Embedded Systems  
Semester 2 Learning Journal

Jazer Barclay

Student ID: 20837308

CI514 – Embedded Systems

9th May 2022

Table of Contents

[Week 1 – An Introduction to Assembly 1](#_Toc102941383)

[Week 2 – Address Modes and Chip Architecture 4](#_Toc102941384)

[Week 3 – Assembly Programs 8](#_Toc102941385)

[Week 4 – Arduino Ports 10](#_Toc102941386)

[Week 5 – Port Structure and Pin Assignment 14](#_Toc102941387)

[Week 6 – Event Recognition and Interrupts 16](#_Toc102941388)

[Week 7 – Interrupt Driven Systems 20](#_Toc102941389)

[Week 8 – Timer Interrupts 24](#_Toc102941390)

[Week 9 – Interrupts Continued and Serial Port 27](#_Toc102941391)

[Week 10 – Sorting Algorithm Preparation 30](#_Toc102941392)

[Week 11 – Sorting Algorithm 35](#_Toc102941393)

[Week 12 – Conclusion 38](#_Toc102941394)

[References 39](#_Toc102941395)

# Week 1 – An Introduction to Assembly

For our first week we were introduced to the assembly language. I was unable to attend this lecture due to work however I did my own research using the slides and reading up on documentation online via the Arduino website.

Graphical user interface, text, application

Description automatically generated

Arduino Mega Documentation with Datasheet

I found that in assembly we would have to design and build our code for the particular chip since pin layouts and certain memory locations existed on some but not others. The C compiler would manage this for us when programming in C however we would now be doing this by hand ourselves.

A close-up of a circuit board

Description automatically generated with medium confidence A close-up of a circuit board

Description automatically generated with medium confidence

ATmega2560 vs ATmega328p

Unlike C, we did not have functions and variables; instead, we will need to work with registers and use the chip instruction set to control the functions of the chip.

The instruction set consist of Opcodes and in many cases take in parameters called Operands. These are like dedicated functions that the chip can perform, and we can give them fixed parameters to move bits around in registers.

Table

Description automatically generated

Snippet from the ATmega2560 Documentation - Instruction Set Summary

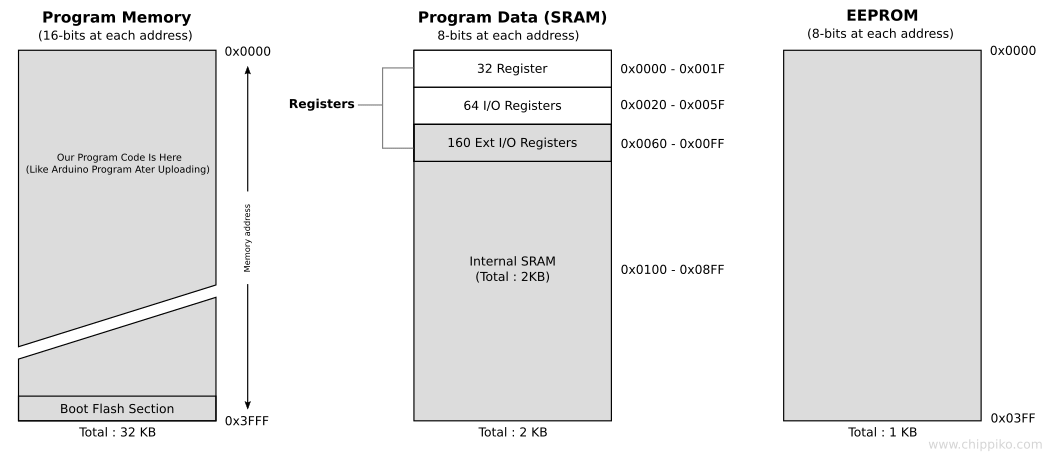
I investigated the datasheet for the Arduino Mega 2560 chip (ATmega 2560) and looked at some of the example code snippets. Each operation exists on its own line and certain Opcodes operate on different parts of memory.

Graphical user interface, table

Description automatically generated with medium confidence

Example Assembly from Documentation

This makes sense in that the program memory storage is 16 bits able to store the opcode and operand for each memory location address.



Memory configuration for ATmega chips from documentation

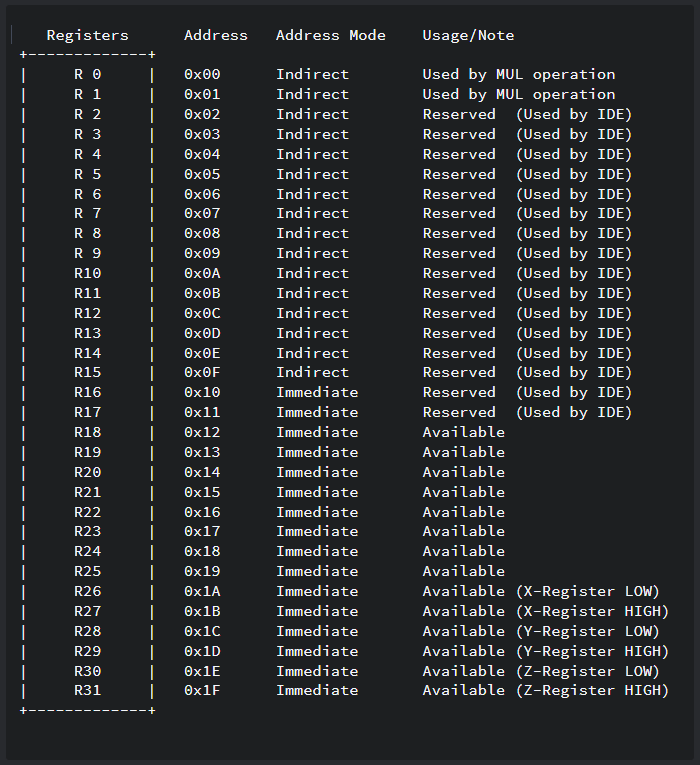
This map also gives a good insight as how we can control the outputs and read inputs from the program registers. Having worked with the PIC16F84A chip before, I can assume we have 32 general working registers which we can directly work with, I/O registers for interacting with the data pins and internal SRAM for storage which can be accessed via specific Opcodes.

This was a lot of research for my introductory week of assembly, and I look forward to looking at more chip architecture next week!

# Week 2 – Address Modes and Chip Architecture

This week’s lecture covered the prior week’s slides and went into more detail on address modes. This included immediate, direct, indirect, indexed, relative and inherent address modes.

We also covered the program data (SRAM) memory space in more detail looking specifically at the 32 general working registers R0 to R31.



Arduino SRAM First 32 Registers - Working Registers

I took this further by looking at how the Arduino’s AVR Address Modes correlate to some example instructions from the documentation. Opcodes such as load data immediate (LDI) which only allows setting of data in immediate working registers (registers 16 to 31) and OUT which writes from working registers (r0 - r31) to input/output registers (0x20 – 0x5F).

Looking at last week’s diagram of the full memory structure, the program memory is 16 bits long compared to the 8 bits in SRAM. I found that this is to store the opcode and operand as binary which takes up 16 bits per line. This matches the code we write in assembly where each line is made up of these base components.

Lab time covered how to interact with the Arduino using inline assembly where we can write small sections of code using assembly and the rest in C. This makes it easier to modify more and more of the code with the assembly equivalent where the goal will be to have only assembly code at the end.

Graphical user interface, text, application

Description automatically generated

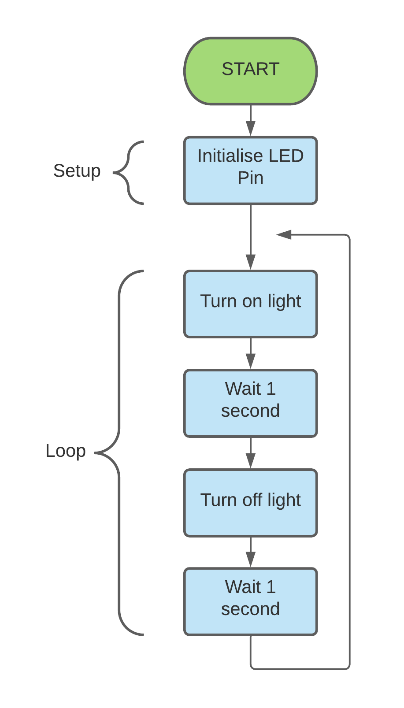
Regular C Function

Inline Assembly

Inline assembly vs regular C functions

I learned that the Arduino supports an ‘.S’ file which contains only assembly and doesn’t require clobbering registers. It also supports using names rather than address locations which will make readability far easier. I intend to move towards putting all my assembly in an ‘.S’ file in the future.

In my own time I recreated the blink sketch using inline assembly which replaced the “pinMode()” function and the “digitalWrite()” function with their assembly equivalent.



Flow chart designed for the blink example sketch

Following my existing flow chart for the blink sketch, I set out to recreate it using inline assembly for as many of the steps as possible. As a result, I was able to replace the led control blocks with the assembly equivalent and left the delay as a regular C function. This sketch can be found in the “week2” folder under the title “asm-inline-blink” sketch.

I then recreated this flow chart using more detail matching what I had learned from this practice. This includes all ASM functions shown as their own smaller flow charts besides the main more general chart shown before.

Diagram

Description automatically generated

New flowchart showing a breakdown of each section after learning how to use LDI and OUT opcodes and their required operands

I uploaded this sketch using the same method as last semester through compiling the code using the compile button and uploading via the upload button built into the IDE. This can be seen in the following picture.

Graphical user interface, application

Description automatically generated

Compile and upload buttons used during development and testing

I intend to replace the delay in the future with the assembly equivalent however so far, I do not have the knowledge to do this and I’ll save this for next week’s lab.

# Week 3 – Assembly Programs

This week’s lecture covered the ASM inline assembly function in a bit more detail and how we can write our opcodes, operands, and comments within. We also looked at some example code in the slides showing how the theory of implementing a delay function in assembly. This would be a great goal to implement during the lab!

For the lab I did more research on the inline assembly ‘ASM’ function and learned that it takes multiple output and input parameters between the colons. At the end the clobbered registers are a method of the Arduino compiler where used registers can be cleaned up preventing weird results due to a register not holding its prior contents since we changed them. This is how the structure looks syntactically:

Text

Description automatically generated

ASM function syntax in Arduino C

I further improved my blink sketch by using the SBI and CBI opcodes. These set a single or clear a single bit in a register. The general flow chart remains the same however the method used to achieve the result now only requires the setting and clearing of single bits in I/O registers. This sketch can be found in the “week3” folder titled “asm-inline-blink-improved”.

I wanted to complete the implementation of the delay using only assembly from the lab material. This was the only part of my blink sketch that still used the C function “delay()”. I looked at the provided example code from the lab and used the flow chart within the lecture slides to understand how this works.

Each operation in assembly takes a certain number of cycles to perform and the processor can perform up to the clock frequency number of operations per second. So, with the clock being 16MHz, we would need to take up 16 million operations to delay for 1 second or 16,000 for 1 millisecond.

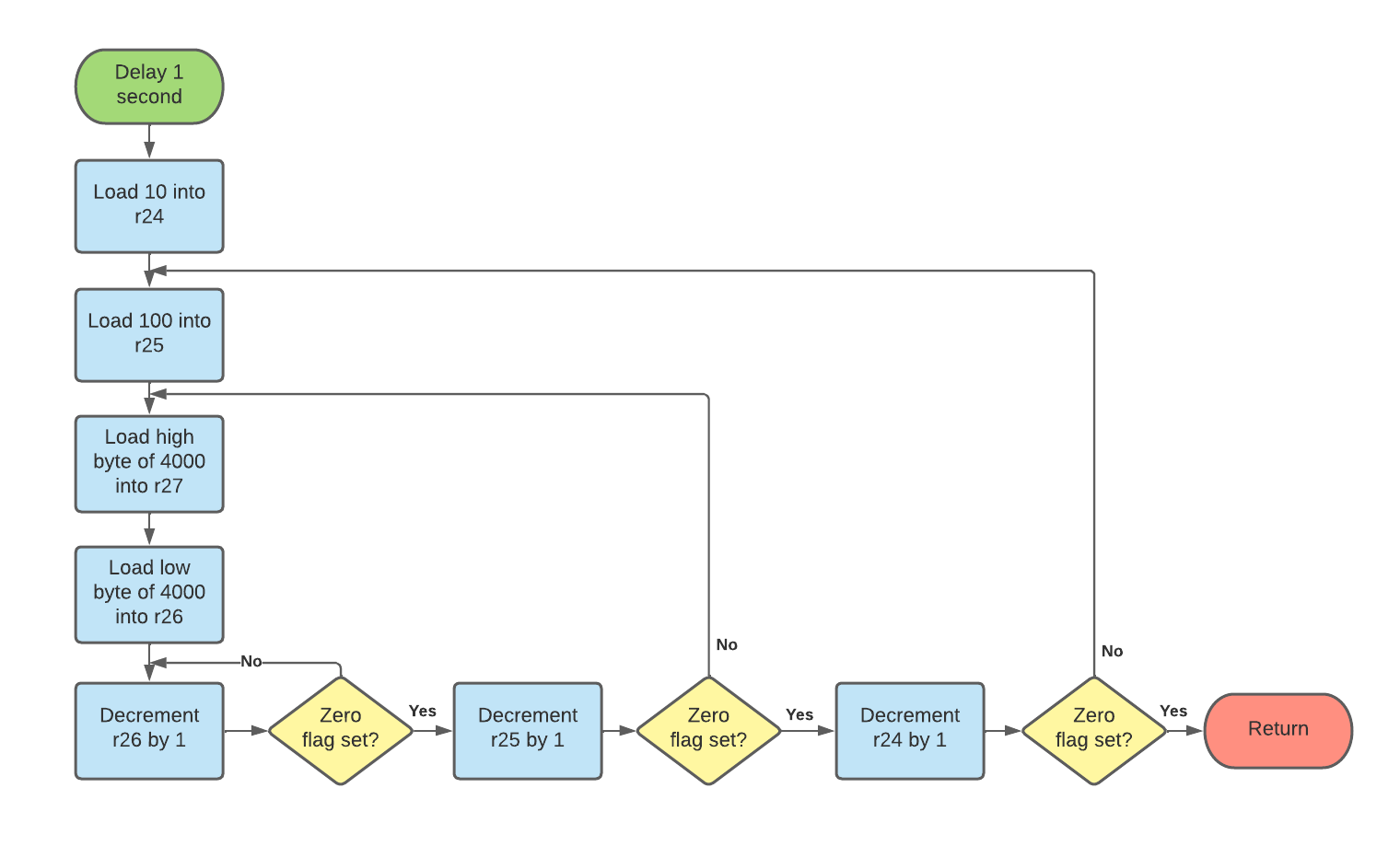
I calculated that a decrement of a register’s value, checking if the zero flag is set and branching would take a total of 4 cycles to perform. We can take 16,000 and divide it by 4 to calculate the total number of cycles we would need to delay for 1 millisecond.

If we divide 16000 by 4, we get 4000. This is too large to fit in a single register which can only store up to 2^8 or 256 so we can make use of the special working registers that work together to store a word which are 16 bits in length which can store 2^16 or 65536.

We can use bit shifting and masking to get the low and high bytes for 4000 and store them in the low and high bytes that make up the word register.

Now that we can delay for 1 millisecond, we can loop over it 100 times to get 100 milliseconds and outer loop that once more to get a full 1 second delay. This can be done with regular registers since these values are smaller than 256.

With this knowledge, I created a flowchart to use as a schematic for creating the delay in assembly. This just shows how to implement the delay without the full blink sketch to save on repetition.



1 second delay function using assembly

Next week I plan on implementing this into a sketch consisting of only assembly code and hopefully using the “.S” file to save on clobbering registers and reinforcing my knowledge of labels and functions.

# Week 4 – Arduino Ports

This week’s lecture went into details about the board pins and how they’re associated with the I/O registers. Each pin is grouped into a port with other pins. Since each register holds 8 bits, pins are grouped by letter into one of these ‘ports’

For instance, on the MEGA we have PORTB which is a collection of digital pins 13, 12, 11, 10, 50, 51, 52 and 53. These go from most significant to least significant bits.

A picture containing text

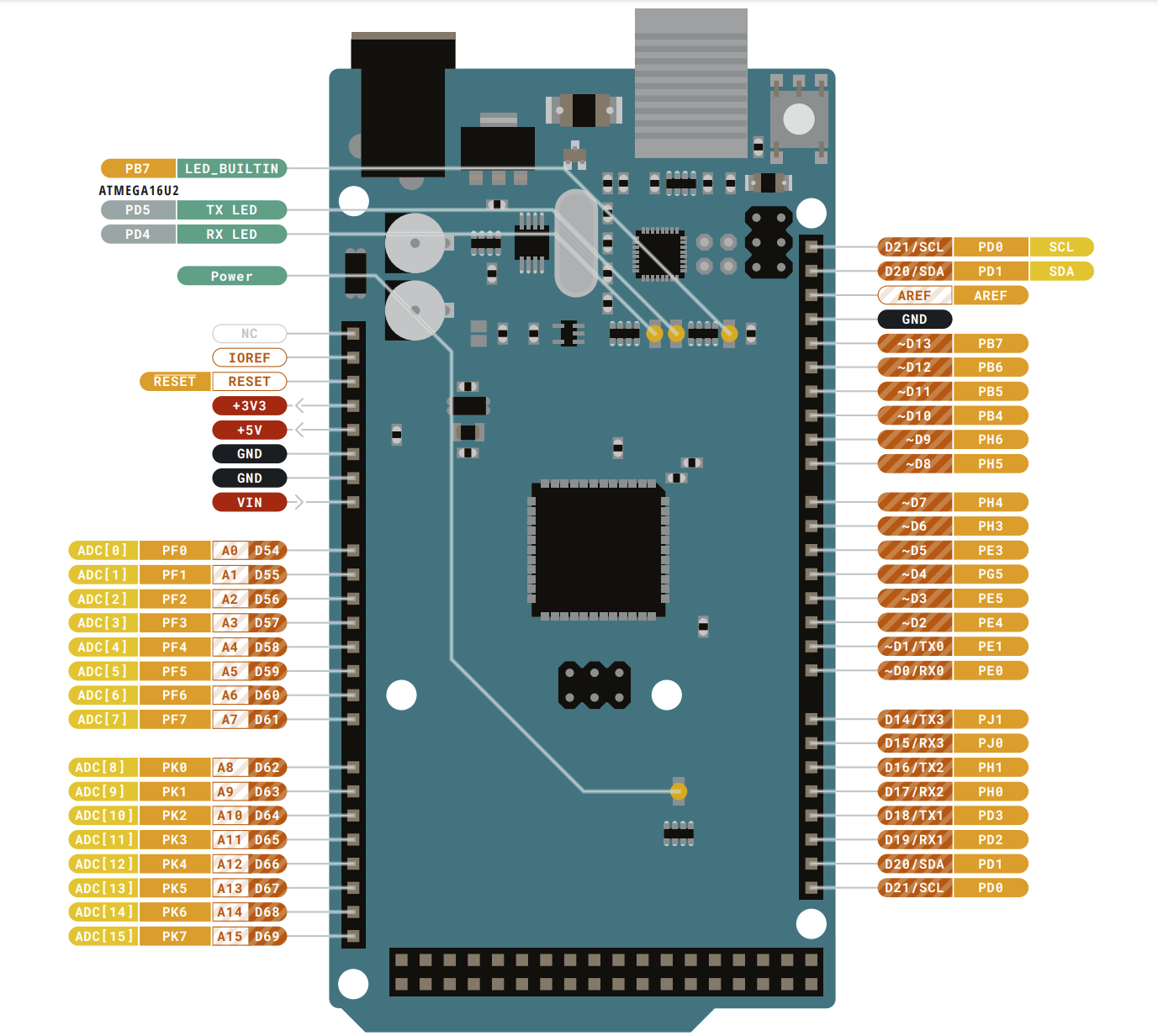
Description automatically generated

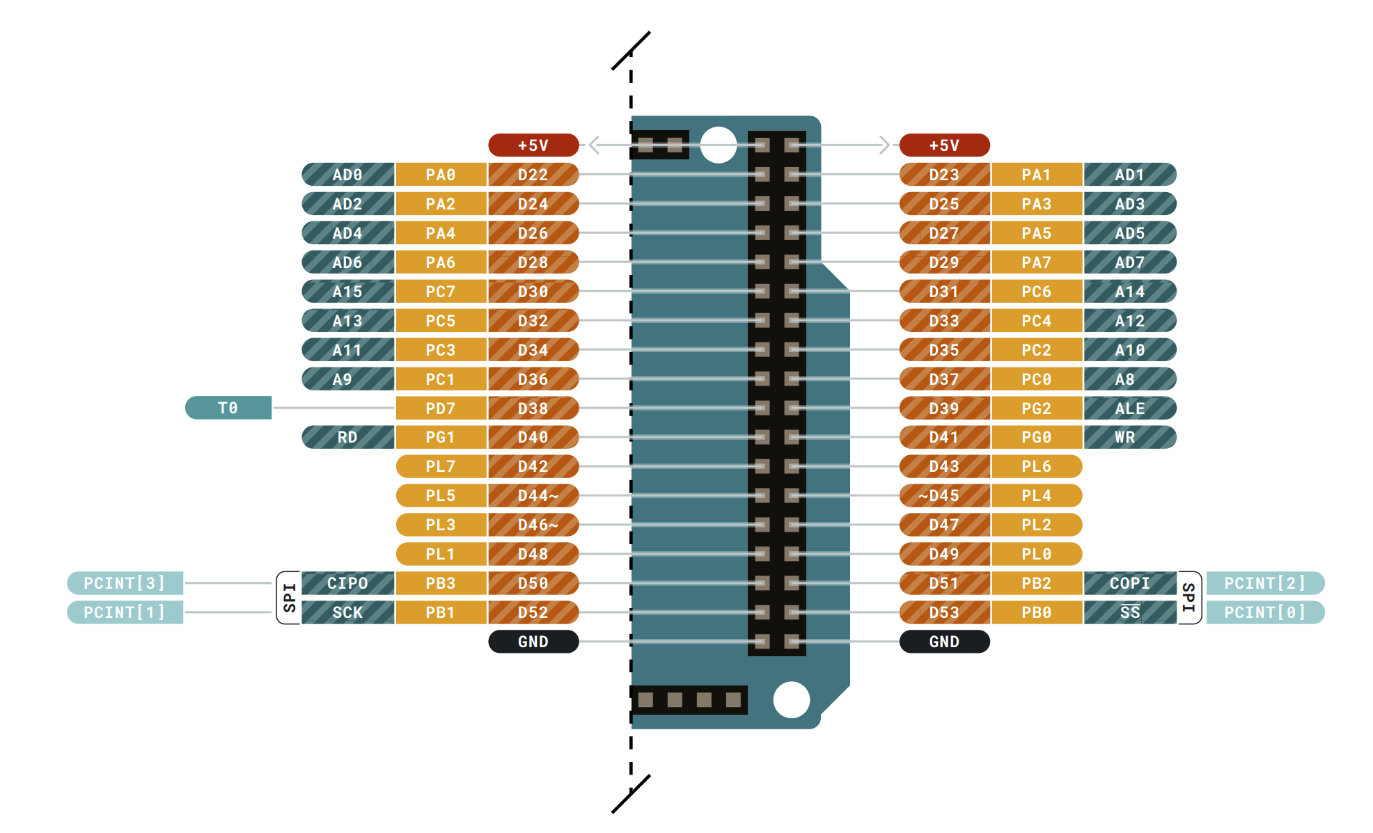
PORTB Pin mapping on the Arduino Mega 2560

There are 3 registers associated with each port which all serve a specific purpose: The Data Direction Register (DDR) which denotes if pins are inputs (0) or outputs (1), the Output Register (PORT) which can be set to low (0) or high (1) to control the pin and finally the Input Register (PIN) which can be read in as low (0) or high (1).

So far, I have been using the DDRB address for setting the pins as outputs and writing values to the PORTB address for setting pins either as high or low.

After looking at the documentation and some online research, I found a great infographic from the Arduino website showing how each of the pins are associated with their ports. This better shows how the ports are spread across the board with port ‘A’ looking the most grouped in the bottom right of the board.





Arduino Mega Pinout Reference

For the lab I took to creating my delay routine from the week prior. This was an easier task being able to work directly from the flowchart. I created the sketch originally using inline ASM and this version can be found in the “week4” folder called “asm-inline-delay”. To challenge myself, I moved to using a separate “.S” file and found a good starting point for the general structure. The last version of this sketch can be found in the “week4” folder titled “asm-blink”.

To run this, first open the “.ino” Arduino sketch file using the Arduino IDE.

Graphical user interface, application, Word

Description automatically generated

Highlighted asm-blink sketch opened in Arduino IDE on the right

Afterwards, the code can be found in the second tab that has the same name as the sketch and ends with ”.S”.

Graphical user interface, text

Description automatically generated

asm-blink assembly code found in second tab

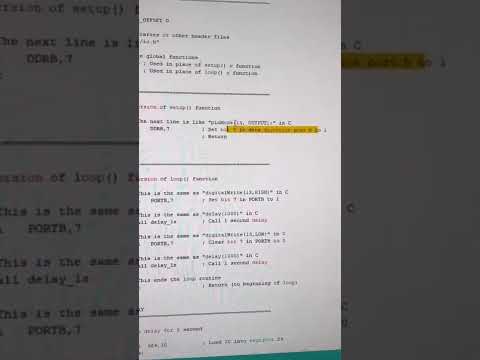
To compile and run the code, the process still is the same. Simply use the two available buttons at the top of the IDE to compile and run, respectively.

Graphical user interface, text

Description automatically generated

Compile and upload buttons stay in the IDE for the assembly “.S” file

This blink sketch can be seen running in the following video hosted on YouTube:

[](https://www.youtube.com/embed/VpJQn13_oso?feature=oembed)

Blink sketch written using assembly on the Arduino Mega

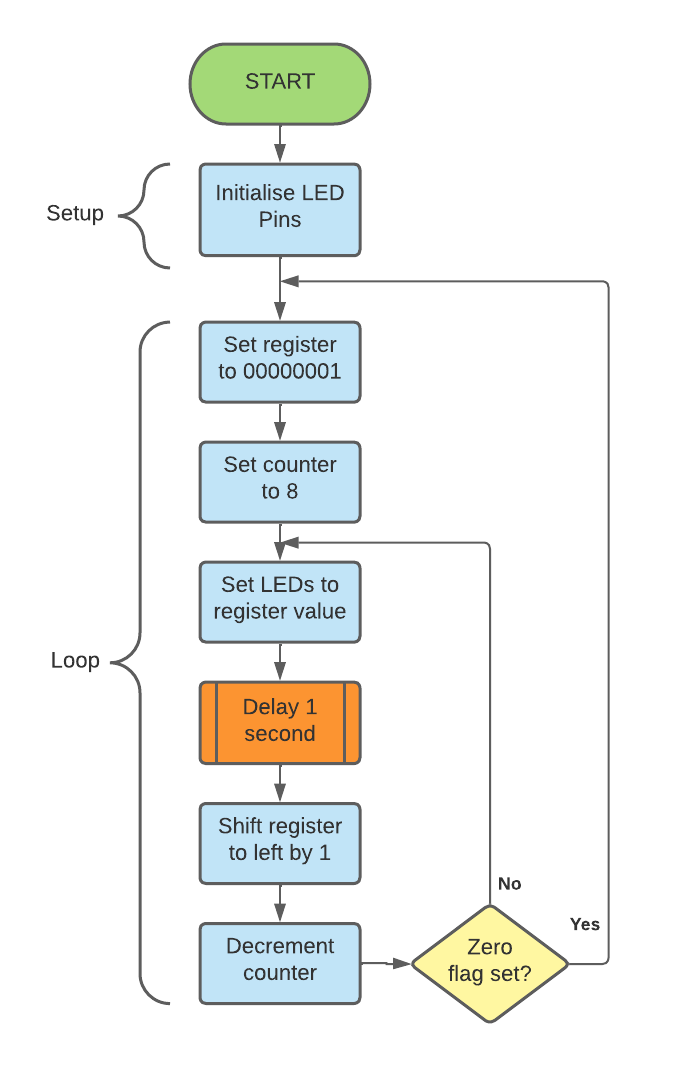
# Week 5 – Port Structure and Pin Assignment

This week’s lecture covered more details on port structure and pin assignment. We also covered the delay function in more detail going over how the BRNE opcode functions as well as why we have two different decrement functions.

To prepare myself for the first main task of this semester, the traffic lights, I worked on understanding how to control multiple LEDs using the SBI, CBI, LDI and OUT opcodes and picked up how to use LSL opcode which shifts a byte left by 1.

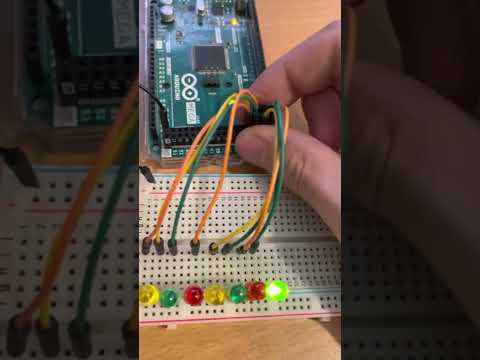
Using all of this, I had created the “asm-inline-strobe” and “asm-strobe” sketches which display a single bit which is shifted from right to left. At each shift, the value of the register is written to port A.

Since I had not intended to create this but rather through trial and error, I learned and understood how to do this without the use of a flowchart. To show my understanding, I created an in-depth flowchart of its operation so looking back I will know the process taken.



Strobe sketch flowchart

A video of this sketch running can be found below:

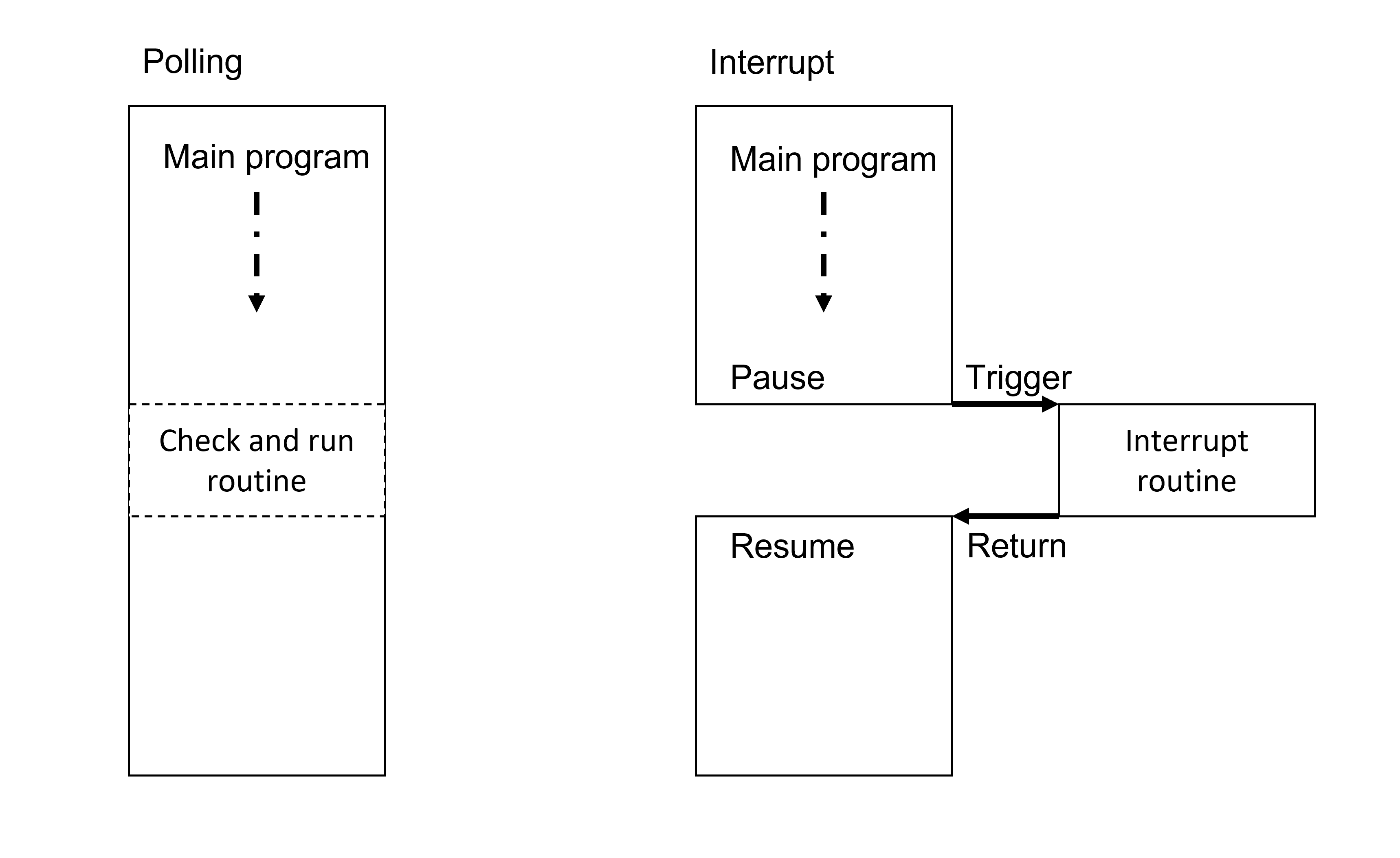
[](https://www.youtube.com/embed/NBCrtij3V6c?feature=oembed)

Strobe sketch running on the Arduino Mega on Port A

Learning this will give me a good grounding to get started with the traffic light task now able to control multiple LEDs in sequence. Next week I plan to make a start on designing and implementing the single traffic lights using assembly.

# Week 6 – Event Recognition and Interrupts

This week’s lecture covered event recognition. This included the comparison of polling vs interrupts. Polling is where an event is checked for through checking it repeatedly. Interrupts are, as the name suggests, a way to stop the currently running process and perform some other event in between.



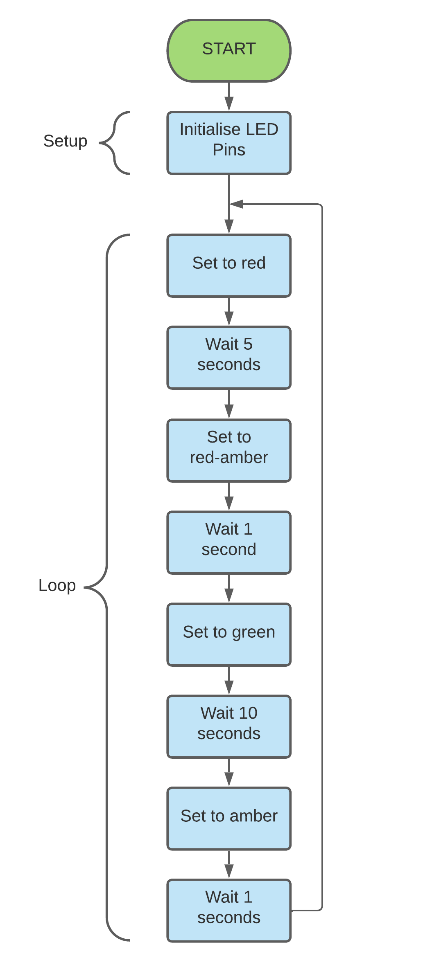
Polling vs Interrupt infographic

For a time-critical system, interrupts are a must. Polling will in most if not all cases check too late for an input. Interrupts are therefore highly desired in these situations.

Performing interrupts via assembly allows for greater efficiency with more granular detail how each step will be carried out making better use of the time compared to a less optimal output from a compiler.

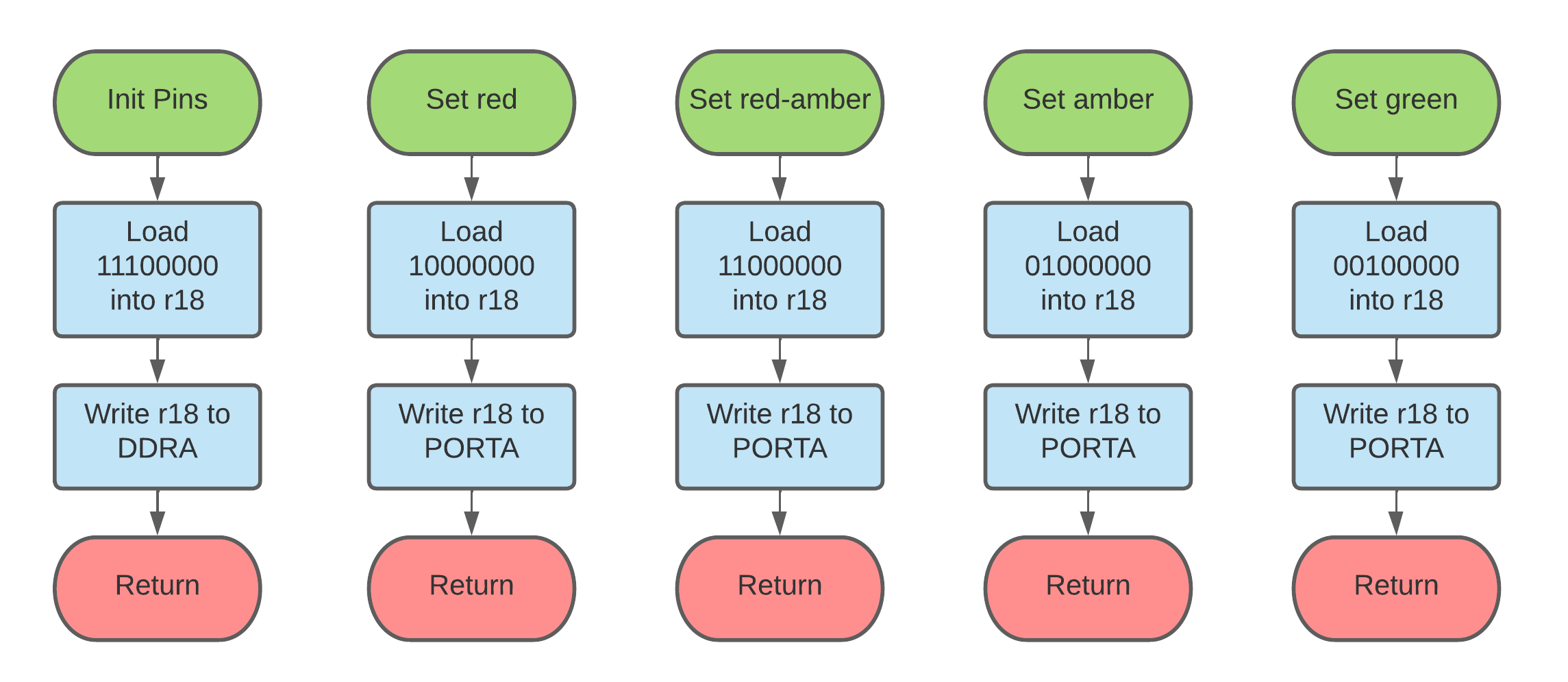
I attempted to create a basic sketch to implement this idea however could not get it working. My lack of knowledge of this concept and having no example to work from, I gave up this week and looked to using the idea for design instead.

With this knowledge fresh in mind and following from last week’s learning, in the lab I set to designing the single traffic lights using a flowchart. This remained somewhat the same from last semester from a top-down approach.



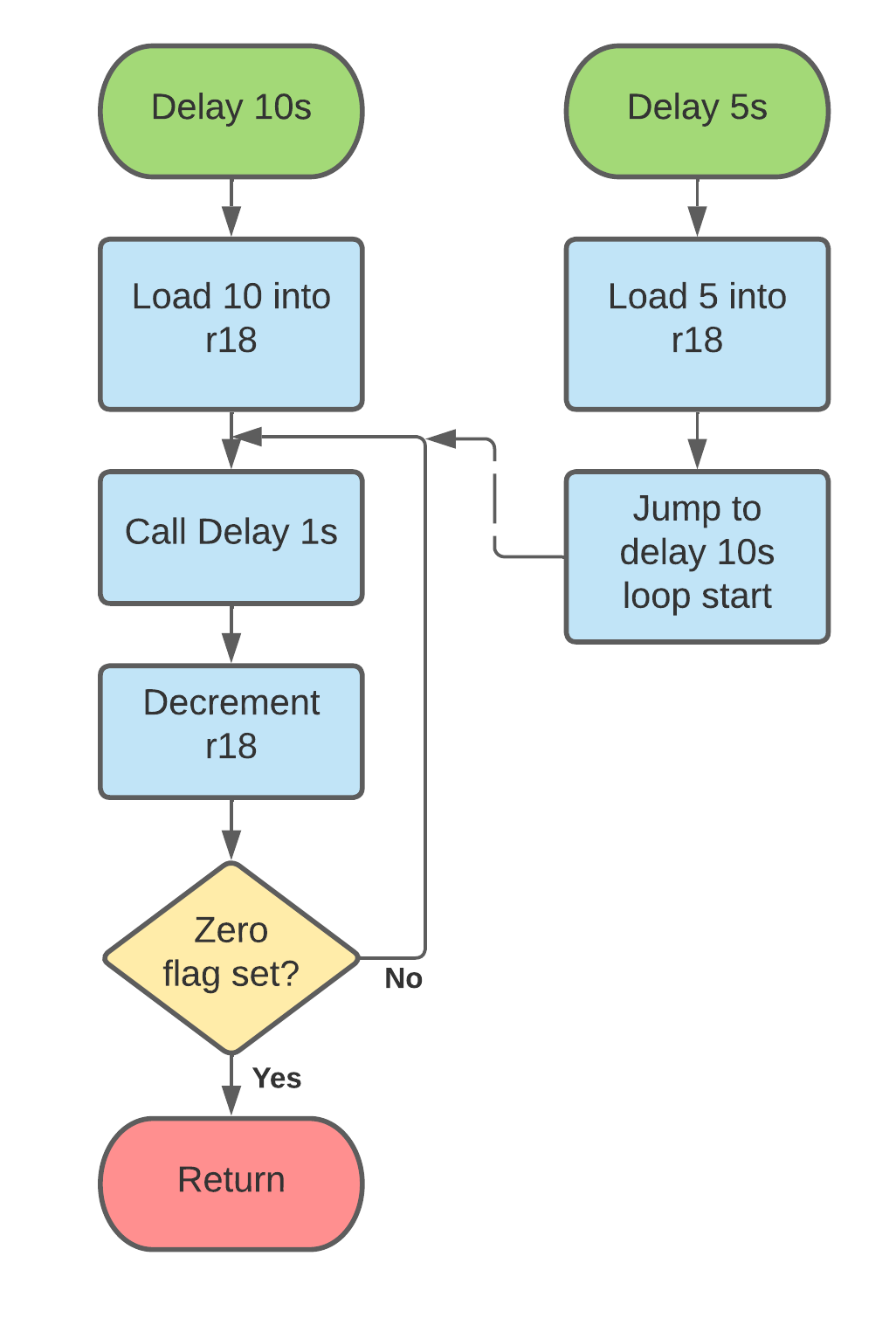
Single set of traffic lights overview flowchart

Using this overall design, I created a more detailed view of each section which would be implemented in assembly. Since my knowledge of time critical systems is minimal and having not worked with any examples, I felt it best to simply complete this section of the task the same as I had done with the strobe sketch.



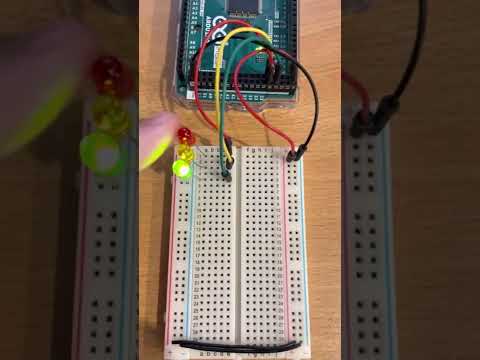
LED control functions used in overview flowchart

To be creative and reduce copies of code, I plan to implement 2 new delay functions for the longer delays. They will share a common loop that iterates over a specified register and depending on the function called, it will set a different value to the loop. At the end they will both use the same return statement to go back to the main application.



Custom delay functions that share a loop on how many 1s delays are called

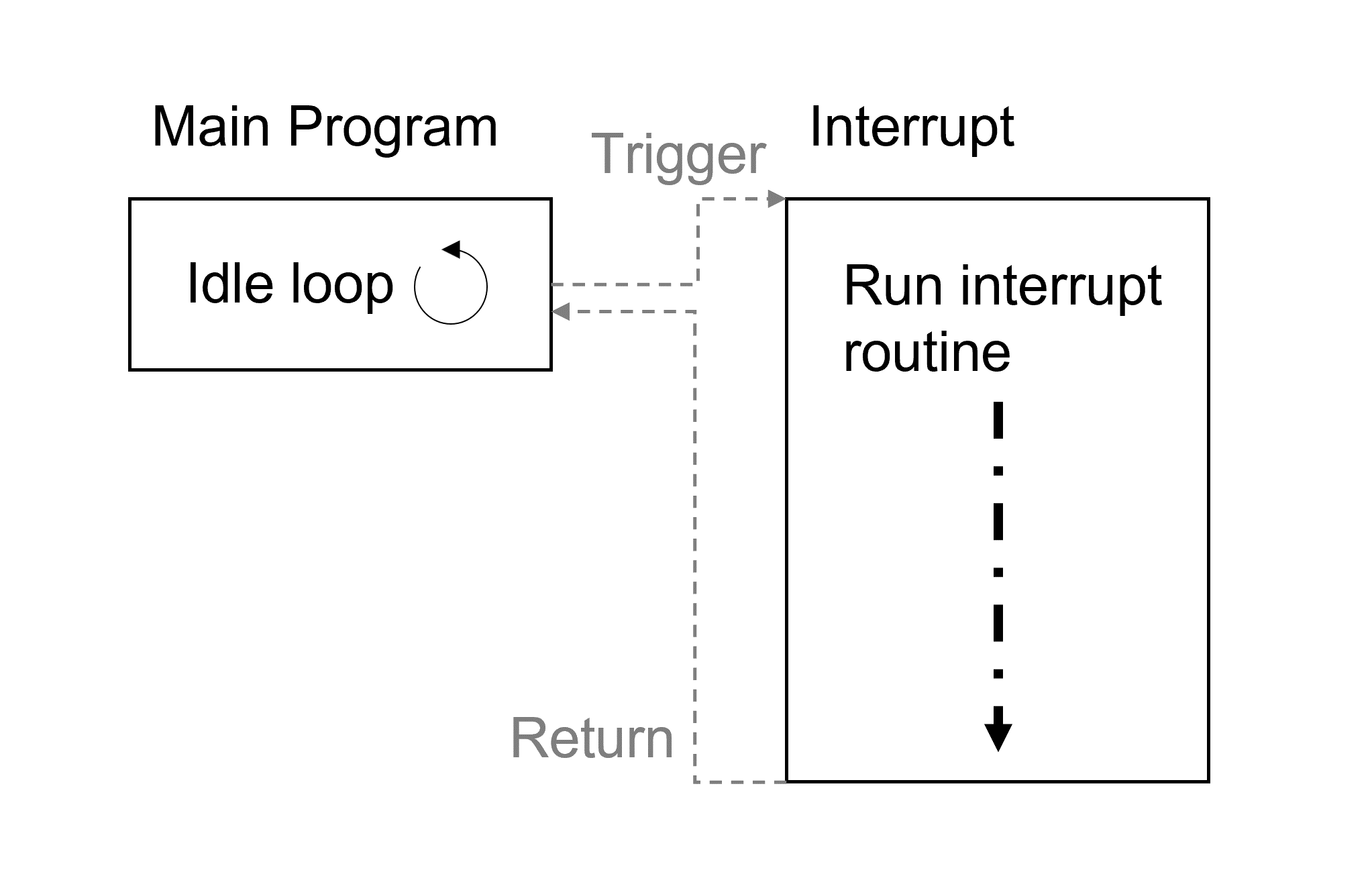
During this weekend I implemented the flowcharts into the “asm-single-traffic-lights” sketch in the week6 folder. This was an easy feat compared to the trial-and-error approach I had in past weeks learning about the different opcodes and methods of branching within the main program. A video of this sketch can be found below hosted on YouTube:

[](https://www.youtube.com/embed/sWYP3CpOmPE?feature=oembed)

Assembly single traffic light sketch running on Arduino mega board

# Week 7 – Interrupt Driven Systems

This week we covered more details on interrupt driven systems and covered the C implementation of external interrupts. We were shown how interrupts are far more efficient compared to polling where a check is made only at a certain point in the code. The ideology of an interrupt driven system is in having an infinite idle loop with interrupts being the only time the system state changes.

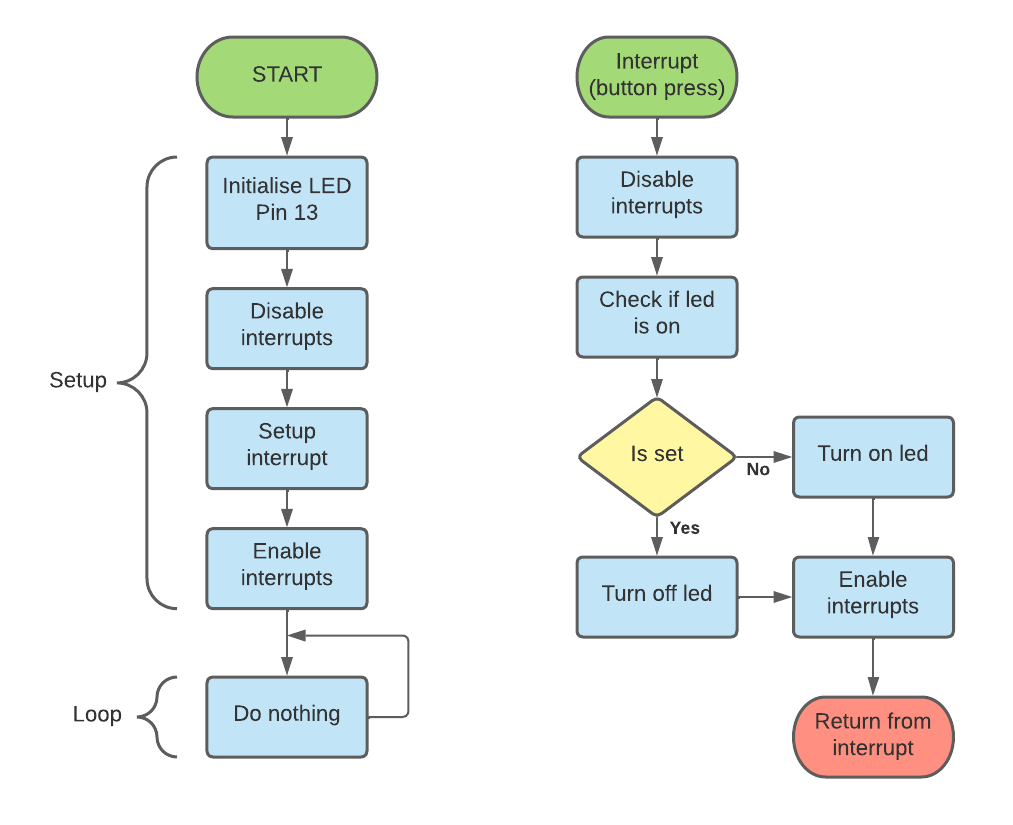


Interrupt driven system design

For my own research I read up on how this is implemented in pure assembly rather than in C since I did this last semester for my previous implementation of the traffic lights. The idea of an empty idle loop however is new to me so I would like to implement this when I understand it better.

There was not much material online for this, so I checked the documentation and learned how this is done using the very sparse examples.

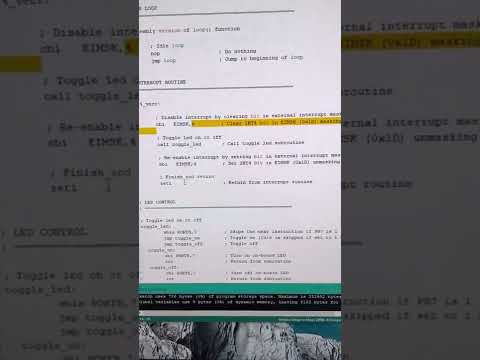
My goal was to implement the ideology of an interrupt driven system by having the main loop only contain an no operation and have the toggle occur within the interrupt function. The interrupt would be triggered by an external button. To show this in more detail I created the following overview in a flow chart



Toggle LED flowchart

Through a great deal of reading, trial-and-error I was able to toggle an LED on and off using an external interrupt. The code for this can be found in “week7” titled “asm-interrupt-toggle-led”

I noticed that the LED would sometimes seem to not change. From my experience with electronics, this issue is caused by the bounce within a mechanical switch.

[](https://www.youtube.com/embed/3UYKC6AAhR0?feature=oembed)

Video evidence of LED flashing on and off multiple times with a single press

To “debounce” the switch I implemented a hardware solution that uses the popular “555-timer”. By using a set resistor/capacitor combination, we can set a delay where the input is held high before resetting regardless of how many perceived “presses” are made by the bounce within the switch. In my case I used a 1MΩ resistor and 0.1μF capacitor.

A close-up of a circuit board

Description automatically generated with low confidence

Debounce circuit using “555-timer” with resistor (1MΩ) and capacitor (0.1μF)

[](https://www.youtube.com/embed/cxjE-lmXbHw?feature=oembed)

Video evidence of debounced switch with LED showing direct (blue) and delayed (white) signals

I plan to use this knowledge in the future for the two-way traffic lights that require a pedestrian button to trigger the pedestrian crossing. For now, I will keep this sketch ready for reference in the future.

# Week 8 – Timer Interrupts

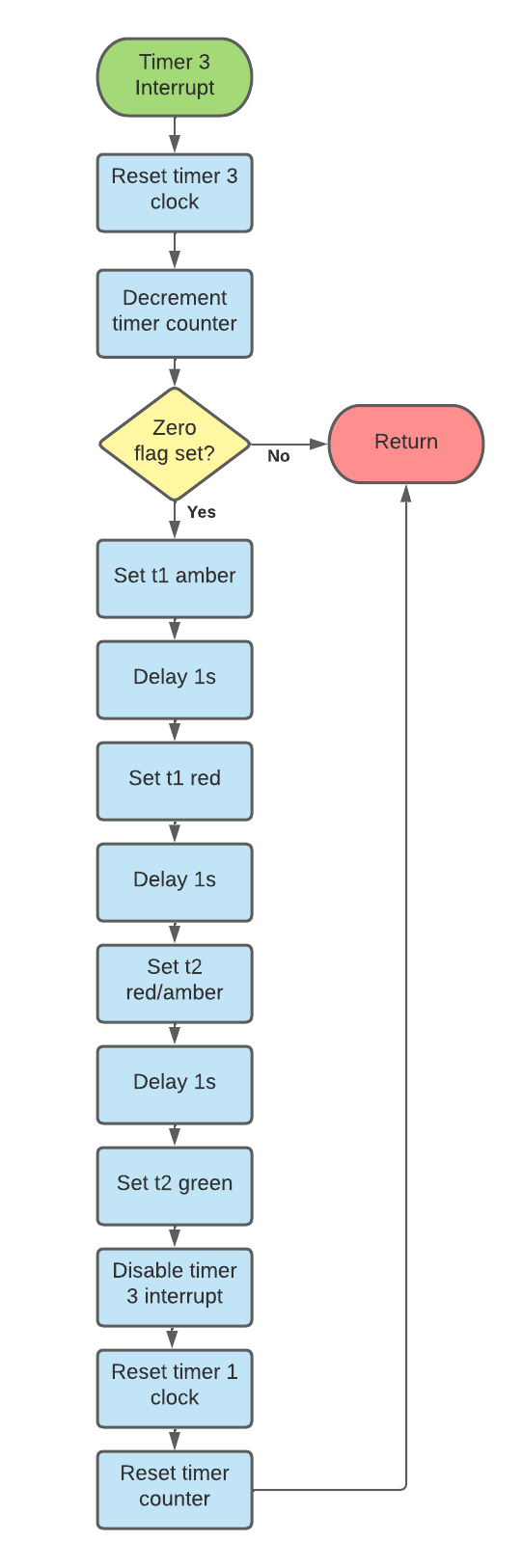
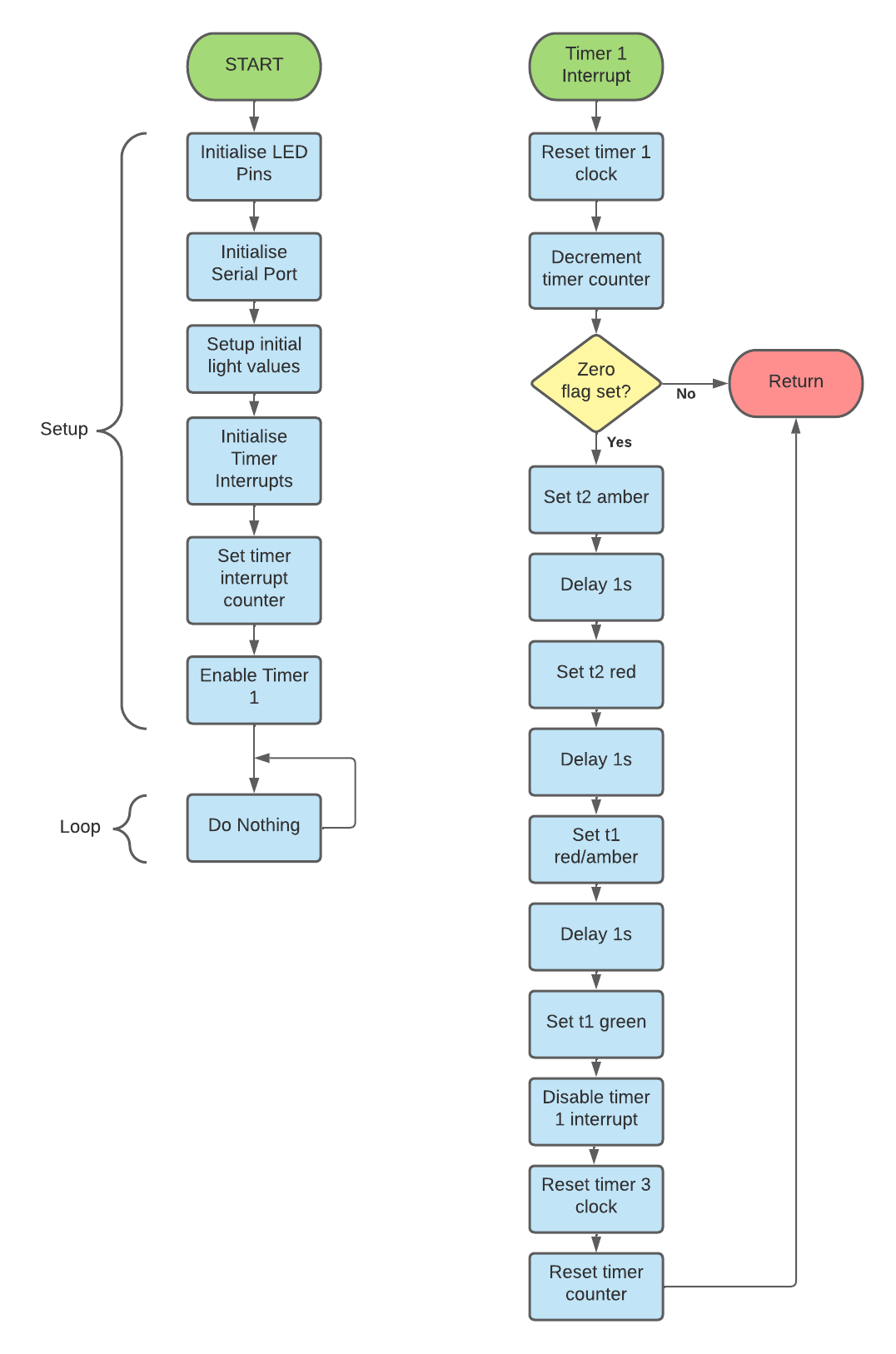
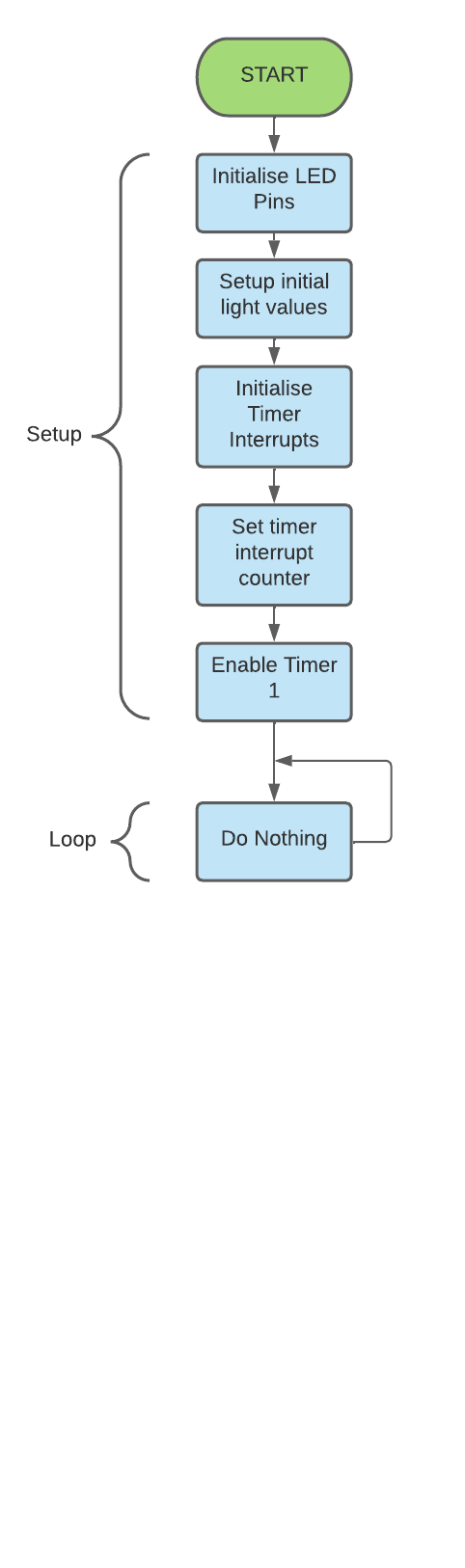
For our 8th week we covered how timer interrupts work. This allows for an internal timer to interrupt our program rather than an external source. I was wondering if we have an empty idle loop how we would run code without some external source of interrupt which this timer can.

Using this we can design systems that can run at a set interval and potentially take external interrupts at the same time. With multiple interrupts however we must consider the priority of each interrupt and how we will manage what can happen and when.

My learning points to using each interrupt’s mask to temporarily disable them while running a high priority interrupt and unmask to allow it to run afterwards.

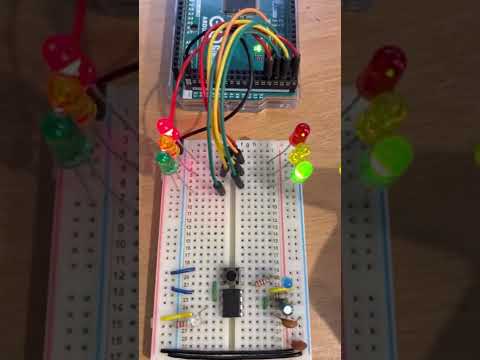
I found the slides somewhat contradictory to the ideology of having an empty idle loop as there are interrupt enables and disables. I will have to do more reading on this to understand why this is an option within this design philosophy.

For the lab I started the two-way traffic lights with the plan of using the interrupt driven ideology. I created a simple flow chart of how the system should function like that of the first semester however without an external interrupt and just use the internal interrupt timers.



Two-way traffic lights overview using 2 timer interrupts

This was at first quite complicated to implement so for practice I re-created the single lights with a single timer interrupt. This test sketch can be found under the “week8” folder titled “asm-interrupt-traffic-lights”. Using this knowledge, I created wrote the final interrupt driven two-way traffic lights. The final result is the “asm-two-way-traffic-lights” sketch in the week8 folder. This can be seen running in the following video:

[](https://www.youtube.com/embed/fu-MBikIrB4?feature=oembed)

Interrupt driven two-way traffic lights (without ped crossing) on YouTube

Since all code is contained within interrupts and gotten push button external interrupts working, adding a pedestrian crossing to this will be far easier than from scratch.­­­ I will plan and hopefully implement this before the easter break giving enough time to work on the number sorting task.

# Week 9 – Interrupts Continued and Serial Port

Week 9 is our last lecture before the easter break. We started by looking at more interrupts and how their priorities can be managed in either hardware or software. By using the priority built into the different pins and timers, we can allow certain interrupts to have priority over others.

For timers and external interrupts however, external always takes priority and can cause issues such as triggering a pedestrian crossing while it is already letting pedestrians cross. For this we can mask interrupts in software limiting their ability to interrupt until after the past one is finished.

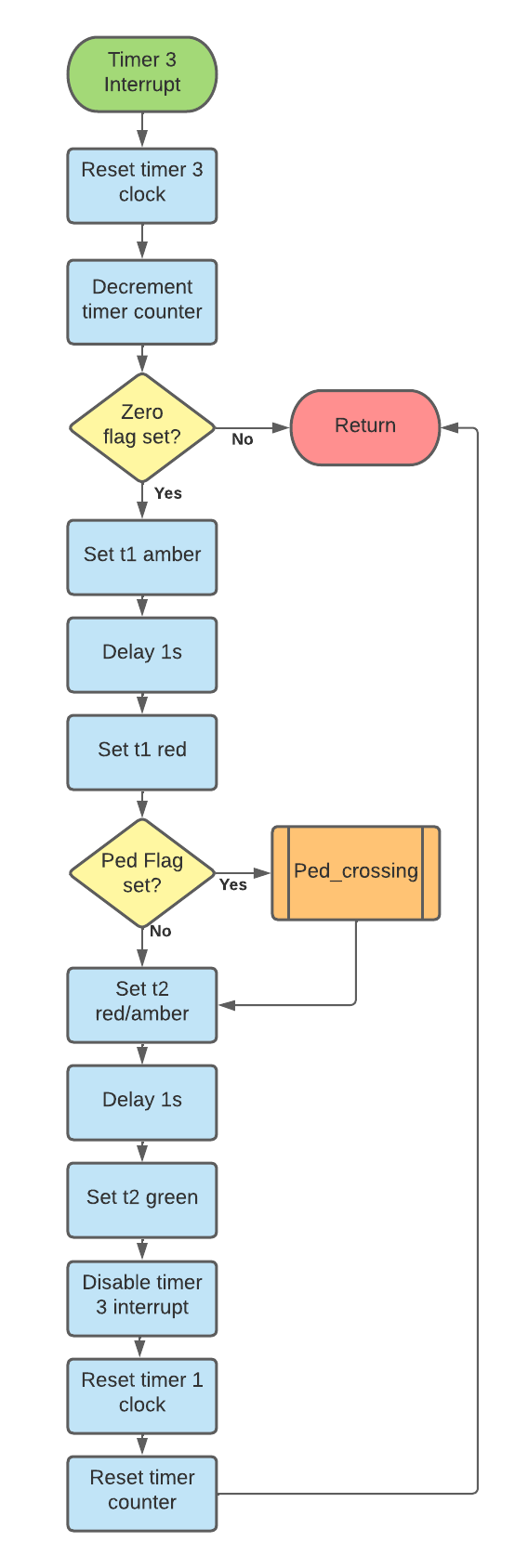
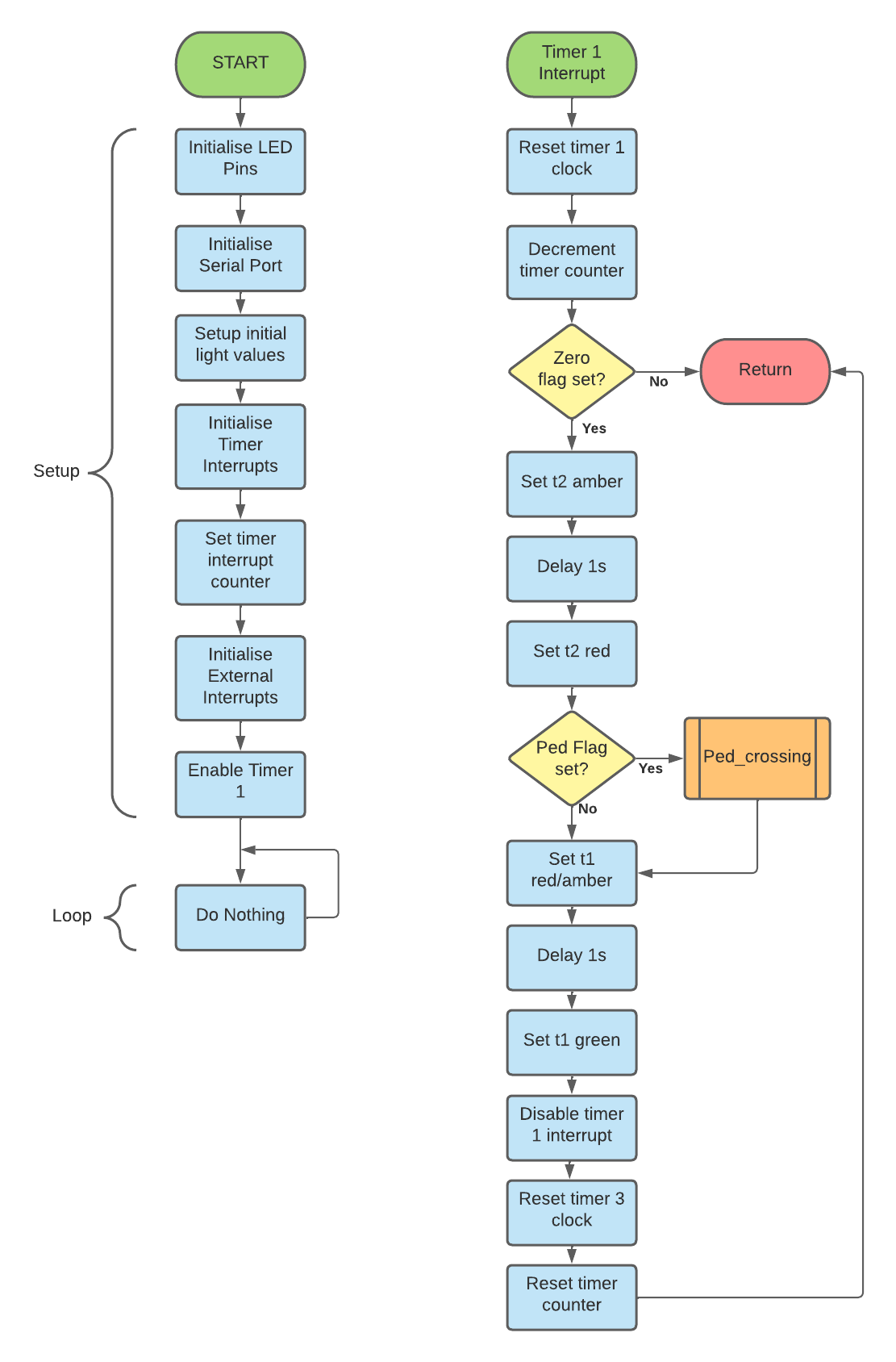
For my lab work I checked the requirements for the two-way traffic lights with pedestrian crossing and noticed that we need serial communication to print the status of the lights. This I hadn’t thought of and needed to learn how to implement this in assembly.

For this I built a test sketch that simply sends a single byte to the serial monitor. For this the single byte had to contain an ASCII character which could be interpreted by the serial monitor. I chose the letter ‘A’ which has the value of 65.

After much testing and working with the registers associated with the serial port, I got it transmitting the ‘A’ character at a regular interval via the main loop. This test sketch can be found in the “week9” folder titled “asm-serial”.

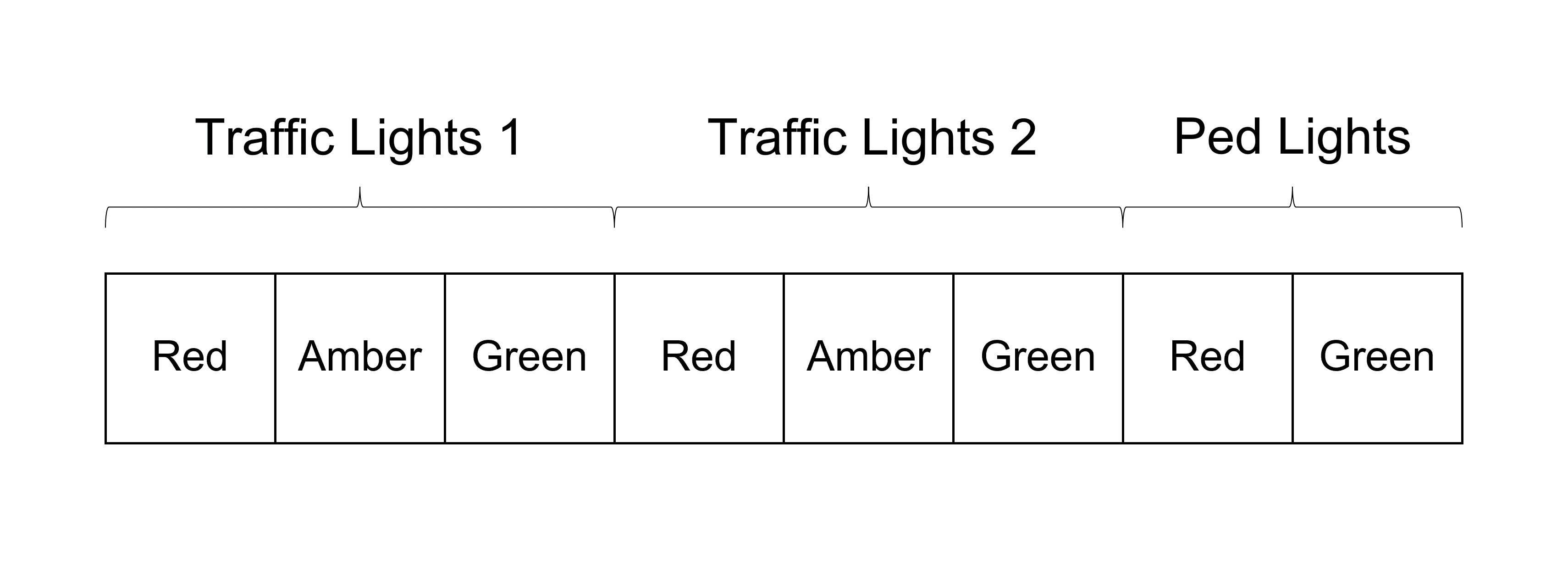
With this test example for the serial communication, I set to designing the two-way traffic lights to include a pedestrian crossing and sending the status of the LEDs to the serial port. The general overview I expanded to include a check for if the pedestrian crossing was pressed and sending the values to the serial monitor.

Diagram

Description automatically generated

Flowchart of two-way traffic lights including a pedestrian crossing and serial output

To store all values for the LEDs, I planned on using a single register which holds all values. This way I can mask and set different LEDs based on the combinations while only requiring a single register to work with. The layout would be as follows:



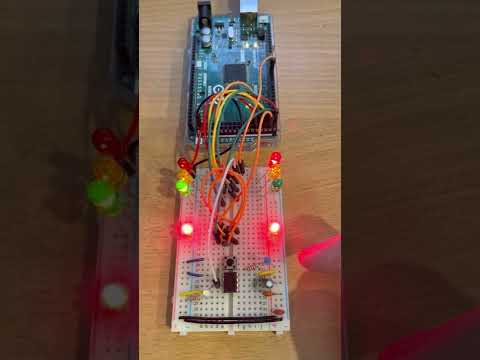
Single register for all LED combinations

After the lab I began implementing this by starting with the existing two-way lights from week 8 and adding the setup for the external interrupt referencing the toggle led sketch from week 7.

Next, I moved to adding the setup for USART and the transmit functions from last week resulting in an almost complete version with pedestrian crossing. I however hit the issue of serial communication requiring ASCII values rather than individual bits from registers.

For this I would have to print the led label such as red1, amber1, etc. along with the bit value set at each. To do this I used my bit shift that I did in the week 5 sketch “asm-strobe” and read the bits individually. By checking if they were set or not, I could write the ASCII representation of them to the serial port.

The final sketch “two-way-ped-traffic-lights” in the “week9” folder is fully interrupt driven, uses only assembly and satisfies the requirements for task 1 of our assignment brief. This sketch can be seen running below in the following YouTube video.

[](https://www.youtube.com/embed/84dJu7dfLzs?feature=oembed)

Video evidence of two-way traffic lights with pedestrian crossing sketch running

# Week 10 – Sorting Algorithm Preparation

I have included week 10 as my easter break study and work as I have yet to start on the sorting algorithm for our second task. I took the time to study how to do the key components of our sorting using only assembly.

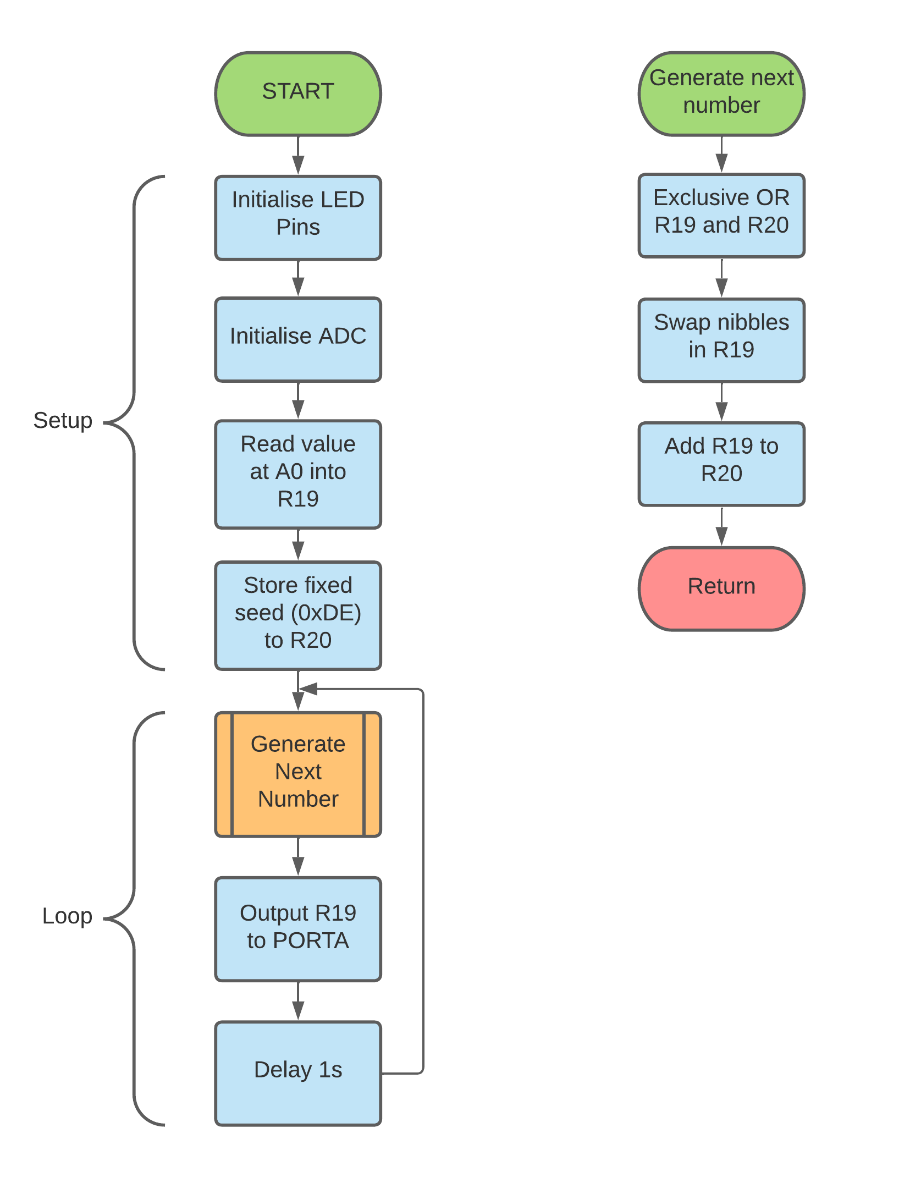
First is the random number generation. I first looked at how I could derive a single random value from the analogue input and simply repeat this 50 times to get the 50 values I would need to start.

Reading the analogue input required turning the analogue signal received and converting it into a digital value. This process called analogue to digital conversion requires setting up the ADMUX and ADCSRA registers to manage the pre-scalar and reference voltage the analogue signal is compared to.

After this setup we can read the value from the ADCL and ADCH registers. These registers store the 10-bit value the ADC reads in. I did hit the issue that doing this more than once in rapid succession does not yield random numbers but instead just 1.

To combat this, I searched for a method to calculate a pseudo-random value with a random seed which I could get from the floating analogue input. For this I found the method of XORing two value and swapping the first and last 4 bits of a byte. Adding this result to the second value seeds the next value to be generated.

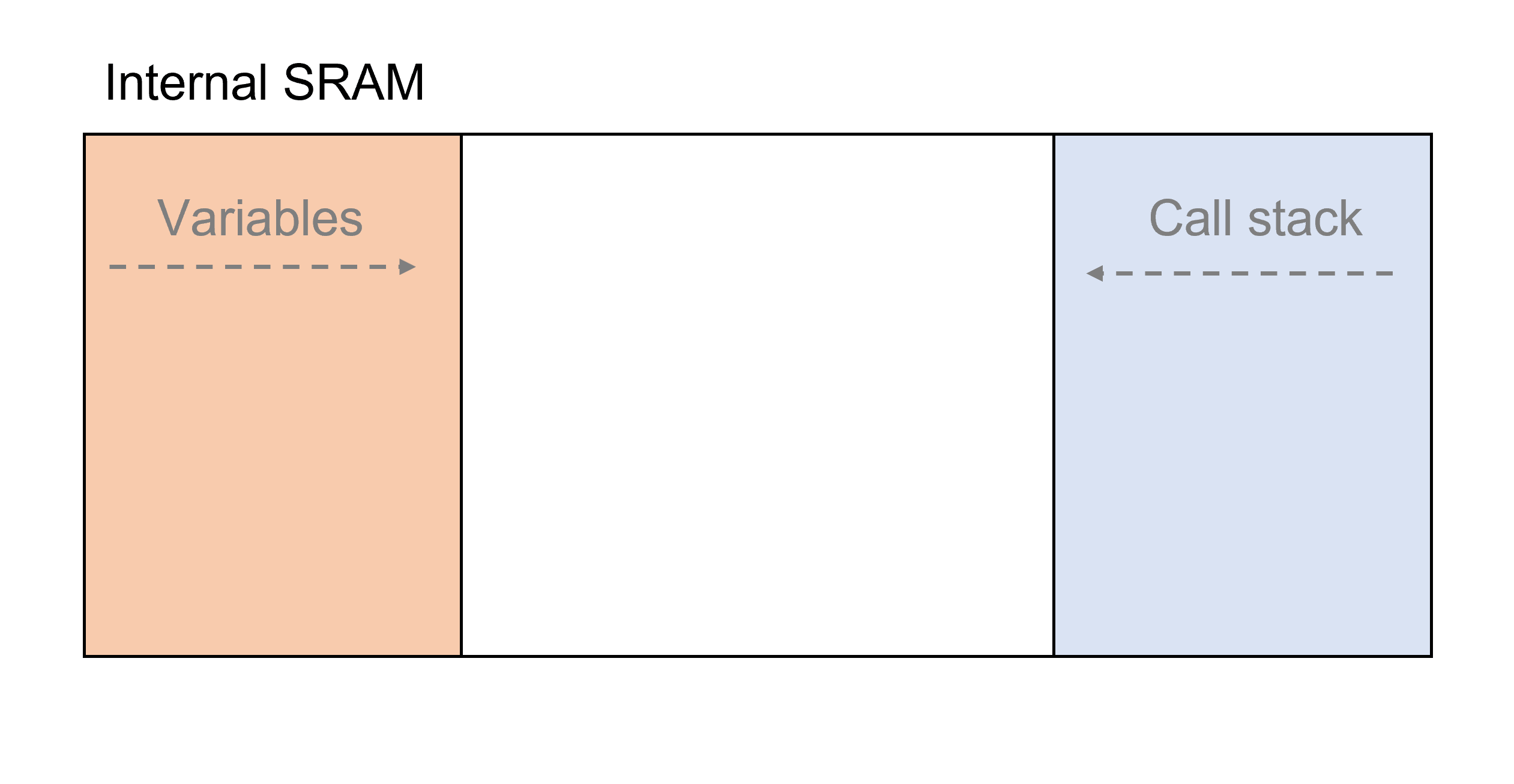
Through some testing I found that the values are not correlated and since these values are only required for sorting, this randomise would be sufficient to generate the values I would need. This example sketch can be found in the “week10” folder titled “asm-random-number”. Below is a flow chart of how this randomises function works.



How I implemented a randomize function for assembly

Next is the question of storing these values in memory to be operated on. The first 32 registers wont nearly be enough to store 50 values. From our memory map from the first week’s study shows the space at the end of memory called Internal SRAM. Here we have space we can use to store values.

We do have to be careful of the space we use as at the other end of this space is where the stack stores values. We can see this in the following infographic showing how variable values and the call stack share the space.



Visualization of Internal SRAM sharing space for variable value storage and the call stack

This space is cleared after every restart and power cycle, so it is perfect for temporary values such as our 50 randomized values. With optimal use of the stack where no overflow can occur, our program should be able to run indefinitely without the values being overwritten by the call stack.

To write these values one after another we need first a starting point and to be able to offset a reference to that point. To do this we can use the ST opcode which uses a word register such as the X,Y or Z register as a pointer to the memory location and the value to be stored in another working register.

Since we can change the value in the word registers, we can increment or decrement though the array and reset back to the starting position when complete.

Reading these values back is the same process however we use the LD opcode to read from the Z register referenced location and store the value into another working register.

A flowchart showing this process is shown here in how setting the Z-register and supplying a value in a normal working register will store it and reading from that register will restore it in the designated register.

Diagram

Description automatically generatedDiagram

Description automatically generated

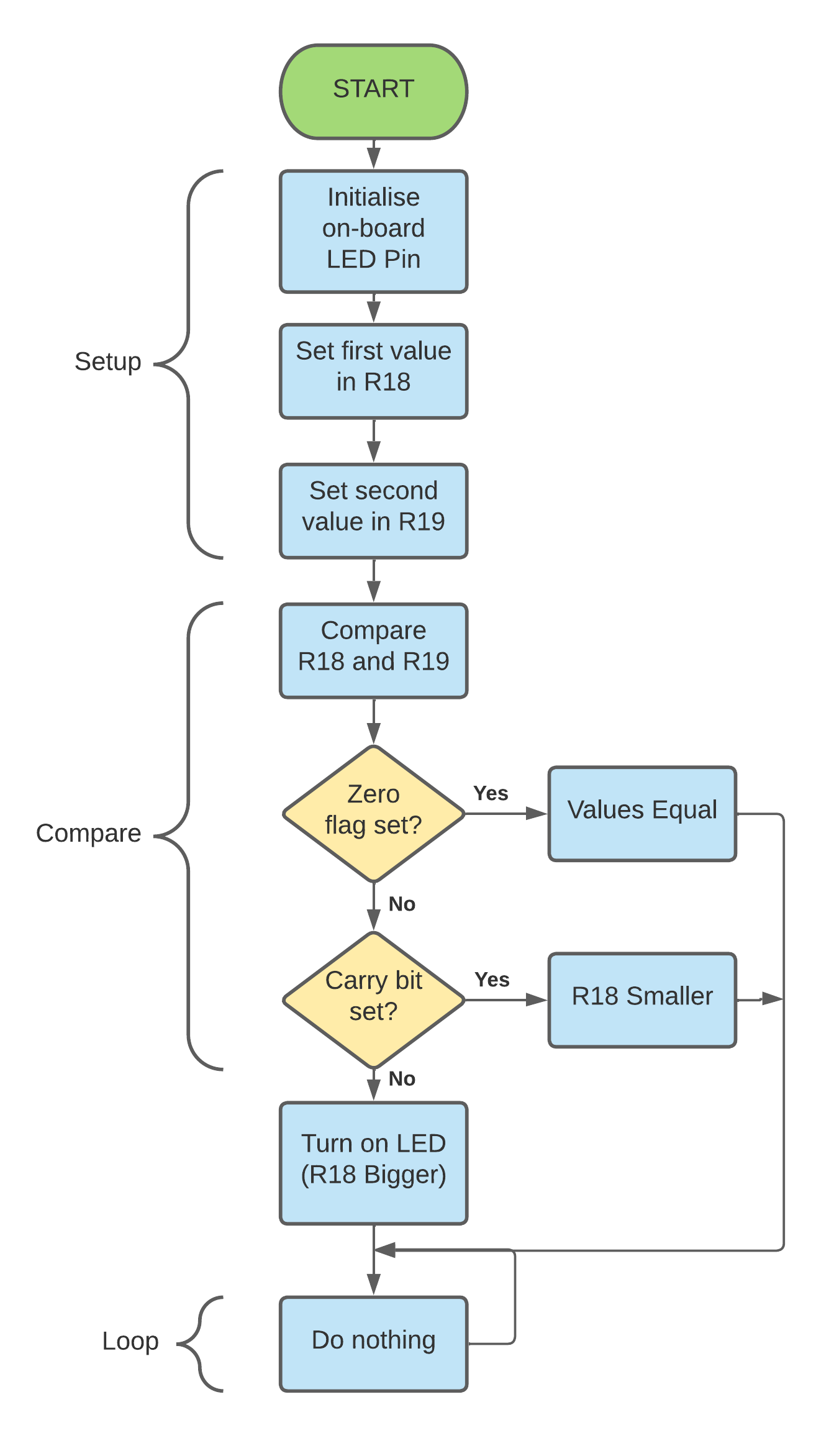
Reading from and writing to SRAM using the Z-register and working register

I created an example sketch for this under “week10” titled “asm-sram” where it simply stores a single value in the setup routine and fetches it in the main loop and sets the LEDs on PORTA accordingly. Changing the value in setup will yield the same byte on the LEDs.

Finally, we need to compare these values to see if they must be swapped or to move onto the next value. For this we can use the CP opcode which compares two values together.

This method of comparison under the hood negates the second number from the first and sets the zero or overflow flags depending on if the values are equal, smaller or larger. The value in the destination register does not change however so we can continue to use this in the future for the next compare.

To assess this, I created a final example to use as reference under “week10” titled “asm-compare” where two fixed values are checked and if the second value is larger than the first then the onboard LED is turned on and if smaller or equal, keeps the led off. The final flowchart for this week’s work shows how these functions.



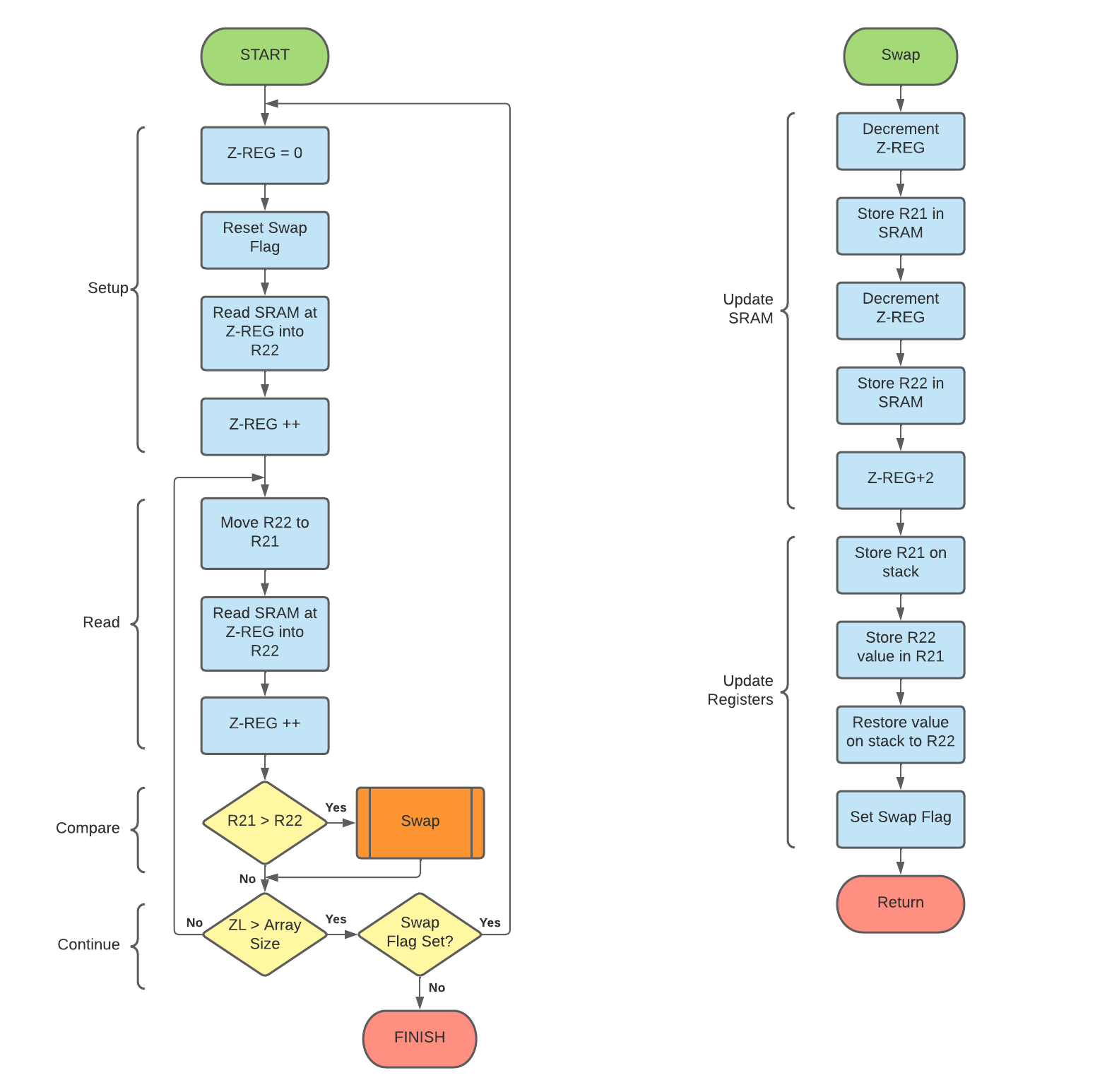
Comparison function for values stored in two different working registers

With the break almost over, I will take a few days to rest and starting next week I will move to implementing a sorting algorithm using these examples as a good foundation to start from.

# Week 11 – Sorting Algorithm

For our final week of learning and our first week back from easter break we covered the implementation of the bubble sorting algorithm with an example with the Motorola instruction set. This was to give us an idea of how values compared, swapped, and stored.

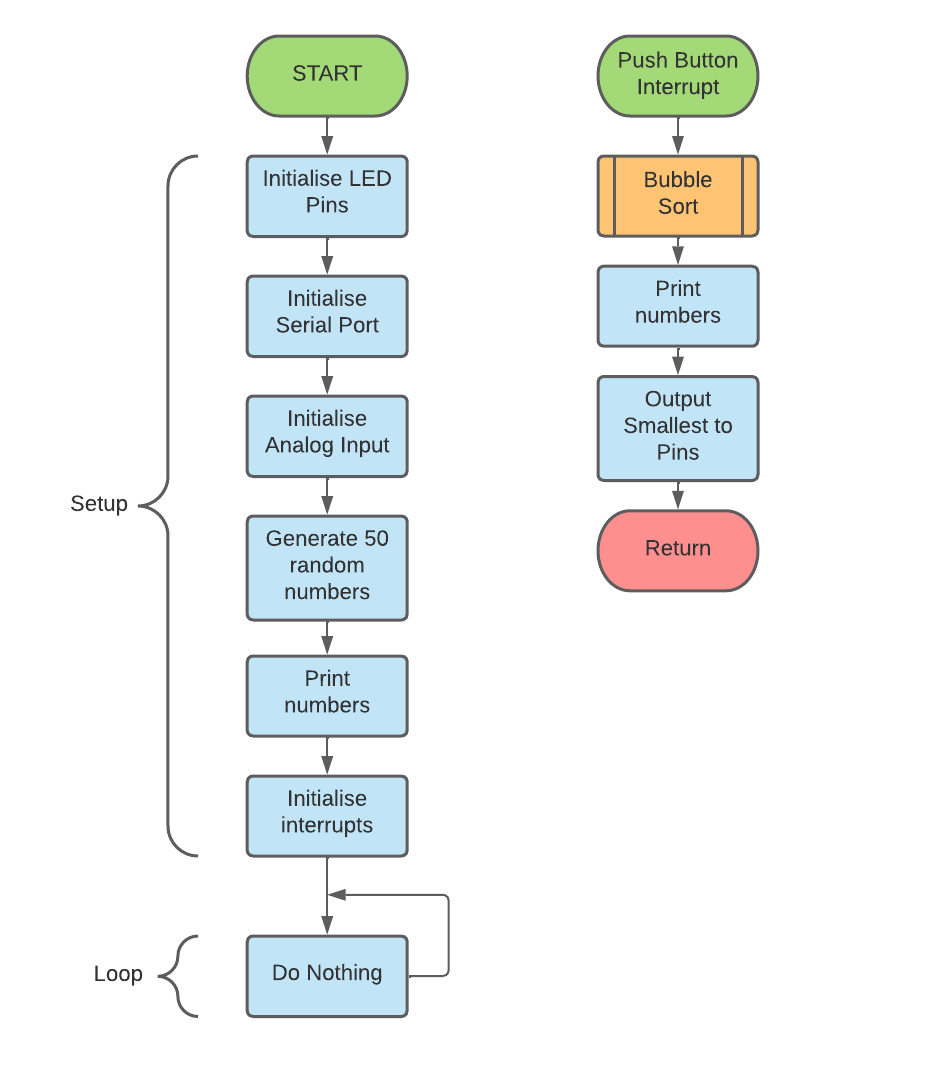
With my learning from the holidays and examples I had been working with, I set to creating a general flowchart of how this would work from a top-down perspective. This was quite simple as it abstracts away the greater details shown from my flow charts during the holiday break.



Flow chart of how bubble sort is carried out

Compared to the example flowchart shown during the lecture, I opted to use the memory location as a point of reference if the end had been reached and set a flag for when no swaps had been conducted to save on chip cycles. No point in testing for swaps if its already sorted!

Next was the flowchart for the full sketch that implements the algorithms from easter break and this new bubble sort design. The result is as follows:

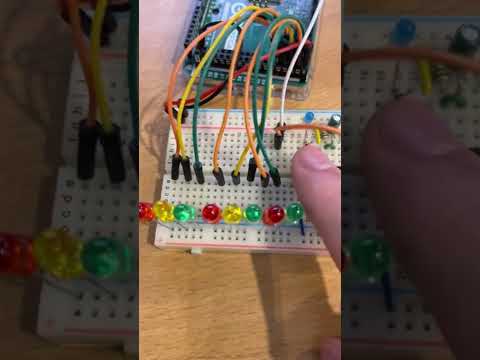


Overview plan of asm-bubble-sort sketch

I set to using my examples from easter and building the bubble sort algorithm. I used all examples from week 10 and added the serial functionality from week 9 to print the values to the serial monitor.

My final result follows the general flowchart on the page before utilizing the bubble sort algorithm the flowchart before and uses the smaller more detailed designs from my work over the easter break.

The result of all this work is the “asm-bubble-sort” sketch in the “week11” folder. It is a complete implementation of the bubble sort algorithm that is interrupt driven using an external interrupt and prints the results to the serial port both unsorted on power up and sorted on button push. Below is a link to this in operation on my Arduino Mega 2560

[](https://www.youtube.com/embed/hy7QZFxjrsE?feature=oembed)

My asm-bubble-sort sketch running on the Arduino Mega 2560

# Week 12 – Conclusion

Looking back at my knowledge of microprocessors, transistors and lower level languages such as the C language, I realise that there was this gap in my knowledge where assembly and microcode fits in perfectly.

Having been working to building my own 8-bit computing using my knowledge of electronics in order to bridge this gap, having learned an implementation of assembly via the AVR chipset and instruction set, I have come away with a greater appreciation of how the compiler abstracts the lower level away and can implement a more optimal implementation by hand if so desired.

In terms of the tasks assigned, I feel that I could have implemented a better solution after having gone through the learning process. For instance the traffic lights, I could have made better use of the interrupt mask and simply unmask it triggering the pedestrian crossing or with the number sorting having the heap sort similar to that of my first semester assignment done in the C language.

Overall this journey looking at the AVR chips under a microscope has been fun and I look forward to implementing this in my future learning journey through university.

# References

The Official Arduino Documentation Website, [Accessed 4th January 2022 – 7th May 2022]

ASCII Table Website (https://www.rapidtables.com/code/text/ascii-table.html)