

Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required.

Take the 2-minute tour



How to generate a Makefile with source in sub-directories using just one makefile



I have source in a bunch of subdirectories like:

```
src/widgets/apple.cpp
src/widgets/knob.cpp
src/tests/blend.cpp
src/ui/flash.cpp
```

In the root of the project I want to generate a single Makefile using a rule like:

```
%.o: %.cpp
$(CC) -c $<

build/test.exe: build/widgets/apple.o build/widgets/knob.o build/tests/blend.o
src/ui/flash.o
$(LD) build/widgets/apple.o .... build/ui/flash.o -o build/test.exe
```

When I try this it does not find a rule for build/widgets/apple.o. Can I change something so that the %o: %.cpp is used when it needs to make build/widgets/apple.o ?

makefile make

asked Oct 23 '08 at 19:56

 [Jeroen Dirks](#)
2,674 7 30 54

7 Answers

The reason is that your rule

```
%o: %.cpp
...
```

expects the .cpp file to reside in the same directory as the .o your building. Since test.exe in your case depends on build/widgets/apple.o (etc), make is expecting apple.cpp to be build/widgets/apple.cpp.

You can use VPATH to resolve this:

```
VPATH = src/widgets

BUILDDIR = build/widgets

$(BUILDDIR)/%.o: %.cpp
...
```

When attempting to build "build/widgets/apple.o", make will search for apple.cpp in VPATH. Note that the build rule has to use special variables in order to access the actual filename make finds:

```
$(BUILDDIR)/%.o: %.cpp
$(CC) $< -o $@
```

Where "\$<" expands to the path where make located the first dependency.

Also note that this will build all the .o files in build/widgets. If you want to build the binaries in different directories, you can do something like

```
build/widgets/%.o: %.cpp
....

build/ui/%.o: %.cpp
....

build/tests/%.o: %.cpp
....
```

I would recommend that you use "canned command sequences" in order to avoid repeating the actual compiler build rule:


```
define cc-command
$(CC) $(CFLAGS) $< -o $@
endef
```

You can then have multiple rules like this:

```
build1/foo.o build1/bar.o: %.o: %.cpp
$(cc-command)

build2/frotz.o build2/fie.o: %.o: %.cpp
$(cc-command)
```

answered Oct 23 '08 at 20:39

 [JesperE](#)
36.8k 9 88 151

`VPATH` does not allow you to have different source files with the same name in different directories, which defeats the purpose of directories in the first place. – [Maxim Egorushkin](#) Jul 17 '13 at 12:58

2 Eh, no I don't think it does. Organizing your source code in different directories has more benefits than allowing multiple source files with the same name. – [JesperE](#) Jul 17 '13 at 15:59



This does the trick:

```
CC      := g++
LD      := g++

MODULES := widgets test ui
SRC_DIR := $(addprefix src,$(MODULES))
BUILD_DIR := $(addprefix build,$(MODULES))

SRC      := $(foreach sdir,$(SRC_DIR),$(wildcard $(sdir)/*.cpp))
OBJ      := $(patsubst src/%.cpp,build/%.o,$(SRC))
INCLUDES := $(addprefix -I,$(SRC_DIR))

vpath %.cpp $(SRC_DIR)

define make-goal
$1/%.o: %.cpp
$(CC) $(INCLUDES) -c $< -o $@
endef

.PHONY: all checkdirs clean

all: checkdirs build/test.exe

build/test.exe: $(OBJ)
$(LD) $^ -o $@

checkdirs: $(BUILD_DIR)

$(BUILD_DIR):
@mkdir -p $@

clean:
@rm -rf $(BUILD_DIR)

$(foreach bdir,$(BUILD_DIR),$(eval $(call make-goal,$(bdir))))
```

This Makefile assumes you have your include files in the source directories. Also it checks if the build directories exist, and creates them if they do not exist.

The last line is the most important. It creates the implicit rules for each build using the function `make-goal`, and it is not necessary write them one by one

You can also add automatic dependency generation, using [Tromey's way](#)

edited Mar 20 '10 at 19:14

answered Mar 20 '10 at 19:08

 [Manzill0](#)
381 3 3

Very good, exactly what I was looking for ;) Thank you Manzill0 – [Geoffroy](#) May 29 '11 at 11:47

Your use of foreach really helped me. I create a long list of C files that I generate an individual object file. Thanks! – [Eric Cope](#) Sep 21 '11 at 6:59

Thing is `$@` will include the entire (relative) path to the source file which is in turn used to construct the object name (and thus its relative path)

We use:

```
#####
# rules to build the object files
$(OBJDIR_1)/%.o: %.c
    @$(ECHO) "$< -> $@"
    @test -d $(OBJDIR_1) || mkdir -pm 775 $(OBJDIR_1)
    @test -d $(@D) || mkdir -pm 775 $(@D)
    @$(RM) $@
    $(CC) $(CFLAGS) $(CFLAGS_1) $(ALL_FLAGS) $(ALL_DEFINES) $(ALL_INCLUDEDIRS:%=-I%) -c
    $< -o $@
```

This creates an object directory with name specified in `$(OBJDIR_1)` and subdirectories according to subdirectories in source.

For example (assume objs as toplevel object directory), in Makefile:

```
widget/apple.cpp
tests/blend.cpp
```

results in following object directory:

```
objs/widget/apple.o
objs/tests/blend.o
```

edited Sep 15 '09 at 23:49



sth

101k 20 168 278

answered Sep 15 '09 at 13:33

Tim Ruijs

This is another trick.

In main 'Makefile' define SRCDIR for each source dir and include 'makef.mk' for each value of SRCDIR. In each source dir put file 'files.mk' with list of source files and compile options for some of them. In main 'Makefile' one can define compile options and exclude files for each value of SRCDIR.

Makefile:

```
PRG          := prog-name

OPTIMIZE     := -O2 -fomit-frame-pointer

CFLAGS += -finline-functions-called-once
LDFLAGS += -Wl,--gc-section,--reduce-memory-overheads,--relax

.DEFAULT_GOAL := hex

OBJDIR       := obj

MK_DIRS      := $(OBJDIR)

SRCDIR       := .
include      makef.mk

SRCDIR := crc
CFLAGS_crc := -DCRC8_BY_TABLE -DMODBUS_CRC_BY_TABLE
ASFLAGS_crc := -DCRC8_BY_TABLE -DMODBUS_CRC_BY_TABLE
include makef.mk

#####

CC          := avr-gcc -mmcu=$(MCU_TARGET) -I.
OBJCOPY     := avr-objcopy
OBJDUMP     := avr-objdump

C_FLAGS     := $(CFLAGS) $(REGS) $(OPTIMIZE)
CPP_FLAGS   := $(CPPFLAGS) $(REGS) $(OPTIMIZE)
AS_FLAGS    := $(ASFLAGS)
LD_FLAGS    := $(LDFLAGS) -Wl,-Map,$(OBJDIR)/$(PRG).map

C_OBJS      := $(C_SRC:%.c=$(OBJDIR)/%.o)
CPP_OBJS    := $(CPP_SRC:%.cpp=$(OBJDIR)/%.o)
AS_OBJS     := $(AS_SRC:%.S=$(OBJDIR)/%.o)

C_DEPS      := $(C_OBJS:%=%.d)
CPP_DEPS    := $(CPP_OBJS:%=%.d)
AS_DEPS     := $(AS_OBJS:%=%.d)

OBJS        := $(C_OBJS) $(CPP_OBJS) $(AS_OBJS)
DEPS        := $(C_DEPS) $(CPP_DEPS) $(AS_DEPS)

hex: $(PRG).hex
lst: $(PRG).lst

$(OBJDIR)/$(PRG).elf: $(OBJS)
    $(CC) $(C_FLAGS) $(LD_FLAGS) $^ -o $@
```

```

%.lst: $(OBJDIR)/%.elf
    -@rm $@ 2> /dev/nul
    $(OBJDUMP) -h -s -S $< > $@

%.hex: $(OBJDIR)/%.elf
    -@rm $@ 2> /dev/nul
    $(OBJCOPY) -j .text -j .data -O ihex $< $@

$(C_OBJJS) : $(OBJDIR)/%.o : %.c Makefile
    $(CC) -MMD -MF $@.p.d -c $(C_FLAGS) $(C_FLAGS_$(call clear_name,$<)) $< -o $@
    @sed -e 's,.*,SRC_FILES += ,g' < $@.p.d > $@.d
    @sed -e "\$\$/\$/ $(subst /,/,$(dir $<))files.mk\n/" < $@.p.d >> $@.d
    @sed -e 's,^[^:]*: *,,' -e 's,[ \t]*,,,' -e 's, \\$$,,,' -e 's,$$, :, ' < $@.p.d >> $@.d
    -@rm -f $@.p.d

$(CPP_OBJJS) : $(OBJDIR)/%.o : %.cpp Makefile
    $(CC) -MMD -MF $@.p.d -c $(CPP_FLAGS) $(CPP_FLAGS_$(call clear_name,$<)) $< -o $@
    @sed -e 's,.*,SRC_FILES += ,g' < $@.p.d > $@.d
    @sed -e "\$\$/\$/ $(subst /,/,$(dir $<))files.mk\n/" < $@.p.d >> $@.d
    @sed -e 's,^[^:]*: *,,' -e 's,[ \t]*,,,' -e 's, \\$$,,,' -e 's,$$, :, ' < $@.p.d >> $@.d
    -@rm -f $@.p.d

$(AS_OBJJS) : $(OBJDIR)/%.o : %.S Makefile
    $(CC) -MMD -MF $@.p.d -c $(AS_FLAGS) $(AS_FLAGS_$(call clear_name,$<)) $< -o $@
    @sed -e 's,.*,SRC_FILES += ,g' < $@.p.d > $@.d
    @sed -e "\$\$/\$/ $(subst /,/,$(dir $<))files.mk\n/" < $@.p.d >> $@.d
    @sed -e 's,^[^:]*: *,,' -e 's,[ \t]*,,,' -e 's, \\$$,,,' -e 's,$$, :, ' < $@.p.d >> $@.d
    -@rm -f $@.p.d

clean:
    -@rm -rf $(OBJDIR)/$(PRG).elf
    -@rm -rf $(PRG).lst $(OBJDIR)/$(PRG).map
    -@rm -rf $(PRG).hex $(PRG).bin $(PRG).srec
    -@rm -rf $(PRG)_eeprom.hex $(PRG)_eeprom.bin $(PRG)_eeprom.srec
    -@rm -rf $(MK_DIRS:%=/*.*.o) $(MK_DIRS:%=/*.*.d)
    -@rm -f tags cscope.out

# -rm -rf $(OBJDIR)/*
# -rm -rf $(OBJDIR)
# -rm $(PRG)

tag: tags
tags: $(SRC_FILES)
    if [ -e tags ] ; then ctags -u $? ; else ctags $^ ; fi
    cscope -U -b $^

# include dep. files
ifneq "$(MAKECMDGOALS)" "clean"
    -include $(DEPS)
endif

# Create directory
$(shell mkdir $(MK_DIRS) 2>/dev/null)

```

makef.mk

```

SAVE_C_SRC := $(C_SRC)
SAVE_CPP_SRC := $(CPP_SRC)
SAVE_AS_SRC := $(AS_SRC)

C_SRC :=
CPP_SRC :=
AS_SRC :=

include $(SRCDIR)/files.mk
MK_DIRS += $(OBJDIR)/$(SRCDIR)

clear_name = $(subst /,_,$(1))

define rename_var
$(2)_$(call clear_name,$(SRCDIR))_$(call clear_name,$(1)) := \
    $($$(subst _,,$(2))_$(call clear_name,$(SRCDIR))) $($$(call clear_name,$(1)))
$(call clear_name,$(1)) :=
endef

define proc_lang

ORIGIN_SRC_FILES := $($$(1)_SRC)

ifneq ($(strip $($$(1)_ONLY_FILES)),)
    $(1)_SRC := $(filter $($$(1)_ONLY_FILES),$(1)_SRC)
else

ifneq ($(strip $(ONLY_FILES)),)
    $(1)_SRC := $(filter $(ONLY_FILES),$(1)_SRC)
else
    $(1)_SRC := $(filter-out $(EXCLUDE_FILES),$(1)_SRC)
endif

```

```

endif

$(1)_ONLY_FILES :=
$(foreach name,$($(1)_SRC),$(eval $(call rename_var,$(name),$(1)_FLAGS)))
$(foreach name,$(ORIGIN_SRC_FILES),$(eval $(call clear_name,$(name)) :=))

endif

$(foreach lang,C CPP AS, $(eval $(call proc_lang,$(lang))))

EXCLUDE_FILES :=
ONLY_FILES :=

SAVE_C_SRC += $(C_SRC:%=$(SRCDIR)/%)
SAVE_CPP_SRC += $(CPP_SRC:%=$(SRCDIR)/%)
SAVE_AS_SRC += $(AS_SRC:%=$(SRCDIR)/%)

C_SRC := $(SAVE_C_SRC)
CPP_SRC := $(SAVE_CPP_SRC)
AS_SRC := $(SAVE_AS_SRC)

```

./files.mk

```

C_SRC := main.c
CPP_SRC :=
AS_SRC := timer.S

main.c += -DDEBUG

```

./crc/files.mk

```

C_SRC := byte-modbus-crc.c byte-crc8.c
AS_SRC := modbus-crc.S crc8.S modbus-crc-table.S crc8-table.S

byte-modbus-crc.c += --std=gnu99
byte-crc8.c += --std=gnu99

```

edited Jul 23 '13 at 1:30

answered Jul 22 '13 at 8:41



31 3

Usually, you create a Makefile in each subdirectory, and write in the top-level Makefile to call make in the subdirectories.

This page may help: <http://www.gnu.org/software/make/>

answered Oct 23 '08 at 20:02



667 4 10

This is commonly done, but is full of problems. The main one is that no one make process knows about all of the dependencies, so things like -j2 on multicore systems won't work. See aegis.sourceforge.net/auug97.pdf – KeithB Oct 23 '08 at 21:11

- 1 Keith's reference is to an excellent paper called 'Recursive Make Considered Harmful'. This is a contribution to the series of articles starting with Dijkstra's 'Go To Considered Harmful' letter, and culminating in '"Considered Harmful' Considered Harmful". – Jonathan Leffler Oct 23 '08 at 21:28
- 2 It's done usually because people don't understand how to write Makefiles. One Makefile per directory sucks. – mxcl Oct 7 '09 at 17:29

Here is my solution, inspired from Beta's answer. It's simpler than the other proposed solutions

I have a project with several C files, stored in many subdirectories. For example:

```

src/lib.c
src/aa/a1.c
src/aa/a2.c
src/bb/b1.c
src/cc/c1.c

```

Here is my Makefile (in the `src/` directory):

```

# make      -> compile the shared library "libfoo.so"
# make clean -> remove the library file and all object files (.o)
# make all   -> clean and compile
SONAME = libfoo.so
SRC = lib.c \
      aa/a1.c \
      aa/a2.c \
      bb/b1.c \
      cc/c1.c
# compilation options
CFLAGS = -O2 -g -W -Wall -Wno-unused-parameter -Wbad-function-cast -fPIC
# linking options

```

```
LDLFLAGS = -shared -Wl,-soname,$(SONAME)

# how to compile individual object files
OBJS = $(SRC:.c=.o)
.c.o:
    $(CC) $(CFLAGS) -c $< -o $@

.PHONY: all clean

# library compilation
$(SONAME): $(OBJS) $(SRC)
    $(CC) $(OBJS) $(LDLFLAGS) -o $(SONAME)

# cleaning rule
clean:
    rm -f $(OBJS) $(SONAME) *~

# additional rule
all: clean lib
```

This example works fine for a shared library, and it should be very easy to adapt for any compilation process.

answered Jan 25 '14 at 19:23



Amaury Bouchard

11 2

This will do it without painful manipulation or multiple command sequences:

```
build/%.o: src/%.cpp
src/%.o: src/%.cpp
%.o:
    $(CC) -c $< -o $@

build/test.exe: build/widgets/apple.o build/widgets/knob.o build/tests/blend.o
src/ui/flash.o
    $(LD) $^ -o $@
```

JasperE has explained why "%.o: %.cpp" won't work; this version has one pattern rule (%.o:) with commands and no prereqs, and two pattern rules (build/%.o: and src/%.o:) with prereqs and no commands. (Note that I put in the src/%.o rule to deal with src/ui/flash.o, assuming that wasn't a typo for build/ui/flash.o, so if you don't need it you can leave it out.)

build/test.exe needs build/widgets/apple.o,
build/widgets/apple.o looks like build/%.o, so it needs src/%.cpp (in this case src/widgets/apple.cpp),
build/widgets/apple.o also looks like %.o, so it executes the CC command and uses the prereqs it just found (namely src/widgets/apple.cpp) to build the target (build/widgets/apple.o)

answered Sep 16 '09 at 0:43



Beta

41.9k 4 46 78

This breaks, because your %.o: rule does not have prerequisites, yet refers to them via \$< (which is therefore empty). "No input files", sorry. — [DevSolar](#) Nov 24 '11 at 10:32