« take me back please!

Getting Started with the STM32F4 and GCC

Preface

Hi.

Overview of this guide

This guide explains how to get a fully working ARM-GCC toolchain working under Ubuntu Linux, and provides makefiles that are specifically targeted towards the STM32F4 series of microcontrollers. Unfortunately, to load the code on the board, a windows computer (or VM) is needed. I am investigating linux based tools but I am yet to get one working.

One important aspect of this guide is that the compiler is built with hardfloat support. One of the major features of the ARM Cortex-M4 series is the hardware acceleration of floating point operations; however, most free toolchains and compilers don't provide support for it (you need to cough up some dough for the non-free compiler).

Assumed background

To follow this guide, you should already know how to navigate and run commands within the linux terminal. Commands will look like this:

```
$ apt-get moo
```

means put "apt-get moo" in your favourite terminal software (or any terminal software really...). You should also know some basic C programming.

A special note for those who have not previously used ARM microcontrollers

Most microcontroller systems have very tight integration between the CPU and the on-chip peripherals due to the manufacturer designing and producing both subsystems. In the case of ARM-based microcontrollers though, a company named ARM Holdings designs the core and licenses it to manufacturers like ST (or NXP, Apple, Samsung, Qualcomm, HP, etc). This means that the manufacturer can spend time on making the on-chip peripherals powerful and reliable while ARM deals with things like power efficiency and instruction set design. On top of this, if the manufacturer follows the CMSIS guidelines, porting software between chips from different

manufacturers is a piece of cake. Although peripheral use can be a touch more complicated than in a microcontroller developed completely by one manufacturer, using ARM-based microcontrollers is a much more cost-effective solution if you need high clock speeds and low power (YMMV, of course).

Stage 1: Build a Toolchain

Our build environment will be based on Ubuntu 11.10, so if you don't already have it grab the Ubuntu ISO now and install it on a PC or in a virtual machine.

The toolchain we will be using is a modified version of Summon-Arm-Toolchain. Summon-Arm-Toolchain is a shell script which downloads, builds and installs a fully working ARM toolchain for the Cortex-M3 (nice!). On top of this, the amazing MikeSmith has already performed the necessary modifications to build for the Cortex-M4 with hardfloat support with no extra work on our end (double nice!). To get started, we need to install all of its dependencies:

```
$ sudo apt-get install git zlib1g-dev libtool flex \
bison libgmp3-dev libmpfr-dev libncurses5-dev libmpc-dev \
autoconf texinfo build-essential libftdi-dev
```

Astute readers will note that a few of these aren't listed as dependencies on the Summon-Arm-Toolchain page. I am not sure why. All of these are essential for building the toolchain.

Once that is complete, let's clone the Summon-Arm-Toolchain repository:

```
$ git clone https://github.com/MikeSmith/summon-arm-toolchain.git
```

Now enter the directory and start the build process:

```
$ cd summon-arm-toolchain
$ ./summon-arm-toolchain
```

This will take a while, so go take a nap.

Once this is complete, add the "~/sat/bin" directory to your path. I did this by adding the following line to my ~/.profile file:

```
export PATH=$PATH:/home/jeremy/sat/bin
```

You can reload the file by running the command:

```
$ . ~/.profile
```

Check if it works by running the following command:

```
$ arm-none-eabi-gcc --version
```

And if you see something like,

```
arm-none-eabi-gcc (Linaro GCC 4.6-2011.10) 4.6.2 20111004 (prerelease)
Copyright (C) 2011 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

you have an ARM toolchain capable of building binaries for the Cortex-M4!

Stage 2: Uh, so what do I do now?

It's time to start writing some code! If you are at all familiar with embedded development, you might be familiar with the Microcontroller Programming Paradigm(tm):

```
1. Decide what peripheral you want to use
2. Look in datasheet for the registers to enable and configure it
3. Set bits in the registers to make peripheral behave the way you want
4. GOTO 1
```

In the case of the STM32F4, the datasheet has very little information on which registers to use. You can find it all tucked away in ST's RM0090: STM32F4xx Advanced ARM-based 32-bit MCUs Reference Manual. There is also some information on the features of the ARM core in the ARM Cortex-M4F Technical Reference Manual. Finally, ST have published UM1472: STM32F4 High-Performance Discovery Board User Manual which tells you how devices are connected together on the board. Make sure you have these three saved somewhere so you can refer to them later.

Stage 2A: Building a Blinky

> **Note:** This section asks you initially to compile code that is syntactically correct but functionally incorrect. This is intentional; I am hoping to demonstrate the sort of traps that you can fall into when developing for these chips. If you get easily frustrated, consider reading the whole section before compiling anything yourself.

To start a project from scratch, the first thing we normally do is work out how to use the compiler to compile our code. Unfortunately, it's a little complicated and so for now we will be jumping straight to the code writing stage. To do this, we will be using my stm32-template project. So the first thing we need to do is get a copy of it:

```
$ mkdir ~/stm32_code
$ cd ~/stm32_code
$ git clone git://github.com/jeremyherbert/stm32-templates.git
```

Once the cloning is complete, take a look in the directory. You should find two directories, both of which contain a template build environment for two different ST development kits. Since we are working with the F4, we obviously want the "stm32f4-discovery" template. So let's copy it to a new place so we can use it:

```
$ cp -r stm32-templates/stm32f4-discovery blinky
```

Now let's have a look at the directory structure of the template project:

```
inc/
lib/
src/
Makefile
stm32_flash.ld
```

The `inc/` folder is there for you to put your *.h files in. Likewise, the `src/` folder is for your *.c files. The `lib/` folder is where we store all of the support files that ST have provided for the STM32F4 family of microcontrollers; you shouldn't need to change anything in here, but have a look if you are curious. The `Makefile` instructs the `make` command on how to build our project (more on this later) and the `stm32_flash.ld` file tells the compiler how to arrange the compiled information.

Before we start writing code, we need to clean up the template. It is set up by default to build the IOToggle example from ST, but we would much prefer to write our own code. So from `blinky/`, run the following commands to remove the files we are not interested in:

```
$ rm inc/stm32f4xx_it.h src/stm32f4xx_it.c src/main.c
$ touch src/main.c
```

Now we need to change `Makefile` to tell the compiler we are only compiling main.c. Change this:

```
SRCS = main.c stm32f4xx_it.c system_stm32f4xx.c
```

to this:

```
SRCS = main.c system_stm32f4xx.c
```

Ah, now we have a nice clean build environment.

Stage 2B: Actually writing some code, for reals this time

Let's now open up `src/main.c` and set it up for some serious C coding:

main-1.c:

```
?
1#include "stm32f4xx_conf.h"
2
3int main(void)
4{
5
6}
```

Now let's start at step 1 of the Microcontroller Programming Paradigm(tm)

and look up how to control the GPIOs (General Purpose Input/Output) on the STM32F4. Looking in the contents, we can see that the GPIO-related information starts on page 136 in section 6, so open up your reference manual to that page and have a quick glance through. If you are used to 8 bit microcontrollers, you might be surprised as to how much more complex these chips are.

If you have decided that it looks too complicated and don't want to continue, try watching this video. Otherwise, let's open up the document to the GPIO register listing (section 6.4/page 148). Read the whole thing if you like, but we will cheat for now and I will tell you that the registers we are interested in are `GPIOx_MODER` and `GPIOx_ODR` which will set the set the direction and output value respectively.

To set `GPIOx_MODER`, let's take a look at the table. There are 16 pins on each GPIO output port, so 16 two bit groups are used to configure the pin direction. Looking through the description below the table, it should be clear that we want "01: General purpose output mode" so we can turn the LED on and off. But which bit-pair do we want? The answer is in the STM32F4-Discovery User Manual (see above for the link), in section 4.4/page 16. It says:

> User LD3: orange LED is a user LED connected to the I/O PD13 of the STM32F407VGT6.

The PD13 means that we want pin 13 of GPIOD and thus we want MODER13; the 13th pair slot. Or to put it another way, we want the 26th bit of `GPIOD_MODER` to be 1. Let's put that in our code:

main-2.c:

```
#include "stm32f4xx_conf.h"

int main(void)
{
    GPIOD->MODER = (1 << 26); // set pin 13 to be general purpose output
}
```

If you have done a bit of microcontroller programming before, you might be surprised to see the "->" (otherwise known as the structure dereference operator) in the left half of the new statement. We use it because ST organises registers by defining them as structures. Looking at `stm32f4xx.h` should make more sense:

stm32f4xx-truncated.h:

```
 /* Lots up here */

 typedef struct
 {
```

```
  __IO uint32_t MODER;    /*!< GPIO port mode register,               Address
offset: 0x00      */
  __IO uint32_t OTYPER;   /*!< GPIO port output type register,        Address
offset: 0x04      */
  __IO uint32_t OSPEEDR;  /*!< GPIO port output speed register,       Address
offset: 0x08      */
  __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register,  Address
offset: 0x0C      */
  __IO uint32_t IDR;      /*!< GPIO port input data register,         Address
offset: 0x10      */
  __IO uint32_t ODR;      /*!< GPIO port output data register,        Address
offset: 0x14      */
  __IO uint16_t BSRRL;    /*!< GPIO port bit set/reset low register,  Address
offset: 0x18      */
  __IO uint16_t BSRRH;    /*!< GPIO port bit set/reset high register, Address
offset: 0x1A      */
  __IO uint32_t LCKR;     /*!< GPIO port configuration lock register, Address
offset: 0x1C      */
  __IO uint32_t AFR[2];   /*!< GPIO alternate function registers,     Address
offset: 0x20-0x24 */
} GPIO_TypeDef;

/* Lots in between */

#define GPIOD               ((GPIO_TypeDef *) GPIOD_BASE)

/* and heaps more down here too */
```

Given that the registers are located sequentially in memory, this structure simply maps the registers to human readable names.

So now that we know how to use registers, let's turn on and off the LED using the ODR register (and a XOR trick).

main-3.c:

```
#include "stm32f4xx_conf.h"

int main(void)
{
    GPIOD->MODER = (1 << 26); // set pin 13 to be general purpose output

    while (1) GPIOD->ODR ^= (1 << 13);
}
```

Now if you build this code, you can load it onto your device using the ST-LINK Utility under a Windows VM/system. Unfortunately though, it won't work.

Stage 2C: Enabling peripheral clocks

Over the last half-decade, dramatically lowering current draw has been a goal for most microcontroller manufacturers. One of the techniques used to achieve this is to switch off on-chip peripherals by removing access to their master clocks. On the STM32 devices, these clocks are known as the hardware and peripheral clocks and are controlled by the RCC (Reset and Clock Control) group of registers. Since there are more than 32 on chip peripherals, there are actually two registers used to switch on a clock: `RCC_AHB1ENR` and `RCC_AHB2ENR` (for the **H**ardware clock, `APB` for the **P**eripheral clock). The clock is controlled by set/reset registers, so to turn a system on you set a bit in the ENR register, and to turn that same peripheral off you set the bit in the corresponding `RCC_AHBxRSTR` register. Go and have a read of the register descriptions now, they start on page 93 (section 5.3) of the STM32F4 Reference Manual. To switch GPIOD on, we do something like this:

main-4.c:

```
?
1 #include "stm32f4xx_conf.h"
2
3 int main(void)
4 {
5     RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // enable the clock to GPIOD
6
7     GPIOD->MODER = (1 << 26); // set pin 13 to be general purpose output
8
9     while (1) GPIOD->ODR ^= (1 << 13);
10}
```

Other on-chip systems use the peripheral bus, so be careful when checking whether you are using `AHB` or `APB` registers. You are only one keystroke away from a well hidden bug.

You should also notice the register define I used to set the bit. ST has kindly written out human-readable names for each bit in configuration registers. The pattern should be fairly obvious: `<register group>_<register name>_<bit name>`. You should always use these defines when configuring your device so that you don’t need to continuously need to refer to the datasheet to look up the register structure.

Now our code is ready for primetime! Load it up on the chip and you will see…

…nothing.

Stage 2D: Slowing down

The problem we have now is that the CPU is toggling the register too damn fast! The STM32F4 family of microcontrollers are rated up to 168MHz, but if

you look closely in the datasheet the hardware clock only ever reaches 100MHz! Although we could just put a very slow piece of code in our loop, I would like to take this opportunity to introduce another on-chip peripheral: the general purpose timer. This peripheral allows us to count up to a value and then generate an Interrupt ReQuest (IRQ) which then triggers the execution of a corresponding Interrupt Service Routine (ISR). There are actually many other modes of operation which you can read about in the datasheet, but this is how we will be using it. In this case we will be using TIM2, so first of all let's enable the peripheral clock for it.

main-5.c:

```
?
1 #include "stm32f4xx_conf.h"
2
3 int main(void)
4 {
5     RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // enable the clock to GPIOD
6     RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; // enable TIM2 clock
7
8     GPIOD->MODER = (1 << 26); // set pin 13 to be general purpose output
9
10    while (1) GPIOD->ODR ^= (1 << 13);
11}
```

Now we need to look at the set registers that configure the timer. The comments should briefly explain what each register does; make sure you read the datasheet if you need more information. One important thing to note is that you need to tell the timer that you've changed its configuration. This is done by setting the first bit of the EGR register to 1.

main-6.c:

```
?
1 #include "stm32f4xx_conf.h"
2
3 int main(void)
4 {
5     RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // enable the clock to GPIOD
6     RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; // enable TIM2 clock
7
8     GPIOD->MODER = (1 << 26); // set pin 13 to be general purpose output
9
10    TIM2->PSC = 0x0; // no prescaler, timer counts up in sync with the peripheral
11clock
12    TIM2->DIER |= TIM_DIER_UIE; // enable update interrupt
13    TIM2->ARR = 0x01; // count to 1 (autoreload value 1)
14    TIM2->CR1 |= TIM_CR1_ARPE | TIM_CR1_CEN; // autoreload on, counter enabled
15    TIM2->EGR = 1; // trigger update event to reload timer registers
16
17    while (1);
```

```
}
```

Then we enable the timer interrupt by setting the correct bit in the NVIC (Nested Vectored Interrupt Controller). You can find the complete list of these on page 197 (section 9.1) of the STM32F4 Reference Manual and you can find more information on the ISERs (Interrupt SEt Register) in the Cortex-M4F manual. The `_IRQn` suffix is added in the define to mean “IRQ number”.

main-7.c:

```
#include "stm32f4xx_conf.h"

int main(void)
{
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // enable the clock to GPIOD
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; // enable TIM2 clock

    GPIOD->MODER = (1 << 26); // set pin 13 to be general purpose output

    NVIC->ISER[0] |= 1<< (TIM2_IRQn); // enable the TIM2 IRQ

    TIM2->PSC = 0x0; // no prescaler, timer counts up in sync with the peripheral
clock
    TIM2->DIER |= TIM_DIER_UIE; // enable update interrupt
    TIM2->ARR = 0x01; // count to 1 (autoreload value 1)
    TIM2->CR1 |= TIM_CR1_ARPE | TIM_CR1_CEN; // autoreload on, counter enabled
    TIM2->EGR = 1; // trigger update event to reload timer registers

    while (1);
}
```

And now we can finally write our interrupt service routine. The interrupt service routine is just a regular C function with a special name that corresponds to an IRQ. You can see the full list of function names in lib/startup_stm32f4xx.s, but I have reproduced them here to make it a little easier. In this case, we want to use `TIM2_IRQHandler`, so let’s add it:

main-8.c:

```
#include "stm32f4xx_conf.h"

void TIM2_IRQHandler(void) {
  // flash on update event
  if (TIM2->SR & TIM_SR_UIF) GPIOD->ODR ^= (1 << 13);

  TIM2->SR = 0x0; // reset the status register
}

```

```
int main(void)
{
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // enable the clock to GPIOD
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; // enable TIM2 clock

    GPIOD->MODER = (1 << 26); // set pin 13 to be general purpose output

    NVIC->ISER[0] |= 1<< (TIM2_IRQn); // enable the TIM2 IRQ

    TIM2->PSC = 0x0; // no prescaler, timer counts up in sync with the peripheral
clock
    TIM2->DIER |= TIM_DIER_UIE; // enable update interrupt
    TIM2->ARR = 0x01; // count to 1 (autoreload value 1)
    TIM2->CR1 |= TIM_CR1_ARPE | TIM_CR1_CEN; // autoreload on, counter enabled
    TIM2->EGR = 1; // trigger update event to reload timer registers

    while (1);
}
```

There are two things you usually need to do in an ISR; you need to check which event occurred and respond, as well as **reset the corresponding status register**. Seriously, you need to **RESET THE STATUS REGISTER!** The chip doesn't do this automatically and it will be a ridiculously annoying bug to track down.

Now, build your code and put it on the chip. The light will be blinking!

Stage 2C: Well, sorta

Even though the light might just look like it has been switched on, I guarantee you it's blinking. (if the light is off though, you've probably done something wrong!). The problem we have now is that it is blinking much too fast for your eyes to see. A quick check with an oscilloscope confirms it:

The light is blinking at 2.6MHz! That's over 45,000 times faster than your eyes can perceive!

So how do we slow it down ? That is an exercise for you, my dear reader. I will give you a hint though: try playing around with the prescaler and autoreload registers. If you understand what they control, it should be obvious what changes you need to make.

&

Happy programming, I will try to put up some information on how to use the on-chip DAC next!

&

I don’t normally put comment boxes on my articles, but perhaps a discussion could be useful in this instance.

47 Comments jeremyherbert 1 Login

Recommend 5 Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

prattmic • 5 years ago

Jeremy,

You said that you have not been able to get any linux flashing tools working with the STM32F4. I'm not sure if you have seen stlink for linux (https://github.com/texane/s..., but I have it working based on your template, with only a couple of tiny modifications. You can try it with my fork of your code, https://github.com/prattmic... . (I linked to an old commit because I am messing with not using STM's code at HEAD.) You should be able to add your stlink folder in the Makefile, then "make && make burn" to flash the Discovery. I have been using this tool since I got my board, and I actually haven't even used Windows at all.

Anyway, thanks for the great work on the templates!

5 • Reply • Share ›

Gordon ➚ prattmic • 5 years ago

I have tried this too, and the stlink library linked above works perfectly on linux!

1 • Reply • Share ›

dmedine • 5 years ago

Great Tutorial! Thank you so much.

One note: I am using the yagarto-4.7.2 toolchain to build on my OSX10.6 laptop. I immediately got a compiler error when I hit make for the first time -- something about 'crt0.o' not being found. I poked around and found that yagarto will look for this if it doesn't have a .s file to look at. It seems as if this was not a problem on earlier yagarto releases. I moved the line (about the 30th line or so) in your Makefile:

SRC += lib/startup_stm32f4xx.s

so that it was the second line of code and it worked like a dream. A note to those struggling with similar problems.

Thanks again for a great tutorial and an awesome template!

3 • Reply • Share ›