Wolfson School of
Mechanical, Electrical and Manufacturing Engineering

Loughborough University

WSC055 Position Sensing Using a Microcontroller
Design Portfolio

Name: [Martin Kozevnikov]
ID Number: [B525785]

This portfolio contains outputs that are generated as a result of completing laboratory sessions.

Note: When asked for code, only insert an excerpt of your code for the specific functions requested. Ensure code is correctly annotated with your own description of function.

1. Digital Outputs via GPIO Interfacing

1.1. Initialisation Code

From Laboratory 1: Task 2.2. Insert below the code used to initialise the GPIO ports

2/3

```
	//Enable clock on GPIO port E
	RCC->AHBENR |= RCC_AHBENR_GPIOEEN;

	// Define mode for each
	// GPIOE is a structure defined in stm32f303xc.h file
	GPIOE->MODER |= 0x55550000;	// Set mode of each pin in port E
to OUTPUT MODE
	GPIOE->OTYPER &= ~(0x0000FF00); // Set output type for each pin
required in Port E for PUSH/PULL configuration
	GPIOE->PUPDR &= ~(0xFFFF0000); // Set Pull up/Pull down resistor
configuration for Port E to a no pull up/no pull down mode.
```

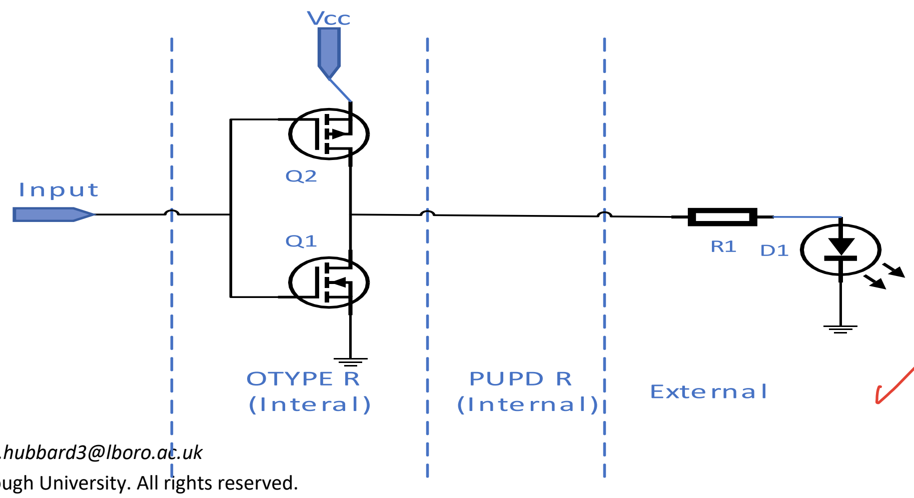
→ which pins are required.

Correct code. Ensure comments describe the specific functions.

1.2. Circuit Diagram

From Laboratory 1: Task 2.1. Sketch a circuit diagram showing the main internal and external components for a single LED connected to a GPIO pin, based on your configuration choices.

3/3



matches config.

Wolfson School of
Mechanical, Electrical and Manufacturing Engineering

1.3. Write Values

From Laboratory 1: Task 2.3. Include the code used to change the values of the LEDs.

specify outside loop. ?

```
while (1) // infinite loop

	{
	//Binary counter

	int myHexAddress=0x00; //starting from appropriate bits

	for(int x=0; x<255; x++)
	{
		myHexAddress = myHexAddress++;   // increment by 1

		GPIOE->BSRRH = 0xFF00;           //BSRRH turns bits
low > turn everything off
		GPIOE->BSRRL = myHexAddress*256; //BSRRL turns bits
high > pin 8 on
		delay(1000000); //set a variable delay
	}
```

3/4

}

2. Analogue Inputs

2.1. Initialisation Code

From Laboratory 2: Task 2.1. Insert below the sub-function created to initialise the onboard ADC peripheral.

This line has no function!

```
void SubFunction(void){
	//ADC
	//Control Register
	//Start voltage regulator
	ADC1->CR |= 0x00000000;
	ADC1->CR |= 0x10000000;
	//Wait until voltage regulator has started
	int x = 0;
	while(x<100){
		x++;
	}

	//Start calibration
	ADC1->CR |= 0x80000000;
	while((ADC1->CR & 0x80000000)!=0); //AND-ing value from register
with second value and waiting until it becomes zero.
	//GPIOE->BSRRL = 0x0100; // testing point

	//Enable clock to ADC
	RCC->CFGR2 |= RCC_CFGR2_ADCPRE12_DIV2;
	RCC->AHBENR |= RCC_AHBENR_ADC12EN;
	ADC1_2_COMMON->CCR |= 0x00010000;
```

Dr. Peter Hubbard - *p.d.hubbard3@lboro.ac.uk*

```
	//Enable clock on GPIOA - Selected pin 0 on port A to be analogue
out ADC1
	RCC->AHBENR |= RCC_AHBENR_GPIOAEN;


	//Set GPIOA as analogue mode
	GPIOA->MODER |= 0x00000002; // Set mode of each pin in port A/
sets second to last bit as a high (doesn't touch anything else)
	GPIOA->MODER &= ~(0x00000001); //  Sets only the last bit as a
0(i.e doesn't touch anything else)


	//Configuration register
	ADC1->CFGR |= 0x0010; //???? ???? ???1 ???? set specific bit to 1
	ADC1->CFGR &= ~(0x2028);  //??0? ???? ??0? 0??? set specific bit
to 0
```

↳ what does this set ?

```
	//Multiplexing - set L bits to 0 and set SQ1 bit to 1 as it's
input one in the PIN table
	ADC1->SQR1 |= 0x00000040; // Set first SQR1 channel bit to a 1
	ADC1->SQR1 &= ~(0x0000078F); // Set L's=0 and Rest of SQE1
channels bits to a 0


	// Setting the Sample time register to 7.5 ADC clock cycles
	ADC1->SMPR1 |= 0x00000018; // sets 2 LSBS of SMP1 to a 1
	ADC1->SMPR1 &= ~(0x00000100); // sets MSB of SMP1 to a 0

	//Enable ADC - ADEN in CR to 1
	ADC1->CR |= 0x00000001;


	// Wait for ADRDY flag
	while(!ADC1->ISR & 0x00000001); //AND-ing value from register
with second value and waiting until it becomes zero.
	//GPIOE->BSRRL = 0x0800; // testing the ADRDY flag bit being
initialised

// sampling is done in the interrupt loop
// reading is done in the interrupt loop
}}
```

5/6

✓ Generally good!

2.2. Analogue Read

From Laboratory 2: Task 2.2. Include the 3 or 4 lines of code used to read from the ADC and change the values of the LEDs.

```
while(1){
            //Start Sampling from ADC
            ADC1->CR |= 0x00000004; //sets the 3rd bit (ADSTART) to a
high in control register
            while(!ADC1->ISR & 0x00000004); //wait for the EOC flag to
go high in the ISR

            //read from Data Register
            ADC1ValueNew = ADC1->DR; // ADC value becomes the variable
for data-reading

            GPIOE->BSRRH = 0xFF00; // Set values to low
            GPIOE->BSRRL = ADC1ValueNew<<8; //shifting by 8 bits from
LSB to MSB section

      }
```

4/4

2.3. Observations

From Laboratory 2: Task 2.3. Comment on the quality of the bitwise counter value observed.

- The counter value seems to be preserved overall, however the last, most significant bit sometimes doesn't get initialised or becomes slightly dim.
- Some irregular and infrequent noise is present at points where LEDs change value
- Cause of noise may be due to sampling the ADC which may introduce quantisation and/or sampling errors.

2/4

Any offset present?

3. Timer-based interrupts

3.1. Timing Calculations

From Laboratory 1: Task 3.1. Show the calculation used to decide a value of PSC and ARR for an 'Update' interrupt to occur once a second.

$$TTR = \frac{(PSC+1)}{SysClock}(ARR+1)$$

Where TTR = Timer duration

ARR = Auto Reload register value (resolution)

PSC = Pre-scalar

$$\frac{(799+1)}{8*10^6}(9999+1) = 1s = TTR$$

3/3

Wolfson School of
Mechanical, Electrical and Manufacturing Engineering

Loughborough University

3.2. Initialisation Code

From Laboratory 1: Task 3.2-3.3. Insert below the lines of code created to initialise an onboard timer to generate an interrupt.

```
// Initialisation
//The timer action is set in motion with the 'enable' command
      TIM3->CR1 |= TIM_CR1_CEN;

      TIM3->DIER |= TIM_DIER_UIE; // Set DIER register to watch out for
an 'Update' Interrupt Enable (UIE) – or 0x00000001

      NVIC_EnableIRQ(TIM3_IRQn); // Enable Timer 3 interrupt request in
NVIC

      void TIM3_IRQHandler(); //calling the interrupt

      while(1);

int counterAddress = 0x00;


// Below code is the actual interrupt action, external to main


void TIM3_IRQHandler()
{
      if ((TIM3->SR & TIM_SR_UIF) !=0) // Check interrupt source is
from the 'Update' interrupt flag
      {
            //...INTERRUPT ACTION HERE
            GPIOE->BSRRH = 0xFF00; //set bits to low
            counterAddress++; // increment by 1
            GPIOE->BSRRL = counterAddress*256; // shift from LSB
section to MSB section
      }
      TIM3->SR &= ~TIM_SR_UIF; // Reset 'Update' interrupt flag in the
SR register
}
```

2/3

This is unnecessary as it is done automatically via the NVIC settings.

3.3. Interrupt Service Routine

From Laboratory 2: Insert below the ISR created to write the analogue voltage to the analogue input

```
//Write values to DAC using interrupt
int counterAddress = 0x00000000;
void TIM3_IRQHandler()
{
      if ((TIM3->SR & TIM_SR_UIF) !=0) // Check interrupt source is
from the 'Update' interrupt flag
      {
            //...INTERRUPT ACTION HERE
                  DAC1->DHR8R1 = counterAddress; //write to the DAC
                  counterAddress++; //increment by 1
```

a bit over-explaining.

```
    TIM3->SR &= ~TIM_SR_UIF; // Reset 'Update' interrupt flag in the
SR register
}
```

3.4. Observations

From Laboratory 1: briefly comment on the two methods used to manage the timing aspect of the LED control

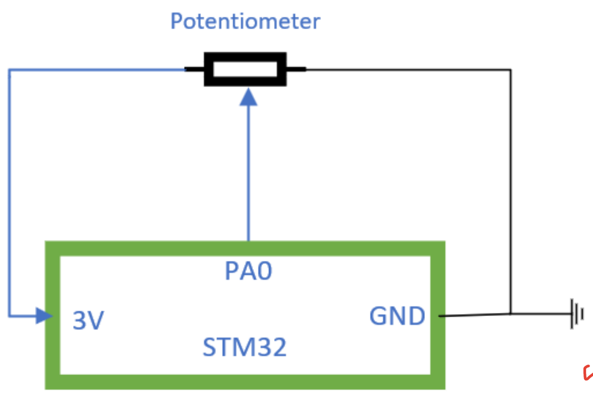
process to stop by occupying the processor.

- Delays force the processor to stop for the set amount without doing anything in the background which is inefficient use of time and the processor. Some important function may need to be executed or performed whilst in the delay, but the processor will have to wait for the set amount before doing the action. Delays are easier to implement but are not as robust.
- Interrupts use pre-scalars which are a factor of the processor clock speed, thus making them more accurate. They are hardware counters that may be fired whenever a certain action (a flag) takes place, making the program more robust since no processing power needs to be allocated to continue checking for a delay. Once the flag has been raised and an interrupt is called, the processor will do the interrupt routine and go back to where it left of.

2/3

4. Sample Rate

4.1. **From Laboratory 3: Task 1.3.** Draw the circuit diagram of the connected potentiometer



3/3

4.2. **From Laboratory 3: Task 2.1.** Comment on the chosen sample time for this system. Recommend a sample time should this be part of an embedded system

A sampling frequency of 40 Hz was used (pre-scaler value of 19) and it was successful in registering all levels of the counter, however it did have some limitation at the higher end of the rate of change of dial angle.

Chosen sample time needs to be high enough to distinguish between each individual level, it this case there are 256 levels, however, it also needs to be as low as possible to achieve that in order to not waste processing power. In case of the potentiometer, the dial could only be turned by about 280 degrees which is a very limited range. It approximates to about 1 degrees of change per level changed. This is a very small range of resolution and a high sampling rate would be required to reduce errors of skipping over quantisation levels. Also, as the rotational velocity of the dial increases, more sampling rate would be required in order not to skip a level. Choosing appropriate sampling rate will depend entirely on application, available error margin and processing involved.

2/3

2

generally good thoughts. how fast do you think the values might change as a max rate? what is perceivable by the user?