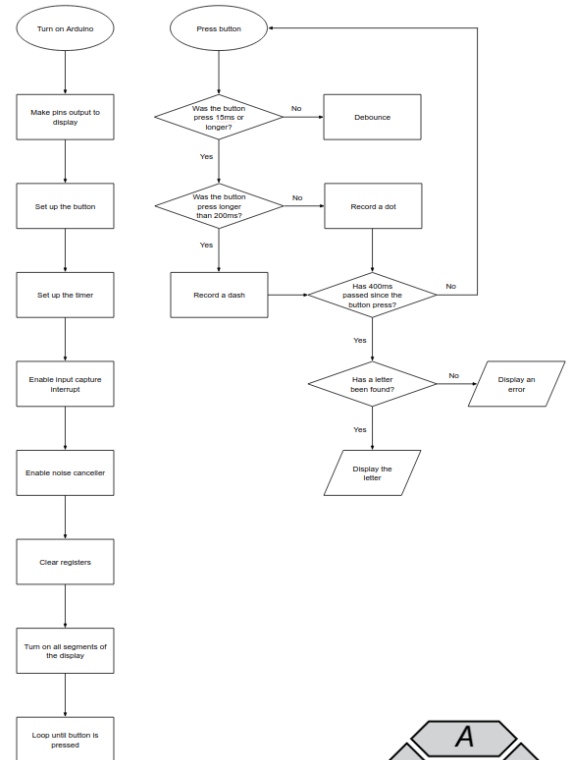


# Group Coursework Report

## Overview of Implementation

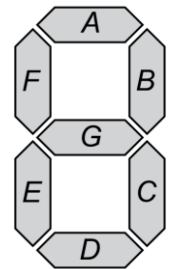
This implementation can, as required, decode any letter in the alphabet (A-Z). If the button is pressed for 15ms or longer - note that any shorter duration is debounced - then the program will attempt to interpret the signal. If the signal is 200ms or less, then a dot is recorded; if it is longer than 200ms, then a dash is recorded. Up to four dots and/or dashes may theoretically be inputted before an error automatically shows on the display. If a character matching the inputted dots and dashes is found, said character will be output onto the display - otherwise, again, an error symbol will be displayed. Our error symbol is represented as three horizontal lines.

Full-sized flowchart: [1]



## Details and Challenges of Implementation







Our solution includes two .S files. The first, get\_morse.S, has two primary functions. Firstly, it contains an index of each letter in the alphabet and its corresponding required segments of the display. Letters A to G are used to refer to an individual segment of the display - these correspond with a command to turn on a bit (display segment) numbered 1 to 7. For example, calling `_BV(7)` will turn on A, the top segment of the display, as shown in the diagram to the right. [2]



Secondly, it converts the morse code into the character that will be output onto the display. It makes use of the lower and upper bytes of Z. Z, in turn, points to a hash table - this represents all the necessary characters (A-Z). It uses a stack with the help of registers to alter these bytes - if an overflow occurs, 1 will be added to the upper byte of Z. This value may then be loaded into r24, which is the register holding the value of the morse code.

The second file, group\_3941.S, measures button presses and hence records dots and dashes in the correct order. Whether it records a dot or a dash is dependent on the duration of the button press. If the button press is less than 200ms, a dot is recorded. Otherwise, if the button press is longer than 200ms, a dash will be recorded. Up to four dots and/or dashes can be recorded before the error state automatically appears (note that this is not included in char.S) - this is because no letter of the alphabet requires more than four dots and/or dashes. At the very start of the file, it also enables output through PORTD, and turns on all segments of the display. Three registers are used - one to count the number of symbols inputted (r20); one to store dots (r21), and one to store dashes (r22).

One of the challenges of using a seven-segment display was its limitations in representing complex letters. Examples include K, M, V, W and X. Below is a diagram of how we represented these letters. In addition, we also had to make conscious choices about using the upper-case or lower-case variant of the letter in order to represent each letter in the clearest way possible.

					
K	M	V	W	X	error

[3] [4]

To follow on from this, we had to decide on a common symbol to represent an error. In the end, we decided to use three horizontal lines, as this would be difficult to mistake for a letter.

Another challenge was that we were unable to bug test the code easily, particularly the code that converts button presses into morse code. As such, we were forced to overcome this by manually debugging, line by line.

One more challenge was the difficulty of representing morse code as a letter. In order to overcome this, we made a hash table in which each morse code sequence if interpreted as a number gives the index of the target letter in the table. The letter is stored as a byte of its representation on the 7 segment display.

## Group Contribution

**Oliver Pinder** was the primary developer of the interpreter, and he came up with the brilliant idea of the hash table. He also helped with building the circuit on occasion.

**Phoebe Revolta** provided significant understanding of assembly language, particularly timers and interrupts. This helped her write the comments for the code.

**Rebecca Gunstone** helped to convert the provided diagram into a functional circuit with the Arduino. She, along with Josh, helped to write the initial versions of the interpreter and 7 segment display output.

**Joshua Miller** primarily wrote the report and designed the flowchart to represent our code. He, along with Rebecca, helped to write the initial versions of the interpreter and 7 segment display output.

**Imogen Hay** helped Phoebe in further understanding assembly language then used this information to assist in implementing code particularly for detecting button presses. She ensured that all group members did their parts correctly and that deadlines were met.

**Alan Chamathil** only turned up once. When he did turn up, he provided no input.

[1] Flowchart:

<https://docs.google.com/drawings/d/16gHXWPwSKtHI6FBDvd0c8rUBhvNSGFbU5eaTT18DDZc>

[2] Representation of letters on 7-segment display:

[https://upload.wikimedia.org/wikipedia/commons/e/ed/7\\_Segment\\_Display\\_with\\_Labeled\\_Segments.svg](https://upload.wikimedia.org/wikipedia/commons/e/ed/7_Segment_Display_with_Labeled_Segments.svg)

[3] Alphabet idea came from this link:

<https://codegolf.stackexchange.com/questions/173837/longest-seven-segment-word>

[4] Images of 7-segment display were taken from this link:

<https://upload.wikimedia.org/wikipedia/commons/d/d1/7-segment.svg>