|  |
| --- |
| **QR Code Generator with Error Correction** |
| AQA GCE Computing COMP4 Project Report |
|  |
| A detailed breakdown on my investigation into how QR codes are generated based on user-entered data including the built in error correction capabilities which allows a portion of the symbol to be covered and the data still to be scanned and extracted |
|  |
| **Umar Chauhan Ali** |
| **Beckfoot School 37101**  **2015-2017** |
|  |

Contents

[Analysis 3](#_Toc35177854)

[Introduction 3](#_Toc35177855)

[Investigation outline – Acceptable Limitations 4](#_Toc35177856)

[Conversation with supervisor (1) 4](#_Toc35177857)

[Research After Conversation (1) – Data Encoding 5](#_Toc35177858)

[Conversation with supervisor (2) 5](#_Toc35177859)

[Research After Conversation (2) – Polynomial Division 5](#_Toc35177860)

[Conversation with supervisor (3) 6](#_Toc35177861)

[Research After Conversation (3) – Polynomial Division Evidence 7](#_Toc35177862)

[Research After Conversation (3) – Galois Fields 8](#_Toc35177863)

[Research – Masking 8](#_Toc35177864)

[Research – Masking Evaluation Conditions 10](#_Toc35177865)

[Research – Function Patterns 16](#_Toc35177866)

[System Objectives / Specification 20](#_Toc35177867)

[Critical Path 21](#_Toc35177868)

[Design 22](#_Toc35177869)

[Table of Key Variables 22](#_Toc35177870)

[Key Function – Data Codeword Generation 26](#_Toc35177871)

[Key Function – Error Correction Codeword Generation 28](#_Toc35177872)

[Key Function – Constructing Final Bit Stream 29](#_Toc35177873)

[Data Structure – Matrix 30](#_Toc35177874)

[Key Function – Data Placement in Matrix 31](#_Toc35177875)

[Key Function – Masking 34](#_Toc35177876)

[Key Function – Masking Evaluation 42](#_Toc35177877)

[Class – Current Module 48](#_Toc35177878)

[Testing 49](#_Toc35177879)

[Testing Evidence 53](#_Toc35177880)

[Test 1 – Empty String 53](#_Toc35177881)

[Test 2 – Illegal Character 54](#_Toc35177882)

[Test 3 - Exceeding Capacity 55](#_Toc35177883)

[Test 4 – Letter Data Only 56](#_Toc35177884)

[Test 5 - Numeric Data Only 57](#_Toc35177885)

[Test 6 - Symbol Data Only 58](#_Toc35177886)

[Test 7 - Mixed Data 59](#_Toc35177887)

[Test 8 – Error Correction: Minor Damage 60](#_Toc35177888)

[Test 9 – Error Correction: Medium Damage 61](#_Toc35177889)

[Test 10 - Error Correction: Major Damage 62](#_Toc35177890)

[Test 11 - Error Correction: Exceeding Damage 63](#_Toc35177891)

[Technical Solution 64](#_Toc35177892)

[Evaluation 86](#_Toc35177893)

[Conversation with Supervisor 86](#_Toc35177894)

[Meeting System Objectives 86](#_Toc35177895)

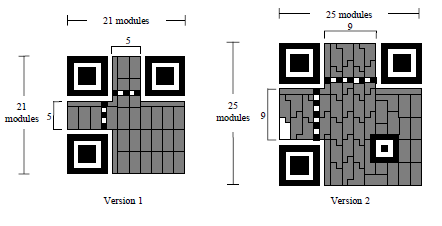
[Desirable Additions 87](#_Toc35177896)

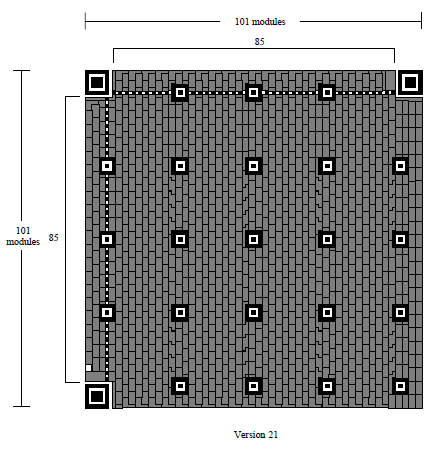
# Analysis

## Introduction

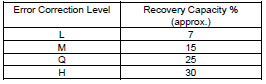
QR Codes (Quick Response codes) are a form of 2-D machine-readable optical barcode which is capable of encoding and storing a wide variety of data types such as numerical (digits 0-9), alphanumerical (digits 0-9, uppercase letters A-Z, and 9 other characters including space and various common symbols), JIS 8-bit character set (Latin and Kana) and Kanji characters. The QR codes themselves are represented by a matrix of black and white symbols arranged in a square and can be read using a variety of imaging devices such as phone cameras.

There are 40 sizes of QR Codes commonly referred to as versions. As the version numbers increase, so does the maximum capacity of data that can be encoded which in turn means the physical size of the produced symbol increases. Version 1 measures 21 x 21 modules (where a module is either a black or white square in the matrix), version 2 measures 25 x 25 and so on, increasing in steps of 4 modules both in the rows and columns. The largest version is 40 which measures 177 x 177 modules.





QR Codes employ Reed-Solomon error correction encoding techniques to generate, from the entered data, a sequence of unique error correction codewords which will be woven into the sequence of encoded data codewords in order to allow the QR symbol to still be scanned, with all the original data still intact, even if a portion of it is covered. There are four levels of error correction, each allowing a different level of recovery capability.



Additionally, once the all the encoded data has been placed within the QR Code, masking patterns are applied. Essentially bit patterns are XOR’ed onto the QR Code symbol itself, which don’t change the fundamental data but do aim to make the data easier to scan by doing things like removing large areas of the same color, removing data that could be confused for a Finder Pattern etc. There are 8 Mask Patterns available and each one must be applied to the QR Code in turn. Once a Mask Pattern is applied, it should be assessed in its effectiveness against certain conditions and scored accordingly. The Mask Pattern with the lowest penalty score should be the one chosen and should be re-applied to the original symbol.

## Investigation outline – Acceptable Limitations

I have chosen to do an ‘Investigation’ rather than a ‘Solution to a Problem’ as realistically no one would be using my solution in a real world capacity, many solutions for encoding QR codes already exist online. Rather an investigation is far more appropriate as it gives me the capacity to explore how QR codes are generated without having to worry about development for a 3rd party’s use.

The programming aspect of this investigation was influenced by the ISO/IEC 18004 International Standard for QR codes which details how exactly QR codes are encoded and built as well as the thonky.com website tutorial on QR Codes which provided a more illustrative detailing on how QR Codes are created. Many of the tables in this project write up have been taken from the ISO QR Standard document and my algorithms in the technical solution were developed based on the standard guidelines in the document.

The scope of my project as well as a few technical details were decided based on a couple of conversations with my supervisor, who is Mr. Kendall, which have all been detailed in this write-up.

## Conversation with supervisor (1)

**Supervisor:** “What is the scope of your project, what will the end result be?”

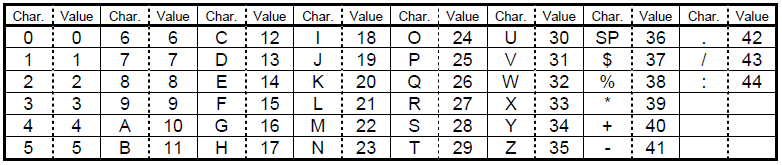
**Me:** “The end user should be able to enter some data and a QR Code will be generated and displayed for them. The user should then be able to scan the QR Code and retrieve the data they originally entered.”

**Supervisor:** “You should look into the different types of data a QR Code can encode as from past experience that different types of data can be encoded in different ways. As you’re doing an investigation it may be worth sticking to a single data type. I would recommend only encoding data listed in the ‘Alphanumeric’ category.

**Me:** “I’ll look into the different types of data that can be encoded.”

## Research After Conversation (1) – Data Encoding

After the first conversation with my supervisor, I read through the first few sections of the ISO QR Standard document and realized that there were a number of types of data that could be encoded, the encoding process for each being quite different between data types. I took my supervisors advice and decided to stick with only encoding Alphanumeric characters as it offered more end-usability than Numeric as the user would be able to encode actual words, and it didn’t seem necessary to encode Kanji as this was comprised mainly of Non-English characters. Below you will find a table of all the characters that the Alphanumeric coding mode is able to encode. As you can see it still offers a wide variety of characters, numbers and symbols available to be encoded.



## Conversation with supervisor (2)

**Me:** “I have decided to stick with just encoding alphanumeric characters.”

**Supervisor:** “Good. What will the key features of your program be once it is complete?”

**Me:** “My program will be able to dynamically choose a version size based on the text entered, it will be able to generate error correction codewords and display a scan able QR Code for the user.”

**Supervisor:** “I would advise that you stick to a set version size and error correction level. As this is just an investigation you should be focusing on making a proof of concept. Dynamically choosing a version size would just be adding un-necessary complexity. The complexity of your program should instead come from generating the error correction codewords as it involves some polynomial division. A good next step would be to practice polynomial division”

## Research After Conversation (2) – Polynomial Division

After the second conversion I decided to stick to a set Version and Error Correction Level and perhaps look at dynamically choosing a Version as a desirable feature in the Evaluation section. I looked into how the error correction is built into the QR Code itself and discovered that Reed Solomon Error correction is used. I also discovered that a large area of generating the error correction data included polynomial long division i.e. dividing one polynomial by another. My understanding is that these are the steps involved:

1. Find the appropriate term to multiply the divisor by. The result of the multiplication should have the same first term as the dividend (in the first multiplication step) or remainder (in all subsequent multiplication steps).
2. Subtract the result from the dividend (in the first multiplication step) or remainder (in all subsequent multiplication steps).
3. Repeat steps 1 and 2 until it is no longer possible to multiply by an integer. The remainder is whatever is left.

## Conversation with supervisor (3)

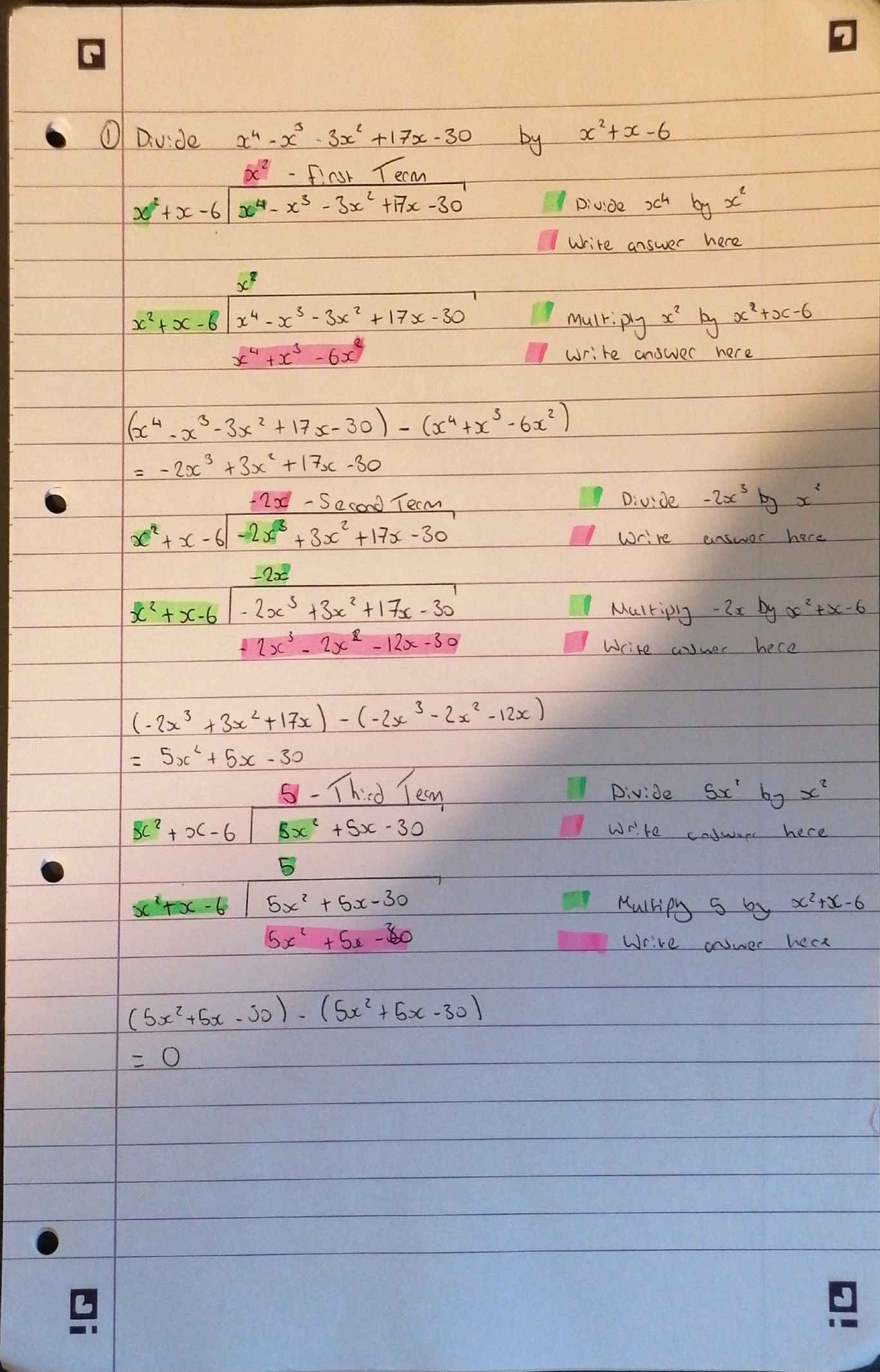
**Me:** “I have reminded myself about polynomial division and I am confident that I can replicate it in my program.”

**Supervisor: “**As part of the error correction step you will be required to store numbers larger than the computer is able to hold .You should also familiarize yourself with Galois Field Arithmetic and Alpha Notation as it is my understanding that this is one was in which you will be able to condense massive polynomial coefficients down and actually work with them. This will likely be the hardest section of the project as it is difficult to understand and even more difficult to implement in a program.”

**Me:** “How would I demonstrate my understanding of Galois Fields, polynomial division etc?”

**Supervisor:** “It would probably be a good idea to work through some examples of polynomial division and Galois Field arithmetic on paper and then simply take pictures and add them to your write up.”

## Research After Conversation (3) – Polynomial Division Evidence

The following image is evidence of me practicising polynomial division following the third converstation with my supervisor. It helped me to understand the steps involved with polynomial division and how I would be able to replicate this process in an algorithm.

## Research After Conversation (3) – Galois Fields

The ISO QR Standard Document says to use bite-wise modulo 2 arithmetic and bye-wise modulo 100011101 arithmetic. This essentially means use a Galois Field 256 which contains all the numbers 0-255 inclusively which is the same range that can be represented by a 8 bit unsigned binary byte (0000 0000 – 1111 1111). This means that all Galois Field mathematical operations will result in numbers that can be represented by a byte, which means that if any mathematical operation results in a number that is outside GF(256) i.e. 256 or larger, it will be necessary to use byte-wise 285 module arithmetic to ensure the result of any calculation falls within the range of GF(256). Essentially the rule is if a mathematical operation results in a number greater than 255, XOR with 285 (it may be necessary to repeatedly apply XOR 285 to get the result in the range of GF(256).

All numbers in GF(256) can be represented as 2n where 0<= n <=255. However we quickly run into an issue as 28 = 256 which is outside the range of GF(256). As mentioned before we can solve this problem by applying a byte-wise modulo 100011101 i.e. If a number is greater than 255, XOR it with 285 until it is within the desired range. Following this rule the powers of 2 can be represented as:

20 = 1

21 = 2

22 = 4

23 = 8

24 = 16

25 = 32

26 = 64

27 = 128

28  = 256 = (256 XOR 285) = 29

29 = 28 \* 2 = 29 \* 2 = 58

210 = 29 \* 2 = 116

211 = 210 \* 2 = 232

212 = 211 \* 2 = 464 = (464 XOR 285) = 205

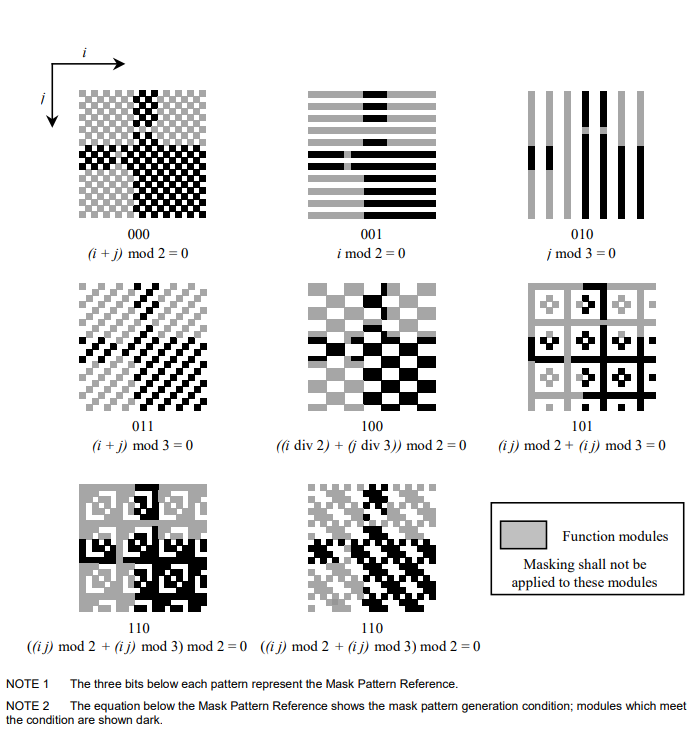
Using this process, all numbers can be represented by powers of 2. Alpha notation that will be mentioned later is this exact same process except where α = 2 i.e. α5 = 25 = 3

## Research – Masking

If a module has been masked it means that if the module was white, it has been changed to a black module and if it was a black module, it has been changed to white. Essentially, masking a module means toggling the current color of the module. The QR Standard Document details 8 different masking patterns, each of which refers to a Mask Pattern Generation Condition which take the form of an equation. If the equation for each mask condition returns true, toggle the color of the current module. In the table below each mask generation condition is detailed.

|  |  |
| --- | --- |
| **Mask Pattern Reference** | **Mask Pattern Generation Condition** |
| 000 | (row + column) mod 2 == 0 |
| 001 | (row) mod 2 == 0 |
| 010 | (column) mod 3 == 0 |
| 011 | (row + column) mod 3 == 0 |
| 100 | (floor(row / 2) + floor(column / 3) ) mod 2 == 0 |
| 101 | ((row \* column) mod 2) + ((row \* column) mod 3) == 0 |
| 110 | ( ((row \* column) mod 2) + ((row \* column) mod 3) ) mod 2 == 0 |
| 111 | ( ((row + column) mod 2) + ((row \* column) mod 3) ) mod 2 == 0 |

Below is a visual representation of each mask pattern; taken from the ISO QR Standard Document.



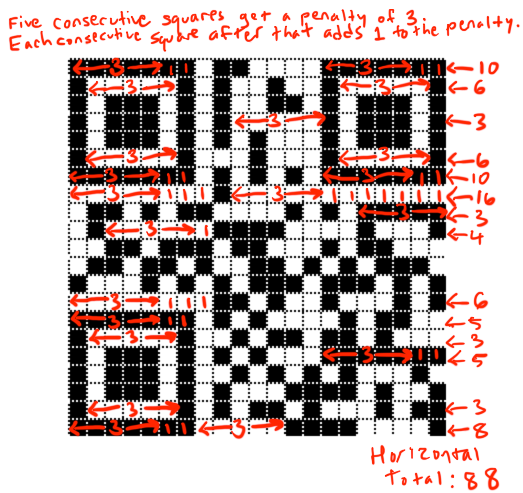
To maximize reliable QR Code reading by a scanner, the White/Black modules need to be displayed in a well-balanced manner and there should not be large areas of either White or Black modules as two concurrent modules of the same color may be mistaken for a single module of that color when scanning takes place. Masking aims to achieve these two conditions and each mask pattern is evaluated and scored based on how well it achieves these two things. Undesirable features of a mask pattern will incur penalty points under evaluation. Once evaluation has been completed for each mask pattern, the mask pattern which incurred the lowest penalty score will be chosen as the most appropriate for that QR Code and finally applied. Each mask pattern is evaluated subject to 4 conditions, which are detailed in the next section.

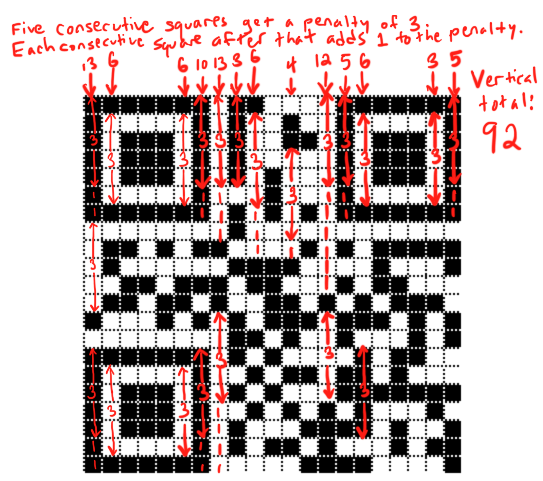
## Research – Masking Evaluation Conditions

Condition 1

Condition 1 searches for consecutive modules of the same color. As this is an undesirable feature, the masking pattern is penalized if consecutive runs of the same color are found. Each row is checked one-by-one and if five consecutive modules of the same color are found, a penalty score of 3 is added for the current mask pattern. If, after the initial five consecutive modules, there are more modules of the same color, a penalty score of 1 is added for every extra module of the same color. Once each row has been checked subject to these conditions, every column is checked subject to the same conditions.

The image below shows an illustration of condition 1 and the penalty score incurred for Mask Pattern 0 with the input text “HELLO WORLD”.



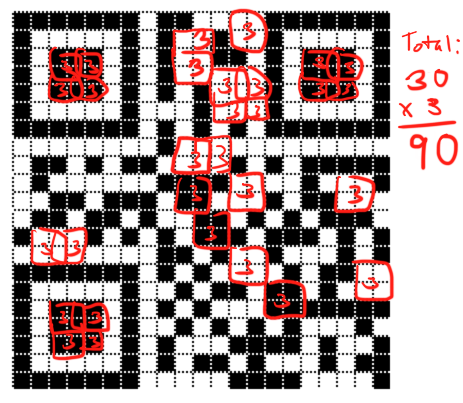


The total incurred penalty score in this case for Mask Pattern 1 would be 88 + 92 = 180.

Condition 2

Condition 2 searches for large areas of the same color, every time one is detected, a penalty score of 3 is incurred. More specifically, every time an area of the same color that is at least 2 x 2 modules is detected, the penalty is incurred. The ISO QR Standard Document states that for an area of the same color that has dimensions of m by n, the incurred penalty score is 3 \* (m – 1) \* (n – 1).

The image below shows an illustration of condition 2 and the penalty score incurred for Mask Pattern 0 with the input text “HELLO WORLD”.



The total incurred penalty score in this case for Mask Pattern 1 would be 90 as thirty 2 x 2 areas of the same color were detected.

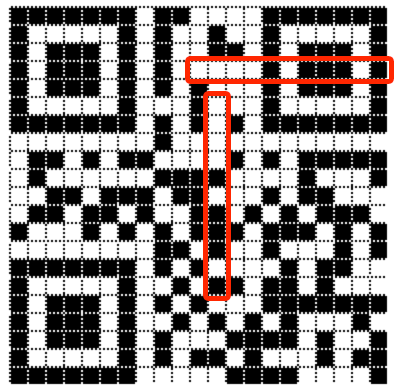
Condition 3

Condition 3 searches for consecutive modules that could be mistaken by the scanner for Finder Patterns and such would increase overall scanning time. For this reason, condition 3 searches for consecutive modules in the pattern black-white-black-black-black-white-black, with 4 white modules on either side. Essentially, either of the two patterns illustrated below are searched for.



Any time either of these two patterns are detected, a penalty score of 40 is incurred. The two patterns are searched for in both the rows and columns of the QR Code.

The image below shows an illustration of condition 3 and the penalty score incurred for Mask Pattern 0 with the input text “HELLO WORLD”.



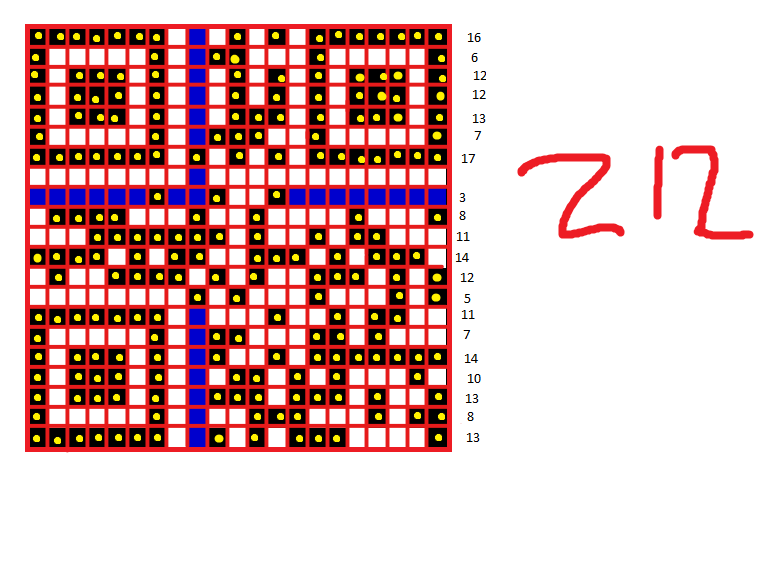
The total incurred penalty score in this case for Mask Pattern 1 would be 80 as 2 occurrences of the two patterns were detected were detected.

Condition 4

Condition 4 is based on the ratio of black modules to white modules. The following steps are followed:

1. Count the total number of modules in the matrix, in our case we’re using Version 1 throughout the project which measures 21 x 21 modules. This gives us a total of 441 modules.
2. Iterate through the entire matrix and count the total number of black modules.
3. Calculate the percentage of black modules in the matrix. This is done by dividing the number of black modules by the total number of modules (in our case always 441), and multiplying the result by 100.
4. Determine the previous and next multiples of 5. E.g. If the percentage of dark modules was 37, the previous multiple of 5 would be 35 and the next multiple of 5 would be 40.
5. Subtract 50 from the previous multiple and the next multiple and record the magnitude of these results. E.g. If the previous result was 35, the result would be the magnitude of (35 – 50), which would be 15.
6. Divide each of the two results by 5.
7. Take the smaller of these two values and multiply it by 10. This is the penalty score for Condition 4

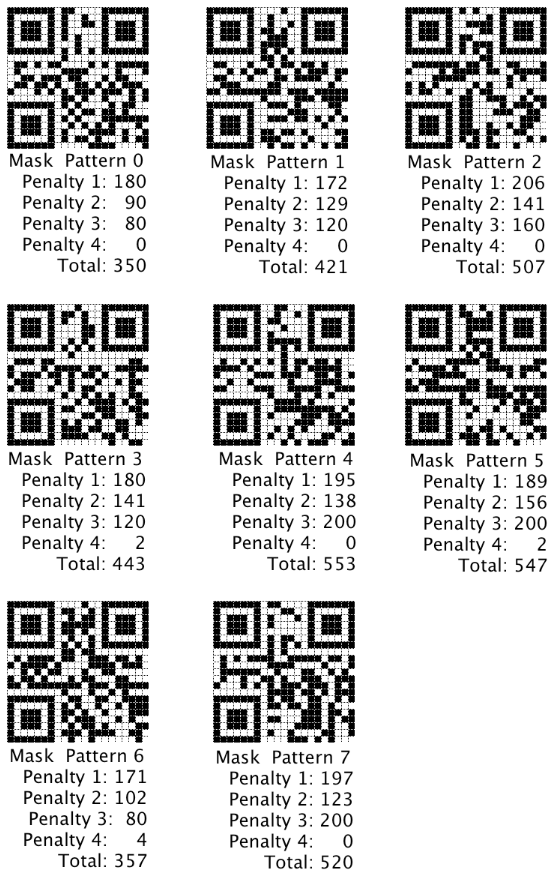
The image below shows an illustration of condition 3 and the penalty score incurred for Mask Pattern 0 is detailed below for the input text “HELLO WORLD”.



1. The total number of modules is 441
2. The total number of black modules is 212
3. The percentage of black modules is 48%
4. The previous multiple of 5 is 45. The next multiple of 5 is 50
5. The magnitude of (45 – 50) is 5. The magnitude of (50 – 50) is 0
6. 5 / 5 = 1. 0 / 5 = 0
7. The smaller of these is 0, therefore multiply 0 by 10. This means that Mask Pattern 0 would not incur a penalty score for condition 4

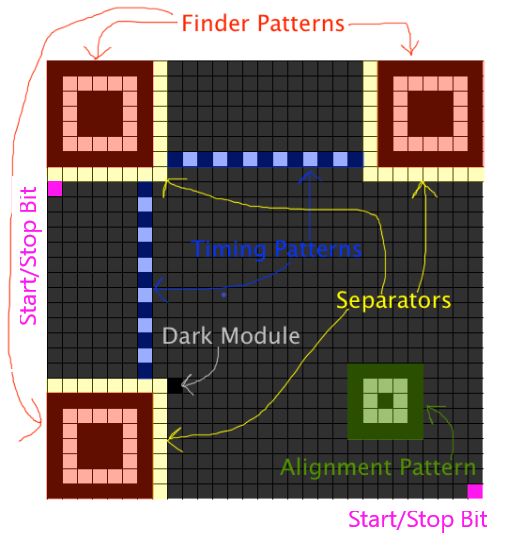
Final Condition Evaluation

The following image shows the input text “HELLO WORLD” encoded in all 8 mask patterns, with the incurred penalty score for each evaluation condition.



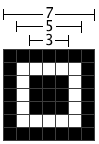
It is clear that in this case we would choose Mask Pattern 0 as it incurred the lowest penalty score.

## Research – Function Patterns

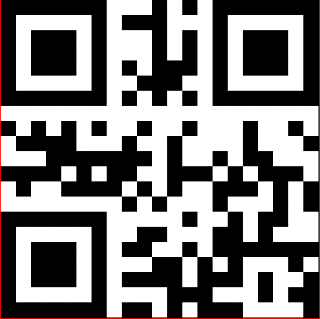
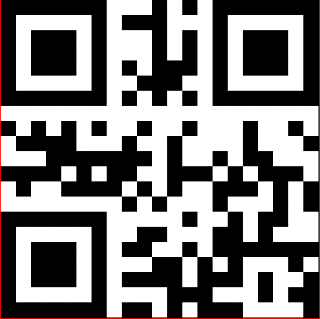
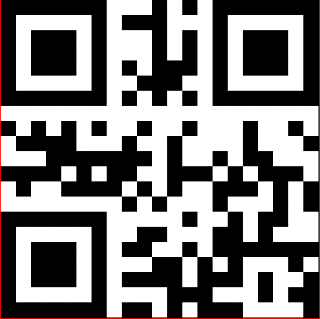
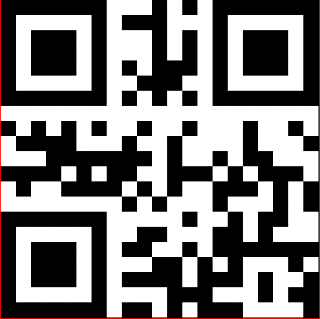


All QR Codes have standard features, mostly included to aid with the scanning process, known as Function Patterns. These are all detailed below, and the physical position and shape of these Function Patterns are highlighted in the image above, which is a Version 2 QR Code.

**Finder Patterns** – Finder Patterns are a black 7 x 7 module square which contains an inner white square of 5 x 5 modules, which in turn contains a black square of 3 x 3 modules. Regardless of the Version used, there are always 3 Finder Patterns of the same size specified. They are also always placed in the bottom left, top right and top left corners of the symbol. They are placed this was such that there is only one possible orientation of the QR Code that could have Finder Patterns in those exact positions.



Finder Pattern with module dimensions



Rotated 270o Clockwise

Rotated 180o Clockwise

Rotated 90o Clockwise

Expected Orientation

Above you can see the same data “HELLO WORLD” encoded in a QR Code. The first symbol is the correct orientation of the symbol, with the following three being rotated in increments of 90 degrees respectively. Without the Finder Patterns, the red Modules indicate where the scanner would start and stop detecting data. Clearly, without Finder Patterns the data that the scanner detects would be drastically different depending on the orientation, even though all 4 of the above QR Codes were fed the same input data. This could lead to the scanner reporting incorrect data. Therefore, the scanner continues orienting the QR Code until it is in an orientation with the Finder Patterns in their respective corners. Only then will the scanner begin detecting data.

**Separators** – These are a one module wide border which surround each of the three finder patterns. They are completely white and improve the recognisablilty of the Finder Patterns as they separate them from the actual data.

**Timing Patterns** – These consist of one vertical and one horizontal line, each of alternating black and white modules. Under scanning, these allow the scanner to accurately measure the width of a single module, aiding with the rest of the scanning process as the chance of mistaking two consecutive modules of the same color as a single module of that color will decrease.

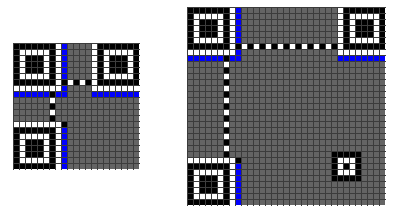
**Alignment Patterns** – Although I don’t have any alignment patterns in my project as they are only required in QR Codes of Version 2 or above, it is still important to know what they are. Essentially, they are just a smaller version of Finder Patterns, placed in specific positions throughout the QR Code. The positions of the Alignment Patterns are dictated by the Version. Similar to Finder Patterns, they consist of a 5 x 5 square of black modules, with an inner 3 x 3 square of white modules with a single black module at it’s center.



Alignment Pattern with module dimensions

**Reserved Areas** – Reserved Areas are found:

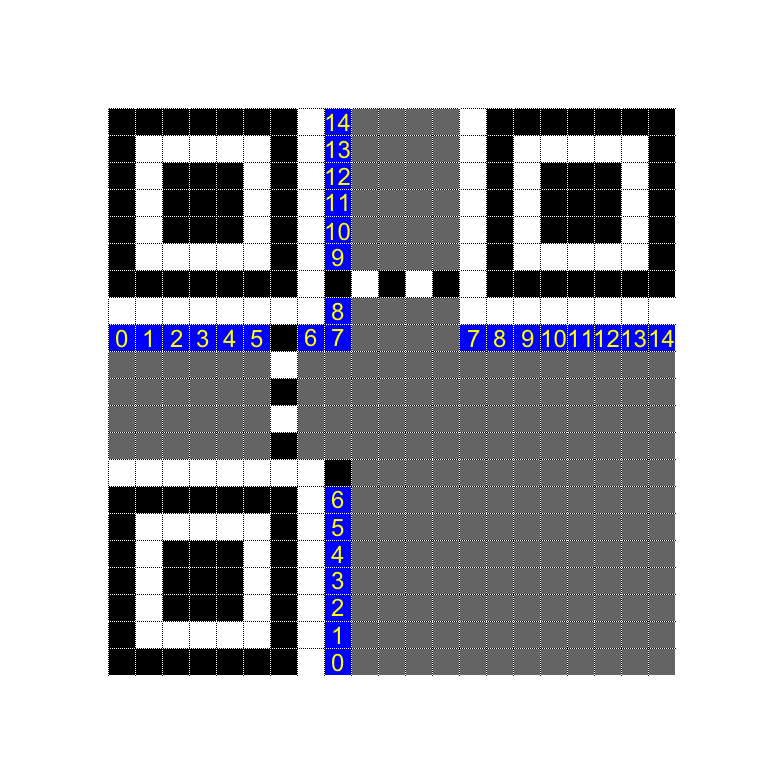
* Near the top left Finder Pattern as a one-module wide strip, directly below and directly to the right of the separator
* Near the top right Finder Pattern as a one-module wide strip, directly below the separator
* Near the bottom left Finder Pattern as a one-module wide strip, directly to the right of the separator



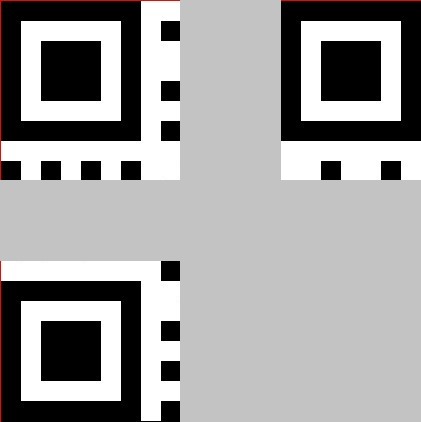
The Reserved Areas are highlighted in blue in the above diagram. As you can see, similar to the Finder Patterns, Reserved Areas are in the same place on each QR Code, regardless of the Version used. They are used to store the Error Correction Level used to encode the data and the Mask chosen.

The Reserved Areas are the last section of the QR Code to be filled in. There are 8 different possible data strings that could be encoded in the Reserved Areas, one for each combination of the Mask and Error Correction Level which are detailed in the table below. Of course if we were using more than one Error Correction Level, or allowing the user to choose an Error Correction Level, there would be more possible data strings for the reserved areas.

|  |  |  |
| --- | --- | --- |
| **Error Correction Level** | **Mask Pattern Used** | **Format Information String** |
| M | 0 | 101010000010010 |
| M | 1 | 101000100100101 |
| M | 2 | 101111001111100 |
| M | 3 | 101101101001011 |
| M | 4 | 100010111111001 |
| M | 5 | 100000011001110 |
| M | 6 | 100111110010111 |
| M | 7 | 100101010100000 |



The format information string is placed in the QR Code according to the diagram on the left. The position 0, refers to the most significant bit of the format information string i.e. the left most bit.

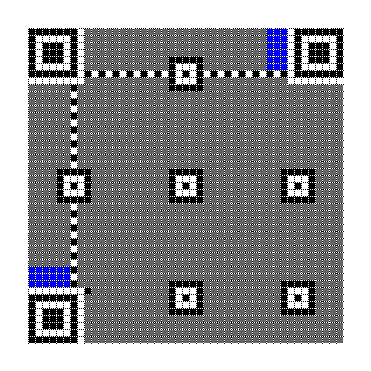


As an example, here are the reserved areas for the string HELLO WORLD. Masking Pattern 0 was used and if we consult the above table, it says the format and information string should be

101010000010010

Therefore the format and information string is mapped to the Reserved Areas in the way detailed in the above diagram.

**Version Information** – For QR Codes that are Version 7 or larger, you must include an 8-bit Version Information string in the bottom left and top right corners of the QR Code, directly adjacent to the Finder Patterns. As my project only focuses on Version 1 QR Codes, this was not a step I had to do however if I increased the capacity in the future it is something I would have to consider. The diagram below shows where the Version Information would be placed, highlighted in blue.



## System Objectives / Specification

My conversations with my supervisor have led me to the following objectives for my program:

1. Allow the user to enter text that they wish to be converted into a QR Code
2. Perform validation on the text that the user enters. This involves ensuring that the user entered text I.e. check that the text field hasn’t been left empty, ensuring that the text entered only contains characters that fall under the ‘Alphanumeric’ category and ensuring that the text entered doesn’t exceed the 20 characters that are allowed for this Version and Error Correction Level
3. Convert the user-entered data into a list of 8-bit Data Codewords
4. Employ Reed-Solomon error correction methods, utilizing a polynomial division algorithm, to generate Error Correction Codewords. The Error Correction Codewords should be generated based on the ‘M’ level of error correction.
5. Structure a final bit stream, combining the generated Data and Error Correction codewords
6. Graphically draw a correctly masked QR Code which can easily and quickly be scanned by a scanner (mobile phone). Each encoded pixel of the QR Code will represent either a binary 1 or 0.
7. Ensure that a section of the QR Code could be covered (dictated by the error correction level used) and it could still be scanned, and the inputted data could be determined

## Critical Path

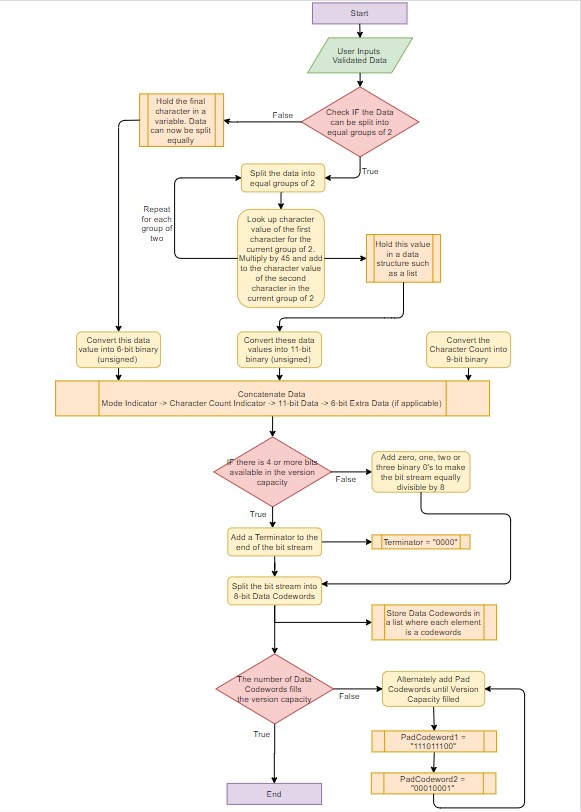
1. Validation – the user should have a capacity to enter data and the already outlined validation protocols should be processed. If the user doesn’t pass this validation stage, they should be able to re-enter the data. If the user does pass this validation stage, then they should move onto the rest of the program
2. Data Encoding – convert the data into a binary bit stream for the alphanumeric mode method. Split this binary data stream into 8-bit data codewords. Pad codewords will be added to ensure the data completely fills the version capacity (128-bits in this case).
3. Error Correction Generation – apply Reed-Solomon error correction algorithms to generate the required number of error correction codewords. The exact generation algorithms will be discussed in the Design section.
4. Structuring Final Data Stream – the data codewords and error correction codewords should be interwoven as described in the ISO QR Standard document.
5. Constructing the QR symbol – place the final codeword message into a matrix of appropriate size. Binary 1’s will be represented as black modules in the matrix and 0’s will be represented as white modules. These will be placed along with Finder Patterns. These can be seen in the above version diagrams as the standard patterned squares in the upper-right, upper-left and lower-left corners of the QR code. These are used to allow the scanner to allow unambiguous scanning in terms of the orientation and location of the symbol.
6. Masking – Apply a masking pattern to the generated QR code which should optimize the dark/light module balance and minimizes the occurrences of undesirable patterns such as large areas of completely dark/light modules. This involves separately applying 8 different masking patterns and scoring them subject to defined conditions thus calculating a score for each one. This allows the optimum mask to be selected and applied.
7. Format/Version Information – Data bits referring to the Format information (error correction level used, mask pattern selected, data encoding mode etc) and Version information (physical size of the matrix) should be added to the QR Code.
8. Display – The QR Code should then be displayed to the user in such a way that the user is able to quickly and effectively scan it to extract all encoded information.

# Design

## Table of Key Variables

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Public Class** CurrentModule | | | | | | | |
| **Function/Sub Name** | | | **Description** | | | | |
| IsBlack | | | Returns a Boolean value If the current coordinate of the symbol matrix has been encoded as black | | | | |
| IsWhite | | | Returns a Boolean value If the current coordinate of the symbol matrix has been encoded as white. It is true that if the function listed above (IsBlack) returns a “False” Boolean value (indicating that the current coordinate isn’t black); it means that the current coordinate is inherently white. However it was helpful during the development process to have two separate functions for IsBlack and IsWhite as it allowed me to easily distinguish between the two | | | | |
| PaintModule | | | This function takes coordinates of a module and a brush color as parameters. It then paints the corresponding module in the outputted symbol, the color specified by the user. I found that I was repeatedly writing very similar code to paint modules, thus it made sense to make a separate class function to simplify the process | | | | |
|  | **Sub** Button1\_Click | | |  | | | |
| **Variable Name** | **Variable Type** | | | **Description** | | | |
| input\_text | String | | | Holds the initial text entered by the user | | | |
| length\_input | Integer | | | Holds the integer value of the length of the user’s entered text | | | |
| length\_stream | Integer | | | Holds the length of the encoded binary data stream | | | |
| encoding\_list | String | | | Holds the encoding table for Alphanumeric Mode (Table 5 in the QR Standard document) | | | |
| split\_list | List(Of String) | | | Holds each group of two characters from the user’s entered text. Each group of two characters is an independent item in the list, ordered as the user entered the text. E.g. if the user entered ‘Hello’, ‘He’ would be the first item in split\_list() | | | |
| extra | String | | | If the user’s entered text doesn’t exactly split into groups of two, this variable would hold the final character, i.e. the extra character | | | |
| validation\_return | Boolean = Validation(input\_text, length\_input, encoding\_list) | | | Calls the subroutine ‘Validation’ which performs a simple validation on the user’s entered text. This sub returns a Boolean value (True or False) which is stored in this variable | | | |
| binary\_string | String | | | Holds the input string data after it has been encoded into binary | | | |
| pad\_codeword\_1 | String = "11101100" | | | If the binary string doesn’t fill the capacity of the QR code version, pad codewords 1 and 2 are added alternately until the max capacity (128 bits in this case) is filled. | | | |
| pad\_codeword\_2 | String = "00010001" | | | See ‘pad\_codeword\_1’ | | | |
|  | | **Public Function** Validation | | |  | | |
| **Variable Name** | | **Variable Type** | | | **Description** | | |
| test | | Boolean | | | If the user entered text passes various validation tests i.e. checking the length of the text is below 20 characters (as the capacity of this QR version allows) and checking the entered text field hasn’t been left empty, then test will hold the Boolean value ‘True’ | | |
|  | | **Public Function** Conversion | | |  | | |
| **Variable Name** | | **Variable Type** | | | **Description** | | |
| mode\_indicator | | String = ‘0010’ | | | The mode of data when encoding QR codes is used to indicate what type of data the current run is. Examples of different modes include numerical (integers 0-9), alphanumeric (integers 0-9, letters A-Z and some symbols. For this investigation, as it’s only a proof of concept, I thought it only necessary to encode data in the alphanumeric mode which is why the mode indicator can be set and never changed in the program | | |
| current | | String | | | Holds the current character as the user-entered string is iterated through. Initially, we will of course start at the first character in the user-entered string and cycle through the list character by character, holding the current character in this variable | | |
| position | | Integer | | | Finds the index position of the current character (stored in the above variable ‘current’) in the alphanumeric list (see the variable ‘encoding\_list’ above or Table 5 in the QR Standard document) | | |
| binary\_list | | List(Of String) | | | Once the user data has been calculated as stated above in the design section and the value calculated, it is converted into binary and stored chronologically in this list | | |
|  | |  | | |  | | |
|  | | | | | | | |
| **Public Function** CodewordSplit | | | | | | | |
| **Variable Name** | | **Variable Type** | | | | | **Description** |
| codewords\_string | | String | | | After the data is encoded into binary, that string (‘binary\_string’) is split into equal groups of 8. | | |
| codewords\_list | | List(Of String) | | | Splits ‘codewords\_string’ into a list, with each element equal to a byte of the binary string | | |
| **Public Function** PlacingIntoMatrix | | | | | | | |
| **Variable Name** | | **Variable Type** | | | | | **Description** |
| Matrix | | 2D Array of type Integer | | | | | Holds the 2-D array which stores the data of each of the 441 pixels (21 x 21). There are 10 different things each pixel can be:  0 = White  1 = Black  2 = Dark Module  3 = Reserved Area  4 = Blank  5 = Timing Pattern ("White")  6 = Timing Pattern ("Black")  7 = Finder Pattern ("White")  8 = Finder Pattern ("Black")  9 = Separator |
| **Public Function** DisplayingMatrix | | | | | | | |
| **Variable Name** | | **Variable Type** | | | | | **Description** |
| PictureBox1 | | Bitmap Image | | | | | Holds the bitmap image where the final QR Code grid, as well as each pixel and it’s corresponding color, is drawn |
| Drawing | | Graphics | | | | | Graphics.FormImage  Allows easy references to drawing tools, used in drawing the QR grid and each pixel |
| Height | | Integer | | | | | Gets the height of the picture box |
| Partition | | Integer | | | | | Used to calculate the width/height of each pixel within the QR Symbol |
| **Public Function** ErrorCorrectionGeneration | | | | | | | |
| **Variable Name** | | **Variable Type** | | | | | **Description** |
| generator\_polynomial\_alpha\_exponents | | List (Of Integer) | | | | | Holds the coefficients for the generator polynomial. Taken from Annex A in the QR standard document, Table 1.A for 10 error correction codewords |
| first\_term\_alpha\_notation | | String | | | | | Gets the current term and looks up and holds the alpha (power of 2) equivalent from the generated anti-log table |
| **Public Function** Masking | | | | | | | |
| **Variable Name** | **Variable Type** | | | | | **Description** | |
| score\_list | List (Of Integer) | | | | | List which holds the scores for each masking pattern, subject to the 4 specified conditions. The smallest of these scores dictates the optimal masking pattern | |
| masked\_matrix | 2D Array of type Integer | | | | | Once the optimal masking pattern has been selected, it is finally applied and the resulted data is stored in this array and returned | |
| **Public Function** EvaluatingMask | | | | | | | |
| **Variable Name** | **Variable Type** | | | | | **Description** | |
| condition\_1\_score | Integer | | | | | For the current mask pattern, this holds the score generated subjected to condition 1. | |
| condition\_2\_score | Integer | | | | | For the current mask pattern, this holds the score generated subjected to condition 2. | |
| condition\_3\_score | Integer | | | | | For the current mask pattern, this holds the score generated subjected to condition 3 | |
| condition\_4\_score | Integer | | | | | For the current mask pattern, this holds the score generated subjected to condition 4 | |
| dark\_module\_count | Integer | | | | | For Condition 4, this variable holds the number of black modules in the entire matrix | |
| ratio\_dark\_module | Integer | | | | | Calculates the ratio of black modules in the matrix and golds it here. | |

## Key Function – Data Codeword Generation



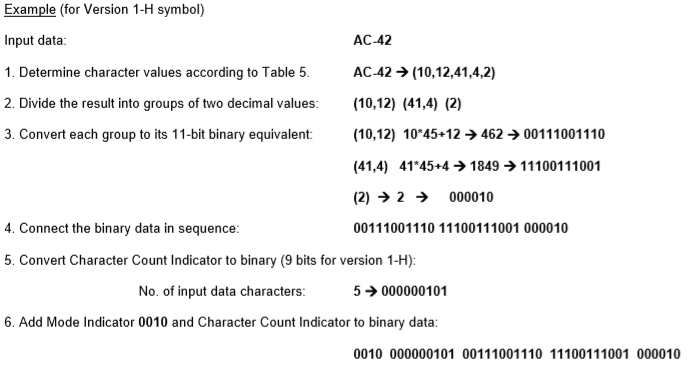
One of the first steps of this project was converting the text the user has entered into a form we can actually display on a QR Code. The QR Code itself is split up into a grid of pixels, with the total number of pixels being dependent on the version selected. As stated previously, we are encoding a Version 1 symbol which is 21 x 21 pixels. This results in a total of 441 pixels that need to be filled with either a white or black square. A binary 0 here represents a white square and a binary 1 represents a black square.

The first step in generating the data codewords was to split the input characters into groups of 2, the first step therefore was to check if the data could be split equally into groups of 2, i.e. if the length of the input text was equally divisible by 2. If the data wasn’t able to be split equally into groups of 2, remove the last character of the input text and hold it in a variable. The input text should now be able to be split into groups of 2, so split the text sequentially into groups of 2 and store them in a list, with each group of 2 occupying its own list position. For each group of 2, look up the character value of the first character and multiply it by 45. Add this to the character value of the second character. Note when I mention character value I am referring to the value listed in the table in the section “Research after Conversation (1) – Data Encoding”. For each group of two there should now be a total decimal value calculated. Convert each of these values into 11-bit unsigned binary. If there was an extra character at the start of this process, look up its character value and convert this into 6-bit unsigned binary. Finally, convert the length of the input text into 9-bit unsigned binary. Note that the mode indicator refers to which mode you are encoding the data into, which in this case will be alphanumeric for all conversions which has the binary value 0010.

The data should then be concatenated in the following way:

MODE INDICATOR 🡪 INPUT TEXT LENGTH (9-bit unsigned binary) 🡪 DATA (11-bit unsigned binary) 🡪 EXTRA (6-bit unsigned binary)

Next you need to add a Terminator to the end of the bit stream we just created which has binary value 0000. This is to signal the end of the encoded data. Split the binary bit stream into codewords, each of 8 bits long and store the codewords in a list with each codeword being its own element in the list. However the bit stream must completely fill the designated storage for this QR Code version. Therefore add the Pad Codewords 11101100 and 00010001 alternately until the data capacity is filled. Below is a worked example of the conversion process taken from the QR Standard document.



## Key Function – Error Correction Codeword Generation

Pseudocode

FOR A 🡨 0 TO 15

Function ErrorCorrectionCodewordGeneration (Message\_Polynomial\_Coefficients)

FOR B 🡨 0 TO LEN(Generator\_Polynomial\_Exponents) – 1

IF Generator\_Polynomial\_Exponents(B) + FirstTermExponent >= 255 THEN

Generator\_Polynomial\_Exponents(B) 🡨 Generator\_Polynomial\_Exponents(B) + FirstTermExponent MOD 255

ELSE

Generator\_Polynomial\_Exponents(B) 🡨

Generator\_Polynomial\_Exponents(B) + FirstTermExponent

END IF

END FOR

FOR C 🡨 0 TO LEN(Generator\_Polynomial\_Exponents) – 1

Message\_Polynomial\_Coefficients(C) 🡨 Message\_Polynomial\_Coeffients(C) XOR Generator\_Polynomial\_Exponents(C)

ENDFOR

Message\_Polynomial\_Coefficients 🡨 Message\_Polynomial\_Coefficients.Remove(0)

RETURN Message\_Polynomial\_Coefficients

EndFunction

Message\_Polynomial\_Coefficients🡨ErrorCorrectionCodewordGeneration (Message\_Polynomial\_Coefficients)

ENDFOR

Pseudocode Explanation

1. Multiply the message polynomial by x10 to ensure that the exponent of the lead term doesn’t become too small during the division process.
2. The lead term of the generator polynomial should be the same as the lead term of the message polynomial so multiply the generator polynomial by x15.
3. Multiply the generator polynomial by the lead term of the message polynomial. Due to the rules of indices, add the exponents together. If the result of this addition is greater than 255, perform modulo 255.
4. XOR the resulting polynomial with the message polynomial.
5. Discard the lead term as this should now be 0.
6. Repeat this process a total of 16 times.
7. Once repetitions have been completed, the coefficients of the remaining polynomial are the error correction codewords to use.

I found that throughout the development of this project, the most complex step was generating the error correction codewords using the Reed-Solomon error correction generation process. The first step was to import the Data Codewords which we generated in the previous section and convert each of them denary. As we added Pad Codewords to fill the version capacity, there should be 16 Data Codewords total that need to be converted into denary. As an example, if the user enters HELLO WORLD, if we follow the process outlines in the previous section we should generate the Data Codewords:

00100000 01011011 00001011 01111000 11010001 01110010 11011100 01001101 01000011 01000000 11101100 00010001 11101100 00010001 11101100 00010001

Converted into denary these codewords produce the decimal numbers:

32-91-11-120-209-114-220-77-67-64-236-17-236-17-236-17

Each of these decimal values corresponds to a coefficient of the **message polynomial**. Note a polynomial is a mathematical function that takes the form axn + bxm… where a and b are coefficients and n and m are exponents. The message polynomial in full for the input of HELLO WORLD in the Version 1-M format is:

32x15+91x14+11x13+120x12+209x11+114x10+220x9+77x8+67x7+64x6+236x5+17x4+236x3+17x2+236x+17

Next we need to create the **generator polynomial**. Whilst it is possible to computationally generate this polynomial, it was not necessary in this case as the polynomial itself is dependent on the Version Error Correction Level being used. As these two factors never change in this project, it was easier to just copy the generator polynomial from the QR Code Tutorial on Thonky.com where all generator polynomials for every combination of Version and Error Correction Level can be found.

## Key Function – Constructing Final Bit Stream

As we are using a small Version 1 QR Code to construct the final bit stream we simply must place the generated Error Correction Codewords after the generated Data Codewords to construct the final bit stream. If in the future we were to encode data into a bigger QR Code we would have had to break the generated Error Correction Codewords into blocks and interleave each block with each Data Codeword.

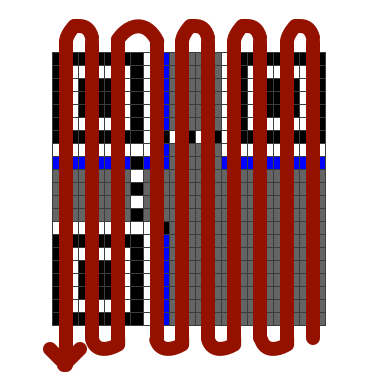
## Data Structure – Matrix

I utilized the built-in 2-Dimensional Array feature in Visual Basic to create a matrix data structure that measured 21 x 21 elements, each element referring to a module of the QR Code. It held integer values, from 0-9, with each single digit integer relating to a different module type. The different type of modules I referred to are detailed in the table below, along with the 0-9 integer value I assigned to them.

|  |  |
| --- | --- |
| **Integer Value Chosen** | **Type of Module** |
| 0 | **White** – This refers to any White data or error correction bits that need to be represented on the QR Code |
| 1 | **Black** - This refers to any Black data or error correction bits that need to be represented on the QR Code |
| 2 | **Dark Module** – This refers to the single Black module that is always placed near the bottom left Finder Pattern. It is always located at the position  ([(4 \* V) + 9], 8) where V is the Version |
| 3 | **Reserved Area** – This refers to the areas that are reserved for format information. These are overwritten at the end of the program, when the format version is added to the QR Code |
| 4 | **Blank** – This refers to the areas that don’t yet have any data written to them |
| 5 | **Timing Pattern White –** This refers to the Timing Pattern White modules |
| 6 | **Timing Pattern Black –** This refers to the Timing Pattern Black modules |
| 7 | **Finder Pattern White –** This refers to the modules in the Finder Pattern which are White |
| 8 | **Finder Pattern Black -** This refers to the modules in the Finder Pattern which are Black |
| 9 | **Separator –** These refer to the White Separator modules |

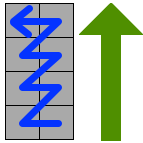
## Key Function – Data Placement in Matrix

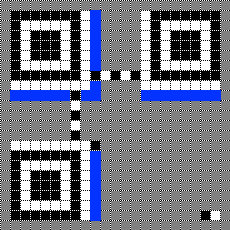
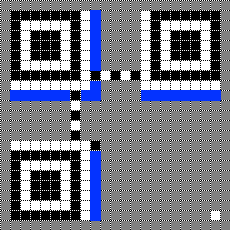
Once all the Function Patterns have been added, it is time to place the data and error correction codewords into the QR Code symbol. The data bits (concatenation of the Error Correction and Data Codewords generated previously), are firstly placed starting in the bottom right of the matrix and proceeding upwards in a two-module wide column. When the column reaches the top of the symbol, the next 2-module wide column starts directly to the left of the previous column and continues downward. If a Function Pattern or Reserved Area is reached, the data bit is placed in the next unused module. When the Vertical Timing Pattern is reached, the next column starts directly to the left of it. This process is demonstrated in the image below.



There are essentially two modes to consider when placing the data bits in the matrix; Upwards and Downwards placement.

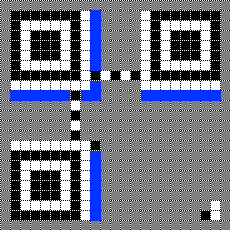
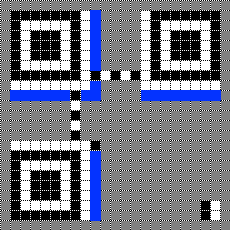
Upwards Placement

This image shows the order in which to place data bits onto the matrix when the column is travelling upwards. The first data bit will be placed in the bottom-right module. The second data bit will be placed in the bottom-left module, adjacent to the first bit. The third data bit will be placed in the module that is diagonally adjacent to the right of the second bit. This process is repeated until a Reserved Area or Function Pattern or the edge of the matrix is reached. The process for upwards placement is illustrated in the sequence of diagrams below.



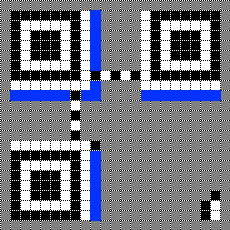
First data bit added

Second data bit added



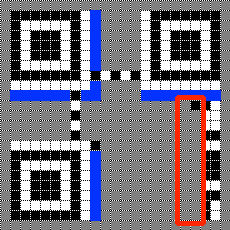
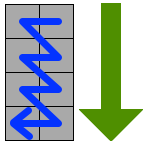
Third data bit added

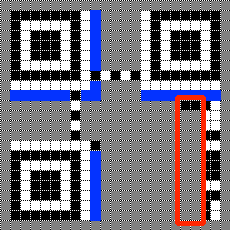
Fourth data bit added



Fifth data bit added

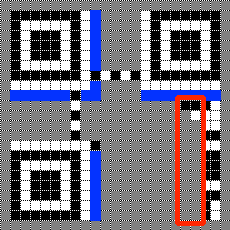
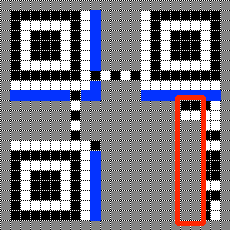
Downwards Placement

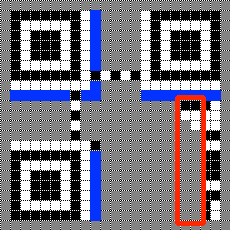
This image shows the order in which to place data bits onto the matrix when the column is travelling downwards. The first data bit will be placed in the top-right module. The second data bit will be placed in the top-left module, adjacent to the first bit. The third data bit will be placed in the module that is diagonally adjacent to the right of the second bit. This process is repeated until a Reserved Area or Function Pattern or the edge of the matrix is reached. The process for downwards placement is illustrated in the sequence of diagrams below.



Second data bit added

First data bit added





Third data bit added

Fourth data bit added

Fifth data bit added

## Key Function – Masking

A detailed description of Masking can be seen in the ‘Analysis’ Section. In this section however you can find pseudocode for each of the 8 Masking Patterns

Mask Pattern 0

1 FOR row 🡨 0 TO 20

2 FOR column 🡨 0 TO 20

3 IF (matrix(row , column) == DataModule(“Black”)) OR (matrix(a , b)) == DataModule(“White”)) THEN

4 IF ((row + column) MOD 2) == 0 THEN

5 IF matrix(row , column) == “Black” THEN

6 matrix(row , column) 🡨 “White”

7 ELSE IF matrix(row , column) == “White” THEN

8 matrix(row , column) 🡨 “Black”

9 END IF

10 END IF

11 END IF

12 END FOR

13 END FOR

Here, lines 1 & 2 are a simple iteration statement which allows us to iterate through every element of the 2-D array ‘matrix’, which is sized as 21 x 21 modules. Line 3 is a selection statement ensuring that we only check modules within the matrix that are actual data modules as masking isn’t applied to any Function Patterns, only data modules.

Line 4 is the Mask Pattern Generation Condition (see the Masking section in Analysis) for Mask Pattern 0. Essentially if the condition:

((row + column) Mod 2 == 0)

is met then toggle the color of the current module we’re looking at. The toggling is taken care of by Lines 5 – 9.

Mask Pattern 1

1 FOR row 🡨 0 TO 20

2 FOR column 🡨 0 TO 20

3 IF (matrix(row , column) == DataModule(“Black”)) OR (matrix(a , b)) == DataModule(“White”)) THEN

4 IF ((row) MOD 2) == 0 THEN

5 IF matrix(row , column) == “Black” THEN

6 matrix(row , column) 🡨 “White”

7 ELSE IF matrix(row , column) == “White” THEN

8 matrix(row , column) 🡨 “Black”

9 END IF

10 END IF

11 END IF

12 END FOR

13 END FOR

Here, lines 1 & 2 are a simple iteration statement which allows us to iterate through every element of the 2-D array ‘matrix’, which is sized as 21 x 21 modules. Line 3 is a selection statement ensuring that we only check modules within the matrix that are actual data modules as masking isn’t applied to any Function Patterns, only data modules.

Line 4 is the Mask Pattern Generation Condition (see the Masking section in Analysis) for Mask Pattern 1. Essentially if the condition:

((row) Mod 2 == 0)

is met then toggle the color of the current module we’re looking at. The toggling is taken care of by Lines 5 – 9.

Mask Pattern 2

1 FOR row 🡨 0 TO 20

2 FOR column 🡨 0 TO 20

3 IF (matrix(row , column) == DataModule(“Black”)) OR (matrix(a , b)) == DataModule(“White”)) THEN

4 IF ((column) MOD 3) == 0 THEN

5 IF matrix(row , column) == “Black” THEN

6 matrix(row , column) 🡨 “White”

7 ELSE IF matrix(row , column) == “White” THEN

8 matrix(row , column) 🡨 “Black”

9 END IF

10 END IF

11 END IF

12 END FOR

13 END FOR

Here, lines 1 & 2 are a simple iteration statement which allows us to iterate through every element of the 2-D array ‘matrix’, which is sized as 21 x 21 modules. Line 3 is a selection statement ensuring that we only check modules within the matrix that are actual data modules as masking isn’t applied to any Function Patterns, only data modules.

Line 4 is the Mask Pattern Generation Condition (see the Masking section in Analysis) for Mask Pattern 2. Essentially if the condition:

((column) Mod 3 == 0)

is met then toggle the color of the current module we’re looking at. The toggling is taken care of by Lines 5 – 9.

Mask Pattern 3

1 FOR row 🡨 0 TO 20

2 FOR column 🡨 0 TO 20

3 IF (matrix(row , column) == DataModule(“Black”)) OR (matrix(a , b)) == DataModule(“White”)) THEN

4 IF ((row + column) MOD 3) == 0 THEN

5 IF matrix(row , column) == “Black” THEN

6 matrix(row , column) 🡨 “White”

7 ELSE IF matrix(row , column) == “White” THEN

8 matrix(row , column) 🡨 “Black”

9 END IF

10 END IF

11 END IF

12 END FOR

13 END FOR

Here, lines 1 & 2 are a simple iteration statement which allows us to iterate through every element of the 2-D array ‘matrix’, which is sized as 21 x 21 modules. Line 3 is a selection statement ensuring that we only check modules within the matrix that are actual data modules as masking isn’t applied to any Function Patterns, only data modules.

Line 4 is the Mask Pattern Generation Condition (see the Masking section in Analysis) for Mask Pattern 3. Essentially if the condition:

((row + column) Mod 3 == 0)

is met then toggle the color of the current module we’re looking at. The toggling is taken care of by Lines 5 – 9.

Mask Pattern 4

1 FOR row 🡨 0 TO 20

2 FOR column 🡨 0 TO 20

3 IF (matrix(row , column) == DataModule(“Black”)) OR (matrix(a , b)) == DataModule(“White”)) THEN

4 IF ((floor(row/2) + floor(column/3)) MOD 2) == 0 THEN

5 IF matrix(row , column) == “Black” THEN

6 matrix(row , column) 🡨 “White”

7 ELSE IF matrix(row , column) == “White” THEN

8 matrix(row , column) 🡨 “Black”

9 END IF

10 END IF

11 END IF

12 END FOR

13 END FOR

Here, lines 1 & 2 are a simple iteration statement which allows us to iterate through every element of the 2-D array ‘matrix’, which is sized as 21 x 21 modules. Line 3 is a selection statement ensuring that we only check modules within the matrix that are actual data modules as masking isn’t applied to any Function Patterns, only data modules.

Line 4 is the Mask Pattern Generation Condition (see the Masking section in Analysis) for Mask Pattern 4. Essentially if the condition:

((floor(row / 2) + floor(column / 3)) mod 2) == 0

is met then toggle the color of the current module we’re looking at. The toggling is taken care of by Lines 5 – 9. Here the ‘floor’ command means to round the result of the calculation in the bracket down to the nearest integer.

Mask Pattern 5

1 FOR row 🡨 0 TO 20

2 FOR column 🡨 0 TO 20

3 IF (matrix(row , column) == DataModule(“Black”)) OR (matrix(a , b)) == DataModule(“White”)) THEN

4 IF (((row\*column) MOD 2) + ((row\*column) MOD 3) == 0 THEN

5 IF matrix(row , column) == “Black” THEN

6 matrix(row , column) 🡨 “White”

7 ELSE IF matrix(row , column) == “White” THEN

8 matrix(row , column) 🡨 “Black”

9 END IF

10 END IF

11 END IF

12 END FOR

13 END FOR

Here, lines 1 & 2 are a simple iteration statement which allows us to iterate through every element of the 2-D array ‘matrix’, which is sized as 21 x 21 modules. Line 3 is a selection statement ensuring that we only check modules within the matrix that are actual data modules as masking isn’t applied to any Function Patterns, only data modules.

Line 4 is the Mask Pattern Generation Condition (see the Masking section in Analysis) for Mask Pattern 5. Essentially if the condition:

((row \* column) mod 2) + ((row \* column) mod 3) == 0

is met then toggle the color of the current module we’re looking at. The toggling is taken care of by Lines 5 – 9.

Mask Pattern 6

1 FOR row 🡨 0 TO 20

2 FOR column 🡨 0 TO 20

3 IF (matrix(row , column) == DataModule(“Black”)) OR (matrix(a , b)) == DataModule(“White”)) THEN

4 IF (((row\*column) MOD 2) + ((row\*column) MOD 3) MOD 2) == 0 THEN

5 IF matrix(row , column) == “Black” THEN

6 matrix(row , column) 🡨 “White”

7 ELSE IF matrix(row , column) == “White” THEN

8 matrix(row , column) 🡨 “Black”

9 END IF

10 END IF

11 END IF

12 END FOR

13 END FOR

Here, lines 1 & 2 are a simple iteration statement which allows us to iterate through every element of the 2-D array ‘matrix’, which is sized as 21 x 21 modules. Line 3 is a selection statement ensuring that we only check modules within the matrix that are actual data modules as masking isn’t applied to any Function Patterns, only data modules.

Line 4 is the Mask Pattern Generation Condition (see the Masking section in Analysis) for Mask Pattern 6. Essentially if the condition:

(((row \* column) mod 2) + ((row \* column) mod 3)) mod 2­) == 0

is met then toggle the color of the current module we’re looking at. The toggling is taken care of by Lines 5 – 9.

Mask Pattern 7

1 FOR row 🡨 0 TO 20

2 FOR column 🡨 0 TO 20

3 IF (matrix(row , column) == DataModule(“Black”)) OR (matrix(a , b)) == DataModule(“White”)) THEN

4 IF (((row+column) MOD 2) + ((row\*column) MOD 3) MOD 2) == 0 THEN

5 IF matrix(row , column) == “Black” THEN

6 matrix(row , column) 🡨 “White”

7 ELSE IF matrix(row , column) == “White” THEN

8 matrix(row , column) 🡨 “Black”

9 END IF

10 END IF

11 END IF

12 END FOR

13 END FOR

Here, lines 1 & 2 are a simple iteration statement which allows us to iterate through every element of the 2-D array ‘matrix’, which is sized as 21 x 21 modules. Line 3 is a selection statement ensuring that we only check modules within the matrix that are actual data modules as masking isn’t applied to any Function Patterns, only data modules.

Line 4 is the Mask Pattern Generation Condition (see the Masking section in Analysis) for Mask Pattern 7. Essentially if the condition:

(((row + column) mod 2) + ((row \* column) mod 3)) mod 2­) == 0

is met then toggle the color of the current module we’re looking at. The toggling is taken care of by Lines 5 – 9.

## Key Function – Masking Evaluation

This section contains Pseudocode for each of the 4 conditions which each Masking Pattern would be evaluated against. For a detailed description of each condition, see the Analysis section.

Condition 1 – Pseudocode

1 SUB Condition\_1

2 FOR row🡨0 TO 20

3 FOR column🡨 0 TO 16

4 IF matrix(row,column)==”Black” AND

5 matrix(row+1,column) ==”Black” AND

6 matrix(row+2,column) ==”Black” AND

7 matrix(row+3,column) ==”Black” AND

8 matrix(row+4,column) ==”Black” THEN

9 condition\_1\_score🡨condition\_1\_score + 3

10 FOR b🡨row+5 TO 20

11 IF matrix(b,column) ==”Black” THEN

12 condition\_1\_score🡨condition\_1\_score + 1

13 row🡨b

14 ELSE

15 row🡨b

16 BREAK CURRENT FOR

17 ENDIF

18 END FOR

19 IF matrix(row,column) ==”White” AND

20 matrix(row+1,column) ==”White” AND

21 matrix(row+2,column) ==”White” AND

22 matrix(row+3,column) ==”White” AND

23 matrix(row+4,column) ==”White” THEN

24 condition\_1\_score🡨condition\_1\_score + 3

25 FOR b🡨row+5 TO 20

26 IF matrix(b,column) ==” White” THEN

27 condition\_1\_score🡨condition\_1\_score + 1

28 row🡨b

29 ELSE

30 row🡨b

31 BREAK CURRENT FOR

32 ENDIF

33 END FOR

34 ENDIF

35 END FOR

36 END FOR

37 END SUB

Lines 2 & 3 are a simple iteration statement which allows us to iterate through every element of the 2-D array ‘matrix’, which has size 21 x 21 modules, one for each pixel of the QR Code. Lines 4-8 form a selection statement which is looking for 5 consecutive modules that are all of the same color, in this case Black. If 5 consecutive Black modules are found, 3 penalty points are added for this particular Masking Pattern (Line 9). Next we check if, after the 5 consecutive modules are found, there are any further consecutive modules of the same color (Black in this case). If there are, 1 penalty point is added for every additional module of the same color (Line 12). If there aren’t any additional modules of the same color, we break from the loop that is checking for extra modules (break command at Line 16), and continue iterating through the current row for groups of 5 consecutive modules of the same color. If we reach the end of the current row, continue onto the next. These steps are repeated for the 21 rows in the matrix. Lines 19-36 completes the exact same steps are previously described, however this time searching for consecutive modules that are White.

Condition 2 – Pseudocode

1 SUB Condition\_2

2 FOR row🡨0 TO 18 STEP 2 #Exclude Left and Bottom#

3 FOR column🡨1 TO 19

4 IF matrix(row,column) == “Black” AND

5 matrix(row,column+1) == “Black” AND

6 matrix(row+1,column) == “Black” AND

7 matrix(row+1,column+1) == “Black” THEN

8 condition\_2\_score🡨condition\_2\_score + 3

9 END IF

10 IF matrix(row,column) == “White” AND

11 matrix(row,column+1) == “White” AND

12 matrix(row+1,column) == “White” AND

13 matrix(row+1,column+1) == “White” THEN

14 condition\_2\_score🡨condition\_2\_score + 3

15 END IF

16 END FOR

17 END FOR

18 FOR row🡨0 TO 18 STEP 2 #Exclude Right and Bottom#

19 FOR column🡨0 TO 18

20 IF matrix(row,column) == “Black” AND

21 matrix(row,column+1) == “Black” AND

22 matrix(row+1,column) == “Black” AND

23 matrix(row+1,column+1) == “Black” THEN

24 condition\_2\_score🡨condition\_2\_score + 3

25 END IF

26 IF matrix(row,column) == “White” AND

27 matrix(row,column+1) == “White” AND

28 matrix(row+1,column) == “White” AND

29 matrix(row+1,column+1) == “White” THEN

30 condition\_2\_score🡨condition\_2\_score + 3

31 END IF

32 END FOR

33 END FOR

34 FOR row🡨1 TO 19 STEP 2 #Exclude Top and Left#

35 FOR column🡨1 TO 19

36 IF matrix(row,column) == “Black” AND

37 matrix(row,column+1) == “Black” AND

38 matrix(row+1,column) == “Black” AND

39 matrix(row+1,column+1) == “Black” THEN

40 condition\_2\_score🡨condition\_2\_score + 3

41 END IF

42 IF matrix(row,column) == “White” AND

43 matrix(row,column+1) == “White” AND

44 matrix(row+1,column) == “White” AND

45 matrix(row+1,column+1) == “White” THEN

46 condition\_2\_score🡨condition\_2\_score + 3

47 END IF

48 END FOR

49 END FOR

50 FOR row🡨1 TO 19 STEP 2 #Exclude Top and Right#

51 FOR column🡨0 TO 18

52 IF matrix(row,column) == “Black” AND

53 matrix(row,column+1) == “Black” AND

54 matrix(row+1,column) == “Black” AND

55 matrix(row+1,column+1) == “Black” THEN

56 condition\_2\_score🡨condition\_2\_score + 3

57 END IF

58 IF matrix(row,column) == “White” AND

59 matrix(row,column+1) == “White” AND

60 matrix(row+1,column) == “White” AND

61 matrix(row+1,column+1) == “White” THEN

62 condition\_2\_score🡨condition\_2\_score + 3

63 END IF

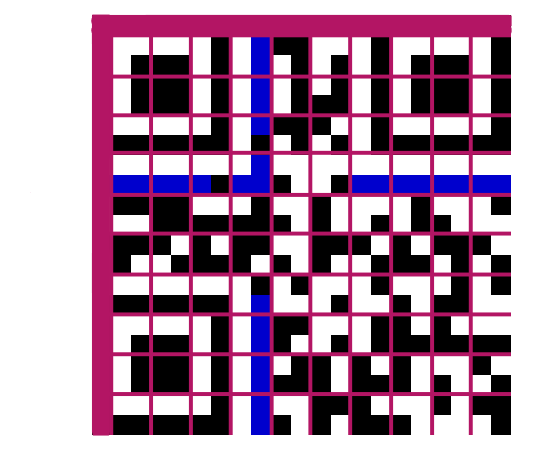
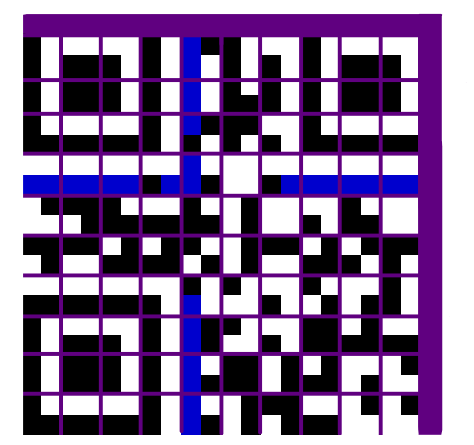
64 END FOR

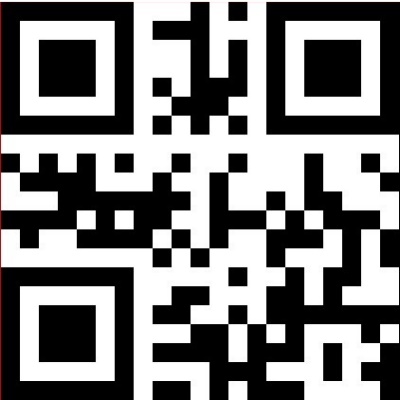
65 END FOR

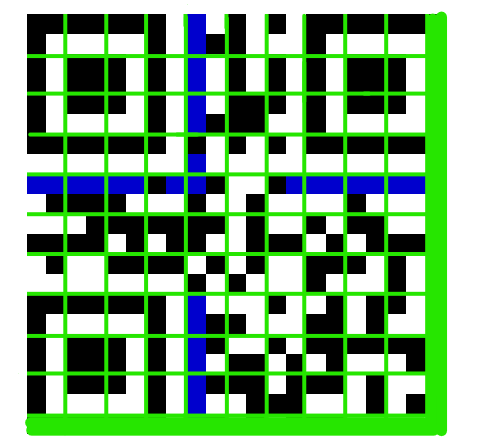
66 ENDSUB

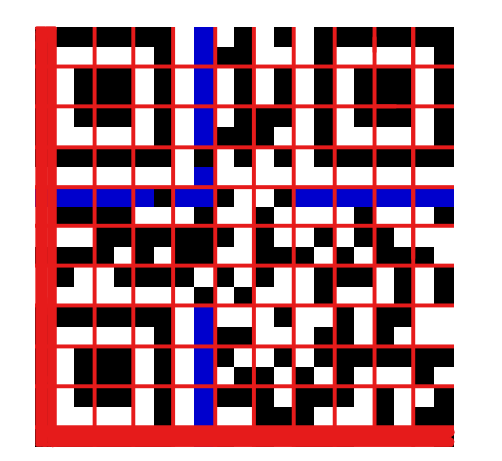
This algorithm is searching the 2-D array ‘matrix’ for 2 x 2 sized areas of the same colour. However our array measures 21 x 21 modules, making iterating through it looking for 2 x 2 squares tricky (as 21 is not divisible equally by 2). Essentially, the process would be much simpler if the matrix we needed to iterate through had a length and width that was divisible by 2, as then an equal number of 2 x 2 squares could fit inside it. To get around this problem I split the original matrix into 4 slightly smaller matrixes; each measuring 20 x 20 modules. Each of the 4 new matrixes were made by taking the original 21 x 21 matrix and eliminating one row and one column, either the top-most row, bottom-most row, left-most column, right-most column. This process is demonstrated in the illustration on the next page.

For each of the 4 new matrixes, we iterate through them searching for 2 x 2 areas of the same color. When such an area is found, 3 penalty points are added. Lines 2-16 take care of this process for the matrix in which we excluded the Left and Bottom lines, colored red on the following page. Lines 18-33 take care of this process for the matrix in which we excluded the Right and Bottom lines, colored green on the following page. Lines 34-49 take care of this process for the matrix in which we excluded the Top and Left lines, colored pink on the following page. Lines 50-65 take care of this process for the matrix in which we excluded the Top and Right lines, colored purple on the following page.









Condition 3 – Pseudocode

1 SUB Condition\_3

2 FOR column🡨0 TO 20

3 FOR row🡨0 TO 10

4 IF matrix(row,column) == “White” AND

5 matrix (row+1,column) == “White” AND

6 matrix(row+2,column) == “White” AND

7 matrix(row+3,column) == “White” AND

8 matrix(row+4,column) == “Black” AND

9 matrix(row+5,column) == “White” AND

10 matrix(row+6,column) == “Black” AND

11 matrix(row+7,column) == “Black” AND

12 matrix(row+8,column) == “Black” AND

13 matrix(row+9,column) == “White” AND

14 matrix(row+10,column) == “Black” THEN

15 condition\_3\_score🡨condition\_3\_score + 40

16 ELSE IF matrix(row,column) == “Black” AND

17 matrix(row+1,column) == “White” AND

18 matrix(row+2,column) == “Black” AND

19 matrix(row+3,column) == “Black” AND

20 matrix(row+4,column) == “Black” AND

21 matrix(row+5,column) == “White” AND

22 matrix(row+6,column) == “Black” AND

23 matrix(row+7,column) == “White” AND

24 matrix(row+8,column) == “White” AND

25 matrix(row+9,column) == “White” AND

26 matrix(row+10,column) == “White” THEN

27 condition\_3\_score🡨condition\_3\_score + 40

28 END IF

29 END FOR

30 END FOR

31 FOR row🡨0 TO 20

32 FOR column🡨0 TO 10

33 IF matrix(row,column) == “White” AND

34 matrix(row,column+1) == “White” AND

35 matrix(row,column+2) == “White” AND

36 matrix(row,column+3) == “White” AND

37 matrix(row,column+4) == “Black” AND

38 matrix(row,column+5) == “White” AND

39 matrix(row,column+6) == “Black” AND

40 matrix(row,column+7) == “Black” AND

41 matrix(row,column+8) == “Black” AND

42 matrix(row,column+9) == “White” AND

43 matrix(row,column+10) == “Black” THEN

44 condition\_3\_score🡨condition\_3\_score + 40

45 ELSE IF matrix(row,column) == “Black” AND

46 matrix(row,column+1) == “White” AND

47 matrix(row,column+2) == “Black” AND

48 matrix(row,column+3) == “Black” AND

49 matrix(row,column+4) == “Black” AND

50 matrix(row,column+5) == “White” AND

51 matrix(row,column+6) == “Black” AND

52 matrix(row,column+7) == “White” AND

53 matrix(row,column+8) == “White” AND

54 matrix(row,column+9) == “White” AND

55 matrix(row,column+10) == “White” THEN

56 condition\_3\_score🡨condition\_3\_score + 40

57 END IF

58 END FOR

59 END FOR

60 END SUB

Here we very simply iterate through every row of the array looking for either of the following patterns:



Every time either of these patterns is found, a penalty score of 40 is incurred. Once this process has finished we check every column of the array for the same patterns. Once again, every time one of these patterns is found, a penalty score of 40 is incurred.

Condition 4 – Pseudocode

1 SUB Condition\_4

2 FOR row🡨0 TO 20

3 FOR column🡨0 TO 20

4 IF matrix(row,column) = “Black” THEN

5 black\_module\_count🡨black\_module\_count + 1

6 ENDIF

7 END FOR

8 END FOR

9 percentage\_black\_module🡨(black\_module\_count / 441) \* 100

10 round\_down🡨(Convert.ToLong(percentage\_black\_module / 5) \* 5)

11 IF (percentage\_black\_module MOD 5) = 0 THEN

12 IF round\_down = percentage\_black\_module THEN

13 round\_down🡨round\_down – 5

14 round\_up🡨round\_down + 10

15 ELSE

16 round\_up🡨round\_down + 5

17 END IF

18 ELSE IF round\_down < percentage\_black\_module THEN

19 round\_up 🡨 round\_down + 5

20 ELSE IF round\_down > percentage\_black\_module THEN

21 round\_up🡨round\_down

22 round\_down🡨round\_down – 5

23 END IF

24 round\_down🡨(magnitude(round\_down – 50) / 5)

25 round\_up🡨(magnitude(round\_up – 50) / 5)

26 IF round\_down > round\_up THEN

27 condition\_4\_score🡨round\_up \* 10

28 ELSE

29 Condition\_4\_score🡨round\_down\*10

30 END IF

31 END SUB

Lines 2-8 here iterate through every element of the 2-D array ‘matrix’ and counts the total number of Black modules for the matrix that the current Masking Pattern would produce. Line 9 then calculates the percentage of Black modules, compared to the total number of elements in the matrix which is always 441. Lines 10-23 calculate the previous and next multiples of 5 for the percentage of Black modules. The previous multiple is stored in ‘round\_down’ and the next multiple is stored in ‘round\_up’. We then subtract 50 from each of these values, find the magnitude of that result and divide by 5; this occurs in Lines 24 & 25. Lines 26-30 find the smaller of the two values, and multiply it by 10, which is the penalty incurred for this Masking Pattern.

## Class – Current Module

I decided to make a class for the current module of a QR Code matrix. There are 3 methods attached to this class; IsBlack(), IsWhite() and PaintModule(). Each of these methods is explained in detail below.

Method IsBlack() and IsWhite()

Both these methods takes 3 parameters; matrix\_copy(,), x\_coord and y\_coord.

matrix­\_copy is a full copy of the QR Code matrix at that point and x\_coord and y\_coord refer to the row and column of the current element in the matrix that we are interested in. IsBlack() checks the current element in the matrix that we are interested in and simply returns true if the current module is to be displayed as a Black module. IsWhite() checks the current element in the matrix that we are interested in and simply returns true if the current module is to be displayed as a White module. At numerous times throughout the development of my program I found that I was having to write long selection statements to check if the current module I was interested in would be Black or White. This is due to the fact that there are multiple integers that were encoded into the 2-D array that could represent a Black/White module.

As an example, the Black sections of the Finder Patterns would be recorded in the matrix as the integer ‘9’ and the Black sections of the Timing Patterns would be recorded in the matrix as the integer ‘6’. Essentially, even though ultimately two different modules would be represented as Black modules, it was important during some development stages that I could differentiate between the black Finder Pattern Modules and the black Timing Pattern Modules for example. However it was also important during some stages to simply check if a module I was interested in was Black or White. As mentioned before this could lead to long selection statements. Before I implemented the class, to check if a current module was white the selection statement would be:

If (matrix\_copy(x\_coord, y\_coord) = 0) Or

(matrix\_copy(x\_coord, y\_coord) = 5) Or

(matrix\_copy(x\_coord, y\_coord) = 7) Or

(matrix\_copy(x\_coord, y\_coord) = 9) Then

Return "Is White"

After I implemented the class to check if a current module was white the selection statement would simply be:

If CurrentModule.IsWhite(matrix\_copy, x\_coord, y\_coord) Then

Return "IsWhite"

As a final note on these two methods, I understand that if the method IsBlack() returned a False Boolean value then, funatmentally, we can deduce that the current module selected is White and such there would be no need for the separate method IsWhite, however I deliberately added both the methods as it was much easier for me to visually distinguish between ‘IsBlack = True’ and ‘IsWhite = True’ during the development process, especially when it came to the more complex selection statements.

PaintModule()

This method simply takes the parameters: matrix(,), x\_coord, y\_coord, PictureBox1, Colour

Once again matrix(,) is the current 2-D array of the QR Code matrix and x\_coord and y\_coord refer to the current module we are interested in. PictureBox1 refers to the graphical bitmap image on the Windows Form which the actual QR Code will be displayed onto and Colour refers to the colour which the moduke you are currently interested in should be painted. During the development process this was quite useful as it allowed me to visually distinguish between, for example, different Finder Patterns simply by painting them in a different colour.

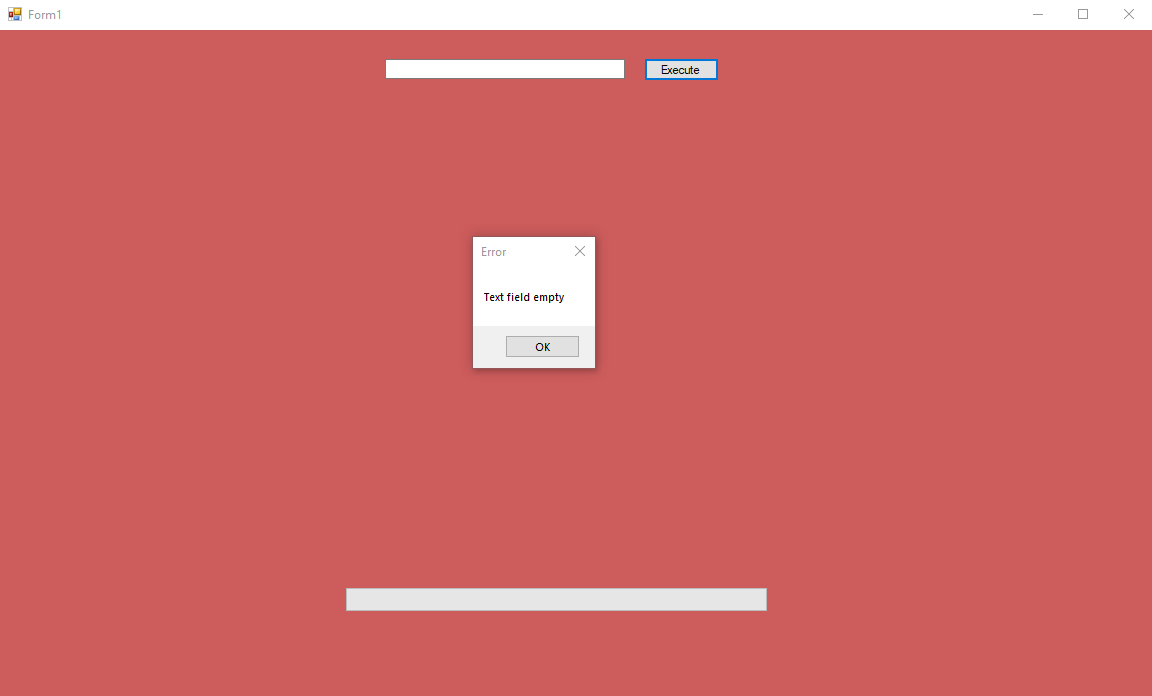
# Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number of Test** | **Data Inputted** | **Expected Result** | **Actual Result** | **Evidence** |
| 1 | EmptyString  Simulating no data being entered and the “Execute” Button being clicked  This will aim to prove that I have met System Objective 1 | The program will quickly recognize that no text has been entered and not attempt to produce a QR Code. The user will be informed of the error and should be able to re-enter data | No QR Code is generated. A message box appears informing the user that the text field is empty. The user is then able to type in data and the program continues as normal | See “Test – 1 Empty String” in the Testing Evidence Section  No scanner evidence required as no QR Code was generated |
| 2 | £  This QR Code can only encode a certain variety of characters and such some characters are considered illegal. For all legal characters see table in the section “Research after Conversation (1) – Data Encoding”  This will aim to prove that I have met System Objective 1 | The program will quickly recognize that illegal characters have been entered and not attempt to produce a QR Code. The user will be informed of the error and should be able to re-enter data | No QR Code is generated. A message box appears informing the user that they have entered an illegal character. The user is then able to type in data and the program continues as normal | See “Test 2 - Illegal Character” in the Testing Evidence Section  No scanner evidence required as no QR Code was generated |
| 3 | XXXXXXXXXXXXXXXXXXXXX  This QR Code generator can encode up to and including 20 characters  This will aim to prove that I have met System Objective 1 | The program will quickly recognize that too many characters have been entered and not attempt to produce a QR Code. The user will be informed of the error and should be able to re-enter data | No QR Code is generated. A message box appears informing the user that they have entered too many characters. The user is then able to type in data and the program continues as normal | See “Test 3 - Exceeding Capacity” in the Testing Evidence Section  No scanner evidence required as no QR Code was generated |
| 4 | HELLO WORLD  Ensure that a QR Code will be generated if only letters are inputted  This will aim to test all the System Objectives as if the scanner outputs the same data that was originally inputted, my program must be working correctly | A QR Code will be generated which when scanned will reveal the original data | The program displays a full QR Code and the scanner (my phone) is able to quickly recognize and scan it, informing the user that the symbol contains: HELLO WORLD | See “Test 4 – Letter Data Only” in the Testing Evidence Section  Scanner Evidence is also included in this section |
| 5 | 1234567890  Ensure that a QR Code will be generated if only numbers are inputted  This will aim to test all the System Objectives as if the scanner outputs the same data that was originally inputted, my program must be working correctly | A QR Code will be generated which when scanned will reveal the original data | The program displays a full QR Code and the scanner (my phone) is able to quickly recognize and scan it, informing the user that the symbol contains: 1234567890 | See “Test 5 – Numeric Data Only” in the Testing Evidence Section  Scanner Evidence is also included in this section |
| 6 | $%\*+-./:  Ensure that a QR Code will be generated if a range of legal symbols are inputted  This will aim to test all the System Objectives as if the scanner outputs the same data that was originally inputted, my program must be working correctly | A QR Code will be generated which when scanned will reveal the original data | The program displays a full QR Code and the scanner (my phone) is able to quickly recognize and scan it, informing the user that the symbol contains: $%\*+-./: | See “Test 6 - Symbol Data Only” in the Testing Evidence Section  Scanner Evidence is also included in this section |
| 7 | HELLO 12345 \*%$  Ensure that a QR Code will be generated if a mixture of letters, numbers and symbols are inputted  This will aim to test all the System Objectives as if the scanner outputs the same data that was originally inputted, my program must be working correctly | A QR Code will be generated which when scanned will reveal the original data | The program displays a full QR Code and the scanner (my phone) is able to quickly recognize and scan it, informing the user that the symbol contains: HELLO 12345 \*%$ | See “Test 7 – Mixed Data” in the Testing Evidence Section  Scanner Evidence is also included in this section |
| 8 | Error Correction: Minor Damage  This test ensures that a generated QR Code can sustain slight damage and still be readable by a scanner  To simulate “Minor Damage” a single 5p coin was placed on the QR Code and the phrase “HELLO WORLD” was inputted  This will aim to prove that I have met System Objective 7 | The QR Code will be readable  The 5p coin covered approximately 9 modules on the printed QR Code relating to approximately 3% of the Data/Error correction modules being damaged. This is lower than the 15% rated damage level for Error Correction Level M QR Codes  Note that this test was carried out multiple times, with the position of the coin being altered with each test | The QR Code was still completely readable by my scanner, regardless of where the 5p coin was placed on the QR Code  Relative scanning time was not affected | See “Test 8 - Error Correction: Minor Damage” in the Testing Evidence Section  Scanner Evidence is also included in this section |
| 9 | Error Correction: Medium Damage  This test ensures that a generated QR Code can sustain medium levels of damage and still be readable by a scanner  To simulate “Medium Damage” two 20p coins were placed on the QR Code and the phrase “HELLO WORLD” was inputted  This will aim to prove that I have met System Objective 7 | The QR Code will be readable  The two 20p coins covered approximately 32 modules on the printed QR Code relating to approximately 10% of the Data/Error correction modules being damaged. This is lower than the 15% rated damage level for Error Correction Level M QR Codes  This test was carried out multiple times, with the positions of the coins being altered with each test | The QR Code was still completely readable by my scanner, regardless of where the two 10p coins were placed on the QR Code  Relative scanning time was not affected | See “Test 9 - Error Correction: Medium Damage” in the Testing Evidence Section  Scanner Evidence is also included in this section |
| 10 | Error Correction: Major Damage  This test ensures that a generated QR Code can sustain a relatively large amount of damage and still be readable by a scanner  To simulate “Major Damage” two 10p coins were placed on the QR Code and the phrase “HELLO WORLD” was inputted  This will aim to prove that I have met System Objective 7 | The QR Code will be readable  The two 10p coins covered approximately 40 modules on the printed QR Code relating to approximately 13% of the Data/Error correction modules being damaged. This is lower than the 15% rated damage level for Error Correction Level M QR Codes | The QR Code was still readable by my scanner, regardless of where the two 10p coins were placed on the QR Code  Scanning time increased by a few seconds | See “Test 10 - Error Correction: Major Damage” in the Testing Evidence Section  Scanner Evidence is also included in the section |
| 11 | Error Correction: Over Damage  Here we are testing that the QR Code will not be able to be read if more modules are damaged than the maximum number dictated by the Error Correction Level  To simulate “Over Damage” three 10p coins were placed on the QR Code and the phrase “HELLO WORLD” was inputted  This will aim to prove that I have met System Objective 7 | The QR Code will not readable  The three 10p coins covered approximately 60 modules on the printed QR Code relating to approximately 20% of the Data/Error correction modules being damaged. This is higher than the 15% rated damage level for Error Correction Level M QR Codes and as such I am not expecting the QR Code to be readable | The QR Code was not readable by my scanner, regardless of where the three 10p coins were placed on the QR Code | See “Test 11 - Error Correction: Over Damage” in the Testing Evidence Section  Scanner Evidence is not included as the QR Code was not able to be scammed |

Note - Throughout the development of my program, System Objectives 2-5 were tested by looking through the worked examples on the QR Code Tutorial found on the Thonky.com website. As an example, for System Objective 4 in which I was generating Error Correction Codewords, I used the same input data that was used on the Thonky.com tutorial and my program produced the same Error Correction Codewords as theirs did. This proved that my program was generating the correct Error Correction Codewords. Additionally, System Objective 6 was met in Tests 4-11 where a QR Code was generated, graphically displayed and scanned to extract the original data.

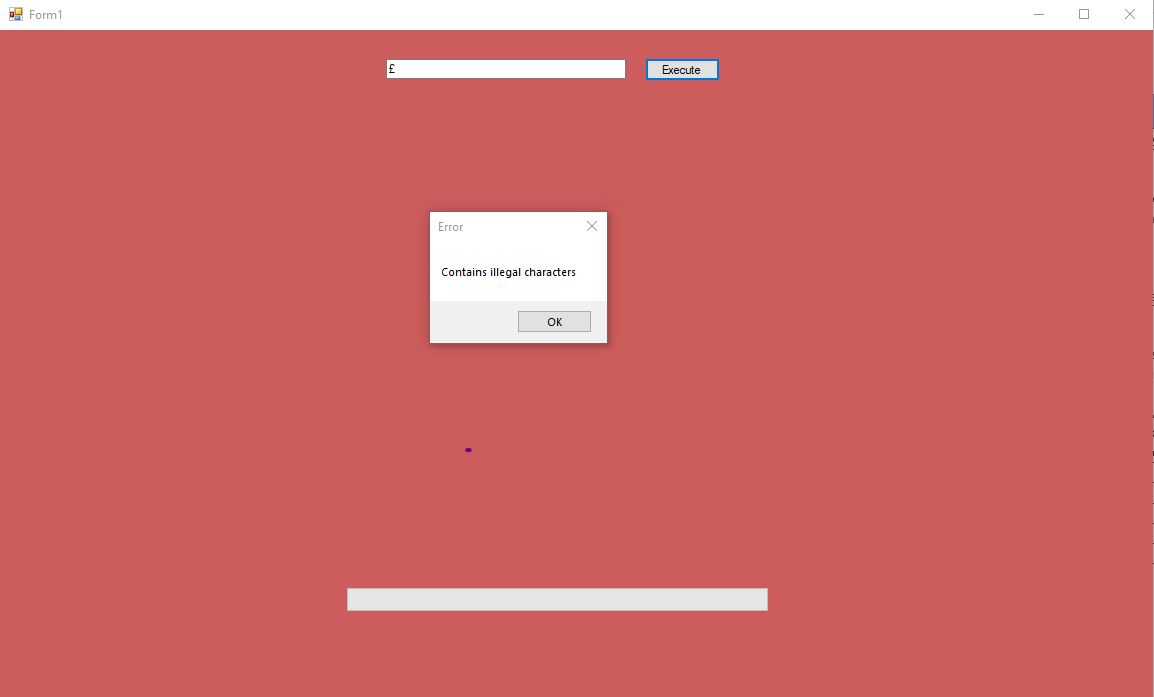
# Testing Evidence

## Test 1 – Empty String



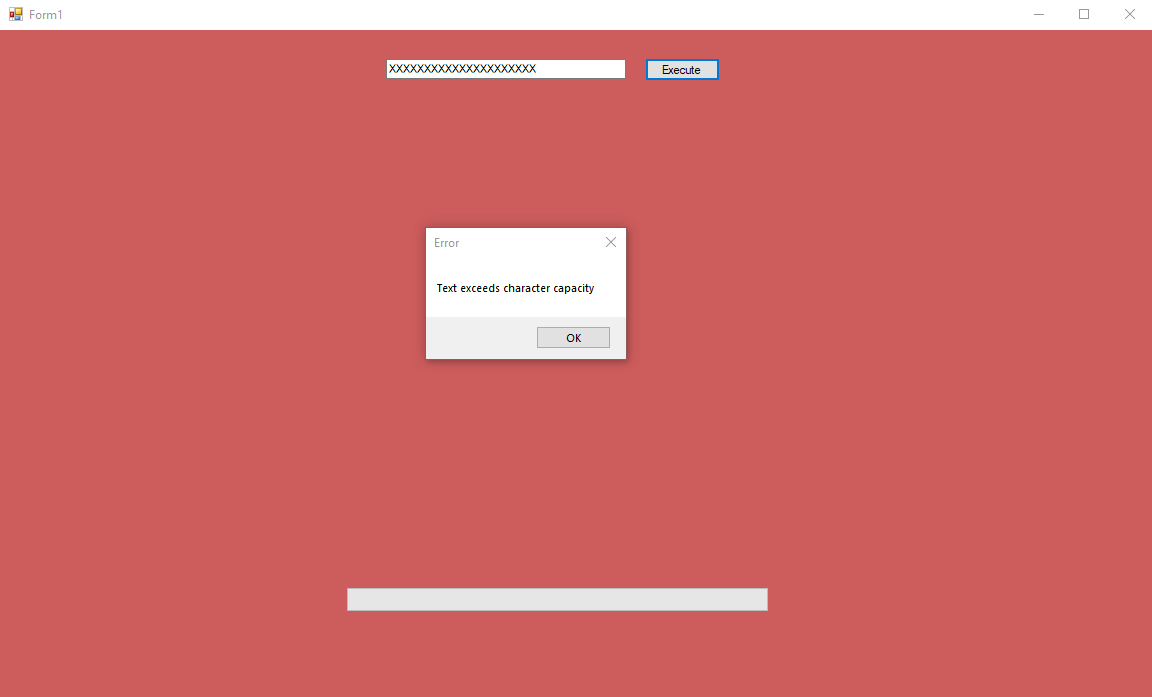
This shows that if the user doesn’t enter any text into the input box, but still clicks the ‘Execute’ button, a message box will appear which informs the user of their error. If the user then clicks ‘OK’ on the message box, they can enter data and ‘Execute’ as normal. The message box will appear anytime the input box is empty, and the user clicks ‘Execute’.

## Test 2 – Illegal Character



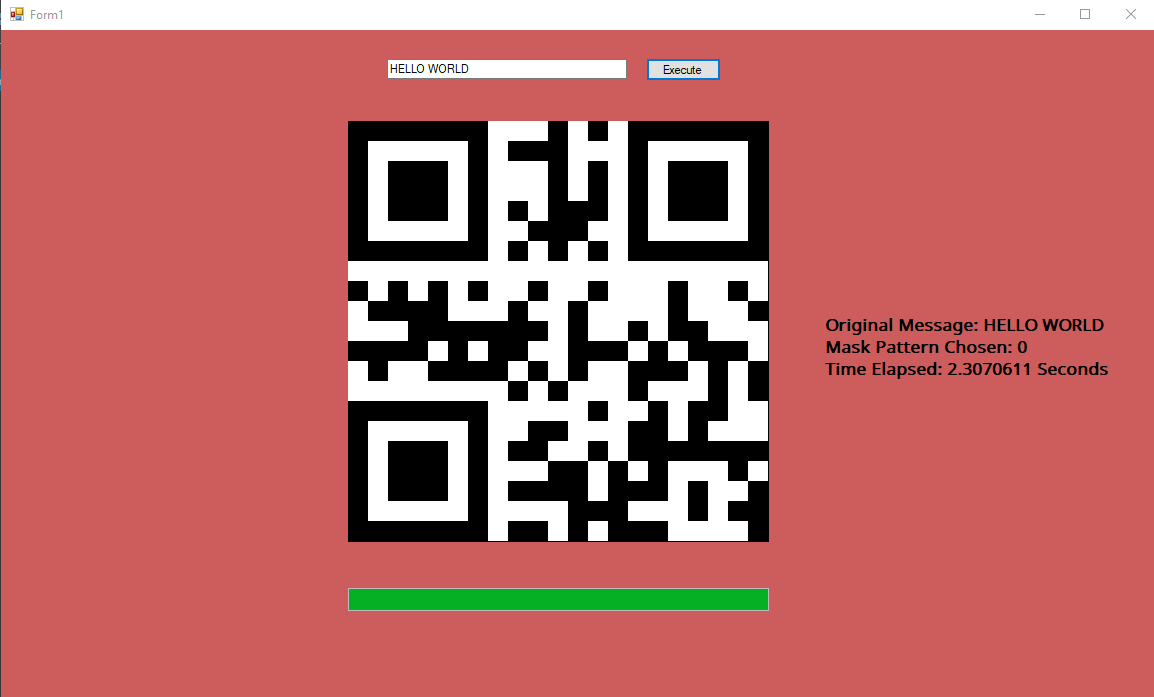
This shows that if the user enters any characters which aren’t classed as ‘Alphanumeric’ (according to the table in the QR Standard Document), and clicks the ‘Execute’ button, a message box will appear which informs the user that they have entered illegal characters. If the user then clicks ‘OK’ on the message box, they can enter data and ‘Execute’ as normal. The message box will appear anytime the user enters illegal characters in their input text and user clicks ‘Execute’.

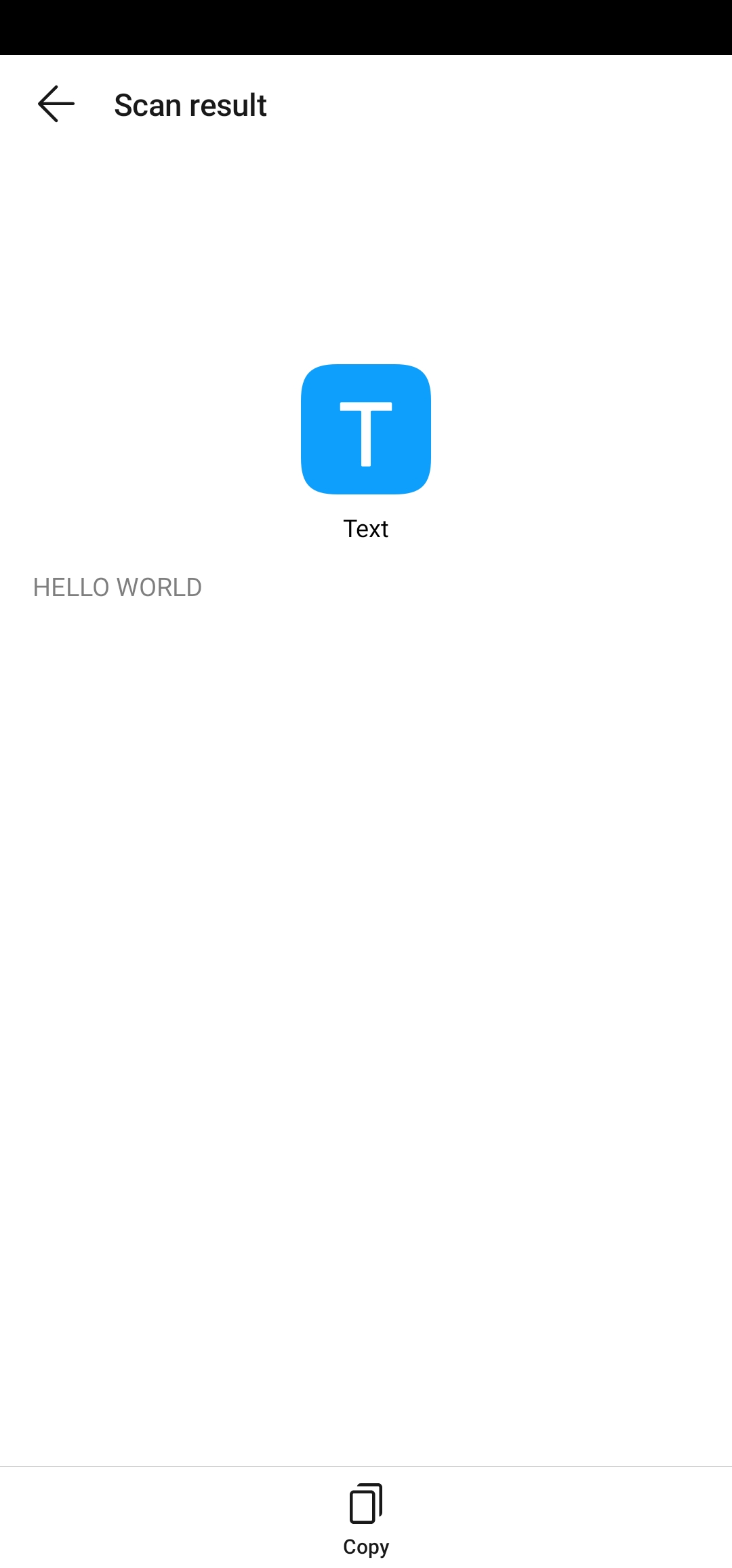
## Test 3 - Exceeding Capacity



This shows that if the user enters text which exceeds the maximum number of input characters (20 characters is the maximum for this Version and Error Correction Level), into the input box, and still clicks the ‘Execute’ button, a message box will appear which informs the user that they have exceeded the character limit. If the user then clicks ‘OK’ on the message box, they can enter data and ‘Execute’ as normal. The message box will appear anytime the input box contains more than 20 characters.

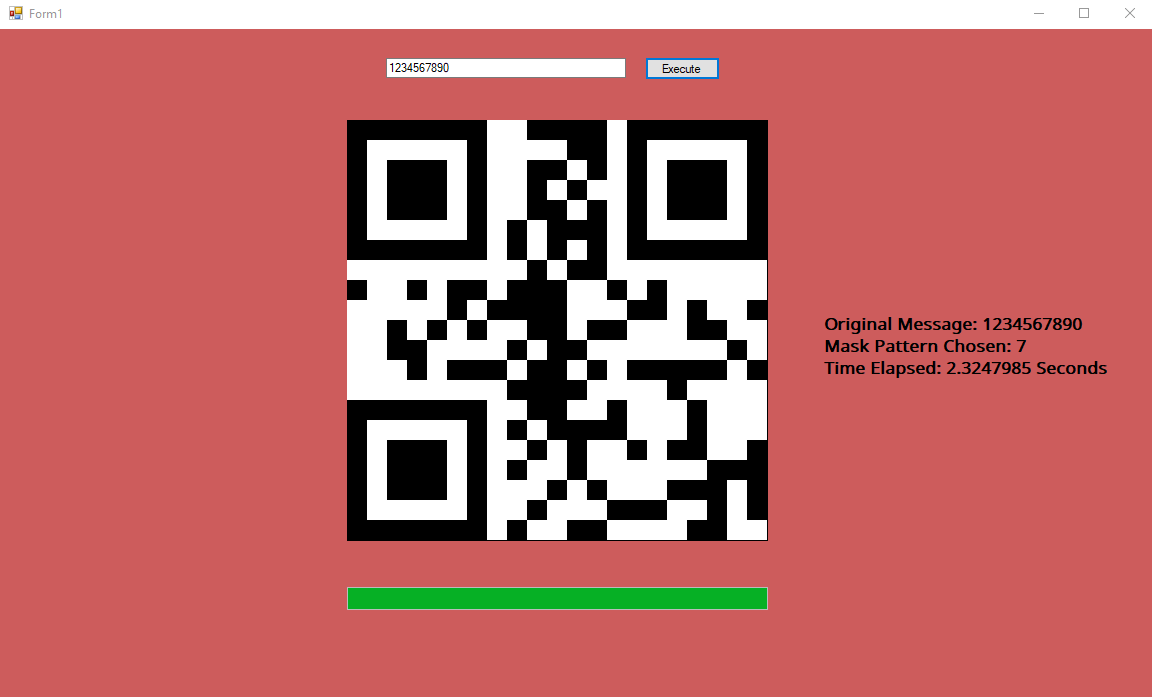
## Test 4 – Letter Data Only

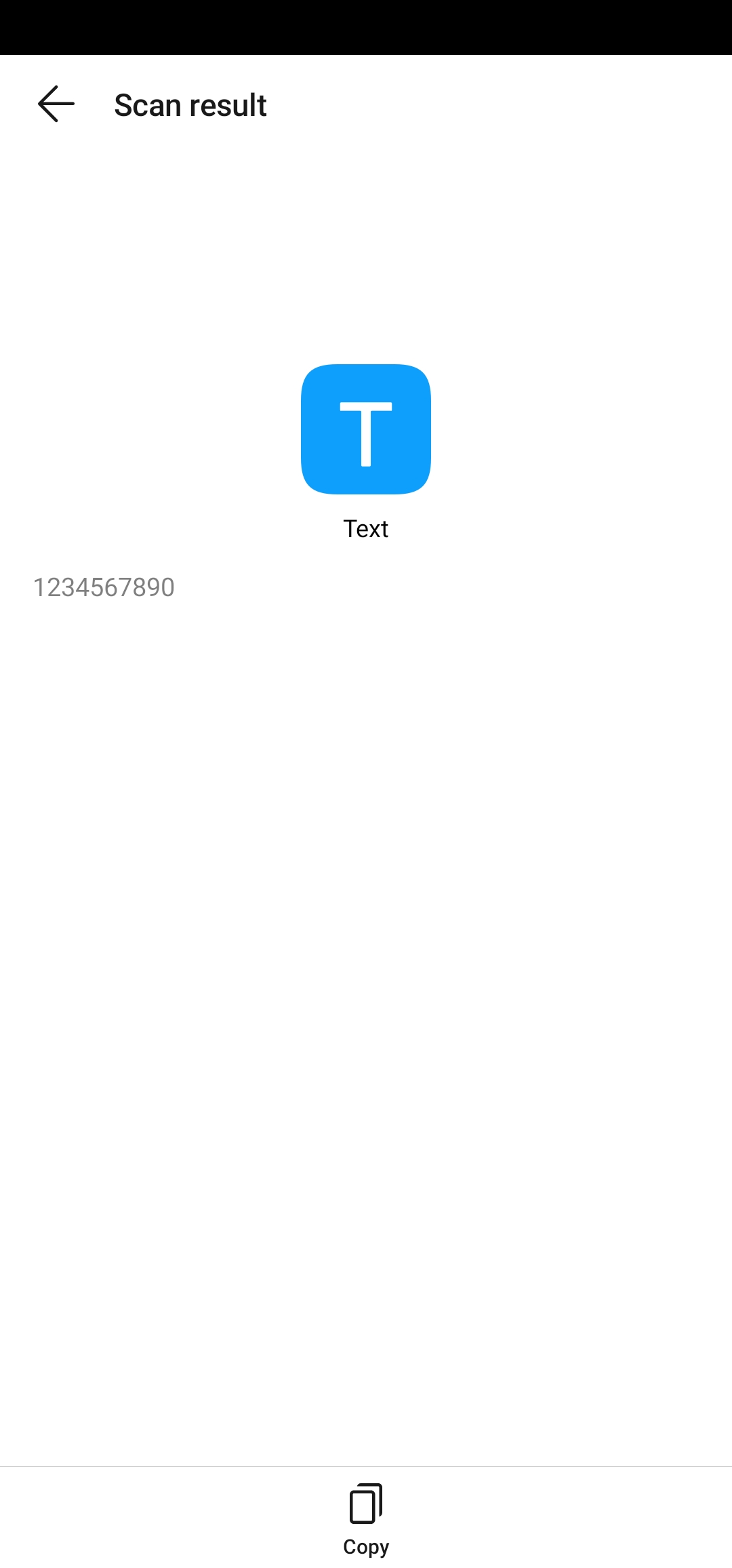




This shows that if the user enters data into the input box that consists only of characters, and clicks the ‘Execute’ button, a QR Code is generated in a few seconds. Also included is a screenshot from the output of the scanner. As you can see the scan result matches the originally entered data.

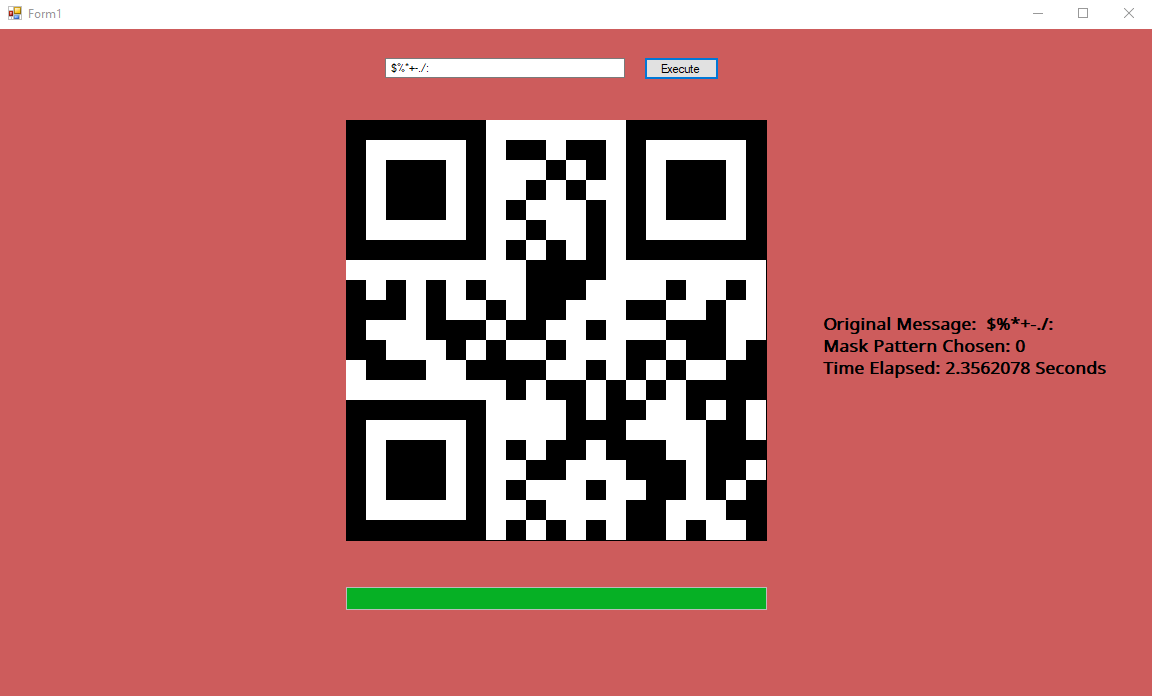
## Test 5 - Numeric Data Only

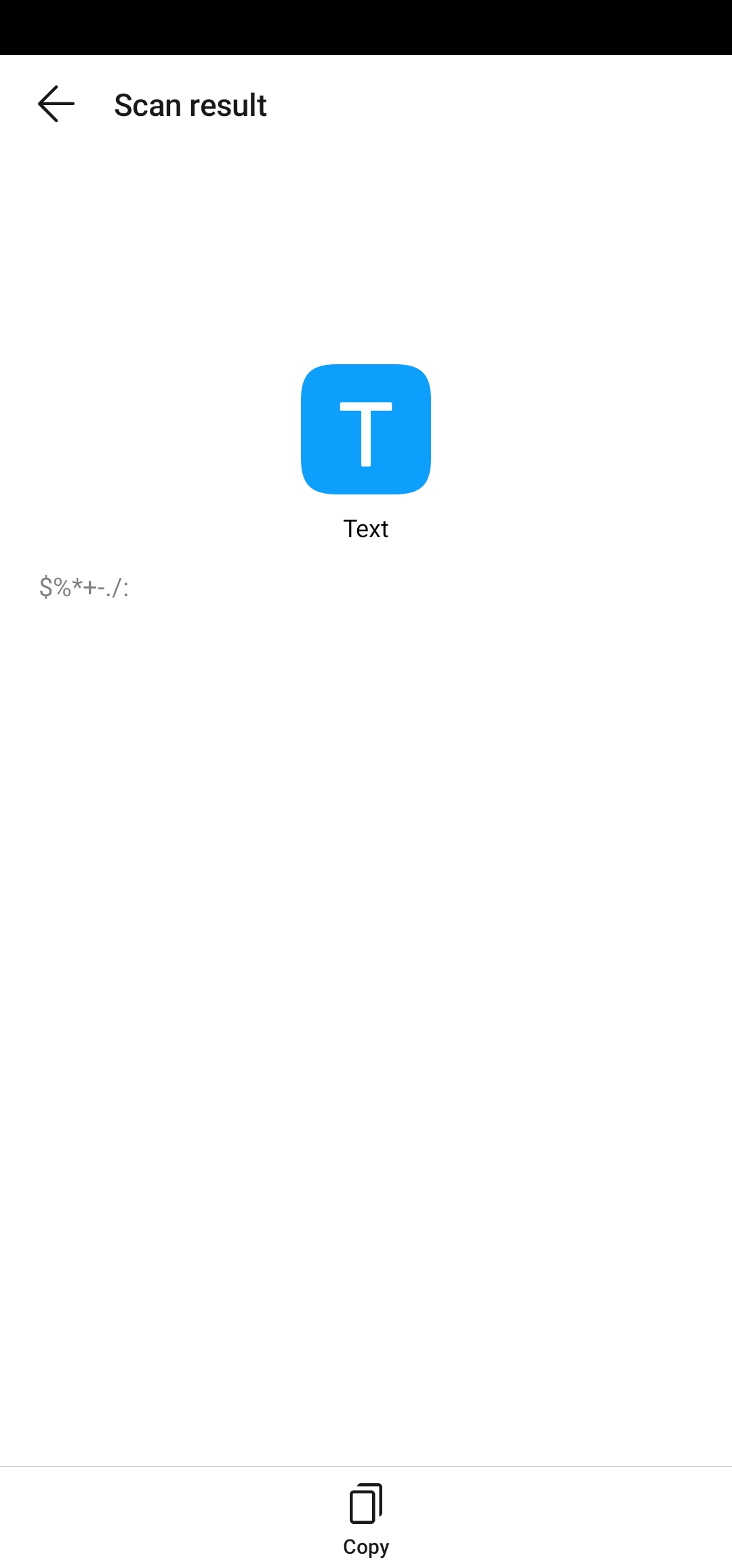




This shows that if the user enters data into the input box that consists only of numeric data, and clicks the ‘Execute’ button, a QR Code is generated in a few seconds. Also included is a screenshot from the output of the scanner. As you can see the scan result matches the originally entered data.

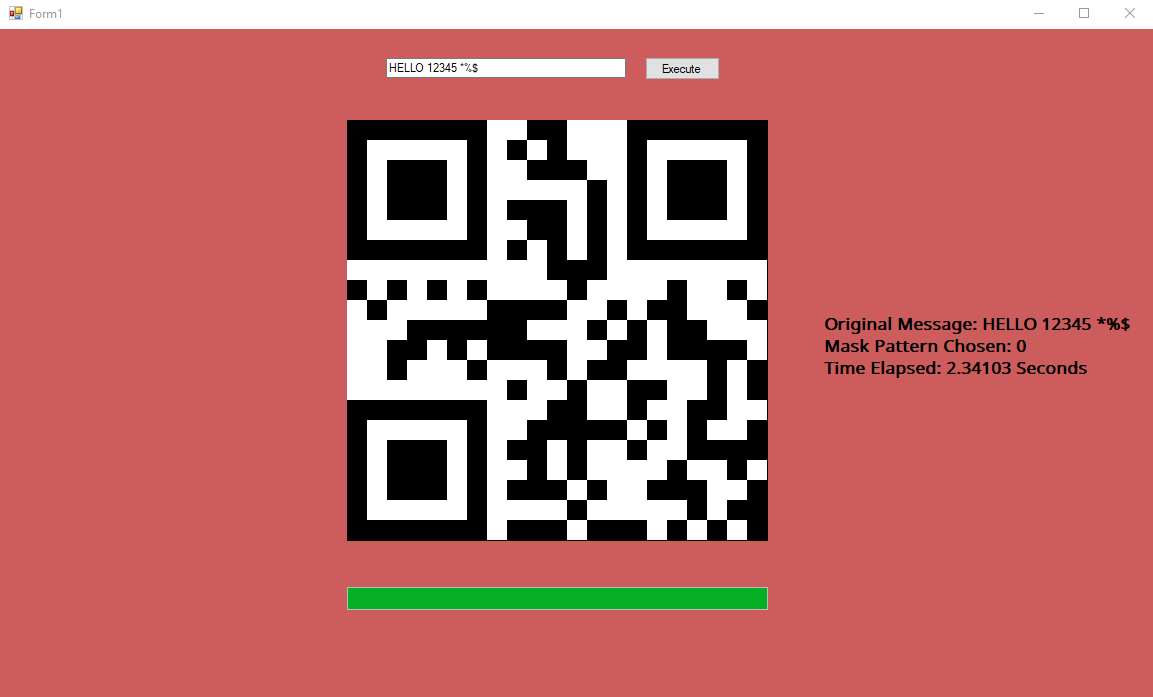
## Test 6 - Symbol Data Only

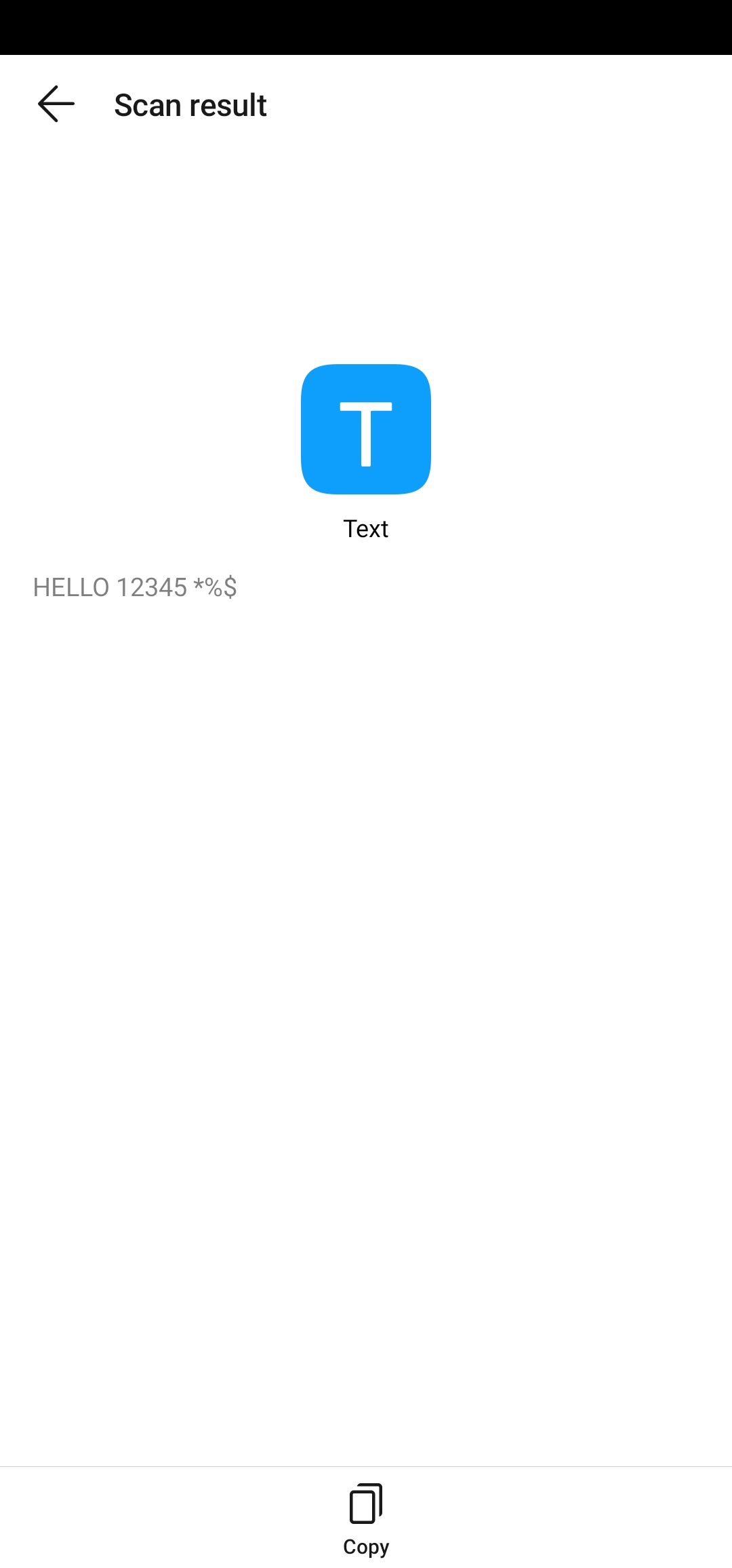




This shows that if the user enters data into the input box that consists only of symbols, and clicks the ‘Execute’ button, a QR Code is generated in a few seconds. Also included is a screenshot from the output of the scanner. As you can see the scan result matches the originally entered data.

## Test 7 - Mixed Data

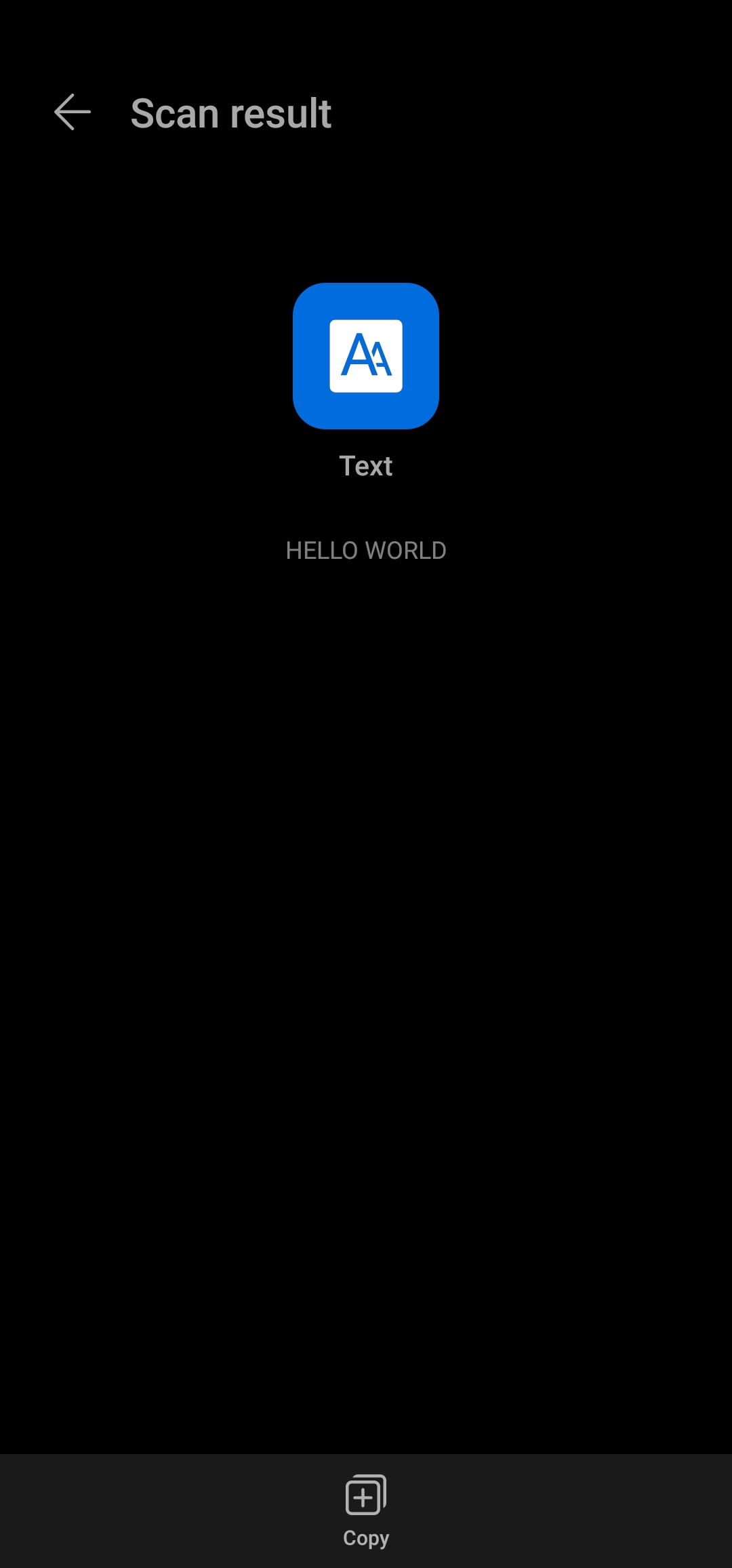




This shows that if the user enters data into the input box that consists of a mix of Letters, Numbers and Symbols, and clicks the ‘Execute’ button, a QR Code is generated in a few seconds. Also included is a screenshot from the output of the scanner. As you can see the scan result matches the originally entered data.

## Test 8 – Error Correction: Minor Damage

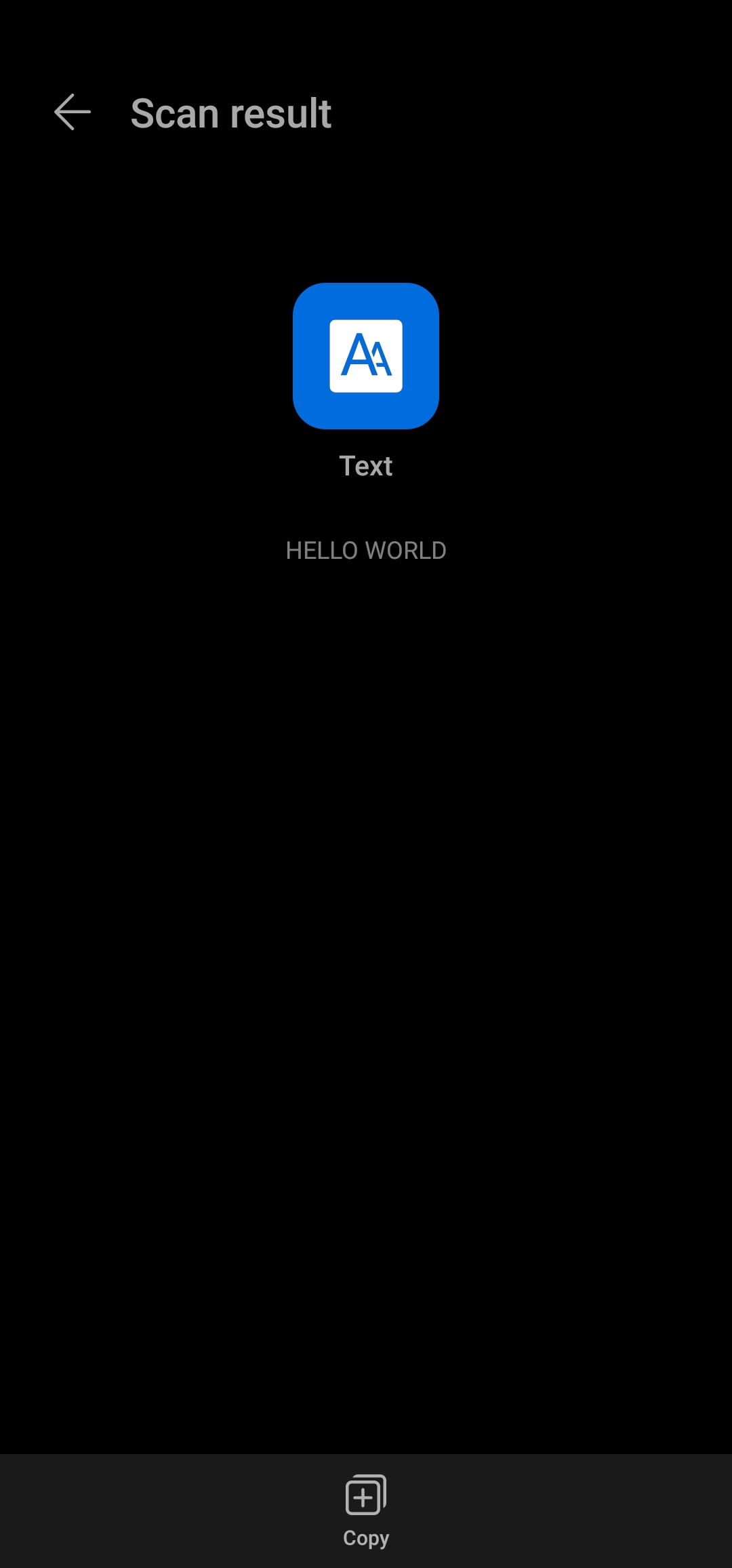




This shows that if a small portion of the QR Code is covered (damaged), it is still able to be scanned. Also included is the scan result of the QR Code, proving that small amounts of damage does not inhibit it’s ability to be scanned.

## Test 9 – Error Correction: Medium Damage

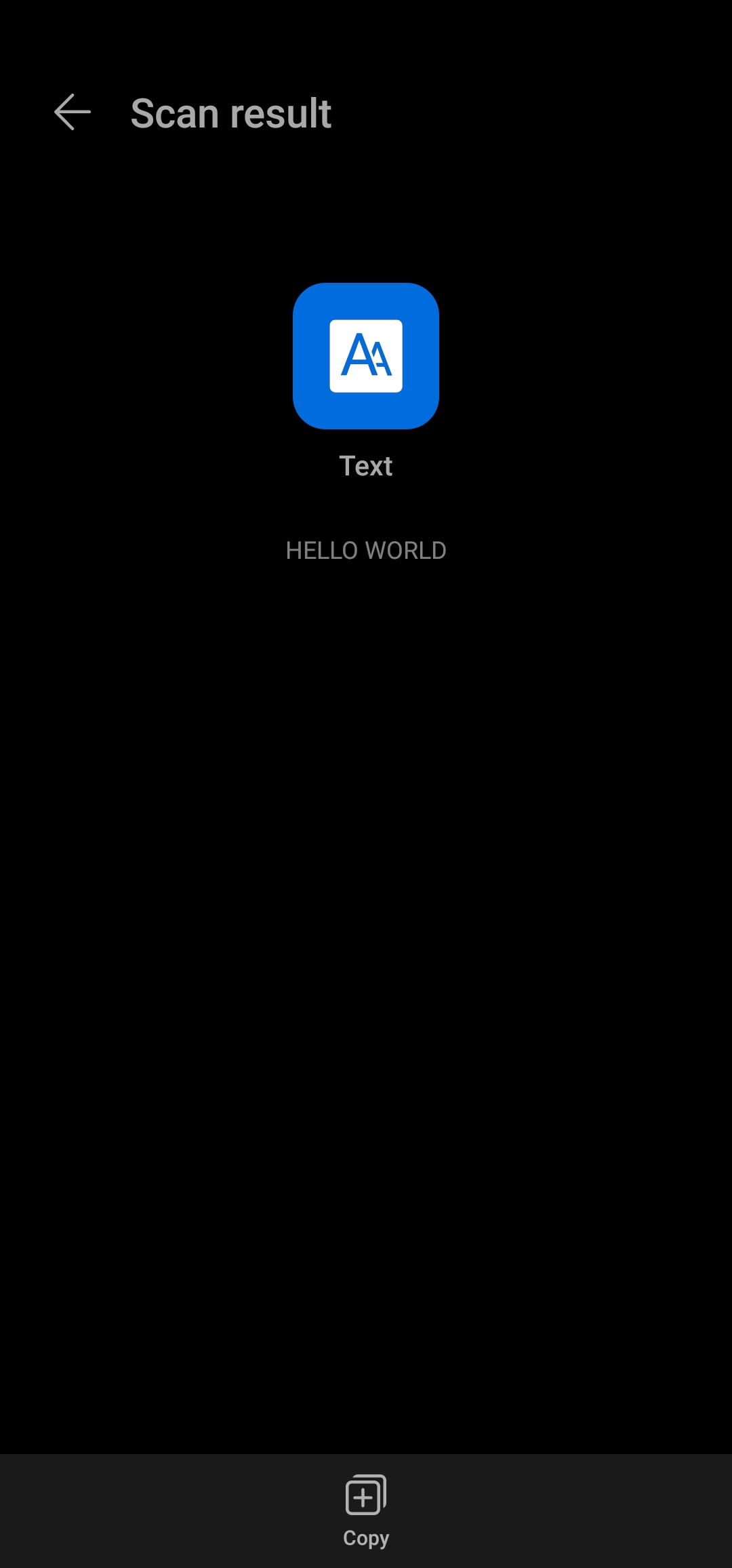




This shows that if a medium portion of the QR Code is covered (damaged), it is still able to be scanned. Also included is the scan result of the QR Code, proving that medium amounts of damage does not inhibit it’s ability to be scanned

## Test 10 - Error Correction: Major Damage





This shows that if a large portion of the QR Code is covered (damaged), it is still able to be scanned. Also included is the scan result of the QR Code, proving that large amounts of damage does not inhibit it’s ability to be scanned.

## Test 11 - Error Correction: Exceeding Damage



This shows that if a very large area of the QR Code is covered, it is not able to be scanned at any angle. No scanner evidence is included as I was not reliably able to scan the QR Code to extract the original data.

# Technical Solution

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

Me.BackColor = Color.FromArgb(204, 0, 0)

Label\_Statistics.Visible = False

End Sub

Private Sub Button1\_Click(sender As Object, e As EventArgs) Handles Button\_Execute.Click

Dim input\_text As String = Input\_Box.Text

Dim length\_input As Integer = Input\_Box.TextLength

Dim length\_stream As Integer = 4 + 9 + (11 \* (length\_input \ 2)) + (6 \* (length\_input Mod 2))

Dim encoding\_list As String = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ $%\*+-./:"

Dim split\_list As New List(Of String)

Dim extra As String = ""

Dim validation\_return As Boolean = Validation(input\_text, length\_input, encoding\_list)

Dim binary\_string As String

Dim pad\_codeword\_1 As String = "11101100"

Dim pad\_codeword\_2 As String = "00010001"

Dim codewords\_string As String

Dim data\_codewords() As String

Dim error\_codewords As New List(Of String)

Dim final\_data As New List(Of String)

Dim message\_polynomial\_coefficiants As New List(Of String)

Dim temp As String

Dim stopwatch As New Stopwatch

stopwatch = Stopwatch.StartNew

ProgressBar1.Value = 0

For Each Lbl As Label In Me.Controls.OfType(Of Label)()

Lbl.Text = ""

Next

Label\_Statistics.Text = "Original Message: " & input\_text & vbCr

split\_list.Clear()

If validation\_return = True Then

If length\_input Mod 2 = 0 Then 'If the input splits exactly into groups of 2

For a = 1 To (length\_input \ 2)

split\_list.Insert(0, input\_text.Substring(length\_input + (a \* -2), 2)) 'Insert each group of two into a list

Next

ElseIf length\_input Mod 2 = 1 Then 'If the input doesn't split exactly into groups of 2 (one extra character)

If length\_input = 1 Then 'If there is only one character then insert it into the list

split\_list.Insert(0, input\_text.Substring(0, 1))

Else

extra = input\_text.Substring(length\_input - 1, 1) 'Else put the single extra character into a temp variable

input\_text = input\_text.Remove(length\_input - 1, 1) 'And insert the remaining groups of two into the list

Dim temp\_length\_input As Integer = input\_text.Length

For a = 1 To (temp\_length\_input \ 2)

split\_list.Insert(0, input\_text.Substring(temp\_length\_input + (a \* -2), 2))

Next

Label4.Text = "Temp Length: " & temp\_length\_input

End If

End If

binary\_string = Conversion(split\_list, extra, length\_input, encoding\_list)

Label9.Text = binary\_string & vbCr & "Original Bit Stream Length (without terminator) : " & length\_stream 'Length of bit stream WITHOUT any terminator or additional padding bits

While ((binary\_string.Length) Mod 8) <> 0 'Codewords must be 8 bits in length, pad with 0 until true

binary\_string &= "0"

End While

Label10.Text = binary\_string & vbCr & "New Length: " & binary\_string.Length & vbCr & "Added: " & ((binary\_string.Length - length\_stream) - 4) & vbCr & ("Difference: ") & (128 - binary\_string.Length)

If binary\_string.Count = 128 Then

codewords\_string = CodewordSplit(binary\_string)

Else

For a = 0 To ((128 - binary\_string.Length) \ 8) - 1

If (a Mod 2) = 0 Then

binary\_string &= pad\_codeword\_1

Else

binary\_string &= pad\_codeword\_2

End If

Next

End If

codewords\_string = CodewordSplit(binary\_string)

codewords\_string = codewords\_string.Replace(",", "")

data\_codewords = codewords\_string.Split("-")

Label11.Text = "Data Codewords"

For b = 0 To data\_codewords.Count - 1

Label11.Text = Label11.Text & vbCr & data\_codewords(b)

Next

'-------------ERROR CORRECTION-------------'

For c = 0 To data\_codewords.Count - 2 'Generates message polynomial coefficiants

temp = Convert.ToInt32(data\_codewords(c), 2)

Label12.Text = Label12.Text & vbCr & temp

message\_polynomial\_coefficiants.Add(temp)

Next

Label13.Text = 0.5 'Ok so im not sure why this works but its fine

Dim antilog\_list As New List(Of String)

For a = 0 To 255

Label13.Text = Label13.Text \* 2

If Label13.Text >= 256 Then

Label13.Text = Label13.Text Xor 285

End If

antilog\_list.Add(Label13.Text)

Next

For a = 0 To antilog\_list.Count - 1

Label14.Text = Label14.Text & vbCr & antilog\_list(a)

Next 'End of the bits im not sure how they work

For a = 0 To 15 'Recursiveley calls the error correction function to generate codewords

message\_polynomial\_coefficiants = ErrorCorrectionGeneration(message\_polynomial\_coefficiants, antilog\_list)

Next

Label18.Text = "Error Codewords"

message\_polynomial\_coefficiants.RemoveRange(10, 6) 'Removes the trailing 0's

'For a = 0 To message\_polynomial\_coefficiants.Count - 1 'Displays error correction codewords

' Label18.Text = Label18.Text & vbCr & message\_polynomial\_coefficiants(a)

'Next

'Label18.Text = message\_polynomial\_coefficiants.Count

Dim hold As Integer

Label18.Visible = False

For a = 0 To message\_polynomial\_coefficiants.Count - 1

hold = CInt(message\_polynomial\_coefficiants(a))

Label18.Text = Convert.ToString(hold, 2).PadLeft(8, "0")

error\_codewords.Add(Label18.Text)

Next

Label18.ResetText()

Label18.Text = "Error Codewords"

For a = 0 To error\_codewords.Count - 1

Label18.Text = Label18.Text & vbCr & error\_codewords(a)

Next

Label18.Visible = False

final\_data.AddRange(data\_codewords)

final\_data.AddRange(error\_codewords)

final\_data.RemoveAt(16)

Label19.Text = "FINAL DATA"

Label21.Text = ""

For a = 0 To final\_data.Count - 1

Label19.Text = Label19.Text & vbCr & final\_data(a)

Label21.Text = Label21.Text & final\_data(a)

Next

Dim final\_data\_string As String = Label21.Text

'-----ERROR CORRECTION-----

DisplayingMatrix(final\_data\_string)

End If

For i As Integer = 0 To split\_list.Count - 1

Label1.Text += (split\_list(i)) & vbCr

Next

Label2.Text = "Groups of two: " & length\_input \ 2 & vbCr & "Extra: " & extra

Label3.Text = "Original Length:" & length\_input

System.Threading.Thread.Sleep(500)

stopwatch.Stop()

Label\_Statistics.Text = Label\_Statistics.Text & "Time Elapsed: " & stopwatch.Elapsed.TotalSeconds & " Seconds"

End Sub

Public Function Validation(ByVal input\_text, ByVal length\_input, ByVal encoding\_list)

Dim test As Boolean = False

If length\_input = 0 Then

MessageBox.Show("Text field empty", "Error")

QR\_Code\_Display.SendToBack()

ElseIf (length\_input > 20) Then 'Encoding strictly in Version 1-M: 21 x 21 modules MAX 20 Alphanumeric Characters

''Max number of binary data bits for 1-M is 128, for Alphanumeric only, we will reach a max of 123 (20 input characters)

MessageBox.Show("Text exceeds character capacity", "Error")

Else

test = True

End If

For a = 0 To length\_input - 1

If encoding\_list.Contains(input\_text.Substring(a, 1)) = False Then 'Checks if the input contains only valid Alphanumeric Characters

test = False

MessageBox.Show("Contains illegal characters", "Error")

Exit For

End If

Next

Return test

End Function

Public Function Conversion(ByVal split\_list As List(Of String), ByVal extra As String, ByVal length\_input As Integer, ByVal encoding\_list As String)

Dim decimal\_list As New List(Of Integer)

Dim current As String

Dim position As Integer = 0

Dim value As Integer = 0

Dim binary\_list As New List(Of String)

Dim binary\_string As String

Dim mode\_indicator As String = "0010" 'Only encoding Alphanumeric for this project

Label5.ResetText()

For a = 0 To (length\_input \ 2) - 1 'For each of the pairs of two elements, gets their equivalent positions in the encoding table

current = split\_list(a)

For b = 0 To 1

position = encoding\_list.IndexOf(current.Substring(b, 1))

decimal\_list.Add(position)

Next

Next

For c = 1 To decimal\_list.Count - 1 Step 2 'For pairs of numbers, multiply the first by 45 and add to the second

value = (decimal\_list(c - 1) \* 45) + decimal\_list(c)

Label6.Text = Convert.ToString(value, 2).PadLeft(11, "0") 'Converts the calculated value to 11-bit Binary String

binary\_list.Add(Label6.Text)

Label5.Text = Label5.Text & vbCr & value

Next

For d = 0 To binary\_list.Count - 1

Label7.Text = Label7.Text & vbCr & binary\_list(d)

Next

If extra <> "" Then 'If theres an extra digit, convert to 6-bit Binary string

binary\_list.Add(Convert.ToString(encoding\_list.IndexOf(extra), 2).PadLeft(6, "0"))

Label7.Text = Label7.Text & vbCr & "Extra: " & binary\_list(binary\_list.Count - 1)

End If

binary\_list.Insert(0, Convert.ToString(length\_input, 2).PadLeft(9, "0")) 'Convert the character count to a 9-bit Binary string ''Added before the encoded data, after mode indicator

Label7.Text = Label7.Text & vbCr & "Character Count: " & binary\_list(0)

binary\_list.Insert(0, mode\_indicator) 'Adds mode indicator for Alphanumeric Mode before character count, before data

binary\_list.Add("0000") 'Terminator added at the very end of the data string

Label7.Text = Label7.Text & vbCr & "Mode Indicator: " & binary\_list(0)

'For a = 0 To binary\_list.Count - 1 ''Prints entire binary string as one

' Label8.Text = Label8.Text & binary\_list(a)

'Next

binary\_string = String.Join(",", binary\_list)

binary\_string = binary\_string.Replace(",", "")

Return binary\_string

End Function

Public Function CodewordSplit(ByVal binary\_string As String)

Dim data\_codewords As New List(Of String)

Dim codewords\_string

For a = 0 To 15

data\_codewords.Add(binary\_string.Substring((a \* 8), 8) & "-")

Next

codewords\_string = String.Join(",", data\_codewords)

Return codewords\_string

End Function

Public Function PlacingIntoMatrix(ByVal final\_data\_string As String)

Dim matrix(20, 20) As Integer '21 Modules x 21 Modules

'0 = White

'1 = Black

'2 = Dark Module

'3 = Reserved Area

'4 = Blank

'5 = Timing Pattern ("White")

'6 = Timing Pattern ("Black")

'7 = Finder Pattern ("White")

'8 = Finder Pattern ("Black")

'9 = Seperator

matrix = FinderPatterns() 'Adds 3 Finder Patterns in correct places

For a = 0 To 7 '-----Upper Left Seperators-----

matrix(7, a) = 9

Next

For a = 0 To 7

matrix(a, 7) = 9

Next '-----Upper Left Seperators-----

For a = 0 To 7 '-----Upper Right Seperators-----

matrix(13, a) = 9

Next

For a = 13 To 20

matrix(a, 7) = 9

Next '-----Upper Right Seperators-----

For a = 0 To 7 '-----Lower Left Seperators-----

matrix(a, 13) = 9

Next

For a = 13 To 20

matrix(7, a) = 9

Next '-----Lower Left Seperators-----

For a = 0 To 8 '-----Upper Left Reserved Area-----

matrix(8, a) = 3

Next

For a = 0 To 8

matrix(a, 8) = 3

Next '-----Upper Left Reserved Area-----

For a = 13 To 20 '-----Upper Right Reserved Area-----

matrix(a, 8) = 3

Next

For a = 14 To 20 '-----Lower Left Reserved Area-----

matrix(8, a) = 3

Next

For a = 8 To 12 'Horizontal Timing Pattern

If a Mod 2 = 0 Then

matrix(a, 6) = 6

Else

matrix(a, 6) = 5

End If

Next

For a = 8 To 12 'Vertical Timing Pattern

If a Mod 2 = 0 Then

matrix(6, a) = 6

Else

matrix(6, a) = 5

End If

Next

matrix(8, 13) = 2 'Dark Module

Dim counter As Integer = 0

Dim data\_string As String

For b = 0 To 11 'First Column: Up

For a = 0 To 1

matrix(20 - a, 20 - b) = CInt(final\_data\_string.Substring(counter, 1))

counter += 1

Next

Next

For b = 0 To 11 'Second Column: Down

For a = 0 To 1

matrix(18 - a, 9 + b) = CInt(final\_data\_string.Substring(counter, 1))

counter += 1

Next

Next

For b = 0 To 11 'Third Column: Up

For a = 0 To 1

matrix(16 - a, 20 - b) = CInt(final\_data\_string.Substring(counter, 1))

counter += 1

Next

Next

For b = 0 To 11 'Fourth Column: Down

For a = 0 To 1

matrix(14 - a, 9 + b) = CInt(final\_data\_string.Substring(counter, 1))

counter += 1

Next

Next

For b = 0 To 20 'Fifth Column: Up

For a = 0 To 1

If matrix(12 - a, 20 - b) <> 4 Then

matrix(12 - a, 20 - b) = matrix(12 - a, 20 - b)

Else

matrix(12 - a, 20 - b) = CInt(final\_data\_string.Substring(counter, 1))

counter += 1

End If

Next

Next

For b = 0 To 20 'Sixth Column: Down

For a = 0 To 1

If matrix(10 - a, b) <> 4 Then

matrix(10 - a, b) = matrix(10 - a, b)

Else

matrix(10 - a, b) = CInt(final\_data\_string.Substring(counter, 1))

counter += 1

End If

Next

Next

For b = 0 To 3 'Seventh Column: Up

For a = 0 To 1

matrix(8 - a, 12 - b) = CInt(final\_data\_string.Substring(counter, 1))

counter += 1

Next

Next

For b = 0 To 3 'Eighth Column: Down

For a = 0 To 1

matrix(5 - a, 9 + b) = CInt(final\_data\_string.Substring(counter, 1))

counter += 1

Next

Next

For b = 0 To 3 'Ninth Column: Up

For a = 0 To 1

matrix(3 - a, 12 - b) = CInt(final\_data\_string.Substring(counter, 1))

counter += 1

Next

Next

For b = 0 To 3 'Tenth Column: Down

For a = 0 To 1

matrix(1 - a, 9 + b) = CInt(final\_data\_string.Substring(counter, 1))

counter += 1

Next

Next

Label22.Text = counter

For a = 0 To 20 'Prints out entire matrix to a label

For b = 0 To 20

Next

Next

Dim matrix\_copy(20, 20) As Integer

Array.Copy(matrix, matrix\_copy, matrix.Length)

matrix = Masking(matrix\_copy, matrix)

For a = 0 To 20

For b = 0 To 20

Label16.Text = Label16.Text & matrix(b, a)

Next

Label16.Text = Label16.Text & vbCr

Next

Return matrix

End Function

Public Function FinderPatterns()

Dim matrix(20, 20) As Integer

For a = 0 To 20 'Fills the matrix with blank character (4)

For b = 0 To 20

matrix(b, a) = 4

Next

Next

For a = 0 To 6 '-----Upper Left Finder Pattern-----

For b = 0 To 6

matrix(b, a) = 8

Next

Next

For a = 1 To 5

For b = 1 To 5

matrix(b, a) = 7

Next

Next

For a = 2 To 4

For b = 2 To 4

matrix(b, a) = 8

Next

Next '-----Upper Left Finder Pattern-----

For a = 14 To 20 '-----Lower Left Finder Pattern-----

For b = 0 To 6

matrix(b, a) = 8

Next

Next

For a = 15 To 19

For b = 1 To 5

matrix(b, a) = 7

Next

Next

For a = 16 To 18

For b = 2 To 4

matrix(b, a) = 8

Next

Next '-----Lower Left Finder Pattern-----

For a = 0 To 6 '-----Upper Right Finder Pattern-----

For b = 14 To 20

matrix(b, a) = 8

Next

Next

For a = 1 To 5

For b = 15 To 19

matrix(b, a) = 7

Next

Next

For a = 2 To 4

For b = 16 To 18

matrix(b, a) = 8

Next

Next '-----Upper Right Finder Pattern-----

Return matrix

End Function

Public Sub DisplayingMatrix(ByVal final\_data\_string As String)

Dim matrix(20, 20) As Integer

QR\_Code\_Display.Image = New Bitmap(QR\_Code\_Display.Width, QR\_Code\_Display.Height)

Dim Drawing As Graphics = Graphics.FromImage(QR\_Code\_Display.Image)

Dim height As Integer = QR\_Code\_Display.Height

Dim partition As Integer = height / 21

Dim CurrentModule As New CurrentModule

matrix = PlacingIntoMatrix(final\_data\_string)

Drawing.DrawLine(Pens.Black, 0, 0, 0, 420) 'Left Border

Drawing.DrawLine(Pens.Black, 0, 0, 420, 0) 'Top Border

Drawing.DrawLine(Pens.Black, 420, 0, 420, 420) 'Right Border

Drawing.DrawLine(Pens.Black, 0, 420, 420, 420) 'Bottom Border

For a = 0 To 20 'Draw vertical grid lines

Drawing.DrawLine(Pens.Black, partition + (a \* partition), 0, partition + (a \* partition), height)

Next

For a = 0 To 20 'Draw horizontal grid lines

Drawing.DrawLine(Pens.Black, 0, partition + (a \* partition), height, partition + (a \* partition))

Next

QR\_Code\_Display.Refresh()

For a = 0 To 20

For b = 0 To 20

ProgressBar1.Increment(2)

If matrix(b, a) = 0 Then 'Fill with a white square if current position is a 0 (White)

CurrentModule.PaintModule(matrix, b, a, QR\_Code\_Display, Brushes.White)

ElseIf matrix(b, a) = 1 Then 'Fill with a black square if current position is a 1 (Black)

CurrentModule.PaintModule(matrix, b, a, QR\_Code\_Display, Brushes.Black)

ElseIf matrix(b, a) = 2 Then 'Fill with a Gold square if current position is a 2 (Dark Module)

CurrentModule.PaintModule(matrix, b, a, QR\_Code\_Display, Brushes.Black)

ElseIf matrix(b, a) = 3 Then 'Fill with a Blue square if current position is a 3 (Reserved Area)

CurrentModule.PaintModule(matrix, b, a, QR\_Code\_Display, Brushes.MediumBlue)

ElseIf matrix(b, a) = 4 Then 'Fill with a Grey square if current position is a 4 (Blank)

CurrentModule.PaintModule(matrix, b, a, QR\_Code\_Display, Brushes.DimGray)

ElseIf matrix(b, a) = 5 Then 'Fill with a White square if current position is a 5 (Timing Pattern ("White"))

CurrentModule.PaintModule(matrix, b, a, QR\_Code\_Display, Brushes.White)

ElseIf matrix(b, a) = 6 Then 'Fill with a Black square if current position is a 6 (Timing Pattern ("Black"))

CurrentModule.PaintModule(matrix, b, a, QR\_Code\_Display, Brushes.Black)

ElseIf matrix(b, a) = 7 Then 'Fill with a White square if current position is a 7 (Finder Pattern ("White"))

CurrentModule.PaintModule(matrix, b, a, QR\_Code\_Display, Brushes.White)

ElseIf matrix(b, a) = 8 Then 'Fill with a Black square if current position is a 8 (Finder Pattern ("Black"))

CurrentModule.PaintModule(matrix, b, a, QR\_Code\_Display, Brushes.Black)

ElseIf matrix(b, a) = 9 Then 'Fill with a White square if current position is a 9 (Seperator Pattern)

CurrentModule.PaintModule(matrix, b, a, QR\_Code\_Display, Brushes.White)

End If

Next

Next

Label\_Statistics.Visible = True

End Sub

Public Function ErrorCorrectionGeneration(ByRef message\_polynomial\_coefficiants As List(Of String), ByVal antilog\_list As List(Of String))

Dim generator\_polynomial\_alpha\_exponents As New List(Of Integer)(New Integer() {0, 251, 67, 46, 61, 118, 70, 64, 94, 32, 45})

Dim first\_term\_alpha\_notation As String

For a = 0 To antilog\_list.Count - 1

If antilog\_list(a) = message\_polynomial\_coefficiants(0) Then

first\_term\_alpha\_notation = a

End If

Next

Label15.Text = first\_term\_alpha\_notation

first\_term\_alpha\_notation = CInt(first\_term\_alpha\_notation)

For a = 0 To generator\_polynomial\_alpha\_exponents.Count - 1

If generator\_polynomial\_alpha\_exponents(a) + first\_term\_alpha\_notation >= 255 Then

generator\_polynomial\_alpha\_exponents(a) = (generator\_polynomial\_alpha\_exponents(a) + first\_term\_alpha\_notation) Mod 255

Else

generator\_polynomial\_alpha\_exponents(a) = generator\_polynomial\_alpha\_exponents(a) + first\_term\_alpha\_notation

End If

Next

For a = 0 To generator\_polynomial\_alpha\_exponents.Count - 1

generator\_polynomial\_alpha\_exponents(a) = antilog\_list(generator\_polynomial\_alpha\_exponents(a))

Label17.Text = Label17.Text & vbCr & generator\_polynomial\_alpha\_exponents(a)

Next

'Label20.Text = "Message: " & message\_polynomial\_coefficiants.Count & vbCr & "Generator: " & generator\_polynomial\_alpha\_exponents.Count

For a = 0 To generator\_polynomial\_alpha\_exponents.Count - 1

message\_polynomial\_coefficiants(a) = message\_polynomial\_coefficiants(a) Xor generator\_polynomial\_alpha\_exponents(a)

Next

message\_polynomial\_coefficiants.RemoveAt(0)

message\_polynomial\_coefficiants.Add(0)

Return message\_polynomial\_coefficiants

End Function

Public Function Masking(ByVal matrix\_copy(,) As Integer, ByVal matrix(,) As Integer)

Dim counter As Integer = 0

Dim score\_list As New List(Of Integer)

Dim masked\_matrix(20, 20) As Integer

Dim mask\_chosen As Integer = 0

Label23.Text = "Scores"

'Initial masks are applied To matrix\_copy, the array 'matrix' should never be altered

'Once mask has been evaluated, copy of matrix Is reset to the original state And the next mask Is applied

'Once all the evaluation scores are In, re-apply selected mask

score\_list.Add(MaskPattern0(matrix\_copy)) 'Adds the score for Mask Pattern 1 to a list

Array.Copy(matrix, matrix\_copy, matrix.Length) 'Resets matrix\_copy to original state

score\_list.Add(MaskPattern1(matrix\_copy)) 'Adds the score for Mask Pattern 2 to a list

Array.Copy(matrix, matrix\_copy, matrix.Length) 'Resets matrix\_copy to original state

score\_list.Add(MaskPattern2(matrix\_copy)) 'Adds the score for Mask Pattern 3 to a list

Array.Copy(matrix, matrix\_copy, matrix.Length) 'Resets matrix\_copy to original state

score\_list.Add(MaskPattern3(matrix\_copy)) 'Adds the score for Mask Pattern 4 to a list

Array.Copy(matrix, matrix\_copy, matrix.Length) 'Resets matrix\_copy to original state

score\_list.Add(MaskPattern4(matrix\_copy)) 'Adds the score for Mask Pattern 5 to a list

Array.Copy(matrix, matrix\_copy, matrix.Length) 'Resets matrix\_copy to original state

score\_list.Add(MaskPattern5(matrix\_copy)) 'Adds the score for Mask Pattern 6 to a list

Array.Copy(matrix, matrix\_copy, matrix.Length) 'Resets matrix\_copy to original state

score\_list.Add(MaskPattern6(matrix\_copy)) 'Adds the score for Mask Pattern 7 to a list

Array.Copy(matrix, matrix\_copy, matrix.Length) 'Resets matrix\_copy to original state

score\_list.Add(MaskPattern7(matrix\_copy)) 'Adds the score for Mask Pattern 8 to a list

Array.Copy(matrix, matrix\_copy, matrix.Length) 'Resets matrix\_copy to original state

For a = 0 To score\_list.Count - 1 'Output the scorelist for each mask pattern

Label23.Text = Label23.Text & vbCr & score\_list(a)

Next

If score\_list.IndexOf(score\_list.Min) = 0 Then 'This IF Statement checks which mask produced the lowest penalty and re-applies it

MaskPattern0(matrix\_copy)

mask\_chosen = 0

ElseIf score\_list.IndexOf(score\_list.Min) = 1 Then

MaskPattern1(matrix\_copy)

mask\_chosen = 1

ElseIf score\_list.IndexOf(score\_list.Min) = 2 Then

MaskPattern2(matrix\_copy)

mask\_chosen = 2

ElseIf score\_list.IndexOf(score\_list.Min) = 3 Then

MaskPattern3(matrix\_copy)

mask\_chosen = 3

ElseIf score\_list.IndexOf(score\_list.Min) = 4 Then

MaskPattern4(matrix\_copy)

mask\_chosen = 4

ElseIf score\_list.IndexOf(score\_list.Min) = 5 Then

MaskPattern5(matrix\_copy)

mask\_chosen = 5

ElseIf score\_list.IndexOf(score\_list.Min) = 6 Then

MaskPattern6(matrix\_copy)

mask\_chosen = 6

ElseIf score\_list.IndexOf(score\_list.Min) = 7 Then

MaskPattern7(matrix\_copy)

mask\_chosen = 7

End If

Label\_Statistics.Text = Label\_Statistics.Text & "Mask Pattern Chosen: " & mask\_chosen & vbCr

Array.Copy(matrix\_copy, masked\_matrix, masked\_matrix.Length)

masked\_matrix = FormatAndMaskInfo(masked\_matrix, mask\_chosen) 'Add the Format and Masking information

Return masked\_matrix 'Return the masked matrix once evaluation and selection of mask has been carried out

End Function

Public Function MaskPattern0(ByRef matrix\_copy(,) As Integer)

Dim score As Integer = 0

'((row + column) Mod 2 == 0)

For a = 0 To 20

For b = 0 To 20

If (matrix\_copy(b, a) = 0) Or (matrix\_copy(b, a) = 1) Then 'Only checking data/error bits

If ((a + b) Mod 2) = 0 Then 'Check if formula for this mask pattern = 0

If matrix\_copy(b, a) = 0 Then 'If current bit is 0, change to 1

matrix\_copy(b, a) = 1

ElseIf matrix\_copy(b, a) = 1 Then 'If current bit is 1, change to 0

matrix\_copy(b, a) = 0

End If

End If

End If

Next

Next

score = EvalutingMask(matrix\_copy)

Return score

End Function

Public Function MaskPattern1(ByRef matrix\_copy(,) As Integer)

Dim score As Integer = 0

'((row) Mod 2 == 0)

For a = 0 To 20

For b = 0 To 20

If (matrix\_copy(b, a) = 0) Or (matrix\_copy(b, a) = 1) Then 'Only checking data/error bits

If ((a) Mod 2) = 0 Then 'Check if formula for this mask pattern = 0

If matrix\_copy(b, a) = 0 Then 'If current bit is 0, change to 1

matrix\_copy(b, a) = 1

ElseIf matrix\_copy(b, a) = 1 Then 'If current bit is 1, change to 0

matrix\_copy(b, a) = 0

End If

End If

End If

Next

Next

score = EvalutingMask(matrix\_copy)

Return score

End Function

Public Function MaskPattern2(ByRef matrix\_copy(,) As Integer)

Dim score As Integer = 0

'((column) Mod 3 == 0)

For a = 0 To 20

For b = 0 To 20

If (matrix\_copy(b, a) = 0) Or (matrix\_copy(b, a) = 1) Then 'Only checking data/error bits

If ((b) Mod 3) = 0 Then 'Check if formula for this mask pattern = 0

If matrix\_copy(b, a) = 0 Then 'If current bit is 0, change to 1

matrix\_copy(b, a) = 1

ElseIf matrix\_copy(b, a) = 1 Then 'If current bit is 1, change to 0

matrix\_copy(b, a) = 0

End If

End If

End If

Next

Next

score = EvalutingMask(matrix\_copy)

Return score

End Function

Public Function MaskPattern3(ByRef matrix\_copy(,) As Integer)

Dim score As Integer = 0

'((row + column) Mod 3 == 0)

For a = 0 To 20

For b = 0 To 20

If (matrix\_copy(b, a) = 0) Or (matrix\_copy(b, a) = 1) Then 'Only checking data/error bits

If ((a + b) Mod 3) = 0 Then 'Check if formula for this mask pattern = 0

If matrix\_copy(b, a) = 0 Then 'If current bit is 0, change to 1

matrix\_copy(b, a) = 1

ElseIf matrix\_copy(b, a) = 1 Then 'If current bit is 1, change to 0

matrix\_copy(b, a) = 0

End If

End If

End If

Next

Next

score = EvalutingMask(matrix\_copy)

Return score

End Function

Public Function MaskPattern4(ByRef matrix\_copy(,) As Integer)

Dim score As Integer = 0

'((floor(row / 2) + floor(column / 3)) mod 2) = 0

For a = 0 To 20

For b = 0 To 20

If (matrix\_copy(b, a) = 0) Or (matrix\_copy(b, a) = 1) Then 'Only checking data/error bits

If ((Math.Floor(a / 2) + Math.Floor(b / 3) Mod 2) = 0) Then 'Check if formula for this mask pattern = 0

If matrix\_copy(b, a) = 0 Then 'If current bit is 0, change to 1

matrix\_copy(b, a) = 1

ElseIf matrix\_copy(b, a) = 1 Then 'If current bit is 1, change to 0

matrix\_copy(b, a) = 0

End If

End If

End If

Next

Next

score = EvalutingMask(matrix\_copy)

Return score

End Function

Public Function MaskPattern5(ByRef matrix\_copy(,) As Integer)

Dim score As Integer = 0

'((row \* column) mod 2) + ((row \* column) mod 3) = 0

For a = 0 To 20

For b = 0 To 20

If (matrix\_copy(b, a) = 0) Or (matrix\_copy(b, a) = 1) Then 'Only checking data/error bits

If (((a \* b) Mod 2) + ((a \* b) Mod 3) = 0) Then 'Check if formula for this mask pattern = 0

If matrix\_copy(b, a) = 0 Then 'If current bit is 0, change to 1

matrix\_copy(b, a) = 1

ElseIf matrix\_copy(b, a) = 1 Then 'If current bit is 1, change to 0

matrix\_copy(b, a) = 0

End If

End If

End If

Next

Next

score = EvalutingMask(matrix\_copy)

Return score

End Function

Public Function MaskPattern6(ByRef matrix\_copy(,) As Integer)

Dim score As Integer = 0

'( ((row \* column) mod 2) + ((row \* column) mod 3) ) mod 2 = 0

For a = 0 To 20

For b = 0 To 20

If (matrix\_copy(b, a) = 0) Or (matrix\_copy(b, a) = 1) Then 'Only checking data/error bits

If (((((a \* b) Mod 2) + ((a \* b) Mod 3)) Mod 2) = 0) Then 'Check if formula for this mask pattern = 0

If matrix\_copy(b, a) = 0 Then 'If current bit is 0, change to 1

matrix\_copy(b, a) = 1

ElseIf matrix\_copy(b, a) = 1 Then 'If current bit is 1, change to 0

matrix\_copy(b, a) = 0

End If

End If

End If

Next

Next

score = EvalutingMask(matrix\_copy)

Return score

End Function

Public Function MaskPattern7(ByRef matrix\_copy(,) As Integer)

Dim score As Integer = 0

'( ((row \* column) mod 2) + ((row \* column) mod 3) ) mod 2 = 0

For a = 0 To 20

For b = 0 To 20

If (matrix\_copy(b, a) = 0) Or (matrix\_copy(b, a) = 1) Then 'Only checking data/error bits

If (((((a + b) Mod 2) + ((a \* b) Mod 3)) Mod 2) = 0) Then 'Check if formula for this mask pattern = 0

If matrix\_copy(b, a) = 0 Then 'If current bit is 0, change to 1

matrix\_copy(b, a) = 1

ElseIf matrix\_copy(b, a) = 1 Then 'If current bit is 1, change to 0

matrix\_copy(b, a) = 0

End If

End If

End If

Next

Next

score = EvalutingMask(matrix\_copy)

Return score

End Function

Public Function EvalutingMask(ByVal matrix\_copy(,) As Integer)

Dim score As Integer = 0

Dim condition\_1\_score As Integer = 0

Dim condition\_2\_score As Integer = 0

Dim condition\_3\_score As Integer = 0

Dim condition\_4\_score As Integer = 0

Dim break As Boolean = False

Dim dark\_module\_count As Integer = 0

Dim ratio\_dark\_module As Integer = 0

Dim round\_down As Integer = 0

Dim round\_up As Integer = 0

Dim pattern\_1\_count As Integer = 0

Dim pattern\_2\_count As Integer = 0

Dim CurrentModule As New CurrentModule

'-----Condition 1-----

For c = 0 To 20 'Check for Black Horizontal

For a = 0 To 16

If CurrentModule.IsBlack(matrix\_copy, a, c) = True And

CurrentModule.IsBlack(matrix\_copy, a + 1, c) = True And

CurrentModule.IsBlack(matrix\_copy, a + 2, c) = True And

CurrentModule.IsBlack(matrix\_copy, a + 3, c) = True And

CurrentModule.IsBlack(matrix\_copy, a + 4, c) = True Then

condition\_1\_score += 3

For b = (a + 4) + 1 To 20

If CurrentModule.IsBlack(matrix\_copy, b, c) = True Then

condition\_1\_score += 1

a = b

Else

a = b

b = 20

End If

Next

End If

Next

Next

For c = 0 To 20 'Check for White Horizontal

For a = 0 To 16

If CurrentModule.IsWhite(matrix\_copy, a, c) = True And

CurrentModule.IsWhite(matrix\_copy, a + 1, c) = True And

CurrentModule.IsWhite(matrix\_copy, a + 2, c) = True And

CurrentModule.IsWhite(matrix\_copy, a + 3, c) = True And

CurrentModule.IsWhite(matrix\_copy, a + 4, c) = True Then

condition\_1\_score += 3

For b = (a + 4) + 1 To 20

If CurrentModule.IsBlack(matrix\_copy, b, c) = True Then

condition\_1\_score += 1

a = b

Else

a = b

b = 20

End If

Next

End If

Next

Next

For c = 0 To 20 'Check for Black Vertical

For a = 0 To 16

If CurrentModule.IsBlack(matrix\_copy, c, a) = True And

CurrentModule.IsBlack(matrix\_copy, c, a + 1) = True And

CurrentModule.IsBlack(matrix\_copy, c, a + 2) = True And

CurrentModule.IsBlack(matrix\_copy, c, a + 3) = True And

CurrentModule.IsBlack(matrix\_copy, c, a + 4) = True Then

condition\_1\_score += 3

For b = (a + 4) + 1 To 20

If CurrentModule.IsBlack(matrix\_copy, c, b) = True Then

condition\_1\_score += 1

a = b

Else

a = b

b = 20

End If

Next

End If

Next

Next

For c = 0 To 20 'Check for White Vertical

For a = 0 To 16

If CurrentModule.IsWhite(matrix\_copy, c, a) = True And

CurrentModule.IsWhite(matrix\_copy, c, a + 1) = True And

CurrentModule.IsWhite(matrix\_copy, c, a + 2) = True And

CurrentModule.IsWhite(matrix\_copy, c, a + 3) = True And

CurrentModule.IsWhite(matrix\_copy, c, a + 4) = True Then

condition\_1\_score += 3

For b = (a + 4) + 1 To 20

If CurrentModule.IsBlack(matrix\_copy, c, b) = True Then

condition\_1\_score += 1

a = b

Else

a = b

b = 20

End If

Next

End If

Next

Next

Label26.Text = condition\_1\_score

'-----Condition 1-----

'-----Condition 2-----

For b = 0 To 18 Step 2 'Checking for 2x2 blocks of the same colour - Excluding Left and Bottom

For a = 1 To 19

If CurrentModule.IsBlack(matrix\_copy, a, b) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 1) = True And

CurrentModule.IsBlack(matrix\_copy, a + 1, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 1, b + 1) = True Then

condition\_2\_score += 3

End If

If CurrentModule.IsWhite(matrix\_copy, a, b) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 1) = True And

CurrentModule.IsWhite(matrix\_copy, a + 1, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 1, b + 1) = True Then

condition\_2\_score += 3

End If

Next

Next

For b = 0 To 18 Step 2 'Checking for 2x2 blocks of the same colour - Excluding Right and Bottom

For a = 0 To 18

If CurrentModule.IsBlack(matrix\_copy, a, b) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 1) = True And

CurrentModule.IsBlack(matrix\_copy, a + 1, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 1, b + 1) = True Then

condition\_2\_score += 3

End If

If CurrentModule.IsWhite(matrix\_copy, a, b) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 1) = True And

CurrentModule.IsWhite(matrix\_copy, a + 1, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 1, b + 1) = True Then

condition\_2\_score += 3

End If

Next

Next

For b = 1 To 19 Step 2 'Checking for 2x2 blocks of the same colour - Excluding Top and Left

For a = 1 To 19

If CurrentModule.IsBlack(matrix\_copy, a, b) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 1) = True And

CurrentModule.IsBlack(matrix\_copy, a + 1, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 1, b + 1) = True Then

condition\_2\_score += 3

End If

If CurrentModule.IsWhite(matrix\_copy, a, b) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 1) = True And

CurrentModule.IsWhite(matrix\_copy, a + 1, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 1, b + 1) = True Then

condition\_2\_score += 3

End If

Next

Next

For b = 1 To 19 Step 2 'Checking for 2x2 blocks of the same colour - Excluding Top and Right

For a = 0 To 18

If b = 17 Then

b = 18

End If

If CurrentModule.IsBlack(matrix\_copy, a, b) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 1) = True And

CurrentModule.IsBlack(matrix\_copy, a + 1, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 1, b + 1) = True Then

condition\_2\_score += 3

End If

If CurrentModule.IsWhite(matrix\_copy, a, b) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 1) = True And

CurrentModule.IsWhite(matrix\_copy, a + 1, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 1, b + 1) = True Then

condition\_2\_score += 3

End If

Next

Next

'-----Condition 2-----

'-----Condition 3-----

For b = 0 To 20 'Iterating through the matrix, checking rows for both patterns

For a = 0 To 10

If CurrentModule.IsWhite(matrix\_copy, a + 0, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 1, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 2, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 3, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 4, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 5, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 6, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 7, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 8, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 9, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 10, b) = True Then

pattern\_2\_count += 1

ElseIf CurrentModule.IsBlack(matrix\_copy, a + 0, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 1, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 2, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 3, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 4, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 5, b) = True And

CurrentModule.IsBlack(matrix\_copy, a + 6, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 7, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 8, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 9, b) = True And

CurrentModule.IsWhite(matrix\_copy, a + 10, b) = True Then

pattern\_1\_count += 1

End If

Next

Next

For a = 0 To 20 'Iterating through the matrix, checking columns for both patterns

For b = 0 To 10

If CurrentModule.IsWhite(matrix\_copy, a, b + 0) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 1) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 2) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 3) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 4) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 5) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 6) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 7) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 8) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 9) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 10) = True Then

pattern\_2\_count += 1

ElseIf CurrentModule.IsBlack(matrix\_copy, a, b + 0) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 1) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 2) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 3) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 4) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 5) = True And

CurrentModule.IsBlack(matrix\_copy, a, b + 6) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 7) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 8) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 9) = True And

CurrentModule.IsWhite(matrix\_copy, a, b + 10) = True Then

pattern\_1\_count += 1

End If

Next

Next

condition\_3\_score = 40 \* (pattern\_1\_count + pattern\_2\_count)

'-----Condition 3-----

'-----Condition 4-----

For a = 0 To 20 'Counts the total number of dark modules in the matrix

For b = 0 To 20

If CurrentModule.IsBlack(matrix\_copy, a, b) = True Then

dark\_module\_count += 1

End If

Next

Next

ratio\_dark\_module = (dark\_module\_count / matrix\_copy.Length) \* 100 'Caclulates the ratio of dark modules to total modules

round\_down = (CLng(ratio\_dark\_module / 5) \* 5)

If ratio\_dark\_module Mod 5 = 0 Then 'Finds the previous and next multiples of 5

If round\_down = ratio\_dark\_module Then

round\_down -= 5

round\_up = round\_down + 10

Else

round\_up = round\_down + 5

End If

ElseIf round\_down < ratio\_dark\_module Then

round\_up = round\_down + 5

ElseIf round\_down > ratio\_dark\_module Then

round\_up = round\_down

round\_down -= 5

End If

round\_down = (Math.Abs(round\_down - 50)) / 5

round\_up = (Math.Abs(round\_up - 50)) / 5

If round\_down > round\_up Then

condition\_4\_score = round\_up \* 10

Else

condition\_4\_score = round\_down \* 10

End If

'-----Condition 4-----

score = condition\_1\_score + condition\_2\_score + condition\_3\_score + condition\_4\_score 'Accumulates penalty scores from the 4 conditions

Return score

End Function

Public Function FormatAndMaskInfo(ByRef masked\_matrix(,) As Integer, ByVal mask\_chosen As Integer)

Dim FormatAndMaskInfoString As String = ""

If mask\_chosen = 0 Then 'Format and Mask Info corresponding to each mask chosen

FormatAndMaskInfoString = "101010000010010"

ElseIf mask\_chosen = 1 Then

FormatAndMaskInfoString = "101000100100101"

ElseIf mask\_chosen = 2 Then

FormatAndMaskInfoString = "101111001111100"

ElseIf mask\_chosen = 3 Then

FormatAndMaskInfoString = "101101101001011"

ElseIf mask\_chosen = 4 Then

FormatAndMaskInfoString = "100010111111001"

ElseIf mask\_chosen = 5 Then

FormatAndMaskInfoString = "100000011001110"

ElseIf mask\_chosen = 6 Then

FormatAndMaskInfoString = "100111110010111"

ElseIf mask\_chosen = 7 Then

FormatAndMaskInfoString = "100101010100000"

End If

For a = 0 To 5 'Following Iteration statements place the Format and Mask Info into the matrix

masked\_matrix(a, 8) = CInt(FormatAndMaskInfoString.Substring(a, 1))

masked\_matrix(8, 5 - a) = CInt(FormatAndMaskInfoString.Substring(a + 9, 1))

Next

For a = 13 To 20

masked\_matrix(a, 8) = CInt(FormatAndMaskInfoString.Substring(a - 6, 1))

Next

For a = 0 To 6

masked\_matrix(8, 20 - a) = CInt(FormatAndMaskInfoString.Substring(a, 1))

Next

masked\_matrix(7, 8) = CInt(FormatAndMaskInfoString.Substring(6, 1))

masked\_matrix(8, 8) = CInt(FormatAndMaskInfoString.Substring(7, 1))

masked\_matrix(8, 7) = CInt(FormatAndMaskInfoString.Substring(8, 1))

Return masked\_matrix

End Function

End Class

# Evaluation

## Conversation with Supervisor

**Me:** “Did you find that my project was an effective implementation of a QR Code generator?

**Supervisor:** “In most cases, yes it was very effective as a QR Code generator. It quickly displayed a QR Code after I had entered some data which, in every case I tested, was able to be scanned and the data I entered was able to be extracted, regardless of the orientation of the actual symbol.”

**Me:** “Did you find any limitations with my program?”

**Supervisor:** “I did find that it was quite limiting in terms of the maximum number of characters that a user can enter was 20. As this is an investigation though that isn’t a major issue as you have demonstrated a proof of concept. If you were to take this further I would recommend increasing the capacity of the QR Codes as it would allow more usability and more opportunity for testing.”

**Me:** “Did you find that the error correction worked?”

**Supervisor:** “Yes, in all cases I found that I was able to cover a portion of the symbol and still scan it fully, which means that you correctly and effectively implemented error correction.”

## Meeting System Objectives

All of my system objectives, which were outlined in the analysis section, have been met.

1. Accepting/Validating user entered data – The user is able to enter data that they wish to be encoded into a QR Code. If the data passes validation, the program continues and a QR Code is generated. If the data does not pass validation, the program informs the user of their error and the user is able to re-enter data. Evidence of this can be seen in the Testing section and a detailed breakdown of the validation process can be seen in the Design section
2. Converting user entered data into Data Codewords – Once validation has been passed, the data is converted into data codewords and a data bit stream is constructed. A detailed breakdown of this can be seen in the Design section
3. Generating Error Correction Codewords – The Data Codewords are subject to Reed-Solomon error correction generation methods in which Galois Field Arithmetic and Polynomial Division steps are repeated until the desired number of Error Correction Codewords are generated. A detailed breakdown of the error correction codeword generation process can be seen in the Design section
4. Constructing Final Bit Steam – The Data Codewords and the Error Correction Codewords are concatenated, ready to be encoded into a QR Code symbol
5. Applying the most appropriate Mask Pattern – The program tests each masking pattern and calculates a score subject to the four conditions outlined in the Masking sub-section within the Design section. The mask pattern with the lowest score is then finally applied to the QR Code symbol. The point of masking is to reduce large areas of White/Black modules and from the QR Codes that the program generates, it can be seen that none of them feature any large areas of White/Black areas, meaning that the mask evaluation/application functions are working correctly.
6. Displaying a scan-able QR Code Graphic – Once data has been entered by the user, a QR Code is constructed within a few seconds which the user can scan. I have tested a multitude of data (see Testing section) and all the produced QR Codes have been able to be quickly scanned by my phone, extracting the original data, meaning the program is working correctly
7. Error Correction Effectiveness – The QR Code that is produced is still able to be scanned even if a small portion of the displayed symbol is covered. A detailed breakdown of this can be seen in the Testing section.

## Desirable Additions

Additional features which are optional but would increase the feature set of the program. As this is an investigation these aren’t necessary but maybe something that could be added in the future:

1. Dynamically chose a version based on the amount of data entered. A version should be selected that isn’t small enough such that it wouldn’t hold all the data and not too large that there would be empty space within the matrix that would have to be rectified by masking methods
2. Allow more than one type of data to be entered. As each different mode of data (numerical, alphanumerical, Kanji etc.) is encoded slightly differently and as such produces a slightly different amount of binary bits, it would be necessary to change several sections of the program to accommodate for this type of mode mixing
3. Increase the Version, allowing more raw input characters to be added.
4. Increase the Error Correction Level which would allow the QR Code to sustain more damage and still be readable by a scanner