COM2108 – Autumn Functional Programming Assignment

Thomas Pearson

# The Enigma Machine Design

#### First Design Document

Upon reading the documentation on encoding the enigma simulation, I created Figure 1 (below), a simplified flow chart in order to visualise the operation of the enigma machine.

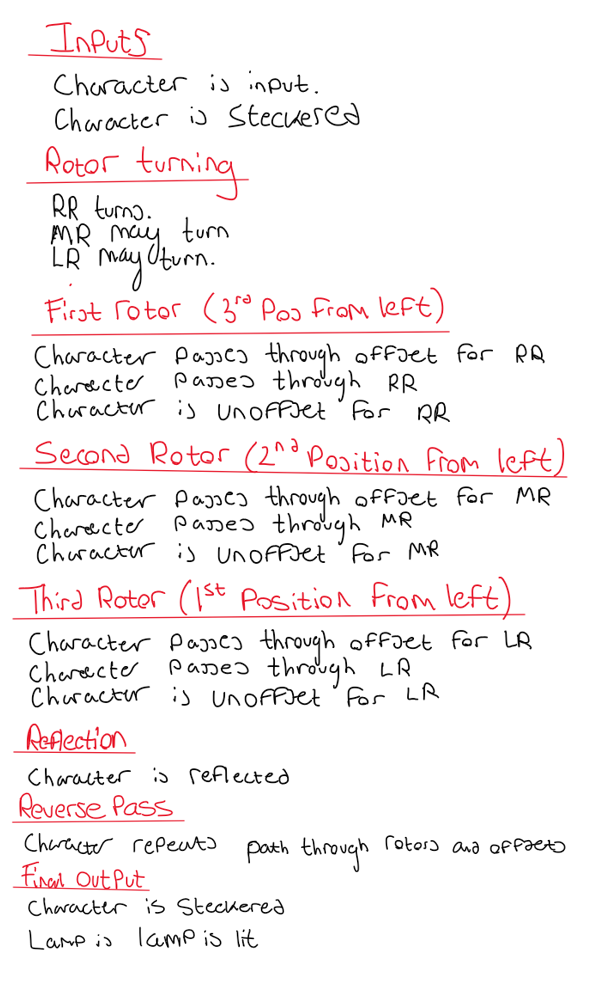


Figure – Design First Draft

#### Rotor Turning

In this implementation the offset of the rotors is shifted after a character is pressed, but before it is ciphered (as shown in Figure 1). The knock-on offset position is also triggered upon reaching it. For example, if the knock-on position was position 17, the knock-on event (turning the connecting rotor) would be triggered upon the rotation from position 16 to position 17. However, in this implementation the characters are offset not the actual rotors. This is discussed further below.

#### Character Offsets

As demonstrated by Figure 1, instead of turning the rotors, which would be very resource intensive, I have decided to offset the input by the rotor amount. This way it is easier to keep track of the position of the current rotor and therefore the needed offset. It also ensures that the connecting rotor receives the correct character when the letter is passed to the next rotor.

For example, if the “Rotor I” was offset by three positions and the character “A” was input it would process the information as follows:

1. Shift “A” along the alphabet by three position to get “D”.
2. Pass letter “D” through the rotor to get “F”
3. Shift “F” backwards by three positions through the alphabet to get character “C”
4. Pass character “C” on to the next rotor and repeat the same process

This approach simulates the rotor having shifted three positions physically as the connecting rotor is unaware of the shifted previous rotor. It is only away of the position of the incoming signal in its own rotor. If we were to not “unshift” the encrypted value it would be receiving the incorrect value from the previous rotor.

#### Second Design Iteration

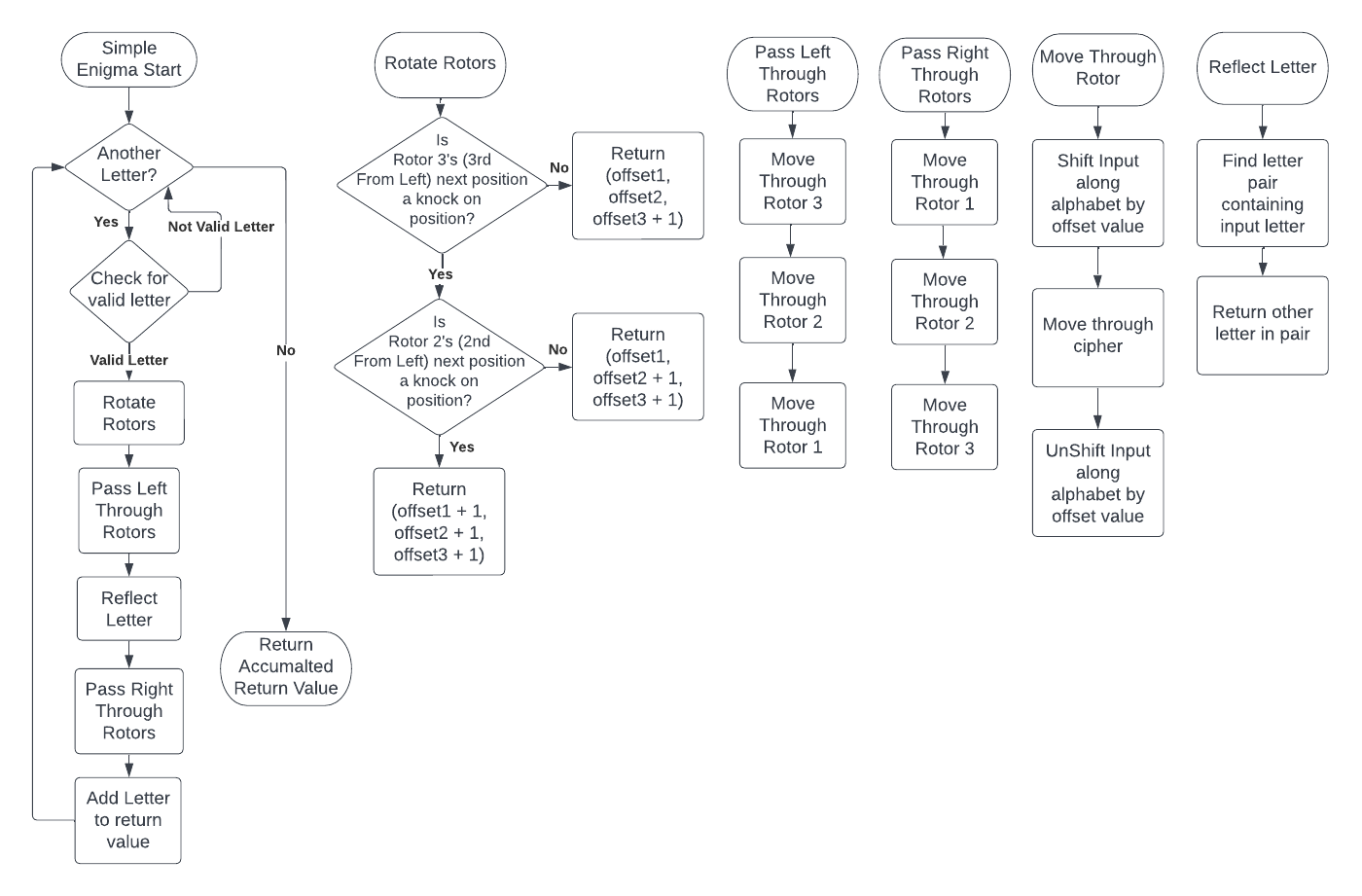
Once the main parts of the design were created it was divided further into smaller functions. The flow of these proposed functions is shown in Figures 2 and 3.

Figure - Simple Enigma Operation

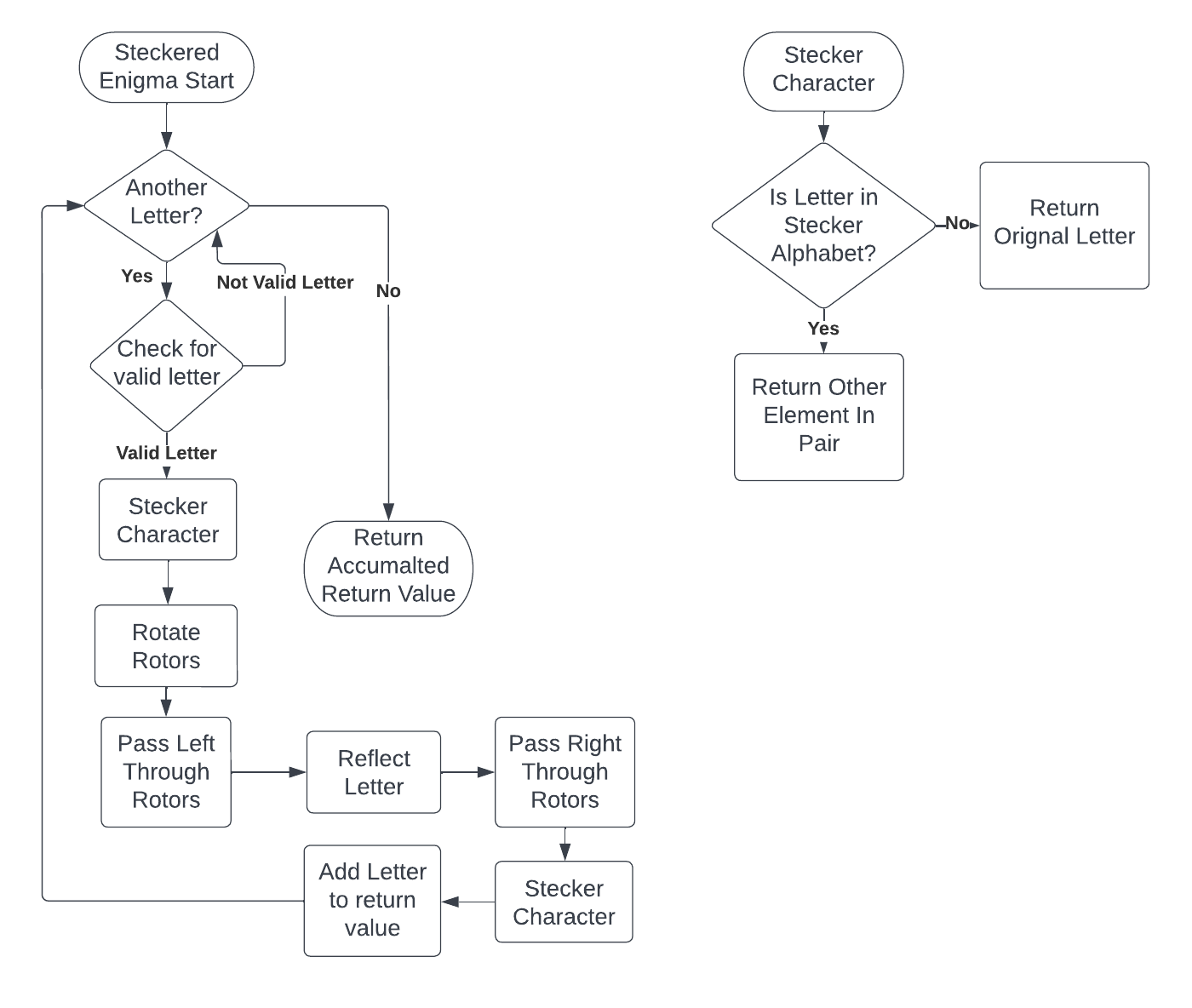


Figure - Steckered Enigma Operation

# Enigma Machine Testing

The first functions that were implemented were the ones that had the least dependencies on other functions. The table below show the order the functions were implemented.

#### Reflector Function

Named “reflectorFunction” in the implementation, it simply takes in a character and list of reflector pairs to find the input character’s paired letter.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Character, Reflector) | Rational Behind Values | Expected | Output |
| ‘A’, reflectorB | Test for values input on the left of the pair are reflected | ‘Y’ | ‘Y’ |
| ‘B’, reflectorB | Test for element other than first | ‘R’ | ‘R’ |
| ‘W’, reflectorB | Test for values input on the right of the pair reflected. | ‘V’ | ‘V’ |

#### Rotate Rotors

Named “rotateRotors” in the implementation. It takes in three separate rotors and an array of their current offsets. If any rotor’s next position is a knock-on position the next rotor in the series will turn. If any rotor’s increment hits 26 it will return to position 0 (rotor has returned to starting position).

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Rotor, Rotor, Rotor, Offsets) | Rational Behind Values | Expected | Output |
| rotor1, rotor2, rotor3, [0,0,0] | First shift for starting input | [0,0,1] | [0,0,1] |
| rotor1, rotor2, rotor3, [0,1,25] | Checks for return to starting value | [0,1,0] | [0,1,0] |
| rotor1, rotor2, rotor3 [0,0,21] | Checks for knock on position for third rotor | [0,1,22] | [0,1,22] |
| rotor1, rotor2, rotor3 [0,4,21] | Checks for dual turning of rotors | [1,5,22] | [1,5,22] |
| rotor1, rotor2, rotor3 [16,4,21] | Ensures first rotor has no impact on other rotors | [17,5,22] | [17,5,22] |

#### Integer to Letter Conversion

Named “int2let” in code. It performs the reverse of the “alphaPos” provided function and converts a letter index into its equivalent letter.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Int) | Rational Behind Values | Expected | Output |
| 0 | Check function converts indexes correctly | ‘A’ | ‘A’ |
| 25 | Correct last letter of alphabet | ‘Z’ | ‘Z’ |
| 13 | Random check | ‘N’ | ‘N’ |

#### Shift Letter in the Alphabet

Named “shiftInput” in the code. It moves the letter input forwards in the alphabet by the specified input number of spaces. It will loop back around to the start of the alphabet if the input goes past ‘Z’ value.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Int, Character) | Rational Behind Values | Expected | Output |
| 0, ‘A’ | Base case | ‘A’ | ‘A’ |
| 1, ‘Z’ | Loop back to first letter of alphabet | ‘A’ | ‘A’ |
| 25, ‘Z’ | Test of function | ‘Y’ | ‘Y’ |
| 4, ‘B’ | Test of function | ‘F’ | ‘F’ |
| 30, ‘A’ | Test if too large of input entered | ‘E’ | ‘E’ |

#### Unshift Letter in the Alphabet

Named “unShiftInput” in the code. This function performs the opposite to “shiftInput” and moves a letter backwards along the alphabet by the specified number of positions. It also loops back to the end of the alphabet if the letter goes past the start of the alphabet.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Int, Character) | Rational Behind Values | Expected | Output |
| 0, ‘A’ | Base Case | ‘A’ | ‘A’ |
| 1, ‘A’ | Loop back to end of the alphabet | ‘Z’ | ‘Z’ |
| 5, ‘H’ | Test functionality | ‘C’ | ‘C’ |
| 30, ‘A’ | Test if too large of input entered | ‘W’ | ‘W’ |

#### Find Passed Through Value

Named “findLetterPosition” in the code. This function takes an array of characters, a character to look for and a counter which by default should be 0. Its purpose it to look for the position of the character in the provided array. This is useful when we passing a character back through a rotor. We have the current value that is being passed back but not its position. This function finds it.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input ([Char], Char, Int) | Rational Behind Values | Expected | Output |
| “EKMFLGDQVZNTOWYHXUSPAIBRCJ”, ‘E’, 0 | Base case | 0 | 0 |
| “EKMFLGDQVZNTOWYHXUSPAIBRCJ”, ‘J’, 0 | End of character array | 25 | 25 |
| “EKMFLGDQVZNTOWYHXUSPAIBRCJ”, ‘A’, 20 | Functionality Checker | 20 | 20 |

#### Pass Left Through Rotors

Named “passLeft” in the code. This function takes a rotor, a character to cipher and the offset of the rotor. First it shifts the input character by a desired number of positions using “shiftInput”. Then it finds the index of the character in the alphabet to find the index of the equivalent value in the rotor. Finally, it unshifts the result by the number of places provided using “unShiftInput”.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Rotor, Character, Int) | Rational Behind Values | Expected | Output |
| Rotor1, ‘A’, 0 | Base case test | ‘E’ | ‘E’ |
| Rotor2, ‘A’, 0 | Base case test | ‘A’ | ‘A’ |
| Rotor1, ‘A’, 1 | Offset increased | ‘J’ | ‘J’ |
| Rotor1, ‘B’, 1 | Offset increased and character changed | ‘L’ | ‘L’ |
| Rotor1, ‘A’, 26 | Checks that result loops back around if offset exceeds alphabet length | ‘E’ | ‘E’ |
| Rotor2, ‘Z’, 30 | Checks that result loops back around if offset exceeds alphabet length | ‘B’ | ‘B’ |

#### Pass Right Through Rotors

Named “passRight” in the code. This function’s operation is similar to pass left except that instead of passing in the letter’s index in the alphabet it uses the letter’s index in the rotor. For example in rotor 1 its alphabet is “EKMFLGDQVZNTOWYHXUSPAIBRCJ”, the letter M is at index 2 (starting at 0). This is neccesary to pass the letter backwards through the rotor.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Rotor, Character, Int) | Rational Behind Values | Expected | Output |
| Rotor1, ‘A’, 0 | Base case test | ‘U’ | ‘U’ |
| Rotor2, ‘A’, 0 | Base case test | ‘A’ | ‘A’ |
| Rotor1, ‘A’, 1 | Offset increased | ‘V’ | ‘V’ |
| Rotor1, ‘B’, 1 | Offset increased and character changed | ‘X’ | ‘X’ |
| Rotor1, ‘A’, 26 | Checks that result loops back around if offset exceeds alphabet length | ‘U’ | ‘U’ |
| Rotor2, ‘Z’, 30 | Checks that result loops back around if offset exceeds alphabet length | ‘Y’ | ‘Y’ |

#### Stecker Input (Plug Board)

Named “steckerPass” in the code. Its purpose is to stecker the input based on the provided stecker pairs. If the letter is not found in the stecker pairs then it will return the original letter.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Int, Character) | Rational Behind Values | Expected | Output |
| 'A', [('F','T'), ('D','U'), ('V','A'), ('K','W'), ('H','Z'), ('I','X')] | Right side value | ‘V’ | ‘V’ |
| 'I', [('F','T'), ('D','U'), ('V','A'), ('K','W'), ('H','Z'), ('I','X')] | Left side value | ‘X’ | ‘X’ |
| 'A', [('F','T'), ('D','U'), ('V','A'), ('K','W'), ('H','Z'), ('I','X')] | Value not in stecker | ‘B’ | ‘B’ |

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Int, Character) | Rational Behind Values | Expected | Output |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Int, Character) | Rational Behind Values | Expected | Output |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Int, Character) | Rational Behind Values | Expected | Output |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Test Input (Int, Character) | Rational Behind Values | Expected | Output |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |