Продвинутые возможности makefile

Оглавление

1. Введение

Утилита make предназначенна для автоматизации процесса преобразования файлов из одной формы в другую.

Пример: **`make [ -f make-файл ] [ цель ] ...`**

Если опция **-f** не указана, используется имя по умолчанию для **make**-файла — **makefile**.

Правила преобразования задаются в скрипте с именем makefile, который должен находиться в корне рабочей директории проекта.

Сам скрипт состоит из набора правил, которые в свою очередь описываются:

1) целями – идентификаторами правил;
2) реквизитами (зависимостями) то, что необходимо для выполнения правила и получения целей;
3) командами, выполняющими данные преобразования.

В общем виде синтаксис makefile можно представить так набор правил, состоящих из:

```
# Индентация осуществляется исключительно при помощи символов табуляции,
# каждой команде должен предшествовать отступ
<цели>: <реквизиты>
<tab> <команда #1>
        ...
<tab> <команда #n>
```

Если разобраться поглубже, make-файлы состоят из 5 составляющих: *явных правил*, *неявных правил*, *определений переменных*, *директив и комментариев*.

Явное правило сообщает, когда и как нужно преобразовать один или несколько файлов, которые называются целевыми правилами.

Определение переменной является строкой, в которой указывается текстовое строковое значение переменной, которая может быть подставлена в текст позже.

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Неявное правило сообщает, когда и как преобразовать файлы, исходя из их названия (расширения) (подробнее будет рассмотрено ниже).

Директива указывает утилите make на выполнение чего-либо особенного при чтении файла makefile (например, чтение другого makefile).

2. Инкрементная компиляция

После изменения одного из исходных файлов, достаточно произвести его трансляцию и линковку всех объектных файлов.

При этом мы пропускаем этап трансляции не затронутых изменениями реквизитов, что сокращает время компиляции в целом.

Такой подход называется инкрементной компиляцией.https://habr.com/ru/articles/211751/

3. Правила

Правило содержится в make-файле. Оно указывает, когда и как заново порождать определенные файлы, называемые целями правила (чаще всего правилу соответствует только одна цель). В нем перечисляются другие файлы, которые являются зависимостями цели, и команды, используемые для создания или обновления цели.

Порядок правил несущественен, за исключением определения главной цели по умолчанию: цели, с которой начинается работа make, если вы ее не определили. По умолчанию главной целью make является цель первого правила в make-файле.

Поэтому обычно makefile пишется так, чтобы первое правило для компиляции всей программы или всех программ, описываемых makefile (обычно с именем цели «all»).

Дла нашего проекта (см. Приложение 1) описаны следующие правила со своими целями, реквизитами (зависимостями) и соответствующими командами (строка 137,138):

```
# default action: build all
all: $(BUILD_DIR)/$(TARGET).elf $(BUILD_DIR)/$(TARGET).hex $(BUILD_DIR)/$(TARGET).bin -
```

(правило по умолчанию с реквизитами без команды, - построить все цели make-файла, где (BUILD_DIR, TARGET) – это переменные).

Основные правила:

```
#####################################
# clean up
#####################################
clean:
    rm -fR $(BUILD_DIR)
```

- правило только с командами - удалить все файлы, которые, созданы в результате работы makefile (строка 171-175).

```
prog:
    openocd -f interface/stlink.cfg -f target/stm32f3x.cfg -c "program $(BUILD_DIR)/$(TARGET).elf verify exit reset"
```

– правило только с командами - вызов отладчика openocd с командой загрузки прошивки в МК при помощи st-link (строка 182-183).

```
reset:
    openocd -f interface/stlink.cfg -f target/stm32f3x.cfg -c "init" -c "reset" -c "shutdown" -
```

правило только с командами - сброс МК при помощи st-link (строка 185,186)

1) Неявные правила (implicit rules)

Писать команды для компиляции отдельных исходных файлов не является необходимым, поскольку make может сам их определить: он имеет неявное правило для обновления '.o'-файла из соответствующего '.c'-файла, используя команду 'cc -c'. Например, он будет использовать команду 'cc -c main.c -o main.o' для компиляции 'main.c' в 'main.o'.

Следовательно, мы можем опустить команды из правил для объектных файлов.

Очень часто используются определенные стандартные способы порождения заново целевых файлов. Например, одним из типичных способов порождения объектного файла является порождение его из исходного C-файла с использованием C-компилятора, cc.

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
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 -c utils.c
clean :
        rm edit $(objects)
```

Например, make-файл выше можно переписать короче, с помощью неявных правил

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

Неявные правила указывают программе make, как использовать типичные приемы, для того, чтобы чтобы вам не требовалось детально определять их тогда, когда вы хотите использовать их. Например, есть неявное правило для С-компиляции. Имена файлов определяют, какие неявные правила вступают в действие. Например, при С-компиляции обычно берется файл с именем, оканчивающемся на '.c' и порождается файл с именем, оканчивающемся на '.o'. Таким образом, программа make применяет неявное правило для С-компиляции, когда она обнаруживает эту комбинацию окончаний имен файлов.

Встроенные неявные правила используют в своих командах несколько переменных, так что путем изменения значений переменных вы можете изменять способ, в соответствии с которым работает неявное правило. Например, переменная CFLAGS управляет опциями, передаваемыми С-компилятору неявными правилами для С-компиляции.

Вот перечень предопределенных неявных правил, которые всегда доступны, если make-файл явным образом не перекрывает и не отменяет их. Опция '-r' или '--no-builtin-rules' отменяет все предопределенные правила.

- **Компиляция С-программ**
  - 'n.o' автоматически порождается из 'n.c' при помощи команды в форме '$(CC) -c $(CPPFLAGS) $(CFLAGS)'.

2) Шаблонные правила

Вы определяете неявное правило с помощью записи шаблонного правила. Шаблонное правило похоже на обычное правило, за исключением того, что его цель содержит символ '%' (ровно один). Цель рассматривается как шаблон для сопоставления с ним имен файлов - символу '%' может соответствовать любая непустая подстрока, в то время как любому другому символу соответствует только он сам. Зависимости также используют символ '%', чтобы показать как их имена соотносятся с именем цели.

Таким образом, шаблонное правило '%.o : %.c' указывает программе make, как на породить любой файл с именем вида 'stem.o' из другого файла, с именем вида 'stem.c'.

Обратите внимание, что в шаблонных правилах подстановка значения, соответствующего символу '%' происходит после подстановки значений всех переменных и функций, что имеет место при чтении make-файла.

Пример шаблонных правил

Вот некоторые примеры шаблонных правил, на самом деле являющихся предопределенными программой make. Сначала рассмотрим правило, которое компилирует '.c'-файлы в '.o'-файлы:

```
%.o : %.c
        $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

Приведенный фрагмент make-файла определяет правило, которое может породить любой файл 'x.o' из файла 'x.c'. Команда использует автоматические переменные '$@ ' и '$<', которые на место которых каждый раз при применении правила подставляются, соответственно, имена целевого и исходного файлов.

Вот второе встроенное правило:

```
% :: RCS/%,v
        $(CO) $( COFLAGS) $<
```

Здесь определяется правило, которое может породить произвольный файл 'x' из соответствующего файла 'x,v' в подкаталоге 'RCS'. Поскольку целью является '%', это правило применится к любому файлу, при условии, что соответствующий файл зависимости существует. Двойное двоеточие делает правило терминальным, и это означает, что его зависимость не может быть промежуточным файлом.

Следующее шаблонное правило имеет две цели:

```
%.tab.c %.tab.h: %.y
	bison -d $<
```

Это указывает программе make на то, что команда 'bison -d X.y' будет порождать как файл 'x.tab.c' так и файл 'x.tab.h'. Если файл 'foo' зависит от файлов 'parse.tab.o' и 'scan.o', а файл 'scan.o' зависит от файла 'parse.tab.h', то при изменении файла 'parse.y' изменяется, команда 'bison -d parse.y' будет выполнена только один раз, в результате чего будут обновлены зависимости, как файла 'parse.tab.o', так и файла 'scan.o'. (Предполагается, файл 'parse.tab.o' будет перекомпилироваться из файла 'parse.tab', а файл'scan.o' - из 'scan.c', в то время как файл 'foo' компонуется из файлов 'parse.tab.o', 'scan.o', и других его зависимостей, и после этого он будет всегда счастливо выполняться.)

Шаблонные правила для создания объектных файлов и файлов прошивки:

```
$(BUILD_DIR)/%.o: %.c Makefile | $(BUILD_DIR)
	$(CC) -c $(CFLAGS) -Wa,-a,-ad,-alms=$(BUILD_DIR)/$(notdir $(<:.c=.lst)) $< -o $@
```

(Правило с реквизитами и командами - создание объектных файлов, из всех исходных файлов, где (`BUILD_DIR, CC, CFLAGS`) – это переменные) (строка 152-153).

```
$(BUILD_DIR)/%.o: %.S Makefile | $(BUILD_DIR)
	$(AS) -c $(CFLAGS) $< -o $@
```

– (правило с реквизитами и командой - создание объектного файлов, из файла startup файла, где (`BUILD_DIR, AS, CFLAGS`) – это переменные, (`$< $@`) – автоматические переменные) (строка 155, 156).

```
$(BUILD_DIR)/$(TARGET).elf: $(OBJECTS) Makefile
	$(CC) $(OBJECTS) $(LDFLAGS) -o $@
	$(SZ) $@
```

– (правило с реквизитами и командами - создание файлов .elf, где (`BUILD_DIR, TARGET, CC, LDFLAGS, OBJECTS, SZ`) – это переменные (`$@`) – автоматические переменные) (строка 158-160).

```
$(BUILD_DIR)/%.hex: $(BUILD_DIR)/%.elf | $(BUILD_DIR)
	$(HEX) $< $@
```

– (правило с реквизитами и командой - создание файлов .hex, где (`BUILD_DIR, HEX`) – это переменные, (`$< $@`) – автоматические переменные) (строка 162,163).

```
$(BUILD_DIR)/%.bin: $(BUILD_DIR)/%.elf | $(BUILD_DIR)
	$(BIN) $< $@
```

– (правило с реквизитами и командой - создание файла .bin, где (`BUILD_DIR, BIN`) – это переменные, (`$< $@`) – автоматические переменные) (строка 165,166).

```
$(BUILD_DIR):
	mkdir $@
```

– (цель (правило) с командой - создание папки содержащей файлы прошивки, где (`BUILD_DIR`) – это переменная) (строка 168,169).

Некоторые переменных для хранения имен программ, используемые в неявных правилах:

AS	Ассемблер; по умолчанию, `as'.
CC	Компилятор Си; по умолчанию, `cc'.
CPP	Препроцессор языка Си, выдающий результат на стандартный вывод; по умолчанию, `$(CC) -E'.
MAKEINFO	Программа для преобразования исходного файла формата Texinfo в файл формата Info; по умолчанию, `makeinfo'.
RM	Команда удаления файла; по умолчанию, `rm -f'.

Ниже приведена таблица переменных, содержащих дополнительные параметры для перечисленных выше программ. По умолчанию, значением этих переменных является пустая строка (если не указано другое).

ARFLAGS Опции, передаваемые программе, манипулирующей с архивными файлам; по умолчанию `rv'.
ASFLAGS Дополнительные параметры, передаваемые ассемблеру (при его явном вызове для файлов `.s' и `.S').
CFLAGS Дополнительные параметры, передаваемые компилятору Си.
CXXFLAGS Дополнительные параметры, передаваемые компилятору C++.
CPPFLAGS Дополнительные параметры, передаваемые препроцессору языка Си и программам, его использующим (компиляторам Си и Фортрана).
LDFLAGS Дополнительные параметры, передаваемые компиляторам, когда предполагается вызов компоновщика `ld'.

4. Переменные

Автоматические переменные

Символ подстановки *. Оба символа * и % называются в технологии Make символами подстановки (wildcards), однако они означают полностью не то, что они означают для имен файловой системы. В файловой системе * при поиске соответствует именам файлов. Рекомендуется всегда использовать символы макроподстановки, обозначающие несколько файлов, с помощью следующей конструкции на основе функции wildcard:

```
# Напечатает информацию о каждом файле .c:
print: $(wildcard *.c)
→ls -la  $?
```

Внимание: * может косвенно использоваться в определениях переменных. Когда * не соответствует ни одном файлу, то он интерпретируется как есть, за исключением работы в функции wildcard. Поэтому обязательно используйте фукцию wildcard.

```
thing_wrong := *.o # Не делайте так! '*' не будет развернут.
thing_right := $(wildcard *.o)

all: one two three four

# Приведет к ошибке, потому что $(thing_wrong) это даст строку "*.o"
one: $(thing_wrong)

# Останется как *.o, если ни одного файла не попадет под этот шаблон :(
two: *.o

# Сработает, как и ожидалось! В этом примере не будет никаких действий.
three: $(thing_right)

# То же самое, что правило three.
four: $(wildcard *.o)
```

Автоматические переменные. Эти переменные используются в шабонный правилах Существует несколько автоматических переменных, но часто используются только несколько:

```
hey: one two
→# Выведет "hey", поскольку $@ соответствует имени цели (target name):
→echo $@

→# Выведет все зависимости (prerequisites), которые более свежие, чем target:
→echo $?
```

```
→# Выведет все зависимости:
→echo $^
→touch hey

one:
→touch one

two:
→touch two

clean:
→rm -f hey one two
```

Значения этих переменных автоматически вычисляются заново для каждого исполняемого правила в зависимости от его целей и пререквизитов.

Полный перечень имеющихся автоматических переменных:

$@ Имя файла цели правила.

Если цель является элементом архива (archive member), то `$@' обозначает имя архивного файла. В шаблонном правиле с несколькими целями `$@' обозначает имя цели, которая вызывала запуск команд данного правила.

$<

Имя первого пререквизита. В случае, если выполняемые команды относятся к неявному правилу, первым пререквизитом является тот, который был указан в неявном правиле (строк 158-160).

```
$(BUILD_DIR)/$(TARGET).elf: $(OBJECTS) Makefile
    $(CC) $(OBJECTS) $(LDFLAGS) -o $@
    $(SZ) $@
```

https://www.opennet.ru/docs/RUS/make_compile/make-10.html#ss10.5

Переменные

https://microsin.net/programming/arm/learning-makefile-with-simple-examples.html

Переменные могут быть только строками. Для них обычно используют присваивание **:=**, но также работает и присваивание **=**, см. далее "Переменные Makefile, часть 2".

```
files := file1 file2
some_file: $(files)
```

Одиночные или двойные кавычки для make ничего не значат. Они просто символы, которые присваиваются переменной.

```
a := one two # эта команда установит строку a в значение "one two"
b := 'one two' # Не рекомендуется. Переменная b установится в строку"'one two'".
all:
→printf '$a'
→printf $b
```

Обращаются к переменным с помощью либо ${}, либо $(). В следующем примере происходит обращение к значению переменной x:

```
x := dude

all:
→echo $(x)
→echo ${x}

→# Плохая практика, но это работает:
```

```
→echo $x
```

Переменные в makefile представляют собой именованные строки и определяются следующую конструкцию:

```
<VAR_NAME> = <value string>
```

В Makefile проекта переменные представлены следующим образом:

`TARGET = gd32f303_blink_make` – (имён исполняемых файлов (`$(TARGET).elf`)) (строка 16).
`BUILD_DIR = build` - (имени папки, куда буду генерироваться файлы (.elf, .bin, .hex)) (строка 32).
`GCC_PATH = c:/ST/STM32CubeIDE_1.15.0/STM32CubeIDE/plugins/com.st.stm32cube.ide.mcu.externaltools.gnu-tools-for-stm32.12.3.rel1.win32_1.0.100.202403111256/tools/bin` -(пути к компилятору arm-none-eabi-gcc) (строка 33)

```
#####################################
# source
#####################################
# C sources
C_SOURCES =  \
$(shell find ./ -type f -name *.c)
```

– (всех исходных файлы (.c) проекта) `(строка 34-39)`

```
# ASM sources
ASM_SOURCES =  \
startup/startup_gd32f30x_hd.S
```

- (файла инициализации МК (ассемблерный файл)) (строк 41-43)

Переменные используемых в неявных правилах проекта:

`PREFIX = arm-none-eabi-` `(строка 49)`

Имена программм (строка 52-64):

```
ifdef GCC_PATH
CC = $(GCC_PATH)/$(PREFIX)gcc
AS = $(GCC_PATH)/$(PREFIX)gcc -x assembler-with-cpp
CP = $(GCC_PATH)/$(PREFIX)objcopy
SZ = $(GCC_PATH)/$(PREFIX)size
else
CC = $(PREFIX)gcc
AS = $(PREFIX)gcc -x assembler-with-cpp
CP = $(PREFIX)objcopy
SZ = $(PREFIX)size
endif
HEX = $(CP) -O ihex
BIN = $(CP) -O binary -S
```

Имена центрального процессора, сопроцессоров передаваемых в компилятор (строка 69-79):

```
# cpu
CPU = -mcpu=cortex-m4
# fpu
FPU = -mfpu=fpv4-sp-d16
# float-abi
FLOAT-ABI = -mfloat-abi=hard
# mcu
MCU = $(CPU) -mthumb $(FPU) $(FLOAT-ABI)
```

Макросы препроцессора (строка 86-97):

```
# C defines
C_DEFS =  \
-DUSE_FULL_LL_DRIVER \
-DHSE_VALUE=8000000 \
-DHSE_STARTUP_TIMEOUT=100 \
-DLSE_STARTUP_TIMEOUT=5000 \
-DLSE_VALUE=32768 \
-DHSI_VALUE=8000000 \
-DLSI_VALUE=40000 \
-DVDD_VALUE=3300 \
-DPREFETCH_ENABLE=1 \
-DGD32F30X_HD\
-DHXTAL_VALUE=8000000
```

Пути к исходным файлам .c (строка 105-109) :

```
# C includes
C_INCLUDES =  \
-ICMSIS \
-ICore/Inc \
-ICMSIS/GD/GD32F30x/Include\
-IGD32F30x_standard_peripheral/Include
```

Переменные дополнительных параметров неявных правил (строка 112 - 135):

```
# compile gcc flags
ASFLAGS = $(MCU) $(AS_DEFS) $(AS_INCLUDES) $(OPT) -Wall -fdata-sections -ffunction-sections

CFLAGS += $(MCU) $(C_DEFS) $(C_INCLUDES) $(OPT) -Wall -fdata-sections -ffunction-sections

ifeq ($(DEBUG), 1)
CFLAGS += -g -gdwarf-2
endif

# Generate dependency information
CFLAGS += -MMD -MP -MF"$(@:%.o=%.d)"

#######################################
# LDFLAGS
#######################################
# link script
LDSCRIPT = ldscripts/gd32f30x_flash.ld

# libraries
LIBS = -lc -lm -lnosys
LIBDIR =
LDFLAGS = $(MCU) -specs=nano.specs -T$(LDSCRIPT) $(LIBDIR) $(LIBS)
-Wl,-Map=$(BUILD_DIR)/$(TARGET).map,--cref -Wl,--gc-sections
```

Для использование значения переменной необходимо провести разименование переменной при помощи следующей конструкции $(<VAR_ NAME>). Пример из проекта - `$(GCC_PATH), $(TARGET).`

5. Функции преобразования текста

Функции обработки имен файлов:

- $(dir *имена*...)

Из каждого имени файла, перечисленного в *именах*, выделяет имя каталога, где этот файл расположен. Именем каталога считается часть имени до последнего встреченного символа `/' (включая и этот символ). Если имя файла не содержит символов `/', его именем каталога считается строка `./'. Результатом следующего примера:

$(dir src/foo.c hacks)

будет строка `src/ ./'.

`$(dir $(C_SOURCES))` стр (146,149).

- $(notdir *имена*...)

Из каждого имени файла, перечисленного в *именах*, удаляет имя каталога, где он находится. Имена файлов, не содержащие символов `/', остаются без изменений. В противном случае (при наличии символов `/'), из имени файла удаляется все, что расположено до последнего встреченного символа `/' (включая и сам этот символ). Это означает, что имя файла, оканчивающееся символом `/' преобразуется в пустую и строку, и, таким образом, количество имен файлов на выходе функции может не совпадать с количеством имен файлов, переданных ей на вход. К сожалению, мы пока не видим лучшей альтернативы. Результатом следующего примера:

$(notdir src/foo.c hacks)

будет строка `foo.c hacks'.

`$(notdir $(<:.c=.lst))` (строка, 145,148,152).

- $(addprefix *префикс*,*имена*...)

Аргумент *имена* рассматривается как последовательность разделенных пробелами имен; аргумент *префикс* рассматривается как строка. Результатом этой функции является список имен (разделенных одиночными символами пробелов), каждое из которых получено из соответствующего "исходного" имени, в начало которого добавлен префикс *префикс*. Например, результатом следующего выражения:

$(addprefix src/,foo bar)

будет строка `src/foo src/bar'.

`$(addprefix $(BUILD_DIR))` (строка 145,148).

- $(wildcard *шаблон*)

Аргумент *шаблон* является шаблоном имени файла и, обычно, содержит шаблонные символы (такие же как в шаблонах имен файлов интерпретатора командной строки). Результатом функции wildcard является список разделенных пробелами имен существующих в данный момент файлов, удовлетворяющих указанному шаблону.

Пример:

`$(wildcard $(BUILD_DIR)/*.d)` (строка 180).

Функция shell

Функция shell служит для "общения" make с внешним миром, она производит подстановку результата выполнения команды. Это означает, что в качестве аргумента она принимает команду

интерпретатора командной строки, а в качестве результата возвращает "выходные данные" этой команды. Единственным преобразованием полученного результата, которое выполняет make перед подстановкой его в окружающий текст, является преобразование символов перевода строки (или пар перевод-строки/возврат-каретки) в одиночные пробелы. Также производится удаление "конечных" (находящихся в конце данных) символов перевода строки (или пар перевод-строки/возврат-каретки).

Команды, указанные в shell, запускаются в момент вычисления этой функции. Как правило, это происходит в момент чтения make-файла. Исключение составляет случай, когда эта функция shell используется в командах правила. В этом случае она будет вычисляться (будут выполняться указанные в ней команды) во время работы команд правила.

Вот несколько примеров использования функции shell:

contents := $(shell cat foo)

В этом примере, в переменную contents записывается содержимое файла `foo' (видоизмененное таким образом, что все символы перевода строки заменены в нем на пробелы). В следующем примере:

files := $(shell echo *.c)

в переменную files записывается список файлов, полученных по маске `*.c'.

`$(shell find ./ -type f -name *.c)` (строка 39).

https://rus-linux.net/nlib.php?name=/MyLDP/algol/gnu_make/gnu_make_3-79_russian_manual.html#SEC101
https://www.opennet.ru/docs/RUS/make_compile/make-4.html

GNU MAKE:

https://www.opennet.ru/docs/RUS/make_compile/

https://www.chiark.greenend.org.uk/doc/make-doc/make.html/index.html#SEC_Contents

Приложение 1

```
#############################################################################################################
####
# File automatically-generated by tool: [projectgenerator] version: [4.2.0-B44] date: [Sun Mar 02 17:41:17
MSK 2025]
#############################################################################################################
###

# ------------------------------------------------
# Generic Makefile (based on gcc)
#
# ChangeLog :
#   2017-02-10 - Several enhancements + project update mode
#   2015-07-22 - first version
# ------------------------------------------------

######################################
# target
######################################
TARGET = blink_f303CCT6


######################################
# building variables
######################################
# debug build?
DEBUG = 1
# optimization
OPT = -Og


#######################################
# paths
#######################################
# Build path
BUILD_DIR = build
GCC_PATH =
C:/ST/STM32CubeIDE_1.15.0/STM32CubeIDE/plugins/com.st.stm32cube.ide.mcu.externaltools.gnu-tools-for-stm32.12
.3.rel1.win32_1.0.100.202403111256/tools/bin
#######################################
# source
#######################################
# C sources
C_SOURCES =  \
$(shell find ./ -type f -name *.c)

# ASM sources
ASM_SOURCES =  \
startup_gd32f30x_xd.S



#######################################
# binaries
#######################################
PREFIX = arm-none-eabi-
# The gcc compiler bin path can be either defined in make command via GCC_PATH variable (> make
GCC_PATH=xxx)
# either it can be added to the PATH environment variable.
ifdef GCC_PATH
CC = $(GCC_PATH)/$(PREFIX)gcc
AS = $(GCC_PATH)/$(PREFIX)gcc -x assembler-with-cpp
CP = $(GCC_PATH)/$(PREFIX)objcopy
SZ = $(GCC_PATH)/$(PREFIX)size
else
CC = $(PREFIX)gcc
AS = $(PREFIX)gcc -x assembler-with-cpp
CP = $(PREFIX)objcopy
SZ = $(PREFIX)size
endif
HEX = $(CP) -O ihex
BIN = $(CP) -O binary -S

#######################################
# CFLAGS
#######################################
```

```
# cpu
CPU = -mcpu=cortex-m4

# fpu
# FPU = -mfpu=fpv4-sp-d16

# float-abi
# FLOAT-ABI = -mfloat-abi=hard

# mcu
MCU = $(CPU) -mthumb $(FPU) $(FLOAT-ABI)

# macros for gcc
# AS defines
AS_DEFS =

# C defines
C_DEFS =  \
-DUSE_FULL_LL_DRIVER \
-DHSE_VALUE=8000000 \
-DHSE_STARTUP_TIMEOUT=100 \
-DLSE_STARTUP_TIMEOUT=5000 \
-DLSE_VALUE=32768 \
-DHSI_VALUE=8000000 \
-DLSI_VALUE=40000 \
-DVDD_VALUE=3300 \
-DPREFETCH_ENABLE=1 \
-DGD32F30X_XD\
-DHXTAL_VALUE=8000000

# AS includes
AS_INCLUDES =

# C includes
C_INCLUDES =  \
-ICore \
-ICore/Inc \
-IDrivers/CMSIS \
-IDrivers/CMSIS/GD/GD32F30X/Include \
-IDrivers/GD32F30x_standard_peripheral/Include


# compile gcc flags
ASFLAGS = $(MCU) $(AS_DEFS) $(AS_INCLUDES) $(OPT) -Wall -fdata-sections -ffunction-sections

CFLAGS += $(MCU) $(C_DEFS) $(C_INCLUDES) $(OPT) -Wall -fdata-sections -ffunction-sections

ifeq ($(DEBUG), 1)
CFLAGS += -g -gdwarf-2
endif


# Generate dependency information
CFLAGS += -MMD -MP -MF"$(@:%.o=%.d)"


#######################################
# LDFLAGS
#######################################
# link script
LDSCRIPT = \
ldscripts/gd32f30x_flash.ld

# libraries
LIBS = -lc -lm -lnosys
LIBDIR =
LDFLAGS = $(MCU) -specs=nano.specs -specs=nosys.specs -u _printf_float -T$(LDSCRIPT) $(LIBDIR) $(LIBS)
-Wl,-Map=$(BUILD_DIR)/$(TARGET).map,--cref -Wl,--gc-sections

# default action: build all
all: $(BUILD_DIR)/$(TARGET).elf $(BUILD_DIR)/$(TARGET).hex $(BUILD_DIR)/$(TARGET).bin


#######################################
# build the application
#######################################
# list of objects
OBJECTS = $(addprefix $(BUILD_DIR)/,$(notdir $(C_SOURCES:.c=.o)))
vpath %.c $(sort $(dir $(C_SOURCES)))
# list of ASM program objects
```

```
OBJECTS += $(addprefix $(BUILD_DIR)/,$(notdir $(ASM_SOURCES:.S=.o)))
vpath %.s $(sort $(dir $(ASM_SOURCES)))

$(BUILD_DIR)/%.o: %.c Makefile | $(BUILD_DIR)
	$(CC) -c $(CFLAGS) -Wa,-a,-ad,-alms=$(BUILD_DIR)/$(notdir $(<:.c=.lst)) $< -o $@


$(BUILD_DIR)/%.o: %.S Makefile | $(BUILD_DIR)
	$(AS) -c $(CFLAGS) $< -o $@

$(BUILD_DIR)/$(TARGET).elf: $(OBJECTS) Makefile
	$(CC) $(OBJECTS) $(LDFLAGS) -o $@
	$(SZ) $@

$(BUILD_DIR)/%.hex: $(BUILD_DIR)/%.elf | $(BUILD_DIR)
	$(HEX) $< $@

$(BUILD_DIR)/%.bin: $(BUILD_DIR)/%.elf | $(BUILD_DIR)
	$(BIN) $< $@

$(BUILD_DIR):
	mkdir $@

#######################################
# clean up
#######################################
clean:
	-rm -fR $(BUILD_DIR)

#######################################
# dependencies
#######################################
-include $(wildcard $(BUILD_DIR)/*.d)

prog:
	openocd -f interface/stlink.cfg -f target/stm32f3x.cfg -c "program $(BUILD_DIR)/$(TARGET).elf verify
exit reset"

reset:
	openocd  -f interface/stlink.cfg -f target/stm32f3x.cfg -c "init" -c "reset" -c "shutdown"
# *** EOF ***
```