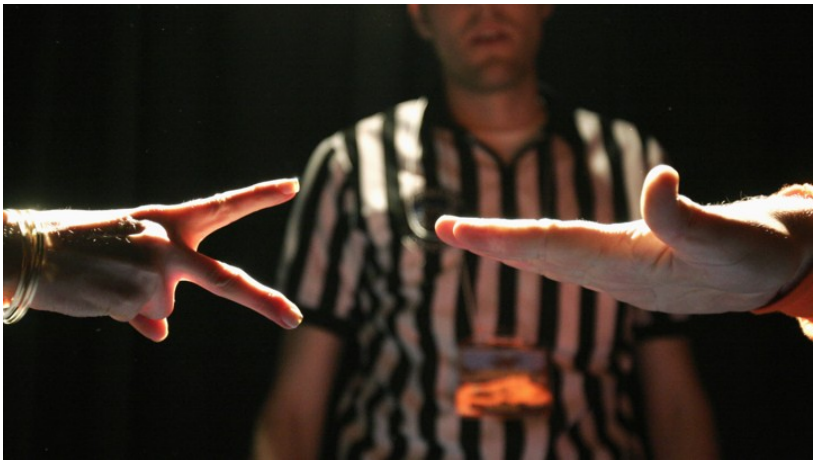
Laboratory 10: Communications

Bryan Ruddy

26 May 2022

THE MUSIC FADES. The lights flash. The crowds roar. It's the most anticipated game of the year. Rock Paper Scissors.



The ultimate battle. (Image from The Atlantic, `https://www.theatlantic.com/entertainment/archive/2015/12/how-rock-paper-scissors-went-viral/418455/`.)

In this lab you'll be wielding your computer systems skills to create a simple radio communication link. Specifically, you will be implementing the age-old classic ~~DOOM~~ Rock-Paper-Scissors on your micro:bit. You have been provided with:

- This document
- A testing framework
- A partially-implemented code template

You will be submitting your finished code, and answering a few questions along the way.

Remote students: Please note that the simulators we have provided for past labs cannot run radio code. You can use the provided test suite to check the operation of your code for tasks 0–4 and more information will be provided on testing task 5 soon.

Basic concepts

There are a few Python and micro:bit features we use here that you will not have seen before, and a few guidelines for how your micro:bits should communicate with each other that you should familiarise yourself with prior to writing the code.

Byte strings

As we saw in the first week, modern text data are usually encoded in a variable-length Unicode format, and Python uses this internally. However, when we are working with the low-level details of a

communication protocol, we can't afford to have a variable length; we have to know the exact length of our data. Python has a special data type, called `bytes`, that exactly represents a sequence of bytes rather than representing specifically text. The syntax is very simple—just put a `b` in front of the string literal to make it a byte string. We can write it in ASCII characters, and Python will treat these as numbers.

These variables do have a few quirks. The big one is that we have to watch out for how we index into bytes objects, as they behave subtly differently to string objects. For example:

```
normal_string = 'abcdefg'
bytes_string = b'abcdefg'

normal_string[0:4]    # returns 'abcd'
normal_string[3]      # returns 'd'
normal_string[3:4]    # returns 'd'

bytes_string[0:4]     # returns b'abcd'
bytes_string[3]       # returns 100
bytes_string[3:4]     # returns b'd'
```

Radio protocol

Your device will need to send messages via the radio with the following message structure:

```
RECIPIENT_ID|SENDER_ID|ACTION|ROUND_NUMBER
```

where `ACTION` is one of `R`, `P`, `S` or `X` (detailed below). `RECIPIENT_ID` and `SENDER_ID` are each exactly two characters long, and `ROUND_NUMBER` is at least one character long and at most four characters long. (The `|` character is used to separate fields in our description of the message—please don't include any in what you send or receive!) This message must be sent as a byte string. An example message would be:

```
b'0060R2'
```

which would denote a choice of `ROCK` by micro:bit `60` for round `2` sent to micro:bit `00`.

Messages that contain a play (`R`, `P`, or `S`) received by a device must be acknowledged with the `ACTION` value of `X`. For example, if the device `50` receives a play from device `49` for round `7`, the message sent to acknowledge this would be:

```
b'4950X7'
```

No other messages (such as acknowledgements or invalid messages) should be acknowledged.

Task 0: Identifying yourself and selecting an opponent

First, put your micro:bit number in the indicated global constant at the top of the code file. **IMPORTANT: If your micro:bit number uses a letter as a hex digit, make sure you use lowercase!**

For the rest of this task, we have implemented the code for you. Players are identified by a two-digit hexadecimal number from 0x00 to 0xff. IDs with value lower than 0x10 are padded with a leading zero. Have a look at the code structure we have used, and ask for advice if you're not sure how it works. This code is a simpler solution to the problem from the first micro:bit lab.

Testing our function

A testing framework is provided to you in `test_framework.py`. This depends on modules defined in `testing_modules`. We have implemented these modules as mocking modules - imitations of the behaviour on the micro:bit. Recall the use of `pytest` in previous labs.

We can test our `choose_opponent` function in the testing suite by defining a test for it. We have provided this test for you.

Things to note when testing with buttons

- The button mocking class needs to mock the button inputs - this is done by queueing button presses with the mocking function `press`.
- We set up our button presses before running our function to test since we cannot (in our implementation) inject button presses while the function is running.

Task 1: Selecting a play

Write code to select a play - one of Rock, Paper, or Scissors - to send to the opponent.

This should return a value from `b'R'`, `b'P'` or `b'S'`, so you can easily construct a bytes message to send. We have given you some helpful graphics for the different plays, which you are welcome to use (or not) as you wish.

Consider using a looping logic similar to the opponent selection above.

You may wish to write and run a test for your code. Your test will not be assessed directly, but we will use similar tests to mark the functionality of your code.

Task 2: Sending a message

Write code to send a message with a play and your ID via the radio.

Have a look at the documentation (`https://microbit-micropython.readthedocs.io/`) and locate an appropriate command. Remember that we are sending bytes objects, and that the function needs to return the time.

Ensure the message you are sending is compliant with the protocol described at the top of this document. You can test your code with the test provided in the test framework.

Notes on testing with the radio

Examine the test code to help you design additional tests. Notice that there are the following additional helper functions and variables:

- `state`: This is the queue of incoming messages. This corresponds to messages that would be passively received by the radio.
- `out_queue`: This the stack of outgoing messages. This corresponds to messages sent by the radio.
- `push_message()`: This puts messages into `state`, i.e. acting as "receiving" messages.
- `get_last_out()`: This pops the last message sent in the `out_queue`. We can use this to check messages that we send.

Task 3: Sending an acknowledgement

Write code to send an acknowledgement message.

This should be similar to the code you have written for sending an arbitrary play, but with a fixed contents value. You can test your code with the test provided in the test framework.

Task 4: Parsing a message

Write code to parse an incoming message according to the communication protocol.

It should:

- Validate that the message is of a valid length
- Validate the sender and receiver are correct
- Respond appropriately (acknowledgement messages)
- Return the correct play values for game resolution

We can use any of the receive functions - just note that we are expecting and parsing bytes objects. Once you have implemented this, you can test your code with the test provided in the test framework.

Task 5: Putting it together

Examine the main control loop, and complete the implementation. Pseudocode for it is given below.

```
def main():
  initialise <int> score to 0
  initialise <int> round_number to 0
  get <bytes> opponentID from buttons
  loop for ever:
    get <bytes> choice from buttons
    send choice via radio
    initialise <bool> acknowledged to False
    initialise <bool> resolved to False
    loop until acknowledged and resolved
      get <bytes> message from radio
      if message is play:
        set <int> result from resolve of choice and message
        update score from result
        set resolved to True
        display result and score
      if message is acknowledgement:
        set acknowledged to True
      if not acknowledged and time has passed:
        send choice via radio
    end loop
    update round_number
  end loop
```

The resolving and scoring functions are already provided to you. When re-sending your play after a lack of acknowledgement, make sure you wait a little while (at least 1 second, no more than 4 seconds) since you sent it before re-sending. Otherwise, you will build a radio jammer instead of a gaming device!

At this point, you should be able to **flash the code onto the micro:bit** for a test run. Find an opponent, and play a few rounds! You can use the battery box and batteries included with the micro:bit to use it without being plugged in to a computer.

We will provide a robotic test opponent for you in the level 4 laboratory at 70 Symonds St. during the laboratory sessions, which should be smiling at you from the lecturer PC. To use it, choose opponent `ff`. When you send it a play, you should see that play briefly flash on its display. You should also find it very easy to defeat!

We are working on a solution for testing the main loop if you don't have access to a micro:bit, and will send out an update with details when it is available.

Task 6: Questions

Please submit short (1-2 sentence) answers to the following questions:

1. What would be the benefit if the radio power were increased? Why might we not wish to do this?
2. Are the bytes in your message likely to be the only information transmitted over the radio? Explain.
3. Are there any parts of the message that you think aren't necessary? Would the protocol be easier to work with or more useful if you added more information to the message?
4. How difficult would it be to make an implementation of rock-paper-scissors that cheats, using this communication protocol? How could you change the protocol to prevent cheating?

Submission instructions

For this lab, please submit two files:

- `commlab.py`: The file you flash onto the microbit
- `questions.txt`: Answer the questions in the lab document

Note that you do not need to submit the test framework, we have only included this to help you debug your code. You do **not** need to submit any other files. Do **not** change the file names for your code from what was provided; however, don't worry if Canvas appends a -1 or -2 to the end.

Remember, all submissions are compared against each other and those from previous years. Copying someone else's code and changing the variable names constitutes academic misconduct by *both* parties. Because you are working in pairs, it is okay if your submission is very similar to your partner's.

Debugging tips

If your program seems to freeze when you play the game, but does work sometimes, it is likely you have not properly handled the situation where a message is sent but never received. There are two cases to think about:

- Your play is sent but not received. To deal with this, make sure you re-send your play periodically if you have not yet seen an acknowledgement. Don't re-send more than once per second, or you may cause radio interference.
- Your acknowledgement is sent but not received. If this happens, your opponent should occasionally re-send their previous play. To deal with this, note that your message parsing code needs to

be able to handle receiving plays from previous rounds, and re-sending acknowledgements if any old plays are received. Don't mistakenly interpret these old plays as the current one!

The other situation that can cause communications problems is sending inappropriate acknowledgements - **do not acknowledge acknowledgements.**

Of course, the communication also might not work if your opponent's implementation is incorrect. Try playing with someone else, like the robotic opponent, or checking against the unit tests in the test script.