Exercise 5

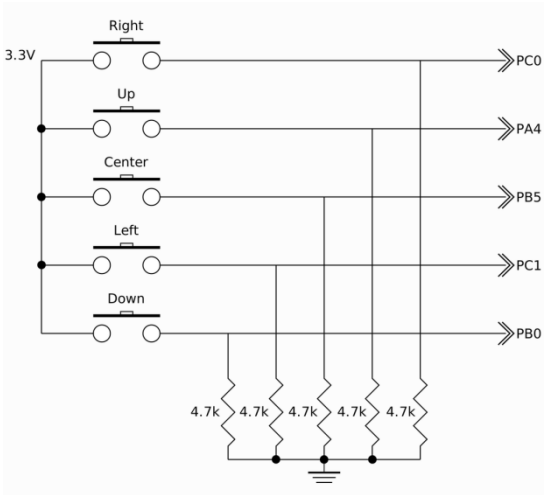
This exercise consists of two parts. First you will detect when the joystick is being interacted with and print the interaction to PuTTY. Second you will control the color of the RGB LED depending on which way the joystick is being pushed.

You will need the documents ” STM32F302x8_Reference_Manual” and ” STM32F302x8_Datasheet” when you do the next couple of exercise which are available on DTU Inside and will be referred to as [RM] and [DS], respectively, in the following. When you first open the documents, you may be a bit intimidated by the fact that [RM] is 1080(!) pages long, but do not fret - we will only be needing a few sections from it.

Exercise 5.1 - Detecting a Joystick Interaction

You will need the ”General-purpose I/Os (GPIO)” chapter (pp. 155) from [RM] when doing this exercise.

The STM32F302R8 microcontroller has 51 *General Purpose Input/Output* (GPIO) pins all of which are available to us on the STM32 Nucleo development board we are using. The pins are controlled through 5 I/O ports (labelled A, B, C, D, and F) with each port controlling up to 16 pins. Each GPIO pin can be used either as input or output and most of them are connected to on-chip peripheral functions such as timers or serial communication devices (see alternate functions in table 15 of [DS], pp. 45).



The joystick on the mbed expansion board is connected to pins PC0 (right), PA4 (up), PB5 (center), PC1 (left), and PB0 (down). The schematic for the connections is shown to the right.

Each I/O port is controlled through a number of registers:

Register	Address Offset	Size [Bits]	Description
GPIOx			Port base address
GPIOx->MODER	0x00	32	Mode
GPIOx->OTYPER	0x04	16	Output type
GPIOx->OSPEEDR	0x08	32	Output speed
GPIOx->PUPDR	0x0C	32	Pull-up / pull-down
GPIOx->IDR	0x10	16	Input data
GPIOx->ODR	0x14	16	Output data
GPIOx->BSRR	0x18	32	Bit set/reset
GPIOx->LCKR	0x1C	32	Configuration lock
GPIOx->AFR[0,1]	0x20-0x24	32+32	Alternate function
GPIOx->BRR	0x28	16	Bit reset

For input you will need to configure MODER and PUPDR while you can read the pin state from IDR. For output you need to set MODER, OTYPER, and OSPEEDR for configuration and ODR to set pin states.

The following code snippet shows how to set pin PA0 to input and PA1 to output:

```
RCC->AHBENR |= RCC_AHBPeriph_GPIOA; // Enable clock for GPIO Port A

// Set pin PA0 to input
GPIOA->MODER &= ~(0x00000003 << (0 * 2)); // Clear mode register
GPIOA->MODER |=  (0x00000000 << (0 * 2)); // Set mode register (0x00 - Input,
    0x01 - Output, 0x02 - Alternate Function, 0x03 - Analog in/out)
GPIOA->PUPDR &= ~(0x00000003 << (0 * 2)); // Clear push/pull register
GPIOA->PUPDR |=  (0x00000002 << (0 * 2)); // Set push/pull register (0x00 -
    No pull, 0x01 - Pull-up, 0x02 - Pull-down)

// Set pin PA1 to output
GPIOA->OSPEEDR &= ~(0x00000003 << (1 * 2)); // Clear speed register
GPIOA->OSPEEDR |=  (0x00000002 << (1 * 2)); // set speed register (0x01 - 10
    MHz, 0x02 - 2 MHz, 0x03 - 50 MHz)
GPIOA->OTYPER &= ~(0x0001 << (1)); // Clear output type register
GPIOA->OTYPER |=  (0x0000 << (1)); // Set output type register (0x00 -
    Push pull, 0x01 - Open drain)
GPIOA->MODER &= ~(0x00000003 << (1 * 2)); // Clear mode register
GPIOA->MODER |=  (0x00000001 << (1 * 2)); // Set mode register (0x00 - Input,
    0x01 - Output, 0x02 - Alternate Function, 0x03 - Analog in/out)

uint16_t val = GPIOA->IDR & (0x0001 << 0);
GPIOA->ODR |= (0x0001 << 1); //Set pin PA1 to high
```

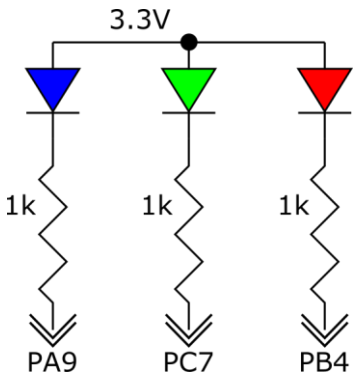
Notice the first line which sets a register we haven't talked about yet. This line enables the clock for port A and is necessary for the GPIO pins to function. You always have to remember to enable the clock line whenever you use a new I/O port otherwise nothing will work. Have a look in `Headers/SPL/inc/stm32f30x_rcc.h`, line 425 onwards to see what can be enabled.

The purpose of using AND operations followed by OR operations when setting registers is to make sure that we only change the settings for the pin we are interested in. If you want to configure all the GPIO pins of a port simultaneously you can just use a normal "equal" sign.

Using the GPIO chapter from [RM] and the above information, please do the following:

- Find out what the default modes of the GPIO pins are after a reset.
- Find out if the signal lines are high or low when the joystick is pushed in a certain direction.
- Create a function, you can call it `readJoystick()`, that returns the state of the joystick. It should return the following format:
  bit 0: 1 if the joystick is pressed up, 0 otherwise
  bit 1: 1 if the joystick is pressed down, 0 otherwise
  bit 2: 1 if the joystick is pressed left, 0 otherwise
  bit 3: 1 if the joystick is pressed right, 0 otherwise
  bit 4: 1 if the joystick is pressed center, 0 otherwise
  bit 5 - 7: 0
- Write the current joystick direction to PuTTY. Note: You should only write the direction when it is changed.

Exercise 5.2 - Controlling the RGB LED

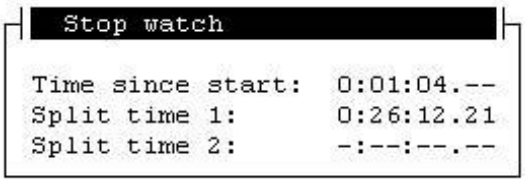


The RGB LED is connected to the mbed expansion board as is shown in the figure to the right. It consists of three individual LEDs (red, green, and blue) that are all placed in a single package. By selectively turning on different combinations of LEDs a total of 7 colors (red, green, blue, cyan, magenta, yellow, and white) can be created. Red is connected to PB4, green to PC7, and blue to PA9.

- Write a function that turns the LED to a specific color depending on the argument. You can call it `setLed(...)`.
- Write a program that lights up the LED in different colors depending on which direction the joystick is pressed.

Exercise 6

In this exercise we will be taking a look at timers. You will create a stopwatch with two split times, all of which will be shown in PuTTY:



For this exercise you will need to consult the "General-purpose timers (TIM2/TIM3/TIM4)" chapter (pp. 550) of [RM]. The chapter on interrupts is unfortunately quite underwhelming providing very little useful knowledge, so you'll have to make do with what we tell you in this guide. More information can be found in the document ”STM32F302x8_Programming_Manual”, chapter 4.3, if you are interested (available on DTU Inside).

Exercise 6.1 - Timers

The STM32 chip has several timers that can be used for different purposes such as PWM and general time keeping. We will be focusing on timer 2 and operating it in up-counting mode where the timer counts up to a reload value. When the reload value is reached an interrupt is generated and the timer starts over. The timer has a prescaler which divides the input clock (64 MHz) with (n + 1) where n = [0...65535]. If you look at the table below (or in section 21.4 of [RM]) you will see that each timer is associated with a lot of different registers, but luckily, we only need to deal with 5 of them, namely `CR1`, `ARR`, `PSC`, `DIER`, and `SR`.

Register	Address Offset	Size [Bits]	Description
TIM2			Port base address
TIM2->CR1	0x00	16	Primary Configuration
TIM2->CR2	0x04	32	Secondary Configuration
TIM2->SMCR	0x08	32	Slave Mode Control
TIM2->DIER	0x0C	32	DMA/Interrupt Enable
TIM2->SR	0x10	32	Status
TIM2->EGR	0x14	32	Event Generation
TIM2->CCMR[1,2]	0x18-0x1C	32+32	Compare Mode
TIM2->CCER	0x20	32	Compare Enable
TIM2->CNT	0x24	32	Counter
TIM2->PSC	0x28	16	Prescaler
TIM2->ARR	0x2C	32	Auto-reload
TIM2->CCR[1,..,4]	0x34-0x40	32+32+32+32	Capture/Compare
TIM2->DCR	0x48	16	DMA Control
TIM2->DMAR	0x4C	16	DMA Address

To configure the timer, you will have to do the following:

1. Write to TIM2->CR1 to disable the timer and configure the mode.
2. Write to TIM2->ARR to set the reload value.
3. Write to TIM2->PSC to set the prescaler.

4. Configure the timer interrupt.
5. Write to TIM2->CR1 to enable the timer and begin counting.

The timer period is given by the following equation:

$$Upcounting\ Mode\ Timeout\ Period = \frac{(1 + Reload\ Value) \cdot (1 + Prescale)}{System\ Clock\ Frequency}$$

If you want an interrupt every microsecond you could use:

$$Reload\ Value = \frac{Timeout\ Period \cdot System\ Clock\ Frequency}{1 + Prescale} = \frac{1 \times 10^{-6}\text{s} \cdot 64 \times 10^{6}\text{Hz}}{1 + 0} - 1 = 63$$

For this exercise we will need the interrupt to be triggered every 100th second to have an accurate stop watch. The serial communication is much too slow to output this in real time, so you should only output the 1/100 seconds information when the clock is stopped.

- Configure timer 2 with the following settings (CR1):
  No UIFREMAP remapping,
  No clock division (CKD),
  Non-buffered auto-reload preload (ARPE),
  Edge-aligned up-counting mode (CMS + DIR),
  One-pulse mode disabled (OPM),
  Any update request source (URS),
  Update events enabled (UDIS).
  Hint: see pp. 596-597 in [RM].
- Set the wanted pre-scale and reload value.
- Enable the timer (EN).

This section of code can be used as inspiration:

```
RCC->APB1ENR |= RCC_APB1Periph_TIM2; // Enable clock line to timer 2;
TIM2->CR1 = 0x...; // Configure timer 2
TIM2->ARR = 0x...; // Set reload value
TIM2->PSC = 0x...; // Set prescale value
...
```

Notice the first line; just like with the GPIO pins we have to enable the clock line to the timer peripheral before it will function.

Next, we have to enable the interrupt to actually use the timer for anything. This is done by writing:

```
...
TIM2->DIER |= 0x0001; // Enable timer 2 interrupts
...
```

But we also need to enable the interrupt in the *Nested Vectored Interrupt Controller* (NVIC) which actually handles the interrupts. This is achieved by writing:

```
NVIC_SetPriority(TIM2_IRQn, priority); // Set interrupt priority
NVIC_EnableIRQ(TIM2_IRQn);             // Enable interrupt
```

The TIM2_IRQn value equals 28 and represents the timer 2 peripheral in the NVIC subsystem. Other valid interrupts can be found in `Headers/SPL/inc/stm32f30x.h`, line 167 onwards. The first line sets the interrupt priority which determines what will be done first if multiple interrupts are triggered simultaneously. The priority is a 4-bit number (0-15) with lower values meaning higher priority. Next, we enable the interrupt by writing to the Interrupt Set Enable register. An interrupt can similarly be disabled by writing to the Interrupt Clear Enable register (`NVIC_DisableIRQ(TIM2_IRQn)`). It is also possible to disable all interrupts using `__disable_irq()` and re-enable them using `__enable_irq()`.

- Activate the timer 2 interrupt as described above.

The final step of configuring our timer is to tell the STM32 what to do whenever the timer interrupt is triggered. Every peripheral of the STM32 capable of generating an interrupt has a specific function name pre-allocated in the memory of the microcontroller. All we have to do is to create a function with the same name and put our code in it. For timer 2, it looks like this:

```
void TIM2_IRQHandler(void) {
...  //Do whatever you want here, but make sure it doesn't take too much
     time!
    TIM2->SR &= ~0x0001; // Clear interrupt bit
}
```

Notice the line: `TIM2->SR &= 0x0001;`. Every time the interrupt is triggered it sets a bit in memory to 1 which then tells the microcontroller to run the interrupt code. However, if nobody resets that specific bit back to 0, the interrupt code will be run again and again indefinitely! We therefore manually have to clear the interrupt bit at the end of our function - hence the specified line of code.

A list of names for all the possible interrupt functions can be found in `ASM Sources/src/startup_stm32f30x.S`, line 233 onwards.

At this point you should be good to have a go at making your stop watch!

- Configure timer 2 to generate an interrupt at 100 Hz at a high priority.
- Configure the interrupt update function such that it updates a structure containing hours, minutes, seconds, and hundredths of seconds.
  Note: The variables should be made global and volatile - do you know why?
- Output the time to PuTTY every time the second variable is changed.
  Note: Do NOT call printf() from the interrupt handler. It is WAY to slow!
- Use the joystick to create a stop watch with two split times:
  Center = start/stop
  Left = split time 1
  Right = split time 2
  down = stop clock and set time to 0:00
  Note: When you copy a structure containing h:m:s:hs an interrupt may occur during copy. You should disable the interrupt during copy to avoid this.

Exercise 6.2 – Serial Read

Controlling the stopwatch using the joystick is pretty cool, but not very intuitive. The next step is therefore to control it from the terminal!

To read from the terminal you can use the command `uart_getc()`. This function waits for an input byte from the UART and then returns it. Note that it will wait forever if nothing is written to the UART!

- Create a function that uses `uart_getc()` to read a specified number of characters into a byte array and then returns the array. The last element of the array should always equal 0x00, regardless of what is written to the UART.

If you run this function, you should see characters appear in PuTTY when you strike keys on the keyboard. After the specified number of characters have been written things should return to normal. If you try printing the array using `printf("%s",array)` you should see the text you wrote being re-written to the terminal. This works because a byte array ending with a zero is treated as a string by C.

To avoid being limited by a fixed number of characters we should update the function.

- Update the function to return if the UART receives a carriage return character (0x0D), i.e., when the 'enter'-key is pressed. Note: you should also set the corresponding element in the array to 0x00.

Now the function will return early if the user presses enter.

At this stage we have the ability to send text to and from the PC. Next, we need to interpret the text so it can be used for user input. An easy way to do this is by using `strcmp()`. This function takes two strings and returns 0 if and only if the two strings are identical (ignoring case differences).

- Create a function that returns a different number depending on whether the user inputs "start", "stop", "split1", "split2", "reset", or "help". You may add other keywords if you like.
- Use the function that you've just created to control your stopwatch. Note: Because the `uart_getc()` function waits indefinitely for an input the screen will not update except when the user presses enter.

Now you have your very own keyboard controlled stopwatch! However, if one of your friends who haven't taken this course wants to use it, they'll have no idea how it works (unless you tell them, of course).

- Create a function that prints out a simple user guide to PuTTY.
- You should call this function at the start of your program and whenever the user inputs "help".
- Create another function that prints out an abbreviated list of commands and call it every time the user inputs an un-recognized command.

There you go: a terminal based stopwatch that anybody will be able to use without introduction!