# Edu: Get your first LED blinking on STM32

Olle Norelius

January 2020

## 1 Introduction

This is a short education intended to get you started programming STM32 microcontrollers using the Infotiv Embedded Platform. You will learn what a toolchain is, and how to set it up. We'll download a template project, and edit some code to change how it works.

## 2 Prerequisites

To get started with this, I'm assuming you feel slightly comfortable using Linux and coding in C++. Nothing special really, but it helps to have used the terminal once or twice.

## 3 Hardware Setup

To start off, let's make sure that we have everything we need, and that it's all connected OK. For this tutorial, the connection is very simple.

### 3.1 Necessary components

- STLink-V2 programmer with USB cable
- STM32 "Blue Pill" board
- Female - Female DuPont connecting wires (5 should be enough)

## 3.2 Connections

To interface with the board, we use the 4-pin angled header mounted on one of the short sides of the board, see figure 1.

Note that we cannot use the USB connector mounted on the other side. Since the MCU (the black square in the middle of the board, the brain box) is not programmed, it cannot make sense of signals coming from the USB connector. The connector is hooked up though, and given the correct programming (known as a bootloader) it can actually be used to program the board, avoiding the STLink completely.

To begin connecting the board, connect the GND pin from the board to one of the GND pins on the STLink according to figures 1 and 2. Note that there are a whole bunch of pins called GND on the STLink, this generally means that they are internally connected, and it does not matter which one you choose.

Why are there so many GND pins? There are a couple of reasons, the main one here being signal integrity in ribbon cables. If one were to connect a standard 1.25 mm pitch ribbon cable to this connector, the connections would be made in order of pin numbering; that is to say Vapp, Vapp, TRST, GND, TDI, GND... The general pattern will end up as: Data, GND, Data, GND, repeating. Alternating signal and ground like this minimizes crosstalk between conductors if the wires are long and the speeds are high. For our application this does not matter much though.

Next connect the SWCLK (Single-wire clock) and SWDIO (Single-wire data I/O) pins between the STLink and the Blue pill according to figures 1 and 2.

Connecting the 3.3V pin, we're going to do something a little special. The reason for this is that the programmer supplies 3.3V on its $V_{DD}$ pin, but expects to see the voltage from the target at $V_{App}$. The idea is that this gives the developer an easy way to debug an annoying problem, that is if power is not actually reaching the processor correctly. For now, let's use the fact that there's two $V_{App}$ pins to our advantage, specifically the fact that these pins are internally connected in the programmer.

Connect pin 1 to pin 19, and then pin 2 to the 3.3 pin on the blue pill board. This will allow the $V_{App}$ pin to see 3.3 volts, and also distribute this voltage to the processor.

This should conclude the hardware setup, and we'll verify that everything is OK in the next step.

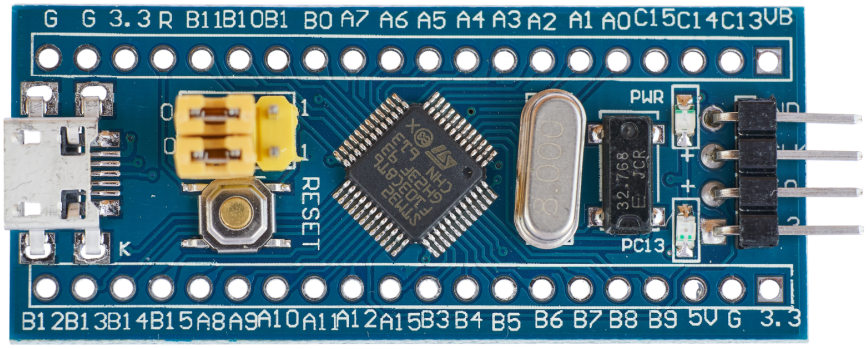


Figure 1: The "Blue Pill" board. The programming header is to the right, with the silkscreen partly obscured by the pins. Top to bottom, the pins are: GND, SWCLK, SWDIO, 3.3V.

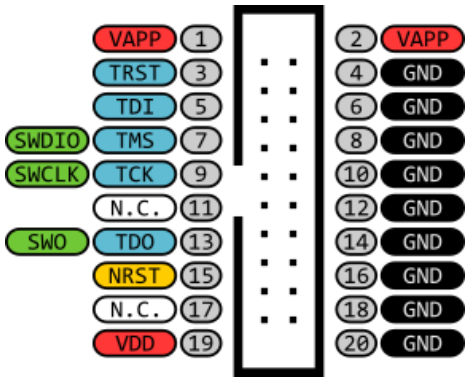


Figure 2: Pinout of the STLink V2. The pins we're using are the ones with green colored labels, they go to the matching pin found on the programming header (see fig 1). Vapp are voltage sensor pins, they're used to tell the programmer (both the device and the person) the board is powered up correctly.

# 4 Toolchain Setup

First of all, let's define what we mean by a toolchain. A toolchain is a set of programs that turns your code, understandable by humans, into something the processor can understand and execute. This is typically done by a set of programs, each fulfilling a role in transforming your code, analogous to a chain.

The toolchain normally used here at Infotiv is GCC + OpenOCD. This toolchain has the advantage of being fast, well documented and FOSS (Free Open Source Software). It is the standard toolchain for embedded development in a UNIX-based environment.

To install it on a linux-based host, run the following commands in a terminal:

```
sudo apt install gdb-multiarch openocd gcc-arm-none-eabi make python3-pip
cd /usr/bin
ln gdb-multiarch arm-none-eabi-gdb -s
```

The first command installs all the things needed to compile, debug and upload code to your target (in our case the blue pill board). The second and third create a symbolic link to GDB (our debugger), since the development environment we'll use expects it to be there.

For console warriors, this setup is all that needs to be done. I personally prefer a graphical environment, so let's set that up. Run the following command to install Visual Studio Code:

```
sudo snap install code
```

The reason I'm recommending VS Code is because it's reasonably lightweight (meaning it starts fast), free and mostly open source, and it has a nice extension called Cortex-Debug that we're going to use.

Go to the Extensions tab (indicated in red in figure 3) and search for 'cortex'. You should find the extension. Click 'install' to install it, and we are done with setting up the tools we need!

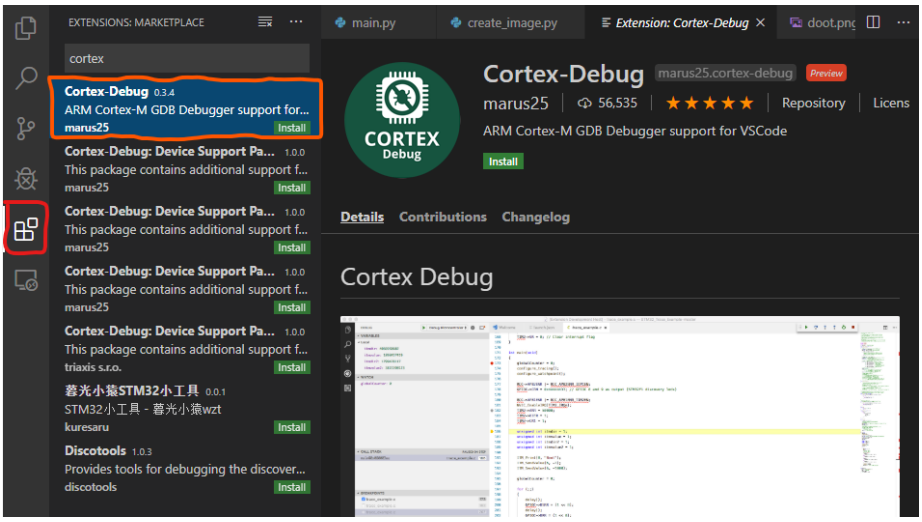


Figure 3: Caption

Before we move on, let's check that we've done everything right so far.

In a terminal, run the following command:
`openocd -f interface/stlink-v2.cfg -f target/stm32f1x.cfg`
The LED on the STLink should start flashing green-red, and you should see something like the following in the terminal:

```
    Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override use 'transpor
Info : The selected transport took over low-level target control. The results might differ c
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
none separate
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v34 API v2 SWIM v7 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 2.839503
Info : stm32f1x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

In this printout you can find relevant information if you cannot connect for some reason. The line containing `VID 0x0483 PID 0x3748` tells you that the

STLink has been found correctly as a USB device. If you don't get that, you might need to add yourself to the `dialout` group (google this), or double check that the cable is plugged in correctly.

Another key line is `Info : Target voltage: 2.839503`. This line tells you the measured voltage on the microcontrollers positive rail. If this is 0.000, you have forgot to connect power to the device (If this voltage seems low, yeah it is. I don't know why I got that when I ran this test, but it should be about 3.3V).

If all this looks alright, press Ctrl-C to quit OpenOCD. Congratulations, you've successfully completed the hardware setup!

# 5 Project Setup

Now to get started on the code, log into Gitlab (https://gitlab.infotivlab.se) and find the Empty project template (link).

When Git asks you for your login, use the first three letters of your first and last names, eg. Olle Norelius -¿ ollnor as your username, and your Infotiv email password as your password.

Now to get a copy of this repository that we can edit, we'll fork it. Click the Fork button near the top right of the page, and it will ask you to select a namespace for your fork. Choose yourself to make the forked project belong to your private projects. After a little while you should be taken to your new project! From here you can clone the project to your computer so you can edit it, but don't do it just yet.

When you first fork the project it'll be named Embedded Platform Empty Project, which isn't very descriptive. Let's give it a better name by going into Settings -> General in the left hand pane. At the top of this page you can change the name to something like "My first blinking LED". The URL to the project also contains "embedded-platform-empty-project", so lets change that by expanding the 'Advanced' header at the bottom of the page. When you expand it, scroll down to find 'Change Path'. Choose a simple path that you like, like 'blinky'. You don't need to worry about collisions with other user's projects, since this project is scoped to yourself.

Now you've got a project set up for yourself, time to clone it to your local machine so we can start working. Go to the main page of your project, and you'll find a button called 'Clone' at the top right. Click it, and copy the URL for HTTP access.

Now open a terminal and navigate to where you want to store the working directory of your project, something like ∼`/projects` for example. Next type `git clone` and paste the URL you got from Gitlab, this should give you a command similar to:
`git clone http://gitlab.infotivlab.se/ollnor/my-blinky-project.git`

When you press enter, git will fetch your project and put it in a folder named blinky (or whatever you called your project). The project also has some dependencies, called *subrepositories*, or *submodules*. These are initialized using this command:

```
git submodule update --init
```

Navigate into your project's folder and type `make gen_all`. This command builds the project- and board specific files needed for compiling your code for the microprocessor. If you see a bunch of text ending with the following, you've build the prerequisites successfully:

```
#
#
# Generate targets/targets.mk
#
#
#
```

Now type `make` and press enter, and it should build your executable. It doesn't do much at this point, but making sure that it builds is good before we start adding more sources of error.

Now open the project folder in VS Code by either starting it manually and selecting 'Open Folder' in the File menu, or just type `code .&` in the terminal you just used to build.

Side note: breaking down this command, you're starting `code` with the argument `.` (period), meaning current folder, and adding an ampersand at the end means it does not lock your terminal.

You should have the project visible in VS Code, and now we're ready to look into the code!

# 6 Code

Now you should see a folder structure open to the left in the VS Code window, navigate to src/main.cpp and open it. This is the main code file for our first

project, and is where you'll put your application code.

You'll see that there already is a little snippet of code in here; let's break down what's going on real quick.

The first few lines are include statements, used to pull in code needed to initialize the microcontroller properly. Specifically the first line, `#include "../config.h"` will pull in all of the things we created in the last step using `make gen_all`.

Line 7 initializes our `led_status` variable, used to keep track of if the LED is on or off.

We then get into our main()-function, a staple of all C++ programs. This function is special, and tells the compiler where the program should start executing. The first statement we encounter is:

```
logger.add("Start time: %d\n", get_system_time());
```

The logger object is a serial port output object which can be very helpful in debugging. Here we use its `add` method, which will simply print its arguments onto the serial bus. Other useful methods are `info`, `debug`, `warning`, and `error`, which will add helpful headers to the messages, so they can be sorted through easily.

The line basically prints a little message telling the time of the internal clock, which will always be zero at the start of the program. More on the logger object and the serial port in the exercises!

Moving on we find the main program loop, a while(1) also known as a 'while true loop'. This loop will never exit, and exists to allow us to repeat our program indefinitely.

Inside the loop we find the `ON_TIMER` macro which is a feature of the Infotiv Embedded Platform (It's not available on Arduino for example). It will execute its contents at most as often as the argument specifies; in our case every 500 ms, or two times a second. Note that this macro does not *ensure* that the code runs this often, it'll just not allow it to run *more* often than this. E.g if you have a function that takes one second to execute, this macro will only execute that often. If you have a function that takes one millisecond however, this macro will ensure that it runs only twice a second.

Inside this macro, we find the actual LED switching logic. The first line may look strange, but it simply performs a binary XOR (exclusive-or) on the `led_status` variable and 1.

XOR is a logic operation that returns 1 if one of the inputs are one, but not

Figure 4: The VS Code debug bar.

both. This means that if `led_status` is 1, we get 1 XOR 1 = 0. If `led_status` is 0, we get 0 XOR 1 = 1. This means we always get the opposite of the current `led_status` every time this operation is performed.

Side note: writing `led_status ^= 1;` is the same as writing `led_status = led_status ^ 1;` where `^` is the XOR operator.

Finally we set the LED output to `led_status`, which actually triggers the blinking.

# 7 Running the code

To actually run the code, press F5 in VS Code. This should trigger a terminal at the bottom of the screen with some text flashing by. When your program has finished compiling, a debug session will start in VS Code, which can be controlled by the small bar at the top of the screen (See figure 4).

At this point, the LED will not be blinking. This is because the program is waiting for you to start it, using the play button to the left in the debug bar. Click it, or press F5, and the LED should blink!

# 8 Exercises, pt. I

- Can you make the LED blink three times per second?
- Can you make the LED blink out an SOS in Morse code? Hints: SOS is `...---...`, and to make the microcontroller wait you can use the `delay(int milliseconds)` function.

# 9 How did it do that anyway?

We use a couple of premade objects here, the LED object and the logger object. These are of course not magic, they are created in `make gen_all` according to our specification from a file called `target_stm32f103board.xml` which is located in `./targets`. This file represents the physical setup of the microcontroller, and contains tags for all the things we want to use on the microcontroller. From the top we see tags defining choice of CPU, MCU and board, before a tag named tick-rate which sets the number of systick events that happen each second. Note that the tick-rate setting has no effect on the clock speed or performance of the microcontroller, it simply changes the time reported by `get_system_time` among a few other things.

After these introductory tags comes the first major tag: the RCC or Reset and Clock Control tag. We have an option for setting whether our board contains a crystal (which it does, it's the shiny silver thing), and some tags for enabling peripherals. We can ignore these for now, a lot of the stuff in this file deserves a second looking at.

We find the GPIO (General Purpose Input Output) tag next, in which we specify the direction and function of the pins on the microcontroller. We only need to set the function of the pins we want to use, the rest are left in a high impedance state so that they won't affect the circuit.

The alternate flag in the <pin> tag specifies whether to use the alternate function of that pin (See the STM32F103C8T6 datasheet page 29).

Further down the page we find the ¡blocks¿ tag, where we actually find the objects that we used in the preceding chapter. The line

```
<Logger name="logger" num_logs="1" log0_stream="UART_1"
log0_color="true" log0_level="ALL" />
```

declares a Logger object called `logger`, connected to `UART_1` (declared right above it).

The line

```
<STM32F4_GPIO_pin name="LED" device="GPIOC" pin="13" />
```

declares a GPIO pin called `LED`, connected to pin C13 (GPIO*C*, and pin=13).

These lines declare objects that are then defined and instantiated in `targets/stm32f103board/board.cpp`. Peeking at that file, we can actually see our objects get created at line 192:

```
    /* Logger */
logger_t loggers[] =
{
        {
                .log_level = Log::ALL,
                .log_color = true,
                .stream = &UART_1
        }
};

Log logger(loggers, sizeof(loggers)/sizeof(logger_t));
/* Setup STM32F4_GPIO_pin LED */

STM32F4_GPIO_pin::parameters const LED_params =
{
        .pin = 13
};

STM32F4_GPIO_pin LED((void*)GPIOC, &LED_params);
```

Note that you never need to touch this file, and any changes you make here will get overwritten as soon as you build the project anyway.

# 10 The Logger Object

We've glossed over the logger object up until now because it requires additional hardware to interface with, but it can actually be a very useful piece of code. To interface with it we'll need to use a USB to UART transceiver; a device that turns a USB port into a serial port, see figure 5. To use this, you'll need two wires: one to connect ground to ground, and one to carry the signal sent out from the microcontroller.

Connection is as follows: GND on the USB device to GND on the microcontroller, RX/RXD on the device to pin A9 on the microcontroller, and optionally TX/TXD to pin A10 on the microcontroller. The TX/TXD connection carries data from your computer to the microcontroller, and isn't needed for simple logging. Note that there is an error in the labeling of some USB devices, see figure 5.

Once these are connected, plug the USB device into your computer and run the following:

```
$ miniterm /dev/ttyUSB0 115200
```

You should be greeted by a simple prompt. If you see an error telling you that you have insufficient privileges, add `sudo` in front of the command. If you now press the reset button on the microcontroller, you should see the message `[CRITICAL] Start time:  1`.

This is the message set on line 11 in main.cpp. The label on the message ([CRITICAL]) comes from the choice of logging method, logger.critical, and the message is what's entered.

As an exercise, try to print a debug message any time the LED turns on, but not when it turns off! Debug messages are printed using `logger.debug()`.

# 11 Conclusion

We've covered setting up the STM32 Blue pill development platform, and programmed it using the GCC open toolchain. We also downloaded a template project, and edited it to add some functionality. Continuing on from here, there are a couple of paths to take: looking at software, we could learn to read inputs both analog (voltages, positions) and digital (buttons, switches). There is also much to be gained from learning some electronics to allow you to create LED displays and motor control circuits, both of which can be nicely controlled with code similar to what's been used here.

Lastly, we touched slightly on digital communications with UART debug messages. The same idea is used for many different kinds of communications, but those are out of scope for this document.

# References

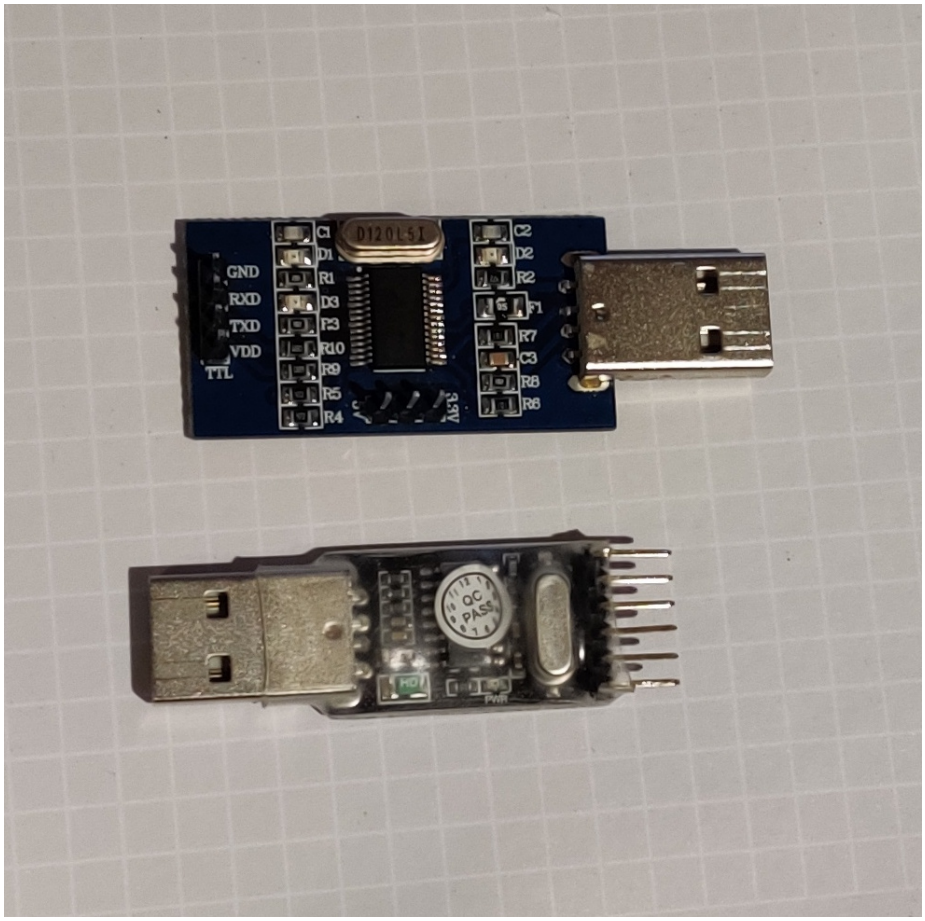


Figure 5: Two USB-UART transceivers. Note: the bottom one has its TX and RX pins reversed on its labeling.