# Teensy 3.5 Makefile (2019/03/16)

To compile code for the Teensy from the command line you could use the Makefile that comes with the Teensyduino install.

Just navigate to the "... /hardware/teensy/avr/cores/teensy3" directory, create a main.cpp file and the Makefile will compile it.

Cool, but I prefer to keep my software projects in my $HOME/src directory. This took me on the long winding road of trying to better understand the Teensy makefile so I could customize it.

Let the adventure begin....
"YOU ARE IN A MAZE OF TWISTING LITTLE PASSAGES"

My goal is to have a project directory with the following structure:

```
~/src/proj/main.c                <--- project source code
          /Makefile              <--- project make file
          /teensy3/Makefile.core <--- core makefile
          /teensy3/*.S,c,cpp,h   <--- teensy core software
          /teensy3/build/*.o     <--- teensy build directory
          /lib/core.a            <--- compiled teensy library
          /include/*.h           <--- teensy include files
```

The project Makefile will recursively call the Makefile.core if any of the compile options change and we need to rebuild the core library. So far I have only need this to switch from using the USB Serial Port to USB Keyboard.

The makefile.core build process will compile the Teensy source into the build directory. The Teensy object files are archived into core.a in the lib directory. The Teensy header files are copied into the include directory.

Before we get to this solution we have a long twisty road ahead.

## Arduino/Teensyduino Install

There are a number of ways you can install and configure the Arduino and Teensyduino software. My approach is to install the Arduino software in $HOME/opt and then install Teensyduino on top of that directory.

```
---> download Arduino software, currently version 1.8.8
$ tar -xf arduino-1.8.8-linux64.tar.xz
$ cp -R arduino-1.8.8 ~/opt
---> download Teensyduino, currently version 1.4.5
$ mv TeensyduinoInstall.linux64 TeensyduinoInstall.linux64_1.4.5
$ chmod 755 TeensyduinoInstall.linux64_1.4.5
```

```
$ ./TeensyduinoInstall.linux64_1.4.5
---> navigate to $HOME/opt/arduino-1.8.8 and install
```

I start by making a copy of the teensy3 code into my local project directory. I like to keep a local copy with the project in case I delete an old arduino version.

```
$ cp -R ~/opt/arduino-1.8.8/hardware/teensy/avr/cores/teensy3 \
            ~/src/myproject/teensy3
```

## Core Makefile Changes

I initally started by making a few changes to the Teensy Makefile. This would work well if I was just going to keep my programs mixed in with the Teensy core software. Seems kind of messy to me.

However, here was my first pass through the Makefile to make it Teensy 1.45 compatable.

```
$ cd teensy3
$ cp Makefile Makefile.1    <--- always nice to make a backup
$ vi Makefile

+edits to the teensy3/Makefile
 -- set microcontroller type to MK64FX512 for teensy 3.5

    # set your MCU type here, or make command line `make MCU=MK20DX256`
    #MCU=MK20DX256
    #MCU=MKL26Z64
    MCU=MK64FX512       <-------- uncomment this for teensy 3.5
    #MCU=MK66FX1M0

 -- set ARDUINOPATH to point to arduino install directory.
 -- This means we will use the compiler tool chain from the Arduino install
 -- and not from the base Linux system
 -- note ?= set the variable if it is not already set, in this case we
 -- want to set it explicitly

    # Those that specify a NO_ARDUINO environment variable will
    # be able to use this Makefile with no Arduino dependency.
    # Please note that if ARDUINOPATH was set, it will override
    # the NO_ARDUINO behaviour.
    ifndef NO_ARDUINO
    # Path to your arduino installation
    # ARDUINOPATH ?= ../../../../..
    endif
```

```
    ARDUINOPATH = /home/john/opt/arduino-1.8.8  <---- arduino install directory

 -- add option of being Serial or Keyboard device
    # option for USB serial for keyboard
    USBFLAG = USB_SERIAL
    ifeq ($(USB),keyboard)
        USBFLAG=USB_KEYBOARD
    endif


 -- change options to match current versions of arduino and teensyduino
    # configurable options
 -->    OPTIONS = -DF_CPU=120000000 -D$(USBFLAG) -DLAYOUT_US_ENGLISH -DUSING_MAKEFILE
    # options needed by many Arduino libraries to configure for Teensy 3.x
 -->   OPTIONS += -D__$(MCU)__ -DARDUINO=10808 -DTEENSYDUINO=145

 -- comment out the upload from this makefile in the elf to hex target
    %.hex: %.elf
        $(SIZE) $<
        $(OBJCOPY) -O ihex -R .eeprom $< $@
-->    # ifneq (,$(wildcard $(TOOLSPATH)))
-->    #    $(TOOLSPATH)/teensy_post_compile -file=$(basename $@) \
                -path=$(shell pwd) -tools=$(TOOLSPATH)
-->    #    -$(TOOLSPATH)/teensy_reboot
-->    # endif
```

You should be able to try this out by typing "make". It will compile a little cpp program (main.cpp) that is conveniently provided. Notice the "-DUSING_MAKEFILE" option. The cpp program can use this to conditionally compile for the IDE or standalone environments.

## Project Makefile

I turned on verbose output with the Arduino IDE and compared that to the Teenyduino Makefile. This is where life got a little confusing. There are a lot of different flags used during the compile. Time to investigate.

In the ~/opt/arduino-1.8.8/hardware/teensy/avr directory the boards.txt has options for the build process listed.

```
teensy35.build.board=TEENSY35
teensy35.build.core=teensy3
teensy35.build.mcu=mk64fx512
teensy35.build.warn_data_percentage=98
teensy35.build.toolchain=arm/bin/
```

```
teensy35.build.command.gcc=arm-none-eabi-gcc
teensy35.build.command.g++=arm-none-eabi-g++
teensy35.build.command.ar=arm-none-eabi-gcc-ar
teensy35.build.command.objcopy=arm-none-eabi-objcopy
teensy35.build.command.objdump=arm-none-eabi-objdump
teensy35.build.command.size=arm-none-eabi-size
teensy35.build.flags.common=-g -Wall -ffunction-sections -fdata-sections -nostdlib
teensy35.build.flags.dep=-MMD
teensy35.build.flags.optimize=-Os
teensy35.build.flags.cpu=-mthumb -mcpu=cortex-m4 -mfloat-abi=hard \
                         -mfpu=fpv4-sp-d16 -fsingle-precision-constant
teensy35.build.flags.defs=-D__MK64FX512__ -DTEENSYDUINO=145
teensy35.build.flags.cpp=-fno-exceptions -felide-constructors -std=gnu++14 \
                         -Wno-error=narrowing -fno-rtti
teensy35.build.flags.c=
teensy35.build.flags.S=-x assembler-with-cpp
teensy35.build.flags.ld=-Wl,--gc-sections,--relax, \
     --defsym=__rtc_localtime={extra.time.local} "-T{build.core.path}/mk64fx512.ld" \
     -lstdc++
teensy35.build.flags.libs=-larm_cortexM4lf_math -lm
```

This look promising. The verbose output from the Arduino IDE also provides some clues to the flags and options we need to compile code.

BTW: To turn on verbose:

```
Arduino IDE --> File --> Prefenences
    --> Show verbose output during: +Compilation +Upload
```

All of the flag/options are in the verbose output and a few extra options that can be set via the Arduino menus. (arduino version, speed, usb type, keyboard).

```
Assembler Flags:
-c -O2 -g -Wall -ffunction-sections -fdata-sections -nostdlib
-MMD -x assembler-with-cpp -mthumb -mcpu=cortex-m4 -mfloat-abi=hard
-mfpu=fpv4-sp-d16 -fsingle-precision-constant -D__MK64FX512__ -DTEENSYDUINO=145
-DARDUINO=10808 -DF_CPU=120000000 -DUSB_SERIAL -DLAYOUT_US_ENGLISH
-I/home/john/opt/arduino-1.8.8/hardware/teensy/avr/cores/teensy3

C Program Flags:
-c -O2 -g -Wall -ffunction-sections -fdata-sections -nostdlib
-MMD -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16
-fsingle-precision-constant -D__MK64FX512__ -DTEENSYDUINO=145 -DARDUINO=10808
-DF_CPU=120000000 -DUSB_SERIAL -DLAYOUT_US_ENGLISH
-I/home/john/opt/arduino-1.8.8/hardware/teensy/avr/cores/teensy3
```

```
CPP Program Flags:
-c -O2 -g -Wall -ffunction-sections -fdata-sections
-nostdlib -MMD -fno-exceptions -felide-constructors -std=gnu++14
-Wno-error=narrowing -fno-rtti -mthumb -mcpu=cortex-m4 -mfloat-abi=hard
-mfpu=fpv4-sp-d16 -fsingle-precision-constant -D__MK64FX512__ -DTEENSYDUINO=145
-DARDUINO=10808 -DF_CPU=120000000 -DUSB_SERIAL -DLAYOUT_US_ENGLISH
-I/tmp/arduino_build_259100/pch
-I/home/john/opt/arduino-1.8.8/hardware/teensy/avr/cores/teensy3

Linking Flags:
-O2 -Wl,--gc-sections,--relax,--defsym=__rtc_localtime=1552744299
-T/home/john/opt/arduino-1.8.8/hardware/teensy/avr/cores/teensy3/mk64fx512.ld
-lstdc++ -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16
-fsingle-precision-constant

Linking Libs -L/tmp/arduino_build_259100 -larm_cortexM4lf_math -lm
```

The Teensyduio Makefile has a few less flags and options:

```
# CPPFLAGS = compiler options for C and C++
CPPFLAGS = -Wall -g -Os -mcpu=$(CPUARCH) -mthumb -MMD $(OPTIONS) -I.

# compiler options for C++ only
CXXFLAGS = -std=gnu++14 -felide-constructors -fno-exceptions -fno-rtti

# compiler options for C only
CFLAGS =

# linker options
LDFLAGS = -Os -Wl,--gc-sections,--defsym=__rtc_localtime=0 --specs=nano.specs
          -mcpu=$(CPUARCH) -mthumb -T$(MCU_LD)

# additional libraries to link
LIBS = -lm
```

So, what to do. Go with the Teensyduino flags/options or copy the Arduino verbose output flags/options.

## Project Makefile Flags & Options

I finally decided to follow the verbose output from the Arduino IDE.

```
# BUILD OPTIONS & FLAGS ============================================
#
# configurable USB serial or keyboard option
# .... serial port is the default
# .... "make keyboard" will compile core software for USB keyboard
USBFLAG = USB_SERIAL
ifeq ($(USB),keyboard)
   USBFLAG=USB_KEYBOARDONLY
endif

# board specific settings -- Teensy 3.5
MCU=MK64FX512
MCU_LD = mk64fx512.ld
CPUARCH = cortex-m4

# options configured thru Arduino menus
OPTIONS = -DF_CPU=120000000 -D$(USBFLAG) -DLAYOUT_US_ENGLISH
# options needed by many Arduino libraries to configure for Teensy 3.x
OPTIONS += -D__$(MCU)__  -DTEENSYDUINO=145 -DARDUINO=10808
# This can be used in cpp/c programs to conditionally compile for ino
# or regular cpp/c programs
OPTIONs += -DUSING_MAKEFILE

# CPP CXX C & AS flags plus options
CPPFLAGS = -c -O2 -g -Wall -ffunction-sections -fdata-sections -nostdlib \
         -MMD -mthumb -mcpu=$(CPUARCH) -mfloat-abi=hard -mfpu=fpv4-sp-d16 \
         -fsingle-precision-constant $(OPTIONS)
CXXFLAGS = -fno-exceptions -felide-constructors -std=gnu++14 -fno-rtti
CFLAGS =
ASFLAGS = -x assembler-with-cpp

# linker options
TS = $(shell date +%s)
LDFLAGS = -O2 -Wl,--gc-sections,--relax,--defsym=__rtc_localtime=$(TS) \
          -T$(COREINC)/$(MCU_LD) -lstdc++  -mthumb -mcpu=$(CPUARCH) \
          -mfloat-abi=hard -mfpu=fpv4-sp-d16 -fsingle-precision-constant

LIBS = -larm_cortexM4l_math -lm

export CPPFLAGS CXXFLAGS CFLAGS ASFLAGS LDFLAGS LIBS MCU_LD
```

Notice how these variables are exported so they are available when the project makefile calls the Teensy core makefile. The project makefile has a target that will call the Teensy core makefile, first to delete the existing core libray then rebuild it.

```
core:
    cd $(COREPATH) &&  make -f Makefile.core clean
    cd $(COREPATH) &&  make -f Makefile.core
```

## Review of the Flags

While reviewing all the flags and options in the Arduino IDE build process and the Teensyduino makefile I collected information on each of them.

Bit of a dry read, but quite enlighten.

### Common Flags (CPPFLAGS)

-Wall
enables all compilers warning messages

-c
Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

-D name=definition
The contents of definition are tokenized and processed as if they appeared during translation phase three in a '#define' directive. In particular, the definition is truncated by embedded newline characters.

-O2
GCC performs nearly all supported optimizations that do not involve a space-speed trade-off. The compiler does not perform loop unrolling or function inlining when you specify -O2. As compared to -O, this option increases both compilation time and the performance of the generated code.

-Os
Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size. -Os disables the following optimization flags: -falign-functions -falign-jumps -falign-loops -falign-labels -freorder-blocks -freorder-blocks-and-partition -fprefetch-loop-arrays -ftree-vect-loop-version

-g
Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.

-ffunction-sections
By default every functions are combined into single .text function. With this flag the compiler creates different text section for each function. Generates a separate ELF section for each function in the source file. The unused section elimination feature of the linker can then remove unused functions at link time.

-fdata-sections, -fno-data-sections
By default variables belongs .data are combined into single .data function. With this flag the compiler creates different .data section for each.

The unused section elimination feature of the linker can then remove unused functions at link time.

-nostdlib
Do not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify are passed to the linker, and options specifying linkage of the system libraries, such as -static-libgcc or -shared-libgcc, are ignored. The compiler may generate calls to memcmp, memset, memcpy and memmove. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified. -nostdlib is disabled by default.

-MMD
Generate a dependency output file. Like -MD except mention only user header files, not system header files. -MD.. Used to generate a dependency output file (with a suffix of .d) as a side effect of the compilation process -M.. output a rule suitable for make describing the dependencies of the main source file. The preprocessor outputs one make rule containing the object file name for that source file, a colon, and the names of all the included files, including those coming from -include or -imacros command-line options.

-mthumb
-marm
Select between generating code that executes in ARM and Thumb states. The default for most configurations is to generate code that executes in ARM state, but the default can be changed by configuring GCC with the –with-mode=state configure option.

-mcpu=name[+extension...]
This specifies the name of the target ARM processor. GCC uses this name to derive the name of the target ARM architecture (as if specified by -march) and the ARM processor type for which to tune for performance (as if specified by -mtune). Where this option is used in conjunction with -march or -mtune, those options take precedence over the appropriate part of this option.

-mfloat-abi=hard
Specifies which floating-point ABI to use. Permissible values are: ‘soft’, ‘softfp’ and ‘hard’. Specifying ‘soft’ causes GCC to generate output containing library calls for floating-point operations. ‘softfp’ allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions. ‘hard’ allows generation of floating-point instructions and uses FPU-specific calling conventions.

-mfpu=fpv4-sp-d16
This specifies what floating-point hardware is available on the target.

-fsingle-precision-constant
Causes floating-point constants to be loaded in single precision even when this is not exact. This avoids promoting operations on single precision variables to

double precision like in x + 1.0/3.0. Note that this also uses single precision constants in operations on double precision variables. This can improve performance due to less memory traffic.

## C++ FLAGS (CXXFLAGS)

–fexceptions, -fno-exceptions
Command-line options that is only meaningful for C++ programs. Enables or disables the generation of code needed to support C++ exceptions. Compiling with -fno-exceptions disables exceptions support and uses the variant of C++ libraries without exceptions. Use of try, catch, or throw results in an error message.

-felide-constructors
Command-line options that is only meaningful for C++ programs. Elide constructors when this seems plausible. The default behavior (-fno-elide-constructors) is specified by the draft ANSI C++ standard. If your programs constructors have side effects, '-felide-constructors can change your programs behavior, since some constructor calls may be omitted.

std=gnu++14
GCC supports the original ISO C++ standard published in 1998, and the 2011 and 2014 revisions. By default, GCC also provides some additional extensions to the C++ language that on rare occasions conflict with the C++ standard. You may also select an extended versionof the C++ language explicitly with -std=gnu++14 (for C++14 with GNU extensions). The default, if no C++ language dialect options are given, is -std=gnu++14.

-fno-rtti
Command-line options that is only meaningful for C++ programs. Disable generation of information about every class with virtual functions for use by the C++ runtime type identification features (dynamic_cast and typeid). If you donot use those parts of the language, you can save some space by using this flag. Note that exception handling uses the same information, but it will generate it as needed.

## LINKER FLAGS

-O2 vs -Os
-O2 nearly all supported optimizations that do not involve a space-speed tradeoff
-Os Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size.

-Wl,option
Pass option as an option to the linker. If option contains commas, it is split into multiple options at the commas.

–gc-sections
This tells linker to remove unused sections.

–relax
An option with machine dependent effects. This option is only supported on a few targets. On some platforms, the '–relax' option performs global optimizations that become possible when the linker resolves addressing in the program On platforms where this is not supported, '–relax' is accepted, but ignored.

–defsym=**rtc_localtime= defsym symbol=expression creates a global symbol in the output file. The teensy firmware uses** rtc_localtime to initialize the RTC. This only ever gets used if you have added the 32.768 kHz RTC crystal, and even then, only if the RTC is found to be uninitialized. The arduino IDE sets this value, a makefile will need a script to do this.

```
TS = $(shell date +%s)
--defsym=__rtc_localtime=$(TS)
```

-T script
Use script as the linker script. This option is supported by most systems using the GNU linker. On some targets, such as bare-board targets without an operating system, the -T option may be required when linking to avoid references to undefined symbols.

-lstdc++
The flag -l simply specifies that the library should be searched when linking. Since we are using arm-none-eabi-gcc, I assume this will use the stdc++ lib in the ~/opt/arduino-1.8.8/hardware/tools/arm/arm-none-eabi/lib directory.

-mthumb
-mcpu=$(CPUARCH)
-mfloat-abi=hard
-mfpu=fpv4-sp-d16
-fsingle-precision-constant
See the common (CPPFLAGS) flags for description.

-l
link with a library file.

The library name is without the lib prefix and the .a or .so extensions.
For example, for static library libmath.a use -lmath

-L
An executable is created with a linker (usually /usr/bin/ld), but the linker is usually called for us by gcc. We tell gcc where the .a or .so files are by using:
-L/path/to/library/code

We also have to tell it which libraries to link with, either by explicitly linking in the .a file or specifying a library by name with the "-l" option.

The core.a library is explicitly specified on the command line.

## objdump Flags

-d
–disassemble
Display the assembler mnemonics for the machine instructions from objfile. This option only disassembles those sections which are expected to contain instructions.

-S
–source
Display source code intermixed with disassembly, if possible. Implies -d.

-C
–demangle[=style]
Decode (demangle) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C ++ function names readable. Different compilers have different mangling styles. The optional demangling style argument can be used to choose an appropriate demangling style for your compiler.

-t
–syms
Print the symbol table entries of the file. This is similar to the information provided by the nm program, although the display format is different

## First Upload

Notes from pjrc.com on using your teensy for the first time.

https://www.pjrc.com/teensy/troubleshoot.html

Teensy uses HID protocol for uploading, not serial. Brand new Teensy boards are shipped with the LED blink example compiled to appear as RawHID. You must program Teensy at least once from Arduino. The COM port (Windows) or Serial Device (Mac, Linux) appears only after Teensy begins running your program. Regular Arduino boards are always serial. Teensy uses HID and supports many protocols. To use serial, make sure the Tools > USB Type menu is set to "Serial", and understand Teensy only becomes a serial device when it runs your program built with this setting.