SCC.111: IMAPCASTER

THE GOAL

This week's lab aims to:

- [ ] Practice working with compound data structures
- [ ] Creating and using strings
- [ ] Solving another problem to meet the spec

PROBLEM STATEMENT

Lancaster is full of surprises. From Roman ruins, buried streams, ghost walks, to gelato bars. The goal of today's task is to practice working with compound data structures to represent 'pins on a map'.

1. Your goal is firstly to declare a data structure suitable to 'record' details relating to a place. Options for this might be a name, latitude and longituide. Another option would be to record the location symbollically, e.g. as a 'what 3 words' reference.
2. Once you have your struct declared, declare an array to enable you to store up to 10 of your favourite locations.
3. Create two new functions: i. add_new_location; and ii. print_locations to work with this array. The format is as follows assuming your structure is called `struct location` then the function signatures would be something like:

```
// Add a new location 'newLoc' to array 'list'.  numLocations is a pointer to a
 ↪  counter
// of how many locations we have.  We return TRUE if we added the location, FALSE if
 ↪  the
// array was full
int add_new_location(struct location newLoc, struct location *list, int
 ↪  *numLocations);
```

```
// Print 'numLocations' number of locations from array 'list'
void print_locations(struct location *list, int numLocations);
```

My output looked like this:

```
My favourite places:
Place 1 is John O'Gaunt at 54.048649, -2.801030
Place 2 is 3 Mariners at 54.051201, -2.802400
```

4. Choose 5-10 locations around Lancaster you'd like to model. For example, using Open Streetmap or What 3 words. Write a main function that calls ' `add_new_location` ' for each of the locations.

Don't forget to show us your solutions to get feedback on your code!

SCC121: QUEUES

Assume we have a queue-like representation using the Element structure discussed in class, reproduced here for convenience.

```
Element {
    data;
    next;
};
```

1. Assume the queue is initially empty. Suppose the following sequence of operation occurs on the queue. State what each operation does from the user perspective and the resulting queue after each operation. Below E and D stand for enqueue and dequeue, respectively.

```
E(5), E(10), D(), E(15), D(), D(), D(), E(10), E(5)
```

You do need not draw pictures. You can represent a queue by using head, nil, -> (for pointers), and the data. So for example, if the queue, starting head-first, has data a,b,c,d, you can represent it as:

```
head -> a -> b -> c -> d -> nil
```

O2. In class, we saw the pseudocode for enqueue. Write the pseudocode for dequeue.

3. Write pseudocode to calculate the size of a queue given the representation above.

4. Represent Element using the C language struct facility. Implement enqueue and dequeue in C and write some driver code (in the main) to test your implementation.
   Hints: You are provided a queue-incomplete.c which contains

- interactive driver code (in the main) for testing
- declaration of the element structure in C
- declaration of the head pointer
- and a function for printing the queue

- empty enqueue and dequeue methods

Your job is to implement in C the logic of the enqueue and dequeue operators. You are given in the comments above enqueue() how to use malloc to create a new element in memory. Remember to use free(ptr), where ptr points to the dequeued element in dequeue().

SCC.131: HANDS-ON WITH THE BBC MICRO:BIT

LEARNING OBJECTIVES

By the end of this week's SCC.131 lab, you are expected to:

- Familiarise yourselves with the Microsoft MakeCode online editor and the toolboxes that are available for micro:bit.
- Combine blocks from different toolboxes in MakeCode to build programs for a micro:bit device.
- Exploit the built-in short-range radio capability of micro:bit to communicate with other micro:bit devices.
- Execute and test your code in MakeCode, which can emulate sensor data and user interactions.
- Connect micro:bit to your computer, upload your code onto micro:bit and see your code running.
- Download and install required software on your computer that will allow you to program micro:bit in C in Week 8.

MAIN TASKS

This section presents four tasks, three of which require the use of Microsoft MakeCode to program micro:bit.

- If you *are not familiar* with MakeCode or micro:bit, you are strongly advised to start from Task 1.
- If you *have experience* in navigating MakeCode *and* block-coding for micro:bit, you can start from Task 2.

To start, go to the Microsoft MakeCode website for micro:bit and click the **Sign In** button at the top-right corner of the page. Select **Continue with Microsoft** and use your Lancaster University credentials (email address and password) to sign in. If you do not sign it, projects that you develop in MakeCode will not be saved.

Task 1: Flashing Heart

Find the **Tutorials** section and select the **Flashing Heart** project. Choose **Blocks** to start the tutorial. This tutorial will familiarise you with the MakeCode environment and with two blocks (forever and show leds

) in the **Basic** toolbox. Follow the on-screen instructions and create a flashing heart animation.

When you complete *Step 4* of the tutorial, the micro:bit simulator on the left-hand side of the screen should display a flashing heart.

At *Step 5*, click the **Download** button located at the bottom-left corner of the browser. Contrary to the on-screen instructions, do NOT connect your micro:bit to your computer; select **Next** and then **Download as File**, as our aim is NOT to pair the micro:bit to the browser but manually transfer the code to micro:bit. Save the file "microbit-Flashing-Heart.hex" on your h-drive, in a folder of your choice.

Use the USB cable to connect your micro:bit to your computer, minimise (or simply ignore) the browser and click on the **Files** icon located in the left-hand side menu of the desktop. A window will open and Micro:bit should show up as a USB drive – named MICROBIT – in the sidebar. To transfer your code, drag-and-drop (or copy-and-paste) the file "microbit-Flashing-Heart.hex" to the MICROBIT drive. If the file is transferred successfully, the micro:bit display should show a flashing heart.

If you want to disconnect micro:bit, find the MICROBIT drive in the sidebar of the **Files** window and notice the small eject/unmount icon next to its name. Click the eject icon to safely remove the device. However, you can leave the device connected until you complete all SCC.131 tasks (unless you find the flashing heart annoying).

> You have learnt how to develop a basic program using block coding in MakeCode, how to simulate its execution in MakeCode, how to upload and run your code on micro:bit, and how to disconnect micro:bit from your computer.

If time permits, you can continue to *Step 6* of the flashing-heart tutorial. Otherwise, click the **Home** icon in the MakeCode environment (located at the top-right corner) and proceed to Task 2. Outside this lab session, you could try other tutorials, including "Tutorials for the new micro:bit (V2)".

Task 2: Rock Paper Scissors

You have already developed code in C for the game "Rock, Paper, Scissors", as part of the SCC.111 lab exercises. This time, you will use *block coding* in MakeCode to develop the same game, and play against micro:bit.

In your browser, scroll down until you find the **Games** section and select **Rock Paper Scissors** (do NOT

select Rock Paper Scissors V2). Choose **Blocks** to start the tutorial. Development of the game requires 16 steps. Follow the on-screen guidance to generate a random number between 1 and 3 whenever you shake the image of micro:bit in the MakeCode editor, and display Rock (small square) for 1, Paper (large square) for 2, or Scissors for 3. When you reach *Step 15*, click the **Download** button to save the "microbit-Rock-Paper-Scissors.hex" file on your h-drive and upload it on your micro:bit, as described in Task 1.

> You have learnt how to build a program in MakeCode that detects when you shake micro:bit, i.e., relies on measurements from the motion sensor (accelerometer) of micro:bit.

If time permits, you can continue to *Step 16* and add a different sound for each outcome (note: you are using micro:bit V2, which can play all available music blocks in the **Music** toolbox). Otherwise, click the **Home** icon and proceed to Task 3.

Task 3: Rock Paper Scissors Teams

The Rock Paper Scissors code developed in Task 2 enables you to play against your own micro:bit, i.e., a human against the machine. How could the game (and the code) be modified when multiple players attempt to participate using their micro:bit devices? The rules of the team game are as follows:

- Participating micro:bit devices form a *group*.
- All players shake their micro:bits devices at the same time.
- Depending on the *tool* (i.e., Rock, Paper or Scissors) selected randomly by each device, the group is automatically split into *teams*. Devices in the *same team* have selected the *same tool*.
- Micro:bit devices in the same team display the *number of players* in their team.
- Teams visually compare numbers and *the team with the most players* wins the game.

Let us start (block)-coding!

Scroll down the homepage of Microsoft MakeCode until you find the **Radio Games** section and select **Rock Paper Scissors Teams**. Right-click on **Show Instructions** and open the link in a new tab of the browser. Stay in the tab that shows the homepage of MakeCode, scroll to the top of the page, go to the **My Projects** section and create a New Project. Call your project "Rock Paper Scissors Teams" and create it. Your

browser should now have two tabs: one tab for the MakeCode editor and one tab containing instructions on how to develop the Rock Paper Scissors Teams code.

Keep in mind that you will need to explore new toolboxes (e.g., **Arrays** in **Advanced**) and create new variables player, serialNumber, match,etc.) and then use them in your code. First read the tips given below and then follow the on-screen instructions:

- Try and re-build the Rock Paper Scissors code without sound (Task 2) from memory and compare it to the code given in the instructions. Note some unimportant differences between the code of Task 2 and the code in the on-screen instructions, e.g., a different name has been used for the variable, and numbers 0, 1 and 2 are randomly generated instead of 1, 2, and 3. Choose the variable name and the range of numbers that you prefer, and stick with your choices as you gradually build your code.
- The Rock Paper Scissors code generates a random number and checks its value whenever you shake the micro:bit. In *Step 1*, you are asked to reorganise the code into two parts: one part for creating a random number and storing its value in a variable whenever you shake the micro:bit, and a separate part for constantly checking the value of the variable.
- *Step 2* introduces three blocks that can be found in the **Radio** toolbox. Notice that:
  - Two radio blocks are included in the on start event. Micro:bit devices that participate in the game and run the same program should belong to the same group, represented by an ID number. When the program starts, the group ID number is set and the serial number of the micro:bit device is transmitted.
  - A radio block is included in the event to ensure that the randomly selected number (which represents Paper, Rock or Scissors) is being broadcast to neighbouring micro:bit devices.
- *Steps 3 to 6* describe the process of initialising an empty array called players, reading the serial number of another transmitting micro:bit device from a received packet and checking if the tool selected by the other micro:bit (stored in receivedNumber) matches the tool selected by your own micro:bit (stored in tool). If the two tools match and the serial number of the other micro:bit is not already in the players array, it is appended to the end of the array. If the two tools do NOT match but the serial number of the other micro:bit happens to be in the array, then it is removed from the array. For more details about the functionality of specific blocks, including blocks for array management, please check the online documentation for the blocks language.
  - "Help, I need to compare tool with receivedNumber but I cannot find receivedNumber in

the list of variables!" - You can drag receivedNumber from the on radio received block on your workspace and place it in other blocks.

- As explained in *Step 7*, the size of the players array represents the size of a team that has selected the same tool and is, therefore, the score of that team.

If you need help during this task, e.g., if the editor returns errors or the simulation does not work as expected, let us know.

The MakeCode editor should show images of two micro:bit devices. Hover the mouse cursor over each device to shake it (or click the Shake button). If the two devices select different tools, number 1 should be displayed by both of them. This is because two teams will be formed and each team will have just one player. Keep shaking one or both devices until they both select the same tool. When this happens, both devices will be members of the same team, therefore they will both display number 2.

> You are now familiar with methods that exploit the capability of micro:bit to communicate wirelessly with other micro:bit devices and exchange information.

If all blocks are in place and the simulation works as expected, download the `hex` file, transfer it to your micro:bit and start playing with others!

Task 4: Setting up the micro:bit runtime environment

The micro:bit runtime environment, also known as CODAL, is software that runs on a micro:bit to support the majority of the micro:bit programming languages. CODAL will help you understand how the micro:bit works. You will use it to write code that is tailored to the requirements of micro:bit. The SCC.131 tasks in the following weeks will be asking you to write and execute C coding exercises using the CODAL framework.

As the final task for this lab session, we want you to download the CODAL repository, compile the sample app and load it on your micro:bit device. In order to achieve this, you will need `git`, a version control software that will allow you to create a local copy of the code from the SCC gitlab service onto you machine. You will also need to install an ARM provided gcc toolchain in order to be able to compile C code into am ARM compatible binary. Here is a list of steps to compile your code.

Note: Any code you develop should be stored on the h-drive folder. Otherwise, your code may be wiped once you log out from your lab machine.

1. Download the software from the SCC gitlab repository.

```
cd ~/h-drive/
git clone http://scc-source.lancs.ac.uk/scc.Y1/scc.131/microbit-v2-samples.git
cd microbit-v2-samples/
```

2. Compile the code. This step will download all the required libraries, compile them, compile your example code, link them and generate a binary bitfile which should be flashed to the micro:bit device.

```
python3 ./build.py
```

3. Connect your micro:bit device to your lab machine using the provided USB cable. If the connection is successful, a new USB disk folder will appear on your screen with the name MICROBIT. If you copy a binary file (.hex) to this folder, then the micro:bit DAPLink mechanism on board the micro:bit will upload the binary file onto the memory of micro:bit.

4. Copy the file MICROBIT.hex from the microbit-v2-samples folder onto the MICROBIT device folder. This process is often called 'memory flash'.

5. Wait for the final flash to happen. Your micro:bit device should start generating random patterns on the LED display and play sounds. This is called the 'out-of-box experience'. Try pressing the left button of the micro:bit and see what happens. Then try the right button, both buttons simulatenously and the touch button above the LED display (this will allow you to record a short message and play it back). Do not forget to shake the micro:bit too!

Well done, you are now ready to start your micro:bit C coding exercises.

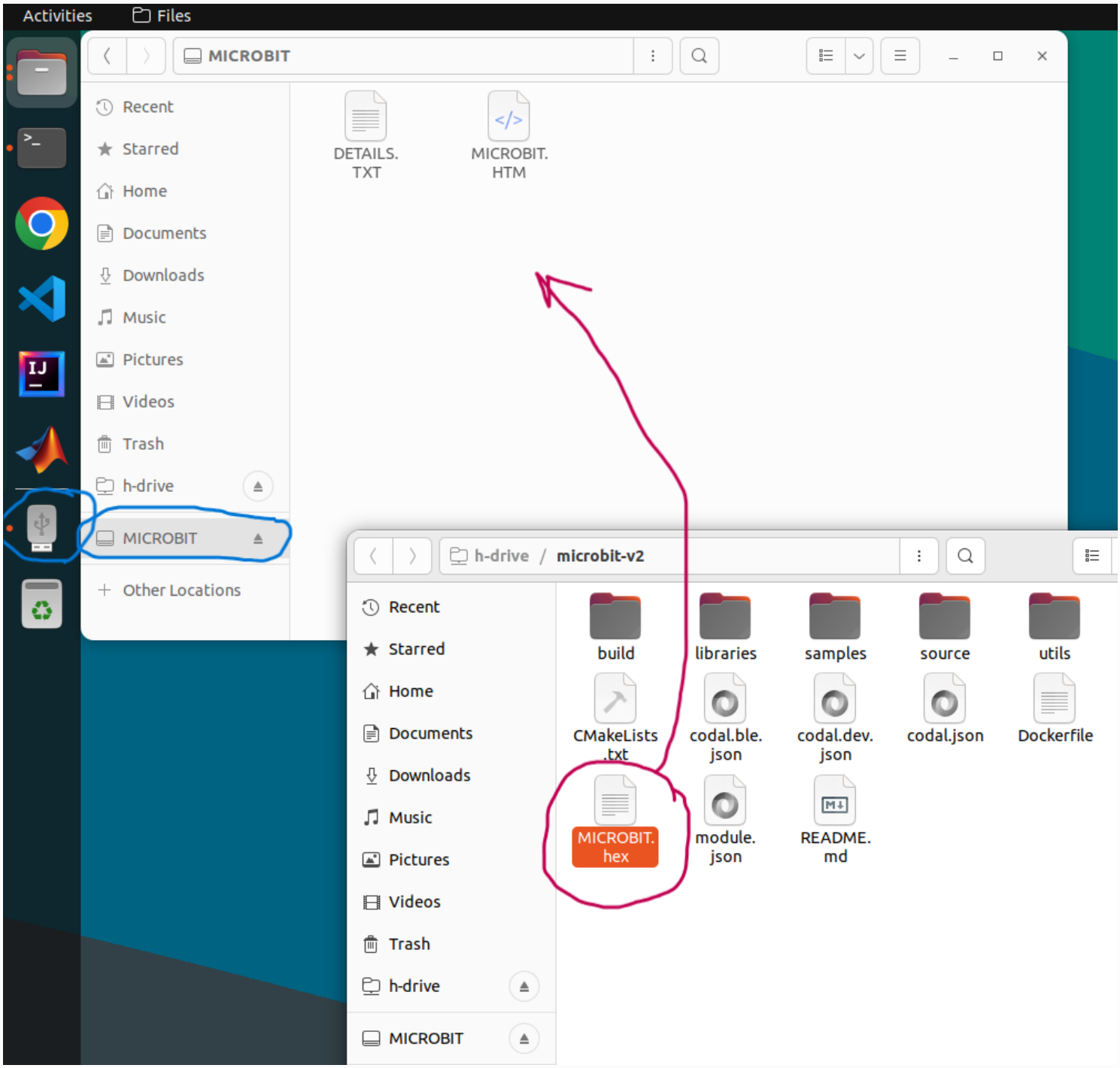

Figure 1: In order to reprogram your micro:bit device, you need to copy your compiled hex file MICROBIT.hex onto the MICROBIT device folder.

HACKER EDITION

SCC.111: IMAPCASTER

1. To be able to visualise your map, read about the KML output format. Write a new function ‘ `export_locations` ’:

```
void export_locations(struct location *list, int numLocations);
```

which produces (prints) KML format output, such that you can ‘redirect’ the output to a file and import it into google maps, google earth or some other mapping tool to visualise the locations.

A simple pair of locations in KML format looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Placemark>
  <name>John O'Gaunt</name>
  <Point>
    <coordinates>-2.801030,54.048649</coordinates>
  </Point>
</Placemark>
<Placemark>
  <name>3 Mariners</name>
  <Point>
    <coordinates>-2.802400,54.051201</coordinates>
  </Point>
</Placemark>
</kml>
```

2. If you want to go further, order the places by distance from the user’s location (closest to the furthest away). I added the function:

```
void sort_locations_by_dist(float lat, float lon, struct location *list, int
↪  numLocs);
```

Note, for this task you may want to use the 'sqrt' function - although you don't have to - in which case you'll want two things:

```
#include <math.h>
```

and, when you compile add the 'maths library':

```
gcc -o myprog myprog.c -lm
```

Don't forget to show us your solutions to get feedback on your code!

SCC.121: QUEUE

A&E (Accident and Emergency) in hospitals triage patients as they arrive to determine the order in which they should be served. Let's say they classify each patient as they come in as either **very serious (VS)**, **medium serious (MS)**, and **not serious (NS)**. Patients should be served in the order they came in except that an MS or NS patient cannot be served as long as there are VS patients waiting to be served and a NS patient cannot be served as long as there are MS patients waiting.

1. Implement this functionality in C based on the queue implementation you created for earlier. You may have to use multiple queues and generalize your queue implementation so that the same queue implementation can be used.
2. Is there any other way you could such a service?
3. Do you see any problem with serving patients like this? What would you propose as a remedy?

SCC.131: TWO-PLAYER DICE GAME

Use the knowledge that you gained from Tasks 1, 2 and 3 to build a dice game for two players. This means that the two plays should set the group ID number to the same value and not share it with others. The game specifications are as follows:

- When a micro:bit device is shaken, a random number between 1 and 6 should be chosen and the corresponding face of a dice should be displayed on the 5x5 LED screen of the micro:bit (e.g., if number 1 is chosen, a single LED should turn on in the centre of the 5x5 display).
- The player with the highest number wins the round and the score for that player increases by 1.
- The face of the dice should be replaced by the score of the player, which should be displayed for 1 second and then all LEDs should turn off.

For example, imagine that the player holding a red micro:bit has won 4 rounds and the player holding a yellow micro:bit has won 2 rounds. In the next round, the micro:bit devices return the following result:

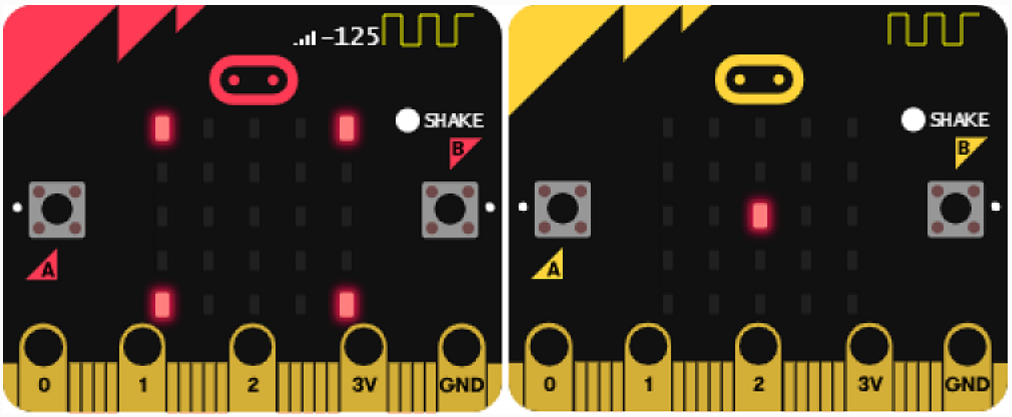

Figure 2: Picture 1 of dice game

Therefore, the player owning the red micro:bit wins the round and each micro:bit displays the updated scores:

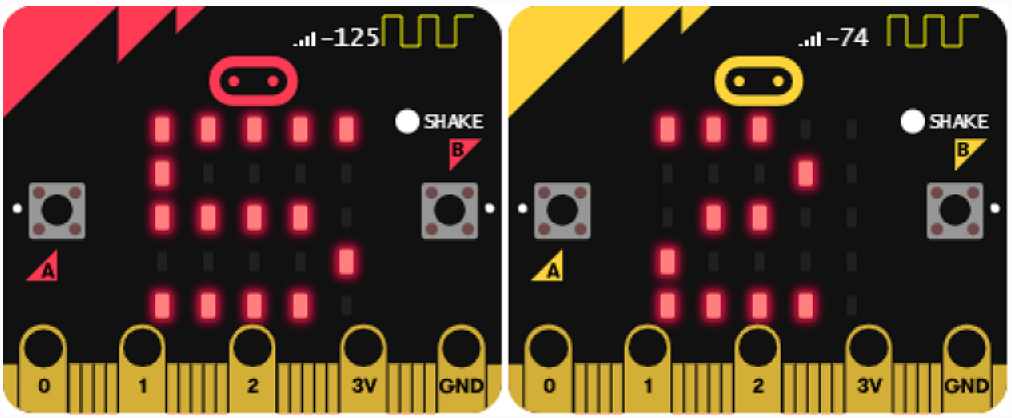

Figure 3: Picture 2 of dice game

The challenge of this task is to ensure that the two devices remain in sync. For instance, consider the first

pair of images above. Each micro:bit device attempts to send the randomly selected number to the other micro:bit device, so that the largest number can be determined and the winner for that round can be identified. In the example above, the yellow micro:bit successfully received the number from the red micro:bit, realised that it has lost the round and did not update its score. Imagine now that the red micro:bit failed to receive the randomly selected number from the yellow micro:bit (e.g., due to errors in reception). In this case, the red micro:bit will carry on displaying the same face of the dice and keep waiting for a retransmission by the yellow micro:bit. As a result, the red micro:bit will not realise that it has won the round and will not update its score. How could you ensure that both devices will move to the next round together, if each micro:bit device can receive only one value at a time (stored in `receivedNumber` , as seen in Task 3)?