

CSE190 Project 1

Blinkenlight Communication

Submit on Gradescope by Fri Jan 17 5 PM

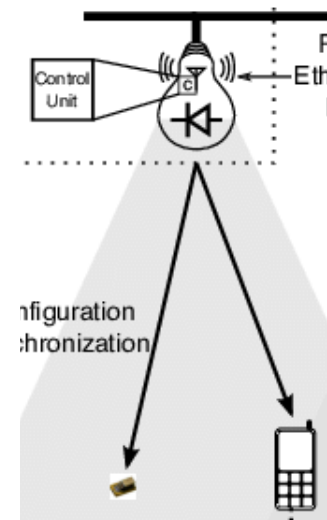
This is a group assignment with at most two people per group. One group member must submit their final code to [Gradescope](#) using the link on Canvas.

[Create your Team's Github Classroom account to get the starter code](#)

Overview

Communicating with visual light is the most basic mode of “wireless” communication. Light-based communication started with early humans. Back then, long distance communication was achieved by encoding messages in puffs of smoke, and transmitting them (naturally) by reflecting light off of the smoke particles (aka smoke signals). Needless to say, this form of communication had a very slow data rate (i.e., bits per minute).

In this project¹, we will be building a comparatively high data rate (320 bits per second), short-range visual light communication system from scratch. The transmitter will be a set of blinky LEDs on your embedded system; namely, the two green LEDs on the top of your development board, and the receiver will be the video camera on your smartphone. You will send a “secret” ASCII character message (ok, it won’t be that secret, it’s gonna be your student ID number, which you will also mention in a README file) that anyone will be able to receive by recording a 60fps video of your device’s blinking LEDs with their smartphone. They will decode the message by playing back the video frame-by-frame and they will be able to decode two bits at a time looking at what LEDs are lit every 3 frames in the video. Capturing and reading LEDs blinking quickly on video is prone to errors, so you will also be sending special error checking bits that can be used to detect if there were any errors when transmitting the message.



You are writing all the source code from scratch, so you will need to build the drivers for each component first, then you can build the final Blinkenlight communication application. Specifically, the project will involve the following steps, each of which is described in detail in this specification:

1. **Setup your development environment** - Since this is your first project, you will need to setup software tools to compile and program the STM32 (ARM Cortex M4) MCU on your development board.

¹ This project is inspired by [Luxapose](#) by Pannuto et al. The diagram above is borrowed from their work.

2. **Implement the LED driver** - This driver will give you a simple function to call to select which LED of the two green on-board LEDs, if any, is illuminated.
3. **Implement the Timer driver** - This driver will provide an abstraction of the Timer peripheral so you can use it in your Blink app to control when a new set of bits is appearing on the LEDs.
4. **Implement the Blinkenlight Communication app** - This application will use the blink library to blink the light at different rates.

Resources (Important!)

For this project, the most important documents you need to understand the circuits and components are:

- [YouLostIt Project Template - STM32CubeIDE](#) (template project on github classroom)
- [STM32CubeIDE](#)
- [Schematic - STM32 Development board](#)
- [Reference Manual - STM32L Microcontroller](#)
- [NVIC header file](#)
- Digikay link [if you want to buy your own development board for the class](#) (we will supply

The datasheet covers multiple variants of the STM32, **we are using the STM32L475VGTX package.**

The CMSIS headers also come with headers for multiple variants of the SAM21, you will only need the headers in the samd21 folder.

Memory map header files: For all of the projects in the class, you will want to refer to the MCU (microcontroller) memory map for the specific MCU we are using that starts with the header file: `CMSIS/Device/ST/Include/stm32l475xx.h` This contains all of the definitions of the variables (pointers and structs) according to our MCU's memory map. This will make it easy for you to access the registers to communicate with the MCU's peripherals. This header links to other header files that provide even more convenient variables and structs that you can use.

During the course of the project, please post any questions you have to the Piazza class discussion forum so we can share the knowledge with all the other students.

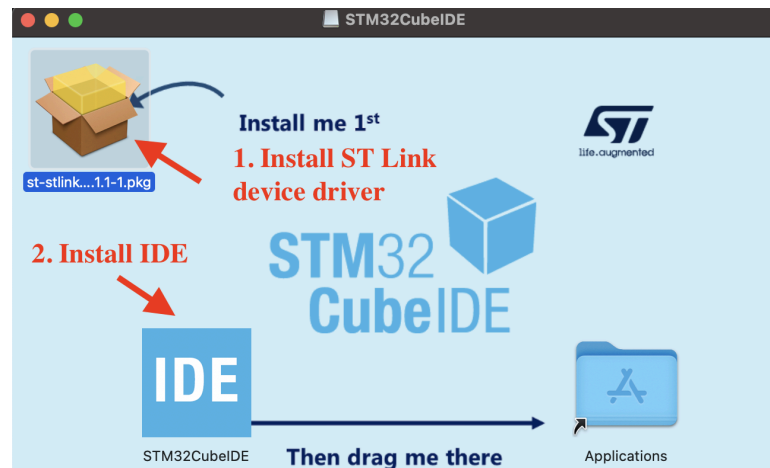
Grading Criteria

- 50% the blink application outputs your secret message correctly when flashed to a test board
- 10% the LED driver is implemented correctly
- 10% the Timer driver is implemented correctly
- 10% the Blink app is implemented correctly
- 20% your code is clear and confusing portions are well commented

Step 0: Setup the development environment

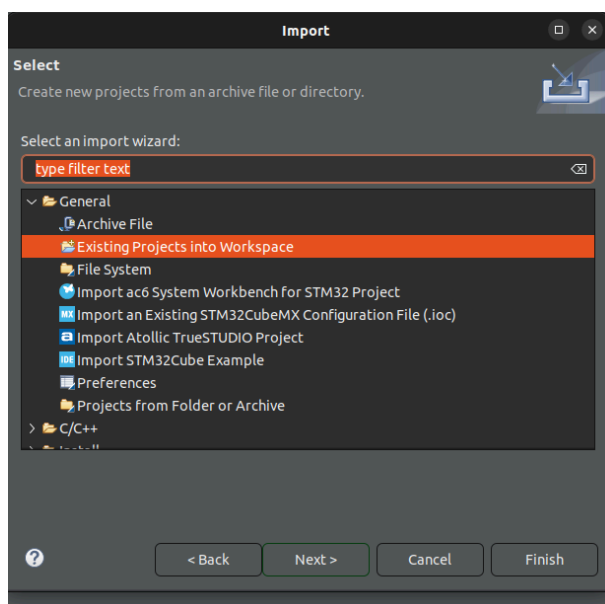
1. [Download the STM32CubeIDE](#). We will use this IDE because it comes with all of the tools that need to build and deploy C and C++ applications onto the TinyZero board. Specifically, the

Arduino IDE includes binaries of the [ARM g++ toolchain](#) (compiler and linker), STLink USB programmer, and USB drivers built for your development platform of choice (Win32/64, Linux, MacOS). It also automatically calls these tools with the correct parameters when you build and deploy your application. If you are adventurous you can try to use the tools outside of the STM32Cube IDE, but that will require substantial effort to bootstrap.

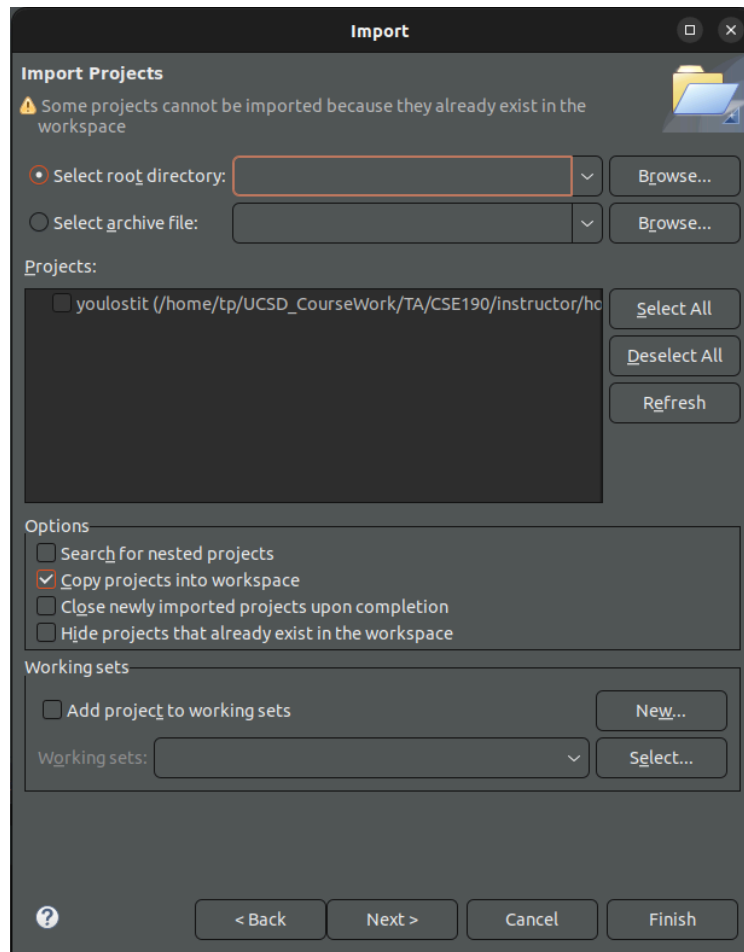



2. Install *both* the ST Link USB driver that will be used for writing firmware to the chip and debugging your code live as it is running. Then, plug your USB cable into the *STLink USB* port on the edge of the development board.

3. To import the project, first create a workspace, then navigate to file -> import. Select "Existing Projects into Workspace" and click next.



Enter into "Select root directory" the path to where you cloned the github repo; select copy projects into workspace; click finish.



4. Open the “YouLostIt” template project and click the build & debug button . In the “Console” window at the bottom you will see the gcc compiler run. Then it will flash your program onto the MCU. Take a look at the arguments given to the compiler to see how they customize it to work with your microcontroller programmed to your chipset to match your deployment. There is no magic, it just compiles your code with these extra flags, then it just copies the resulting binary file into the flash on the MCU, then it resets the MCU and starts execution immediately after.

Step 1: Implement the LED driver

The LED driver will provide a basic abstraction to control the two on-board LEDs, so other drivers and libraries can control them. To do this you will need to know how to use the GPIO (general purpose input/output) pins. This microcontroller has 38 GPIO pins, but we will only be using 5 for this project.

We recommend making a copy of the template project as a starting point. It has all the basic code you need to write, compile, and debug code in C for the microcontroller. It even demonstrates how to use GPIO to turn off all of the LEDs on the 16 LED daughterboard.

To see all the other GPIO registers you can set, check out *port.h* in the SAM header file directory.

Files that will be graded: `leds.c`, `leds.h`

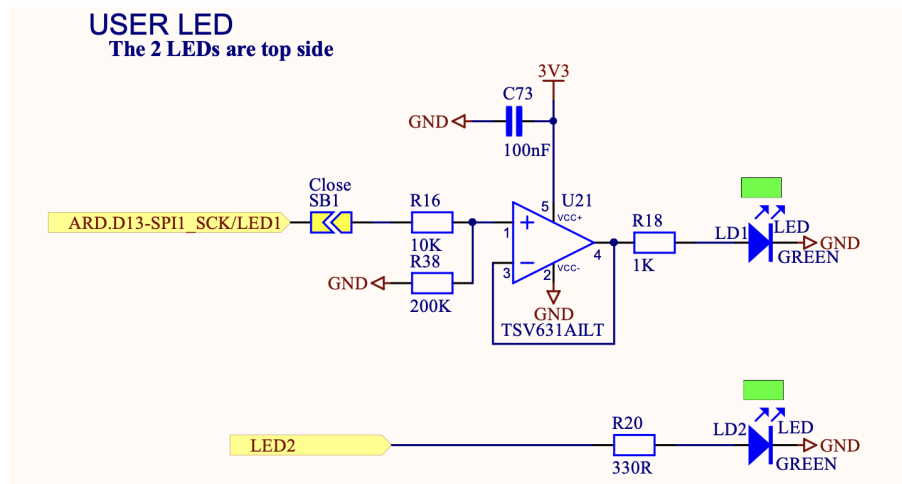
Relevant datasheets: STM32L4X Reference Manual - Chapter 8 - General-Purpose I/Os (GPIO)

Relevant header files:

- `CMSIS/Device/ST/Include/stm321475xx.h`

You will implement the following LED control function: turn on the selected LED on the LED display. Note that you will have to select which LED you are controlling using the GPIO pins to select one LED to turn on at a time.

By outputting a 1 and a 0 on any GPIO pin that is connected to a LEDs, you will be able to turn it on and off. You can see why this works by looking at the schematic of the LEDs on our board (shown below, but also in the [full schematic document](#)). LEDs have a neat feature, they are actually diodes that also happen to output light. As diodes, they act as one way valves in your circuit, they only let current flow in one direction, the direction indicated by the large **blue arrow** next to the LD* marker on the schematic. The arrow points from the high voltage to the low voltage when the LED is on. The input to the LED is shown as a **yellow flag**. That yellow flag indicates that the wire connects to another part of the schematic. Look for the other side of where the wire connects, and you will see it is to the microcontroller GPIO pins **PA5** and **PB14**. Note that LED1 has an additional amplifier (U21) that allows for faster and crisper transitions between on/off for that LED. Although that is cool, we will not be pushing it to that limit in this project.



There are several ways to turn an LED off using a GPIO. (1) The LED will not output if you have a low voltage connected to both sides of the LED. (2) The LED will not output if you put the GPIO pin in the "tri-state", or "high-impedance mode". In high-impedance mode, essentially, this will completely disconnect the wire from the LED, making it impossible for current to flow through the LEDs. You can achieve this by disabling the output mode of the GPIO, or setting the GPIO pin to be an input (see the register descriptions in the port section for more details).

Aren't GPIOs truly remarkable devices! You may think they just output a binary 1 or a 0, but they are not binary devices! they actually can output three values (0, 1, and disconnected).

```
void leds_set(uint8_t led);
```

This function will set the two user LEDs on or off.

The input parameter `led` is a 2-bit bitmask with each bit representing each of the two user LED's in order. So bit 0 represents LED1 (LD1), bit 1 represents LED2 (LD2). Your driver implementation will use this value to decide which LEDs need to be turned on. For example passing a value of 3 (0b11) will turn on both LEDs LD1 and LD2, and a value of 2 (0b10) will only turn on LD2.

Step 2: Implement the Timer driver

The timer driver will provide an abstraction of the hardware "timer" peripheral. The goal is to make an easy interface for drivers and libraries to use hardware timers. You will use the timer driver in the YouLostIt app to create a timer that fires once every N milliseconds. You will use this timer to produce a time interval that you will change your bit values shown on the two user LEDs. Our microcontroller's hardware timer is capable of many features, you will only need one specific thing, counting up and firing an interrupt and repeating when that is done.

Relevant documents: SAMD21 MCU datasheet

- Chapter 31 General-purpose timers (TIM2/TIM3/TIM4/TIM5)
- Chapter 6 Reset and clock control (RCC)

Relevant header files:

- `core_cm4.h` (NVIC Header file)

Files that will be graded: `timer.c`, `timer.h`

```
void timer_init(TIM_TypeDef* timer);
```

Setup hardware timer (TIM2). You will need to set up the clock source that the timer will use to keep track of time (*hint: you will be referring to the clock tree in Chapter 6.2 Figure 15 to find out how the clock source connects to your timer peripheral*). The timer init should do the following:

1. Stop the timer and clear out any timer state and reset all counters.
2. Setup the timer to auto-reload when the max value is reached.
3. Enable the timer's interrupt both internally and in the interrupt controller (NVIC).
 - a. You will need to use the NVIC functions `NVIC_EnableIRQ`, `NVIC_SetPriority` with the parameter `TIM2_IRQn`
4. Setup the clock tree to pass into the timer and divide it down as needed (*hint: look at the ENR registers in the RCC peripheral*). Note: The default clock speed of your microcontroller after reboot is 4 MHz. You may want to slow this down by setting the dividers in the clock tree so the timer has a slower clock to operate with (Chapter 6).
5. Enable the timer.

```
void timer_reset(TIM_TypeDef* timer);
```

Reset timer 2's (TIM2) counters, but do not reset the entire TIM peripheral. The timer can be in the middle of execution when it is reset and it's counter will return to 0 when this function is called.

```
void timer_set_ms(TIM_TypeDef* timer, uint16_t period_ms);
```

Set the *period* that the timer will fire (in milliseconds). A timer interrupt should be fired for each timer period.

```
Void TIM2_IRQHandler() {};
```

Interrupt handler that will fire at the end of each period of TIM2. This function can go in main.c. Note that global variables that are modified in interrupt handlers must be declared as *volatile*. For more information about the *volatile* keyword check out this [tutorial](#).

Note: If the interrupt handler is not being called and you think it should be, it could be because you have not enabled the interrupt controller, you also may not have enabled interrupts in the CPU.

Step 3: Implement the YouLostIt Blinkenlight app

This is the main code that you will implement for this project. The led and timer libraries provide a convenient interface for setting up blinking LEDs.

Files that will be graded: main.c

Use the functions from steps 1 and 2 in main.c to make the “two LED display” show your student ID, 2 bits at a time. LED1 will always show the lower order bit and LED2 will show the higher order bit. You should update the LEDs with new bits every 1/20th of a second (20 Hz). This is called your *symbol rate* or the rate you are transmitting new bits (you can do the math yourself to figure what the system’s bitrate is, believe me it ain’t great).

The character sequence you will be displaying is the last 4 digits of one of your group members’ student ID number in binary format as a big-endian 16-bit binary number. This means you will change the two LEDs a total of 8 times to show your full ID number. For instance if your ID number is 5555 (binary 0001010110110011) your LEDs will blink in the following pattern (read left to right):

LED 2	0	0	0	0	1	1	0	1
LED 1	0	1	1	1	0	1	0	1

Sounds easy enough right? However there is one problem you still need to solve, how does a receiver that is watching the lights blink know when the pattern is beginning? This is actually a fundamental problem that all wireless communication systems need to solve. We will solve it in this project by adding an 8-bit *preamble* that will be sent before the 16-bit student ID number. Specifically, you will transmit the following pattern before each time you transmit the student ID (read left to right):

LED 1	0	1	0	1
LED 2	1	0	1	0

To receive your SID number, you will use your smartphone's video camera! Put your phone in 60 frames per second video capture mode, you should see your LEDs display change to the next two bytes every 3rd frame. You can use [VLC media player](#) to see if your timing is exactly correct because you can use the "e" key to advance one frame at a time during playback. This allows you to count if the LEDs are indeed changed every 3 frames.