

ECSE 444: Microprocessors

Lab 0: Blinky! Blinky!

Abstract

In this lab you will confirm that you have installed the STM32Cube toolchain, can successfully configure your B-L475E-IOT01A1 development board, and deploy simple software to it, in this case, making an LED blink.

Deliverables

None.

Grading

Not graded.

Changelog

- 14-Sep-2020 Updates to overview: hardware requirements and ST-LINK-Server.
- 11-Jun-2020 Initial revision.

Overview

In this first lab, we will build the canonical “hello world” program of development boards: we’ll make an LED blink on our STMicroelectronics [B-L475E-IOT01A1 Discovery kit for IoT](#) development board.

First things first: there are several elements of the tool chain that must be downloaded and installed. This year, we’re using the STMicroelectronics tool chain, STM32Cube, which works in *Windows, Linux, and OS X*. For more information about this software ecosystem, go [here](#).

You’ll need the following following hardware before beginning:

- STM [B-L475E-IOT01A1](#) development board.
- Micro USB data+charging cable; note, charging-only cables are not adequate.

You’ll need to install the following software before beginning:

- [STM32CubeMX](#)
- [ST-LINK-Server](#)
- [STM32CubeIDE](#)

Installation Notes

- You may need to update your Java JDK before you can run STM32CubeMX.
- You need to install [ST-LINK-Server](#) before STM32CubeIDE will be able to communicate with the development board.
- STM32CubeIDE will also prompt you to update the firmware of your B-L475E-IOT01A1.

For more information on installation, follow the link for your operating system.

- [Linux](#)
- [OS X](#)
- [Windows](#)

All of our projects will start with STM32CubeMX. The purpose of this software is to help us configure our development board: (a) we select our development board, (b) we configure our peripherals, and then the magic happens: (c) the software generates a code template, or skeleton, for us.

We'll then edit and run this code in STM32CubeIDE. This software is an Eclipse-based compilation, deployment, and debugging environment. We first make changes to our software, editing the skeleton previously generated for us, adding new source files, etc. Then we can deploy the software to our development board and use a debugger to slowly step through the program, check the values of variables, etc. Finally, we can run the software at full speed on our development board once we are satisfied that it is operating correctly.

Our development board comes with a wide variety of peripherals; in this lab, we'll use a *timer* to make an *LED* blink!

Because this is the first lab, it is structured more like a tutorial: we will walk through the steps required for configuration and coding, step by step. In the future, you will be expected to navigate the software more autonomously, with the assistance of the vast resources available to support professionals as they develop software for this platform.

The remainder of this document is organized as follows:

1. Code Generation with STM32CubeMX
2. Writing, Deploying, Debugging, and Running Your Code in STM32CubeIDE
3. Additional Exercises

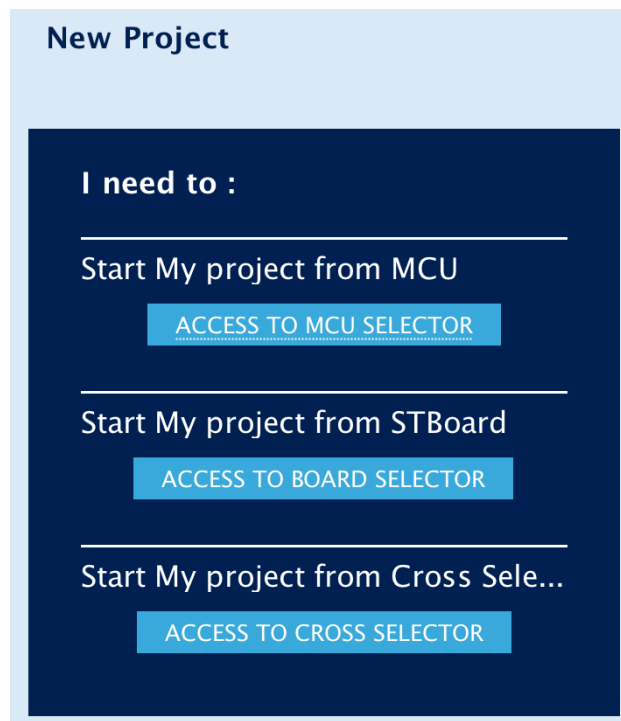
Code Generation with STM32CubeMX

First, we'll use STM32CubeMX to generate the skeleton code that will initialize our development board and its peripherals.

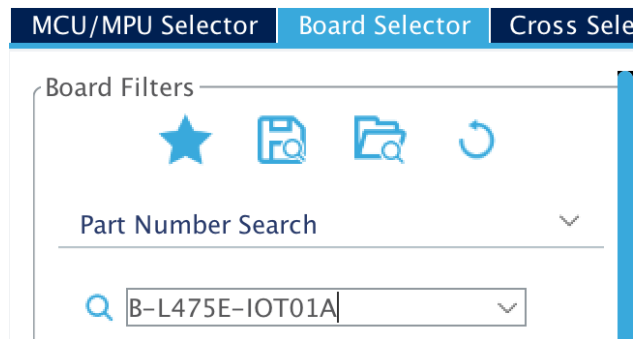


Selecting the Discovery kit for IoT Development Board

Open the software and begin a new project using the “board selector.”



Search for our board, the B-L475E-IOT01A1.



The screenshot shows a web interface with three tabs: "MCU/MPU Selector", "Board Selector", and "Cross Sele". The "Board Selector" tab is active. Below the tabs is a "Board Filters" section containing icons for a star, a document with a magnifying glass, a folder with a magnifying glass, and a refresh icon. Below these icons is a "Part Number Search" section with a search icon and a text input field containing "B-L475E-IOT01A".

To make things easier in the future, mark it as a favorite by clicking the star.

Boards List: 1 item

*	Overview	Part No 
		B-L475E-IOT01A

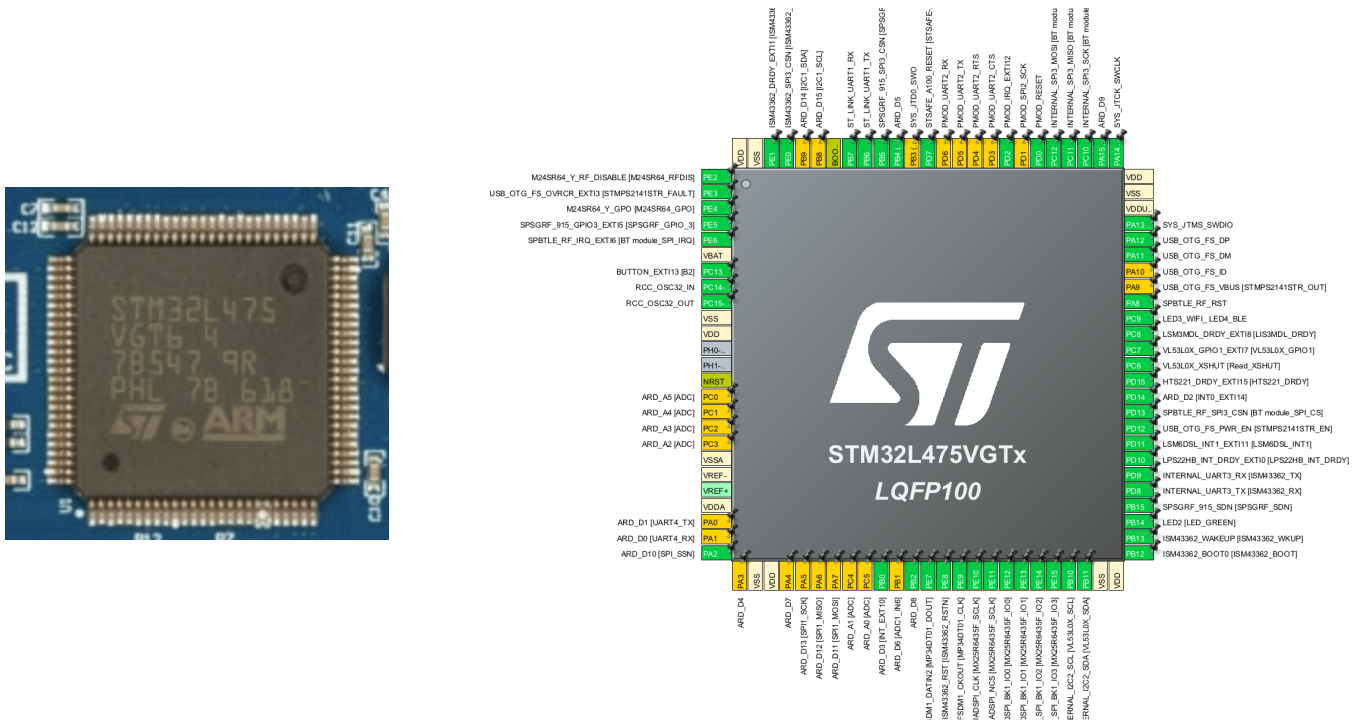
Then “start project.”



You will be prompted: “Initialize all peripherals with their default Mode ?” Choose “yes.”

Configuring Peripherals

Our development board has a lot of peripherals. The view you're greeted by when you start a project is showing you what each pin of the processor package is connected to.



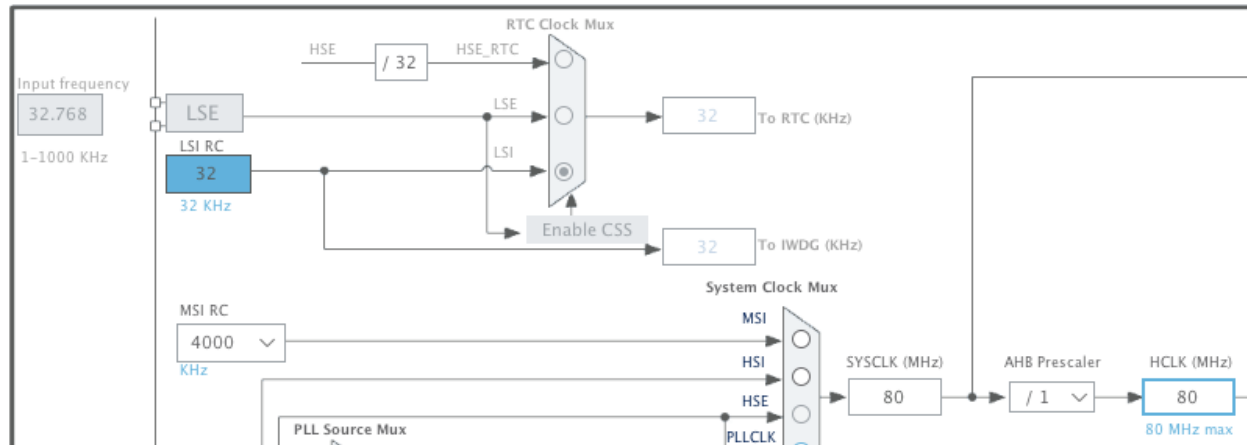
Each configured pin (amongst other things) means more code to generate, making our first project harder to understand; for the sake of making things easier to understand, we'll clear the configuration and only set up those two peripherals we need for this lab.

Configuring the Clock

First, however, we should confirm that the system clock is set correctly. Unless stated otherwise, the system clock for your processor should always be 80 MHz, the maximum operating frequency available. Select the *Clock Configuration* tab. There are a number of clock sources available, and a lot of options here for configuring not only which clock drives the core, but which clocks drive peripherals. Choices here matter: timers count clock edges; whether these edges occur at 80 MHz or not has consequences for system operation.

Pinout & Configuration

Clock Configuration

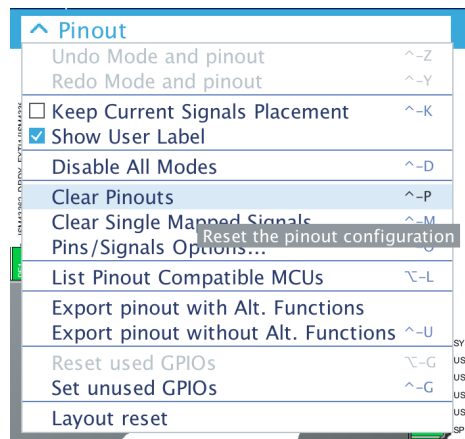


HCLK is key for now. If it isn't set to 80 (MHz), do so. The tool will find the appropriate configuration of multiplexors and clock sources to reach the target.

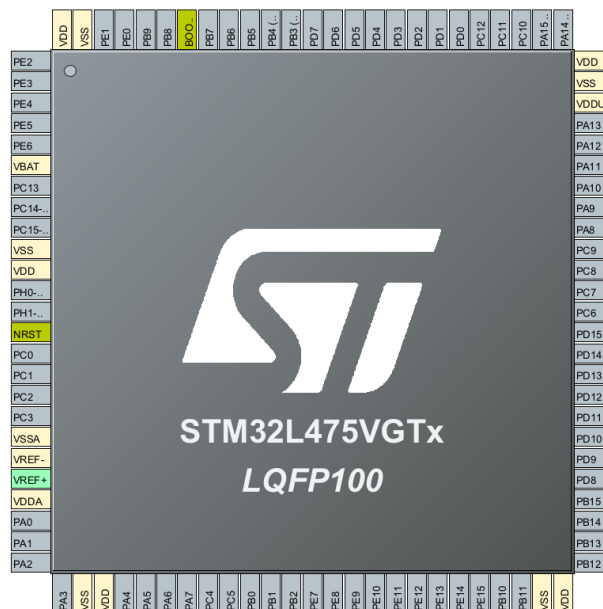
Configuring LED2

Now return to the *Pinout & Configuration* tab. We're going to clear the pinout and set up individual pins, but before we do: find PB14, and observe it's label LED2 [LED_GREEN]. You can find this by simply looking around the edge of the chip, or by searching for "LED" in the search box below the illustration of the pinout. The PB14 pin is by default configured to drive an LED; after clearing the pinout, resetting all pins, including this one, we'll reconfigure it to drive the LED again.

Select the *Pinout* pulldown menu, and “clear pinouts.” This will reset the pins, remove the labels, and eliminate all associated skeleton code.



After confirmation, your pinout should look like this, and be ready for us to begin configuring.



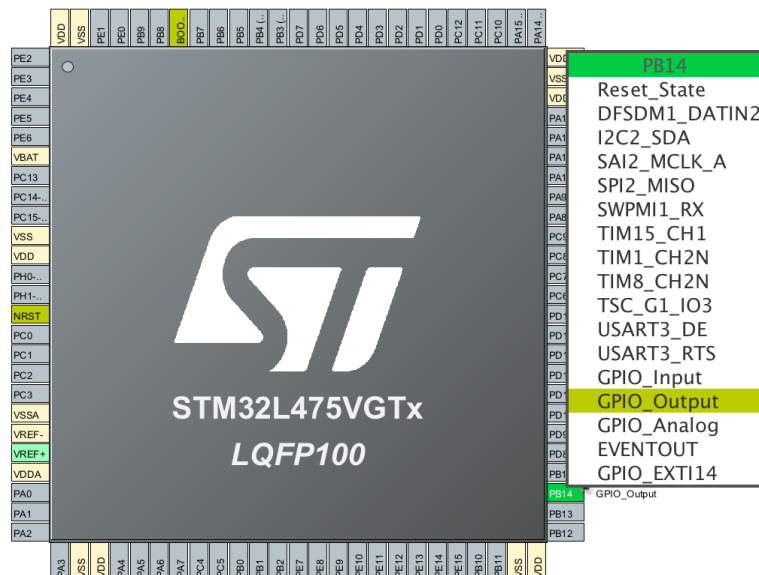
Find PB14 again. How do we know this is connected to the LED? (a) It was before, and (b) reference material for the development board tells us so.

For instance, the [user's manual](#) details the available peripherals, and their interface(s):

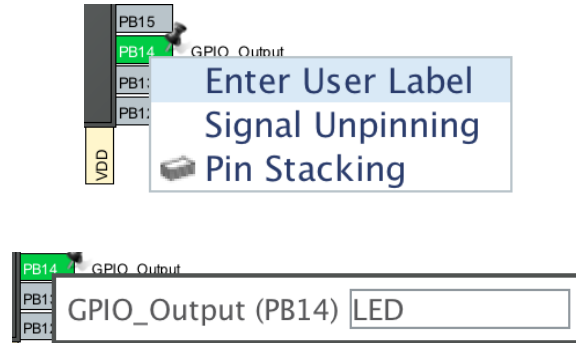
Table 2. Button and LED control port

Reference	Color	Name	Comment
B1	black	Reset	-
B2	blue	Wake-up	Alternate function Wake-up
LD1	green	LED1	PA5 (alternate with ARD.D13)
LD2	green	LED2	PB14
LD3	yellow	LED3 (Wi-Fi)	PC9, Wi-Fi activity
LD4	blue	LED4 (BLE)	PC9, Bluetooth activity
LD5	green	5V Power	5 V available
LD6	Bicolor (red and green)	ST-LINK COM	green when communication
LD7	red	Fault Power	Current upper than 750 mA
LD8	red	V _{BUS} OCRCR	PE3
LD9	green	V _{BUS} OK	5 V USB available

In order to connect PB14 to LED2, we need to set the mode of this pin to *GPIO_Output*.
Left-click on the pin, and select this option.



It's useful at this point to additionally label the pin. Labels make it easier to find things in STM32CubeMX (using the search feature below the pinout), and can also be used in STM32CubeIDE when we're writing code. **Right-click** on PB14 and "Enter User Label." Label the pin as "LED" and we'll use this to access it when we write our code.



This is all that is necessary to set up our LED!

Configuring TIM2

Next, we'll configure the timer that we'll use to make our light blink. We'll see over the course of the semester how this works, in detail. The short version: timers count clock edges, and take configurable action when the count reaches certain values. Our timer will count up to 1 second, triggering an interrupt, which we will use to toggle the LED.

The timers are internal to the processor: i.e., they aren't accessible through pins. To get started with configuring *TIM2*, we first select it from the timers. Expand "Timers" on the left hand side, and select "TIM2," as illustrated below. This will expand *TIM2 Mode and Configuration*.

- In *Mode*, select "Internal Clock" from Clock Source. This means the timer will count processor clock edges at 80 MHz.
- In *Configuration*, under *Parameter Settings*, set *Prescaler* to 40000, and *Counter Period* to 2000.
- In *Configuration*, under *NVIC Settings*, tick "Enabled" for *TIM2 global interrupt*.

The prescaler is the number of clock cycles before the timer's counter increments. The counter period is the number of counter increments before an interrupt is raised. With these settings, after 40,000 clock cycles, the counter increments; after 2,000 increments ($= 2,000 \times 40,000 = 80,000,000$ clock cycles), an interrupt is raised. Since the clock runs at 80 MHz, this means the timer TIM2 will raise an interrupt every 1 second. We will use this to trigger code that will change the output of pin PB14, thereby changing whether LED2 is lit or not!

TIM2 Mode and Configuration

Mode

Slave Mode: Disable

Trigger Source: Disable

Clock Source: Internal Clock

Channel1: Disable

Channel2: Disable

Channel3: Disable

Channel4: Disable

Combined Channels: Disable

Use ETR as Clearing Source: Disable

☐ XOR activation

☐ One Pulse Mode

Configuration

Reset Configuration

✔ User Constants ✔ NVIC Settings ✔ DMA Settings

✔ Parameter Settings

Configure the below parameters :

Search (Ctrl+F)

▼ Counter Settings

Prescaler (PSC – 16 bits value): 40000

Counter Mode: Up

Counter Period (AutoReload R...): 2000

Internal Clock Division (CKD): No Division

auto-reload preload: Disable

✔ User Constants	✔ NVIC Settings	✔ DMA Settings	
✔ Parameter Settings			
NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM2 global interrupt	✔	0	0

Name Your Project and Generate Code

At this point, the configuration is complete, and we need to generate the code skeleton that we'll modify to complete the project.

In STM32CubeMX, select “Project Manager” and

- *Set the name of your project.* In OS X, leaving spaces in the name results in errors when STM32CubeIDE attempts to open the project.
- *Set the location of your project.*
- *Select the toolchain for your project.* In this course, instructions will be given assuming you are using STM32CubeIDE; select this. SW4STM32 is another free option.

Project Settings

Project Name
2020-06-08-lab-0-blinky-blinky

Project Location
/Users/bhm/mcgill/stm32/ Browse

Application Structure
Basic ▼ ☐ Do not generate the main()

Toolchain Folder Location
/Users/bhm/mcgill/stm32/2020-06-08-lab-0-blinky-blinky/

Toolchain / IDE
STM32CubeIDE ▼ ☒ Generate Under Root

Now, generate your code!

GENERATE CODE

You will be given the option to open the project in STM32CubeIDE; do so!

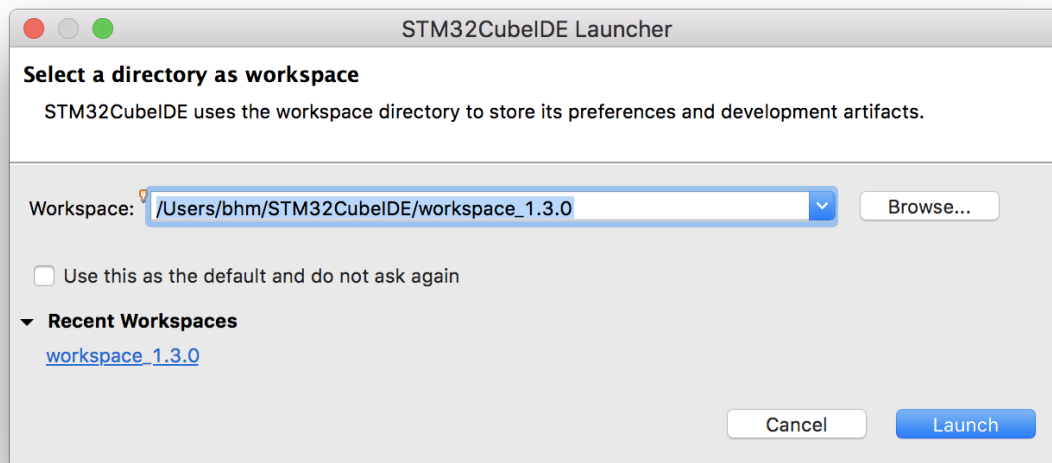
Writing, Deploying, Debugging, and Running Your Code in STM32CubeIDE

Writing Code

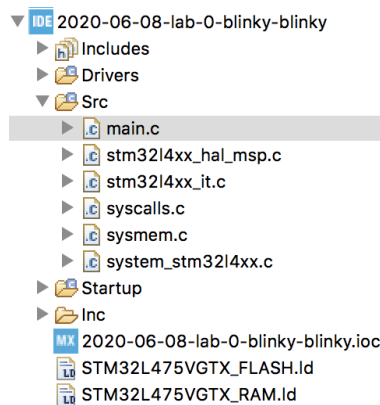
If STM32CubeIDE isn't already open, it will open now.



The first thing you'll have to do is select a workspace. This workspace need not be in the same location as your project: your source files will be saved with your project, but any artifacts (e.g., executables) will be saved in the workspace.



First, we want to open main.c. This is where we'll write our code. You can find it on the left, in the Src directory.



main.c is structured in such a clever, if at first obtuse way. The heavily commented file indicates many areas with “/* USER CODE BEGIN ...” and “/* USER CODE END ...”, delimiting where you, the *user*, are allowed to write your code. **Adhering to the template is essential:** if you need to regenerate the code skeleton in STM32CubeMX (e.g., to change the configuration of a peripheral), as long as you've written your code in user code areas, your hard work will be preserved. However, any code written outside of user code areas may be overwritten.

```

1  /* USER CODE BEGIN Header */
2  /**
3   *
4   * @file      : main.c
5   * @brief     : Main program body
6   *
7   * @attention
8   *
9   * <h2><center>&copy; Copyright (c) 2020 STMicroelectronics.
10  * All rights reserved.</center></h2>
11  *
12  * This software component is licensed by ST under BSD 3-Clause license,
13  * the "License"; You may not use this file except in compliance with the
14  * License. You may obtain a copy of the License at:
15  *
16  *             opensource.org/licenses/BSD-3-Clause
17  *
18  */
19  /* USER CODE END Header */
20
21  /* Includes -----*/
22  #include "main.h"
23
24  /* Private includes -----*/
25  /* USER CODE BEGIN Includes */
26
27  /* USER CODE END Includes */
28
29  /* Private typedef -----*/
30  /* USER CODE BEGIN PTD */
31
32  /* USER CODE END PTD */

```

At this point, there is very little code that we need to write to finish our project: just three lines! The first line of code turns on our timer. Look for `USER CODE BEGIN 2`. In this section, write:

```
HAL_TIM_Base_Start_IT(&htim2);
```

and a comment.

```

85  /* USER CODE BEGIN SysInit */
86
87  /* USER CODE END SysInit */
88
89  /* Initialize all configured peripherals */
90  MX_GPIO_Init();
91  MX_TIM2_Init();
92  /* USER CODE BEGIN 2 */
93  // start the timer and associated interrupt
94  HAL_TIM_Base_Start_IT(&htim2);
95  /* USER CODE END 2 */
96
97  /* Infinite loop */
98  /* USER CODE BEGIN WHILE */
99  while (1)
100 {
101     /* USER CODE END WHILE */
102
103     /* USER CODE BEGIN 3 */
104 }
105 /* USER CODE END 3 */

```

This code takes the address of the timer's configuration register (`&htim2`, defined for you by the skeleton code), and uses a hardware abstraction layer (HAL) function call to direct the timer to start, triggering its interrupt (that's the IT part of the function name) whenever the timer fills and resets.

Remember to write your code between the `BEGIN 2` and `END 2` comments; that way, if you have to regenerate the skeleton code with MX, your code will be preserved.

Now look for `USER CODE BEGIN 4`. In this section, write the following.

```

224 /* USER CODE BEGIN 4 */
225 /**
226  * @brief Interrupt handler for TIM2; toggles LED.
227  * @retval None
228  */
229 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
230     HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
231 }
232 /* USER CODE END 4 */

```

`HAL_TIM_PeriodElapsedCallback` is a virtual function that, when defined, is called by the interrupt handler for timers. More on that later. For now: when the timer reaches its maximum and resets, this function is called.

`HAL_GPIO_TogglePin` takes two inputs: a block of GPIO pins, or port, and a specific pin. Fortunately, we don't need to know the addresses for either of these, or do the math to work it

out from pin numbers: since we labeled the pin LED, we can simply use `LED_GPIO_Port` to refer to the port, and `LED_Pin` to refer to the pin. These macros have already been defined for us elsewhere in the skeleton code.

Deploying, Debugging, and Running Code

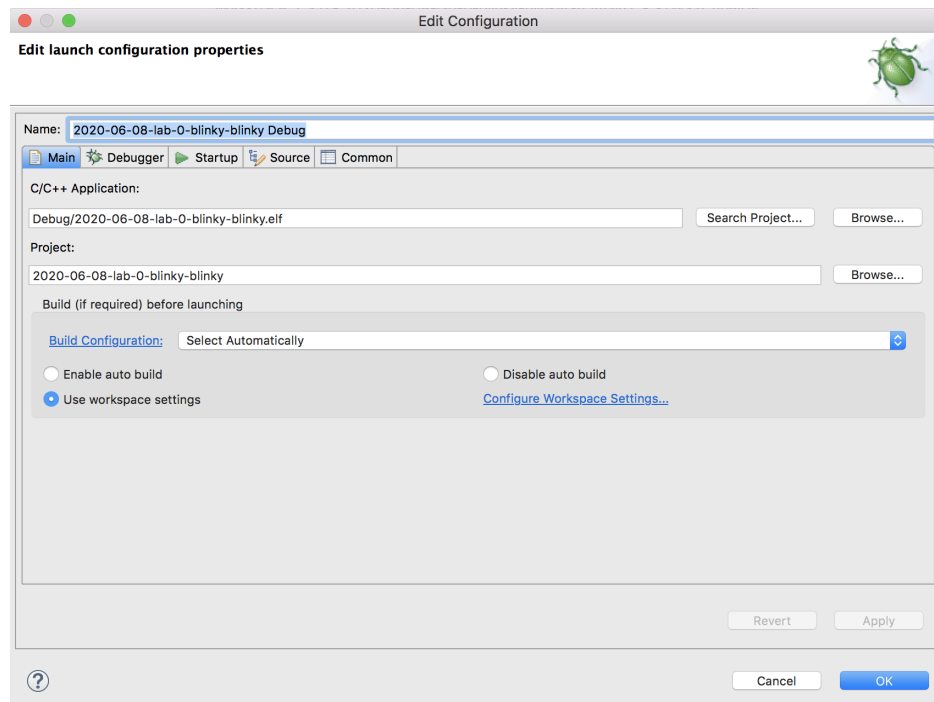
Now we're ready to test our code! If you've used Eclipse before, the interface will be familiar.



Before clicking anything, connect your Discovery kit to your computer. A red LED in the corner next to the USB port should light.

Clicking the bug starts debugging (the program will be compiled, and started on the development board, with a breakpoint set at the first line of `main`). Clicking the play button starts the program without connecting a debugger (the program will be compiled, and started).

When you first click the bug or play button for any project, you'll be prompted:



Click "OK" to continue.

If you've configured everything properly, and written your code as shown above, compilation should complete without error or warning, and execution should begin.

Note: if STM32CubeIDE complains that [ST-LINK-Server](#) could not be found, please install this software and try again.

If you started the debugger, you can walk through the code until the infinite while loop using these controls.



Press the Resume button (left of pause, right of terminate and relaunch) to allow the program to run (until you pause it). The green LED should begin to blink, toggling on for one second, and then off again for one second!

Additional Exercises

Want to convince yourself that you understand everything that's going on? Here are a few variations that you can explore.

1. Change the configuration of TIM2 in STM32CubeMX so LED2 toggles four times faster.
2. Change the configuration of TIM2 in source files in STM32CubeIDE so that TIM2 toggles four times slower.
3. Change the configuration of the system in STM32CubeMX so a different LED toggles.