Logic-Based Systems Showcased by Arduinos and Assembly Language

A Collection of System Architecture Lab Activities

Module: **Embedded Architecture and Programming**

Focus: **Designing a Variety of Logic-Based Systems**

A report originally created to fulfil the degree of

BSc (Hons) in Computer Science

Submitted 12/11/2018

Abstract

This paper demonstrates how decimal numbers can be converted into alternate number formats, including binary and hexadecimals, through the use of arithmetic and algorithms. The inputs can be either positive or negative. The paper also covers the use of arithmetic to add or subtract two 8-bit binary numbers, as well as developing binary functions in C. This paper then goes on to apply similar techniques in order to develop traffic light intersections, utilising both Arduinos and assembly. These are then further demonstrated via the use of communication protocols in order to send data, first between two wired Arduinos, followed by a wireless connection. The paper will show example code that makes this possible, as well as the differences between the various communications protocols.

Contents

Task 1: Designing Numeral Converters

7-Bit Decimal to Binary Converter

The first task was to make a 7-Bit decimal to binary converter for values in the range of 0 to 127. I started off by making a table of values which I would be able to look up at any point using HLOOKUP. I did this as I knew I would most likely have to repeat this action for multiple tasks. Once I had my table of bits corresponding to their decimal equivalent, I then used HLOOKUP to automatically take the correct values and input them into my decimal to binary converter, ready for the next process.

Binary Numbers	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	-128	64	32	16	8	4	2	1

Figure 1: A 7-bit binary number display that is used by all of the following figures in the task ahead.

The second step involved taking the users input and subtracting it from each bit in the table, starting from the highest bit number. I used IF functions to determine if the decimal was greater or equal to the value of the selected bit number. A 1 would be outputted in the cell below if true, otherwise the output would be 0, and the same decimal number was used for the next bit number. However, if a 1 was outputted, then the looked-up value from the table was subtracted from the current decimal number, and then the result would be used as the next decimal number. Once all 7-bits had been calculated, the final step was to concatenate all the 0's and 1's in the order they were outputted, starting from the most significant bit. This output was then fed back to the user as the final result.

Task 1: 7-Bit Decimal to Binary Converter (For Values 0 to 127)	
Decimal Input	Binary Output
31	**0011111**

Binary Conversion								Result	Check
Bit Number	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0		
Binary Bit Lookup Number	64	32	16	8	4	2	1		
Current Decimal Number	31	31	31	15	7	3	1	**0011111**	0011111
Is Bit No. < Decimal No.?	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE		
Conversion to Binary	**0**	**0**	**1**	**1**	**1**	**1**	**1**		
Resulting Decimal Number	31	31	15	7	3	1	0		

Binary Conversion	
Bit Number	Comments
Binary Bit Lookup Number	Looks up the value of the bit in a table of binary numbers in decimal form.
Current Decimal Number	The decimal number that is compared to the above value.
Is Bit No. < Decimal No.?	If the bit number looked up from the table is less than the decimal above, output True.
Conversion to Binary	If the above value is True, output a 1, otherwise output a 0.
Resulting Decimal Number	If the above output is 1, subtract the Binary Bit Lookup Number from the Current Decimal Number.

Figure 2: A 7-bit decimal to binary converter, with an input of 31 and a binary output of 0011111.

The area that the user inputs their decimal input is kept on a separate sheet to where the conversion takes place. The final result from the conversion table is then sent back to the sheet where the user inputted their number. The result produced is also checked against functions built into Excel, such as DEC2BIN (31) in this example, to make sure that is correct. Any values including a decimal point are converted to integers using the INT function. If a value outside the range is inputted into the system, the output prints "Value out of Range", to let the user know the conversion cannot be processed.

Task 1: 7-Bit Decimal to Binary Converter (For Values 0 to 127)	
Decimal Input	Binary Output
128	**Value Out of Range**

Figure 3: *The error message that is displayed if you try to input a number that cannot be converted.*

In order for negative values to be inputted into the converter, another bit was needed to show if the binary output was positive or negative. If the decimal input was positive, then the process would unfold the same as before, except with an extra 0 simply slotted in the front of the result. If the decimal input was negative however, then a 1 would be added to the front instead. This is because the largest bit number in decimal form is always negative when using two's complement. Another way of achieving the result would be to invert all the 0's and 1's, and then adding 1 to the answer, however this would require a carry in the event of two 1's being added together. This would have required more steps to achieve the same outcome, and therefore would have been less efficient.

Task 1a: Signed 8-Bit Decimal to Binary Converter (For Values -127 to 127)	
Decimal Input	Binary Output
-15	**11110001**

Binary Conversion									Result	Check
Bit Number	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0		
Binary Bit Lookup Number	-128	64	32	16	8	4	2	1		
Current Decimal Number	-15	113	49	17	1	1	1	1	**11110001**	11110001
Is Bit No. < Decimal No.?	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE		
Conversion to Binary	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**		
Resulting Decimal Number	113	49	17	1	1	1	1	0		

Binary Conversion	
Bit Number	Comments
Binary Bit Lookup Number	Looks up the value of the bit in a table of binary numbers in decimal form.
Current Decimal Number	The decimal number that is compared to the above value.
Is Bit No. < Decimal No.?	If the bit number looked up from the table is less than the decimal above, output True.
Conversion to Binary	If the above value is True, output a 1, otherwise output a 0.
Resulting Decimal Number	If the above output is 1, subtract the Binary Bit Lookup Number from the Current Decimal Number.

Figure 4: *A signed 8-bit decimal to binary converter, with an input of -15 and an output of 0011111.*

Decimal to Hexadecimal Converter

The second converter involved taking a decimal number between 0 and 127 and converting it into a hexadecimal equivalent. The first step was to find the modular of the user's decimal number, using the MOD function with a divisor of 16. The output of this would later form the second half of the hexadecimal output. The second step was to subtract this result from the original decimal input. This output would be used to form the first half of the output. Once the two numbers were found, they were checked to see if they equalled or were greater than 10. If either result contained multiple digits, they were then converted to a single hexadecimal digit using HLOOKUP on another table. Once the two halves were calculated, they were concatenated to form a hexadecimal output.

Hex Numbers	10	11	12	13	14	15	16
	A	B	C	D	E	F	0

Figure 5: *A display of hexadecimal numbers that is used by the following figures in the task ahead.*

Task 2: Decimal to Hexadecimal Converter (For Values 0 to 127)	
Decimal Input	Hexadecimal Output
83	**53**

Binary Conversion			Result	Check
Step by Step Guide	Number	Hex		
The Decimal Input	83	N/A		
Modular of the Input (Divisor of 16)	3	**3**	**53**	53
Decimal Input - The Modular Value	80	N/A		
Modular of the Above (Divisor of 16)	5	**5**		

Binary Conversion	
Step by Step Guide	Comments
The Decimal Input	This is the number that will be converted to hexadecimal.
Modular of the Input (Divisor of 16)	This number will form the second half of the hexadecimal.
Decimal Input - The Modular Value	This equation is used to find the first half of the hexadecimal.
Modular of the Above (Divisor of 16)	This number will form the first half of the hexadecimal.

Figure 6: A decimal to hexadecimal converter, with an input of 83 and a hexadecimal output of 53.

The process for allowing negative inputs required adding 00 on the front of positive numbers, and FF on the front of negative ones. One F represents the 16th and largest digit of hexadecimal. Two F's are used because 16 multiplied by 16 is 256, which is the total number of possible outputs when using an input that can be represented as an 8-bit number, the maximum this converter can calculate.

Task 2a: Signed Decimal to Hexadecimal Converter (For Values -127 to 127)	
Decimal Input	Hexadecimal Output
-83	**FFAD**

Binary Conversion			Result	Check
Step by Step Guide	Number	Hex		
The Decimal Input	-83	N/A		
Modular of the Input (Divisor of 16)	13	**D**	**FFAD**	FFAD
Decimal Input - The Modular Value	-96	N/A		
Modular of the Above (Divisor of 16)	10	**A**		

Binary Conversion	
Step by Step Guide	Comments
The Decimal Input	This is the number that will be converted to hexadecimal.
Modular of the Input (Divisor of 16)	This number will form the second half of the hexadecimal.
Decimal Input - The Modular Value	This equation is used to find the first half of the hexadecimal.
Modular of the Above (Divisor of 16)	This number will form the first half of the hexadecimal.

Figure 7: A signed decimal to hexadecimal converter, with an input of -83 and an output of FFAD.

Decimal to Octal Converter

Converting a decimal input to an octal output is similar to the previous converter, just requiring a few more steps to produce an output. To start off, the input was converted to an absolute value, so any negative values would be treated as a positive until later on in the process. A modular of this result was then taken, with a divisor of 8, which would later form the last digit of the octal output. To find the middle digit, an absolute value was taken again, but this time divided by 8. If this results in the fraction, the number is rounded down to become an integer. A modular of this result was then taken, and like the previous time a divisor of 8 was used. This number would later form the second digit of the output. To find the first digit, an absolute value was found again, this time dividing it by 64 (equal to 8^2). A modular of the rounded down value, with a divisor or 8, produced the first digit. The 3 digits were then concatenated in the order previously stated to form the final octal output.

Task 3: Decimal to Octal Converter (For Values 0 to 127)	
Decimal Input	Octal Output
83	**123**

Binary Conversion			Result	Check
Step by Step Guide		Number		
Input	The Decimal Input	83		
Oct No.	Absolute Value of the Input	83		
Part 3	Modular of the Above (Divisor of 8)	**3**	**123**	123
Oct No.	Abs. Dec. Input / 8 (Rounded Down)	10		
Part 2	Modular of the Above (Divisor of 8)	**2**		
Oct No.	Abs. Dec. Input / 64 (Rounded Down)	1		
Part 1	Modular of the Above (Divisor of 8)	**1**		

Binary Conversion	
Step by Step Guide		Comments
Input	The Decimal Input	This is the number that will be converted to octal.
Oct No.	Absolute Value of the Input	The input needs to be positive for it to be converted.
Part 3	Modular of the Above (Divisor of 8)	This will be the last digit in the converted octal number.
Oct No.	Abs. Dec. Input / 8 (Rounded Down)	To find the next digit, the input must be divided by 8^1.
Part 2	Modular of the Above (Divisor of 8)	This will be the middle digit in the converted octal number.
Oct No.	Abs. Dec. Input / 64 (Rounded Down)	To find the next digit, the input must be divided by 8^2.
Part 1	Modular of the Above (Divisor of 8)	This will be the first digit in the converted octal number.

Figure 8: A decimal to octal converter, with an input of 83 and a calculated octal output of 123.

Two more steps were needed when adapting the converter for negative numbers, the first being to check if the original input was indeed negative. If this was true, the second step was to subtract the concatenated value from 778. I discovered after some research that the reason for it being this number is due to the nature of base 8 only containing the digits 0 – 7, and the process of finding the modular of the decimal input is completed three times, so the smallest negative input creates an output of 777. Since the smallest negative input possible is -1, the number that the concatenated value is subtracted from must be 777 minus -1, which is equal to 778.

Task 3a: Signed Decimal to Octal Converter (For Values -127 to 127)	
Decimal Input	Octal Output
-83	**655**

Binary Conversion			Result	Check
Step by Step Guide		Number		
Input	The Decimal Input	-83		
Oct No.	Absolute Value of the Input	83		
Part 3	Modular of the Above (Divisor of 8)	**3**		
Oct No.	Abs. Dec. Input / 8 (Rounded Down)	10	**655**	655
Part 2	Modular of the Above (Divisor of 8)	**2**		
Oct No.	Abs. Dec. Input / 64 (Rounded Down)	1		
Part 1	Modular of the Above (Divisor of 8)	**1**		
Final	Concatenated Number (Parts 1, 2 & 3)	**123**		
Step	Is the Input a Positive Number?	FALSE		

Binary Conversion		
Step by Step Guide		Comments
Input	The Decimal Input	This is the number that will be converted to octal.
Oct No.	Absolute Value of the Input	The input needs to be positive for it to be converted.
Part 3	Modular of the Above (Divisor of 8)	This will be the last digit in the converted octal number.
Oct No.	Abs. Dec. Input / 8 (Rounded Down)	To find the next digit, the input must be divided by 8^1.
Part 2	Modular of the Above (Divisor of 8)	This will be the middle digit in the converted octal number.
Oct No.	Abs. Dec. Input / 64 (Rounded Down)	To find the next digit, the input must be divided by 8^2.
Part 1	Modular of the Above (Divisor of 8)	This will be the first digit in the converted octal number.
Final	Concatenated Number (Parts 1, 2 & 3)	Slots the individual parts together to make a final number.
Step	Is the Input a Positive Number?	If the input is negative, subtract the number above from 778.

Figure 9: A signed decimal to octal converter, with an input of -83 and a calculated output of 655.

7-Bit Binary Multi-Output Converter

The next converter involved taking a 7-bit binary number and converting it into multiple outputs, including decimal, hexadecimal and octal. To convert the input into a decimal number, the process of using HLOOKUP to find the equivalent table values that was utilised before was used again here.

Task 4: 7-Bit Binary Converter (For Values 0000000 to 1111111, or 0 to 127)									
Binary Input							Equivalent Outputs	
No Sign	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Binary Number	Decimal
Bit Used	1	1	1	0	0	1	1	**1110011**	**115**

Binary Conversion									Result	Check
Bit Number	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0		**1110011**	1110011
Binary Input	1	1	1	0	0	1	1			
Dec Conversion	64	32	16	0	0	2	1		**115**	115

Figure 10: The decimal section of the multi-output binary converter, with a decimal output of 115.

The next output was a hexadecimal value, which involved breaking down the 7-bit binary input into 2 smaller binary inputs, a 3-bit and 4-bit binary number respectively. The first 3 bits were used to form one value, the largest being 7 if all bits were 1's, while the last 4 bits were used to form another, with the largest being 15 if all bits were 1's. Because hexadecimal values convert numbers with multiple digits into a single value, anything over 10 is converted into a corresponding letter using HLOOKUP, similarly to task 2. In this case, the 15 would be converted into an F.

Task 4: 7-Bit Binary Converter (For Values 0000000 to 1111111, or 0 to 127)									
Binary Input								Equivalent Outputs	
No Sign	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Binary Number	Hex
Bit Used	1	1	1	1	1	1	1	**1111111**	**7F**

Binary Conversion								Result	Check
Bit Number	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	**1111111**	1111111
Binary Input	1	1	1	1	1	1	1		
Hex Conversion	4	2	1	8	4	2	1	**7F**	7F
	7			15					
	7			**F**					

Figure 11: *The hexadecimal section of the multi-output binary converter, with an output of 7F.*

The final output for this task was an octal value, which meant breaking down the 7-bit binary input into 2 smaller binary inputs yet again. The second binary number, in this case 15, was used with a divisor of 8 to produce a remainder of 7. This would later form the middle digit of the octal output. Finally, the first digit was found by taking the second binary number and dividing it by 8. The result is rounded down to the nearest integer, resulting in a value of 1 on this occasion. Therefore, the final octal output would be a concatenated value of the 3 outputs above, resulting in a value of 177.

Task 4: 7-Bit Binary Converter (For Values 0000000 to 1111111, or 0 to 127)									
Binary Input								Equivalent Outputs	
No Sign	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Binary Number	Octal
Bit Used	1	1	1	1	1	1	1	**1111111**	**177**

Binary Conversion								Result	Check
Bit Number	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	**1111111**	1111111
Binary Input	1	1	1	1	1	1	1		
Oct Conversion	8	4	2	1	4	2	1	**177**	177
	15				7				
	7				**1**				

Figure 12: *The octal section of the multi-output binary converter, with an octal output of 177.*

After I completed all three functions separately, they were then slotted together to use the same input for all three conversions. This allowed for a side by side comparison between binary, decimal, hexadecimal and octal values, ranging from 0 to 127 in the decimal equivalent. The final version of this task can be seen below, with each of the three functions working simultaneously.

Task 4: 7-Bit Binary Converter (For Values 0000000 to 1111111, or 0 to 127)											
Binary Input								Equivalent Outputs			
No Sign	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Binary Number	Decimal	Hex	Octal
Bit Used	1	1	1	0	0	1	1	**1110011**	**115**	**73**	**163**

Binary Conversion								Result	Check
Bit Number	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	**1111111**	1111111
Binary Input	1	1	1	1	1	1	1		
Dec Conversion	64	32	16	8	4	2	1	**127**	127
Hex Conversion	4	2	1	8	4	2	1	**7F**	7F
	7			15					
	7			F					
Oct Conversion	8	4	2	1	4	2	1	**177**	177
	15				7				
	7				1				

Binary Conversion	
Bit Number	Comments
Binary Input	This is the input that is used when converting to different number types.
Dec Conversion	Looks up the value of the bit in a table of binary numbers in decimal form.
Hex Conversion	The 7-Bit binary number is first of all converted to a 3-Bit number and a 4-Bit number.
	The sum of the numbers above are converted to form the first and second half of the hexadecimal.
	If the result is between 10 and 15 then it will converted to it's corresponding hexadecimal letter.
Oct Conversion	The 7-Bit binary number is first of all converted to a 3-Bit number and a 4-Bit number.
	The sum of the numbers above are converted to form the first and second half of the octal number.
	A modular of 8 for the first value above (14 in this case), are used to form the final octal number.

Figure 13: A combination of the previous 3 figures, producing a multi-output binary converter.

Like the previous tasks, each process had to be slightly modified to allow for negative values to be converted. The decimal conversion simply needed another bit added on the front to represent the sign of the number. If the first bit is equal to 1, the corresponding value is equal to -128.

Binary Conversion								Result	Check
Bit Number	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	**11111111**	11111111
Binary Input	1	1	1	1	1	1	1		
Dec Conversion	-128	64	32	16	8	4	2	**-1**	-1

To be able to produce a negative hexadecimal value, the same idea of splitting the binary input into 2 smaller binary numbers still occurs. However, since the input is now an 8-bit number due to an extra bit used to show if the number is positive or negative, it is split into two 4-bit numbers instead. The first 4 bits were used to form one value, while the last 4 bits were used to form another.

Binary Conversion								Result	Check
Bit Number	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	**11111111**	11111111
Binary Input	1	1	1	1	1	1	1		
Dec Conversion	-128	64	32	16	8	4	2	**-1**	-1
Hex Conversion	8	4	2	1	8	4	2	**FFFF**	FFFF
	15				15				
	F				F				

The process of finding a negative octal value compared to just a positive one has a slight change, but otherwise it is practically the same. If the most significant bit of the binary input is a 1, then the other 7-bits are inverted for the rest of the process, with all 0's becoming 1's, and all 1's becoming 0's. The converter then follows the same steps as the previous iteration, splitting up the remaining 7-bits into 4-bit and 3-bit binary numbers. The first 4 bits were used to form one value, while the last 3 bits were used to form another. The steps remain the same up until the output is about to be produced. If the original binary input was a positive number, the output steps are the same as before. However, if the input is negative, then the result is subtracted from 777.

Binary Conversion								Result	Check
Bit Number	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	**11111111**	11111111
Binary Input	1	1	1	1	1	1	1		
Dec Conversion	-128	64	32	16	8	4	2	**-1**	-1
Hex Conversion	8	4	2	1	8	4	2	**FFFF**	FFFF
		15				15			
		F				F			
Oct Conversion	777	0	0	0	0	0	0	**777**	777
		0	0	0	0	0	0		
			0			0			
			0			0			
			0			**777**			

Binary Conversion	
Bit Number	Comments
Binary Input	This is the input that is used when converting to different number types.
Dec Conversion	Looks up the value of the bit in a table of binary numbers in decimal form.
Hex Conversion	The 8-Bit binary number is first of all converted to a 3-Bit number and a 4-Bit number.
	The sum of the numbers above are converted to form the first and second half of the hexadecimal.
	Each result is converted to the matching letter. If the binary number is negative, FF is slotted in front.
Oct Conversion	If the input is negative, the 0's and 1's are inverted in this step before moving on with the process.
	The 8-Bit binary number is next converted to a 3-Bit number and a 4-Bit number.
	The sum of the numbers above are converted to form the first and second half of the octal number.
	A modular of 8 for the first value above (0 in this case), are used to get to the final octal number.
	If the input is positive, the answer is from the process above (0), otherwise it is subtracted from 777.

Figure 14: A signed version of the multi-output binary converter created from the previous 4 figures.

Decimal Values Converter

The final conversion involved taking a signed binary number to produce a decimal equivalent, this time using values less than 1 in the 3 least significant bits via the use of a decimal point. The same process was performed except when it came to calculating the last 3-bits. The values of the decimal digits were not inverted like the others if the binary input was negative.

Binary Numbers (Inc. Decimal Point)	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	-128	64	32	16	8	4	2	1	0.5	0.25	0.125

Task 5: Signed 11-Bit Binary to Decimal Converter (For Values 00000000.000 to 11111111.111, or 0 to 127.875)													
Binary Input											Binary Number	Decimal Output	
Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	**00001100.100**	**12.5**	
0	0	0	0	1	1	0	0	1	0	0			

Binary Conversion												Result	Check
Bit Number	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	**00001100.100**	00001100.100
Binary Number	0	0	0	0	1	1	0	0	1	0	0		
Dec Conversion	0	0	0	0	8	4	0	0	0.5	0	0	**12.5**	12.5

An example of a negative binary inputted into the converter can be seen below. Whilst the bits to the left of the decimal point are inverted from 0's to 1's and 1's to 0's, like in the previous tasks that use two's complement to produce negative values, the bits to the right remain the same.

Binary Conversion												Result	Check
Bit Number	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	**11111111.100**	11111111.100
Binary Number	1	1	1	1	1	1	1	1	1	0	0		
Dec Conversion	-128	64	32	16	8	4	2	1	0.5	0	0	**-1.5**	-1.5

Figure 15: A signed decimal to binary converter, capable of calculating up to 3 significant figures.

8-Bit Binary Addition and Subtraction

The following task was to make a binary calculator that could add or subtract two 8-bit binary inputs. I decided to break down the task into two smaller steps, creating an adder, followed by a subtractor. After researching about adders online, I discovered a few diagrams that made it easier to visualise. I was then able to successfully create a full adder that I could use for my binary calculator. The next step was to then find a circuit diagram of a subtractor that I could recreate using logic functions.

I was on my way to completing it when I realised that the logic behind the subtractor was quite similar to the adder I had previously built for the first half of the task. If I were building the calculator using logic gates on a circuit board, I would try to use the least number possible in order to make the process as efficient as possible. Due to this, I attempted a similar approach with both my adder and subtractor. Although this meant I required a third input, I was able to remove almost half of the logic by combined the two processes together in order for them to both run on the same logic gates. The logic circuit I used in the end was one I drew myself after seeing the parallels between the processes.
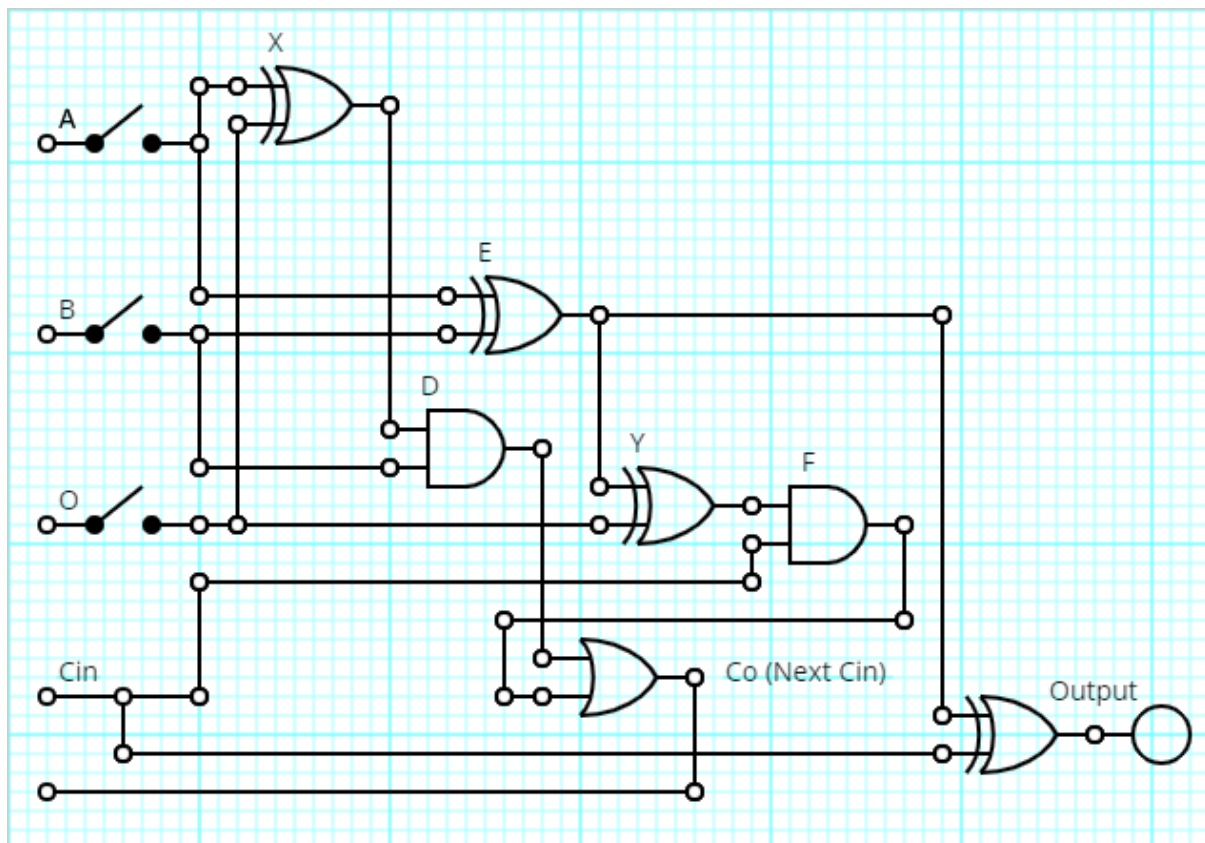
Figure 16: *A logic gate system I created representing binary arithmetic to make it easier to visualise.*
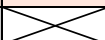
The first step I took when merging the two components together was to create the three inputs, with the first and second being the two 8-bit binary numbers, whilst the third was for the operation. A logic state of 0 indicated the calculator would be performing addition, while a logic state of 1 indicated it would be performing subtraction. If something else was inputted into the calculator for the operation, the process would simply output "Incorrect Output Detected", so that the user knew to change it before the calculator could move on the next step. This is where the merge between addition and subtraction first needed to be worked out. If the user wanted to add their two binary numbers, this step would simply produce a copy of the first input. If the user wanted to perform

subtraction, then this step would produce an inverted copy instead, with all 0's becoming 1's and all 1's becoming 0's. This was called process X, so that it was easier to understand the logic when looking back through the various steps. It is important to note that the original inputs are stored and remain unchanged, as they are needed later on further down the line.

Binary Calculator									
Symbol	Process	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
A	Input 1	0	0	1	1	0	0	1	1
B	Input 2	0	0	0	0	1	0	1	0
O	Input 3	1							
X	XOR(A, O)	1	1	0	0	1	1	0	0

Figure 17: *The input section of the 8-bit binary calculator, capable of both addition and subtraction.*

The next process involved creating a working carry, which allowed the correction of any overflow from adding two 1's together or subtracting a 1 from a 0. This was done by taking the resulting carry out from the previous bit and using it for the next carry in. Since the first carry has no value to take from, it defaults to 0, symbolled as a cross in the picture in order to clearly show it is not in use.

Symbol	Process	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Cin	Carry: Last Co Out.	0	0	0	1	0	0	0	✕

The next step involved finding the values of B and process X. If both of them were 1, the output would be 1, otherwise it would be 0. This was called process D.

Symbol	Process	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D	AND(X, B)	0	0	0	0	1	0	0	0

Afterwards, the bits from the two binary inputs are checked against each other using an XOR gate. If the bits are both 1 or 0, then a result of 0 is produced, but if they are different, a result of 1 will be outputted instead. This was called process E.

Symbol	Process	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
E	XOR(A, B)	0	0	1	1	1	0	0	1

The result of this process is then checked against the third input, the one determining the operation, to see if they are also the same using another XOR gate. If the resulting bits are both 1 or 0, the result will be 0, but if there are different then the output will be 1. This was called process Y.

Symbol	Process	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Y	XOR(E, O)	1	1	0	0	0	1	1	0

If both the carry in and process Y bits are equal to 1, then the result via the use of an AND gate will be a 1. This was called process F.

Symbol	Process	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
F	AND(Cin, Y)	0	0	0	0	0	0	0	0

The carry output was then worked out via the use of an OR gate, using process D and process F and inputs. If either of the processes produced a 1, the result would be a 1.

Symbol	Process	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Co	OR(D, F)	0	0	0	0	1	0	0	0

Finally, the bit being outputted was calculated with another XOR gate, this time using the carry in and process E as inputs. If either of the inputs were 1, the bit being outputted would be 1. This chain of processes was completed for each of the 8-bits, until finally an 8-bit result was produced.

Symbol	Process	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Output	**XOR(Cin, E)**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**

Once each of the 8-bits had been calculated, the cells were concatenated to create the final 8-bit binary number. Both the result and the individual 8-bits were then sent back to the input screen, directly under the two binary inputs and the operation used. The final version of this screen, the binary calculator and the comments for each process can be seen below.

Task 6: 8-Bit Binary Addition and Subtraction									
8 Bit Adder	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Result
Input 1	0	0	1	1	0	0	1	1	00110011
Input 2	0	0	0	0	1	0	1	0	00001010
Operation	-								Subtraction
Output	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**00101001**

Binary Calculator										Result	Check
Symbol	Process	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0		
A	Input 1	0	0	1	1	0	0	1	1		
B	Input 2	0	0	0	0	1	0	1	0		
O	Input 3	1									
X	XOR(A, O)	1	1	0	0	1	1	0	0		
Cin	Carry: Last Co Out.	0	0	0	1	0	0	0			
D	AND(X, B)	0	0	0	0	1	0	0	0	**00101001**	00101001
E	XOR(A, B)	0	0	1	1	1	0	0	1		
Y	XOR(E, O)	1	1	0	0	0	1	1	0		
F	AND(Cin, Y)	0	0	0	0	0	0	0	0		
Co	OR(D, F)	0	0	0	0	1	0	0	0		
Output	**XOR(Cin, E)**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**		

Binary Calculator		
Symbol	Process	Comments
A	Input 1	This is the first input used in the binary calculator, representing the first value.
B	Input 2	This is the second input used in the binary calculator, representing the second value.
O	Input 3	This is the third input used in the binary calculator, with a 0 for addition or a 1 for subtraction.
X	XOR(A, O)	If the operation is addition, the result will simply be the first input, otherwise it will be the inverse.
Cin	Carry: Last Co Out.	This is used for when values carry over, i.e. when both input bits are set to 1.
D	AND(X, B)	If the bit values of X and B are both 1, then the output is 1, otherwise it will be 0.
E	XOR(A, B)	If A and B have different states then the result will be 1, otherwise it will be 0.
Y	XOR(E, O)	If the operation is addition, the result will simply be the result above, otherwise it will be the inverse.
F	AND(Cin, Y)	If the bit values of Cin and Y are both 1, then the output is 1, otherwise it will be 0.
Co	OR(D, F)	If either the bit values of D or F are 1, then the output is 1, otherwise it will be 0.
Output	**XOR(Cin, E)**	If Cin and E have different states then the result will be 1, otherwise it will be 0. This is the final output.

Figure 18: *The entirety of the 8-bit binary calculator, including the explanations on each step above.*

If I were to attempt a task similar to this in this future, I would take the approach I used of drawing the logic circuit of the binary calculator and apply it to the earlier converters. Although I was able to produce solutions for them, I believe they could have been solved slightly easier if I had produced logic circuits for them beforehand. Overall, I am very happy with the knowledge I have gained from working on this particular task, especially the skills needed to create the final binary calculator.

Task 2: Developing Converters in C

Converting Inputs into Integers

The next program I created involved reading an input from the keyboard and converting it into an integer, which is then displayed on the terminal. This is simply done by reading the string into a null terminated char array, stopping only when it reaches the end of the input once a newline character ('\n') is read. After this occurs, the program then detects if the input is an integer using the built-in *atoi()* function. If the input is an integer, the program prints the converted number to the terminal, however if it is not then a '0' is printed instead.

Code for Converting Inputs into Integers

```
const byte numChars = 32;           // Creates a char array called 'numChars'.
char inputtedChars[numChars];       // An array to store the received chars from the user.
boolean allowConversion = false;    // Initialises a Boolean used later on in the program.

void setup()
{
    Serial.begin(9600);
    Serial.println("Please input an integer: ");
}

int readInt(void)
{
    if (allowConversion == true)        // Function continues if input is fully read.
    {
        int convertedToInt;
        char copyStr[32];                   // Creates a char array called 'copyStr'.
        strcpy(copyStr, inputtedChars);     // Copies 'inputtedChars' into 'copyStr'.
        convertedToInt = atoi(copyStr);     // Converts the String 'copyStr' into an int.
        Serial.print("String Value = ");
        Serial.print(inputtedChars);
        Serial.print(", Integer Value = ");
        Serial.println(convertedToInt);
        Serial.println("Please input an integer: ");

        allowConversion = false;        // Sets 'allowConversion' back to false so the
function will not run again until the next number is inputted.
        return(0);
    }
}

void detectInput()
{
    static byte spaceLimit = 0;
    char stoppingPoint = '\n';
    char inputChar;
    while (Serial.available() > 0 && allowConversion == false) {

        inputChar = Serial.read();

        if (inputChar != stoppingPoint)     // Loops until the stopping point is reached.
        {
            inputtedChars[spaceLimit] = inputChar;
            spaceLimit++;
            if (spaceLimit >= numChars) {spaceLimit = numChars - 1;}
        }
```

```
        else
        {
            inputtedChars[spaceLimit] = '\0'; // Terminates string, reaching the input end.
            spaceLimit = 0;
            allowConversion = true;            // Allows conversion of the user's input.
        }
    }
}

void loop()
{
    detectInput();
    readInt();
}
```

Testing Converting Inputs into Integers
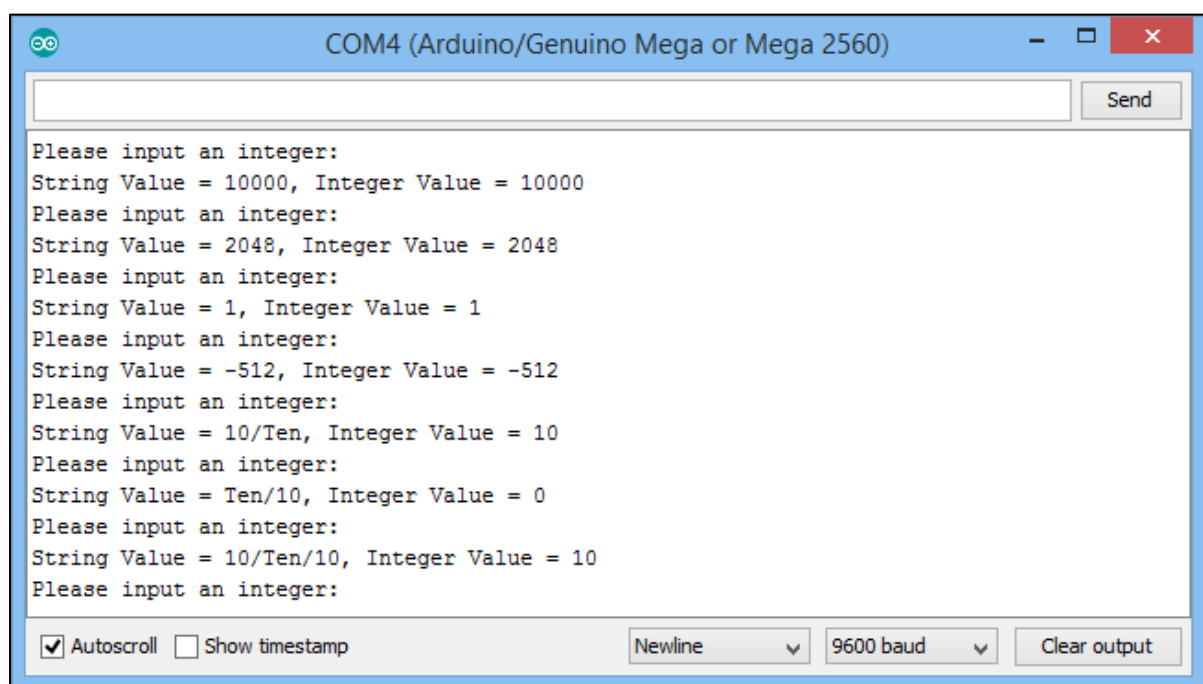
Figure 19: *A test of various numbers inputted into the program, and the resulting outputs from them.*

As you can see from the test, numbers are converted into integers, including negative numbers. If, however, anything other than a number or negative sign is inputted, then a '0' is printed to the terminal as previously mentioned. In the event of a number being inputted before a string of text, the number will be converted, with anything after the number being discarded. This can be seen in the last line of testing, where the input '10/Ten/10' causes only the first '10' to be converted.

Converting Decimal into Binary

The second program involved taking the *readInt()* function I created in the previous task and using it to convert decimal numbers into binary. Like the previous task, the program first checks if the input is valid, in this case checking if the input is both an integer and in the range of -128 to 127. If this is true, the program converts it into its binary equivalent using a function called *decimalToBinary()*. However, if this is not the case, then the program will instead output that the value is out of range.

Code for Converting Decimal into Binary

```
const byte numChars = 32;            // Creates a char array called 'numChars'.
char inputtedChars[numChars];        // An array to store the received chars from the user.

void setup()
{
    Serial.begin(9600);
    Serial.print("Input a number in the range -128 to 127: ");
}

String decimalToBinary(String message, byte n)
{
    int c = 0;
    int intInput = n;
    int intCorrected = intInput;
    char binaryNumber[8] = {0};
    if (intInput >= 128) {intCorrected = (256 - intInput) / -1;}
    Serial.print(message);

    if (intInput != intCorrected)
    {
        Serial.print("1");    // Prints a '1' to the terminal as part of the binary output.
    }
    else
    {
        Serial.print("0");    // Prints a '0' to the terminal as part of the binary output.
    }

    while (c < 7)
    {
        Serial.print(intInput >> (6-c)&1);  // Gets bit c of int check.
        n /= 2;
        c += 1;
    }

    c = 0;
    Serial.print(" ");
    Serial.print(intCorrected);
    Serial.println(binaryNumber);
    Serial.print("Input a number in the range -128 to 127: ");
}

int readInt(void)
{
    int convertedToInt;
    char copyStr[32];                    // Creates a char array 'copyStr'.
    strcpy(copyStr, inputtedChars);      // Copies 'inputtedChars' into 'copyStr'.
    convertedToInt = atoi(copyStr);      // Converts the String 'copyStr' into an int.

    if (convertedToInt < -128 || convertedToInt > 127)
    {
        Serial.println("Value out of range.");
        Serial.print("Input a number in the range -128 to 127: ");
    }
    else
    {
        decimalToBinary("bin = ", convertedToInt);
    }
}
```

```
void detectInput()
{
    static byte spaceLimit = 0;
    char stoppingPoint = '\n';
    char inputChar;

    while (Serial.available() > 0)
    {
        inputChar = Serial.read();

        if (inputChar != stoppingPoint)     // Loops until the stopping point is reached.
        {
            inputtedChars[spaceLimit] = inputChar;
            spaceLimit++;
            if (spaceLimit >= numChars) {spaceLimit = numChars - 1;}
        }
        else
        {
            inputtedChars[spaceLimit] = '\0';      // Terminates string, reaching the end.
            spaceLimit = 0;
            readInt();
        }
    }
}

void loop()
{
    detectInput();
}
```

Testing Converting Decimal into Binary
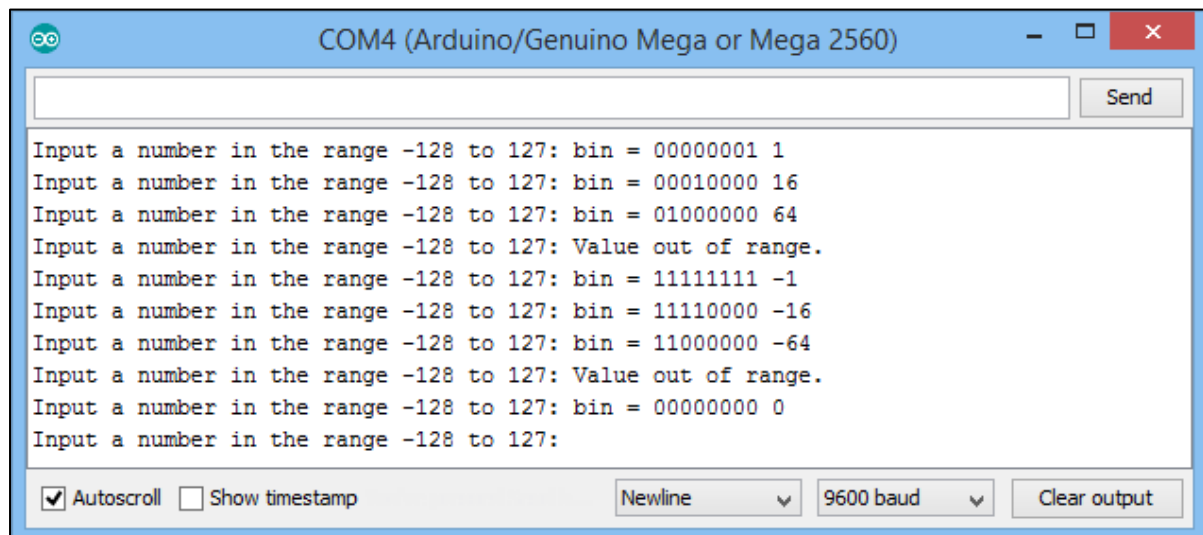
Figure 20: *A test of various numbers inputted into the program, and the resulting outputs from them.*

Note that the last test produced the output 'bin = 00000000 0" instead of 'Value out of range'. This is because the same logic is used from the previous question, where in the event of a number being inputted before a string of text, only the number will be converted. Since no number is inputted before the input 'one', the input is treated as being empty as opposed to being out of range.

Dumped Registers

Similar to the previous programs, this one starts by using the *readInt()* function to check for two integer inputs. After each input is verified, the *decimalToBinary()* function was used to convert them into their respective 16-bit binary outputs. The program followed the previous logic when converting the numbers, except for the end where each 16-bit binary number was split into 2 separate 8-bit binary numbers. These values are then printed to the terminal, alongside a third integer input in the range of 1 and 8.

Code for Dumped Registers

```
boolean startMessage = true;

int loopNumber = 1;

int binaryInt1;
int binaryInt2;
int binaryInt3;

String binaryBR;
String binaryAD;

String B;
String R;
String A;
String D;

const byte numChars = 32;          // Creates a char array called 'numChars'.
char inputtedChars[numChars];      // An array to store the received chars from the user.

void setup()
{
    Serial.begin(9600);
}

void trace(word BR, word AD, byte n)
{
    binaryBR = decimalToBinary(BR); // Calls the decimalToBinary(int) function to produce
a binary value in the form of a string labelled 'binaryBR' from an interger input.

    binaryAD = decimalToBinary(AD); // Calls the decimalToBinary(int) function to produce
a binary value in the form of a string labelled 'binaryAD' from an interger input.

    for (int x = 0; x < 8; x++)
    {
        B.concat(binaryBR.charAt(x));      // Amends the first 8 chars (the high byte)
from the string 'binaryBR' to the variable B.

        R.concat(binaryBR.charAt(x+8));    // Amends the last 8 chars (the low byte) from
the  string 'binaryBR' to the variable R.

        A.concat(binaryAD.charAt(x));      // Amends the first 8 chars (the high byte)
from the string 'binaryAD' to the variable A.

        D.concat(binaryAD.charAt(x+8));    // Amends the last 8 chars (the low byte) from
the string 'binaryAD' to the variable D.
    }
```

```arduino
    Serial.print("\nn=");
    Serial.print(n);
    Serial.print(" D=");
    Serial.print(D);
    Serial.print(" B=");
    Serial.print(B);
    Serial.print(" A=");
    Serial.print(A);
    Serial.print(" R=");
    Serial.print(R);
    Serial.print("\n\n");
}

String decimalToBinary(byte n)
{
    int c = 0;
    int bitNum = 16;
    int intInput = n;
    String binaryString = "";

    while (c < (bitNum - 1))         // Loops until all of the bits have been checked.
    {
        int digit = (intInput >> ((bitNum - 2) - c) & 1); // Checks if the digit for the
selected bit should be 0 or 1.
        String digitString = String(digit);        // Converts the digit to a string.
        binaryString += digitString;                    // Adds the previously converted
digit onto the final binary output string.
        c += 1;
    }

    if (intInput >= 0) {binaryString = "0" + binaryString;}      // Places a '0' on the
start of the binary output if the number is positive.
    else {binaryString = "1" + binaryString;}                 // Places a '1' on the
start of the binary output if the number is negative.

    return(binaryString);    // Returns the final binary number in the form of a string.
}

int readInt(void)
{
    int convertedToInt;
    char copyStr[32];        // Creates a char array called 'copyStr' with length 32.
    strcpy(copyStr, inputtedChars);        // Copies the String 'inputtedChars' into
another String called 'copyStr'.
    convertedToInt = atoi(copyStr);        // Converts the String 'copyStr' into an int.

    if (loopNumber == 1 || loopNumber == 2)
    {
        if (convertedToInt < -32768 || convertedToInt > 32767)
        {
            Serial.println("Value out of range.");
            loopNumber -= 1;
        }
        else
        {
            return(convertedToInt);
        }
    }
```

```
        else
        {
            if (convertedToInt < 1 || convertedToInt > 8)
            {
                Serial.println("Value out of range.");
                return(-1);
            }
            else
            {
                Serial.println(convertedToInt);
                return(convertedToInt);
            }
        }
    }
}

void detectInput()
{
    static byte spaceLimit = 0;
    char stoppingPoint = '\n';
    char inputChar;

    while (startMessage == true)
    {
        if (loopNumber == 1) {Serial.print("Input BR in range -32768 to 32767: ");}
        else if (loopNumber == 2) {Serial.print("Input AD in range -32768 to 32767: ");}
        else {Serial.print("Input n in range 1 to 8: ");}
        startMessage = false;
    }

    while (Serial.available() > 0)
    {
        inputChar = Serial.read();

        if (inputChar != stoppingPoint)     // Loops until the stopping point is reached.
        {
            inputtedChars[spaceLimit] = inputChar;
            spaceLimit++;
            if (spaceLimit >= numChars) {spaceLimit = numChars - 1;}
        }
        else
        {
            inputtedChars[spaceLimit] = '\0';       // Terminates the string.
            spaceLimit = 0;

            if (loopNumber == 1)
            {
                binaryInt1 = readInt();           // Calls the readInt() function,
converting the users string input into an interger and storing it as 'binaryInt1'.
                Serial.println(binaryInt1);
                B = "";
                R = "";
            }
            else if (loopNumber == 2)
            {
                binaryInt2 = readInt();             // Calls the readInt() function,
converting the users string input into an interger and storing it as 'binaryInt2'.
                Serial.println(binaryInt2);
                A = "";
                D = "";
            }
```

```
        else
        {
            binaryInt3 = readInt();              // Calls the readInt() function,
converting the users string input into an interger and storing it as 'binaryInt3'.

            if (binaryInt3 != -1)
            {
                loopNumber = 0;
                trace(binaryInt1, binaryInt2, binaryInt3);    // Calls the trace(int,
int, int) function, taking the inputs from the user to produce an output.
            }
            else {loopNumber =-1;}
        }

        loopNumber += 1;
        startMessage = true;
    }
  }
}

void loop()
{
    detectInput();    // Constantly calls detectInput() to check if the user has inputted.
}
```

Testing Dumped Registers
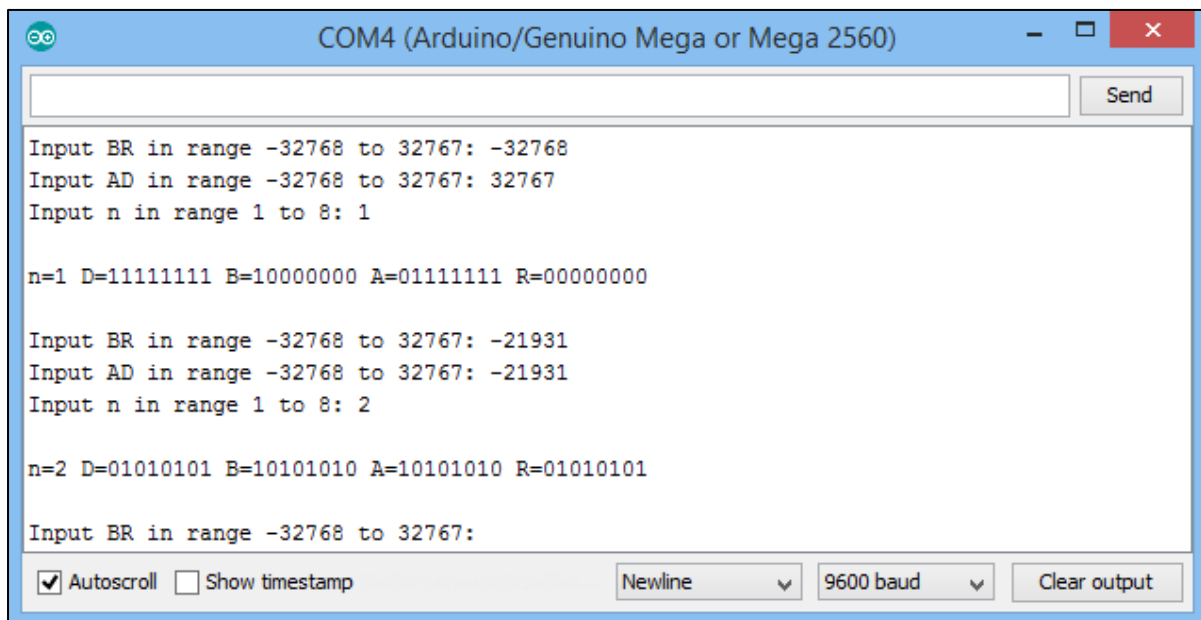
Figure 21: A test of various numbers inputted into the program, and the resulting outputs from them.

If I were to attempt a task similar to this one in this future, I would have created flowcharts from the beginning to make the programs easier to visualise, as discussed in the previous task. I would also spend more time researching online the different assembly registers, as I suspect this would have benefited all of the programs produced, particularly the last one. Overall, I am happy with the extra knowledge that I have gained from developing the conversions in C.

Task 3: Simulating Traffic Light Intersections

Singular Traffic Light Flowchart
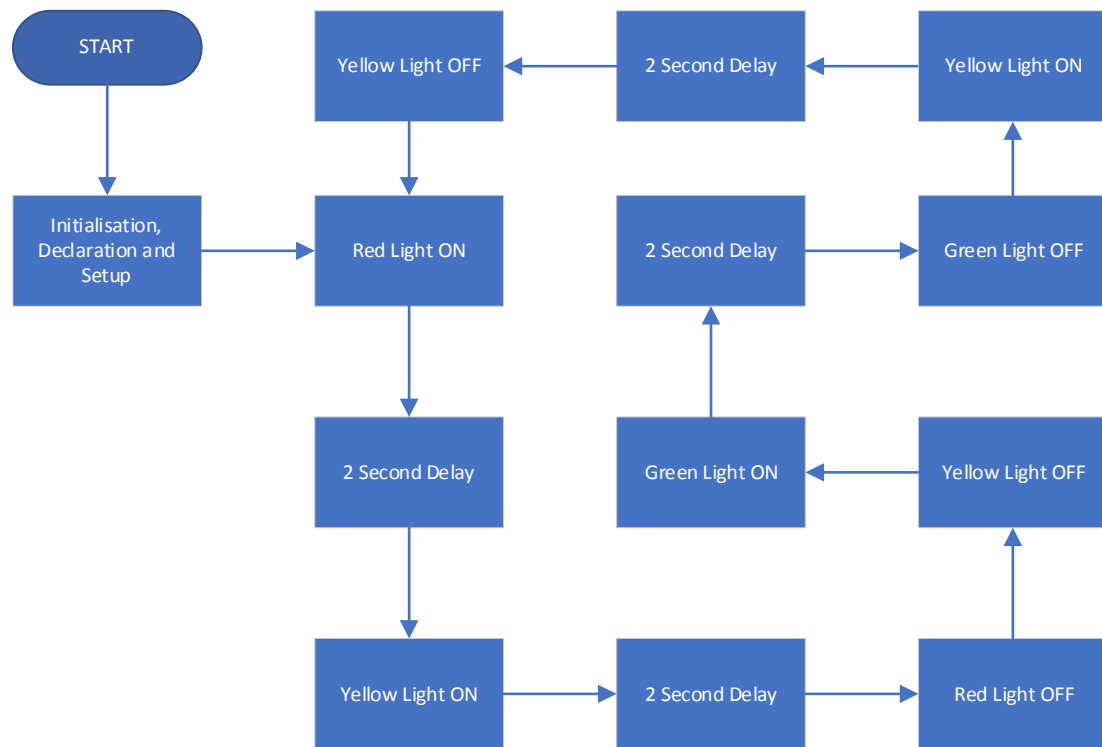
Figure 22: *A flowchart of a singular traffic light, with its four distinct phases and light changes.*

Singular Traffic Light Code

```
#define RED 1
#define YELLOW 2
#define GREEN 3

void setup()
{
    pinMode(RED, OUTPUT);
    pinMode(YELLOW, OUTPUT);
    pinMode(GREEN, OUTPUT);
}

void loop()
{
    digitalWrite(RED, HIGH);
    delay(2000);
    digitalWrite(YELLOW, HIGH);
    delay(2000);
    digitalWrite(RED, LOW);
    digitalWrite(YELLOW, LOW);
    digitalWrite(GREEN, HIGH);
    delay(2000);
    digitalWrite(GREEN, LOW);
    digitalWrite(YELLOW, HIGH);
    delay(2000);
    digitalWrite(YELLOW, LOW);
}
```
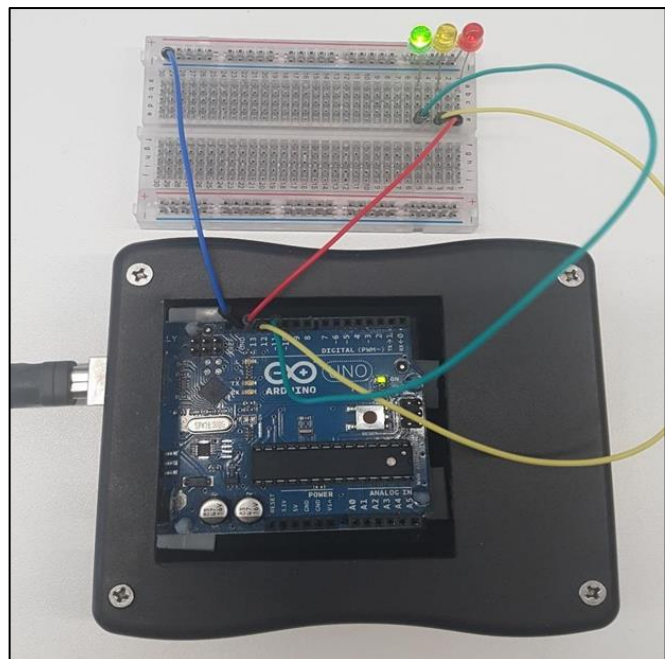

Figure 23: *The Arduino's code and following output.*

Pedestrian Crossing Flowchart
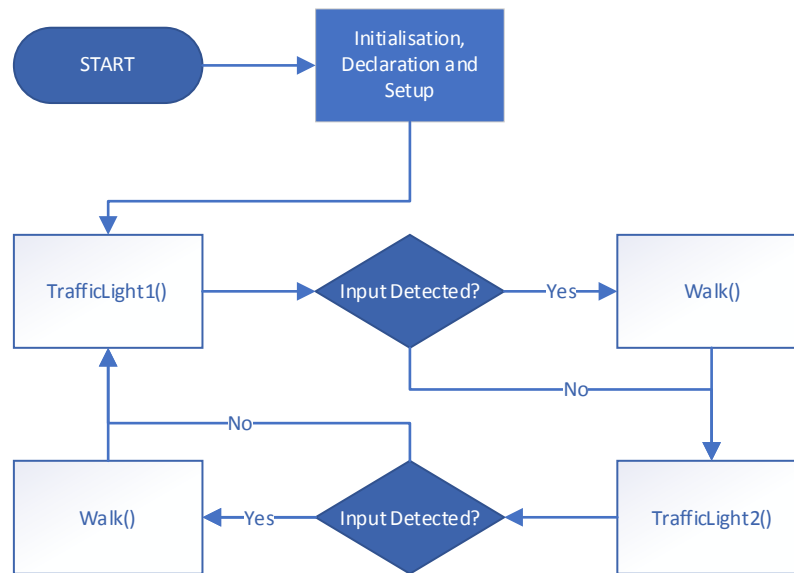
Figure 24: *A flowchart of a pedestrian crossing, with two traffic lights and a crosswalk.*
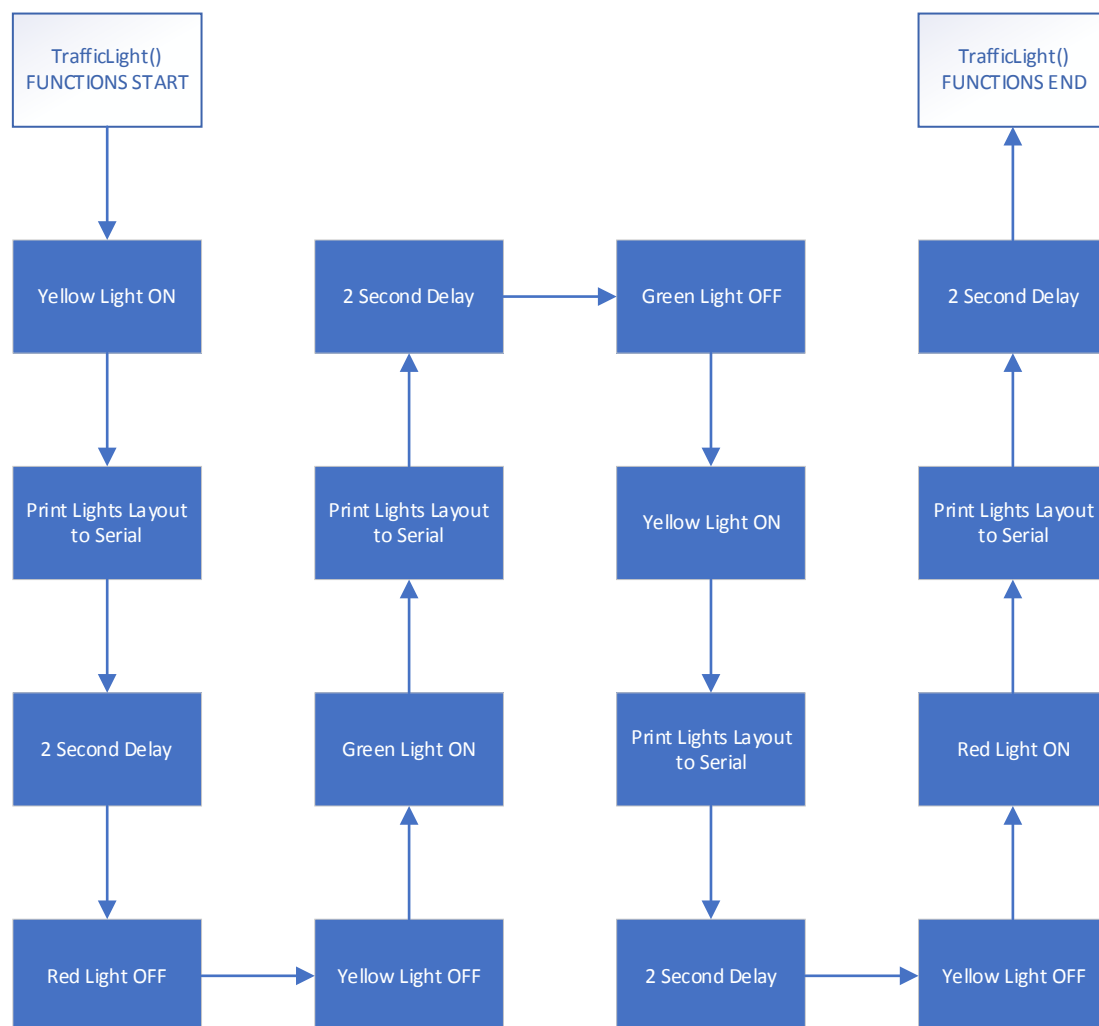
Figure 25: *A flowchart representing the two TrafficLight() functions crossing, similar to Figure 20.*
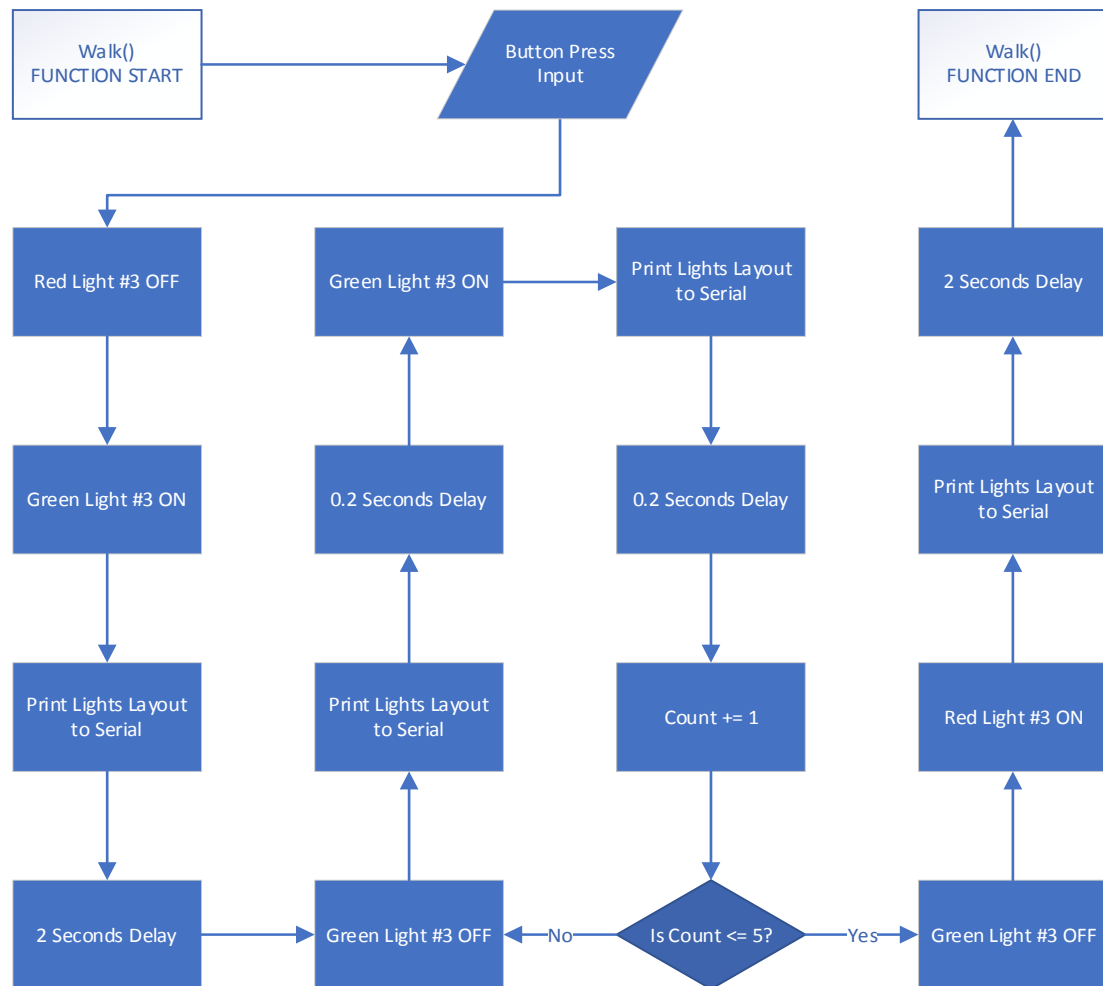
Figure 26: *A flowchart representing the walk() function, triggered by an input in the serial display.*
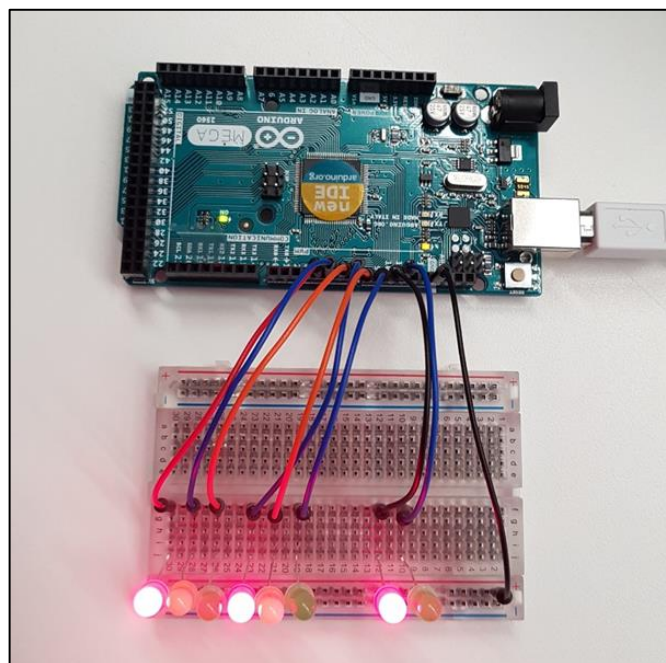
Pedestrian Crossing Code

```
#define RED1 4
#define YELLOW1 5
#define GREEN1 6
#define RED2 7
#define YELLOW2 8
#define GREEN2 9
#define RED3 10
#define GREEN3 11

int flashing = 0;
int incomingByte = 0;

void setup()
{
    Serial.begin(9600);

    pinMode(RED1, OUTPUT);
    pinMode(YELLOW1, OUTPUT);
    pinMode(GREEN1, OUTPUT);
    pinMode(RED2, OUTPUT);
    pinMode(YELLOW2, OUTPUT);
    pinMode(GREEN2, OUTPUT);
```


Figure 27: *The Arduino's code and following output.*

```
    pinMode(RED3, OUTPUT);
    pinMode(GREEN3, OUTPUT);
    digitalWrite(RED1, HIGH);
    digitalWrite(RED2, HIGH);
    digitalWrite(RED3, HIGH);

    Serial.println("Light 1: On, Off, Off");
    Serial.println("Light 2: On, Off, Off");
    Serial.println("Crossing: On, Off\n");
    delay(2000);
}

void walk()
{
    incomingByte = Serial.read();

    digitalWrite(RED3, LOW);
    digitalWrite(GREEN3, HIGH);
    Serial.println("Light 1: Off, Off, Off");
    Serial.println("Light 2: Off, Off, Off");
    Serial.println("Crossing: Off, On\n");
    delay(2000);

    for(flashing = 0; flashing <= 5; flashing = flashing + 1)
    {
        digitalWrite(GREEN3, LOW);
        Serial.println("Light 1: Off, Off, Off");
        Serial.println("Light 2: Off, Off, Off");
        Serial.println("Crossing: Off, Off\n");
        delay(200);

        digitalWrite(GREEN3, HIGH);
        Serial.println("Light 1: Off, Off, Off");
        Serial.println("Light 2: Off, Off, Off");
        Serial.println("Crossing: Off, On\n");
        delay(200);
    }

    digitalWrite(GREEN3, LOW);
    digitalWrite(RED3, HIGH);
    Serial.println("Light 1: Off, Off, Off");
    Serial.println("Light 2: Off, Off, Off");
    Serial.println("Crossing: On, Off\n");
    delay(2000);
}

void TrafficLight1()
{
    digitalWrite(YELLOW1, HIGH);
    Serial.println("Light 1: On, Off, Off");
    Serial.println("Light 2: On, On, Off");
    Serial.println("Crossing: On, Off\n");
    delay(2000);

    digitalWrite(RED1, LOW);
    digitalWrite(YELLOW1, LOW);
    digitalWrite(GREEN1, HIGH);
    Serial.println("Light 1: On, Off, Off");
    Serial.println("Light 2: Off, Off, On");
    Serial.println("Crossing: On, Off\n");
    delay(2000);
```

```
        digitalWrite(GREEN1, LOW);
        digitalWrite(YELLOW1, HIGH);
        Serial.println("Light 1: On, Off, Off");
        Serial.println("Light 2: Off, On, Off");
        Serial.println("Crossing: On, Off\n");
        delay(2000);

        digitalWrite(YELLOW1, LOW);
        digitalWrite(RED1, HIGH);
        Serial.println("Light 1: On, Off, Off");
        Serial.println("Light 2: On, Off, Off");
        Serial.println("Crossing: On, Off\n");
        delay(2000);
}

void TrafficLight2()
{
        digitalWrite(YELLOW2, HIGH);
        Serial.println("Light 1: On, On, Off");
        Serial.println("Light 2: On, Off, Off");
        Serial.println("Crossing: On, Off\n");
        delay(2000);

        digitalWrite(RED2, LOW);
        digitalWrite(YELLOW2, LOW);
        digitalWrite(GREEN2, HIGH);
        Serial.println("Light 1: Off, Off, On");
        Serial.println("Light 2: On, Off, Off");
        Serial.println("Crossing: On, Off\n");
        delay(2000);

        digitalWrite(GREEN2, LOW);
        digitalWrite(YELLOW2, HIGH);
        Serial.println("Light 1: Off, On, Off");
        Serial.println("Light 2: On, Off, Off");
        Serial.println("Crossing: On, Off\n");
        delay(2000);

        digitalWrite(YELLOW2, LOW);
        digitalWrite(RED2, HIGH);
        Serial.println("Light 1: On, Off, Off");
        Serial.println("Light 2: On, Off, Off");
        Serial.println("Crossing: On, Off\n");
        delay(2000);
}

void loop()
{
        TrafficLight1();
        if (Serial.available() > 0) { walk(); }
        TrafficLight2();
        if (Serial.available() > 0) { walk(); }
}
```

One thing I changed when designing this during the assignment was how I wired up my LED's.
Originally, I had each LED with a separate ground wire connecting to the one grounding wire
connecting to the Arduino. Through building my circuit however I realised that I could put all of my
LED's onto one grounded connection, which meant I only needed one wire for grounding my circuit.
making my wiring more compact. Once this was done, I moved onto recreating it in assembly.

Traffic Lights in Assembly Language

```c
word millisecs;        // C variable read by assembly code contains blink delay

void setup()
{
    // -------------------------------------------------------------------------
    // set portB (digital pins 8-13) as outputs
    // achieved by writing to port B Data Direction Register (DDRB) at I/O address 4
    // if a bit is set, the corresponding pin is an output, otherwise it is an input
    // bit designations for Data Direction Register B (DDRB) and Port B (PORTB)
    // bit5 = Pin13; bit4 = Pin12; bit3 = Pin11; bit2 = Pin10; bit1 = Pin9; bit0 = Pin8
    // -------------------------------------------------------------------------

    asm volatile(
        "          ldi r16,0x3F      ; r16 = 00111111\n"
        "          out 4,r16         ; set pins 8-13 as outputs in DDRB\n"
        ::: "r16");

    millisecs = 1000;             // 1s blink delay
    Serial.begin(9600);
}

void loop()
{
    long starttime = millis();      // make a note of the start time

    asm volatile(

        // jump to "blink" - i.e. jump around the delay_ms subroutine
    "   rjmp  blink%=               ; relative jump to 'blink' \n"

        // -------------------------------------------------------------------------
        // input variable - millisecond count in register pair r30:r31
        //
        // r31 - millisecond count (low byte)
        // r30 - millisecond count (high byte)
        // r17 - 100 microsecond count
        // r16 - 1 microsecond count
        //
        // overall delay (ms) = r30:r31 * r17 * r16
        // -------------------------------------------------------------------------

        "delay_ms%=:    ldi r17, 10   ; setting the address r17 to 10, taking 62.5ns \n"
        "delay_100us%=: ldi r16, 100  ; setting the adresss r16 to 100, taking 62.5ns \n"
        "delay_1us%=:                 ; begins with a loop made up of 13 nop commands \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               nop           ; performs no operation, taking 62.5ns \n"
        "               dec r16       ; decrease the value of r16 by 1, taking 62.5ns \n"
```

```
            "                brne delay_1us%=      ; loop back to delay_1us if > 0, taking
62.5ns if false or 125.0ns if true \n"
            "                dec r17               ; decrease the value of r17 by 1, taking
62.5ns \n"
            "                brne delay_100us%=    ; loop back to delay_100us if > 0, taking
62.5ns if false or 125.0ns if true \n"
            "                sbiw r30, 1           ; decrease the value of r30:r31, taking
62.5ns \n"
            "                brne delay_ms%=       ; loop back to delay_ms if > 0, taking
62.5ns if false or 125.0ns if true \n"

            "                ret                   ; return from the subroutine \n"

        " lights%=:                               ; start of lights code \n"

        // holds each LED state for 1 second

            "                out 5, r18            ; output the loaded LED state to port B \n"
            "                lds r30, millisecs    ; r30 = hi byte \n"
            "                lds r31, millisecs + 1 ; r31 = lo byte \n"
            "                call delay_ms%=       ; call millisec delay subroutine \n"
            "                ret                   ; return from subroutine \n"
        " blink%=:                                ; start of blink code \n"

        // phase one of traffic light system

            "                ldi r18, 0x21         ; loads position of the red LED \n"
            "                call lights%=         ; call lights subroutine \n"

        // phase two of traffic light system

            "                ldi r18, 0x32         ; loads position of red and yellow LEDs \n"
            "                call lights%=         ; call lights subroutine \n"

        // phase three of traffic light system

            "                ldi r18, 0x0c         ; loads position of the green LED \n"
            "                call lights%=         ; call lights subroutine \n"

        // phase four of traffic light system

            "                ldi r18, 0x16         ; loads position of the yellow LED \n"
            "                call lights%=         ; call lights subroutine \n"

        ::: "r16", "r17", "r18", "r30", "r31"); // clobbered registers

    // calculate the execution time of the traffic light routine, and print details

    long endtime = millis();                    // make a note of the end time
    float ms = endtime - starttime;             // calculate the interval
    float expected = 4 * millisecs;             // millisecs * 4 (4 delays in blink)
    float overheads = 34;                       // overheads due to the timing
    expected = expected + overheads;
    float error_percent = 100.0 * (ms-expected) / expected;

    Serial.print("delay="); Serial.print(ms);
    Serial.print("ms  "); Serial.print("error: ");

    if(error_percent>0) { Serial.print("+"); }
    Serial.print(error_percent); Serial.println("%");
}
```

*Figure 28: A showcase of the assembly language code seen above, with the accuracy of it displayed.*

When the program is run, it first sets all the pins on port B as outputs, so that they can be used to light up the LED's. A variable called milliseconds is then set to 1000, which is used later to determine the error percentage of each delay in both programs. Next the serial data transmission rate is then set to 9600 bps, where each bit is sent sequentially. Lastly, a long variable called *starttime* is set to *millis()*, which records the start time before any delay occurs; this is stored using the long data type.

After all the setup has occurred, the program runs the loop function, which starts by jumping to the line containing the word "blink". In both programs pin 13 is set to high, which lights up a red LED on the connected breadboard. The next line causes both programs to jump to the light's subroutine. This function was created to compact the code, as each state of LEDs goes through the same process of setting the output to port B and calling the millisecond delay subroutine. This meant that it made sense to take out the repeated lines of code for each blink or traffic light state, and instead call that section of code whenever it is needed. This helped improve the efficiency of my code.

Once the LED has been on for 1 second via the delay subroutine, the next state is processed, which in this case involves turning on the yellow LED. The delay subroutine is then called again, holding the LED state for another second. The third state causes both the red and yellow LED's to turn off, and the green LED being turned on. One second later, the green LED is turned off and the yellow LED is turned on. After one more second, the program moves to the next section of code, where it calculates if the timings were correct.

After the four traffic light stages have occurred, a variable called *endtime* is set to *millis()*, recording the time after all the delays have occurred. Like the *starttime* variable at the start of the program, this is also stored using the long data type. Next, the variable *ms* is then created, which subtracts *starttime* from *endtime*, to find out the number of milliseconds the delay was. This number is stored as a float, as a margin of error is monitored further down the program. The expected delay is 2000 milliseconds, with an overhead of 17 milliseconds due to the timings of the code being run. The program then calculates the error margin as a percentage, by comparing to the expected number of milliseconds. Both the actual number of milliseconds and the error margin are then outputted to the serial via the print command.

Task 4: Serial Communication Between Arduinos
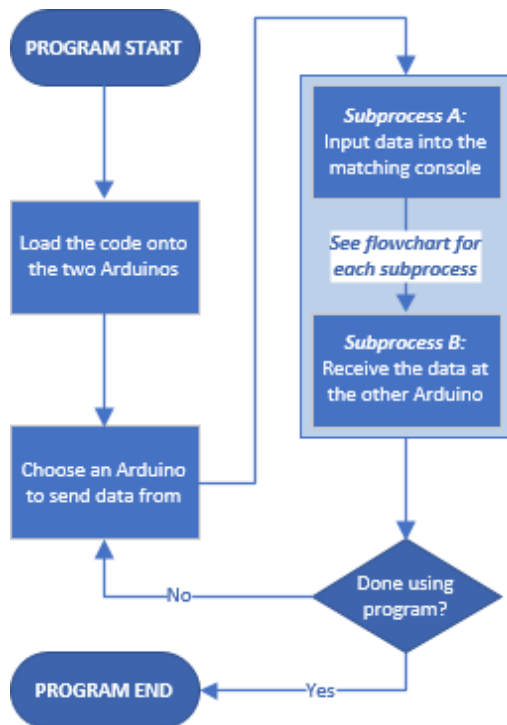
Figure 29: *The basic setup of the Arduinos.*

Serial transmission is asynchronous when using the Arduino IDE; this means that data is transmitted intermittently rather than in a steady stream. In this case, two Arduino's have the same code uploaded to them, allowing two users to send and receive data to and from each other.

How this is done is laid out in detail in the Subprocess A flowchart on the following page. In simple terms, one Arduino acts as a sender, transmitting a byte worth of binary data at a time, whilst the opposite Arduino acts as a receiver, checking to make sure that the data received is valid data.

In order to check that the data received has not been corrupted, the information is sent using a Start / Stop Protocol, which encompasses the data inside 1 start bit and 2 stop bits, alongside a calculated parity bit. Since 4 bits are being used to detect that the data received is corrected, this leaves 4 bits for the data itself, resulting in a 50% efficiently as seen below.

Asynchronous Transmission Efficiency $= \dfrac{data\ transmitted}{total\ bits\ sent} \times 100 = \dfrac{4\ bits}{8\ bits} \times 100 = 50\%$

Since only 50% of the data received by the second Arduino is actually data inputted by the user, this type of transmission is inefficient compared to synchronous transmission, where efficiency of up to 99.9% can be achieved by sending much more data in-between start and stop bits. Asynchronous transmissions are however much simpler to produce and inexpensive to implement. It is mainly used with Serial Ports, such as communicating between two Arduinos as seen in this particular example.

The calculated parity bit is used to help work out if the data sent has become corrupted. This is done by working out how many 1's there are being sent in the byte. If there is an odd number of 1's, the parity will be even, to ensure an even number of 1's that are sent across. If there is an even number, the parity bit will instead be odd. This is called even parity, although the inverse could be used (odd parity), so long as both the receiver and the sender are using the same. The two must also agree on the number of bits per package of data, the bits per second (equal to the Baud rate, set to 9600 in this example) and what to do when an incorrect parity bit is detected. In this example, the sender Arduino resends the previously sent byte of data. Once all the data has been sent to the receiver Arduino and concatenated together, it then converts the binary number into the message that the user inputted from the first Arduino. A breakdown of this can be seen in the following flowcharts.

Transmission Output
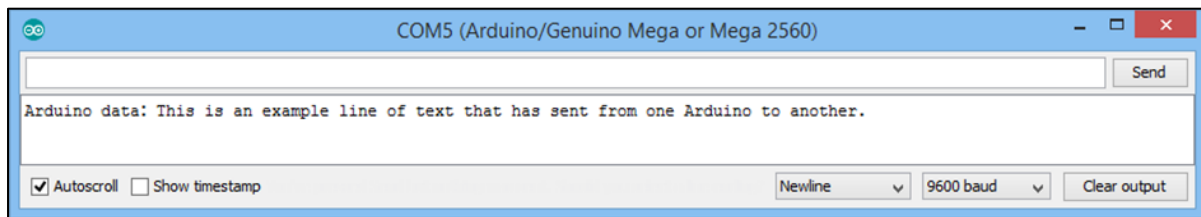
Figure 30: *An example output transmitted through the serial communication between the Arduinos.*

Subprocess A - Inputting Data
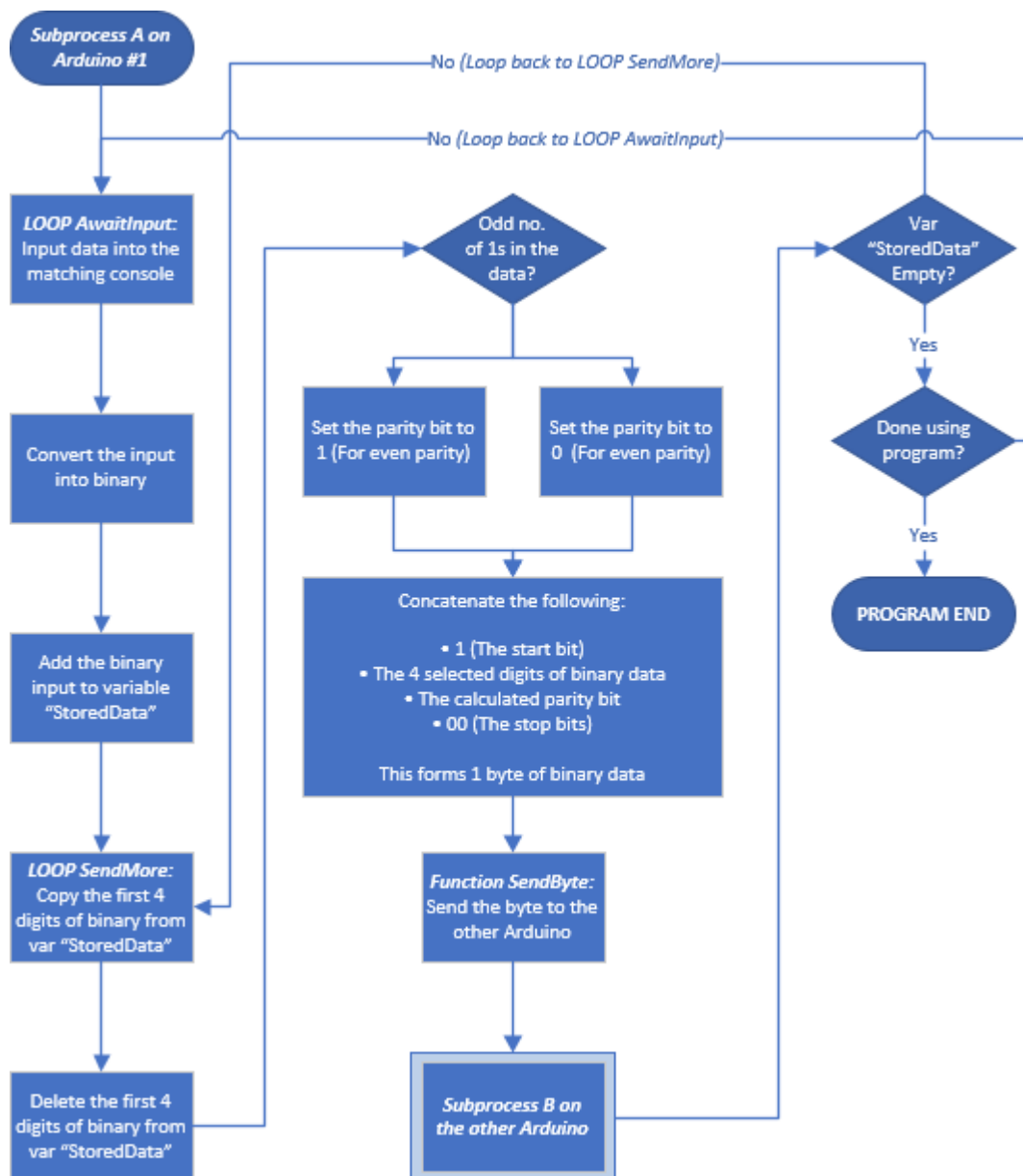
Figure 31: *A breakdown of Subprocess A, which takes the inputted data and converts it into binary.*
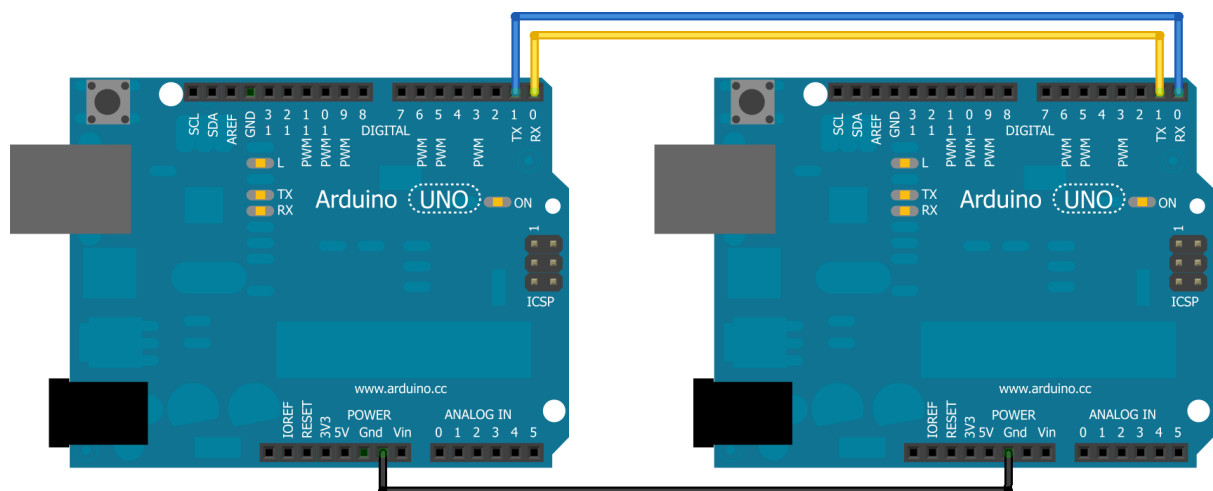
Subprocess B - Receiving Code

Figure 32: *A breakdown of Subprocess B, which takes the received binary data and converts it back.*

Hardware Set Up

Figure 33: *Connecting two Arduino Uno's in order to send data through their serial communication.*

Individual bytes are sent and received via the TX (1) and RX (0) pins between two separate Arduinos, via the use of the UART chip on each Arduino board. Each Arduino has its TX pin connected to the other Arduinos RX pin and vice versa and are both grounded together. Both Arduinos also share the same uploaded code, so that either of them is able to send and receive user inputted data.

Full System Flowchart

Figure 34: *An entire breakdown of the previous processes required to transmit between the Arduinos.*

The start-stop protocol code for this program is written in the following page; it is run of both of the connected Arduinos, as each one can be a sender and a receiver. This allows for the use of two-way communication instead of a simple one-way system, allowing for more flexible data transmission.

Start-Stop Protocol Code

```
// Sending data

int parity;
int dataNumber = 0;
boolean inputReady;
boolean inputInProgress;
byte sendDataByte1stHalf;
byte sendDataByte2ndHalf;

// Receiving data

int dataReceived;
char receivedChar;
boolean newData = false;

void setup()
{
    Serial.begin(9600);        // Open the serial port, setting the data rate to 9600 bps
    Serial.println("Arduino is ready");
}

void loop()
{
    readInput();      // Read the users inputted data
    getChar();        // Gets the byte sent from the other Arduino
    outputData();     // Outputs the users inputted data
    delay(1000);      // Creates a short delay of 1 second
}

void readInput()
{
    inputReady = true;
    inputInProgress = false;
    if(inputReady = true)
    {
        String inputtedData = String(Serial.read());
        for(int i = 0; i < inputtedData.length(); i++)
        {
            char singleChar = inputtedData.charAt(i);
            for(int i = 7; i >= 4; i--)
            {
                sendDataByte1stHalf = bitSet(singleChar, i); // Setting half of the bits
            }

            parity = 0;
            for (int i = 0; i <= 7; i++)
            {
                parity += bitRead(sendDataByte1stHalf, i);
            }

            if (parity % 2 == 0)                // Setting the parity bit for error detection
            {
                parity = 0;
            }
            else
            {
                parity = 1;
            }
```

```
            sendDataByte1stHalf = bitSet(1, 7);            // Setting the START bit
            sendDataByte1stHalf = bitSet(0, 1);            // Setting the first STOP bit
            sendDataByte1stHalf = bitSet(0, 0);            // Setting the second STOP bit
            sendDataByte1stHalf = bitSet(parity, 2);       // Setting the second STOP bit

            for(int i = 3; i >= 0; i--)
            {
                sendDataByte2ndHalf = bitSet(singleChar, i); // Set 2nd half to sep. byte
            }

            parity = 0;

            for (int i = 0; i <= 7; i++)
            {
                parity += bitRead(sendDataByte2ndHalf, i);
            }

            if (parity % 2 == 0)              // Setting the parity bit for error detection
            {
                parity = 0;
            }
            else
            {
                parity = 1;
            }

            sendDataByte2ndHalf = bitSet(1, 7);            // Setting the START bit
            sendDataByte2ndHalf = bitSet(0, 1);            // Setting the first STOP bit
            sendDataByte2ndHalf = bitSet(0, 0);            // Setting the second STOP bit
            sendDataByte2ndHalf = bitSet(parity, 2);       // Setting the second STOP bit
        }

        inputReady = false;
    }

    Serial.print(sendDataByte1stHalf);
    Serial.print(sendDataByte2ndHalf);
}

void getChar()
{
    if (Serial.available() > 0)
    {
        receivedChar = Serial.read();
        newData = true;
    }
}

void outputData()
{
    if (newData == true)
    {
        int data = dataReceived;
        dataReceived = data << 8 | newData;
        Serial.print("Arduino Data: ");              // Alert user of incoming data
        Serial.println(receivedChar);                // Print data on serial monitor
        newData = false;
    }
}
```

Task 5: Wireless Arduino Communication

A WiFly chip can be connected to an Arduino and programmed via predefined protocols to allow for data to be transmitted over a wireless network. The industry standard is the IEEE 802.11 protocol, designed by the IEEE Computer Society, a professional institute of computer scientists and electrical engineers. The 802.11 protocol is used by many wireless networks around the world, due to its easy use and varying control based on its specific application. The WiFly chip used here to transmit data between the Arduinos utilises another commonly used 802.15.4 protocol. This protocol is specifically designed for low power consumption of only 38mA in use, and only 4uA in sleep mode. Additionally, the WiFly chip also has a built-in power management with programmable wakeups, which when combined with its low power allows for devices to be extremely durable and long lasting.

802.11n and 802.15.4 Differences
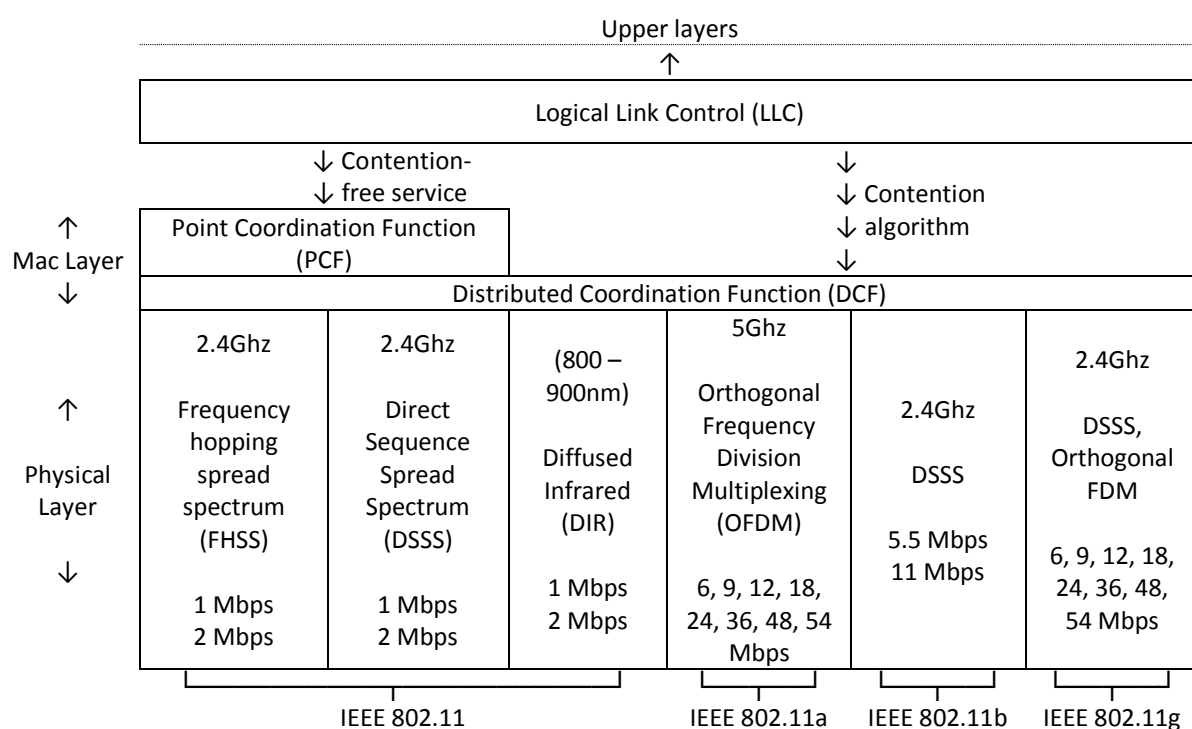
Figure 35: *The upper layers of the IEEE 802.11n Protocols alongside their slightly different standards.*

The lowest layer of the IEEE 802.11 protocol is the physical layer, defining the operating frequency bands, the supported data rates, and the details of radio transmission. There are various IEEE 802.11 standards, all of which have slightly different physical criterions which are caused by the operating frequency band and modulation techniques used as seen in the model above. The IEEE 802.11b protocol is the most popular and widespread, as it is capable of delivering a throughput of up to 11 Mbps, although the observed throughput is considerably lesser at about 6 Mbps due to the fact it faces interference from microwave ovens, cordless phones, and other such devices.

IEEE 802.15.4 in particular was designed for lightweight devices to allow even lower cost and power consumption than the 802.11 variation. This was achieved by opting for a 10-meter communications range with a decreased transfer rate of only 250 Kbps, with the ability to go as low as 20 Kbps. This trade-off allowed WPAN's (Wireless Personal Area Network) to have exceptionally low operation and manufacturing costs of various applications, without sacrificing flexibility or generality.

The Tera Term package, a software implemented terminal emulator, can be used to support serial port connections. It was designed for users who want to migrate their current 802.15.4 architecture to a standard TCP/IP based platform, without needing to redesign any of their existing hardware. This allows for projects to be moved to a standard Wi-Fi network without needing any other new hardware. Configuring the WiFly chip with the Tera Term package, it can be used to transmit data via the XMODEM-1K protocol, a simple file transfer protocol that breaks up the original data into a series of packets, before sending them across the setup wireless network.

Like most file transfer protocols, XMODEM has built in error detection, to make the correct data is received. If the data structure sent across the network does not match the received structure, it is assumed to be corrupted, resulting in it being discarded. A request for the data to be resubmitted by the receiver is then sent back to the original sender. An outline of how the XMODEM-1K protocol was set up for this wireless Arduino communication can be seen demonstrated below.

The XMODEM-1K Protocol

Start off by connecting the RN module to the PC and open Tera Term. For an increase in download speeds, you can set the baud rate to 230400. This can be done via the following WiFly commands.

```
set uart baud 230400
set uart flow 1            // enables the UART flow control save reboot
save
reboot
```

Once the baud rate is set to 230400, enable Hardware flow control, and enter the flowing WiFly command to enable Xmodem mode.

```
xmodem <option> <filename>
```

where `<option>` is:

`u` – download firmware and set as boot image, `<filename>` is the name of the firmware (this file type can be either `.img` or `.mif`)

`c` – clean the file system before performing firmware update, deleting all the files on the flash file system (including user defined configuration files), except the current boot image and the factory default boot image.

An example WiFly command is given below.
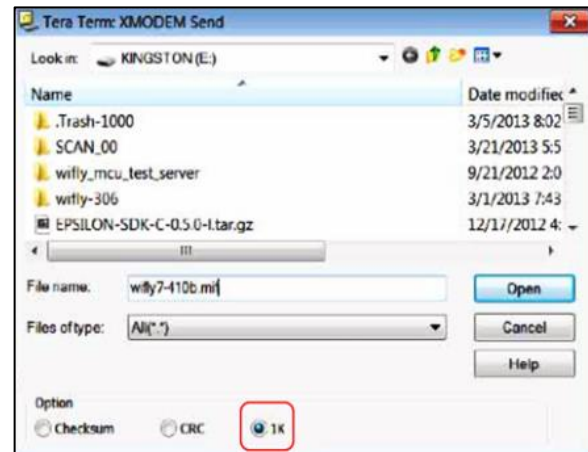
```
xmodem cu wifly7-400.mif
```

Once the command is entered, the flowing outputs appears.

```
xmodem cu wifly7-400.mif
del 4 wifly-EZX-405
del 5 config
del 6 reboot
del 8 logo.png
del 13 wps_app-EZX-131
del 14 eap_app-EZX-105
del 15 web_app-EZX-112
del 16 web_config.html
del 17 link.html
xmodem ready...
```

Once XMODEM is ready on the UART, proceed to the XMODEM file transfer option in Tera Term. To do this, select *File > Transfer > XMODEM > Send*, as seen below in **Figure 36**. In the following options, select *1K*, enter the `.img` or `.mif` file path, and click *Open*, as seen below in **Figure 37**.
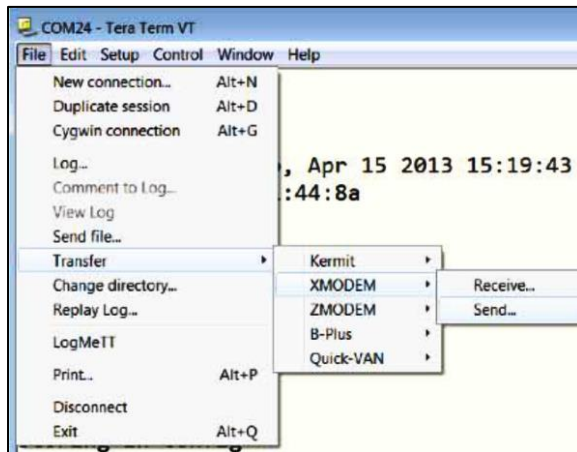

Figure 36: The XMODEM File Transfer Option.   *Figure 37: The 1K and file path sending options.*

The new firmware and any required web files will now download to the RN module, followed by an **XMOD OK** message. To check that the firmware has been downloaded correct, issue a ls command to ensure everything is working as it should. By default, the XMODEM has a 30 second timeout, which can be disabled with the command `set ftp timeout 0`.
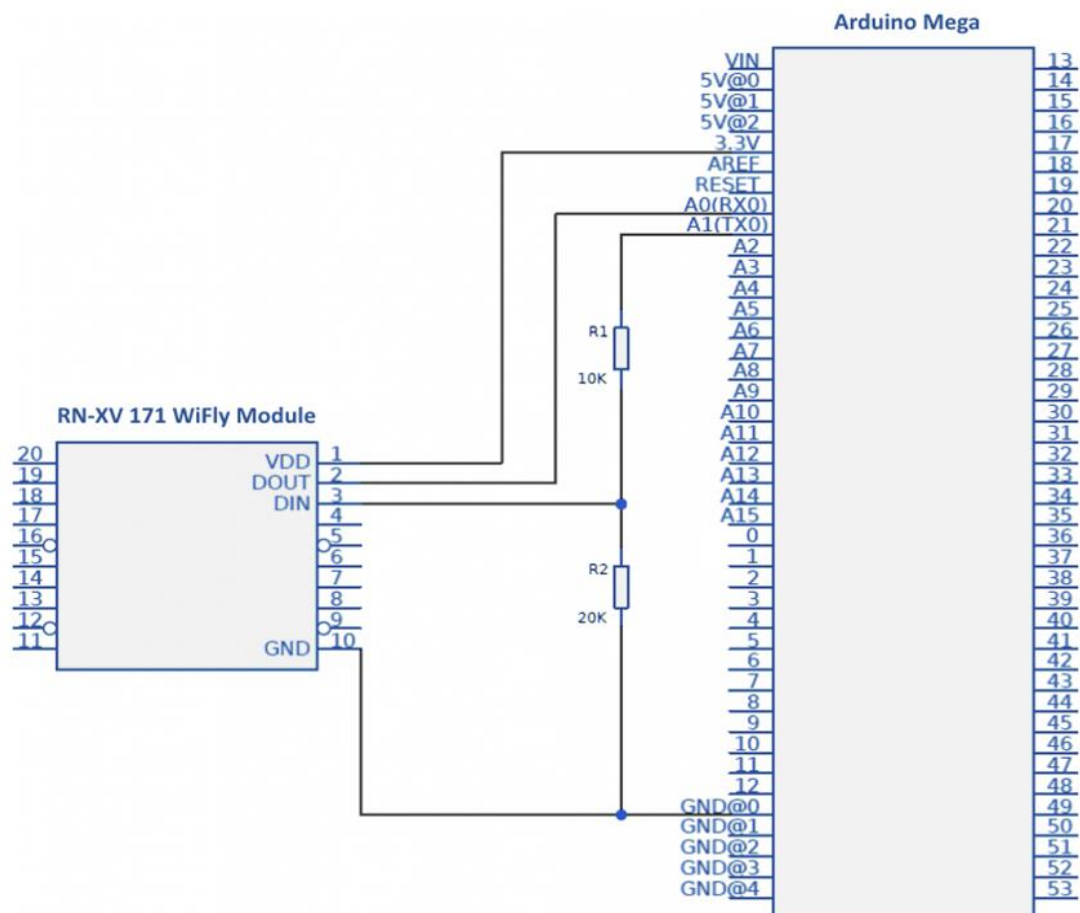
Figure 38: A hardware configuration example of the WiFly Chip connected to an Arduino Mega.

To access the UART (Universal Asynchronous Receiver-Transmitter) interface for communication, you can use an Arduino Mega as a USB to TTL converter by connecting the boards Reset and Ground pins, as seen in **Figure 38**. This enables you to use the RX and TX pins with the WiFly Module. The serial communications settings should match the WiFly module's stored settings; by default, the settings are 9.600 bauds, 8 bits, no parity, 1 stop bit and hardware flow controlled disabled.

UART Connection Code

```
String cmd = "";

void setup() {
        Serial.begin(9600);
        pinMode(13, OUTPUT);
        configureContact();
}

void configureContact() {

        while (Serial.available() <= 0) {
                Serial.print('X', BYTE);             // sends a capital X
                delay(1000);                         // delays for 1 second
        }
}

void loop() {

        if (Serial.available() > 0) {                // incoming byte
                char incomeByte = Serial.read();
                cmd = cmd + incomeByte;
                checkCmd();
        }
}

void checkCmd() {

        if(cmd.endsWith("\r\n")) {

                if (cmd == "OFF\r\n") {
                        digitalWrite(13, LOW);
                } else if (cmd == "ON\r\n") {
                        digitalWrite(13, HIGH);
                }
                cmd = "";
        }
}
```

The setup function initializes serial communication and sets digital pin 13 as an output for turning the built in LED on and off. The `configureContact()` function begins the communication, where an ASCII character is sent through the serial connection until there is a reply from the WiFly module, relaying the character over a WiFi network. The main loop continues indefinitely whilst the hardware is enabled, resulting in the code constantly waiting for a byte of data from the WiFly module. When there is data, it is appended to the global variable `cmd`, which is currently an empty string object. As the global variable `cmd` is being amended, it is being checked whether it is a command or not. This is done by simply checking whether it is a line ended by carriage return and line feed character *(CR = \r and LF = \n)*. Depending on the command, the LED connected to digital pin 13 is either turned on or turned off. At the end of the check function, the global variable `cmd` is cleared, ready to be filled up by another series of character that will build the next command.

Final Reflection

The first task was in my opinion the easiest of them all, as I already had plenty of experience with binary arithmetic. The second task was also relatively easy once I figured out how use to replicate the functions of binary anthemic from excel into code. The third task however was the one I found to be the most challenging initially, as I had no previous experience using *nop* commands to create delays. The main issue I had with this program however was the fact that I couldn't figure out how to get the program to output several different pins independently from each other with only one line of code, so that no extra delay would be added to the program. After a long time, I finally figured out how to use the hexadecimal system to change the output of several pins at the same time with only one line of code. This is something new that I have learnt how to do, which I will be able to use later in future projects. Another thing that I found challenging initially with this task was understanding some of the default instruction sets, such as RETI. This was not too hard to overcome; however, I did find myself having to look up and check often that I was using the correct sets during the task.

I chose to compact the code for each of the traffic light phases, to make the code more efficient. If I were to expand on the traffic light program further, for example by adding in multiple sets of lights, then the compacted code would be extremely useful due to many more LEDs states being used. One change I also considered but ultimately decided against was adding a third nested loop in the delay subroutine. This would have compacted the code even further by allowing just one *nop* command to be looped around. This may have increased the error margin beyond 0.05%, however, since another BRNE instruction is being used. As mentioned in the explanation of the code, this may have caused the number of milliseconds to differ too much, and so ultimately, I decided against it. The main reason for the code not always being exactly 2000 milliseconds is because some of the instructions used, such as BRNE, can have either 1 or 2 cycles. If the condition is false, the program will only have gone through 1 cycle. If the condition is true, however, the program will go through an extra cycle to loop back to the desired location. Changing the program to follow a different flowchart could eliminate the error margin. For example, an extra *nop* command could be cycled through if a certain number of cycles had not yet been reached, resulting in less of an error margin.

One thing I learned when writing code in assembly is that the code can be executed faster than if I were to write it in C, however I personally found it harder to write code in assembly. I felt there was a steeper learning curve compared to when I learned other languages, such as C, as it seemed less intuitive in comparison. I believe that the increase of efficiency in terms of speed does not outweigh the time it takes to create and maintain the code itself. This is because I ended up spending a larger amount of time trying to debug certain sections of code, compared to previous projects where I had coded in C. I believe that both assembly and C have their uses when it comes to writing programs, but for this one I believe there is no real benefit for choosing assembly over C. Since this program does not take up a large amount of memory, I believe the efficiency gained by writing this code in assembly does not outweigh the practically of writing it in C.

Upon reflection of my work, I believe I have expanded on my knowledge of how logic-based systems can be represented, especially with the aid of Arduinos. I made sure to test the code controlling the Arduinos throughout in order to make sure that no bugs creeped up, using the flowcharts to outline the logic of my program to aid me in visualising the steps when coding the task. If I were to attempt a similar project in the future, I would definitely create more flowcharts again, as they helped me to visualise my code. I would also research further how an Arduino could be utilised, in order to further my understanding of utilising WI-FI adapters, potentially finding a way to connect it to a Raspberry Pi for example. Overall, I am happy with the knowledge I have gained from completing these tasks.