Lab 6: 7-Segment Displays

Introduction

The objective of this lab is to investigate how we can drive a 7-segment display using the ATmega328P/ATmega328PB microcontroller. While we've already looked at exporting the data from the microcontroller using protocols such as UART, a lot of embedded systems require the ability to display information locally, for instance using LEDs, 7-segment displays, LCD panels, and so on.

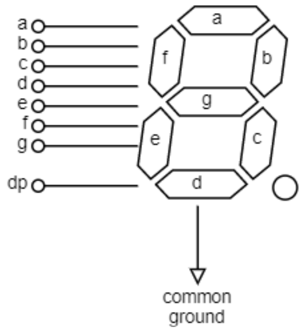
Read the entire lab manual before you start. This lab should take you approximately **5 hours**. There is a pre-lab, four compulsory parts and an optional activity. Before starting the lab, pull the lab repository from GitHub as it contains the Proteus model(s). Remember to commit and push your saved work to lab repository on GitHub after completing each task. **The final answers should be written in this document, with supporting calculations in your logbook.**

Pre-Lab: Basics of a 7-Segment Display

At this point, you should be familiar with the ATmega328P simulated on Proteus and the ATmega328P/ATmega328PB on the Xplained Mini. Before the lab, we will refresh our understanding of external input/output (I/O), and experiment with a single 7-segment display to understand how it works.

1) Open the Lab6_PI_Proteus.pdsprj Proteus project.
2) Note the pins of the 7-segment display that are connected to the ATmega328P (also note that the DP is not connected).

You may consider the common-cathode 7-segment display used here as 8 independent LEDs forming each segment and decimal point. It is wired as follows:



Common-cathode means that the anode of each segment is used to light the LED(s) in it by applying a positive voltage (i.e. logic 1). These anodes are coming out as signals "a" to "g" and "dp". Then, the cathodes of each segment are tied together and made available as a single pin that needs to be connected to the ground of the circuit (i.e. the cathodes are "common" to all the segments).

Q P.1: Given the above component, and assuming "dp" is the MSB and "a" is the LSB, complete the table to indicate what segments should light for each digit:

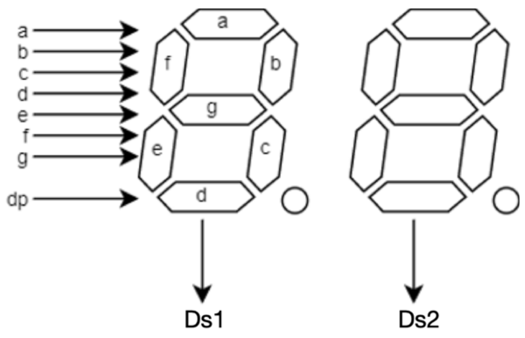
Digit	*Segments (byte / hex)*	*Digit*	*Segments (byte / hex)*
0	*0b00111111 / 0x3F*	*5*	0b01101101 / 0x6D
1	0b00000110 / 0x06	*6*	0b01111101 / 0x7D
2	0b01011011 / 0x5B	*7*	0b00000111 / 0x07
3	0b01001111 / 0x4F	*8*	0b01111111 / 0x7F
4	0b01100110 / 0x66	*9*	0b01101111 / 0x6F

3) Observe that the component in Proteus has two 7-segment digits. Each has its own common cathode made available as an output. The common cathodes are controlled using I/O pins to turn-on or turn-off an entire digit. Usually, a buffer (e.g. BJT) is used to control the common cathode if the display require significant current to drive the LEDs. Since we are using a very low power display, it is ignored here. Setting the common cathode of a digit to 0V (i.e. logic 0) will turn-on that digit. Setting the common cathode of a digit to logic 1 will turn-off that digit. This allow us to *multiplex* the displays saving the number of I/O pins needed to drive a multi-digit display. We'll talk more about this later.
4) Create a new GCC C Executable Project in Atmel Studio, selecting ATmega328P as the device family.
5) Use the appropriate DDRn registers to set up PORTB, PORTC, and PORTD.
   a. Set the I/O pin connected to the push button as an input.
   b. The I/O pins connected to the common cathodes of the two 7-segment digits (Ds1 and Ds2) and the anodes of the LED(s) in each segment should be set to outputs.
   c. Now, set Ds1 = 1, and Ds2 = 0 to configure the circuit to use only the second digit, Ds2. Thus, it will be as if only the second digit is enabled.
6) Using <util/delay.h> and _delay_ms(time), write code to achieve the following:
   a. A counter starts from 0 and counts up at a rate of 1 count per second (approximately) up to 9.
   b. Once the counter reaches 9, it resets to 0 and starts counting up again.
   c. Every time the counter value changes, the I/O pins driving the anodes of the LED(s) in each segment (i.e., signals "a" to "g" and "dp") are updated to reflect the value. To do this consider storing the values you listed as your answer to QP.1, in an array. The first element of the array can represent the bit pattern to output to segments in order to display '0'. The last element of the array can represent the bit pattern to output to segments in order to display '9'. This allows you to easily determine the bit pattern to output to each segment of the digit to display a number between 0-9 by reading the corresponding element of the array.
   d. If the push button is pressed, the counter should be reset to 0, and start counting up again (Hint: You may need to loop a short 100ms delay in order to sample the push button faster than the 7 segment updates).
   e. This behaviour should repeat.
7) Build, link, and run the program in Proteus. Visually inspect that the push button resets the counter, the second digit (Ds2) of the display is counting at a rate close to 1s, and that all 10 digits (from 0 to 9) display correctly.

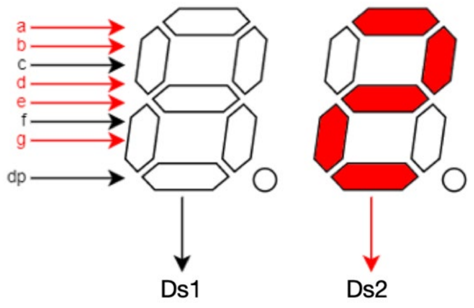
Part 1: Multiplexing LEDs

It is impractical to have one output pin driving the anodes of the LED(s) in each segment in large display. Imagine how many wires you'd need for a high definition

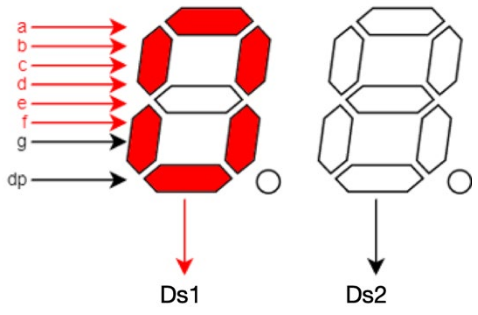
OLED TV! Instead, designers must find ways to control large numbers of LEDs using a small number of pins. There are two popular mechanisms for achieving this. To drive the display used in this project we will use both.
In Part 1 of this lab we will examine the first technique: *multiplexing*. Here, LEDs are arranged such that they share inputs, and rather than all having the same output, they have *banks* of outputs. Let's examine this using the two-digit display in Proteus as an example:



As in the circuit diagram above (as well as Proteus), the incoming *a, b, c, d, e, f, g,* and *dp* signals from the I/O pins of the microcontroller are wired to anodes of corresponding segments in both digits Ds1 and Ds2. For example, signal line *a* generated by an I/O pin is connected to the anode of segment 'a' in digit 1 (Ds1) as well as the anode of segment 'a' in digit 2 (Ds2). Now, if we input the signals *a, b, c, d, e, f, g,* and *dp* to display number "2" (i.e. 0b01011011), and made the common cathode of Ds2 to logic 0 (0V) to enable that digit, then the number "2" will be displayed in on Ds2 as shown:



Ds1 was set to logic 1 (5V) to disable the first digit while we were displaying number "2" on Ds2. Likewise, we could make the common cathode of Ds1 to logic 0 (0V) while the common cathode of Ds2 is set to logic 1 (5V) to only enable Ds1 and display a completely different number, for example "0" as shown:



We could then alternate between Ds1 and Ds2, displaying two different numbers in each 7-segment digit. If we alternate between the two digits fast enough, it will appear as both digits are turned-on at the same time. This is the principle behind multiplexing a display. The rate at which you alternate between digits is usually referred to as the refresh rate (given in Hz), and should be typically above 30Hz. Let's try this out now.

The nice thing about LEDs is that they can turn on and off quite rapidly – faster than human eyes can make out! So as explained earlier, if we flash our two displays rapidly enough, each with their own digit, it will appear as if both digits are on simultaneously - even though they are not! This is known as *persistence of vision*. As this needs to be done quickly and consistently we should use a *timer interrupt*.

Q 1.1: Modify your program from the pre-lab to count from 0 to 99, incrementing it by 1 every 1s using _delay_ms(time). Once the counter reaches 99, it should reset to 0 and start counting up again. Store the counter value in a global variable. Instead of outputting the counter value to the display in the main loop, now use a timer interrupt that triggers every 10ms (e.g. using Timer0 – you may wish to reuse your code from Lab 5 Part 3) to display the counter value on the two digit 7-segment display. When the interrupt occurs, perform the following steps:

1. *Choose which digit to update using a flag*
    a. *If you last displayed digit 1, you will display digit 2*
    b. *If you last displayed digit 2, you will display digit 1*
2. *From the array that stores the bit pattern to output to the segments, determine the segments to be turned-on for the chosen digit from the global counter variable (you will need to make this array a global variable)*
3. *Disable both digits*
4. *Set the I/O pins connected to the segments to display your chosen digit*
5. *Enable your chosen digit and update a flag indicating next digit to update*

You may wish to place the global array that stores the bit pattern to output to segments and the global counter variable in a separate display.c file rather than your main.c. If so, it's best to add a function to manipulate or increment the counter variable rather than accessing the global variable from main.c. Compile and test your program using Lab6_PI_Proteus.pdsprj Proteus project. You should see the counter value displayed on both digits successfully before moving on (i.e. the display counting from 0 to 99 every 1s).

Q 1.2: In the pseudo-code steps above, why do you think that we output the segments before we enable the display (i.e. why is Step 4 before Step 5)?

Preventing ghosting where the taint pattern trom the previous lit digit momentarily appears on the new digit.

Q 1.3: In the pseudo-code steps above, why do you think that we alternate the digit we update every 10ms?

To achive a high retresh rate that leverges the phenomenon of persistence of version, creating a stable and flicker-tree image on the display.

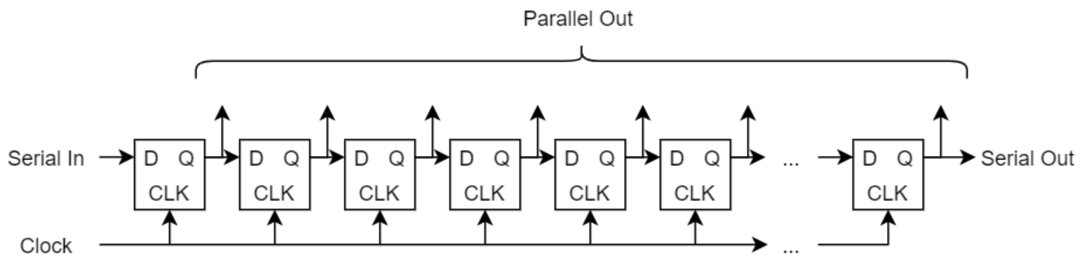
Part 2: Shift Registers

While multiplexing is a powerful technique for cutting down on I/O pins needed to drive a display, it often "isn't enough". Consider the 2 digit circuit from Part 1 – to control two 7-segment digits and the decimal points, we still need 10 I/O.

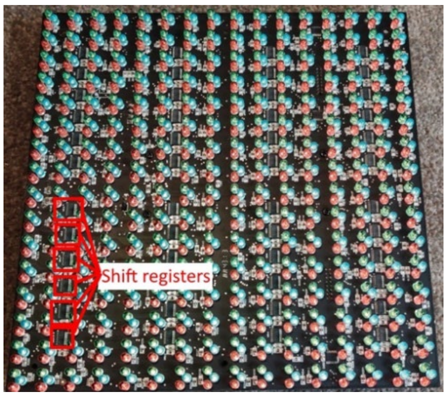
Q 2.1: If only using multiplexing, how many I/O pins are needed to drive a display that has N number of 7-segment digits where each digit also has a decimal point?

8 + N pins

Shift registers are a popular component used to resolve this issue. As a refresher, shift registers are a type of data storage device that convert *serial input* into *parallel output*. Internally, here's how a generic shift register is wired:

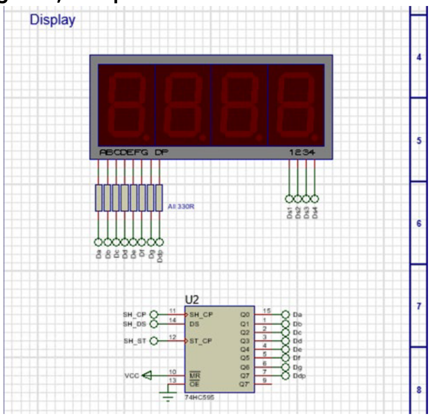


Here, each of the components in this diagram is a D type flip flop. On a rising clock edge, they will capture the data at D, and then emit that at Q. As such, after the first clock tick (i.e. rising edge), the first flip flop will be emitting the first bit of data from "Serial In". You then change the "Serial In" to the next bit of data, then pulse the clock (i.e. toggle clock signal from low to high and back to low) once more. Now, the first flip flop is outputting the second bit of data, and the second flip flop is outputting the first. On and on this goes until you completely fill your shift register, and your data stream is now present across the parallel output of the shift register. Refer to your COMPSYS 201 notes to read more about shift registers. It's hard to overstate how powerful this is as a technique – many commercial systems use simple shift registers such as this in their displays, for instance:

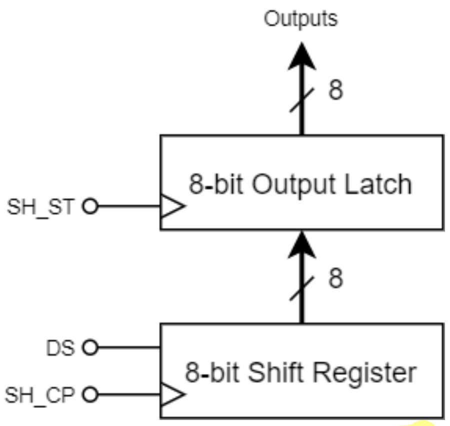


Thanks to the magic of shift registers, this particular panel needs just 8 I/O pins to control 768 LEDs! You can see details of this project here. In your project we will be utilising a similar technique, involving *both* shift registers *and* multiplexing. This also isn't uncommon – using the example of the commercial panel above, if displays of multiple panels are required, they may be combined together using multiplexing! Let's first consider just the shift register component.

Now open the Proteus Project Lab6_PII_Proteus.pdsprj. You will notice the *Display* section of this Proteus project, depicted here:



This is now made up of two components – firstly, the 74HC595 shift register U2, and also a common cathode four-digit 7-segment display (similar to LTC-5723HR). In total there are 32 LEDs to drive (4 sets of 7 segments, and 4 decimal points). However, between multiplexing the outputs and using a shift register for the inputs we need just 7 I/O pins to drive/operate this whole display! So how do we control the 74HC595 shift register? Let's take a closer look at a simplified version of the internal block diagram of this device:



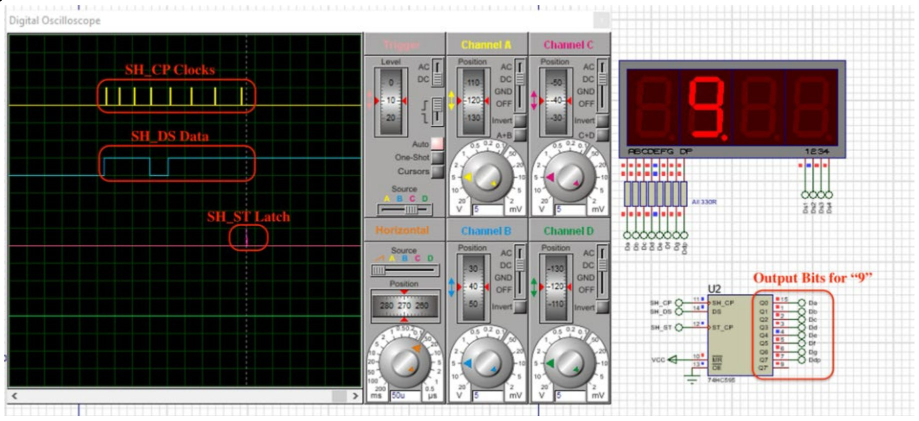
Here, data (i.e. "Serial In") is presented on the DS pin and clocked in using the SH_CP pin (i.e. toggling SH_CP). Once all 8 bits of data is loaded, we latch it to the outputs by toggling the SH_ST pin. Note that in this context toggling means the signal is changed from low to high and back to low. Let's try this now.

Q 2.2: Create a new Atmel Studio project for the ATmega328P microcontroller. Now, open Lab6_PII_Proteus.pdsprj in Proteus, and find the I/O pins connected to SH_ST, DS (which is labelled as SH_DS), and SH_CP. Configure these I/O pins as outputs in your code (ideally in a function called init_display()). You will also need to find the I/O pins used to enable/disable each digit, Ds1, Ds2, Ds3, and Ds4. Configure these also as outputs in your init_display() function. To begin with, develop code to display number "7" on Ds4. To do this, you have to set Ds4 to be logic 0 (0V), and set Ds1, Ds2, and Ds3 to all be logic 1 (5V). Now, to turn on the correct segments through the shift-register, set up code to perform the following actions in a function called send_next_character_to_display() (reuse code from the first part of this lab where you can):

1. *Ensure SH_CP and SH_ST are both set to logic 0*

2. *Create an 8-bit variable that holds the bit pattern to turn on the segments needed to display number "7" (you can read this bit pattern from the array you created as part of the pre-lab)*
3. *For each bit in this variable, starting with the MSB: (Hint: remember to include the decimal point!)*
   a. *Set the SH_DS pin to be either "0" or "1" as per the bit being transferred to the shift register*
   b. *Toggle the SH_CP pin on (i.e., "1") and off (i.e., "0") to "shift in" this bit to the shift register*
   c. *Proceed to the next bit in the variable and repeat the above process*
4. *Once all 8-bits are "clocked out" (or loaded),*
   a. *Toggle the SH_ST pin on (i.e., "1") and off (i.e., "0") to latch the output – this will set the 8-bit pattern from the shift register to drive the segments*

Once you've completed this, simulate your system using the Proteus Project Lab6_PII_Proteus.pdsprj. You should see number 7 appear on the display. For troubleshooting and debugging purposes, you'll see that an integrated oscilloscope is currently set to read the pins going into the shift register. You may use this to verify the correctness of your code. An example is shown here where we are displaying "9." on Ds2:



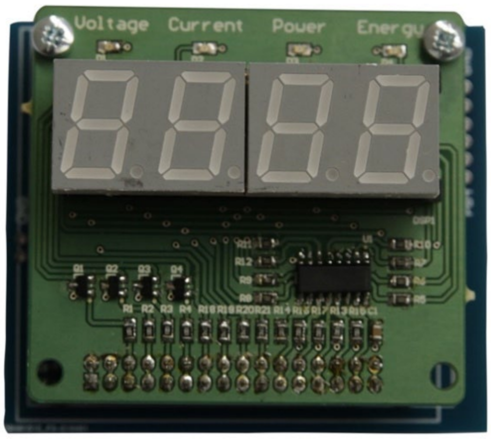
Well done! It's now time to refactor your code to feature everything so far and display a number using all 4 digits.

Q 2.3: Modify your program to now feature a counter that counts from 0 to 9999. The counter is incremented every 400ms using _delay_ms(time). To achieve this, create a new file display.c, using the skeleton code provided below. Here, every time the counter is updated, the current counter value is passed to display.c from main.c using seperate_and_load_characters(). Using previous work, configure Timer0 to have an ISR which triggers every 10ms. In the ISR, call the send_next_character_to_display() function (as indicated in the skeleton code the function you wrote in Q2.2 needs to be modified to scan through the 4 digits). The init_display() function can be reused. Make sure to create a display.h which contains the prototypes for functions in display.c. Build and simulate your project using Proteus – you should have a 4-digit counter! Note that in slower computers the display animation may not work properly (rare but possible). Check the *oscilloscopes to make sure the signals, SH_CP, SH_DS, SH_ST, Ds1, Ds2, Ds3 and Ds4 are correct.*

display.c

```
#include "display.h"
#include <avr/io.h>

//Array containing which segments to turn on to display a number between 0 to 9
//As an example seg_pattern[0] is populated with pattern to display number ‘0’
//TODO: Populate this array using your answer to QP.1
const uint8_t seg_pattern[10]={0x3F, 0, 0, 0, 0, 0, 0, 0, 0, 0};

//4 characters to be displayed on Ds1 to Ds 4
static volatile uint8_t disp_characters[4]={0,0,0,0};

//The current digit (e.g. the 1's, the 10's) of the 4-digit number we're displaying
static volatile uint8_t disp_position=0;

void init_display(void){
	//TODO: Finish this function
	//Configure DDR bits of the I/O pins connected to the display
}

//Populate the array ‘disp_characters[]’ by separating the four digits of ‘number’
//and then looking up the segment pattern from ‘seg_pattern[]’
void seperate_and_load_characters(uint16_t number, uint8_t decimal_pos){
	//TODO: finish this function
	//1. Separate each digit from ‘number’
	// e.g. if value to display is 1230 the separated digits will be
	//      ‘1’, ‘2’, ‘3’ and ‘0’
	//2. Lookup pattern required to display each digit from ‘seg_pattern[]’
	//   and store this pattern in appropriate position of ‘disp_characters[]’
	// e.g. For digit ‘0’ in example above disp_characters[0] = seg_pattern[0]
	//3. For the project you may modify this pattern to add decimal point at
	//   the position ‘decemal_pos’
}

//Render a single digit from ‘disp_characters[]’ on the display at ‘disp_position’
void send_next_character_to_display(void){
	//TODO: finish this function
	//1. Based on ‘disp_position’, load the digit to send to a local variable
	//2. Send this bit pattern to the shift-register as in Q2.2
	//3. Disable all digits
	//4. Latch the output by toggling SH_ST pin as in Q2.2
	//5. Now, depending on the value of pos, enable the correct digit
	//   (i.e. set Ds1, Ds2, Ds3 and Ds4 appropriately)
	//6. Increment ‘disp_position’ so the next of the 4 digits will be displayed
	//   when function is called again from ISR (reset ‘disp_position’ after 3)
}
```

Together with the emulator board you are also provided a display shield that can be attached on top of the Xplained Mini 328P/328PB board to test and validate your program on real hardware:



Q 2.4: Load your program on to the Xplained Mini board. Test using the emulator board and the display shield to make sure the 4-digit counter works on real hardware. If you have a Xplained Mini 328PB, remember to change the device from an ATmega328P to an ATmega328PB in your Atmel Studio project (found under Project > "Your Project Name" Properties > Device). Then compile and load the program.

Optional Task: Display an ADC voltage

So far we have configured our display so that it can write four digits from a variable to the display. It's now up to you to extend this further for your project. You'll need to decide on a method to display voltage, current, and power values. This might mean you need to extend your segment logic to support displaying units (e.g. P for power). You will likely also need to be able to use the decimal point.
A proteus model named Overall_Project_Proteus.pdsprj is provided to you to test and debug the code you will develop to implement the energy monitor. You may find it easier to debug the code on Proteus as it provides a powerful debugging interface. The 7-segment display animation may not function properly, but the debugging features should be very useful to find and correct issues in your code. We recommend you to use this Proteus model together with the emulator board to develop your code until your energy monitor hardware is ready. Note that, although a simulator captures most of the real life behaviour of the physical MCU, it may not be 100% accurate. Therefore, you may have to do minor modifications to the code, in order to achieve the desired outcomes in the real world.
Take this opportunity to modify your program from Part 2 to read the voltage applied to ADC2 and display the raw ADC value (between 0-1023) on the display. You can use the emulator board to test your code by changing the variable resistor on it. Now, process the ADC value to convert it to a voltage. How will you decide where to put the decimal point? The easiest is to pre-determine how many decimals points you will include when displaying voltage. Modify the program from

above to display the actual voltage (0V to 5V) with a precision of 2 decimal points on the display. Use the right most digit to display character “U” for voltage.
It is also worth thinking about steps you will follow to complete the project. Getting both the 7-segment display working and the calculations done require good mastery of microcontrollers. This is because they both require precise timing. Therefore, it may be a good option to first complete your voltage, current, power and energy calculation algorithms using terminal as a display. Ones you are confident this part works, then you can try to implement the 7-segment display. However, before you start coding, it is important to think about the overall organisation of your program. This could be done using a flowchart or an UML diagram. Otherwise, when you come to implementing, for example the 7-segment display, you may realise that the resources needed were allocated for a different functionality. Or simply the components of your software may not integrate well and effectively. It is also good to think about interrupt priority and which parts of your code need to be prioritised over others. Also remember that hardware modules work in parallel to the main program execution. You need to plan carefully to utilise the limited resources on the ATmega328P/PB in the best possible way.
These are the sorts of questions we will leave up to you. Remember that software you have developed as answers to the labs in this course forms about 80% of the program you will develop for your project. Well done for completing this lab series, and good luck for the rest of your project!

To get signed off for the lab:

- Record all the workings in hardcopy or in digital format so we can give you marks for workings if final answers are not correct
- Modify the simulation models provided as per the lab tasks and save
- Commit and push your saved work to the lab repository on GitHub after completing each task and make sure it is up to date
- Complete this document summarising your final answers, commit and push to lab repository on GitHub
- Update logbook indicating yours and teams progress, meeting notes, etc.
- Go to your assigned interview session to check-in with a TA who will check your solutions and ask a few questions
- Report on your weekly progress