LAB 1 – ALARM SYSTEM

TABLE OF CONTENTS

INTRODUCTION

In Lab 1, you will create a motion-triggered alarm system, using the on-board motion sensor.

Throughout this exercise, we'll build on our knowledge of the general-purpose input/output (GPIO) peripheral, adding interrupts and timed events. We'll also expand our debug capabilities and learn how to modularize and generalize our code into device drivers and applications.

OBJECTIVES

After successfully completing Lab 1, you'll be able to:

1. Implement and test a device driver to abstract a hardware peripheral such as GPIO.
2. Generate a periodic interrupt for the SysTick system timer and use the resulting time base to accurately measure delays and events, without blocking the CPU.
3. Configure an interrupt and register a callback function to respond to a change in a GPIO input.
4. Build a state machine task for a simple embedded application.
5. Use `printf()` to relay status and debug messages to the IDE console.

ACADEMIC INTEGRITY

1. Both group members must be involved from start to finish. A student who did not actively participate in the lab work cannot claim credit towards their course grade. Absence for medical or other valid reasons shall be communicated with the course instructor.
2. All lab deliverables—including demoed/submitted source code and written explanations—must be your own original work. Copying between groups and the use of generative AI are prohibited.

PROJECT SETUP

Follow the procedure introduced in prior labs to set up a new project for Lab 1:

1. Create a new STM32 project for the STM32L552ZET6Q MCU. Name your project `CEG3136_Lab1` and create a New Folder in a safe location (e.g. H: or OneDrive). Select the 'Empty' project type.
2. Download the Lab Drivers (CMSIS) from Brightspace. Move the `Drivers` folder contained within to your project folder. Refresh your project in the IDE.
3. In your project Properties, find your compiler build settings, add the following preprocessor define:
   ```
   STM32L552xx
   ```
4. Also add the following include paths:
   ```
   ../Drivers/CMSIS/Include
   ../Drivers/CMSIS/Device/ST/STM32L5xx/Include
   ```

PRELUDE: DEBUG MESSAGES

Many embedded software developers find the `printf()` function included in the C standard library an extremely useful debugging tool. Unlike personal computers, most embedded systems have limited display capabilities and must rely on alternate communication channels for debug messaging.

Serial Wire Viewer (SWV)—a data trace feature provided by certain processors in the ARM Cortex-M family—can be used to output messages to a console in the STM32CubeIDE. SWV is facilitated by a hardware module called the Instrumentation Trace Macrocell (ITM), so you will see these terms used interchangeably.

REDIRECTING THE STANDARD OUTPUT

To redirect the standard output (stdout) from functions like `printf()` to the SWV ITM (or anywhere else for that matter), all we need to do is implement the `__io_putchar()` function somewhere in our code. You can find the function prototype in the autogenerated project file `syscalls.c`. The `weak` attribute allows the programmer to override the function definition, which does nothing by default.

```
extern int __io_putchar(int ch) __attribute__((weak));
```

DEBUG.C

In the Project Explorer, under the `Src` folder, add a new source file named `debug.c`. Replace its entire contents with the code shown on the right. (Typing in code by hand helps to reinforce the learning process.)

`ITM_SendChar()` is a function provided by CMSIS that writes to a hardware register in the ITM to send data via Stimulus Port 0. If the SWV feature is disabled, the message is silently discarded.

No corresponding header file is required.

debug.c

```
1  #include "stm32l5xx.h"
2
3  // printf() support via ITM/SWV
4  int __io_putchar(int c) {
5    ITM_SendChar(c);
6    return c;
7  }
```

IMPORTANT:

- In the `#include` line, the character after `stm32` is a lowercase L, not a numeral 1.
- There are *two* underscores before `io`.

MAIN.C

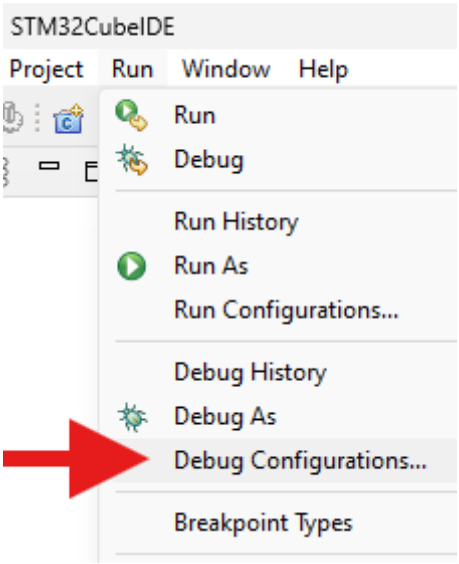
Replace the entire contents of file `main.c` by copying/pasting the following test code. Build your project and resolve any compile or link errors.

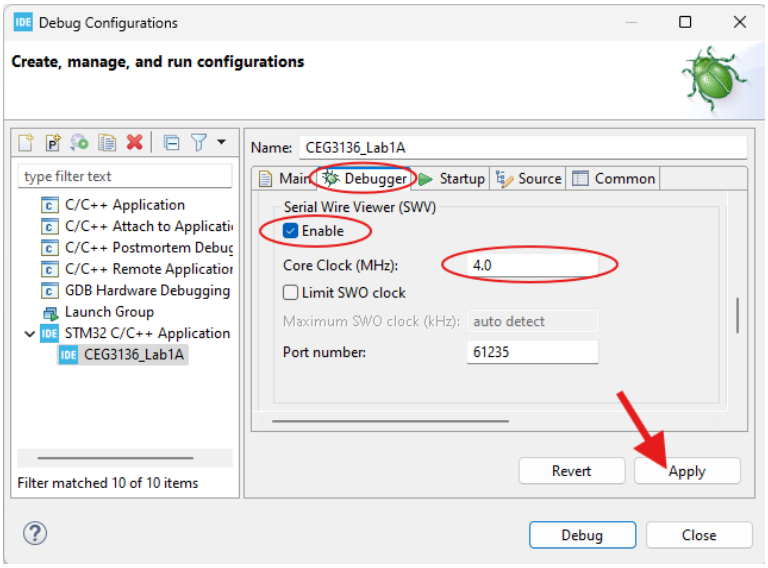
```c
#include <stdio.h>

int main(void)
{
    printf("Welcome to %c%c%c%d!\n", 67, 69, 71, 0xC40);
}
```

ENABLING THE SERIAL WIRE VIEWER

Before ITM can be used, you must enable Serial Wire Viewer (SWV) in your debug configuration:

1. Under the Run menu, select Debug Configurations.
2. Select the existing configuration or create a new one for a STM32 C/C++ Application.
3. Switch to the Debugger tab.
4. In the Debugger tab, under Serial Wire Viewer (SWV):
    a. Check the Enable box.
    b. Set the Core Clock to 4.0MHz to match the SYSCLK frequency of the MCU.
    Remember to update this setting if you ever change the MCU clock configuration!
5. Click Apply to save the changes.

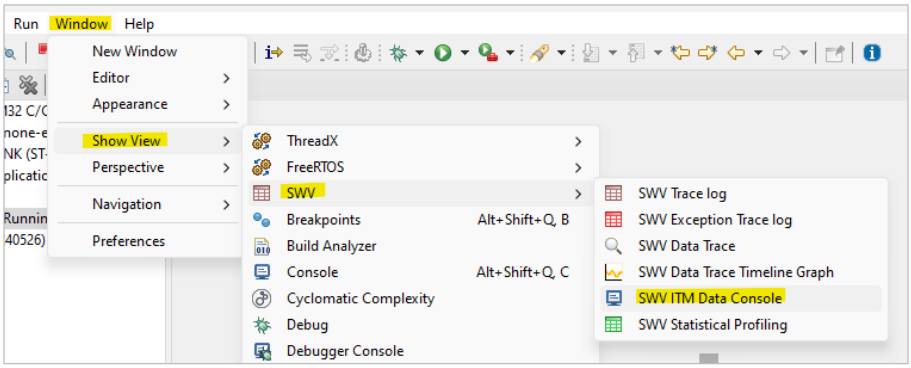




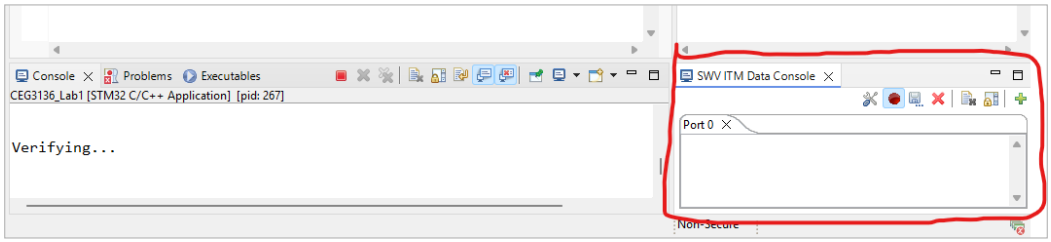
OPENING THE DATA CONSOLE

Finally, we need to open a special console so we can see the messages:

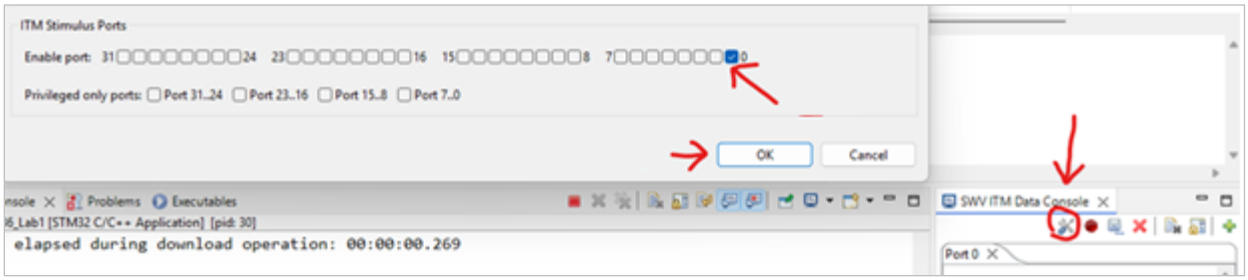
1. Run your project in Debug mode (bug icon). You must be running the program for the next steps.
2. If prompted, check the box to automatically switch to the Debug Perspective each time.
3. Under the Window menu, select Show View, SWV, SWV ITM Data Console.



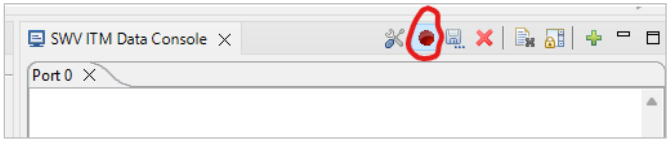
4. The new Data Console pane will appear at the bottom of your IDE. Drag its tab to the right to split the panel from the build/debug Console.



5. Click the Configure Trace icon (screwdriver and wrench). A settings window will appear.
6. Enable the checkbox to enable ITM Stimulus Port 0. Click OK.



7. Click the Start Trace icon (red circle). Trace is active when the icon's background is shaded.



TESTING DEBUG OUTPUT

Continue running your program by pressing Resume. You should see your `printf()` message from `main.c` displayed in the Data Console.

Exploration #1: How do the parameters passed to `printf()` generate the message you observed?

SIDE NOTE: UART INTERFACE

Another option for debug messaging is to leverage the universal asynchronous receiver/transmitter (UART) in the ST-Link debug interface, which can be connected to a UART peripheral in the MCU. This method is more complicated to configure and less user-friendly compared to SVW and thus won't be covered in this lab.

For processors such as the Cortex-M0+ that don't support SWV, a UART interface is often the best option for debug messaging.

PART A: BUILDING BLOCKS

In the first part of Lab 1, we'll learn how to code a *device driver*—a layer of software that abstracts a piece of hardware, hiding the details of its operation from the programmer. The resulting modularity simplifies the software development process and improves code readability.

We'll build two different drivers in this lab: SysTick and GPIO.

SYSTICK DRIVER

This driver configures the CPU system timer (SysTick) to generate a periodic interrupt. It provides method functions for accurately measuring delays and time between events.

SYSTICK.H

Under the `Inc` folder, add a new header file named `systick.h`. Replace its entire contents by typing in the code shown (below left).

SYSTICK.C

Under the `Src` folder, add a new header file named `systick.c`. Replace its entire contents by copying and pasting the code shown in mini-text (below right).

systick.h

```
1  #ifndef SYSTICK_H_
2  #define SYSTICK_H_
3
4  #include "stm32l5xx.h"
5
6  typedef unsigned int Time_t;
7  #define TIME_MAX (Time_t)(-1)
8
9  void StartSysTick();
10 void WaitForSysTick();
11 void msDelay(int t);
12 Time_t TimeNow();
13 Time_t TimePassed(Time_t since);
14
15 #endif /* SYSTICK_H_ */
```

systick.c

```
// Manage the system timer

#include "systick.h"

#define SYSTICKS 4000  // 1ms with 4MHz clock

static volatile Time_t sysTime;

void StartSysTick() {
    sysTime = 0;
    SysTick->LOAD = (uint32_t)(SYSTICKS - 1);
    SCB->SHPR[12+SysTick_IRQn] = 7 << 5;
    SysTick->VAL = 0;
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
                    SysTick_CTRL_TICKINT_Msk   |
                    SysTick_CTRL_ENABLE_Msk;
}

// Interrupt handler
void SysTick_Handler (void) {
            sysTime++;
}

// Wait for system time to change
void WaitForSysTick (void) {
    int wasTime = sysTime;
    while (sysTime == wasTime)
        // Instruction to keep CPU asleep until next interrupt
        __asm volatile ("wfi");
}

// Delay measured in milliseconds
void msDelay (int t) {
            for (int i = 0; i < t; i++)
                        WaitForSysTick();
}

// Obtain the current system time
Time_t TimeNow (void) {
            return sysTime;
}

// Calculate the elapsed system time since a previous event
Time_t TimePassed (Time_t since) {
            Time_t now = sysTime;
            if (now >= since)
                        return now - since;
            else
                        return now + 1 + TIME_MAX - since;
}
```

EXPLORING THE CODE

The SysTick driver provides the following method functions:

`StartSysTick()` – Initializes and configures the system timer to produce a tick every 1ms.

`WaitForSysTick()` – Puts the CPU to sleep and waits for the next interrupt. The `WFI` instruction waits for *any* interrupt, not only the one assigned to SysTick. However, the `while` loop condition ensures the CPU will go back to sleep if the system time has not yet advanced.

`msDelay()` – Uses the system timer to accurately measure a delay in milliseconds. This is a replacement for the counting loop–based `delay()` function used in Lab 0. However, both functions are blocking, meaning they keep control of the CPU for the entire delay time.

`TimeNow()` – Returns the current system time in milliseconds since SysTick was started.

`TimePassed()` – Calculates the elapsed time since a previous system time. It can be used to implement non-blocking delays and to measure the elapsed time between external events.

The SysTick driver also includes an interrupt handler. Interrupts are explained in Part B of this lab.

`SysTick_Handler()` – When the SysTick timer expires (once every 1ms), a hardware interrupt is generated and this function is called to increment the system time variable.

Exploration #2: How could you modify the SysTick driver to use a different tick period instead of 1ms?

Exploration #3: What is the purpose of the `else` clause of the `TimePassed()` function?

GPIO DRIVER

This driver provides a generic interface to configure, control, and observe GPIO pins. You'll need to implement 5 different method functions based on the provided prototypes. Methods pertaining to interrupts are provided for you, as they are considerably more complex.

GPIO.H

In the Project Explorer, under the `Inc` folder, add a new header file named `gpio.h`. Replace its entire contents by typing in the code shown below.

Make sure to type in the comments for `struct Pin_t`. The comments for macro `GPIO_PORT_NUM()` can be left out, but take careful note, as it will be useful later!

gpio.h

```
#ifndef GPIO_H_
#define GPIO_H_

#include "stm32l5xx.h"

// This preprocessor macro converts a GPIO register base address
// (GPIOA, GPIOB, GPIOC, etc.) to a zero-based port index (0, 1, 2, etc.)
// The address pattern can be identified from the device header file
#define GPIO_PORT_NUM(addr) (((unsigned)(addr) & 0xFC00) / 0x400)

// Structure representing a single GPIO pin
typedef struct {
    GPIO_TypeDef *port;  // GPIOA, GPIOB, etc.
    int           bit;   // Bit index 0..15
} Pin_t;

typedef enum {INPUT=0b00, OUTPUT=0b01, ALTFUNC=0b10, ANALOG=0b11} PinMode_t;
typedef enum {LOW=0, HIGH=1} PinState_t;
typedef enum {FALL=0, RISE=1} PinEdge_t;

void GPIO_Enable(Pin_t pin);
void GPIO_Mode(Pin_t pin, PinMode_t mode);
PinState_t GPIO_Input(Pin_t pin);
void GPIO_Output(Pin_t pin, PinState_t state);
void GPIO_Toggle(Pin_t pin);
void GPIO_Callback(Pin_t pin, void (*func)(void), PinEdge_t edge);

#endif /* GPIO_H_ */
```

GPIO.C

Under `Src`, add a new source file named `gpio.c`. Replace its entire contents by copying and pasting the code shown in mini-text below.

gpio.c

```
// General-purpose input/output driver

#include "gpio.h"

// ------------------------------------------------------------
// Initialization
// ------------------------------------------------------------

// Enable the GPIO port peripheral clock for the specified pin
void GPIO_Enable (Pin_t pin) {
    // ...
}

// Set the operating mode of a GPIO pin:
//   Input (IN), Output (OUT), Alternate Function (AF), or Analog (ANA)
void GPIO_Mode (Pin_t pin, PinMode_t mode) {
    // ...
}

// ------------------------------------------------------------
// Pin observation and control
// ------------------------------------------------------------

// Observe the state of an input pin
PinState_t GPIO_Input (const Pin_t pin) {
    // ...
}

// Control the state of an output pin
void GPIO_Output (Pin_t pin, const PinState_t state) {
    // ...
}

// Toggle the state of an output pin
void GPIO_Toggle (Pin_t pin) {
    // ...
}

// ------------------------------------------------------------
// Interrupt handling
// ------------------------------------------------------------

// Array of callback function pointers
// Bits 0 to 15 (each can select one port GPIOA to GPIOH)
// Rising and falling edge triggers for each
static void (*callbacks[16][2]) (void);

// Register a function to be called when an interrupt occurs,
// enable interrupt generation, and enable interrupt vector
void GPIO_Callback(Pin_t pin, void (*func)(void), PinEdge_t edge)
{
                callbacks[pin.bit][edge] = func;

                // Enable interrupt generation
                if (edge == RISE)
                  EXTI->RTSR1 |= 1 << pin.bit;
                else     // FALL
                  EXTI->FTSR1 |= 1 << pin.bit;
                EXTI->EXTICR[pin.bit / 4] |= GPIO_PORT_NUM(pin.port) << 8*(pin.bit % 4);
                EXTI->IMR1 |= 1 << pin.bit;

                // Enable interrupt vector
                NVIC->IPR[EXTI0_IRQn + pin.bit] = 0;
                __COMPILER_BARRIER();
                NVIC->ISER[(EXTI0_IRQn + pin.bit) / 32] = 1 << ((EXTI0_IRQn + pin.bit) % 32);
                __COMPILER_BARRIER();
}

// Interrupt handler for all GPIO pins
void GPIO_IRQHandler (int i) {
                // Clear pending IRQ
    NVIC->ICPR[(EXTI0_IRQn + i) / 32] = 1 << ((EXTI0_IRQn + i) % 32);

    // Detect rising edge
                if (EXTI->RPR1 & (1 << i)) {
                                EXTI->RPR1 = (1 << i);  // Service interrupt
                                callbacks[i][RISE]();   // Invoke callback function
                }

    // Detect falling edge
                if (EXTI->FPR1 & (1 << i)) {
                                EXTI->FPR1 = (1 << i);  // Service interrupt
                                callbacks[i][FALL]();   // Invoke callback function
                }
}

// Dispatch all GPIO IRQs to common handler function
void  EXTI0_IRQHandler() { GPIO_IRQHandler( 0); }
void  EXTI1_IRQHandler() { GPIO_IRQHandler( 1); }
void  EXTI2_IRQHandler() { GPIO_IRQHandler( 2); }
void  EXTI3_IRQHandler() { GPIO_IRQHandler( 3); }
void  EXTI4_IRQHandler() { GPIO_IRQHandler( 4); }
void  EXTI5_IRQHandler() { GPIO_IRQHandler( 5); }
void  EXTI6_IRQHandler() { GPIO_IRQHandler( 6); }
void  EXTI7_IRQHandler() { GPIO_IRQHandler( 7); }
void  EXTI8_IRQHandler() { GPIO_IRQHandler( 8); }
void  EXTI9_IRQHandler() { GPIO_IRQHandler( 9); }
void EXTI10_IRQHandler() { GPIO_IRQHandler(10); }
void EXTI11_IRQHandler() { GPIO_IRQHandler(11); }
void EXTI12_IRQHandler() { GPIO_IRQHandler(12); }
void EXTI13_IRQHandler() { GPIO_IRQHandler(13); }
void EXTI14_IRQHandler() { GPIO_IRQHandler(14); }
void EXTI15_IRQHandler() { GPIO_IRQHandler(15); }
```

The GPIO driver provides a total of 6 method functions.

The first 5 methods are supplied as skeleton code, which you will complete later in the lab:

- `GPIO_Enable()` – Enable the system clock to a GPIO peripheral via RCC_AHB2ENR.
- `GPIO_Mode()` – Configure the mode of a GPIO pin via GPIOx_MODER.
- `GPIO_Input()` – Return the state of a GPIO input pin by reading and testing GPIOx_IDR.
- `GPIO_Output()` – Drive a GPIO output pin to the specified state using GPIOx_BSRR.
- `GPIO_Toggle()` – Toggle the state of a GPIO output pin by modifying GPIOx_ODR.

The final method is already provided for you:

- `GPIO_Callback()` – Register a callback function for a GPIO interrupt for a specific input pin and edge direction.

The remaining code provides handlers for GPIO interrupts, which will be explored in Part B.

DRIVER TEST CODE

Before using a newly-built driver in an application, it's a good idea to test its functionality, preferably with known good code. Furthermore, knowing how a driver is intended to be used *before* coding it can make it much easier to figure out the proper implementation!

MAIN.C

Replace the entire contents of `main.c` by typing in the test code shown below. Include the comments for the `Pin_t` declarations, as you will need to refer to these in Part B.

As you may recognize, this code is performing the same task that we did in Lab 0. However, it has now been adapted to use the SysTick and GPIO drivers instead of accessing the registers directly. It also includes `printf()` statements to show the program status.

main.c

```c
#include <stdio.h>
#include "gpio.h"
#include "systick.h"

static const Pin_t BlueLED = {GPIOB,  7};  // Pin PB7  -> User LD2
static const Pin_t Button  = {GPIOC, 13};  // Pin PC13 <- User B1

#define BLINK_PERIOD 1000  // 1 second

static unsigned int loopCount = 0;

int main(void)
{
    GPIO_Enable(BlueLED);
    GPIO_Mode(BlueLED, OUTPUT);
    GPIO_Output(BlueLED, LOW);

    GPIO_Enable(Button);
    GPIO_Mode(Button, INPUT);

    StartSysTick();

    printf("Welcome to %c%c%c%d!\n", 67, 69, 71, 0xC40);

    while (1) {
        msDelay(BLINK_PERIOD / 2);
        if (!GPIO_Input(Button))
            GPIO_Output(BlueLED, HIGH);

        msDelay(BLINK_PERIOD / 2);
        GPIO_Output(BlueLED, LOW);

        loopCount++;
        printf("Loop count %u\n", loopCount);
    }
}
```

IMPLEMENTING THE GPIO DRIVER

Open `main.c` and `gpio.c` side-by-side in your IDE. To complete the GPIO driver, you'll need to define each of the 5 empty GPIO functions. The places to add code are denoted by comment with an ellipsis (...).

CODING HINTS

If you need some information to get started:

- *Exploring the Hardware* section of the Lab 0 Manual provides code snippets showing how the C programming language can manipulate the GPIO registers and includes some figures from your MCU Reference Manual. Note that the Lab 0 code was written for specific GPIO pins (PB7 and PC13). GPIO driver must do it in a generic way instead!

- Tutorial 3 teaches the basics of bitfield manipulation bitfields in C. The solution to the first example problem at the end (controlling four RGB LEDs) is partly generic.

Here are some examples of transformation from specific to generic code:

Specific code (Lab 0)	**Generic code (Lab 1 driver)**
`GPIOB->MODER`	`pin.port->MODER`
`0b11 << 14`	`0b11 << 2*pin.bit`
`GPIOC->ODR`	`pin.port->ODR`
`1 << 7`	`1 << pin.bit`
`GPIOC->IDR`	`pin.port->IDR`

The `GPIO_PORT_NUM()` macro may come in handy for implementing `GPIO_Enable()`. Refer to your MCU Reference Manual for the definition of `RCC_AHB2ENR` and Tutorial 3 for how to set a bit in C.

Don't use the `MODIFY_FIELD()` macro anywhere in the GPIO driver. Its token-pasting technique won't work with the variable `pin.bit`.

DEBUGGING YOUR CODE

Build your project, then run it with Debug to test it. This program should behave like the Lab 0 program. If not, you'll need to debug it!

You can Step Over the code in `main()` or choose to Step Into the GPIO driver function calls.

Another useful debug aid is the SFRs tab in the watch pane to the right. Find the relevant sections and registers (i.e. RCC->AHB2ENR, GPIOB/C->MODER, GPIOC->IDR, GPIOB->ODR). Step through the code and watch the register values to see if they are updated as expected.

DESIGNING FOR CONCURRENCY

Embedded systems often need to execute multiple software tasks concurrently. Despite its improved accuracy, the `msDelay()` function still suffers from the same problem as the Lab 0 `delay()` function, in that it completely occupies the CPU while the delay is being counted out.

MAIN.C

We can use two additional functions provided by the SysTick driver to implement a simple *cooperative multitasking* scheme. Update your program based on the highlighted lines below.

Exploration #4: Run your updated program in Debug mode. Watch the messages appearing on the Data Console. Make note of how much the value of `loopCount` increases between each LED state change.

main.c

```
1  #include <stdio.h>
2  #include "gpio.h"
3  #include "systick.h"
4
5  static const Pin_t BlueLED = {GPIOB,  7};  // Pin PB7  -> User LD2
6  static const Pin_t Button  = {GPIOC, 13};  // Pin PC13 <- User B1
7
8  #define BLINK_PERIOD 1000  // 1 second
9  static Time_t lastToggle;
10
11 static unsigned int loopCount = 0;
12
13 int main(void)
14 {
15     GPIO_Enable(BlueLED);
16     GPIO_Mode(BlueLED, OUTPUT);
17     GPIO_Output(BlueLED, LOW);
18
19     GPIO_Enable(Button);
20     GPIO_Mode(Button, INPUT);
21
22     StartSysTick();
23     lastToggle = TimeNow();
24
25     printf("Hello, World!\n");
26
27     while (1) {
28         if (TimePassed(lastToggle) >= BLINK_PERIOD / 2) {
29             if (!GPIO_Input(Button))
30                 GPIO_Toggle(BlueLED);
31             else
32                 GPIO_Output(BlueLED, LOW);
33             lastToggle = TimeNow();
34             printf("Loop count %u\n", loopCount);
35         }
36
37         loopCount++;
38         WaitForSysTick();
39     }
40 }
```

Implement a non-blocking delay by repeatedly checking the time

Use this opportunity to test your toggle function

Code for another task could be inserted here and would run once every millisecond

PART B: APPLICATION

We will now use our newly built and tested drivers to create a simple alarm system app.

REQUIREMENTS

IMPORTANT: *Read this section carefully! Your lab grade will depend in part on how well your final program conforms to this specification. If you have any questions, ask your instructor or TA for clarification.*

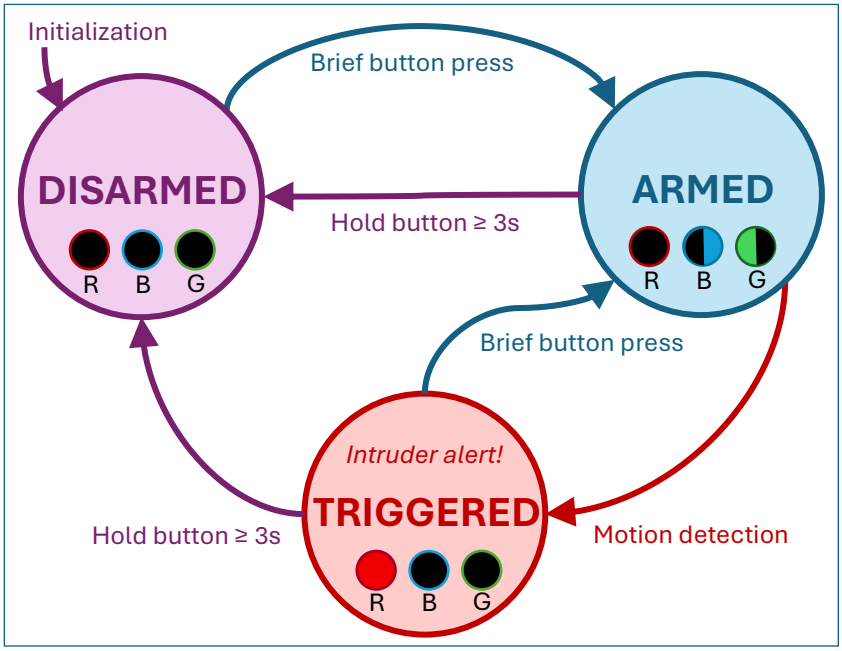
EXTERNAL COMPONENTS

Refer to your Lab User's Guide for information about these components and their MCU pin connection.

- **Red/Blue/Green User LEDs** on the white NUCLEO development board
- **Motion sensor module** on the Leafy main board
- **E-STOP push-button** on the Leafy main board (not debounced in hardware!)

EXPECTED BEHAVIOUR

The requirements for the alarm system are described in the form of a state machine.



In the DISARMED state:

- No LEDs are illuminated.
- Motion sensor is ignored.

In the ARMED state:

- Green/blue LEDs alternate illumination every second.
- Motion detection shall cause transition to the TRIGGERED state.

In the TRIGGERED state:

- Red LED is illuminated.

Push-button control:

- Briefly pressing the button (held for less than 2 seconds) shall cause transition to the ARMED state. This action can be used either to arm the system or to clear a triggered alarm without disarming.
- Holding the button for 3 seconds or more shall cause a transition to the DISARMED state. To ensure a responsive user experience, this transition shall occur *before* the button is released.

Status messaging:

- Each state transition shall produce an appropriate status message on the data console that includes the current system time, the new state, and the reason for the transition.

LEARNING ABOUT INTERRUPTS

In computing, an *interrupt* is a hardware signal sent to the CPU that causes it to temporarily suspend regular program execution to take care of something of greater urgency. Code that deals with an interrupt is called an *interrupt handler* or *interrupt service routine* (ISR).

Computers use interrupts for a wide variety of purposes, including interfacing with peripherals such as touchscreens, storage, and wireless networking. A high priority interrupt ensures that the motion of your mouse cursor remains smooth, even when the apps on your laptop are running slowly.

SYSTICK INTERRUPT

Our SysTick driver includes an interrupt handler for the SysTick system timer that increments the system time counter. The handler function uses a predetermined name that associates it with the interrupt source when the project is built.

```
static volatile Time_t sysTime = 0;
…
void SysTick_Handler (void) {
       sysTime++;
}
```

GPIO INTERRUPTS

We can also generate an interrupt when a specific GPIO pin experiences a change in its input state. A transition from low to high is called a *rising edge*. A transition from high to low is called a *falling edge*. Either or both transitions can be monitored for a given input pin to trigger an interrupt.

Our GPIO driver includes a set of interrupt handlers to provide such functionality. The application code can use `GPIO_Callback()` to register its own *callback function*, which the driver's interrupt handler will call whenever the specified GPIO pin experiences the specified state transition (`RISE` or `FALL`).

Here's an example of a callback for a GPIO input observing a signal from the motion sensor. The callback registration is done once during initialization. After that point, `CallbackMotionDetect()` will be automatically called whenever there is new motion detected by the sensor.

```
static const Pin_t Motion = …
…
void Init_Alarm (void) {
    …
    GPIO_Callback(Motion, CallbackMotionDetect, RISE);
}
…
void CallbackMotionDetect (void) {
    // ...
}
```

STRUCTURING YOUR CODE

ALARM.H

In the Project Explorer, under the `Inc` folder, add a new header file named `alarm.h`. Replace its entire contents by typing in the code shown on the right.

`Init_Alarm()` and `Task_Alarm()` contain the app's initialization code and task code respectively. In a later step, you will implement them inside `alarm.c`.

alarm.h

```
1  #ifndef ALARM_H_
2  #define ALARM_H_
3
4  void Init_Alarm();
5  void Task_Alarm();
6
7  #endif /* ALARM_H_ */
```

MAIN.C

Replace the entire contents of file `main.c` by typing in the code shown to the right.

If you want to preserve your Part A code in a working state without creating a new project, you can split your `main.c` file into two parts and use preprocessor directives to select which part is currently active, as shown below.

`main.c` – Code swap example

```
#if 1  // 1 for Part B, 0 for Part A

// ... Part B code

#else

// ... Part A code

#endif
```

main.c

```
1  #include <stdio.h>
2  #include "systick.h"
3  #include "alarm.h"
4
5  int main(void)
6  {
7      // Initialize apps
8      Init_Alarm();
9
10     StartSysTick();
11
12     while (1) {
13         // Run apps
14         Task_Alarm();
15
16         WaitForSysTick();
17     }
18 }
```

In any case, the driver test code from Part A is *not required* as a Lab 1 demo/report deliverable.

As you can see, our new `main()` function is quite simple:

- `Init_Alarm()` is called once whenever the program is restarted.
- `Task_Alarm()` is called once every millisecond in an endless loop.

In later labs we'll have multiple apps running concurrently, each with their own `Init_xxx()` and `Task_xxx()` functions that will be called in series. This technique keeps our code modular and extensible, as the features of our embedded system continue to grow!

ALARM.C

In the Project Explorer, under the `Src` folder, add a new source file named `alarm.c`. Replace its entire contents by copying and pasting the skeleton code shown below.

`alarm.c` – Skeleton code

```
// Alarm system app

#include <stdio.h>
#include "alarm.h"
#include "systick.h"
#include "gpio.h"

// GPIO pins
// ...

static enum {DISARMED, ARMED, TRIGGERED} state;

// Constants
// ...

// Variables
// ...

// Declare interrupt callback functions
static void CallbackMotionDetect();
static void CallbackButtonPress();
static void CallbackButtonRelease();

// Initialization code
void Init_Alarm (void) {
    // ...
}

// Task code (state machine)
void Task_Alarm (void) {
    switch (state) {

    case DISARMED:
        // ...

        if ( /* condition */ ) {
            // ...
            state = ARMED;
            printf("... at time %u\n", TimeNow());

            // ...
        }

        break;

    case ARMED:
        // ...

    case TRIGGERED:
        // ...
    }
}

void CallbackMotionDetect (void) {
    // ...
}

void CallbackButtonPress (void) {
    // ...
}

void CallbackButtonRelease (void) {
    // ...
}
```

Complete the code according to these guidelines:

1. Declare GPIO pins following the style in Part A, including keywords and comment.
2. `#define` constants used in your code. Add a same-line comment with a few descriptive words and units of measure.
3. Declare global variables used by your code (e.g. flags and timestamps). Use `static` to keep them private to this compilation unit. Make sure each variable is initialized before it can be read.
4. `Init_Alarm()` – Use your GPIO driver to configure each GPIO pin and register the callback functions.
5. `Task_Alarm()` – Build your state machine here using the skeleton code as a template.
   - Code immediately after the case statement will *run on every 1ms tick* while in that state, so probably not the best place for `printf()`.
   - State transitions and other actions can be initiated by checking a flag, observing the state of a pin, and/or how much time has elapsed.
   - When transitioning between states, consider what preparation might be needed.
   - Don't forget to include `break` statements!
6. `Callback…()` – Interrupt handlers should have short execution times. Consider setting a flag for the task code to check or recording a timestamp. (Remember to clear flags in your task code!) Avoid using `printf()` in these functions.
7. `CallbackButtonRelease()` – Detect a "brief" button press here, conditionally setting a flag. You will also need to implement software debouncing (e.g. ignore any button release that occurs within 50ms of a button press).

TESTING AND DEBUGGING

Once you've completed the code and resolved any build errors, run your project on the MCU in Debug mode.

Test your app by making sure that each of the requirements are met and that no abnormal behaviour is seen. If something isn't working as you expect, use the debugging skills you've developed so far in these labs to track down the problem!

Exploration #5: How could you accurately measure how long the Green and Blue LEDs each remain illuminated while in the ARMED state?

KNOWLEDGE EXTENSION

To further your understanding of the material, choose at least one of the following options. Complete both items to score a bonus on your lab grade.

1. **Bouncy buttons!** Use an oscilloscope to observe the push-button signal as the button is pressed and released. Modify your code so that it does *not* debounce the button anymore and retest your alarm system. What happens now when you try to disarm the system from the TRIGGERED state? In your demo, show your lab TA the waveform captures and briefly explain the purpose of debouncing.
2. **Do something cool!** Using your lab work as a starting point, extend its functionality to do something new and interesting. Demonstrate your creation to your lab TA and classmates.

DELIVERABLES

LAB DEMO

After you've completed and tested your project, you will demonstrate your final program to your lab TA.

Your demo will be graded on how well your program conforms to the requirements described in the lab manual. You will be asked to confirm that your source code is entirely your original work and that both group members actively participated in the lab work.

LAB REPORT

Following your demo, your group will submit two files to Brightspace under **Lab 1 Report**:

- A .pdf file containing the following:
  - A title page with the course code, lab number, lab section, student names, and student IDs;
  - Source code authored by your group (i.e. 5 functions in `gpio.c` and entire `alarm.c`); and
  - Written answers to the Exploration Questions below.
- A .zip file containing your complete project used for your demo:
  - In the STM32CubeIDE, right click your project and select Clean Project.
  - Exit the IDE completely.
  - Compress your entire project folder into a single .zip archive.

EXPLORATION QUESTIONS

Keeping your answers brief, discuss these exploration questions in your lab report:

1. What message is output by this code? Explain how it is generated in terms of C format specifiers and the additional parameters passed.

```
printf("Welcome to %c%c%c%d!\n", 67, 69, 71, 0xC40);
```

2. Explain how you would modify the SysTick driver to configure a tick period of 250µs instead of 1ms. Provide a general formula and calculation for both tick periods based on the SYSCLK frequency.

3. Examine the code for the `TimePassed()` method function of the SysTick driver.
   a. Explain the purpose of the `else` clause.
   b. If the clause were removed, calculate the earliest time from program start when the function might produce an erroneous result.
   c. Explain how you could modify the code to expose the problem sooner.
   Hint: The modification can be made elsewhere in the same source file.

4. In the driver test code (`main.c`) for Part A, compare the original code using `msDelay()` with the updated code using `TimePassed()`.
   a. How long does each iteration of the `while` loop take to execute in each case?
   b. By how much does `loopCount` increase between consecutive `printf()` messages?
   c. Explain the potential benefit of the latter approach when designing an embedded system.

5. Propose a test method to verify that the Green and Blue LEDs are toggling at the required rate (each illuminated for 1 second) while the alarm system is in the ARMED state.

6. **Bonus:** Explain why it is important for interrupt handlers to have a very short execution time. Describe a situation (can be unrelated to this lab exercise) where an embedded system might malfunction if this is guideline is not met.