Program Development Tools

GNU `make` *(much of this material is adapted from "GNU Make" by Richard Stallman)*

The `make` utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.

Make can be used with any programming language whose compiler can be run with a shell command.

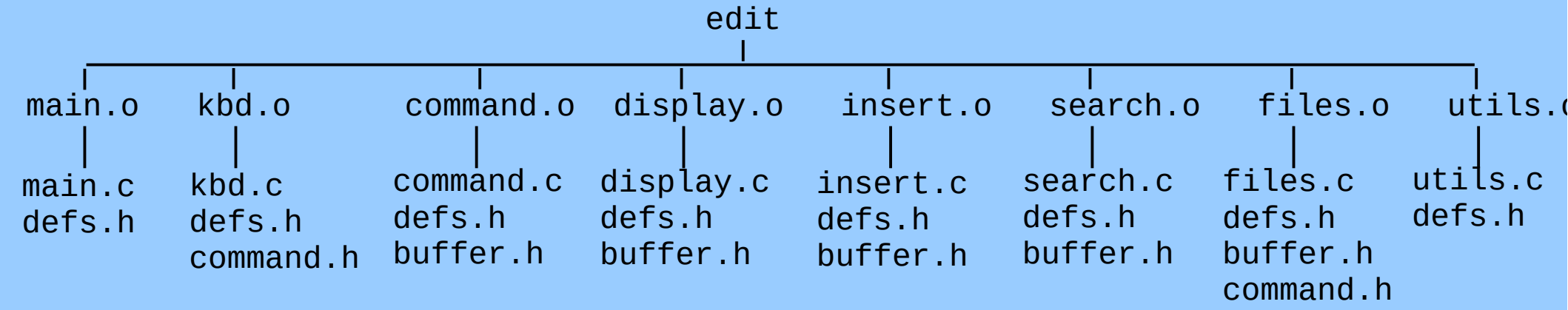
Make is not limited to programs. It can be used to describe any task where some files must be updated automatically from others whenever the others change.

We will use `make` to automatically generate executable images (*.hex files) for downloading from our *.c and *.h files.

Program Development Tools

GNU make *(much of this material is adapted from "GNU Make" by Richard Stallman)*

`Make` works by building a source file dependency graph and checking file creation times to determine which files are now out of date and recompiling only as needed.



For example, if `command.h` was changed, its timestamp would be newer than the `*.c` and `*.h` files that depend on it.

Thus `make` would recompile `kbd.c` and `files.c` and relink all the object files to produce `edit`. No other files would be compiled as there is no need.

Program Development Tools

GNU make)

`Make` is controlled by a "makefile" which describes dependencies and actions to take to rebuild the executable image.

The makefile is usually called either `Makefile` or `makefile`.

Using make and a makefile:

- -documents exactly how to rebuild the executable
- -reduces rebuilding a complex program to a single command
- -can build other files you need: .eeprom, .lst, .hex
- -is very general purpose: VHDL, Verilog, LaTeX, etc.
- -one makefile can be used reused for other projects with minor edits

Program Development Tools

Makefiles

make looks for a makefile in the current directory. The makefile describes the dependencies and actions using makefile "rules"..

A *rule* is formed as follows:

```
target:     prerequisites
 <tab>      commands
```

← a "rule"

`target:` usually the name of a file that is generated by the command below. It can also be the name of an action to carry out (a phony target).

`prerequisites:` file(s) that that are used as input to create the target. A target usually depends on several files.

`commands:` usually shell command(s) used to create the target. Can also specify commands for a target that does not depend on any prerequisites. (e.g.; clean)

Program Development Tools

Makefiles

An example rule:

```
sr.o : sr.c
	avr-gcc -g -c -Wall -O2 -mmcu=atmega128 -o sr.o sr.c
```

This rule tells make that:

- `sr.o` is dependent on `sr.c,` and...
- if `sr.c` has a more recent timestamp than `sr.o`
  `sr.o` must be recreated by executing the avr-gcc command

Program Development Tools

Makefiles

A rule, then, explains how and when to remake certain files which are the targets of the particular rule. `make` carries out the commands on the prerequisites to create or update the target.

`make` does its work in two phases:

1. include necessary files, initialize variables, rules, and build dependency graph
2. determine which targets to rebuild, and invoke the rules to do so

The order of the rules is insignificant, except for determining the default goal.

Rules tell `make` when targets are out of date and how to update them if necessary.

Targets are out of date if it does not exist or if its older than any of its prerequisities.

Program Development Tools

A simple makefile for `edit`

```
edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o insert.o search.o \
                   files.o utils.o
main.o : main.c defs.h
        cc -c main.c
kbd.o :  kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc – utils.c
clean :
        rm edit main.o kbd.o command.o display.o insert.o search.o files.o \
         utils.o
```

line continuation

prerequisites

target

command

phony target, only has a command

Program Development Tools

How `make` processes the makefile for `edit`.....

By default, `make` starts with the first target/rule. Given the command: `make`

- `make` reads `makefile` in the current directory and begins by processing the first rule. This is a command to relink `edit`.

- Before relinking, `make` must process the files that `edit` depends on.

- Each of these `*.o` files are processed according to their rules which say.....
  -recomple `*.c` to get `*.o` if `*.c` or any `*.h` is newer than `*.o` or if the `*.o` does not exist.

- Other targets are processed if they appear as prerequisites of the goal. (they are)

- If the goal does not depend upon a target, that rule is not processed.

- After any `*.o` files are created, `edit` is (re)created in a final linking step.

Program Development Tools - make

A simple make file for `sr.c` including variables

declaring a variable called PRG with the value "sr"

dereferencing a variable

phony target and dependancies with no command

```
PRG = sr_works

all         : $(PRG).elf $(PRG).lst $(PRG).hex
$(PRG).o    : $(PRG).c
      avr-gcc -g -c -Wall -O2 -mmcu=atmega128 -o $(PRG).o $(PRG).c (Cmd #1)  (Cmd #2)
$(PRG).elf :  $(PRG).o
      avr-gcc -g -Wall -O2 -mmcu=atmega128 -Wl,-Map,$(PRG).map -o $(PRG).elf $(PRG).o
$(PRG).hex :  $(PRG).elf
      avr-objcopy -j .text -j .data -O ihex $(PRG).elf $(PRG).hex  (Cmd #3)
program     :  $(PRG).hex
      avrdude -c usbasp -p m128 -e -U  flash:w:$(PRG).hex -v  (Cmd #4)
$(PRG).lst : $(PRG).elf
      avr-objdump -h -S $(PRG).elf > $(PRG).lst  (Cmd #5)
clean       :
        rm -rf $(PRG).o $(PRG).elf $(PRG).hex $(PRG).lst $(PRG).map  (Cmd #6)
```

When executed with `make` (no arguments) we see the commands executed:

```
bash-3.2$ make
avr-gcc -g -c -Wall -O2 -mmcu=atmega128 -o sr_works.o sr_works.c (Cmd #1)     (Cmd #2)
avr-gcc -g -Wall -O2 -mmcu=atmega128 -Wl,-Map,sr_works.map -o sr_works.elf sr_works.o
avr-objdump -h -S sr_works.elf > sr_works.lst  (Cmd #5)
avr-objcopy -j .text -j .data -O ihex sr_works.elf sr_works.hex  (Cmd #3)
```

Except for the default rule, the order of rules does not matter.

Program Development Tools - `make`

We would really like to make the last makefile more general purpose. We can, with...:

Implicit rules

-`make` knows how to figure out how to make some files. Its has implicit rules for updating `*.o` files from a correspondingly named `*.c` file using the compiler. We can therefore omit the commands for making `*.o` files from the object file rules.

Pattern Rules

You can define an implicit rule by writing a pattern rule . Pattern rules contain the character "%". The target is considered a pattern for matching file names. A pattern rule:

```
%.o : %.c
```

says how to make anyfile "`filename.o`" from a file "`filename.c`". Where "`filename`" represents any file name.

Program Development Tools - `make`

Automatic variables

These are variables that are computed afresh for each rule that is executed based on the target and prerequisites of the rule.

The scope of automatic variables is thus very limited. They only have values within the command script. They cannot be used in the target or prerequisite lists.

Some of the automatic variables: *(ones we use)*

$@	The file name of the target of the rule.
$<	The name of the first prerequisite
$?	The names of all the prerequisites newer than the target
$^	The names of all the prerequisites

Program Development Tools - make

```
#variables in this makefile
PRG            = sr
OBJ            = $(PRG).o
MCU_TARGET     = atmega128
OPTIMIZE       = -O0    # options are 1, 2, 3, s
OBJCOPY        = avr-objcopy
OBJDUMP        = avr-objdump
CC             = avr-gcc


#compiler and linker/loader flags
override CFLAGS        = -g -Wall $(OPTIMIZE) -mmcu=$(MCU_TARGET)
override LDFLAGS       = -Wl,-Map,$(PRG).map
#-Wall = enables all warnings about constructions that may be questionable
#-g = produce debugging information in the operating system's native format
#-Wl,-Map,$(PRG).map = pass information to linker

#automatic variables used in this makefile:
#   $@        file name of target, i.e., left hand side of :
#   $<        name of first prerequisite

all: $(PRG).elf lst text eeprom
#############################################################
#  The dependency for 'all" is the file sr.elf and three phony targets.
#  The target "all" is a phony target not a file.
#  If no arugment is given to make, it processes this first rule.
#############################################################
```

Program Development Tools - make

```
$(PRG).elf: $(OBJ)
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^
##########################################################
#  Make knows how to make an .obj from .c (implicit rule)
#  Thus we only need to say we want a .elf from an .obj
#  $@ is make shorthand for left hand side of  ":"  (think “bulls eye”)
#  $^ is make shorthand for right hand side of ":" (all the prerequisites)
##########################################################

clean:
    rm -rf $(PRG).o $(PRG).elf $(PRG).lst $(PRG).map $(PRG).bin
    rm -rf $(PRG)_eeprom.*
##########################################################
#  The target "clean" is a phony target that cleans
#  up all the extra files we make at each compile.
##########################################################

program: $(PRG).hex
    avrdude -c usbasp -p m128 -e -U  flash:w:$(PRG).hex -v
##########################################################
#  Target "program" depends on the .hex file which it
#  downloads to the target board using the parallel port.
##########################################################
```

Program Development Tools - make

A *pattern rule*. This matches names on each side.
It allows you to make any file xxx.lst from xxx.elf.

```
lst:  $(PRG).lst
%.lst: %.elf
    $(OBJDUMP) -h -S $< > $@
##########################################################
#  The target "lst" is another phony target. Its depends
#  on the .lst (list)  file.  The list file shows the
#  assembly language output of the compiler intermixed
#  with the C code so it is easier to debug and follow.
#  avr-objdump ($OBJDUMP) is the avr binutil tool we us to
#  get information from the .elf file.
#
#  $@ is the file name of target, i.e., left hand side of :
#  $< is shorthand for source file for the single dependency
##########################################################
```

Program Development Tools - make

```
#############################################################
# Following are rules for building the .text rom images
# This is the stuff that will be put in flash.
#############################################################
text: hex bin srec  #.text section holds code and read only data

hex:  $(PRG).hex
bin:  $(PRG).bin
srec: $(PRG).srec

%.hex: %.elf
    $(OBJCOPY) -j .text -j .data -O ihex $< $@
#############################################################
#  Take any .elf file and build the .hex file from it using
#  avr-objcopy.  avr-objcopy is used to extract the downloadable
#  portion of the .elf file to build the flash image.
#############################################################

%.srec: %.elf
    $(OBJCOPY) -j .text -j .data -O srec $< $@
#Make the Motorola S-record file

%.hex: %.elf
    $(OBJCOPY) -j .text -j .data -O binary $< $@
#Make a binary imagae also
```

Program Development Tools - make

```
#############################################################
# Following are rules for building the .eeprom images
# This is the stuff that will be put in EEPROM.
#############################################################
eeprom: ehex ebin esrec

ehex:  $(PRG)_eeprom.hex
ebin:  $(PRG)_eeprom.bin
esrec: $(PRG)_eeprom.srec

%_eeprom.hex: %.elf
    $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O ihex $< $@
#############################################################
#This builds the EEPROM image from the .elf file for downloading.
#############################################################

%_eeprom.srec: %.elf
    $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O srec $< $@
#############################################################
#This builds the S-record image from the .elf file if necessary.
#############################################################

%_eeprom.bin: %.elf
    $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O binary $< $@
#############################################################
#This builds a binary image from the .elf file if necessary.
#############################################################
```

Program Development Tools - make

```
#############################################################
# For programs existing in multiple files we would make the following
# changes.  In this case we have the files:
#  sr.c, init.c, lcd.h
#

OBJS   = sr.o init.o lcd.o    #multiple .o files needed

lcd.o   :   lcd.h             #add dependency for building lcd.o


#############################################################
#If you wanted to add an assembly file called sw_spi.S to the mix...
#avr-gcc knows to get the sw_spi.o it will need to assemble sw_spi.S

OBJS   = sr.o init.o lcd.o sw_spi.o  #multiple .o files needed

lcd.o   :   lcd.h             #add dependency for building lcd.o
```

Program Development Tools - `make`

Remember, in *make* :

- -Commands are preceded with a hard tab.
- -Most make files should have:
  - `make all :` make all files necessary to do anything you want to do
  - `make clean :` clean up files you made last time (.o, .lst, .hex, etc.)
- -Without an argument, make executes the first rule.
- -Variables are referenced as: `$(variable_name)`
- -Lines may be continued with the backslash `"\"`
- -Show what make would do, but don't do it `make -n [target]`
- -Wildcard characters in names
  - `make` wildcard characters are "`*`", "`?`" and "`[...]`" *(same as Bourne shell)*
  - example: `rm -f *.o` removes all files that end in "`.o`"
- -Timestamps may be updated with the shell command `touch`. i.e.,

```
ll count.c
-rwxr-xr-x 1 bob upg1018 304 Sep 30 2007 count.c*
touch count.c
ll count.c
-rwxr-xr-x 1 bob upg1018 304 Aug 12 2008 count.c*
```

Program Development Tools - make

Automatic Prerequisite Generation for Header Files

Generally, translational units are turned into object files. But header files are not standalone translational units, make has no way of knowing that it needs to rebuild a .c file when a .h file changes.

The GNU C compiler is able to read your source file and generate a list of prerequisite files for it with the -MM switch.

We can take this output and create another makefile from it using some shell commands. There will be one makefile for each source file. Each make file is then included in our top level make file so that the dependencies are taken care of.

Here is the makefile code for this:

```
%.d: %.c
            @set -e; rm -f $@; \
             $(CC) -M $(CFLAGS) $< > $@.$$$$; \
             sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
             rm -f $@.$$$$

-include ($SRCS:.c =.d)
```