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-IOT01A or B-L4S5I-IOT01A development board, and deploy simple software to it, in this case, making an LED blink.

Deliverables

None.

Grading

Not graded.

Changelog

● 1-Sep-2023	Added a note about the STM32CubeIDE version in the lab (1.13.1).
● 3-Jul-2023	Updates for Fall 2023: references to CubeMX removed; parallel instructions for the B-L4S5I-IOT01A board added.
● 9-Sep-2021	Updates to account for changes in CubeMX 6.3.
● 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-IOT01A (B-L4S5I-IOT01A) 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 development board.
- Micro USB data+charging cable; note, charging-only cables are not adequate.

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

- ST-LINK-Server
- STM32CubeIDE

Installation Notes

- The lab computers are running STM32CubeIDE v.1.13.1 this year. You should do the same to make your projects portable.
- You need to install ST-LINK-Server before STM32CubeIDE will be able to communicate with the development board. OS X: it is included as a separate package in the disk image; install it first!
- STM32CubeIDE runs slowly the first time you open it, begin a project, etc; the installation package is over 1 GB, but there are still more files to download! Be patient.
- STM32CubeIDE may also prompt you to update the firmware of your development board; do so.

For more information on installation, follow the link for your operating system; common issues and fixes will appear there as they arise.

- Linux
- OS X
- Windows

All of our projects will start with creating a project in STM32CubeIDE and configuring our development board. 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. At its core, STM32CubeIDE 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. Board configuration and Code Generation
2. Writing, Deploying, Debugging, and Running Your Code
3. Additional Exercises

Board Configuration and Code Generation

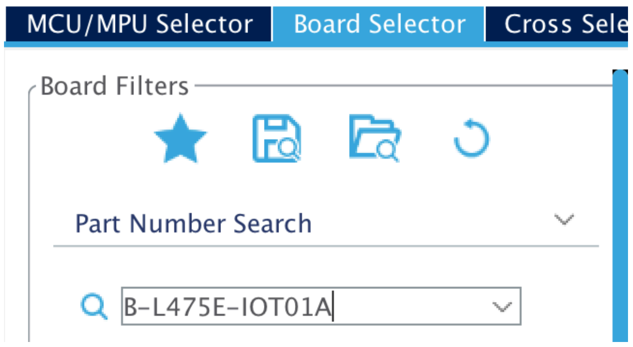
First, we'll create a new project for our development board and generate the skeleton code that will initialize its peripherals.

Selecting the Discovery kit for IoT Development Board

Open the software and *Create a New STM32 project*. This will open the Target Selector.

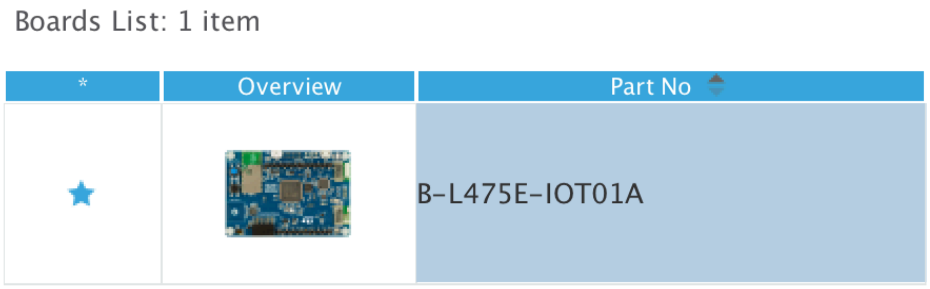
Note: the first time you do this, STM32CubeIDE will download the target selector database; this can take a while.

Search for your board, e.g., the B-L475E-IOT01A1. Check the bottom of the box it came in to see which you are using.



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 click *Next* >, and:

- Name the project,
- Select "C" as the target language,
- Select "Executable" as the targeted binary type,

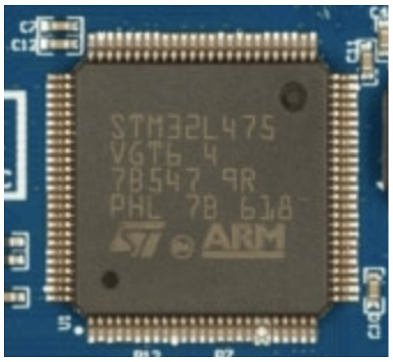
then click *Finish*.

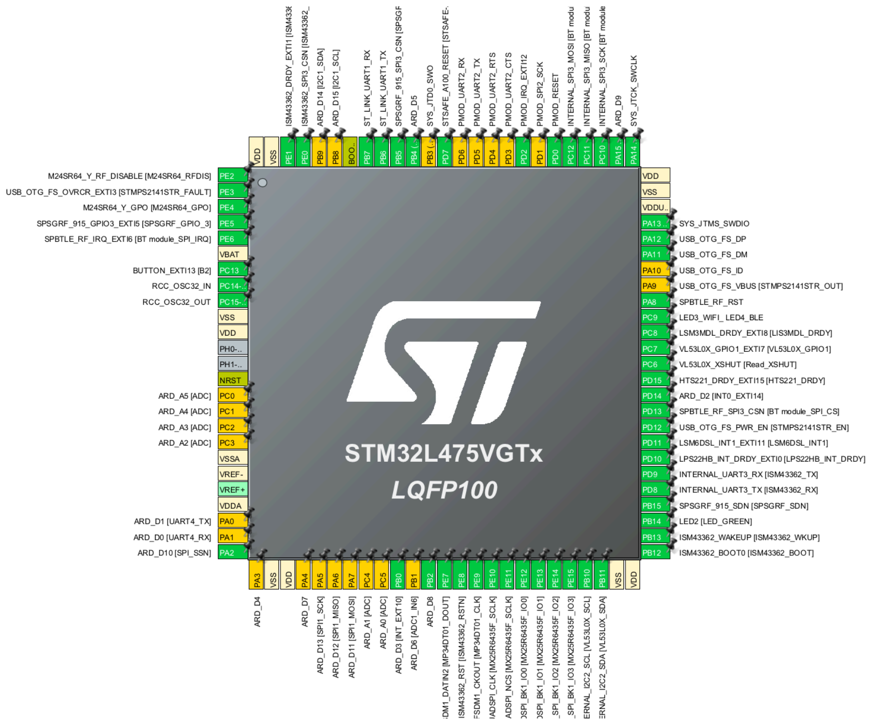
Note: the first time you do this, STM32CubeIDE will download all the support software for the target you've selected; this can take a while.

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.

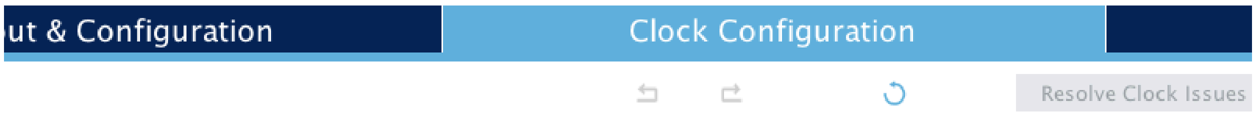


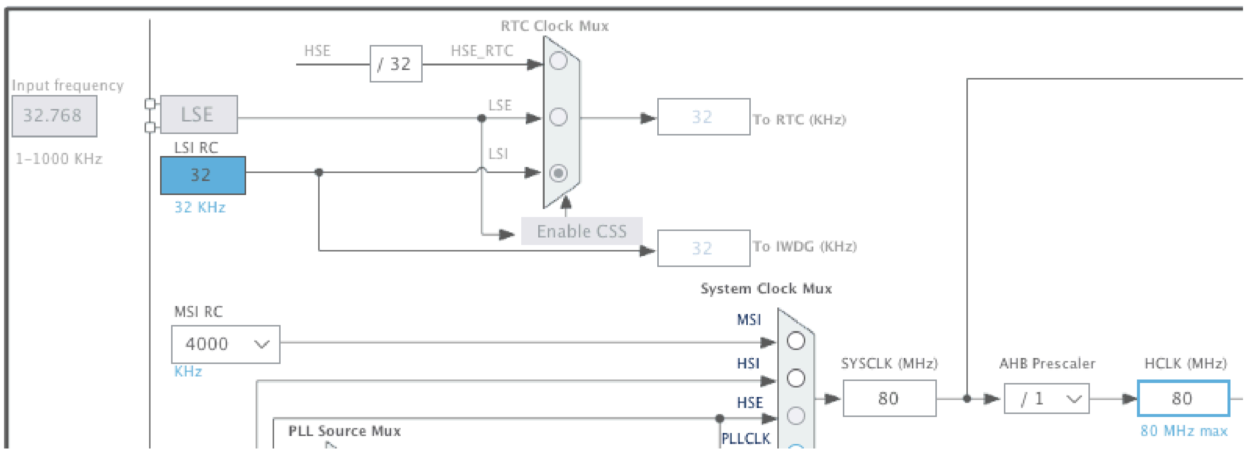

STM32L475VGTx
LQFP100
M24SR64_Y_RF_DISABLE [M24SR64_RFDIS]
PE2
USB_OTG_FS_OVRCR_EXTI3 [STMPS2141STR_FAULT]
PE3
M24SR64_Y_GPO [M24SR64_GPO]
PE4
SPSGRF_915_GPIO3_EXTI5 [SPSGRF_GPIO_3]
PE5
SPBTLE_RF_IRQ_EXTI6 [BT module_SPI_IRQ]
PE6
VBAT
BUTTON_EXTI13 [B2]
PC13
RCC_OSC32_IN
PC14-..
RCC_OSC32_OUT
PC15-..
VSS
VDD
PH0-..
PH1-..
NRST
ARD_A5 [ADC]
PC0
ARD_A4 [ADC]
PC1
ARD_A3 [ADC]
PC2
ARD_A2 [ADC]
PC3
VSSA
VREF-
VREF+
VDDA
ARD_D1 [UART4_TX]
PA0
ARD_D0 [UART4_RX]
PA1
ARD_D10 [SPI_SSN]
PA2
VDD
VSS
PE1
ISM43362_DRDY_EXTI1 [ISM433
PE0
ISM43362_SPI3_CSN [ISM43362_
PB9
ARD_D14 [I2C1_SDA]
PB8
ARD_D15 [I2C1_SCL]
BOO..
PB7
ST_LINK_UART1_RX
PB6
ST_LINK_UART1_TX
PB5
SPSGRF_915_SPI3_CSN [SPSGF
PB4 (..
ARD_D5
PB3 (..
SYS_JTD0_SWO
PD7
STSAFE_A100_RESET [STSAFE-
PD6
PMOD_UART2_RX
PD5
PMOD_UART2_TX
PD4
PMOD_UART2_RTS
PD3
PMOD_UART2_CTS
PD2
PMOD_IRQ_EXTI12
PD1
PMOD_SPI2_SCK
PD0
PMOD_RESET
PC12
INTERNAL_SPI3_MOSI [BT modu
PC11
INTERNAL_SPI3_MISO [BT modu
PC10
INTERNAL_SPI3_SCK [BT module
PA15 ..
ARD_D9
PA14 ..
SYS_JTCK_SWCLK
VDD
VSS
VDDU..
PA13 ..
SYS_JTMS_SWDIO
PA12
USB_OTG_FS_DP
PA11
USB_OTG_FS_DM
PA10
USB_OTG_FS_ID
PA9
USB_OTG_FS_VBUS [STMPS2141STR_OUT]
PA8
SPBTLE_RF_RST
PC9
LED3_WIFI_ LED4_BLE
PC8
LSM3MDL_DRDY_EXTI8 [LIS3MDL_DRDY]
PC7
VL53L0X_GPIO1_EXTI7 [VL53L0X_GPIO1]
PC6
VL53L0X_XSHUT [Read_XSHUT]
PD15
HTS221_DRDY_EXTI15 [HTS221_DRDY]
PD14
ARD_D2 [INT0_EXTI14]
PD13
SPBTLE_RF_SPI3_CSN [BT module_SPI_CS]
PD12
USB_OTG_FS_PWR_EN [STMPS2141STR_EN]
PD11
LSM6DSL_INT1_EXTI11 [LSM6DSL_INT1]
PD10
LPS22HB_INT_DRDY_EXTI0 [LPS22HB_INT_DRDY]
PD9
INTERNAL_UART3_RX [ISM43362_TX]
PD8
INTERNAL_UART3_TX [ISM43362_RX]
PB15
SPSGRF_915_SDN [SPSGRF_SDN]
PB14
LED2 [LED_GREEN]
PB13
ISM43362_WAKEUP [ISM43362_WKUP]
PB12
ISM43362_BOOT0 [ISM43362_BOOT]
PA3
ARD_D4
VSS
VDD
PA4
ARD_D7
PA5
ARD_D13 [SPI1_SCK]
PA6
ARD_D12 [SPI1_MISO]
PA7
ARD_D11 [SPI1_MOSI]
PC4
ARD_A1 [ADC]
PC5
ARD_A0 [ADC]
PB0
ARD_D3 [INT_EXT10]
PB1
ARD_D6 [ADC1_IN6]
PB2
ARD_D8
PE7
DM1_DATIN2 [MP34DT01_DOUT]
PE8
ISM43362_RST [ISM43362_RSTN]
PE9
FSDM1_CKOUT [MP34DT01_CLK]
PE10
ADSPI_CLK [MX25R6435F_SCLK]
PE11
ADSPI_NCS [MX25R6435F_SCLK]
PE12
SPI_BK1_IO0 [MX25R6435F_IO0]
PE13
SPI_BK1_IO1 [MX25R6435F_IO1]
PE14
SPI_BK1_IO2 [MX25R6435F_IO2]
PE15
SPI_BK1_IO3 [MX25R6435F_IO3]
PB10
ERNAL_I2C2_SCL [VL53L0X_SCL]
PB11
ERNAL_I2C2_SDA [VL53L0X_SDA]
VSS
VDD

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 (or, 120 MHz for the B-L4S5I-IOT01A), 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 (120 MHz) or not has consequences for system operation.



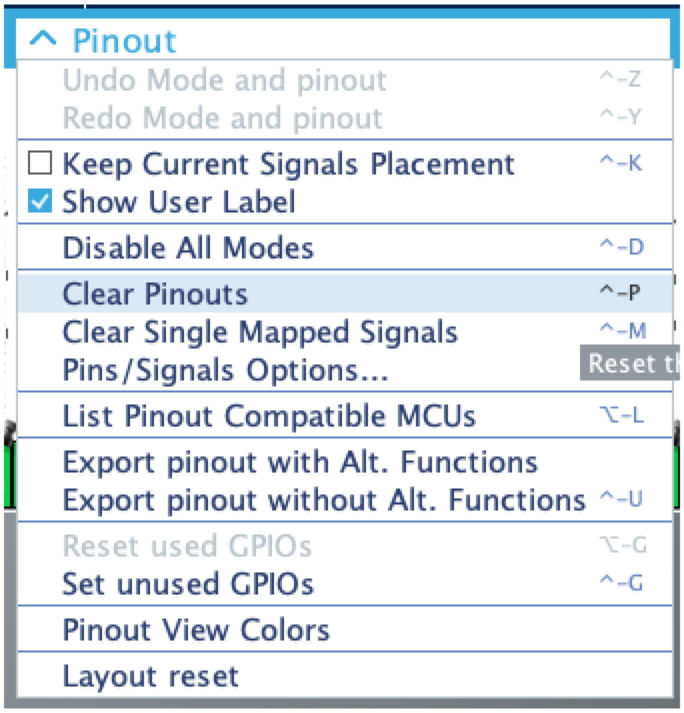


HCLK is key for now. If it isn't set to 80 (MHz), do so. The tool will find the appropriate configuration of multiplexors and clock sources to reach the target.

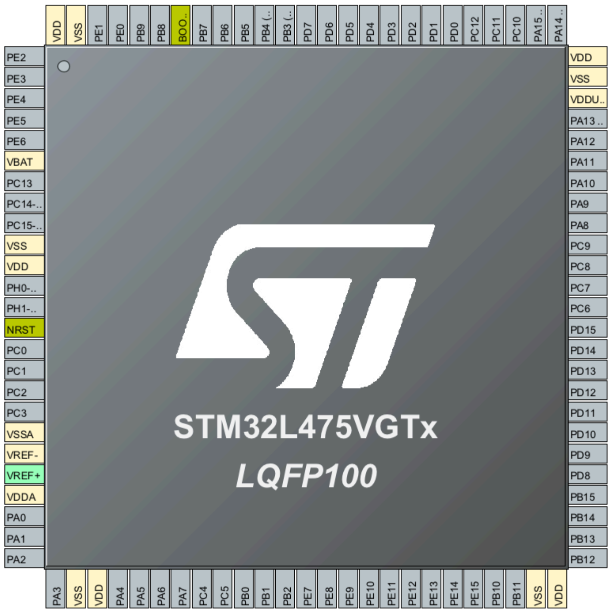
Configuring LED2

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 its 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 and 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 B-L475E-IOT01A user's manual details the available peripherals, and their interface(s):

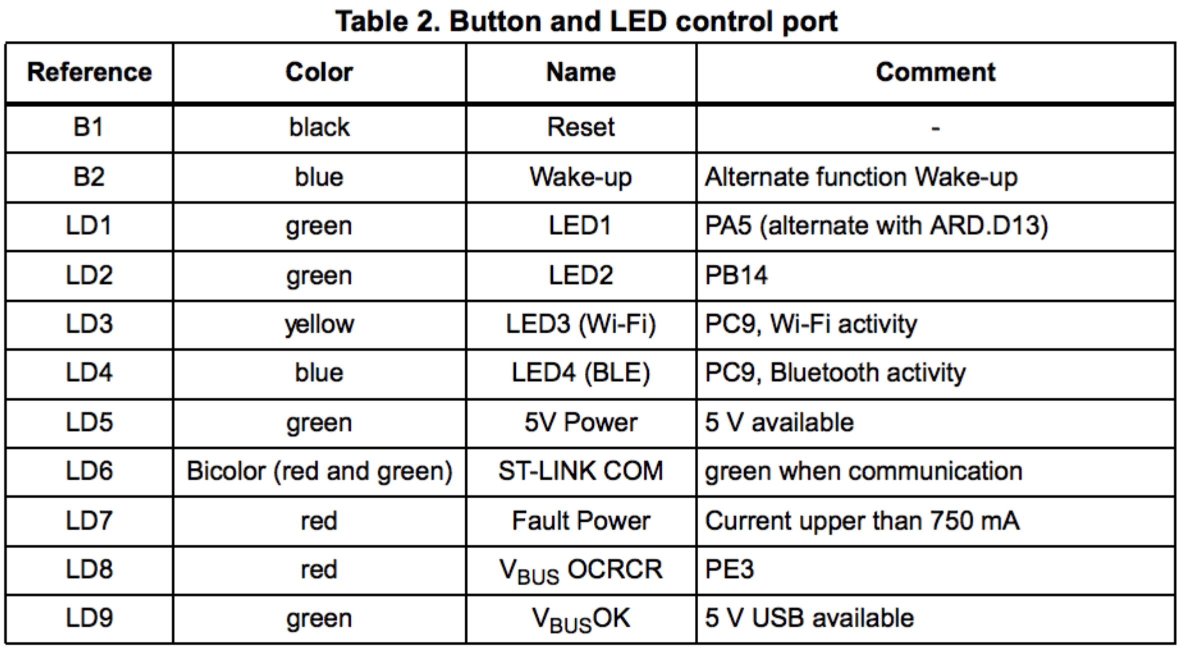
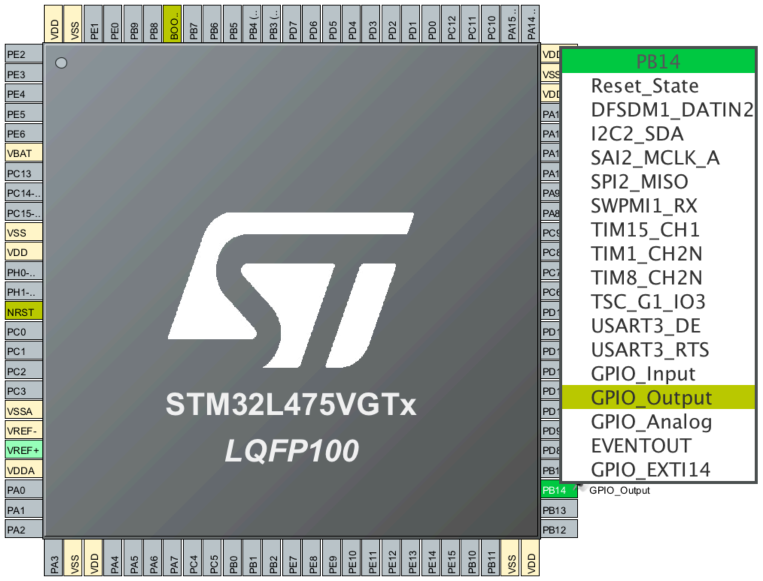


Table 2. Button and LED control port

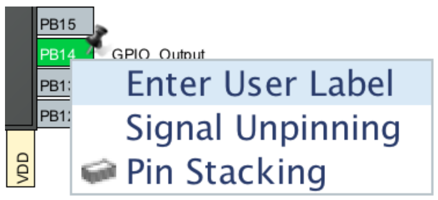
Reference	Color	Name	Comment
B1	black	Reset	-
B2	blue	Wake-up	Alternate function Wake-up
LD1	green	LED1	PA5 (alternate with ARD.D13)
LD2	green	LED2	PB14
LD3	yellow	LED3 (Wi-Fi)	PC9, Wi-Fi activity
LD4	blue	LED4 (BLE)	PC9, Bluetooth activity
LD5	green	5V Power	5 V available
LD6	Bicolor (red and green)	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

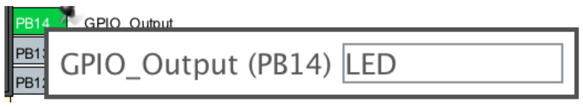
(The same specs appear in Table 4 of the B-L4S5I-IOT01A user's manual.)

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 (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 TIM17

Before configuring the timer that we'll use to make our LED blink, there is one other change to make to how timer 17, TIM17, is being used.

Expand "System Core" on the left hand side, and select *SYS*. This will expand *SYS Mode and Configuration*.

- In *Mode*, select “SysTick” from Timebase Source.

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 clock edges for 1 second, triggering an interrupt, which we will use to toggle the LED.

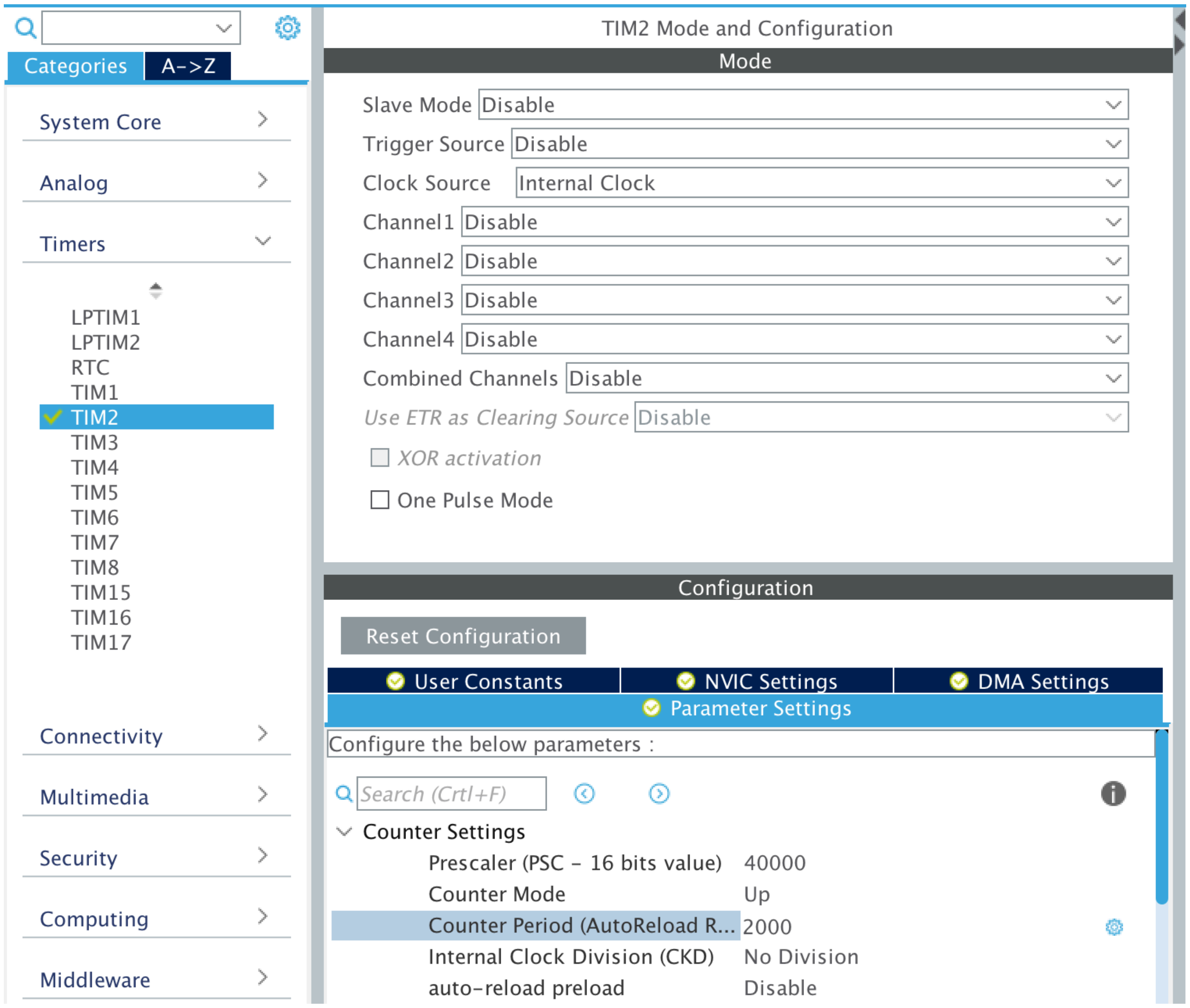
The timers are internal to the processor: i.e., they aren’t accessible through pins. To get started with configuring *TIM2*, we first select it from the timers. Expand “Timers” on the left hand side, and select “TIM2,” as illustrated below. This will expand *TIM2 Mode and Configuration*. Now we need to configure the clock used for counting, and configure the target value.

- In *Mode*, select “Internal Clock” from Clock Source. This means the timer will count processor clock edges at 80 MHz (120 MHz).

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. If we set the prescalar to 40,000, and counter period to 2,000, after 40,000 clock cycles, the counter increments; after 2,000 increments (= 2,000 x 40,000 = 80,000,000 clock cycles), an interrupt is raised. If 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!

If you are using the B-L4S5I-IOT01A, you need to pick a prescaler and counter period that counts 120 million cycles instead! E.g., 40,000 and 3,000.

- In *Configuration*, under *Parameter Settings*, set *Prescaler* to 40000, and *Counter Period* to 2000 (3000).
- In Configuration, under *NVIC Settings*, tick “Enabled” for *TIM2 global interrupt*.



TIM2 Mode and Configuration

Mode

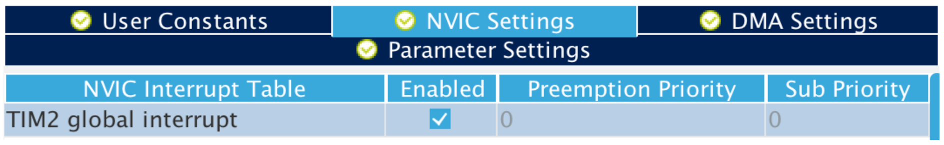
Slave Mode: Disable
Trigger Source: Disable
Clock Source: Internal Clock
Channel1: Disable
Channel2: Disable
Channel3: Disable
Channel4: Disable
Combined Channels: Disable
Use ETR as Clearing Source: Disable
☐ *XOR activation*
☐ One Pulse Mode

Configuration

Configure the below parameters :

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



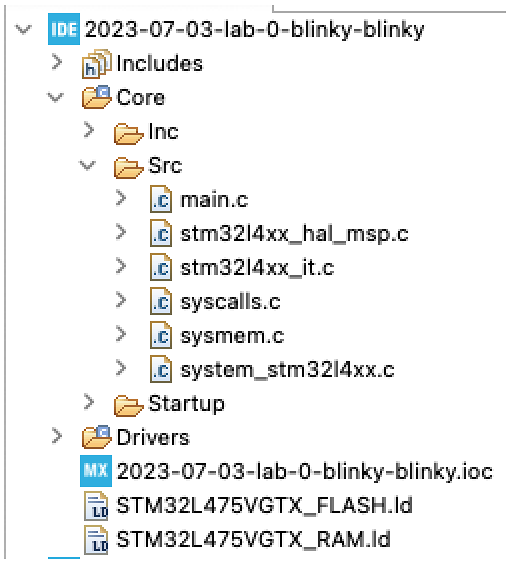
NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM2 global interrupt	✓	0	0

At this point, the configuration is complete, and we need to generate the code skeleton that we'll modify to complete the project. There are a variety of ways to trigger code generation:

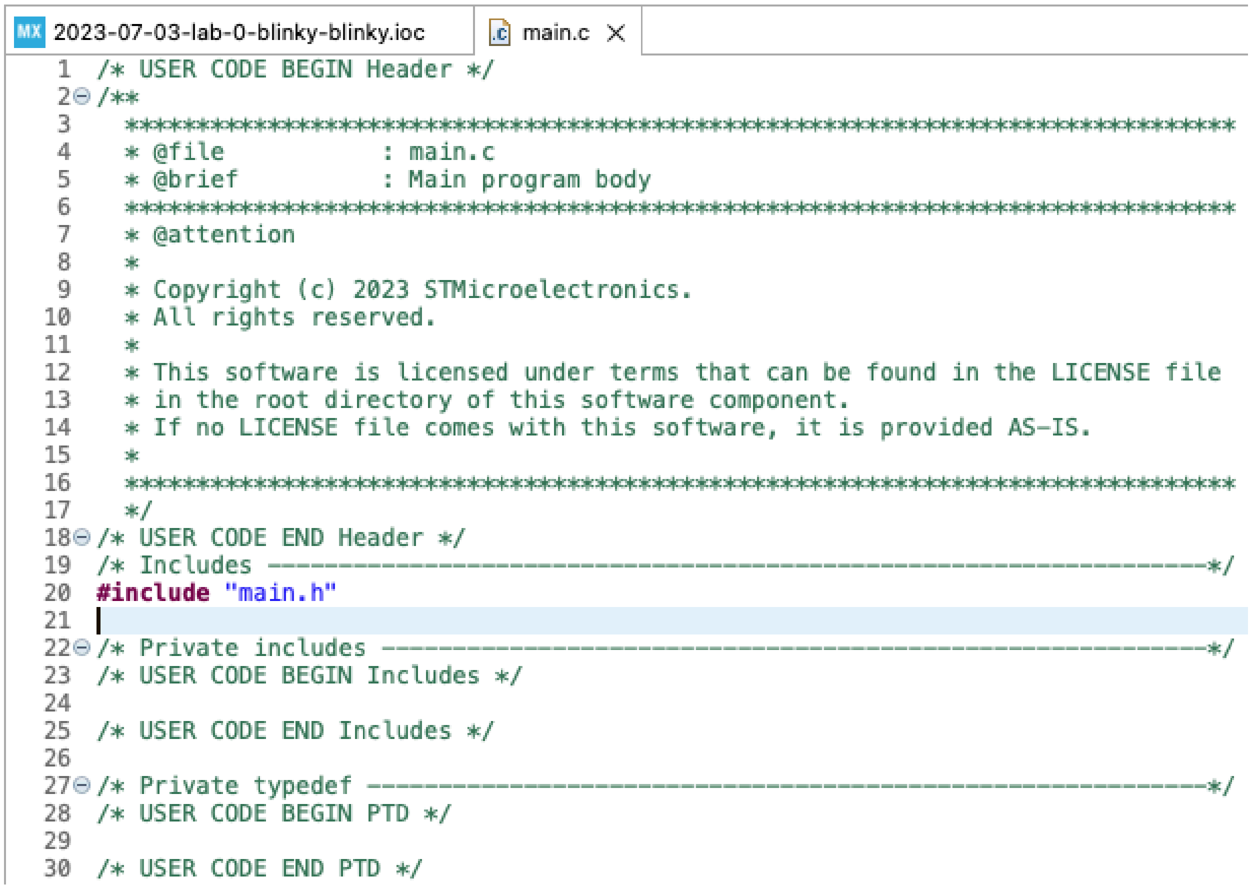
- Save the IOC file using the save icon. This will prompt you to generate code.
- **Right-click** the IOC file in the Project Explorer pane and select *Generate Code*.
- From the Project pull-down menu select *Generate Code*.

Writing, Deploying, Debugging, and Running Your Code

First, we want to look at `main.c`; this is where we'll write our code! It may have opened automatically upon completing code generation. You can also 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 (e.g., after you change the configuration of a peripheral in the project IOC), 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.

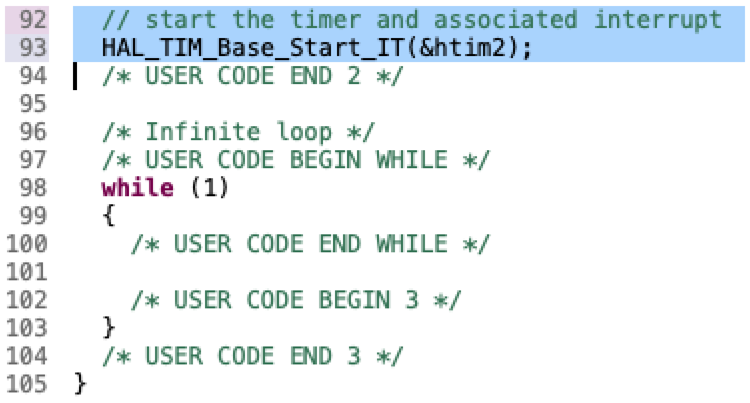

2023-07-03-lab-0-blinky-blinky.ioc   main.c

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2023 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* 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.

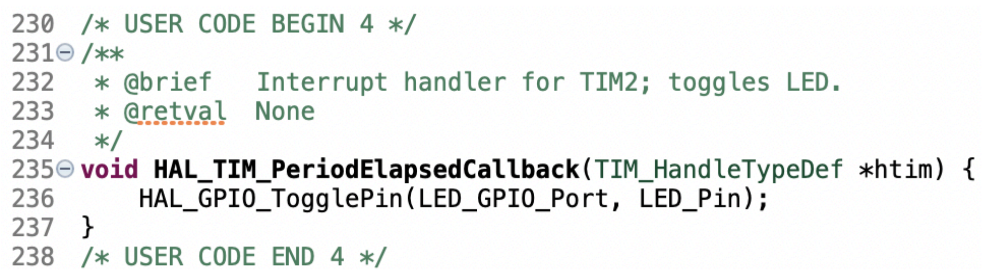


```
  // start the timer and associated interrupt
  HAL_TIM_Base_Start_IT(&htim2);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* 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.

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

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



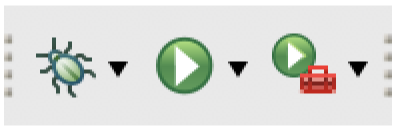
```
/* USER CODE BEGIN 4 */
/**
  * @brief   Interrupt handler for TIM2; toggles LED.
  * @retval  None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
}
/* USER CODE END 4 */
```

`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 memory 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

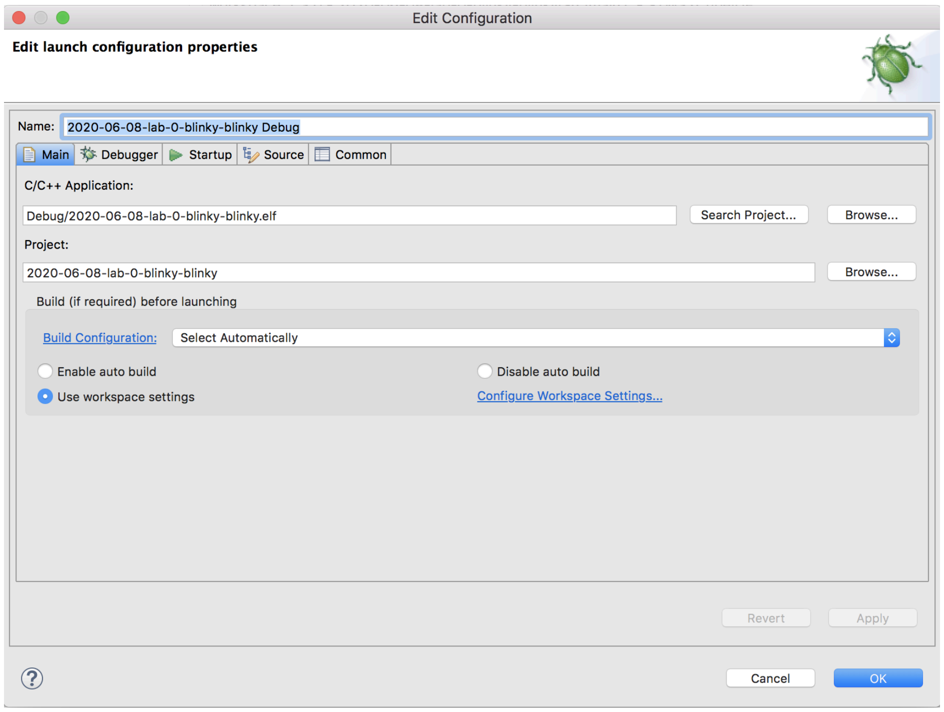
Now we're ready to test our code! If you've used Eclipse before, the interface will be familiar.



Before clicking anything, connect your Discovery kit to your computer. A red LED in the corner next to the USB port should light. *If the red LED is blinking slowly, this means that it has not successfully connected to your computer; this may be due to using a USB charging cable, rather than a data cable.*

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:



Name: 2020-06-08-lab-0-blinky-blinky Debug

C/C++ Application: Debug/2020-06-08-lab-0-blinky-blinky.elf

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

Build (if required) before launching

Build Configuration: Select Automatically

Enable auto build
Disable auto build
[✓ Use workspace settings]

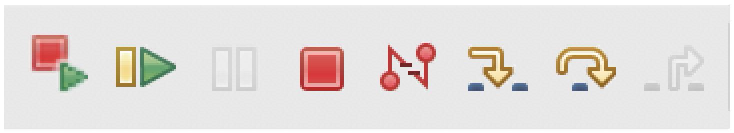
Click "OK" to continue.

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

At this point, you may be prompted to update the firmware on your development board. Do so at this time, and then start the debugger again.

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 the board configuration (IOC file) so LED2 toggles four times faster.
2. Change the configuration of TIM2 in source files in STM32CubeIDE so that TIM2 toggles four times slower. Remember, the IOC file is just a fancy UI to facilitate code generation!
3. Change the configuration of the system so a different LED toggles.