

BotsAndUs – Senior C++ Engineer

Coding Exercise

Overview

This coding exercise is part of the application process for a C++ software engineering role at BotsAndUs. It involves the implementation of a small application which can load a list of ID codes from a file, process the data, and output a specially-formatted PNG image for each ID.

This is not intended to be a difficult and time-consuming task. It's a way for you to demonstrate the approach you take to writing software – how you follow specifications, how you provide documentation, the way you test functionality, and how you make decisions about what to build. It also gives us an opportunity to discuss this process with you afterwards.

This task is closely based on a real-world problem we have previously had to solve for a customer application, and is broadly representative of the kind of tasks you would perform in this role. In this case, the task should be implemented using C++, but there are no specific rules regarding what tools, libraries or frameworks you can use to implement this exercise. We are more interested in your approach to the problem than the specific tools you have experience with already – and choosing an appropriate tool is part of the exercise.

The task

A customer has requested that we use our robot to survey their premises once a day and report the location of various key assets. The customer has an existing installation of remotely updateable, six-character, seven-segment displays (fig. 2), and one of these is attached to each asset.

During normal working hours, these displays show customer-specific information. We have agreed with the customer that we will use these displays outside of working hours to determine the location of their assets. The customer will remotely update each display so that it shows a unique ID code that identifies the asset, then our robot will autonomously drive through the entire premises and use a camera-based OCR (Optical Character Recognition) system to detect each display, read the code, and work out the real-world position of the asset.

The content of each display can be controlled by uploading a specially-crafted PNG image file that contains 256 bits of data encoded as black and white pixels. 48 pixels in this image directly control the state of the segments on the displays, and the rest are reserved for other purposes.

We will be provided with a list of asset IDs, each of which is a numeric value between 0 and

9999, supporting up to 10,000 unique displays. For each of these asset IDs, we will supply the customer with a PNG file that will be uploaded to the display and will program the display to show the asset ID.

We only require four characters to represent each unique ID, and we have decided to use two extra characters to implement error detection – we will include a checksum that we can verify when the display is read, allowing us to discard most codes which have an error in them.

The task to be completed in this exercise is to consume the list of asset IDs and output the corresponding PNG files that will be supplied to the customer.

Functional requirements and specification

Your solution should take the following steps to generate the required output:

- 1. Given a text file containing a list of unique, four-digit numerical asset IDs in a text file format, calculate a two-digit checksum for each ID and add it as a prefix to the ID to generate a six-digit code.**

Each individual asset ID is represented as a 4-digit number between 0 and 9999. A simple checksum can be implemented by using modular arithmetic with a base of 97 to calculate two check digits. The checksum c of a number a with 4 digits can be calculated as:

$$c = (a_1 + (10 * a_2) + (100 * a_3) + (1000 * a_4)) \bmod 97$$

For example, the checksum for the asset ID 1337 can be calculated as follows:

$$(1 + (10 * 3) + (100 * 3) + (1000 * 7)) \bmod 97 = 7331 \bmod 97 = 56$$

This can be prefixed to the original asset ID to give the six-digit checksummed code 561337.

- 2. For each asset ID and checksum, calculate the pattern of bits required to activate the required segments on the display, so that all six characters will be shown.**

Each of the six characters on the display consists of seven segments which can be turned on or off. The ten decimal digits (fig. 1) can be displayed as follows:



Fig. 1 - Decimal digits

Each character is represented by a different 8-bit pattern in which each bit controls one segment. The bits are arranged as follows:

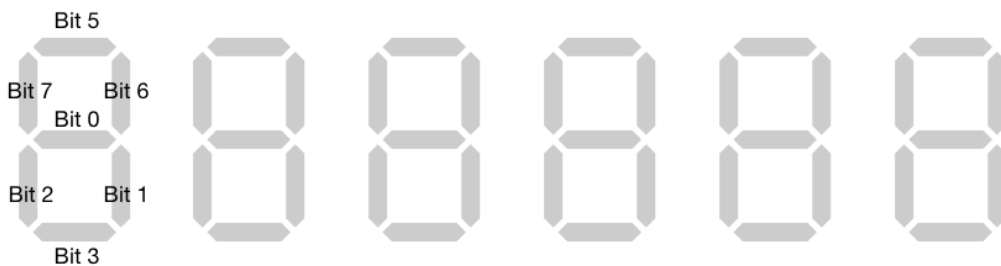


Fig. 2 - Six digit seven segment display

where bit 4 is always set to zero. For example, In this system, the decimal character '5' would be encoded as the bit string 11010101.

3. For each pattern of bits, generate a PNG file that encodes this pattern, and output it to disk with a file name of the original asset ID.

After constructing a pattern of bits for each character, they must be assembled into a PNG file which encodes this information. Your code must generate a valid PNG file which is 256 pixels wide, 1 pixel in height, and uses a 1-bit colour depth.


Each pixel in the PNG file will represent the value of a single bit in a 256-bit pattern – if the pixel is white, the bit is 0, with a 1 bit represented by a black pixel.

Only 48 bits ($6 * 8$) are required to represent all six characters to be displayed, using the bit pattern you constructed in the last step. You should encode this bit pattern into the PNG file by setting the appropriate pixel values, starting with an offset of 8 bits. Bits 0–7 and bits 56–255 are reserved for other uses and should be set to zero.

For example, encoding the checksummed asset ID 561337 as described above would result in the bit pattern:

110101011111010101000010110101101101011001000110

Following the steps above to generate a PNG file, the resulting output would look like the following (very small!) image:

 Fig. 3 - Example encoded PNG image

4. Follow up question (for discussion only - no code required):

After the initial implementation the customer realises that they need to support more than 10,000 unique items using the same 6 digit displays. Describe any possible way(s) you could change the implementation detailed above to support this request.

Non-functional requirements

- You should include appropriate documentation about your code.
- Your code should follow best practices and a consistent coding style.
- Your submission should include whatever test cases you consider appropriate.

Data and deliverables

Included with this document are the following files:

- `lcd-number.png` – a diagram of the segments to be turned on to represent each possible decimal digit.
- `lcd-bits.png` – an illustration demonstrating which segment in each character is controlled by each bit in the PNG.
- `1337.png` – an example output file for the code 1337.
- `test.txt` – a text file containing 10 sample IDs.

You should deliver:

- Your code that implements the process described above.
- Any documentation or test cases you have produced.
- Details about how the code can be built and executed.
- The results of running your code over the supplied list of IDs in the file `test.txt`.

You can submit your results in a Git repository or a ZIP file. We encourage you to reach out to us with any questions you may have about the task before submitting your response.