Project due date: Sunday 15th September 10pm.

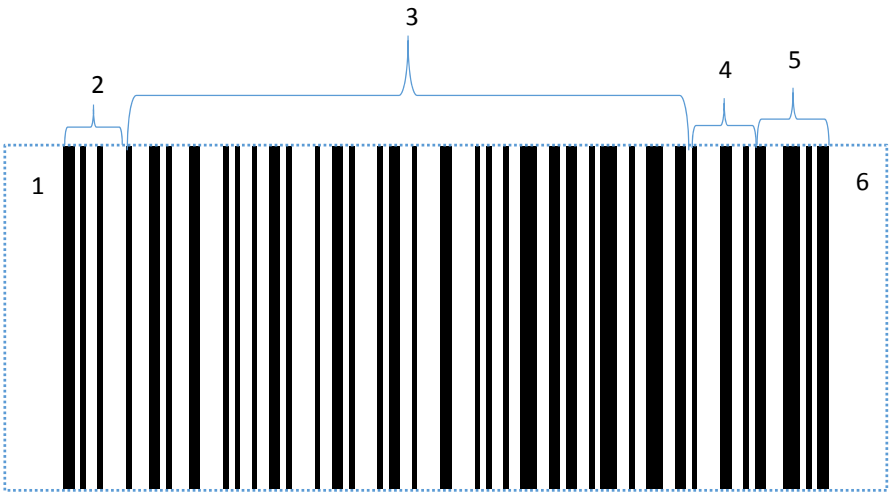
Barcode your name (and anything else!)

Introduction

Barcodes provide a simple way of encoding information in a machine-readable format. Barcodes can be easily read using a device such as an optical scanner or a smartphone (if your phone has a camera and an appropriate reader app installed). There are many different barcode standards. Some standards just encode numerical data (e.g. EAN-13 codes, the 13 digit codes which appear on almost every product that you purchase). Other coding schemes can be used to encode text (e.g. QR codes, which are commonly used to encode web addresses). For this project we will use the code 128 standard (in particular code set B), which can be used to encode text.

For detailed information on code 128, including a table of values and codes, see the Wikipedia page: http://en.wikipedia.org/wiki/Code_128

The following image shows the different regions of a code 128 barcode.



1. Quiet Zone: white space which is at least ten times wider than the narrowest bar
2. Start Character: A pattern of six bars that identifies which code128 code set is being used; A, B or C. **We will use only code set B**.
3. Encoded Data: each character in the phrase is encoded using a pattern of six bars (bars alternate between black and white). Different bar widths are used to encode different characters.
4. Check Character: a checksum value encoded with the appropriate six bar pattern
5. Stop Character: a special pattern of seven bars that identifies the end of the barcode
6. Quiet Zone

Worked example: Encoding the message 'Hi'

To understand each of the six regions better, we will work through the problem of generating a barcode pattern by hand, for the message 'Hi'. We will assume the narrowest bar will be exactly one pixel wide. We will use 1 to represent a black pixel and 0 to represent a white pixel. One row of the barcode image will then correspond to a pattern of 1s and 0s.

1. Quiet Zone

The barcode must start with a quiet zone of white space at least ten times wider than the narrowest bar. As we are choosing the narrowest bar to be one pixel wide, we need ten white pixels:

0000000000

Visually this just equates to blank white space:

0	0	0	0	0	0	0	0	0	0

Or without the 0s:

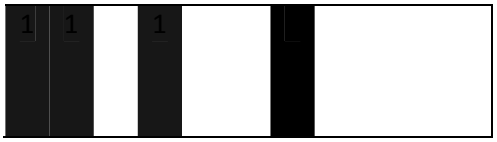
2. Start Character

Code128 actually consists of three different code sets (A, B and C). The start character is a pattern of six bars that indicates which code set is to be used (and it is even possible to switch between the three code sets within the same barcode by using appropriate six bar patterns). To simplify things we will stick to using only code set B. If we use one pixel as the width of one bar then the start pattern for code set B can be represented as follows: 11010010000

Visually it would look like this:

1	1	0	1	0	0	1	0	0	0	0

Or without the 1s and 0s:



3. Encoded Data

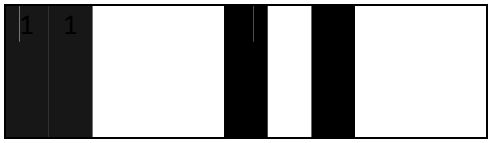
We want to encode the message ‘Hi’. Each character in our message will be encoded by a pattern of six bars. We need to look up in a table the pattern corresponding to each character (see the table on Wikipedia). We also need to look up the corresponding code128B value for each character, as this value will be used to calculate the checksum which follows the encoded data.

From the table of data, we can find that the letter ‘H’ corresponds to the code128B value 40 and the pattern: 11000101000

Visually it would look like this:

1	1	0	0	0	1	0	1	0	0	0

Or without the 1s and 0s:



From the table of data, we can find that the letter 'i' corresponds to the code128B value 73 and the pattern 10000110100

Visually it would look like this:

1	0	0	0	0	1	1	0	1	0	0

Or without the 1s and 0s:

4. Check Character

The check character is calculated using the code128B values for each character in the message (including the start character). We use the following algorithm to calculate a code128B check character.

1. Generate a set of weighted values by multiplying the code128B value for each character in the message by its corresponding index in the message
2. Calculate a total by adding the value 104 to the sum of all the weighted values
3. Find the remainder after dividing the total by 103, this is your checksum

The code128B values for 'Hi' were 40 and 73 so our checksum calculation is as follows

1. Generate weighted values: 40x1, 73x2
2. Calculate total: 104 + 40 + 146
3. Find remainder: 290/103 = 2 with a remainder of 84

Hence the checksum for the sequence 40, 73 is **84**. Note we always divide by 103. We take the remainder so that we have a value that can be encoded (the sum 290 could not be encoded using a pattern for one character whereas the remainder of 84 can be encoded using a pattern)

From the table of data, we can find that the code128B value 84 corresponds to the pattern: 10011110100

Visually it would look like this:

1	0	0	1	1	1	1	0	1	0	0

Or without the 1s and 0s:

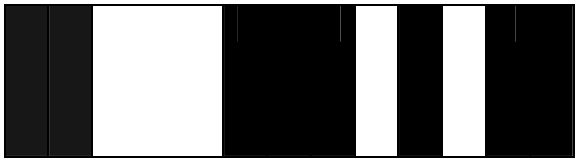
5. Stop Character

The stop character is a pattern of seven bars that indicates the end of the barcode. Seven bars are used, rather than the usual six, as the sixth bar is white so a seventh black bar is needed so that the width of the sixth bar can be read. If we use one pixel as the width of one bar then the stop pattern can be represented as follows: 1100011101011

Visually it would look like this:

1	1	0	0	0	1	1	1	0	1	0	1	1

Or without the 1s and 0s:



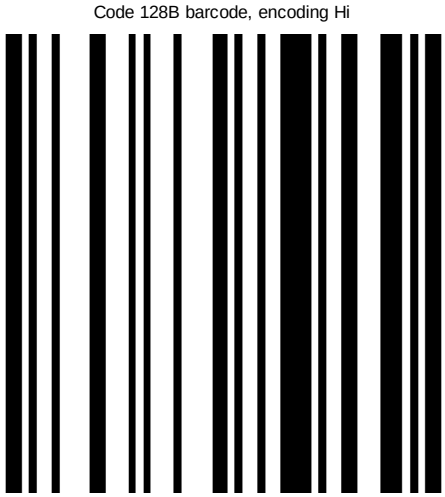
6. Quiet Zone

The barcode must end with a quiet zone of white space at least ten times wider than the narrowest bar. As we are choosing the narrowest bar to be one pixel wide, we need ten white pixels:

0000000000

Final Assembly

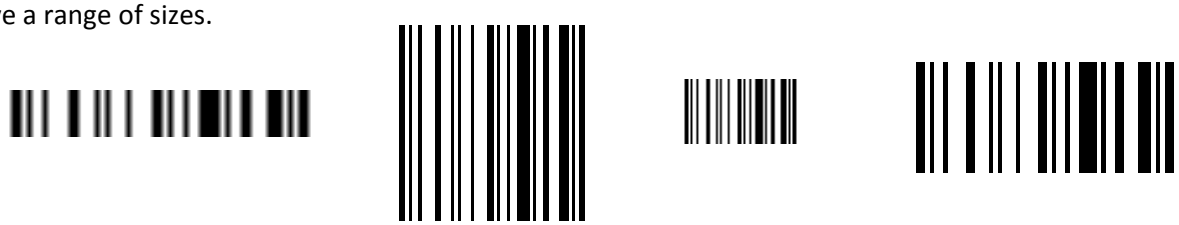
Putting each pattern one after the other gives us our final barcode:



A note on bar width

In the above worked example we assumed that the narrowest bar was exactly one pixel wide. **We will make the same assumption for parts A and B of the project**. Using this approach simplifies both generating a 2D barcode and reading it.

In practice, a photo or scan of a barcode is not guaranteed to have exactly one pixel for the narrowest bar. Consider the following range of barcodes which ALL encode the same message, yet have a range of sizes.



Part C of the project is concerned with reading barcodes like those above, where it is possible the narrowest bar may be many pixels in width.

Part A – Generating a Code 128B barcode

You have been supplied with a short script called `GenerateBarcode.m`

This script asks the user to enter a phrase to encode and then calls two functions in order to generate a barcode, before displaying the barcode image and writing it out to a file.

Before you can successfully run `GenerateBarcode` you will need to implement the two functions it calls:

- `CreateBarcodePattern`
- `CreateBarcodeImage`

The ultimate aim of Part A is to code these functions, so that you can generate a barcode for any user-specified phrase. Remember, however, that when approaching a complicated problem it is best to first break it down into smaller more manageable components. Rather than trying to write `CreateBarcodePattern` straight away, we will first develop some helper functions, which will then be used by your `CreateBarcodePattern` function. Finally you will write your `CreateBarcodeImage` function.

Recall from the introduction that if we wish to create a barcode that encodes a string there are three fundamental operations we need to be able to perform:

- look up the code128B value associated with any given character
- look up the pattern associated with a code128B value
- calculate a checksum for a sequence of code128B values

We will write one function for each of these fundamental operations. These functions will then be used as building blocks (they will be called by your `CreateBarcodePattern` function). We start with the simplest function to implement: looking up a pattern associated with a value.

Task A.1: Write the `GetPatternForValue` function

`GetPatternForValue` will take a single input (a code 128B value **number**) and return a **string** as the single output (the code128B pattern of 1s and 0s associated with the given value). Your `GetPatternForValue` function will need to use the information stored in the code128B **cell array** (which can be loaded from the supplied **code128B.mat** file)

Some example functions calls are shown below:

Call	Expected output
`p = GetPatternForValue(0)`	`p = 11011001100`
`p = GetPatternForValue(40)`	`p = 11000101000`
`p = GetPatternForValue(106)`	`p = 1100011101011`

Task A.2: Write the `GetValueForChar` function

`GetValueForChar` will take a single input (a **string** containing one character) and return a single output (the code128B value **number** associated with the string). Your `GetValueForChar` function will need to use the information stored in the code128B **cell array** (which can be loaded from the supplied **code128B.mat** file)

Example functions calls are shown below:

Call	Expected output
`v = GetValueForChar('H')`	`v = 40`
`v = GetValueForChar('e')`	`v = 69`
`v = GetValueForChar('<')`	`v = 28`
`v = GetValueForChar(' ')`	`v = 0`

Task A.3: Write the Code128BChecksum function

`Code128BChecksum` will take a single input (an array of **numerical** values) and return a single output **number** (the checksum value, calculated according to the code128B checksum algorithm).

The checksum for a sequence of code128B values can be found using the following algorithm:

1. Generate a set of weighted values by multiplying every original value by its corresponding index in the array of numerical code128B values
2. Calculate a total by adding the value 104 to the sum of all the weighted values
3. Find the remainder after dividing the total by 103, this is your checksum

You should work through this algorithm by hand a few times. Here is an example of finding the checksum for the values 10, 20, 30:

1. Generate weighted sequence: 10x1, 20x2, 30x3
2. Calculate total: 104 + 10 + 40 + 90
3. Find remainder: 244/103 = 2 with a remainder of 38

Hence the checksum for the sequence 10, 20, 30 is **38**.

Example functions calls are shown below:

Call	Expected output
`cs = Code128BChecksum([10 20 30])`	`cs = 38`
`cs = Code128BChecksum([4 3 2 1])`	`cs = 21`
`cs = Code128BChecksum([40 69 69 76 76 79])`	`cs = 102`

Task A.4: Write the `CreateBarcodePattern` function

`CreateBarcodePattern` will take a single input (a **string** containing a phrase) and return a single output (a **string** of 1s and 0s representing the barcode pattern).

Your barcode pattern will need to contain all 6 sections required for a valid code 128B barcode:

1. Quiet Zone (10 zeros in a row)
2. Start Character for code set B (11010010000)
3. Encoded Data (the appropriate pattern for each character in your phrase)
4. Check Character (the appropriate pattern for the checksum value)
5. Stop Character (1100011101011)
6. Quiet Zone (10 zeros in a row)

You will need to use your `GetValueForChar` function to get the value for each character in your phrase (so that you can calculate a checksum using `Code128BChecksum`). You should use `GetPatternForValue` to convert each value into the appropriate pattern for the barcode.

Here is an example of CreateBarcodePattern being called:

```
p = CreateBarcodePattern('Hi')
```

```
p =
```

00000000001101001000011000101000100001101001001111010011000111010110000000000

As it is time consuming to manually check the output of `CreateBarcodePattern`, a test script called `TestCreateBarcodePattern` will be supplied. To use it, simply put it in the same directory as your `CreateBarcodePattern` function and then run the script.

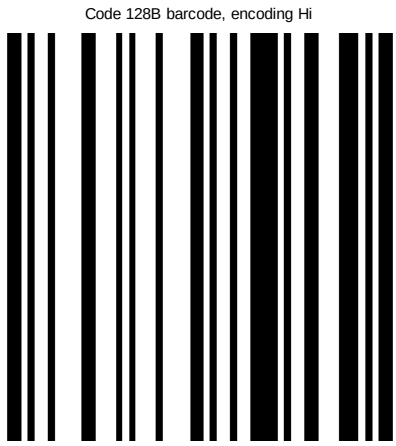
Task A.5: Write the `CreateBarcodeImage` function

`CreateBarcodeImage` will take a single input (a **string** of 1s and 0s representing the barcode pattern) and return a single output (a 2D array of **numbers** representing a greyscale image of the barcode that corresponds to the input pattern). Your image should have **60 rows** and the number of columns should match the length of the input string.

You can think of the barcode pattern string as representing a single row of an image, where a 0 corresponds to a white pixel and a 1 corresponds to a black pixel.

IMPORTANT: remember that when creating a greyscale image in a 2D array, shades of grey are represented by an intensity value between 0 and 1. *A white pixel has the intensity value 1 while a black pixel has the intensity value 0.* This is the OPPOSITE of the representation used in the pattern string (where a 1 represents black and a 0 represents white).

When you have successfully coded this function, running the supplied `GenerateBarcode` script will produce an image of the encoded phrase. For example, the phrase `'Hi'` encodes to:



You may like to test your image by reading it using a smartphone application that supports reading code128 barcodes (e.g. ZBar barcode reader for iphone). Alternatively once you (or a friend) have implemented the second part of the project, you will be able to test your image by running the `ReadBarcode` script to read your generated barcode. While you should **not** be sending bits of code to each other, feel free to send generated barcodes to each other (all going well you should be able to send messages in the form of barcode images!)

Part B – Reading a code 128B barcode

You have been supplied with a short script called `ReadBarcode.m`

This script asks the user to enter an image filename (of an image that contains a code 128B barcode). It opens the image, scans one line of the image to extract the barcode pattern and then calls the function `ReadPattern` to return the encoded message, before displaying it.

Before you can successfully run `ReadBarcode` you will need to implement the `ReadPattern` function.

Unlike part A, this task has purposefully not been broken down into smaller components and you are not provided with any algorithms. This is to give you practice and experience at doing this yourself. Use the problem solving steps outlined in class to work through the problem of decoding a barcode. It is likely you will want to write some helper functions, as you did in part A.

Task B: Write the `ReadPattern` function

`ReadPattern` will take a single input (a **string** containing a code128B pattern of 1s and 0s). It will return TWO outputs. The first output is a string containing the encoded message. The second output is a Boolean value which is true if the checksum read from the barcode matched the checksum calculated for the message and is false otherwise.

Part C – Handling photos (bonus marks)

In parts A and B we assumed that **the narrowest bar was one pixel wide**. Using this approach simplifies both generating a 2D barcode and reading it, however in practice an image of a barcode is not guaranteed to have exactly one pixel for the narrowest bar. Exactly how many pixels wide the narrowest bar is will depend on many factors including the size of the original barcode, the resolution of the scanner/camera and how far away the scanner/camera was from the barcode. Part C of the project is concerned with reading photos of barcode 128B images, where it is possible the narrowest bar may be many pixels in width.

Task C: Modify the `ReadPattern` function to handle photos of barcodes

Modify your `ReadPattern` function so that it will handle photos of barcodes. It should still have the same inputs and outputs as were specified in Task B. Your modification may include writing helper functions that are called by `ReadPattern` or it may just include modifying the code within `ReadPattern`. It is up to you which approach you take.

You can test your code by taking photos of code128B images and attempting to read them. Some test photos are supplied on cecil.

How the project is marked

A mark schedule will be published two weeks before the due date, outlining exactly how marks will be allocated for each part of the project. You will receive marks both for correctness (does your code work?) and style (is it well written?). Style includes elements such as using sensible variable names, having header comments, commenting the rest of your code, avoiding code repetition and using correct indentation.

Each function will be marked independently for correctness, so even if you can't get everything to work, please do submit the functions you have written. Note it is still possible to get marks for good style, even if your code does not work at all!

How the project is submitted

Submission is done by uploading a zip file to the Aropa website. More information will be provided on how to submit the project to Aropa in a Cecil email announcement. If you want to be out of Auckland over the break and still want to work on the project, that is perfectly fine as long as you have access to a computer with Matlab installed and can access the internet to upload your final submission.

Any questions?

If you have any questions regarding the project please ask them on the class Piazza forum. Remember that you should NOT be posting any of your project code on Piazza, so if your question requires that you include some of your code, make sure that it is posted as a PRIVATE piazza query.