and how to write a Makefile.

What is make?

The command make reads a file called Makefile and analyze the rules in this file. It determines then whether a set of commands should be executed based on the time-stamp of the target-file and the time-stamps of the source files.

In other words think of a program as something that consists of multiple files which are compiled together in various steps. Make will make sure that only the chain of steps will be executed where a source file is found to be more recent than the previously compiled program file.

For very small programs one could as well write just a script (batch file) with all the compile commands one after each other. For small program not much time is saved if unnecessary steps are omitted. On the other hand makefiles are much easier to write than batch scripts. That's why we use makefiles even for small programs.

How a makefile looks like

Makefiles are just text files and they look like this:

```
# a comment starts with a hash

# a variable
MCU=atmega168

target1: prerequisites
  command1

target2: prerequisites
  command2
```

Note: The commands must be indented with a TAB, not spaces!

The targets are things that can be generated (made by the compiler) or they are just names for things to do.

If you type

make target1

Then "make" will look for a things that are listed in prerequisites of that target and if needed generate them via other targets. Once it has all the prerequisites (=files) available it will run command1.

As you can see the execution in Makefiles follows a chain of possibly nested commands because the prerequisites must first be generated before a command specified under a target can be executed.

It is therefore easy to write messy "generic" makefiles that nobody can really follow and understand. Those messy makefiles are the main reason why "make" is a bit magic. I recommend to write small and easy to read makefile.

A real example

```
MCU=atmega8
CFLAGS=-g -mmcu=$(MCU) -Wall -Wstrict-prototypes -Os -mcall-prologues
#
halloweenled.hex : halloweenled.elf
        avr-size halloweenled.elf
        avr-objcopy -R .eeprom -O ihex halloweenled.elf halloweenled.hex

halloweenled.elf : halloweenled.o
        avr-gcc $(CFLAGS) -o halloweenled.elf -Wl,-Map,halloweenled.map halloweenled.o

halloweenled.o : halloweenled.c
        avr-gcc $(CFLAGS) -Os -c halloweenled.c
#
```

```
load: halloweenled.hex
        avrdude -p m8 -c stk500v2 -e -U flash:w:halloweenled.hex
```

In this very basic example halloweenled.hex is made from halloweenled.elf which in turn is made from halloweenled.o and that is made from halloweenled.c.

To get this make-target to execute one has to type
make halloweenled.hex
Since it is the top most make-target in the file one can also just type
make
without any arguments.

Make keeps track of which files need to be re-done. In this case it is more or less obvious but we can test this by running *make halloweenled.hex* after we have already compiled the software and not changed anything. The answer will be:

```
make halloweenled.hex
make: `halloweenled.hex' is up to date.
```

In other words make knows that there is nothing to be done.

There is a second make target called "load". This target does not generate any file called "load". It is barely a name for the task. Since there will never be a file generated called "load" you can run
make load
as often as you like without getting a message about "load being up to date". The purpose of this *make load* is to load the software into the microcontroller chip. You may program a whole set of chips one after the other and every time you just type *make load*.

An interesting aspect of even this simple makefile which compiles really just a single file is that you can run *make load* without running first *make halloweenled.hex*. The make target "load" has as a prerequisite the file halloweenled.hex. In other words if halloweenled.hex is not up to date then make will generate it as per the rules for halloweenled.hex.

The command
make
compiles halloweenled.hex.

make load
Compiles halloweenled.hex and runs avrdude to load the code into the microcontroller. If halloweenled.hex was already compiled before then it will be just loaded without compiling again. If there is an error during compilation then it will stop there and avrdude will not be called.

"make" is very well designed for the task of compiling and loading software.

Help Bill help

People with a Mac or a Linux PC are usually quite happy to use the command line because it works really well. You write a bit of code and compile it. You add more and you compile and load it. After a while one has to just press arrow up to repeat the previous commands. Very easy.

To do the same thing in windows is unfortunately not as easy. One feels totally handicapped on the command line. Fortunately there is the possibility to write a batch file and set the environment such that make and all the commands that the makefile uses will run. You can then just save this batch file in the same directory as the makefile. To run it you open the file manger and double click on the batch file.

If you have an IDE for C or C++ then you can just configure it to execute this batch file at compilation time.

```
REM *** you need to edit this file and adapt it to your WinAVR
REM *** installation. E.g replace c:\avrgcc by c:\WinAVR-20090313
@echo -------- begin winmake.bat --------
set AVR=c:\avrgcc
set CC=avr-gcc
set PATH=c:\avrgcc\bin;c:\avrgcc\utils\bin
make -f Makefile
@echo --------  end  --------
pause
```

A similar batch file can be used for loading of the software:

```
@echo -------- begin winload.bat --------
set AVR=c:\avrgcc
set CC=avr-gcc
set PATH=c:\avrgcc\bin;c:\avrgcc\utils\bin
make -f Makefile load
@echo --------  end  --------
pause
```

Adapting existing Makefiles

Compiling the software works usually as one has to just have avr-gcc and the make command in the search path (unix: export PATH=${PATH}:/where/it/is and Windows: set PATH=c:\where\it\is; c:\other\location). The loading of the software is a bit more complicated.

1. There are different types of programmers (different hardware + different software)
2. Even if the same programmer hardware (avrusb500) and the same programmer software (avrdude) is used there are still differences dependent on the operating system used.

You will therefore generally have to change the part of the makefile that is responsible for the loading of the software. Here are a few examples:

```
Linux:
avrdude -P /dev/ttyUSB0 -p m168 -c stk500v2 -e -U flash:w:main.hex

Mac:
avrdude -P /dev/tty.usbserial-A9006MOb -p m168 -c stk500v2 -e -U flash:w:main.hex

Windows:
avrdude -P COM5 -p m168 -c stk500v2 -e -U flash:w:main.hex
```

In all three cases the same programmer hardware, a stk500v2 compatible programmer such as the avrusb500, is used. We program an atmega168 chip (option -p m168). The file that we want to load is main.hex.

The most obvious difference between the operating systems is the name of the serial device and you will normally have to adapt it. Under windows it might be COM3, COM4, COM5, ... it depends on how many other devices you already had. Macs name it /dev/tty.usbserial-.... followed by some random name. Linux is the most predictable. The number ttyUSB0 or ttyUSB1 depends on which device was inserted first and how many there are. Normally it's ttyUSB0.

It is also possible to configure the serial device in a system wide configuration file called avrdude.conf. If it is specified there then you will not need the -P option. If you specify -P option then you overwrite whatever is configured in avrdude.conf. In other words if you have a "good avrdude.conf" then you can delete the "-P ..." part from the Makefile.

Inserted into a Makefile the section for loading software will then look like this:

```
load: main.hex
    avrdude -P /dev/ttyUSB0 -p m168 -c stk500v2 -e -U flash:w:main.hex
```

To execute this part you would run
make load

We will make it!

References

- The tuxgraphics online shop: shop.tuxgraphics.org

© Guido Socher, tuxgraphics.org

2009-12-30, generated by tuxgrparser version 2.57