First Code

We've built the simplest possible board based on the **Maple Mini**, now it's time to verify that it works, and that we have an operational **toolchain** to go from an idea to actual software running on actual hardware.

To that end, we're going to create a software project named **MapleMini_Blink**, which purpose will be to blink the **LED** on the Maple Mini. It doesn't need to be more complicated than that. Especially since getting there will take quite a few steps.

1.1. STM32CubeIDE

I've already mentioned it in the page on programing. If you've missed it, well, that's why I put a navigation menu on my website.

If you haven't, go ahead and install **STM32CubeIDE** on your PC. This can take a while, so I'm going to blabber a little to prevent you from falling asleep. Feel free to skip to the next section, I won't mind.

I've mentioned that there are several viable options for developing software for **STM32** boards. The most powerful, as you'd expect, are expensive. They are also geared towards businesses who make products you can buy, and who have different priorities than you or I have. For example, if your business relies on being able to develop code, you will want actual technical support (which **StackOverflow** *isn't*). And real technical support costs money.

The free **STM32CubeIDE** (which I call "Cube" for short) strikes a balance between cost (zero Euro) and capability. It's a lot like an actual professional tool you'd pay money for, and it's nothing like that simplistic Visual Basic joke the Arduino community calls an "IDE". Cube also has native support for all of ST's **Nucleo** boards, and of course for all STM32 microcontrollers.

Moreover, it comes with **low-level libraries** and a **HAL** (Hardware Abstraction Layer) which greatly accelerate the development of prototypes. And which we'll be putting to use soon.

Last but not least, it's the only STM32 IDE I know of that has a **graphical tool** for configuring your microcontroller and its peripherals. This tool generates all the initialization code for your STM32, which saves you a lot of time and errors. Even

Search & hit enter...

Navigation

open all | close all

- Legal
- Contact Nefastor
- Computers
- Microcontrollers
  - ESP32
  - STM32
    - Libraries
    - STM32F1
      - Maple Mini
        - The Electr
        - Programm
        - Basic Prot
        - First Code
        - OLED Prot
        - Adding Int
        - I²C Progra
      - STM32F3
    - STM32H7
    - USB
    - Getting Started
- Robotics

Meta

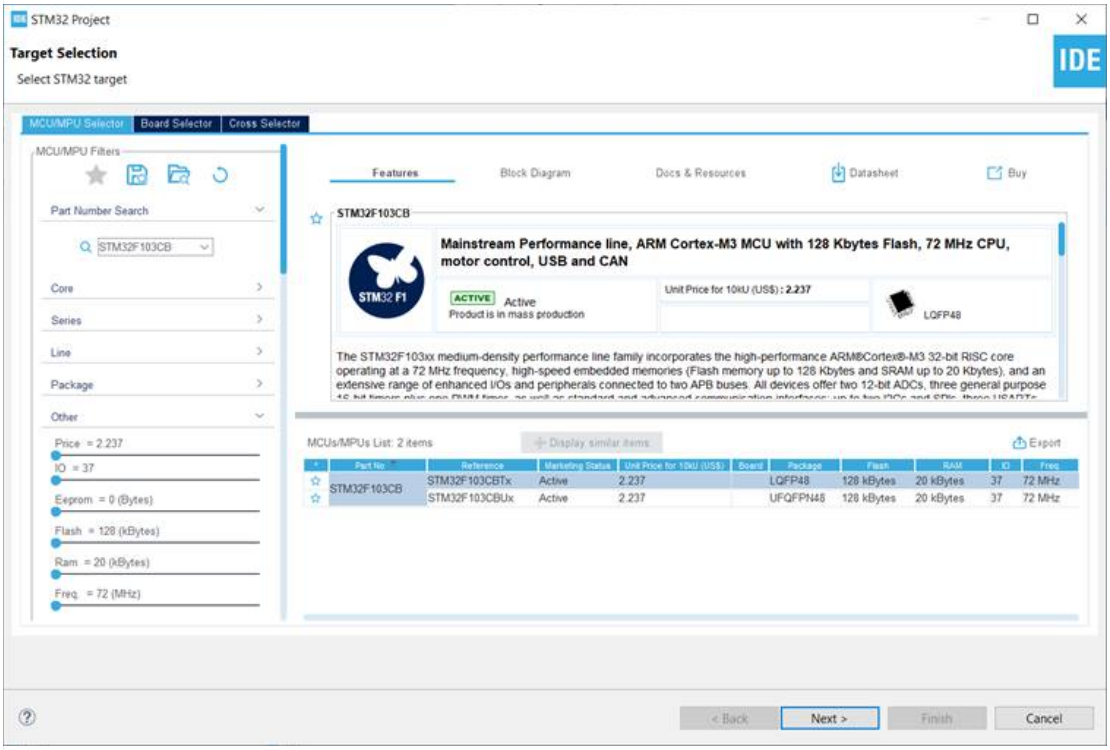
better, you can modify your configuration and regenerate that code at any time.

For all those reasons, Cube is my default choice when working with STM32.

OK, enough talk, launch Cube.

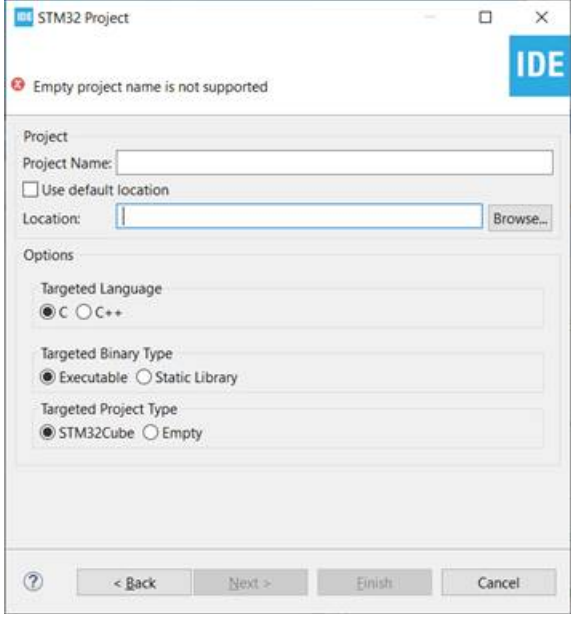
1.2. Create a Project

Cube is based on Eclipse. As a result, its menus may be needlessly complicated. Don't be scared, there's a lot of stuff we won't be using. To create a new project, open the **File** menu, **New** submenu, and select **STM32 Project**. This will open the "STM32 Target Selector" where you can pick the device you want to program. We know the Maple Mini uses the **STM32F103CBT6** so we can just type that in the search box :

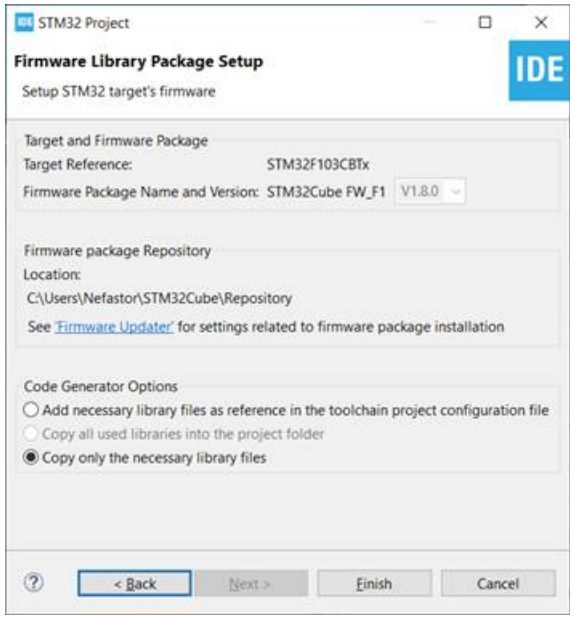


If you are targeting any board made by ST you should go to the "Board Selector" tab instead of selecting the microcontroller itself. Doing so will let Cube pre-configure your project for the hardware that's present on those boards. But since our board is custom, we need to pick its microcontroller and do that work ourselves. Select the CBT variant (LQFP48) and hit **Next**.

The dialog box will now ask you the usual stuff : what's the name of your project and where do you want to store it.



Note : at the time of writing, C++ **isn't** actually supported. Choosing this option will just change the extension of some source files, and which compiler is invoked when building, but that's all. C++ code *will* compile, but Cube will still only generate C code, and managing the cohabitation of C and C++ code in the same project just isn't worth the effort. At least not for rapid prototyping. I suggest sticking to the default options. Name your project **MapleMini_Blink** and hit **Next** again :



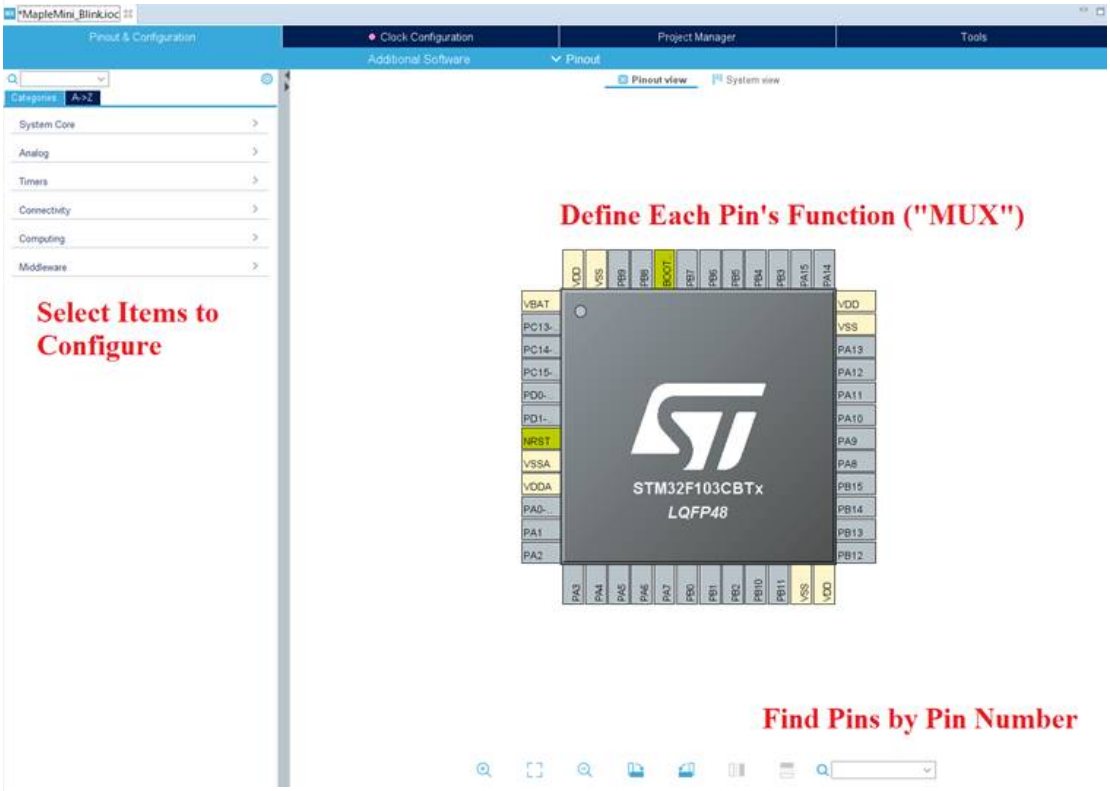
On the final dialog, you get to choose which version of ST's libraries you will use on this project. I've never seen a reason to use anything but the latest. In my screenshot the drop-list is greyed out because I only have one version of the STM32F1 libraries installed.

Now hit **Finish**, so we can start having fun.

1.3. Device Pinout Configuration

The first thing that opens is a device configuration tab. This used to be a standalone program called **STM32CubeMX** but it is now integrated into the IDE for convenience.

You start on the pinout tab, where you can do three things I've written in red on this screenshot :



The goal of the game is to fill in the picture as much as possible to describe your board, since Cube has no idea what you've been soldering. The Maple Mini is a very simple device, so this will go quickly :

Expand **System Core** and select **RCC**. This is the block of the STM32 which covers system clocking. Most STM32 have four oscillators, two internal, two external :

- **HSE** : High Speed External. The Maple Mini connects an 8 MHz crystal to it.
- **HSI** : High Speed Internal. An RC oscillator for low-cost, low-accuracy applications.
- **LSE** : Low Speed External. The Maple Mini doesn't use it.
- **LSI** : Low Speed Internal. Another internal RC oscillator.

STM32 clocking is a vast topic, beyond the scope of this page. For now, simply select "**Crystal / Ceramic Resonator**" for the **HSE** and leave the LSE disabled. Notice how the picture of the microcontroller changes, pins turning green as they are automatically assigned the function that matches your settings.

Still under **System Core**, select **SYS**. This block lets you set the type of debug interface your board implements. In our case that's **SWD**, so you must select "**Serial Wire**".

So far, so good. Since the oscillator and debug pins are unique, Cube could set them up for you without risk of ambiguity. Now let's take care of the LED and the user button. Those are GPIO (general purpose I/O) pins and Cube can't know which ones

we use. But we do, thanks to the Maple Mini's pin map. The LED is connected to **PB1** and the button to **PB8**.

If you can't find those pins on the drawing (which *can* be tedious on larger MCU's with hundreds of pins) use the search box at the bottom of the configurator. Once you found the pin you want, left-click on it. This will open a list of all possible functions for that pin. You can also right-click on a pin to rename it.

Set **PB1** as a "**GPIO_Output**" and rename it **LED**.

Set **PB8** as a "**GPIO_Input**" and rename it **BUTTON**.

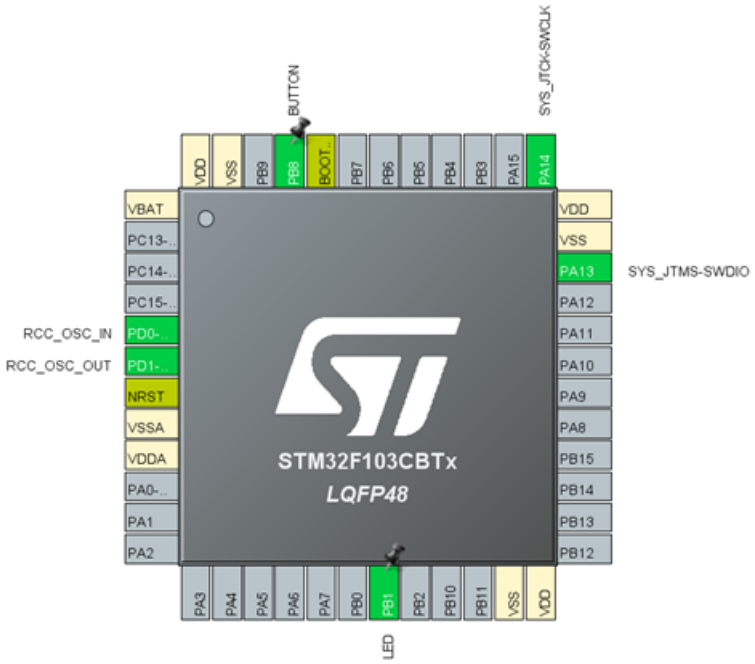
The names aren't just cosmetic : when Cube generates your project's initialization code, all the constants for your pins (the port and pin numbers) will have those names in them. It's all about making your life easier.

Now go back to the **System Core** section and select **GPIO**. You should see a list with the two pins you've configured for the LED and button. The GPIO block lets you setup additional features for each pin, most importantly their drive mode (push-pull or open-drain), initial state and whether or not their internal pull-up or pull-down is enabled.

The **LED** on the Maple Mini is connected between PB1 and GND : it needs to be driven by a **push-pull** output.

The **button** on the Maple Mini is a normally-open type with a 10 K pull-down. When the button is pressed, it connects PB8 to VCC through a 1 K resistor. Therefore, there's no need to use the internal pulling resistors and we also know that the pin will read as zero by default, and as one when pressed.

At this point, the chip in the device configurator should look like this :



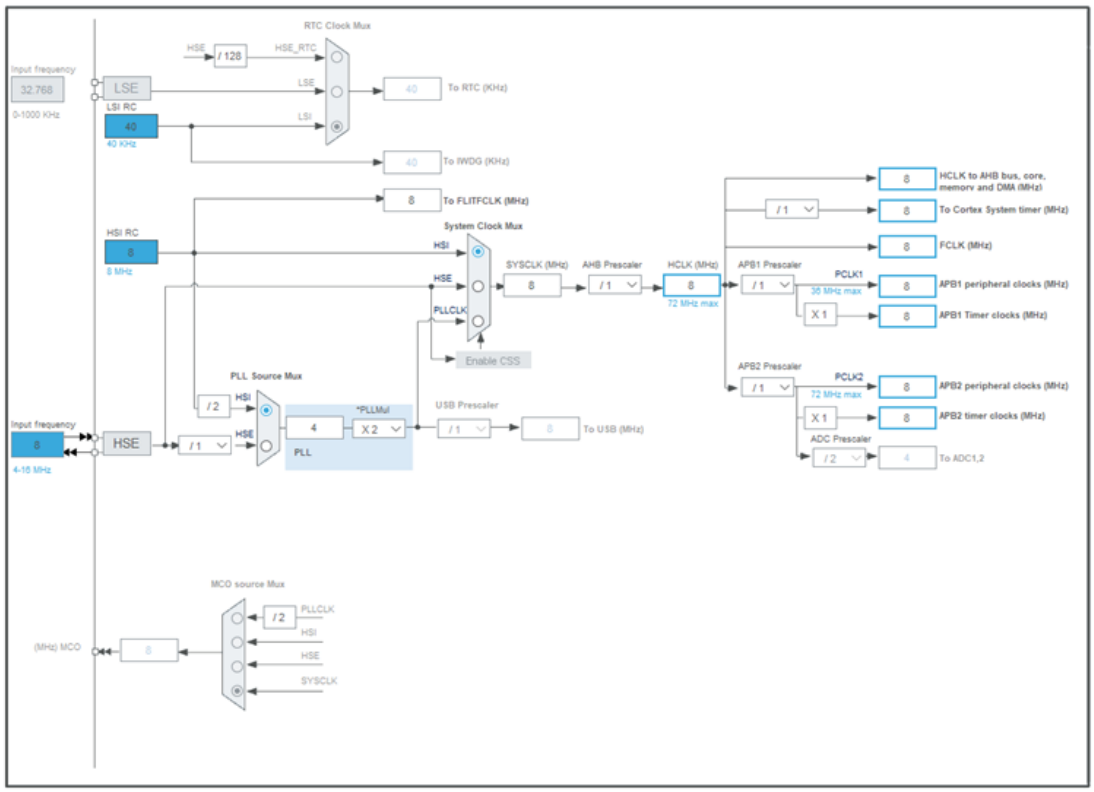
Pretty self-explanatory if you ask me. But then again, I'm really good at this stuff.

Keep in mind that you don't have to setup all the pins used on your board from the get go : you can come back at any time, modify your configuration and regenerate the initialization code as many times as you want. No pressure. Just make sure you don't configure an input as an output and blow up a component on your board.

Anyway, it's now time to setup the microcontroller's clocking. Hit the **Clock Configuration** tab at the top of the device configurator and try to stay calm.

1.4. Device Clocking Configuration

You will be greeted by this arcane schematic :



I ain't gonna lie, that's not very intuitive. And bear in mind that this is pretty much **the simplest STM32 microcontroller**. The largest devices (some of which have multiple cores) look like oil refineries. Thankfully, we're only trying to blink an LED for now.

This schematic is interactive. Near the **HSE** block, you need to enter the frequency of the Maple Mini's crystal in the **Input Frequency** box. That's 8 MHz.

From this value, Cube will compute the frequency each clock domain will receive every time you change a setting. Those frequencies are regrouped on the right side of the diagram. Notice that the ARM Cortex-M3 core will receive only 8 MHz. This is significantly slower than the 72 MHz advertised for the STM32F103... but it would be more than enough for blinking an LED.

Still, I hate leaving performance on the table even if I don't need it. And this is a good

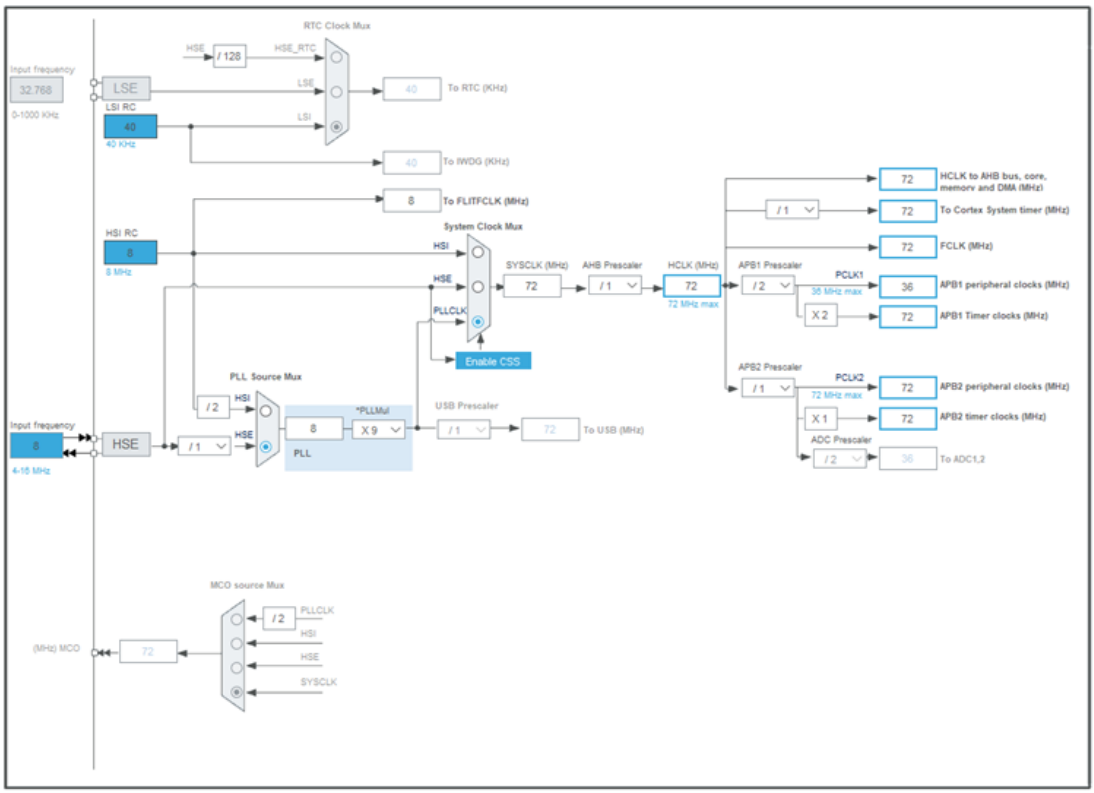
opportunity to introduce the topic of clock configuration.

In *theory* you could just enter 72 in the core frequency box and let Cube try to solve how to setup the clock circuitry. In practice, this usually leads to imperfect results or even outright failure :

The best approach is to fiddle with all the settings until you get what you need. A bit of logic helps a lot. For example :

- HSE is 8 MHz, so is HSI, therefore the only way to get 72 MHz is to use the **PLL**. Feed the HSE into the PLL and then select **PLLCLK** as the system clock (SYSCLK)
- Doing so will turn the “Enable CSS” button blue. Ignore it. It’s a safety feature irrelevant to most users. Enabling it does no harm, though.
- Core clock is now set to 16 MHz. To get it to 72 we need to change the PLL’s multiplier from 2 to 9. Yes, learning those multiplication tables when you were a kid is **finally** paying off ! Who woulda thunk !
- Core clock is now 72 MHz, *but* a couple of blocks have turned red : we’ve exceeded their maximum operating frequency. Set the **APB1 Prescaler** to /2 instead of /1 to fix that.

The schematic should now look like this :



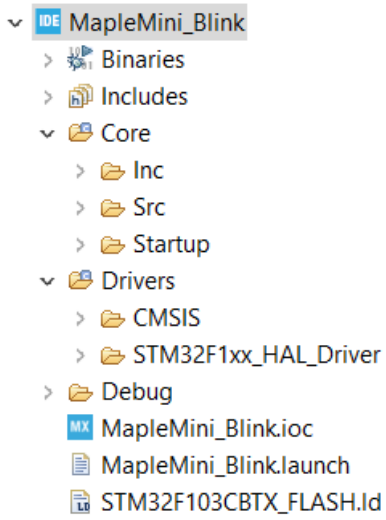
Cube checks for configuration errors in real time. Even if your clock settings are valid, they might still result in frequencies that are too high or too low for some of the peripherals you've enabled. Go back to the **Pinout & Configuration** tab and look for red error marks.

In this case you shouldn't see any. Time to generate the code !

In the **Project** menu, hit **Generate Code** and let ~~mayhem begin~~ the magic happen.

1.5. Add Your Application Code

Once code generation has completed, a lot of folders and files will appear in your project's hierarchy :



"**Core**" is where the initialization and application code live.

"**Drivers**" contains standard ARM libraries (**CMSIS**) and the ST HAL (**Hardware Abstraction Layer**) libraries.

"**Debug**" is the output folder where every file generated during the build process will end-up. That includes the executable binary that will be flashed into the STM32.

The **.ioc** file is your device configuration : open it to return to the device configurator.

At this point, the project will actually compile, flash and run... but of course it would do nothing. All it is at this point is a truckload of (unused) libraries, startup code that will setup your chip as you've specified, and a main function with an empty infinite loop.

Let's change that.

To blink an LED we only need to learn how to do two things besides C syntax :

- **Assign** a state to an output pin
- **Wait** a while for our human senses to register that state

Used to be, learning how to assign a pin's state would require learning register addresses and bit organization in those registers. As for waiting, we'd have had to write a delay function based on a loop or a timer interrupt, which would have led to timing calculations. This is 2020, nobody has time for all that anymore. That's why we have the **HAL**.

First, let's open the source file we'll be working on : under **Core\Src**, double-click "**main.c**". Scroll down until you find the **main** function, and keep scrolling until you reach the infinite loop, in the form of a "while (1)" statement.

1.6. GPIO Operations

Turning the LED on and off means assigning a state to a GPIO pin.

Each peripheral within the STM32 has its own HAL header file. Therefore, we need to find the HAL GPIO header file. It'll be under **Drivers\STM32F1xx_HAL_Driver\Inc**. Each file in that folder pertains to a specific peripheral. Look for the one that has GPIO in its name : "**stm32f1xx_hal_gpio.h**". Open it. I hope you speak C.

This header file will tell you a few interesting things :

- There's an enumeration named **GPIO_PinState** that's got two entries : SET and RESET, with RESET equal to zero. Doesn't take a genius to know what it's used for. That's our pin state data type.
- There are prototypes for all sorts of I/O functions, one of which is called **HAL_GPIO_WritePin**. Again, doesn't take genius to know what it does.

The ST HAL naming scheme is very intuitive. It's clear that the **HAL_GPIO_WritePin** function requires a port number, a pin number and a pin state. We've already found the data type for the pin state.

The HAL header also contain generic macros for the port and pin numbers, and we could use those, but remember we *did* name the LED pin. Cube generated macros based on that name. You will find them in "**main.h**", under **Core\Inc** :

```
1.  #define LED_Pin GPIO_PIN_1
2.  #define LED_GPIO_Port GPIOB
```

We now have all the information necessary to call the HAL and turn the LED on and off.

1.7. Delay Function

Finding the HAL's inevitable delay function is a bit less obvious, since it isn't a hardware peripheral.

There is a HAL header file which name doesn't specify a peripheral : "**stm32f1xx_hal.h**". Inside it, you'll find the prototype for a function named **HAL_Delay**. I'll give you one guess as to what it does.

OK, maybe that's a bit mean of me... sure, there's a prototype but there's no comment to tell us if the delay it creates has to be specified in microseconds, butterfly lifespans or eons. This will happen. Don't be afraid to Google STM32 HAL function names, there's a lot of documentation online.

Let me save you the trouble this time : **HAL_Delay** takes a value in **milliseconds**.

And now we know how to make sure the LED blinks slow enough that our limited human eyes will be able to notice.

Let's put everything together.

1.8. Paint Inside the Lines

Locate the empty "while" loop in "**main.c**" and add code to blink the LED.

This can be done in many different ways, for example :

```
1.     /* Infinite loop */
2.     /* USER CODE BEGIN WHILE */
3.     GPIO_PinState state = 0;
4.     while (1)
5.     {
6.       /* USER CODE END WHILE */
7.
8.       /* USER CODE BEGIN 3 */
9.       HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin,
   state);
10.      state = 1 - state;
11.      HAL_Delay (250);
12.    }
13.    /* USER CODE END 3 */
```

The exact way you choose to blink that LED isn't important, what you really need to notice in this code snippet is that I painted inside the lines.

The code generated by Cube is sprinkled with "USER CODE" comments to tell you where *your* code should start and where it should end. You need to respect that. And whatever you do, you need to keep those comments. **Real Bad Thingsâ„¢** will happen if you change or delete one. So don't.

Those comments are just comments. If you write code outside of the areas delimited by "USER CODE" comments, it'll still compile and work. However, if you ever **regenerate** the project then anything you've written outside those comments **will be lost**. And there's no Control-Z-ing this.

Conversely, if you write all your application code between the "USER CODE" comments, Cube will preserve it every time you regenerate.

Needless to say, this **only** applies to source files created by Cube. Cube won't mess with any source file you create on your own.

1.9. Mind the Default Firmware

It's taken a while, but here we are : with a software project designed to configure the STM32F103 and then proceed to blink an LED. All that's left to do is **compile** and **flash** it into the Maple Mini.

That's easy enough : there's a nice "run" button in Cube's tool bar that will take care of everything. It looks like this :

Before you hit it, make sure that your board is powered (for example by plugging a USB charger into the Maple Mini's USB socket. A computer will also work). Make sure the SWD probe is connected to both your PC and your board.

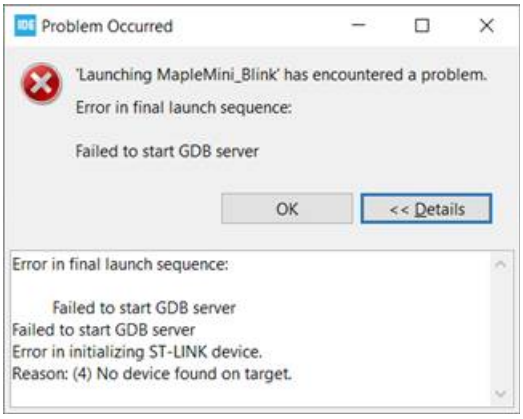
It's possible that your probe's firmware may be outdated, in which case Cube won't be able to flash your MCU. Go to Cube's **Help** menu, select **ST-LINK Upgrade** and follow the instructions. It's also a good way to check that your probe is connected to your PC, if there's any doubt.

Now you can hit that "run" button.

As with any other IDE you will see the compilation messages flow down in Cube's console pane (under your code editor). If you've been meticulous and did everything I've told you to do, there will be no error and no warning, and Cube will proceed to flash your Maple Mini.

And *that's* when you might hit a **snag**.

The first time you try to flash a new Maple Mini module, there's every chance it'll fail. You will see this error message :

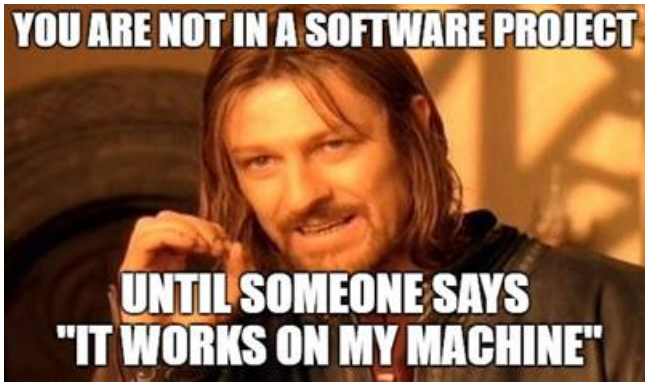


It is very likely that any Maple Mini you get your hands on will come with a **pre-programmed STM32** microcontroller. It appears our friendly Chinese factory workers are kind enough to burn a **USB DFU bootloader** into every Maple Mini they

produce, at no extra cost to you and me, I'm sure.

That bootloader will prevent you from using **SWD** and so it needs to disappear. Try flashing again but this time **hold the module's RESET button pressed** while you start flashing. This will "bypass" the bootloader. It will also hold flashing on pause after the message "Starting server with the following options:" appears in Cube's console. At that point, **release the reset button** and flashing will proceed. Good news : you only need to do this **once**. After your program has overwritten the DFU bootloader, SWD will operate normally and won't require a reset ever again.

And that's it... if all went well, you should be looking at a blue LED blinking twice per second. If not, well it's not my fault. It *always* works when *I* do it.



Next, we'll attach a generic **OLED display** to one of the Maple Mini's SPI buses, thereby improving our board's usability by a factor I couldn't quantify, but that marketing evaluates it somewhere between "ginormous" and "xxxtreme".