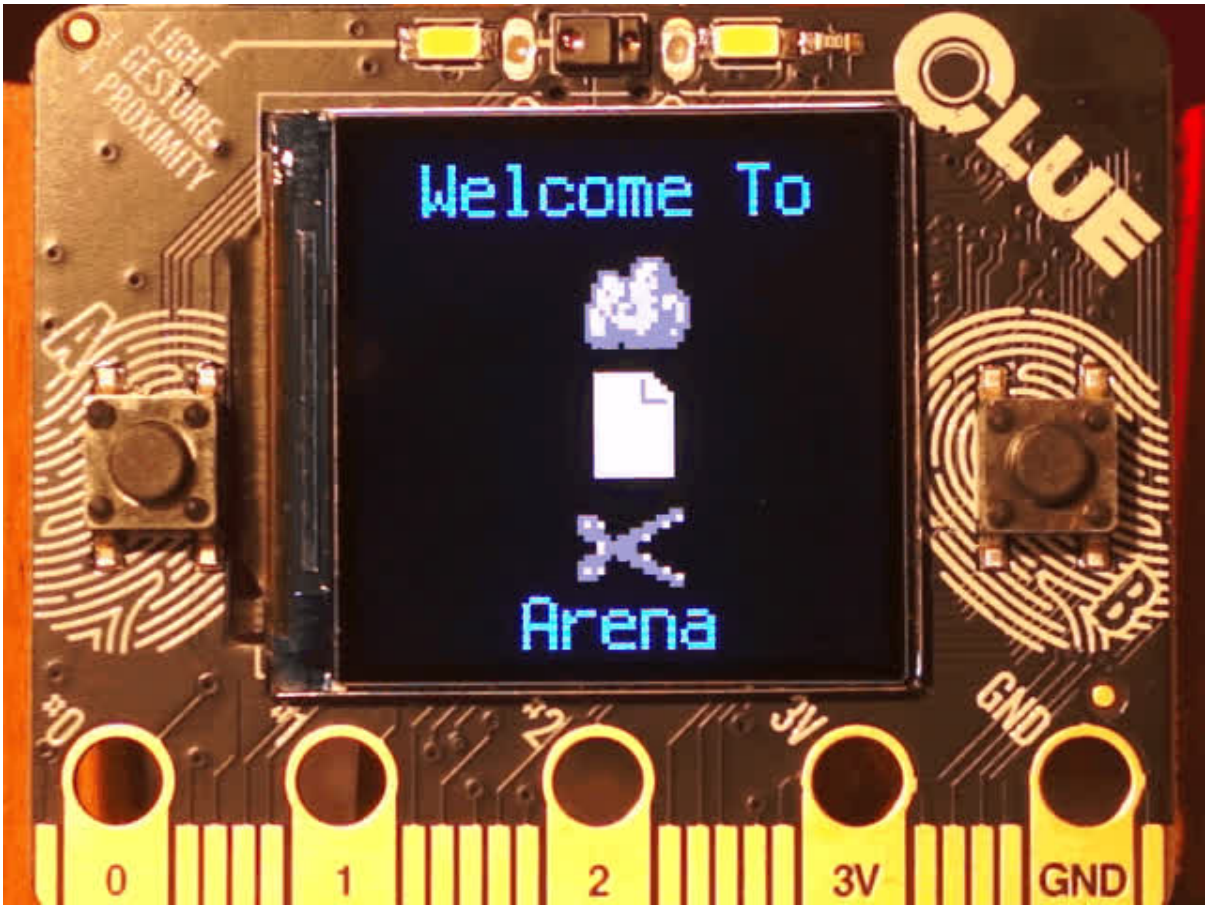
CLUE Rock, Paper, Scissors Game using Bluetooth

Created by Kevin Walters



https://learn.adafruit.com/rock-paper-scissors-circuitpython

Last updated on 2025-03-19 11:40:28 AM EDT

Table of Contents

Overview 5
- Parts

Design 7
- Rules
- Flow Diagram

User Interface 9
- User Interface

TFT Gizmo Detection 13
- The Challenge of Detecting the TFT Gizmo

Exchanging Choices 14
- Cheating
- Simultaneous Exchange of Player Choices
- Identity and Authentication

Networking 19
- Connection-oriented and Connection-less Communication
- Simple Game
- Advanced Game

Scores 25
- Multi-player Scoring
- Score Table Presentation

CircuitPython on CLUE 26
- Set up CircuitPython Quick Start!

CircuitPython on Circuit Playground Bluefruit 28
- Install or Update CircuitPython

CircuitPython 30
- Libraries

Very Simple Game 33
- Installing Project Code
- Code Discussion
- Evolving the Game and some History

Simple Game 37
- Example Video
- Installing Project Code
- Code Discussion
- Current Issues
- Evolving the Game

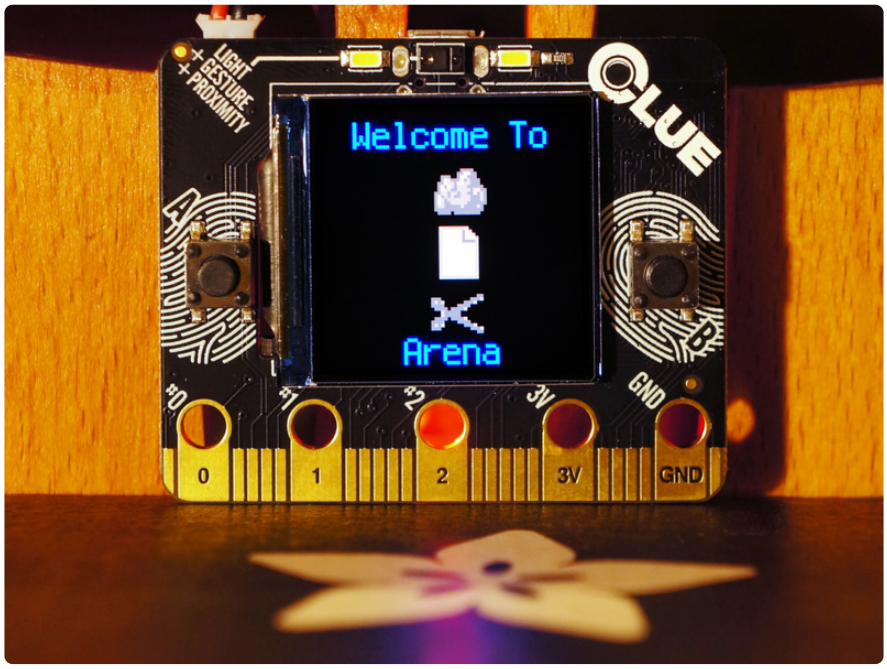
Advanced Game 49
- Example Video
- Installing Project Code
- Configuration

- Code Discussion
- SampleJukebox class
- RPSDisplay class
- Dynamic Advertising Interval
- Advertisement Matching
- Current Issues

Going Further 71

- Ideas for Areas to Explore
- Related Projects
- Further Reading

Overview



This project features three versions of a rock, paper, scissors game in CircuitPython. The first is a very simple, single-player text game showing the basic logic of the game. The second is a two-player game using Bluetooth Low Energy advertising to exchange the players' choices. The third is a multi-player game building on the foundation of the second version, adding displayio (https://adafru.it/EGh) graphics and sound. It has been tested with six players and potentially can work with far more.

The programs are written in CircuitPython for version 5.3.0 or later. The code runs on the CLUE and the Circuit Playground Bluefruit (CPB) with TFT (LCD) Gizmo screen. The third version of the game can also be used on just a CPB with the NeoPixels for output.

Thank-you to Matilda for recording the sound samples for the announcer in the third version of the game and the TMS5220 chip (https://adafru.it/Na0) for inspiring the post-processing on these.

Parts

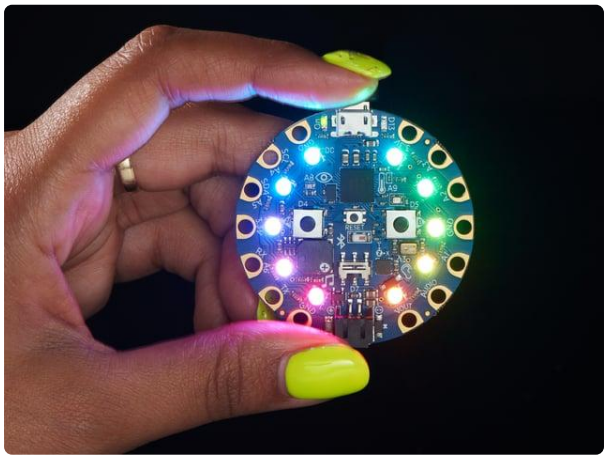
CLUE

[Adafruit CLUE - nRF52840 Express with Bluetooth LE]

Do you feel like you just don't have a CLUE? Well, we can help with that - get a CLUE here at Adafruit by picking up this sensor-packed development board. We wanted to build some...

https://www.adafruit.com/product/4500

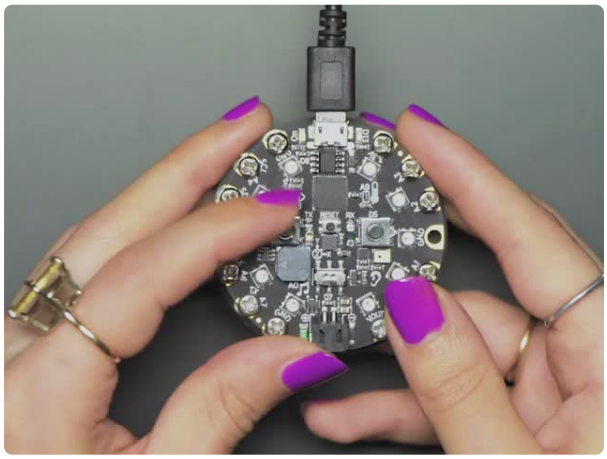
Circuit Playground Bluefruit with TFT Gizmo



[Circuit Playground Bluefruit - Bluetooth Low Energy]

Circuit Playground Bluefruit is our third board in the Circuit Playground series, another step towards a perfect introduction to electronics and programming. We've...

https://www.adafruit.com/product/4333



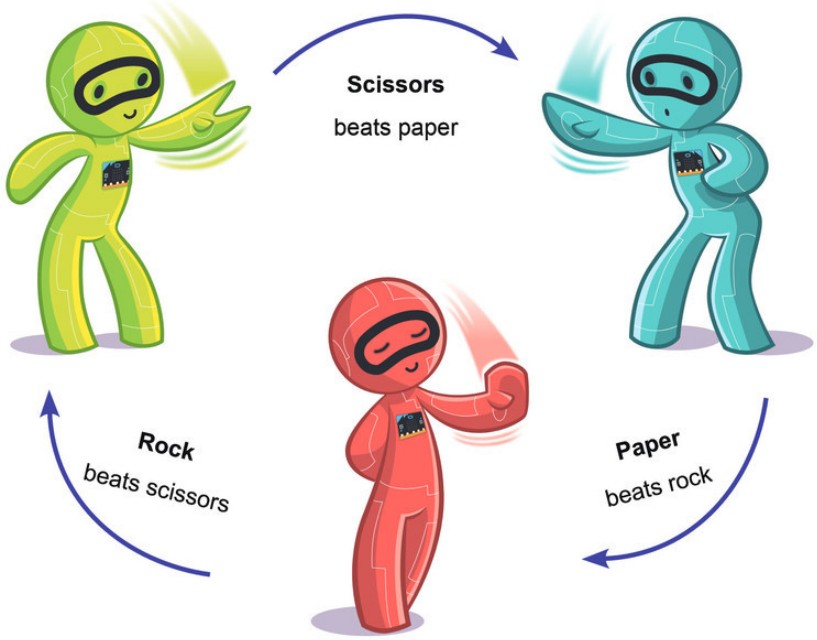
[Circuit Playground TFT Gizmo - Bolt-on Display + Audio Amplifier]

Extend and expand your Circuit Playground projects with a bolt on TFT Gizmo that lets you add a lovely color display in a sturdy and reliable fashion. This PCB looks just like a round...

https://www.adafruit.com/product/4367

USB cable - USB A to Micro-B
This here is your standard A to micro-B USB cable, for USB 1.1 or 2.0. Perfect for connecting a PC to your Metro, Feather, Raspberry Pi or other dev-board or...
https://www.adafruit.com/product/592

Design



Rules

To play the game:

- Each of the two players make an independent decision and reveal that to each other simultaneously.
- The rules to determine the winner are shown in the diagram above. If the two players make the same choice then the game is a draw (tie).

More than Two Players

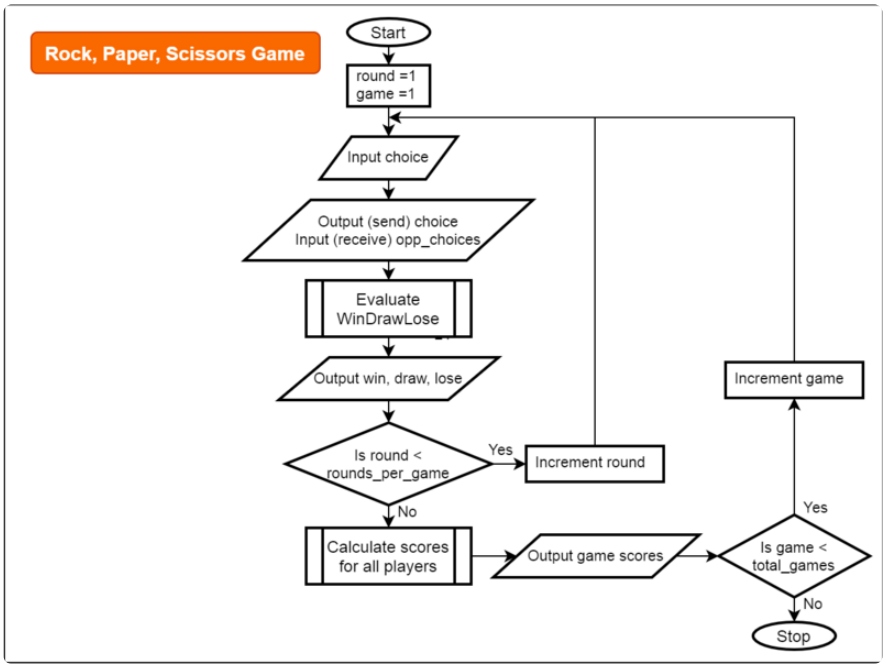
The game is normally a two-player game but can be extended to multiple players by simply using the player's choice against each opponent. The diagram above can also

serve to demonstrate how a three player game works, with each arrow representing a game:

- green player (scissors)
  - win vs blue,
  - lose vs red;
- blue player (paper),
  - win vs red,
  - lose vs green;
- red player (rock),
  - win vs green,
  - lose vs blue.

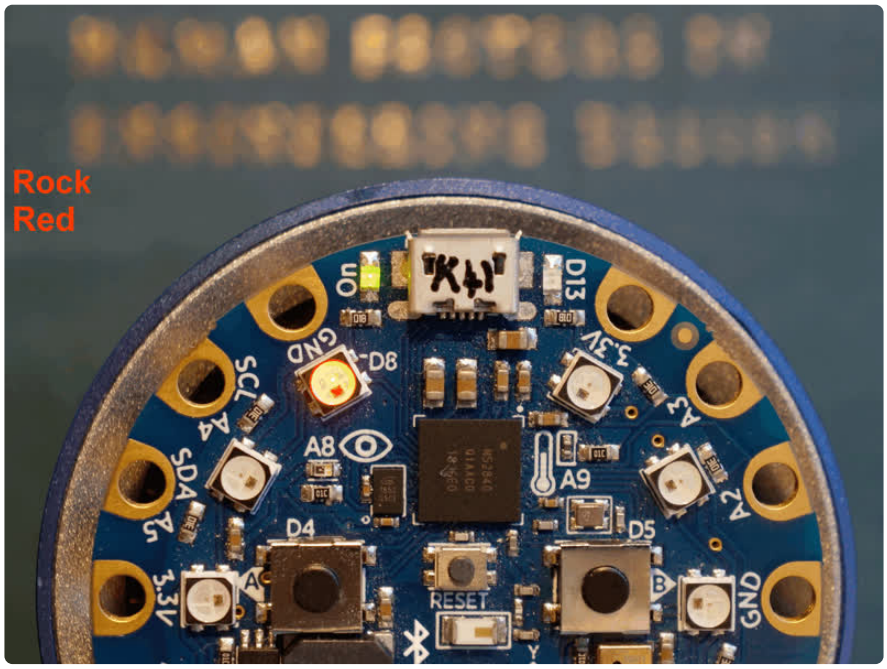
Flow Diagram

The flowchart below shows a high-level view of a multi-round game. This broadly matches the Advanced game, although the final implementation deviates from this playing games indefinitely.



The next pages discuss some aspects of the design of the different versions of the game. These include the user interface (UI), how the TFT Gizmo on a Circuit Playground Bluefruit can be detected, how the players' choices are exchanged between multiple devices and how the scores are presented at the end of each game.

User Interface



User Interface
Choice selection

The game needs:

- a method for the player to select a choice and
- a method to declare that choice is the final choice and exchange it with other player(s).

A three button interface could combine those two actions. The CLUE has two buttons and three touch-capable pads. The Circuit Playground Bluefruit (CPB) has two buttons, a switch and seven touch-capable pads.

A player needs to keep their choice secret from the other player(s) until they are exchanged. While it is easy to prevent others from spying on the choice directly there is the possibility of exposing this value if the opponent(s) can see an indirect indication of the choice. This could be from observing which touch pad a finger presses or by counting the number of presses on a button from finger movements or hearing button clicks. This is a form of side-channel attack. (https://adafru.it/Na1)

Avoiding the use of the touch pads for a one pad-per-choice selection helps to reduce the risk of overlooking. This also allows the boards to be used with a case that covers the touch pads, keeps the button actions identical between the CLUE and the CPB and maintains a consistent button layout.

Using a single button to cycle through the selections still needs some caution to avoid revealing the choice. It must

- not reset the player's choice to a fixed value per round
- and must not leave the player's choice on their last selected value.

The Simple and Advanced versions of the game both use the left button to select the choice and the right button to send it to the other players. The Advanced version deals with the issue of "button press counting" by randomising the player's choice each round.

Display on NeoPixels

The Advanced game can be played on a display-less Circuit Playground Bluefruit. The NeoPixels are used to show output to the player. These can be very bright and reflected light might be visible to others indicating the player's choice. The risk of this is reduced by using very low brightness values for the selected value indicated by the NeoPixels.

The NeoPixel position in addition to the colour is also used to represent the player's choice. It's tempting to use the default order of the NeoPixels but this would result in the third NeoPixel being underneath/near the player's left finger/thumb. The NeoPixels at the top of the Circuit Playground Bluefruit are used as they are clear of any fingers using buttons. The clockwise representation is:

- rock, red on NeoPixel 0;
- paper, purple on NeoPixel 9;
- scissors, sapphire blue on NeoPixel 8.

These can be seen in the animated image at the top of screen which cycles through the three.

The game number and round number are shown at the start of each round using the NeoPixels. They briefly light up in green with the number of NeoPixels illuminated indicating the game number. One NeoPixel then flashes white, the position indicates the round number.

The scores for the game are shown for each player in turn by illuminating the NeoPixels gradually in a circle. For 1-10 orange is used. For values larger than 10 a new colour is used after orange has been lit: yellow for 11-20, green for 21-30, blue for 31-40, indigo for 41-50 and violet for 51-60.

Screens

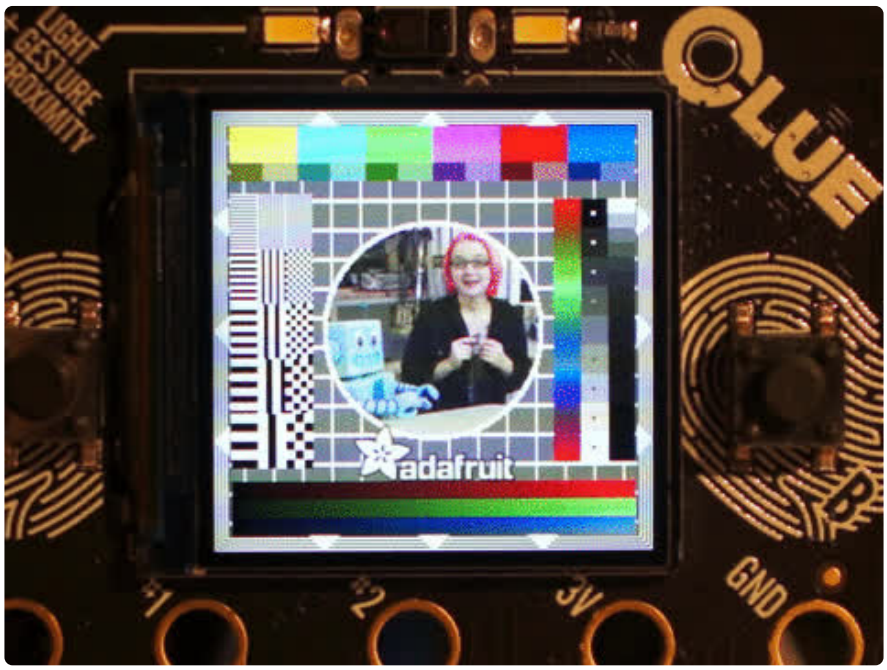
A game may have different phases. The Advanced game warrants:

1. A title screen.
2. A brief user guide.
3. A player list for the assembling players.
4. The local player's selection for the current round including information about the round number and a summary of the results so far.
5. The score at the end of each game for all players.

Screen Transitions

The transition between screens could just be implemented with an immediate change like a cut (https://adafru.it/Na2) in a film but there are other options to transition to a subsequent screen and provide a visual cue that something significant has changed or occurred.

The CLUE and TFT Gizmo's screens use liquid-crystal display (LCD) (https://adafru.it/Na3) technology. These displays use a backlight for illumination. Fortunately, this backlight has a brightness control and this provides a very efficient way to implement a smooth fade to black and back.



Sprites

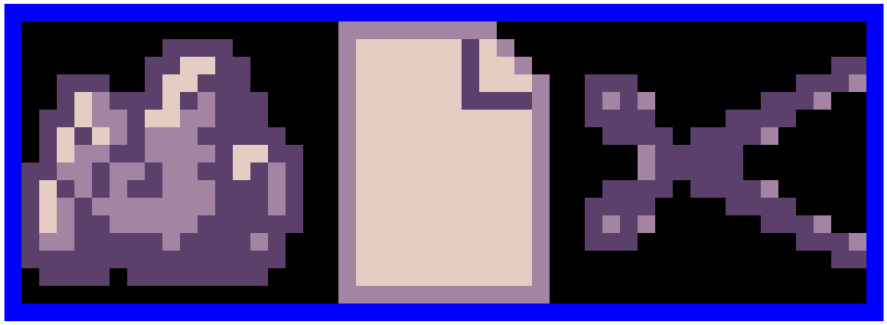
The displayio (https://adafru.it/Na4) library includes a `TileGrid` object which can be used together with the adafruit_imageload (https://adafru.it/Na5) library to load a sprite sheet (https://adafru.it/GC4) into memory. A sprite sheet is an ordered set of sprites (https://adafru.it/DJ7) contained in a single image file.

The Advanced game makes intentional use of low resolution 16x16 pixel images with a very limited palette for a retro feel. The three sprites representing the rock, paper and scissors yield a 48x16 sprite sheet. These were:

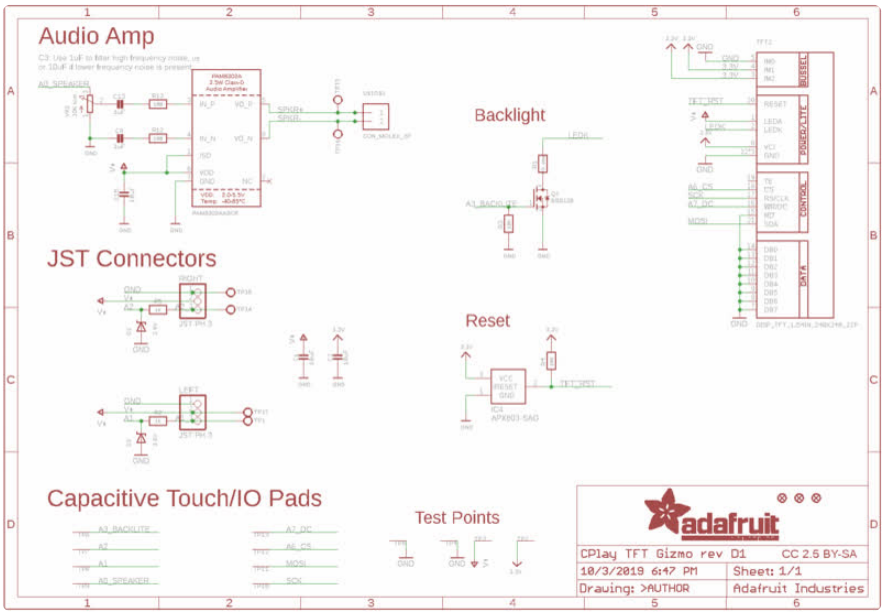
1. Drawn in the basic sprite editor in MakeCode Arcade. (https://adafru.it/DCY)
2. Displayed on screen (https://adafru.it/Na6) to allow them to be saved from the editor. MakeCode Arcade is unusual in embedding its source code inside a png (https://adafru.it/Na7) file which is its native file format.
3. Cropped and converted to an indexed bmp in GIMP (https://adafru.it/MbZ).

The rock is inspired by classic Arcade games and the paper by the Xerox Star (https://adafru.it/Na8) Document Icon (https://adafru.it/Na8). The pictorial form of an icon (https://adafru.it/Na9) is a conventional feature of a graphical user interface (GUI) (https://adafru.it/Naa). It also provides a language-neutral representation for the choice reducing the work for internationalization (i18n) (https://adafru.it/Nab) of a program or product.

The enlarged sprite sheet is shown below with a blue border.



TFT Gizmo Detection



There are two approaches to supporting the optional TFT Gizmo (https://adafru.it/Gfi) display on a Circuit Playground Bluefruit (CPB):

1. Provide configuration data to turn on (enable) the display. For a board with an attached display the enable would be manually set in the configuration for that board.
2. Detect the presence of the attached display.

The configuration approach could be implemented with a configuration file (https://adafru.it/Nac) to cleanly separate the per-instance data from the code. Detection of the screen can either be an alternative approach or the two can be combined.

The Challenge of Detecting the TFT Gizmo

The TFT Gizmo connects to a Circuit Playground Bluefruit (or Circuit Playground Express) using four of the pads for the display interface and one other pad for the backlight control. All of those pads are set as outputs making the interface unidirectional as a whole. This is known as simplex communication (https://adafru.it/Nad). Unfortunately, this means there is no method via standard signalling to detect whether the TFT Gizmo is present or not.

The 10k pull-down resistor (https://adafru.it/Nae) in the backlight circuitry shown in the schematic at the top of the page luckily provides a crude way to detect the presence of that resistor. The **A3** pad can momentarily be configured as an input with the processor's internal weaker (higher resistance) pull-up competing with the pull-down resistor. The external pull-down resistor is likely to win and this provides a method of

detecting the TFT Gizmo. It can be fooled by other things attached to **A3** but this is good enough to differentiate between nothing and the TFT Gizmo.

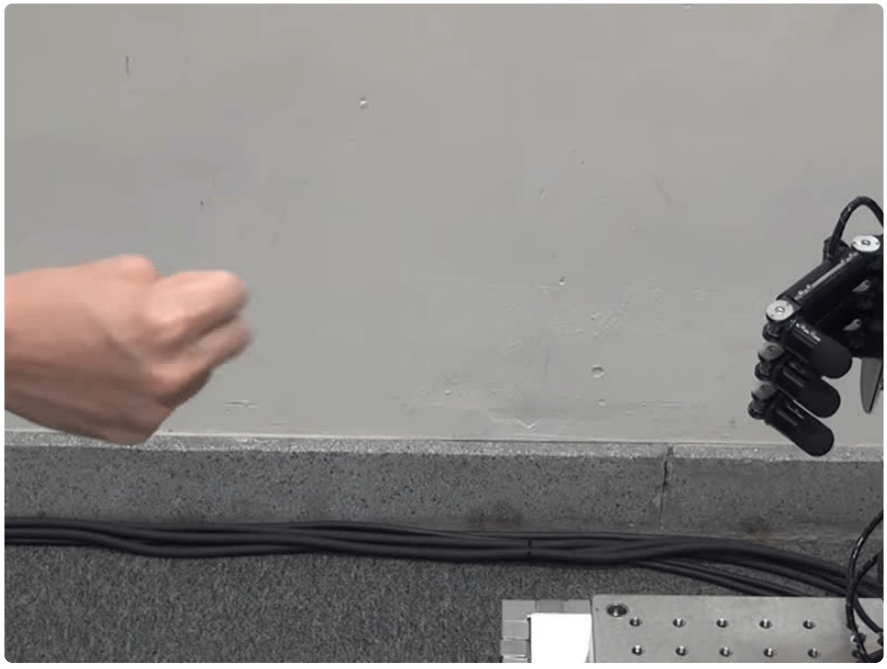
A sample CircuitPython function to detect the TFT Gizmo is shown below.

```
def tftGizmoPresent():
    """Determine if the TFT Gizmo is attached.
       The TFT's Gizmo circuitry for backlight
       features a 10k pull-down resistor.
       This attempts to verify the presence
       of the pull-down to determine
       if TFT Gizmo is present.
       This is likely to get confused if
       anything else is connected to pad A3.
       Only use this on Circuit Playground Express
       or Circuit Playground Bluefruit boards."""
    present = True
    try:
        with digitalio.DigitalInOut(board.A3) as backlight_pin:
            backlight_pin.pull = digitalio.Pull.UP
            present = not backlight_pin.value
    except ValueError:
        ### The Gizmo is already initialised, i.e. showing console output
        pass

    return present
```

This assumes that if **A3** is already in use then the display is already active. Once the TFT Gizmo is used on a CPB (or CPX) it remains active until the power is removed.

Exchanging Choices



Cheating

The classic game between human players uses a style of play where both players present their choice at the same time to reduce the possibility of cheating by reacting to the other player's choice. Human reaction times (https://adafru.it/Naf) make this a reasonable approach.

If there is not an agreed time to exchange choices then this allows a nefarious player to observe/receive an opponent's choice and then react to it. The video above shows a crafty, custom robot which carefully watches the opponent's hand and rapidly reacts with a winning choice within the human hand's movement time.

The game can be written to avoid immediately revealing the opponent's choice. This would prevent a player waiting and then reacting to that opponent. Unfortunately, a modified game could still reveal this and facilitate cheating. This could be solved with:

1. some method for preventing modification of the code or robustly verifying its integrity;
2. a truly simultaneous exchange;
3. a trusted third party, an arbitrator, collecting and then distributing the choices or wins together;
4. an approach where the choices are exchanged in some form but cannot be read until all are received.

The first option isn't feasible as CircuitPython and the associated hardware don't support this feature and code modification is intentionally trivial. The second option seems tempting but it is difficult to synchronise the devices to exchange data precisely enough. The exchange could only be near simultaneous with more than two devices and even two devices would struggle to achieve this with the busy radio spectrum used by Bluetooth.

The fourth option is desirable and practical as it removes the need in the third option for a referee or asymmetric roles for the players. It also doesn't place any constraints on the timing of communication between devices.

Simultaneous Exchange of Player Choices

The players could send an encrypted version of their choice and then send the key only when they have received the choices from all of the opponents. Since the key is revealed it cannot be reused making it a per-message key.

First Idea

An encryption algorithm E might produce the following with a certain key k:

- E_k(rock) = **chey**
- E_k(paper) = **ennem**
- E_k(scissors) = **roncttla**

It only takes a few seconds to see the flaw in this approach. The length of the output ciphertext (https://adafru.it/Nag) gives away the choice because the input plainttext (https://adafru.it/Nah) has a known format with a well-known, small set of possible values with unique lengths. This is a very simple form of traffic analysis (https://adafru.it/Nai).

Second Idea

Adding some characters to the messages to make them the same length removes the ability to deduce the choice in this case. This is referred to as padding (https://adafru.it/Naj).

The one-time pad (https://adafru.it/Nak) (also known as Vernam cipher) is a cipher which has the useful property of being unbreakable when used correctly. It should not be confused with padding mentioned in the previous paragraph. Given the short plaintext it is a very tempting choice here. An example of this is shown below.

```
&gt;&gt;&gt; random.seed(0)
&gt;&gt;&gt; plaintext = "rock....".encode("ascii")
&gt;&gt;&gt; key = bytes([random.randrange(256) for _ in range(len(plaintext))])
&gt;&gt;&gt; ciphertext = bytes([pt ^ k for pt, k in zip(plaintext, key)])
&gt;&gt;&gt; ciphertext
b'\x9e\x1c4\x8b\x93~w\xb0'
```

There are two issues here. The first is hinted at by the inclusion of a superfluous looking execution of `random.seed()`, the second is a bit more subtle and discussed later.

> Any one who considers arithmetical methods of producting random digits is, of course, in a state of sin.
>
> John Von Neumann (1951)

The seed sets where a random library starts its random number sequence and that sequence will be predictable from a seed value. CircuitPython running on nRF52840-based boards like the CLUE and Circuit Playground Bluefruit initialise the seed with a

number from the os.random() (https://adafru.it/Nal) function. `os.random()` provides true random numbers from a hardware generator.

Third Idea

The one-time pad seems suitable if the seed is initialised with a true random number. Unfortunately for cryptography, the pseudo-random number generator (PRNG) (https://adafru.it/lhE) commonly found in libraries produces a predictable stream of values. This is clearly noted in both the Python (https://adafru.it/Nam) and CircuitPython (https://adafru.it/Nan) documentation.

> Numbers from this module are not cryptographically strong! Use bytes from `os.urandom` (https://adafru.it/Nal) directly for true randomness.

The values from a standard PRNG must not be used for proper cryptography. In this case the key is intentionally revealed to everyone by the design of the protocol including any eavesdroppers. Over time, this provides an attacker with a sequence of consecutive numbers from the PRNG which is a very useful aid to determine the seed.

Fourth Idea

The solution looks robust in terms of the key if a true random number generator is used to generate that key.

```
&gt;&gt;&gt; plaintext = "rock....".encode("ascii")
&gt;&gt;&gt; key = os.urandom(len(plaintext))
&gt;&gt;&gt; ciphertext = bytes([pt ^ k for pt, k in zip(plaintext, key)])
&gt;&gt;&gt; ciphertext
b'\x82\x0e\xe7\xf4\xe3y]\x9c'
```

The CircuitPython `^` (xor) operator applies the key to the plaintext and is a classic final step for stream ciphers (https://adafru.it/Nao) to create the ciphertext. This creates the more subtle problem raised previously as it makes discovering a new alternative key for a different plaintext trivial and very fast. This alternative key can be given to an opponent who will decrypt the message producing a plaintext which is different to the original one.

An example is shown below where the player has chosen **rock** and sent the encrypted **rock** but the player can then send an alternative key to convince an opponent that **paper** was their choice.

```
&gt;&gt;&gt; desiredpt = "paper...".encode("ascii")
&gt;&gt;&gt; newkey = bytes([dpt ^ ct for dpt, ct in zip(desiredpt, ciphertext)])
&gt;&gt;&gt; newkey
b'\xf2o\x97\x91\x91Ws\xb2'
&gt;&gt;&gt; bytes([ct ^ k for ct, k in zip(ciphertext, key)])
b'rock....'
&gt;&gt;&gt; bytes([ct ^ nk for ct, nk in zip(ciphertext, newkey)])
b'paper...'
```

Fifth Idea

The one-time pad cannot be used with this simple protocol for the aforementioned reason. An essential property for all other encryption algorithms is the key cannot be selected or found easily to provide a chosen output for encryption or decryption. Hence, one of the common algorithms is likely to solve this issue.

AES (https://adafru.it/Nap) (Rijndael) is a modern, widely-used block cipher (https://adafru.it/Naq) but it is not currently offered by the standard CircuitPython libraries. Its block size is 128bit (16 bytes) which is a little coarse for direct use with short messages. The Chacha20 (https://adafru.it/Nar) stream cipher is easy to implement in Python and was selected for this program. Its key length (https://adafru.it/Nas) was shortened as this is just for fun. A short key makes a system vulnerable to brute-force attacks (https://adafru.it/Nat).

The number of flaws quickly found here shows how mistakes can easily be made with cryptography. A slightly more complex and more robust protocol for doing this can be found in the Simultaneous Exchange of Secrets section of Bruce Schneier's Applied Cryptography (https://adafru.it/Nau).

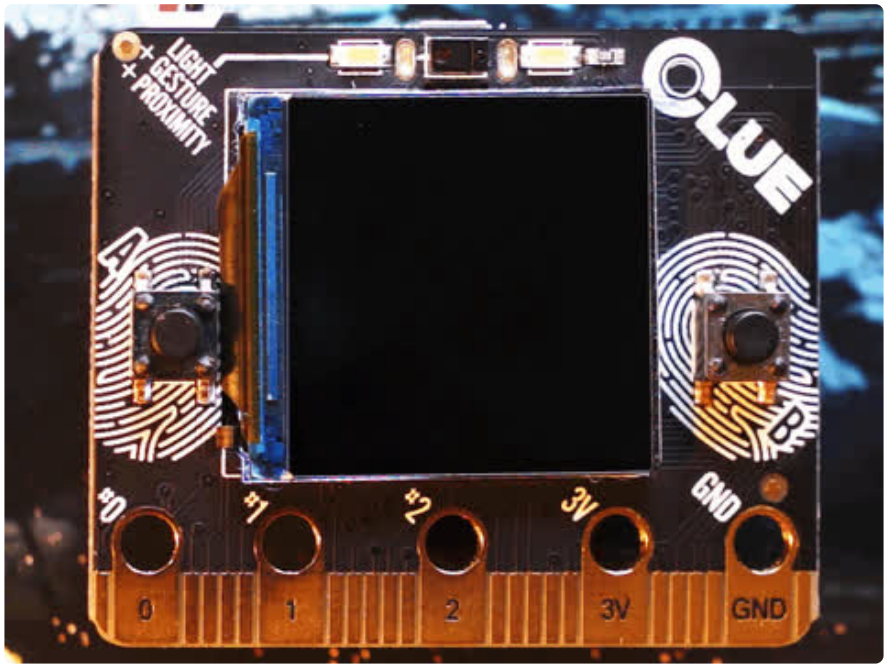
> Applications using encryption need to use well-known, mature protocols, high-quality libraries, have a thorough implementation review by experts and be operated as prescribed to achieve good security.

Identity and Authentication

For a game over Bluetooth, the players can be identified by the network (MAC) hardware address if the address privacy feature (https://adafru.it/Nav) is not in use or by a Bluetooth name. The issue of player and message authentication (https://adafru.it/Naw) has not been covered. For a rock, paper, scissors game outside of a formal competition this seems reasonable!

Bluetooth Low Energy (BLE) typically has limited range, often around 5-10m (16-33ft) which can be perceived as a privacy feature. The range can be far higher with sensitive receivers or powerful transmitters. Exploring Bluetooth 5 - Going the Distance (https://adafru.it/Nax) cites an example of 350m (1150ft) range for BLE!

Networking



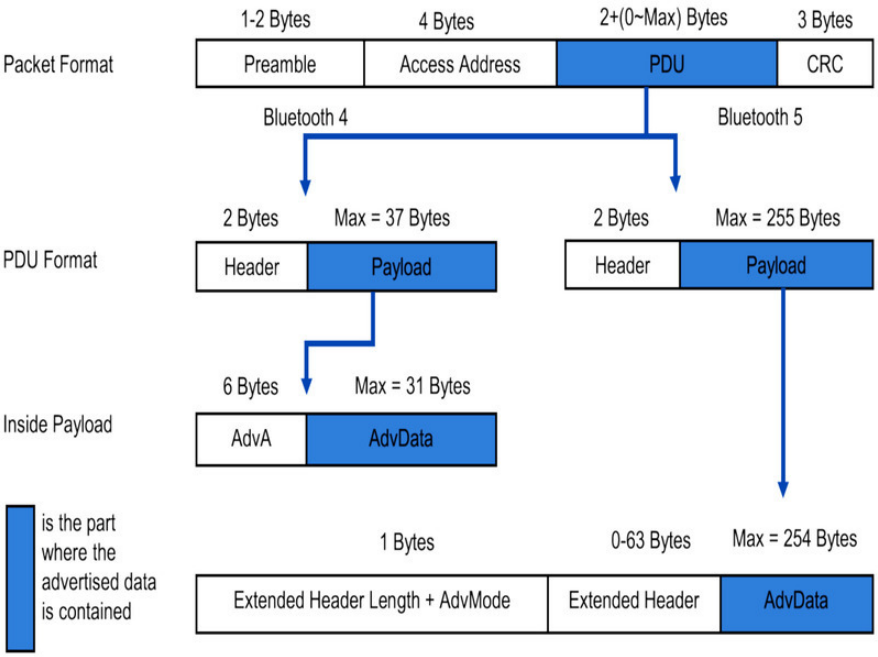
Connection-oriented and Connection-less Communication

Bluetooth Low Energy (BLE) allows a device to make a connection to another device. In BLE terminology, a device with the central role connects to one with a peripheral role. The connection provides reliable, bidirectional data exchange between two devices.

The BLE advertising mechanism (https://adafru.it/iCo) which as its name suggests is used by a device to advertise its presence can also be used to transmit a small amount of data. Bluetooth's use of (omnidirectional (https://adafru.it/Nay)) radio as a medium makes it inherently a broadcast based technology. BLE advertising uses the term broadcasting in the sense that all nearby devices receive and process the

packet if they are in scanning mode. Advertising uses three disparate channels (radio frequencies) to transmit data to increase the the reliability of transmission.

The original BLE advertising format was limited to 31 bytes of `AdvData` data. This is now sometimes referred to as the legacy format as Bluetooth 5 introduced an extended format for longer packets, shown below.



Broadcasting data is attractive as an efficient mechanism for sensors or BLE beacons (https://adafru.it/Naz) to send data to multiple nearby devices - the BLE roles here are broadcaster and observer. This can also be used for other forms of communication like simple multi-player games. However, if there is a requirement for reliable communication then the application or a library needs to add features to provide that.

Protocols like TCP (https://adafru.it/NaA) check for lost packets and re-transmit these even on LANs (https://adafru.it/NaB) which when lightly-loaded can be almost loss-free. The unlicensed Bluetooth spectrum (https://adafru.it/NaC) is shared with other systems like Wi-Fi and is often very busy in areas crowded with people and their devices. This creates a high degree of packet loss. For advertising, a lower layer in BLE (https://adafru.it/19Ak) discards corrupted packets detected by a 3 byte (24 bit) CRC (https://adafru.it/NaE) but it needs additional features for a sender to detect end-to-end loss and re-transmit.

Custom Advertisement Packets using ManufacturerData

AdafruitColor (https://adafru.it/NaF) is an example of an Advertisement class in CircuitPython. The source file mentions how classes can be created for other applications using the `ManufacturerDataField` field and Adafruit's Bluetooth company identifier (https://adafru.it/NaG) (`0x0822`):

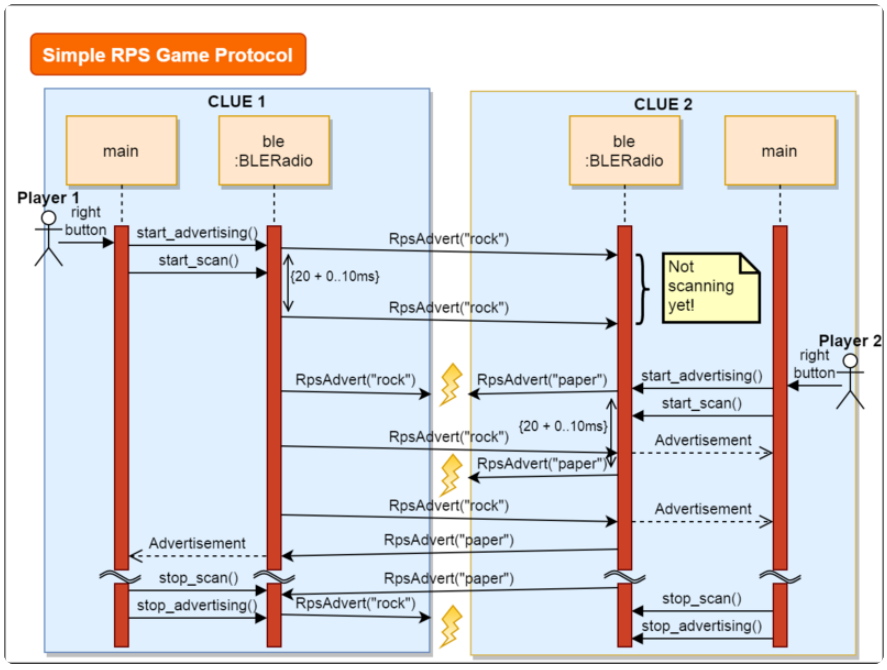
> Adafruit manufacturing data is key encoded like advertisement data and the Apple manufacturing data. However, the keys are 16-bits to enable many different uses. Keys above 0xf000 can be used by Adafruit customers for their own data.

This game uses values in that range for each field in the `Advertisement` sub-classes. There is currently no registry for allocation of these values so they were chosen at random for this game.

Simple Game

This version of the game is a two-player game which makes the exchange of players' choices relatively straightforward. A simple approach in the face of packet loss is to send the packets lots of times with the expectation that one will get through. This is a crude form of low-performance forward error correction (FEC) (https://adafru.it/NaH).

The diagram below shows how two players exchange data. Time runs downwards in the diagram. **Player 1** presses their button first to transmit their choice followed by **Player 2**.



This diagram is not a true sequence diagram as it does not show the complete algorithm with all the timeouts, rather it's showing a specific, abbreviated example of unsynchronised players and some packet loss.

The start_scan() (https://adafru.it/NaI) method returns an iterable object (https://adafru.it/NaJ) returning any received advertising packets in real-time until the (optional) specified `timeout` value. The returned packets which are parsed into an `Advertisement` class or sub-class are represented on the diagram with the dotted lines.

The advertising interval is set to the minimum value of 20 milliseconds with the hardware adding a random 0-10ms to the interval. This jitter (https://adafru.it/Lam) is added to reduce the chance of a prolonged clash with another device advertising at the same interval and starting at the same time.

The algorithm is transmitting packets using start_advertising() (https://adafru.it/NaI) for a specified maximum amount of time, this is shortened to a fixed duration when a packet is received from the other player. A single application-specific type of packet is used by both players which holds a text representation of the player's choice. These duration values need to be chosen to:

- allow for the time difference between players sending their choice by pressing their right button;
- provide enough time for the volume of packets sent to provide a very high probability of receiving at least one;
- keep the overall time low enough for players not to get bored or frustrated.

The final implementation uses 20 seconds for the maximum advertising duration reduced to 6 seconds when the other player's choice is received. The minimum permissible advertising interval of 20ms is used. Apple recommends this very short advertising interval in the context of a peripheral advertising for a connection in Accessory Design Guidelines for Apple Devices (https://adafru.it/NaK):

> The accessory should first use the recommended advertising interval of 20 ms for at least 30 seconds.
>
> If it is not discovered within the initial 30 seconds, Apple recommends using one of the following longer
> intervals to increase chances of discovery by the device: 152.5ms, 211.25ms, 318.75ms, 417.5ms, 546.25ms, 760ms, 852.5ms, 1022.5ms, 1285ms.

This short advertising interval should only be used for brief periods particularly when multiple devices are advertising simultaneously in a coordinated fashion.

Connection-based communication is likely to be a better choice for most two player games using Bluetooth.

Advanced Game

This version of the game allows multiple players to play and has a more sophisticated protocol featuring four types of message represented by different `Advertisement` sub-classes.

Making the List of Players

The Advanced game first has to determine who is playing. This could be part of the configuration data but that's not a very flexible approach.

The game advertises itself once at start-up using `JoinGameAdvertisement` packets. Up to eight players are shown on the display as they are discovered together with the first received signal strength indicator (RSSI) (https://adafru.it/NaL) value. The names and RSSI come from the default scan response (https://adafru.it/iCo).

The RSSI isn't strictly needed but it's interesting to see the value. It provides an approximate indication of proximity and reliability. Typical values will be between `-30` for very close and `-90` for distant (~6m/20ft) and unreliable.

For those interested in signal strength, there are some links for articles and research on attempts at using RSSI for range-finding (https://adafru.it/NaM) in Adafruit Forums: Proximity estimation using BLE (https://adafru.it/eca).

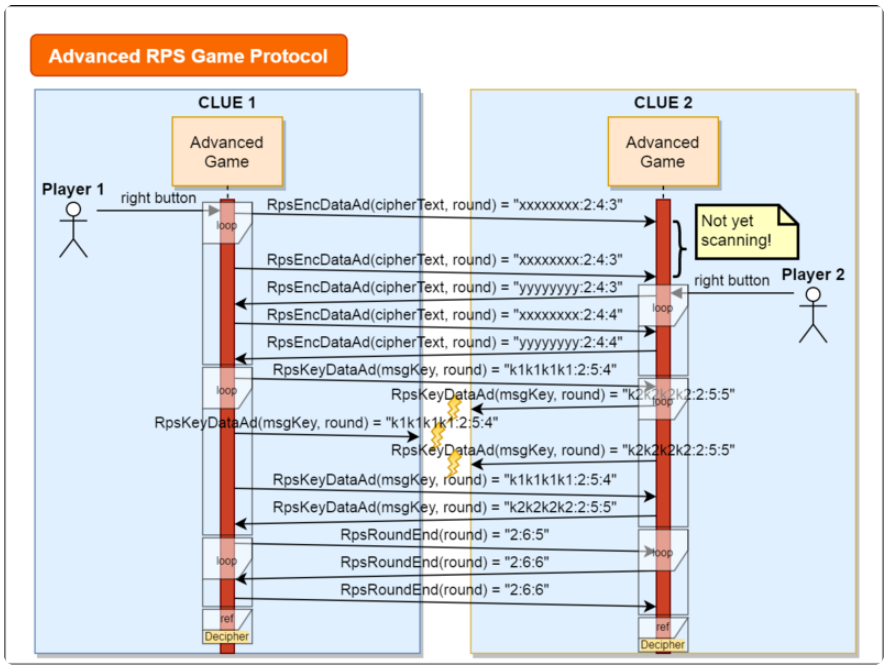
Exchanging Data between Players

The diagram below shows how two players exchange data. The sequence and algorithm is the same for more players. The essence of this per-round exchange is:

1. Each program encrypts the player's choice and sends it in an `RpsEncDataAdvertisement` message. When the same message is received from all of the opponents then an acknowledgement (ack) field is added or updated in the sent message.
2. Each program then sends their encryption key in an `RpsKeyDataAdvertisement` message when they have received all the encrypted values from the opponents.
3. The final `RpsRoundEndAdvertisement` message is sent. Its main purpose is to provide a reasonable assurance that the ack has been received by all for the previous message.

All of these messages also contain the round number of the game to ensure the players agree on the current round number.

The diagram below could show more players but it would get even more crowded! Time runs downward with **Player 1** pressing their button, then **Player 2**.



Again this is not a true sequence diagram as a lot of the logic in the algorithm is not shown and it shows a particular sequence of transmissions reacting to packet loss indicated by the lightning symbols.

The arrows between the **CLUE 1** and **CLUE 2** are showing the BLE advertising broadcasts with a representation of the data inside each packet. The first one, `RpsEncDataAd(cipherText, round)`, has a value of `"xxxxxxxx:2:4:3"` which is representing:

- `xxxxxxxx` - 8 bytes of cipherText,
- a round number of `2`,
- a message sequence number of `4` and
- an ack of message `3` from the previous round.

When **Player 1** first transmits, **Player 2** is not yet scanning for the messages. **Player 1** re-transmits the message at the advertising interval. A similar thing happens when there is packet loss. The diagram shows how the ack values are only incremented when the packet is actually received.

If there were more than two players this can take a lot longer as the message needs to be received by every opponent before the next message is sent in this simple protocol.

The advertising interval is increased when there are more than four players to reduce the use and congestion on the BLE advertising channels.

Practical and efficient reliable multicast (https://adafru.it/NaN) protocols are complex and difficult to design particularly in the face of high or uneven packet loss. There are

lots of improvements that could be made in this protocol but this is a functioning starting point and one that can be refined in further work. Bluetooth Mesh (https://adafru.it/NaO) offers some features for basic reliable multicast but this is not currently supported by CircuitPython.

Scores

Scores are fundamental to many competitive games. A two-player rock, paper, scissors game is normally scored with a simple tally of the wins. The scoring can be extended for a multi-player game to something more akin to a league.

Multi-player Scoring

The Advanced game allows more than two players and this means the draw can become more significant as a tie-breaker between players with the same number of wins. To recognise this, this version of the game uses a score of

- 2 for a win and
- 1 for a draw.

For comparison, some sports have moved to three points for a win (https://adafru.it/NaP) system.

Score Table Presentation

The final scores are initially presented with the local player (yellow) at the top followed by the remote players (cyan (https://adafru.it/NaQ)) in the order in which they joined the game. CircuitPython has the `sorted()` function and `sort()` method from Python but for novelty, the scores are visually sorted with a bubble sort (https://adafru.it/NaR). The swapping of the elements is intentionally slow enough to be able

to observe the algorithm at work. The video at the top of the page shows four players being sorted into descending score order.

The CircuitPython builtin sorts use the quicksort (https://adafru.it/NaS)algorithm. As an O (https://adafru.it/NaT)(n log n) algorithm, this is more efficient than the bubble sort's O (https://adafru.it/NaT)(n^2).

CircuitPython on CLUE

CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the **CIRCUITPY** flash drive to iterate.

The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

Set up CircuitPython Quick Start!

Follow this quick step-by-step for super-fast Python power :)

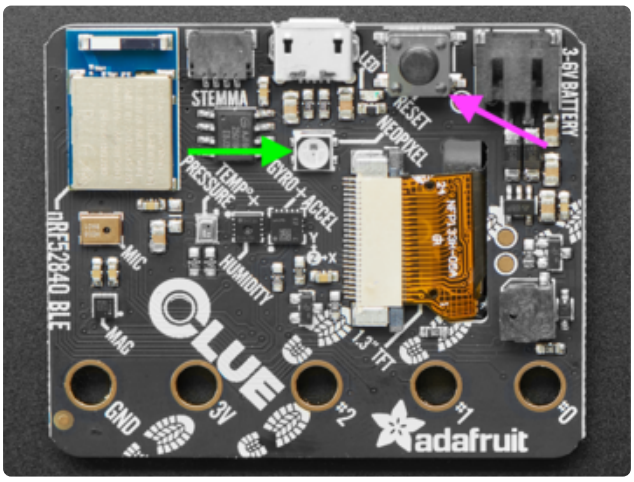
Download the latest version of CircuitPython for CLUE from circuitpython.org

https://adafru.it/IHF

Click the link above to download the latest version of CircuitPython for the CLUE.

Download and save it to your desktop (or wherever is handy).

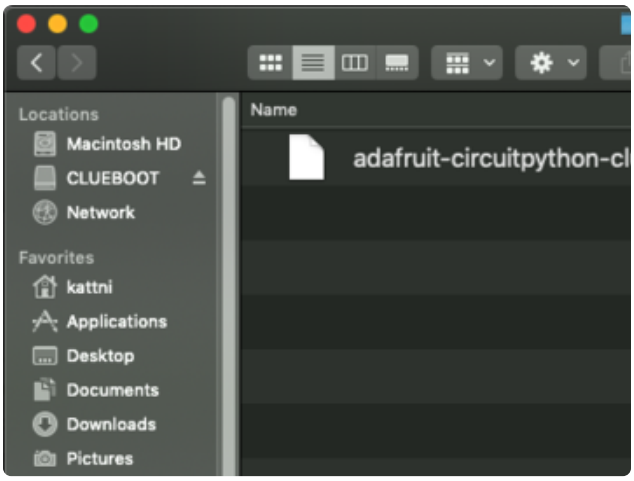
Plug your CLUE into your computer using a known-good USB cable.



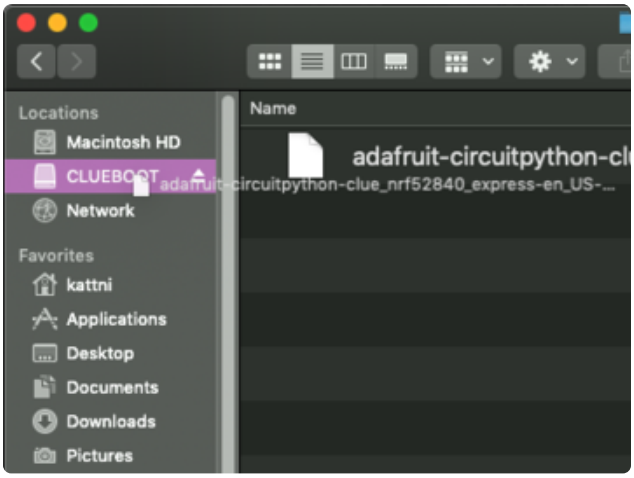
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the **Reset** button on the top (magenta arrow) on your board, and you will see the NeoPixel RGB LED (green arrow) turn green. If it turns red, check the USB cable, try another USB port, etc. **Note:** The little red LED next to the USB connector will pulse red. That's ok!

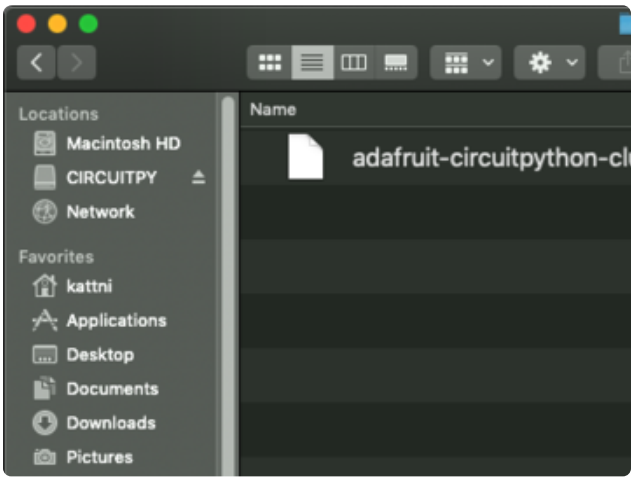
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!



You will see a new disk drive appear called **CLUEBOOT**.



Drag the **adafruit-circuitpython-clue-etc.uf2** file to **CLUEBOOT.**



The LED will flash. Then, the **CLUEBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

If this is the first time you're installing CircuitPython or you're doing a completely fresh install after erasing the filesystem, you will have two files - **boot_out.txt**, and **code.py**, and one folder - **lib** on your **CIRCUITPY** drive.

If CircuitPython was already installed, the files present before reloading CircuitPython should still be present on your **CIRCUITPY** drive. Loading CircuitPython will not create new files if there was already a CircuitPython filesystem present.

That's it, you're done! :)

CircuitPython on Circuit Playground Bluefruit

Install or Update CircuitPython

Follow this quick step-by-step to install or update CircuitPython on your Circuit Playground Bluefruit.

Download the latest version of CircuitPython for this board via circuitpython.org

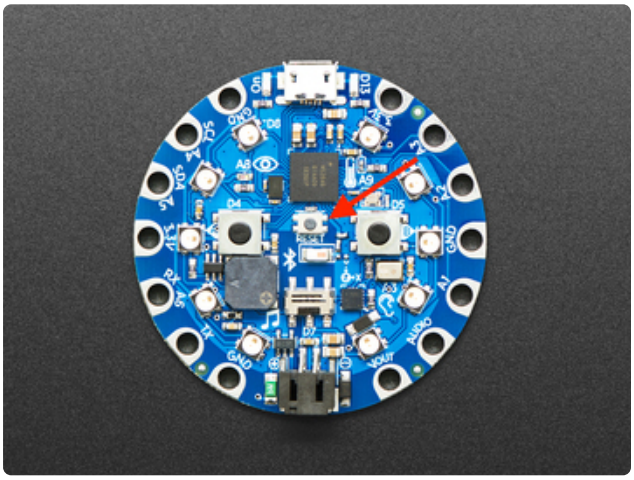
https://adafru.it/FNK

Click the link above and download the latest UF2 file

Download and save it to your Desktop (or wherever is handy)

Plug your Circuit Playground Bluefruit into your computer using a known-good data-capable USB cable.

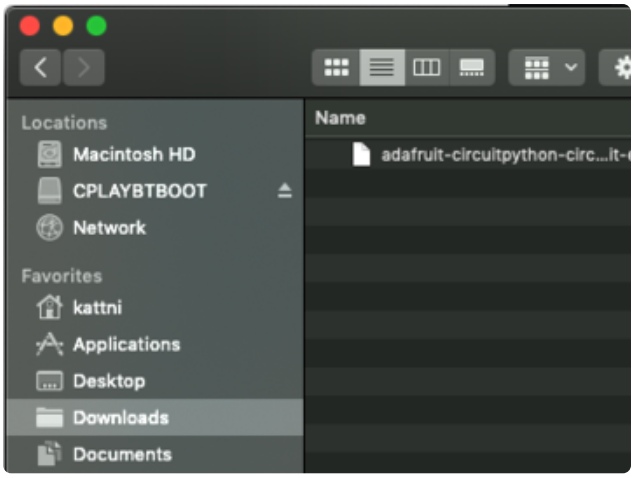
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.



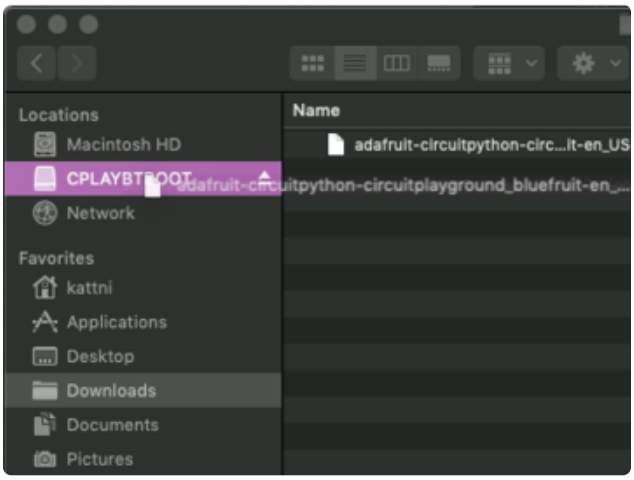
Double-click the small **Reset** button in the middle of the CPB (indicated by the red arrow in the image). The ten NeoPixel LEDs will all turn red, and then will all turn green. If they turn all red and stay red, check the USB cable, try another USB port, etc. The little red LED next to the USB connector will pulse red - this is ok!

If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!

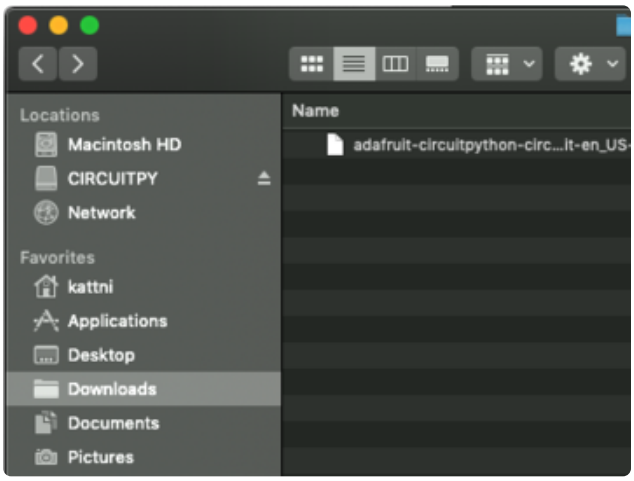
(If double-clicking doesn't do it, try a single-click!)



You will see a new disk drive appear called **CPLAYBTBOOT**.



Drag the **adafruit_circuitpython_etc.uf2** file to **CPLAYBTBOOT.**



The LEDs will turn red. Then, the **CPLAYBTBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

That's it, you're done! :)

CircuitPython

Libraries

Once you've gotten CircuitPython onto your CLUE or Circuit Playground Bluefruit (CPB) board, it's time to add some libraries. You can follow this guide page (https://adafru.it/GdM) for the basics of downloading and transferring libraries to the board.

Download the latest library bundle from circuitpython.org

https://adafru.it/ENC

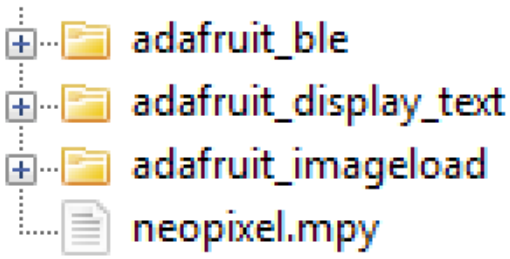
Libraries for Very Simple Game

No libraries are required from the bundle for this version.

The `random` library is used but this library is built into the CircuitPython interpreter.

Libraries for Simple Game

From the library bundle you downloaded in that guide page, transfer the following libraries onto the CLUE board's **/lib** directory:



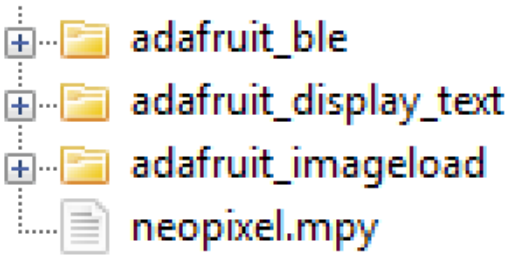
adafruit_ble
adafruit_display_text
adafruit_imageload
neopixel.mpy

The **adafruit_imageload** and **neopixel.mpy** aren't used by this version but are needed later for the Advanced game...

The adafruit_ble library must be from a bundle later or equal to 20200825.

Libraries for Advanced Game for CLUE

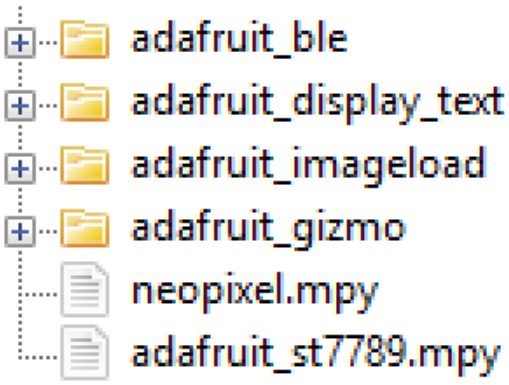
From the library bundle you downloaded in that guide page, transfer the following libraries onto the CLUE board's **/lib** directory:



adafruit_ble
adafruit_display_text
adafruit_imageload
neopixel.mpy

Libraries for Advanced Game for Circuit Playground Bluefruit with TFT Gizmo

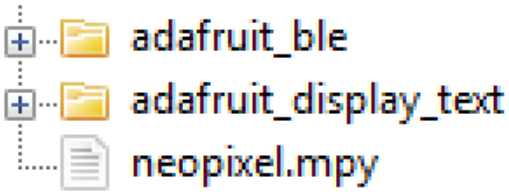
From the library bundle you downloaded in that guide page, transfer the following libraries onto the CPB board's **/lib** directory:



adafruit_ble
adafruit_display_text
adafruit_imageload
adafruit_gizmo
neopixel.mpy
adafruit_st7789.mpy

Libraries for Advanced Game for Circuit Playground Bluefruit only

From the library bundle you downloaded in that guide page, transfer the following libraries onto the CPB board's **/lib** directory:



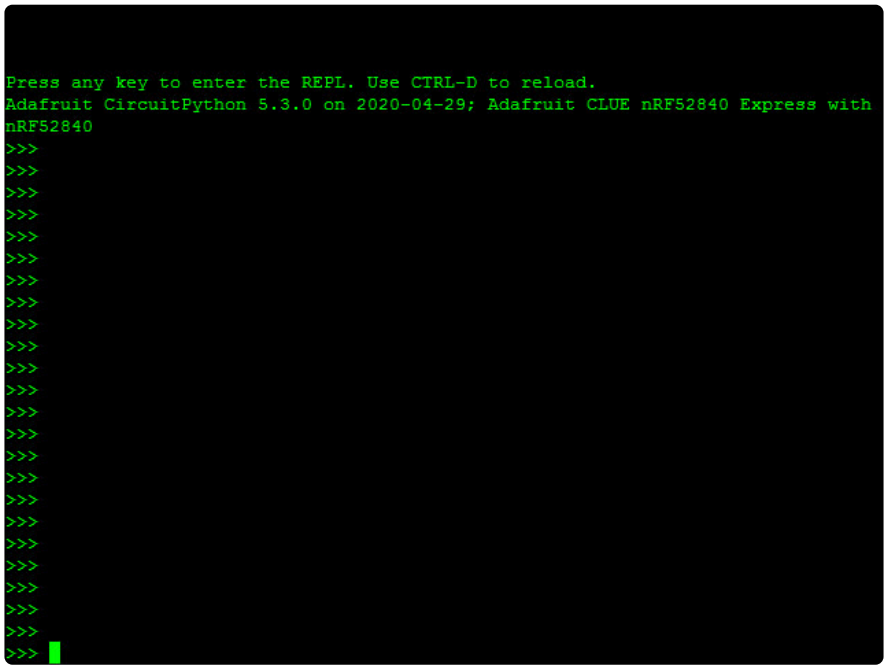
adafruit_ble
adafruit_display_text
neopixel.mpy

The **adafruit_display_text** is not actively used but is needed due to how the implementation uses and `import`s libraries.

Development Testing

During development, the application was tested on the CLUE and CPB boards using CircuitPython 5.3.1 with libraries from the **adafruit-circuitpython-bundle-5.x-mpy-20200825.zip** bundle. It should work on subsequent versions, the latest version is recommended for the CLUE (https://adafru.it/IHF) and CPB (https://adafru.it/FNK).

Very Simple Game



This is a very simple, single-player version of the rock, paper, scissor game written by Chris Bradfield (https://adafru.it/NaU) at KidsCanCode which demonstrates the essence of the game in a remarkably short program.

It is a text game which plays over the USB serial console, harking back to the teleprinters (https://adafru.it/eko) and terminals from the early days of computing. The video (animated gif) above shows the game being played.

This runs on any board that supports CircuitPython and can also run on CPython.

Installing Project Code

To use with CircuitPython, you need to first install a few libraries, into the lib folder on your **CIRCUITPY** drive. Then you need to update **code.py** with the example script.

Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CLUE_Rock_Paper_Scissors/very-simple/** and then click on the directory that

matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:

CIRCUITPY

```
# SPDX-FileCopyrightText: 2020 Kevin J Walters for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# clue-verysimple-rpsgame v1.0
# CircuitPython rock paper scissors game simple text game
# based on https://www.youtube.com/watch?v=dhaaZQyBP2g

# Tested with CLUE and Circuit Playground Bluefruit (Alpha)
# and CircuitPython and 5.3.0

# copy this file to CLUE/CPB board as code.py

# MIT License

# Copyright (c) 2015 Chris Bradfield, KidsCanCode LLC

# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:

# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.

import random

moves = ["r", "p", "s"]
player_wins = ["pr", "sp", "rs"]

print("Rock, paper scissors game: enter first letter for move or q for quit")
while True:
    player_move = input("Your move: ")
    if player_move == "q":
        break

    computer_move = random.choice(moves)
    print("You:", player_move)
    print("Me:", computer_move)
    if player_move == computer_move:
        print("Tie")
    elif player_move + computer_move in player_wins:
        print("You win!")
    else:
        print("You lose!")
```

Code Discussion

The code uses a `while` loop which runs forever repeating game after game. The player's text input is compared against the computer's random choice with the following code.

```
# Logic to determine winner
    if player_move == computer_move:
        print("Tie")
    elif player_move + computer_move in player_wins:
        print("You win!")
    else:
        print("You lose!")
```

The `if` is checking for a draw (tie) with a straightforward comparison. The `elif` is using a different technique, it concatenates (https://adafru.it/NaY) the two players' values with `+` and uses `in` to check if they are present in the `player_wins` `list`. Note: this is different to the `in` used with for statements (https://adafru.it/NaZ).

```
player_wins = ["pr", "sp", "rs"]
```

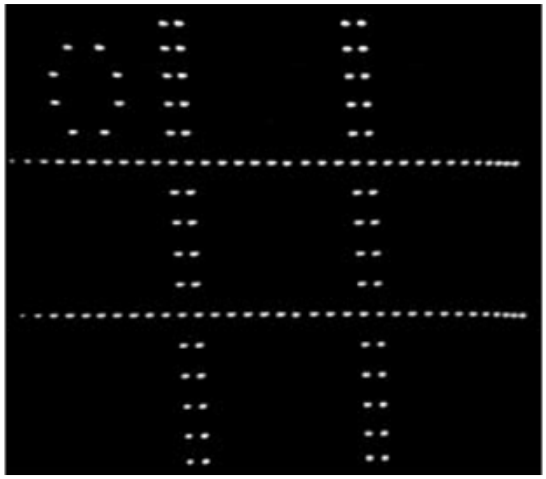
This is a form of lookup table (https://adafru.it/Na-) where the data is stored in a Python list. The winning combinations are stored without any accompanying additional data - their presence indicates a win. If draws have already been eliminated then the only remaining result if it is not a win is a loss. The `else` statement efficiently implements this.

In this case, the hybrid use of the equality test plus a lookup table keeps the code very compact. Lookup tables are often used to increase performance with the trade-off of some additional memory/storage use. They can be used in hardware too and are sometimes known by the acronym LUT.

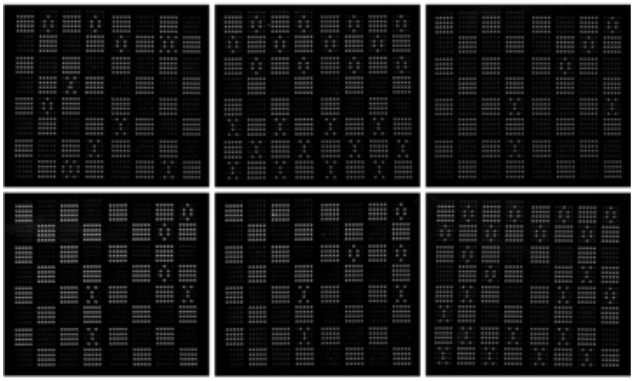
There are a few features missing from this game. One notable absence is the lack of data validation (https://adafru.it/Nb0) on the player's typed input. Invalid input is not reported and causes the player to lose.

Evolving the Game and some History

While very old multi-user computers are often associated with green screen terminals, many of the first electrical computers in the 1950s did have graphical screens and inevitably some games were written for these.

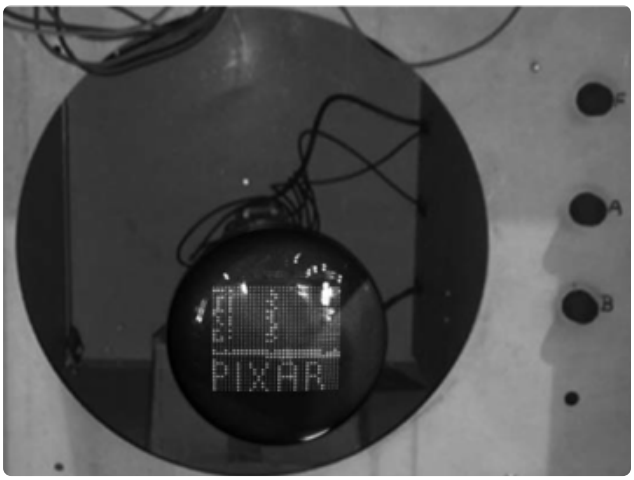


The console screens were small cathode-ray tubes. These were related to a type of dynamic memory technology from that era, the Williams-Kilburn (https://adafru.it/BJC) tube.



Christopher Strachey's photographs of his noughts and crosses (tic-tac-toe) game on the EDSAC (https://adafru.it/Nb1) and draughts (checkers) on the Ferranti Mark I (https://adafru.it/Nb2) from 1952 are shown here.

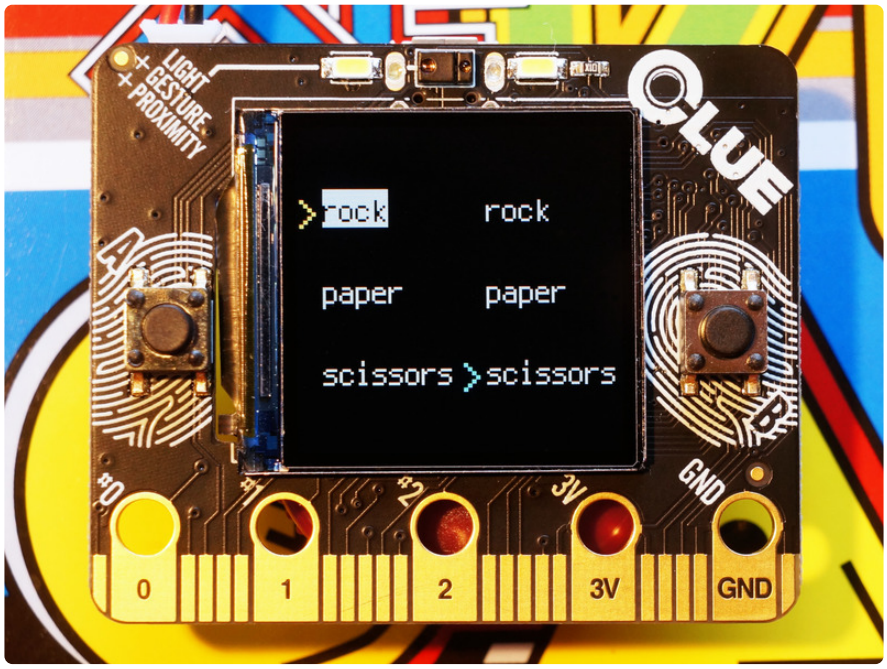
Alvy Ray Smith's photograph is from a replica of the Manchester Baby (SSEM) (https://adafru.it/Nb3) proving that simple animation of images was possible in 1948.



The images here are from The Dawn of Digital Light (https://adafru.it/Nb4).

The next version of the game uses the TFT LCD screen on the CLUE or TFT Gizmo attached to a Circuit Playground Bluefruit for a display. It is a two-player game using Bluetooth Low Energy for communication.

Simple Game



This is a two-player version of rock, paper, scissors game which uses Bluetooth Low Energy (BLE) advertising packets to exchange the players' choices and presents them on a graphical screen. It uses text to keep the program relatively short and simple.

Example Video

The video below shows the game being played on a pair of devices side-by-side. The two players would normally hide their choices. Bluetooth facilitates this as the game works well up to 4m (13ft) apart. The boards have been placed next to each other in this demonstration to ease the video recording.

The sequence of events in the video:

- 00:03 The program has already started before the video. The choices are selected on the two boards with the left button.
- 00:06 The right buttons are pressed starting the exchange of choices over BLE.
- 00:12 The appearance of the cyan cursor on the right indicates the opponent's choice has been received. The winning choice is evaluated and flashes (inverts foreground/background (https://adafru.it/Nb5)) on screen. In this case, rock blunts scissors.
- 00:23 Start of second round.
- 00:29 Paper wraps rock, paper flashes.
- 00:39 Start of third round.
- 00:46 Cyan cursor without any flashing indicates both players have chosen paper - a draw (tie).

Installing Project Code

To use with CircuitPython, you need to first install a few libraries, into the lib folder on your **CIRCUITPY** drive. Then you need to update **code.py** with the example script.

Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CLUE_Rock_Paper_Scissors/simple/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:

CIRCUITPY

```
# SPDX-FileCopyrightText: 2020 Kevin J Walters for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# clue-simple-rpsgame v1.3
# CircuitPython rock paper scissors game over Bluetooth LE

# Tested with CLUE and Circuit Playground Bluefruit Alpha with TFT Gizmo
# and CircuitPython and 5.3.0

# copy this file to CLUE/CPB board as code.py

# MIT License

# Copyright (c) 2020 Kevin J. Walters

# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:

# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.

import time
import os
import struct
import sys

import board
from displayio import Group
import terminalio
import digitalio

from adafruit_ble import BLERadio
```

```
from adafruit_ble.advertising import Advertisement, LazyObjectField
from adafruit_ble.advertising.standard import ManufacturerData,
ManufacturerDataField

from adafruit_display_text.label import Label


debug = 3

def d_print(level, *args, **kwargs):
    """A simple conditional print for debugging based on global debug level."""
    if not isinstance(level, int):
        print(level, *args, **kwargs)
    elif debug >= level:
        print(*args, **kwargs)


def tftGizmoPresent():
    """Determine if the TFT Gizmo is attached.
       The TFT's Gizmo circuitry for backlight features a 10k pull-down resistor.
       This attempts to verify the presence of the pull-down to determine
       if TFT Gizmo is present.
       Only use this on Circuit Playground Express (CPX)
       or Circuit Playground Bluefruit (CPB) boards."""
    present = True
    try:
        with digitalio.DigitalInOut(board.A3) as backlight_pin:
            backlight_pin.pull = digitalio.Pull.UP
            present = not backlight_pin.value
    except ValueError:
        # The Gizmo is already initialised, i.e. showing console output
        pass

    return present


# Assuming CLUE if it's not a Circuit Playround (Bluefruit)
clue_less = "Circuit Playground" in os.uname().machine

# Note: difference in pull-up and pull-down
#       and not use for buttons
if clue_less:
    # CPB with TFT Gizmo (240x240)

    # Outputs
    if tftGizmoPresent():
        from adafruit_gizmo import tft_gizmo
        display = tft_gizmo.TFT_Gizmo()
    else:
        display = None

    # Inputs
    # buttons reversed if it is used upside-down with Gizmo
    _button_a = digitalio.DigitalInOut(board.BUTTON_A)
    _button_a.switch_to_input(pull=digitalio.Pull.DOWN)
    _button_b = digitalio.DigitalInOut(board.BUTTON_B)
    _button_b.switch_to_input(pull=digitalio.Pull.DOWN)
    if display is None:
        def button_left():
            return _button_a.value
        def button_right():
            return _button_b.value
    else:
        def button_left():
            return _button_b.value
        def button_right():
            return _button_a.value

else:
```

```
    # CLUE with builtin screen (240x240)

    # Outputs
    display = board.DISPLAY

    # Inputs
    _button_a = digitalio.DigitalInOut(board.BUTTON_A)
    _button_a.switch_to_input(pull=digitalio.Pull.UP)
    _button_b = digitalio.DigitalInOut(board.BUTTON_B)
    _button_b.switch_to_input(pull=digitalio.Pull.UP)
    def button_left():
        return not _button_a.value
    def button_right():
        return not _button_b.value

if display is None:
    print("FATAL:", "This version of program only works with a display")
    sys.exit(1)

choices = ("rock", "paper", "scissors")
my_choice_idx = 0

# Top y position of first choice and pixel separate between choices
top_y_pos = 60
choice_sep = 60

DIM_TXT_COL_FG = 0x505050
DEFAULT_TXT_COL_FG = 0xa0a0a0
DEFAULT_TXT_COL_BG = 0x000000
CURSOR_COL_FG = 0xc0c000
OPP_CURSOR_COL_FG = 0x00c0c0


def setCursor(c_idx, who, visibility=None):
    """Set the position of the cursor on-screen to indicate the player's
selection."""
    char = None

    if visibility == "show":
        char = ">"
    elif visibility == "hide":
        char = " "

    if 0 <= c_idx < len(choices):
        dob = cursor_dob if who == "mine" else opp_cursor_dob
        dob.y = top_y_pos + choice_sep * c_idx
        if char is not None:
            dob.text = char


def flashWinner(c_idx, who):
    """Invert foreground/background colour a few times
       to indicate the winning choice."""

    if who == "mine":
        sg_idx = rps_dob_idx[0] + c_idx
    elif who == "opp":
        sg_idx = rps_dob_idx[1] + c_idx
    else:
        raise ValueError("who is mine or opp")

    # An even number will leave colours on original values
    for _ in range(5 * 2):
        tmp_col = screen_group[sg_idx].color
        screen_group[sg_idx].color = screen_group[sg_idx].background_color
        screen_group[sg_idx].background_color = tmp_col
        time.sleep(0.5)
```

```
# The 6x14 terminalio classic font
FONT_WIDTH, FONT_HEIGHT = terminalio.FONT.get_bounding_box()
screen_group = Group()

# The position of the two players RPS Label objects inside screen_group
rps_dob_idx = []

# Create the simple arrow cursors
left_col = 20
right_col = display.width // 2 + left_col
for x_pos in (left_col, right_col):
    y_pos = top_y_pos
    rps_dob_idx.append(len(screen_group))
    for label_text in choices:
        rps_dob = Label(terminalio.FONT,
                        text=label_text,
                        scale=2,
                        color=DEFAULT_TXT_COL_FG,
                        background_color=DEFAULT_TXT_COL_BG)
        rps_dob.x = x_pos
        rps_dob.y = y_pos
        y_pos += 60
        screen_group.append(rps_dob)

cursor_dob = Label(terminalio.FONT,
                   text=">",
                   scale=3,
                   color=CURSOR_COL_FG)
cursor_dob.x = left_col - 20
setCursor(my_choice_idx, "mine")
cursor_dob.y = top_y_pos
screen_group.append(cursor_dob)

# Initially set to a space to not show it
opp_cursor_dob = Label(terminalio.FONT,
                       text=" ",
                       scale=3,
                       color=OPP_CURSOR_COL_FG,
                       background_color=DEFAULT_TXT_COL_BG)
opp_cursor_dob.x = right_col - 20
setCursor(my_choice_idx, "your")
opp_cursor_dob.y = top_y_pos
screen_group.append(opp_cursor_dob)

display.root_group = screen_group

# From adafruit_ble.advertising
MANUFACTURING_DATA_ADT = 0xFF
ADAFRUIT_COMPANY_ID = 0x0822

# pylint: disable=line-too-long
# According to https://github.com/adafruit/Adafruit_CircuitPython_BLE/blob/master/
adafruit_ble/advertising/adafruit.py
# 0xf000 (to 0xffff) is for range for Adafruit customers
RPS_ACK_ID = 0xfe30
RPS_DATA_ID = 0xfe31


class RpsAdvertisement(Advertisement):
    """Broadcast an RPS message.
       This is not connectable and elicits no scan_response based on defaults
       in Advertisement parent class."""

    flags = None

    _PREFIX_FMT = "<BHBH"
    _DATA_FMT = "8s"  # this NUL pads if necessary

    # match_prefixes tuple replaces deprecated prefix
```

```
    # comma for 1 element is very important!
    match_prefixes = (
        struct.pack(
            _PREFIX_FMT,
            MANUFACTURING_DATA_ADT,
            ADAFRUIT_COMPANY_ID,
            struct.calcsize("<H" + _DATA_FMT),
            RPS_DATA_ID
        ),
    )
    manufacturer_data = LazyObjectField(
        ManufacturerData,
        "manufacturer_data",
        advertising_data_type=MANUFACTURING_DATA_ADT,
        company_id=ADAFRUIT_COMPANY_ID,
        key_encoding="<H",
    )

    test_string = ManufacturerDataField(RPS_DATA_ID, "<" + _DATA_FMT)
    """RPS choice."""


NS_IN_S = 1000 * 1000  * 1000
MIN_SEND_TIME_NS = 6 * NS_IN_S
MAX_SEND_TIME_S = 20
MAX_SEND_TIME_NS = MAX_SEND_TIME_S * NS_IN_S

# 20ms is the minimum delay between advertising packets
# in Bluetooth Low Energy
# extra 10us deals with API floating point rounding issues
MIN_AD_INTERVAL = 0.02001

ble = BLERadio()

opponent_choice = None

timeout = False
round_no = 1
wins = 0
losses = 0
draws = 0
voids = 0

TOTAL_ROUND = 5


def evaluate_game(mine, yours):
    """Determine who won the game based on the two strings mine and yours_lc.
       Returns three booleans (win, draw, void)."""
    # Return with void at True if any input is None
    try:
        mine_lc = mine.lower()
        yours_lc = yours.lower()
    except AttributeError:
        return (False, False, True)

    r_win = r_draw = r_void = False
    # pylint: disable=too-many-boolean-expressions
    if (mine_lc == "rock" and yours_lc == "rock"
            or mine_lc == "paper" and yours_lc == "paper"
            or mine_lc == "scissors" and yours_lc == "scissors"):
        r_draw = True
    elif (mine_lc == "rock" and yours_lc == "paper"):
        pass  # r_win default is False
    elif (mine_lc == "rock" and yours_lc == "scissors"):
        r_win = True
    elif (mine_lc == "paper" and yours_lc == "rock"):
        r_win = True
    elif (mine_lc == "paper" and yours_lc == "scissors"):
```

```
        pass  # r_win default is False
    elif (mine_lc == "scissors" and yours_lc == "rock"):
        pass  # r_win default is False
    elif (mine_lc == "scissors" and yours_lc == "paper"):
        r_win = True
    else:
        r_void = True

    return (r_win, r_draw, r_void)


# Advertise for 20 seconds maximum and if a packet is received
# for 5 seconds after that
while True:
    if round_no > TOTAL_ROUND:
        print("Summary: ",
              "wins {:d}, losses {:d}, draws {:d}, void {:d}".format(wins, losses,
draws, voids))

        # Reset variables for another game
        round_no = 1
        wins = 0
        losses = 0
        draws = 0
        voids = 0
        round_no = 1

    if button_left():
        while button_left():
            pass
        my_choice_idx = (my_choice_idx + 1) % len(choices)
        setCursor(my_choice_idx, "mine")

    if button_right():
        tx_message = RpsAdvertisement()

        choice = choices[my_choice_idx]
        tx_message.test_string = choice
        d_print(2, "TXing RTA", choice)

        opponent_choice = None
        ble.start_advertising(tx_message, interval=MIN_AD_INTERVAL)
        sending_ns = time.monotonic_ns()

        # Timeout value is in seconds
        # RSSI -100 is probably minimum, -128 would be 8bit signed min
        # window and interval are 0.1 by default - same value means
        # continuous scanning (sending Advertisement will interrupt this)
        for adv in ble.start_scan(RpsAdvertisement,
                                  minimum_rssi=-90,
                                  timeout=MAX_SEND_TIME_S):
            received_ns = time.monotonic_ns()
            d_print(2, "RXed RTA",
                    adv.test_string)
            opponent_choice_bytes = adv.test_string

            # Trim trailing NUL chars from bytes
            idx = 0
            while idx < len(opponent_choice_bytes):
                if opponent_choice_bytes[idx] == 0:
                    break
                idx += 1
            opponent_choice = opponent_choice_bytes[0:idx].decode("utf-8")
            break

        # We have received one message or exceeded MAX_SEND_TIME_S
        ble.stop_scan()

        # Ensure we send our message for a minimum period of time
```

```
        # constrained by the ultimate duration cap
        if opponent_choice is not None:
            timeout = False
            remaining_ns = MAX_SEND_TIME_NS - (received_ns - sending_ns)
            extra_ad_time_ns = min(remaining_ns, MIN_SEND_TIME_NS)
            # Only sleep if we need to, the value here could be a small
            # negative one too so this caters for this
            if extra_ad_time_ns > 0:
                sleep_t  = extra_ad_time_ns / NS_IN_S
                d_print(2, "Additional {:f} seconds of advertising".format(sleep_t))
                time.sleep(sleep_t)
        else:
            timeout = True

        ble.stop_advertising()

        d_print(1, "ROUND", round_no,
                "MINE", choice,
                "| OPPONENT", opponent_choice)
        win, draw, void = evaluate_game(choice, opponent_choice)

        if void:
            voids += 1
        else:
            opp_choice_idx = choices.index(opponent_choice)
            setCursor(opp_choice_idx, "opp", visibility="show")
            if draw:
                time.sleep(4)
                draws += 1
            elif win:
                flashWinner(my_choice_idx, "mine")
                wins += 1
            else:
                flashWinner(opp_choice_idx, "opp")
                losses += 1
            setCursor(opp_choice_idx, "opp", visibility="hide")
        d_print(1, "wins {:d}, losses {:d}, draws {:d}, void {:d}".format(wins,
losses, draws, voids))

        round_no += 1
```

Code Discussion

The main loop runs forever checking for three conditions.

```
while True:
    if round_no &gt; TOTAL_ROUND:
        print("Summary: ",
              "wins {:d}, losses {:d}, draws {:d}, void {:d}".format(wins, losses,
draws, voids))

        ### Reset variables for another game
        round_no = 1
        wins = 0
        losses = 0
        draws = 0
        voids = 0
        round_no = 1

    if button_left():
        while button_left():
            pass
        my_choice_idx = (my_choice_idx + 1) % len(choices)
        setCursor(my_choice_idx, "mine")
```

```
    if button_right():
```

The first `if` checks to see if the last round of the game has occurred in order to print a summary to the serial console and resets the per-round counters. The astute reader will spot an unintentional repetition of `round_no = 1` - this is harmless but does waste a few bytes of memory.

The second `if` checks to see if the left button has been pressed. A function is used here to allow the button to be customised for the board being used. The short `while` loop is waiting for the finger to release the button and then the `my_choice_idx` is incremented. This wraps around the three choices, the value after `2` is `0`. `setCursor()` updates the display, the second parameter is controlling whose selection needs updating. The success of waiting for `button_left()` not to be `True` (pressed) as a debounce mechanism is discussed further down.

The first part of the code inside the third `if` is shown below.

```
# Right button code excerpt 1/3
    if button_right():
        tx_message = RpsAdvertisement()

        choice = choices[my_choice_idx]
        tx_message.test_string = choice
        d_print(2, "TXing RTA", choice)

        opponent_choice = None
        ble.start_advertising(tx_message, interval=MIN_AD_INTERVAL)
        sending_ns = time.monotonic_ns()

        # Timeout value is in seconds
        # RSSI -100 is probably minimum, -128 would be 8bit signed min
        # window and interval are 0.1 by default - same value means
        # continuous scanning (sending Advertisement will interrupt this)
        for adv in ble.start_scan(RpsAdvertisement,
                                  minimum_rssi=-90,
                                  timeout=MAX_SEND_TIME_S):
            received_ns = time.monotonic_ns()
            d_print(2, "RXed RTA",
                    adv.test_string)
            opponent_choice_bytes = adv.test_string

            # Trim trailing NUL chars from bytes
            idx = 0
            while idx &lt; len(opponent_choice_bytes):
                if opponent_choice_bytes[idx] == 0:
                    break
                idx += 1
            opponent_choice = opponent_choice_bytes[0:idx].decode("utf-8")
            break

        # We have received one message or exceeded MAX_SEND_TIME_S
        ble.stop_scan()
```

This is creating a message using the `RpsAdvertisement` class and setting `test_string` (a poor name leftover from a prototype!) to the lower-case text representation of the player's choice. `start_advertising()` then starts sending

that data repeatedly by broadcasting an advertising packet - this occurs in the background until it's explicitly stopped. Advertisements are received with the use of `start_scan()` which is commonly used in a loop to iterate over each packet as it arrives. For a two-player game the code is only waiting for the opponent's choice. As soon as the first `RpsAdvertisement` packet is received it's complete in terms of receiving data, hence the `break` to terminate the `for` loop. A `stop_scan()` then terminates the scanning.

The data needs to be converted from the underlying fixed-size `bytes` type back to a text string and this involves removing any NUL padding. There's no guarantee this data is from a trusted source since it's an unauthenticated packet received over the air. A robust program would validate this data as soon as possible. A few other programming languages have a feature to tag risky data from less trusted sources which has not been validated, for example taint checking (https://adafru.it/Nb8).

The code then continues to send advertising packets as it does not know if these have been received. This is shown in the next code excerpt below.

```
# Right button code excerpt 2/3
        # Ensure we send our message for a minimum period of time
        # constrained by the ultimate duration cap
        if opponent_choice is not None:
            timeout = False
            remaining_ns = MAX_SEND_TIME_NS - (received_ns - sending_ns)
            extra_ad_time_ns = min(remaining_ns, MIN_SEND_TIME_NS)
            # Only sleep if we need to, the value here could be a small
            # negative one too so this caters for this
            if extra_ad_time_ns &gt; 0:
                sleep_t  = extra_ad_time_ns / NS_IN_S
                d_print(2, "Additional {:f} seconds of advertising".format(sleep_t))
                time.sleep(sleep_t)
        else:
            timeout = True

        ble.stop_advertising()

        d_print(1, "ROUND", round_no,
                "MINE", choice,
                "| OPPONENT", opponent_choice)
```

It does this for `MIN_SEND_TIME_NS` or less if that would exceed `MAX_SEND_TIME_NS` in total. `NS` stands for nanoseconds, billionths of a second. Nanosecond precision is not required here, it's simply the units returned by `time.monotonic_ns()` which is the most precise time function available (https://adafru.it/Nb9). The advertising packets are sent in the background, the code only needs to sleep for the calculated duration and then run `stop_advertising()`.

The final part of the code

1. checks who has won,
2. shows the opponent's choice on the display with `setCursor()`,

3. indicates a win or lose with `flashWinner()` and
4. then finally increments the integer variable `round_no`.

```
# Right button code excerpt 3/3
        win, draw, void = evaluate_game(choice, opponent_choice)

        if void:
            voids += 1
        else:
            opp_choice_idx = choices.index(opponent_choice)
            setCursor(opp_choice_idx, "opp", visibility="show")
            if draw:
                time.sleep(4)
                draws += 1
            elif win:
                flashWinner(my_choice_idx, "mine")
                wins += 1
            else:
                flashWinner(opp_choice_idx, "opp")
                losses += 1
            setCursor(opp_choice_idx, "opp", visibility="hide")
        d_print(1, "wins {:d}, losses {:d}, draws {:d}, void {:d}".format(wins,
losses, draws, voids))

        round_no += 1
```

The tempting variable name `round` should be avoided as this clashes with CircuitPython's `round()` function. The code would be valid and would run but it would be likely to cause confusion, bugs or both.

The `evaluate_game()` function is a little different to the technique used in the Very Simple game for deciding the winner. This version makes no use of lookup tables and is far longer to the extent that **pylint** doesn't like it. The `else` is only reached if the inputs are invalid - this is indicated by the `void` variable in the returned tuple being set to `True`. C/C++ programmers would instinctively avoid the use of `void` as a variable name as it's a reserved word (https://adafru.it/Nba) in those languages but Python does not use it.

```
def evaluate_game(mine, yours):
    """Determine who won the game based on the two strings mine and yours_lc.
       Returns three booleans (win, draw, void)."""
    # Return with void at True if any input is None
    try:
        mine_lc = mine.lower()
        yours_lc = yours.lower()
    except AttributeError:
        return (False, False, True)

    r_win = r_draw = r_void = False
    # pylint: disable=too-many-boolean-expressions
    if (mine_lc == "rock" and yours_lc == "rock"
            or mine_lc == "paper" and yours_lc == "paper"
            or mine_lc == "scissors" and yours_lc == "scissors"):
        r_draw = True
    elif (mine_lc == "rock" and yours_lc == "paper"):
        pass  # r_win default is False
    elif (mine_lc == "rock" and yours_lc == "scissors"):
        r_win = True
    elif (mine_lc == "paper" and yours_lc == "rock"):
```

```
        r_win = True
    elif (mine_lc == "paper" and yours_lc == "scissors"):
        pass  # r_win default is False
    elif (mine_lc == "scissors" and yours_lc == "rock"):
        pass  # r_win default is False
    elif (mine_lc == "scissors" and yours_lc == "paper"):
        r_win = True
    else:
        r_void = True

    return (r_win, r_draw, r_void)
```

The return type permits some "illegal" combinations of values. The first and second elements in the tuple could be `True` which means the player has both won and drawn. The current implementation is small enough to verify and will never return this combination. This would be more risky in a larger or more complex function.

A less typical form of assignment is used in `evaluate_game()`.

```
r_win = r_draw = r_void = False
```

This is a compact way to assign the same value to several variables. It works because Python is like many other languages and has a right-associative (https://adafru.it/Nbb) `=` operator. This means `r_void = False` occurs first, and then the result of this is the value `False` which is assigned to `r_draw` and then `r_win`.

Current Issues

Debouncing

A user who plays the game for a while will notice that occasionally the selection advances by two choices rather than one. This happens infrequently and it would be easy to dismiss this as a mysterious glitch but in this case it is a straightforward case of switch bounce (https://adafru.it/lif).

For this program the main `while` loop can iterate very rapidly as there's only a small amount of code to be run for the case of a left button press. If the display updates were immediate then this processing would probably take long enough for the button's contacts to stop bouncing but `displayio` updates are part of a background task - they happen soon after but not necessarily immediately.

This could be fixed with the help of the adafruit_debouncer (https://adafru.it/Nbc) library.

BLE - Advertising vs Connections

If a game is inherently a two-player game or there is no requirement or future possibility of a multi-player game then the more common connection-based approach is going to be superior in most cases.

Evolving the Game

The game's look (https://adafru.it/Nbd) is a bit dull and it doesn't make much use of the 240x240 graphical display. It also has a slow feel (https://adafru.it/Nbd) to the exchange of player's choices over BLE. The next version adds some graphics, sound and has a more sophisticated networking protocol for the exchange improving the responsiveness and facilitating more than two players.

Advanced Game

This is a multi-player version of rock, paper, scissors with:

- Simple `displayio` graphics.
- An announcer and sound effects implemented with sound samples in the `wav` format.
- A dynamic group for the players, formed after the button guide appears when the program starts at power-up (or reset).
- A more sophisticated data exchange protocol providing anti-cheat features and a more responsive feel.
- Configurable name for each player.
- A sorted score table per game for the players.

This has been tested with up to six players.

Example Video

The video below shows the game being played on four devices. Each player would normally hide their choice. Bluetooth facilitates this as the game works well up to 4m (13ft) apart. The boards have been placed near each other in this demonstratation to ease the video recording.

The sequence of events in the video:

- 00:00 The program has started running on three of the boards as those boards were reset about 2 seconds before the video starts.
- 00:04 The introduction screen on the three boards with screens.
- 00:08 One click on reset button of the Circuit Playground Bluefruit with no display. The introduction is shorter without a display, hence the staggered start.
- 00:10 The button guide scrolls into view.
- 00:21 The **join game** phase, the boards send and receive packets to establish the group for the game.
- 00:23 The names of the players and the rssi of the first advertising packet received are shown on the devices with screens. All devices briefly flash blue (this is disabled by default in the code) on the NeoPixels when an advertising packet is received in the **join game** phase.
- 00:44 The boards conclude their search for other players and then commence a four player game with three rounds per game.
- 01:00 All four players have made their choice and are exchanging them, the game announces "Ready" and hums during transmission.
- 01:14 Exchange of data concludes and results from the first round are shown and announced on each device.
- 01:56 Second round begins.
- 02:38 Third and final round begins.
- 02:53 The Circuit Playground Bluefruit shows the scores on the NeoPixels.
- 03:07 The other three boards with displays show the scores for the game and sort them into descending order.

Installing Project Code

To use with CircuitPython, you need to first install a few libraries, into the lib folder on your **CIRCUITPY** drive. Then you need to update **code.py** with the example script.

Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CLUE_Rock_Paper_Scissors/advanced/** and then click on the directory that matches

the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:

CIRCUITPY

```
# SPDX-FileCopyrightText: 2020 Kevin J Walters for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# clue-multi-rpsgame v1.20
# CircuitPython massively multiplayer rock paper scissors game over Bluetooth LE

# Tested with CLUE and Circuit Playground Bluefruit Alpha with TFT Gizmo
# using CircuitPython and 5.3.0

# copy this file to CLUE/CPB board as code.py

# MIT License

# Copyright (c) 2020 Kevin J. Walters

# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:

# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.


import gc
import os
import random

from micropython import const
import board
import digitalio

import neopixel
from adafruit_ble import BLERadio

# These imports works on CLUE, CPB (and CPX on 5.x)
try:
    from audioio import AudioOut
except ImportError:
    from audiopwmio import PWMAudioOut as AudioOut

# RPS module files
from rps_advertisements import JoinGameAdvertisement, \
                               RpsEncDataAdvertisement, \
                               RpsKeyDataAdvertisement, \
                               RpsRoundEndAdvertisement
from rps_audio import SampleJukebox
```

```
from rps_comms import broadcastAndReceive, addrToText, MIN_AD_INTERVAL
from rps_crypto import bytesPad, strUnpad, generateOTPadKey, \
                       enlargeKey, encrypt, decrypt
from rps_display import RPSDisplay, blankScreen


# Look for our name in secrets.py file if present
ble_name = None
try:
    from secrets import secrets
    ble_name = secrets.get("rps_name")
    if ble_name is None:
        ble_name = secrets.get("ble_name")
        if ble_name is None:
            print("INFO: No rps_name or ble_name entry found in secrets dict")
except ImportError:
    pass   # File is optional, reaching here is not a program error


debug = 1

def d_print(level, *args, **kwargs):
    """A simple conditional print for debugging based on global debug level."""
    if not isinstance(level, int):
        print(level, *args, **kwargs)
    elif debug >= level:
        print(*args, **kwargs)


def tftGizmoPresent():
    """Determine if the TFT Gizmo is attached.
       The TFT's Gizmo circuitry for backlight features a 10k pull-down resistor.
       This attempts to verify the presence of the pull-down to determine
       if TFT Gizmo is present.
       This is likely to get confused if anything else is connected to pad A3.
       Only use this on Circuit Playground Express (CPX)
       or Circuit Playground Bluefruit (CPB) boards."""
    present = True
    try:
        with digitalio.DigitalInOut(board.A3) as backlight_pin:
            backlight_pin.pull = digitalio.Pull.UP
            present = not backlight_pin.value
    except ValueError:
        # The Gizmo is already initialised, i.e. showing console output
        pass

    return present


# Assuming CLUE if it's not a Circuit Playround (Bluefruit)
clue_less = "Circuit Playground" in os.uname().machine

# Note: difference in pull-up and pull-down
#       and logical not use for buttons
if clue_less:
    # CPB with TFT Gizmo (240x240)
    # from adafruit_circuitplayground import cp  # Avoiding to save memory

    # Outputs
    if tftGizmoPresent():
        from adafruit_gizmo import tft_gizmo
        display = tft_gizmo.TFT_Gizmo()
        JG_RX_COL = 0x0000ff
        BUTTON_Y_POS = 120
    else:
        display = None
        JG_RX_COL = 0x000030  # dimmer blue for upward facing CPB NeoPixels
        BUTTON_Y_POS = None
```

```
    audio_out = AudioOut(board.SPEAKER)
    #pixels = cp.pixels
    pixels = neopixel.NeoPixel(board.NEOPIXEL, 10)

    # Enable the onboard amplifier for speaker
    #cp._speaker_enable.value = True  # pylint: disable=protected-access
    speaker_enable = digitalio.DigitalInOut(board.SPEAKER_ENABLE)
    speaker_enable.switch_to_output(value=False)
    speaker_enable.value = True

    # Inputs
    # buttons reversed if it is used upside-down with Gizmo
    _button_a = digitalio.DigitalInOut(board.BUTTON_A)
    _button_a.switch_to_input(pull=digitalio.Pull.DOWN)
    _button_b = digitalio.DigitalInOut(board.BUTTON_B)
    _button_b.switch_to_input(pull=digitalio.Pull.DOWN)
    if display is None:
        def button_left():
            return _button_a.value
        def button_right():
            return _button_b.value
    else:
        def button_left():
            return _button_b.value
        def button_right():
            return _button_a.value

else:
    # CLUE with builtin screen (240x240)
    # from adafruit_clue import clue  # Avoiding to save memory

    # Outputs
    display = board.DISPLAY
    audio_out = AudioOut(board.SPEAKER)
    #pixels = clue.pixel
    pixels = neopixel.NeoPixel(board.NEOPIXEL, 1)
    JG_RX_COL = 0x0000ff
    BUTTON_Y_POS = 152

    # Inputs
    _button_a = digitalio.DigitalInOut(board.BUTTON_A)
    _button_a.switch_to_input(pull=digitalio.Pull.UP)
    _button_b = digitalio.DigitalInOut(board.BUTTON_B)
    _button_b.switch_to_input(pull=digitalio.Pull.UP)
    def button_left():
        return not _button_a.value
    def  button_right():
        return not _button_b.value


blankScreen(display, pixels)

# Set to True for blue flashing when devices are joining the playing group
JG_FLASH = False

IMAGE_DIR = "rps/images"
AUDIO_DIR = "rps/audio"

audio_files = (("searching", "welcome-to", "arena", "ready")
               + ("rock", "paper", "scissors")
               + ("start-tx", "end-tx", "txing")
               + ("rock-scissors", "paper-rock", "scissors-paper")
               + ("you-win", "draw", "you-lose", "error")
               + ("humiliation", "excellent"))

gc.collect()
d_print(2, "GC before SJ", gc.mem_free())
sample = SampleJukebox(audio_out, audio_files,
                       directory=AUDIO_DIR)
```

```
del audio_files  # not needed anymore
gc.collect()
d_print(2, "GC after SJ", gc.mem_free())

# A lookup table in Dict form for win/lose, each value is a sample name
# Does not need to cater for draw (tie) condition
WAV_VICTORY_NAME = { "rp": "paper-rock",
                     "pr": "paper-rock",
                     "ps": "scissors-paper",
                     "sp": "scissors-paper",
                     "sr": "rock-scissors",
                     "rs": "rock-scissors"}

# This limit is based on displaying names on screen with scale=2 font
MAX_PLAYERS = 8
# Some code is dependent on these being lower-case
CHOICES = ("rock", "paper", "scissors")

rps_display = RPSDisplay(display, pixels,
                         CHOICES, sample, WAV_VICTORY_NAME,
                         MAX_PLAYERS, BUTTON_Y_POS,
                         IMAGE_DIR + "/rps-sprites-ind4.bmp",
                         ble_color=JG_RX_COL)

# Transmit maximum times in seconds
JG_MSG_TIME_S = 20
FIRST_MSG_TIME_S = 12
STD_MSG_TIME_S = 4
LAST_ACK_TIME_S = 1.5


# Intro screen with audio
rps_display.introductionScreen()

# Enable the Bluetooth LE radio and set player's name (from secrets.py)
ble = BLERadio()
if ble_name is not None:
    ble.name = ble_name


game_no = 1
round_no = 1
wins = losses = draws = voids = 0

# TOTAL_ROUNDS = 5
TOTAL_ROUNDS = 3

CRYPTO_ALGO = "chacha20"
KEY_SIZE = 8  # in bytes
KEY_ENLARGE = 256 // KEY_SIZE // 8

# Scoring values
POINTS_WIN = 2
POINTS_DRAW = 1

WIN = const(1)
DRAW = const(2)  # AKA tie
LOSE = const(3)
INVALID = const(4)

def evaluateRound(mine, yours):
    """Determine who won the round in this game based on the two strings mine and
yours.
       Returns WIN, DRAW, LOSE or INVALID for bad input."""
    # Return INVALID if any input is None
    try:
        mine_lc = mine.lower()
        yours_lc = yours.lower()
    except AttributeError:
```

```
        return INVALID

    if mine_lc not in CHOICES or yours_lc not in CHOICES:
        return INVALID

    # Both inputs are valid choices if we got this far
    # pylint: disable=too-many-boolean-expressions
    if mine_lc == yours_lc:
        return DRAW
    elif (mine_lc == "rock" and yours_lc == "scissors"
          or mine_lc == "paper" and yours_lc == "rock"
          or mine_lc == "scissors" and yours_lc == "paper"):
        return WIN

    return LOSE


rps_display.playerListScreen()

def addPlayer(name, addr_text, address, ad):
    # pylint: disable=unused-argument
    # address is part of call back
    """Add the player name and mac address to players global variable
       and the name and rssi (if present) to on-screen list."""

    rssi = ad.rssi if ad else None

    players.append((name, addr_text))
    rps_display.addPlayer(name, rssi=rssi)


# Make a list of all the player's (name, mac address as text)
# where both are strings with this player as first entry
players = []
my_name = ble.name
rps_display.fadeUpDown("down")
addPlayer(my_name, addrToText(ble.address_bytes), None, None)


# These two functions mainly serve to adapt the call back arguments
# to the called functions which do not use them
def jgAdCallbackFlashBLE(_a, _b, _c):
    """Used in broadcastAndReceive to flash the NeoPixels
       when advertising messages are received."""
    return rps_display.flashBLE()

def jgEndscanCallback(_a, _b, _c):
    """Used in broadcastAndReceive to allow early termination of the scanning
       when the left button is pressed.
       Button may need to be held down for a second."""
    return button_left()

# Join Game
gc.collect()
d_print(2, "GC before JG", gc.mem_free())

sample.play("searching", loop=True)
rps_display.fadeUpDown("up")
jg_msg = JoinGameAdvertisement(game="RPS")
(_, _, _) = broadcastAndReceive(ble,
                                jg_msg,
                                scan_time=JG_MSG_TIME_S,
                                scan_response_request=True,
                                ad_cb=(jgAdCallbackFlashBLE
                                       if JG_FLASH
                                       else None),
                                endscan_cb=jgEndscanCallback,
                                name_cb=addPlayer)
del _  # To clean-up with GC below
```

```
sample.stop()
gc.collect()
d_print(2, "GC after JG", gc.mem_free())

# Wait for button release - this stops a long press
# being acted upon in the main loop further down
while button_left():
    pass

scores = [0] * len(players)
num_other_players = len(players) - 1

# Set the advertising interval to the minimum for four or fewer players
# and above that extend value by players multiplied by 7ms
ad_interval = MIN_AD_INTERVAL if len(players) <= 4 else len(players) * 0.007

d_print(1, "PLAYERS", players)

# Sequence numbers - real packets start range between 1-255 inclusive
seq_tx = [1]  # The next number to send

new_round_init = True

# A nonce by definition must not be reused but here a random key is
# generated per round and this is used once per round so this is ok
static_nonce = bytes(range(12, 0, -1))

while True:
    if round_no > TOTAL_ROUNDS:
        print("Summary: ",
              "wins {:d}, losses {:d},"
              " draws {:d}, void {:d}\n\n".format(wins, losses, draws, voids))

        rps_display.showGameResult(players, scores, rounds_tot=TOTAL_ROUNDS)

        # Reset variables for another game
        round_no = 1
        wins = losses = draws = voids = 0
        scores = [0] * len(players)
        game_no += 1

    if new_round_init:
        rps_display.showGameRound(game_no=game_no, round_no=round_no,
rounds_tot=TOTAL_ROUNDS)
        # Make a new initial random choice for the player and show it
        my_choice_idx = random.randrange(len(CHOICES))
        rps_display.fadeUpDown("down")
        rps_display.showChoice(my_choice_idx,
                               game_no=game_no, round_no=round_no,
rounds_tot=TOTAL_ROUNDS,
                               won_sf=wins, drew_sf=draws, lost_sf=losses)
        rps_display.fadeUpDown("up")
        new_round_init = False

    if button_left():
        while button_left():  # Wait for button release
            pass
        my_choice_idx = (my_choice_idx + 1) % len(CHOICES)
        rps_display.showChoice(my_choice_idx,
                               game_no=game_no, round_no=round_no,
rounds_tot=TOTAL_ROUNDS,
                               won_sf=wins, drew_sf=draws, lost_sf=losses)

    if button_right():
        gc.collect()
        d_print(2, "GC before comms", gc.mem_free())

        # This sound cue is really for other players
        sample.play("ready")
```

```
        my_choice = CHOICES[my_choice_idx]
        player_choices = [my_choice]

        # Repeating key four times to make key for ChaCha20
        short_key = generateOTPadKey(KEY_SIZE)
        key = enlargeKey(short_key, KEY_ENLARGE)
        d_print(3, "KEY", key)

        plain_bytes = bytesPad(my_choice, size=8, pad=0)
        cipher_bytes = encrypt(plain_bytes, key, CRYPTO_ALGO,
                               nonce=static_nonce)
        enc_data_msg = RpsEncDataAdvertisement(enc_data=cipher_bytes,
                                               round_no=round_no)

        # Wait for ready sound sample to stop playing
        sample.wait()
        sample.play("start-tx")
        sample.wait()
        sample.play("txing", loop=True)
        # Players will not be synchronised at this point as they do not
        # have to make their choices simultaneously - much longer 12 second
        # time to accomodate this
        _, enc_data_by_addr, _ = broadcastAndReceive(ble,
                                                     enc_data_msg,
                                                     RpsEncDataAdvertisement,
                                                     RpsKeyDataAdvertisement,
                                                     scan_time=FIRST_MSG_TIME_S,
                                                     ad_interval=ad_interval,
                                                     receive_n=num_other_players,
                                                     seq_tx=seq_tx)

        key_data_msg = RpsKeyDataAdvertisement(key_data=short_key,
round_no=round_no)
        # All of the programs will be loosely synchronised now
        _, key_data_by_addr, _ = broadcastAndReceive(ble,
                                                     key_data_msg,
                                                     RpsEncDataAdvertisement,
                                                     RpsKeyDataAdvertisement,
                                                     RpsRoundEndAdvertisement,
                                                     scan_time=STD_MSG_TIME_S,
                                                     ad_interval=ad_interval,
                                                     receive_n=num_other_players,
                                                     seq_tx=seq_tx,
                                                     ads_by_addr=enc_data_by_addr)
        del enc_data_by_addr

        # Play end transmit sound while doing next decrypt bit
        sample.play("end-tx")

        re_msg = RpsRoundEndAdvertisement(round_no=round_no)
        # The round end message is really about acknowledging receipt of
        # the key_data_msg by sending a non-critical message with the ack
        _, re_by_addr, _ = broadcastAndReceive(ble,
                                               re_msg,
                                               RpsEncDataAdvertisement,
                                               RpsKeyDataAdvertisement,
                                               RpsRoundEndAdvertisement,
                                               scan_time=LAST_ACK_TIME_S,
                                               ad_interval=ad_interval,
                                               receive_n=num_other_players,
                                               seq_tx=seq_tx,
                                               ads_by_addr=key_data_by_addr)
        del key_data_by_addr, _  # To allow GC

        # This will have accumulated all the messages for this round
        allmsg_by_addr = re_by_addr
        del re_by_addr
```

```
    # Decrypt results
    # If any data is incorrect the opponent_choice is left as None
    for p_idx1 in range(1, len(players)):
        print("DECRYPT GC", p_idx1, gc.mem_free())
        opponent_name = players[p_idx1][0]
        opponent_macaddr = players[p_idx1][1]
        opponent_choice = None
        opponent_msgs = allmsg_by_addr.get(opponent_macaddr)
        if opponent_msgs is None:
            opponent_msgs = []
        cipher_ad = cipher_bytes = cipher_round = None
        key_ad = key_bytes = key_round = None
        # There should be either one or two messges per type
        # two occurs when there
        for msg_idx in range(len(opponent_msgs)):
            if (cipher_ad is None
                    and isinstance(opponent_msgs[msg_idx][0],
                                   RpsEncDataAdvertisement)):
                cipher_ad = opponent_msgs[msg_idx][0]
                cipher_bytes = cipher_ad.enc_data
                cipher_round = cipher_ad.round_no
            elif (key_ad is None
                  and isinstance(opponent_msgs[msg_idx][0],
                                 RpsKeyDataAdvertisement)):
                key_ad = opponent_msgs[msg_idx][0]
                key_bytes = key_ad.key_data
                key_round = key_ad.round_no

        if cipher_ad and key_ad:
            if round_no == cipher_round == key_round:
                key = enlargeKey(key_bytes, KEY_ENLARGE)
                plain_bytes = decrypt(cipher_bytes, key, CRYPTO_ALGO,
                                      nonce=static_nonce)
                opponent_choice = strUnpad(plain_bytes)
            else:
                print("Received wrong round for {:d} {:d}: {:d} {:d}",
                      opponent_name, round_no, cipher_round, key_round)
        else:
            print("Missing packets: RpsEncDataAdvertisement "
                  "and RpsKeyDataAdvertisement:", cipher_ad, key_ad)
        player_choices.append(opponent_choice)

    # Free up some memory by deleting any data that's no longer needed
    del allmsg_by_addr
    gc.collect()
    d_print(2, "GC after comms", gc.mem_free())

    sample.wait()  # Ensure end-tx has completed

    # Chalk up wins and losses - checks this player but also has to
    # check other players against each other to calculate all the
    # scores for the high score table at the end of game
    for p_idx0, (p0_name, _) in enumerate(players[:len(players) - 1]):
        for p_idx1, (p1_name, _) in enumerate(players[p_idx0 + 1:], p_idx0 + 1):
            # evaluateRound takes text strings for RPS
            result = evaluateRound(player_choices[p_idx0],
                                   player_choices[p_idx1])

            # this_player is used to control incrementing the summary
            # for the tally for this local player
            this_player = 0
            void = False
            if p_idx0 == 0:
                this_player = 1
                p0_ch_idx = None
                p1_ch_idx = None
                try:
                    p0_ch_idx = CHOICES.index(player_choices[p_idx0])
                    p1_ch_idx = CHOICES.index(player_choices[p_idx1])
```

```
                except ValueError:
                    void = True  # Ensure this is marked void
                    print("ERROR", "failed to decode",
                          player_choices[p_idx0], player_choices[p_idx1])

                # showPlayerVPlayer takes int index values for RPS
                rps_display.showPlayerVPlayer(p0_name, p1_name, p_idx1,
                                              p0_ch_idx, p1_ch_idx,
                                              result == WIN,
                                              result == DRAW,
                                              result == INVALID or void)

            if result == INVALID or void:
                voids += this_player
            elif result == DRAW:
                draws += this_player
                scores[p_idx0] += POINTS_DRAW
                scores[p_idx1] += POINTS_DRAW
            elif result == WIN:
                wins += this_player
                scores[p_idx0] += POINTS_WIN
            else:
                losses += this_player
                scores[p_idx1] += POINTS_WIN

            d_print(2,
                    p0_name, player_choices[p_idx0], "vs",
                    p1_name, player_choices[p_idx1],
                    "result", result)

        print("Game {:d}, round {:d}, wins {:d}, losses {:d}, draws {:d}, "
              "void {:d}".format(game_no, round_no, wins, losses, draws, voids))

        round_no += 1
        new_round_init = True
```

Configuration

If you wish, the player's name can be set by adding an entry to the **settings.toml** file. This can be either `"rps_name"` or `"ble_name"`, for example:

```
rps_name="Huey"
```

The file is typically used by projects for confidential information like credentials (https://adafru.it/Nbf). Here, it's just a useful file to reuse as it keeps the configuration separate from the code.

Code Discussion

This is a much larger program. It's split into a number of files with related functions and there's some limited use of classes.

The clue and cpb Objects

This program does not use the clue (https://adafru.it/Ka-) and cpb (https://adafru.it/CuQ) objects from the libraries `adafruit_clue` and `adafruit_circuitplayground.bluefruit`, respectively. These are very useful for

interactive experimentation and small programs but they do `import` a lot of libraries. For larger programs like the Advanced game that do not use most of the functionality, it is more economical on memory to `import` the specific libraries used in the program.

Join Game

After the introduction screen and button guide, the game establishes the group playing with a `JoinGameAdvertisement` message sent amongst all the boards that want to play the game. The code is show below.

```
def addPlayer(name, addr_text, address, ad):
    # pylint: disable=unused-argument
    # address is part of call back
    """Add the player name and mac address to players global variable
       and the name and rssi (if present) to on-screen list."""

    rssi = ad.rssi if ad else None

    players.append((name, addr_text))
    rps_display.addPlayer(name, rssi=rssi)


# Make a list of all the player's (name, mac address as text)
# where both are strings with this player as first entry
players = []
my_name = ble.name
rps_display.fadeUpDown("down")
addPlayer(my_name, addrToText(ble.address_bytes), None, None)


# These two functions mainly serve to adapt the call back arguments
# to the called functions which do not use them
def jgAdCallbackFlashBLE(_a, _b, _c):
    """Used in broadcastAndReceive to flash the NeoPixels
       when advertising messages are received."""
    return rps_display.flashBLE()

def jgEndscanCallback(_a, _b, _c):
    """Used in broadcastAndReceive to allow early termination of the scanning
       when the left button is pressed.
       Button may need to be held down for a second."""
    return button_left()

# Join Game
gc.collect()
d_print(2, "GC before JG", gc.mem_free())

sample.play("searching", loop=True)
rps_display.fadeUpDown("up")
jg_msg = JoinGameAdvertisement(game="RPS")
(_, _, _) = broadcastAndReceive(ble,
                                jg_msg,
                                scan_time=JG_MSG_TIME_S,
                                scan_response_request=True,
                                ad_cb=(jgAdCallbackFlashBLE
                                       if JG_FLASH
                                       else None),
                                endscan_cb=jgEndscanCallback,
                                name_cb=addPlayer)
del _   # To clean-up with GC below
sample.stop()
```

```
gc.collect()
d_print(2, "GC after JG", gc.mem_free())
```

The `addPlayer()` function adds a player to the the `players` list, a global variable, and calls the `addPlayer()` method on the (global) `rps_display` object to add the player's details to the list on the display. This is used once to add the local player and then passed as a callback (https://adafru.it/Nbg) to `broadcastAndReceive()` to add the remote players as the device receives `JoinGameAdvertisement` from other players. The use of the callback means the return values are not needed from `broadcastAndReceive()` - a Python convention is to assign these to the `_` variable and in this case only the third element in the tuple will survive, the rest are over-written. This isn't needed and in order to minimise memory in use it is deleted. This makes the variable's previous value available for garbage collection (https://adafru.it/FI5) by the subsequent `gc.collect()`.

Some other callbacks are used here:

- `ad_cb` is called whenever an `Advertisement` is received and is used for flashing the NeoPixel's blue if `JG_FLASH` is `True` (code currently has it set to `False`);
- `endscan_db` is called periodically and will terminate the scanning if it returns a `True` value, a left button press returns `True` here.

A sound effect is playing continuously during this part of the program, the `sample` object is a `SampleJukebox` which is described further down.

Main Loop

The main loop has the same three conditional statements as the previous game plus one extra one to deal with rounds per game.

The first `if` checks for the end of each game, displaying the score on the display or NeoPixels using `rps_display.showGameResult()` and then resetting all the variables for the next game.

```
# Main loop code excerpt 1/8 - end of a game
while True:
    if round_no &gt; TOTAL_ROUNDS:
        print("Summary: ",
              "wins {:d}, losses {:d},"
              " draws {:d}, void {:d}\n\n".format(wins, losses,
                                                  draws, voids))

        rps_display.showGameResult(players, scores,
                                   rounds_tot=TOTAL_ROUNDS)

        # Reset variables for another game
        round_no = 1
        wins = 0
        losses = 0
```

```
        draws = 0
        voids = 0
        scores = [0] * len(players)
        game_no += 1
```

The second `if` checks a boolean to see if it is the start of a new round. The game and round number are updated with `rps_display.showGameRound()` and then a random starting choice is made for the player and displayed on screen using the `rps_display.showChoice()` with surrounding fades for a visually pleasant transition.

```
# Main loop code excerpt 2/8 - end of a round
    if new_round_init:
        rps_display.showGameRound(game_no=game_no, round_no=round_no,
                                  rounds_tot=TOTAL_ROUNDS)
        # Make a new initial random choice for the player and show it
        my_choice_idx = random.randrange(len(CHOICES))
        rps_display.fadeUpDown("down")
        rps_display.showChoice(my_choice_idx,
                               game_no=game_no, round_no=round_no,
                               rounds_tot=TOTAL_ROUNDS,
                               won_sf=wins, drew_sf=draws,
                               lost_sf=losses)
        rps_display.fadeUpDown("up")
        new_round_init = False
```

The `if` for the left button is very similar to the Simple game with the `rps_display.` Here, `showChoice()` takes the place of `setCursor()` from the Simple game.

```
# Main loop code excerpt 3/8 - left button press
    if button_left():
        while button_left():  ### Wait for button release
            pass
        my_choice_idx = (my_choice_idx + 1) % len(CHOICES)
        rps_display.showChoice(my_choice_idx,
                               game_no=game_no, round_no=round_no,
                               rounds_tot=TOTAL_ROUNDS,
                               won_sf=wins, drew_sf=draws,
                               lost_sf=losses)
```

The final `if` for the right button contains a lot of code for the exchange of choices between all the players. This would be better if it was split into some functions to make the code easier to read and understand.

The first part creates a short, per-message 8 byte encryption `key` with `generateOTPPadKey()` stretched with `enlargeKey()`. The player's choice is padded and and encrypted and incorporated into the `RpsEncDataAdvertisement` object. The padding here just adds NUL characters to ensure the message is 8 bytes long.

Padding schemes like PKCS#5 (https://adafru.it/Naj) are used for real applications using encryption.

```
# Main loop code excerpt 5/8 - right button press i
    if button_right():
        gc.collect()
        d_print(2, "GC before comms", gc.mem_free())

        # This sound cue is really for other players
        sample.play("ready")

        my_choice = CHOICES[my_choice_idx]
        player_choices = [my_choice]

        # Repeating key four times to make key for ChaCha20
        short_key = generateOTPadKey(KEY_SIZE)
        key = enlargeKey(short_key, KEY_ENLARGE)
        d_print(3, "KEY", key)

        plain_bytes = bytesPad(my_choice, size=8, pad=0)
        cipher_bytes = encrypt(plain_bytes, key, CRYPTO_ALGO,
                               nonce=static_nonce)
        enc_data_msg = RpsEncDataAdvertisement(enc_data=cipher_bytes,
                                               round_no=round_no)

        # Wait for ready sound sample to stop playing
        sample.wait()
```

The next part, shown below, starts the humming sound sample which runs while the program is sending messages. The three messages, `RpsEncDataAdvertisement`, `RpsKeyDataAdvertisement` and `RpsRoundEndAdvertisement`, are sent in that order and only when the prior message has been received from all other players. The data from the other players' messages is accumulated and left in `allmsg_by_addr`.

```
# Main loop code excerpt 6/8 - right button press ii
        sample.play("start-tx")
        sample.wait()
        sample.play("txing", loop=True)
        # Players will not be synchronised at this point as they do not
        # have to make their choices simultaneously - much longer 12 second
        # time to accomodate this
        _, enc_data_by_addr, _ = broadcastAndReceive(ble,
                                                     enc_data_msg,
                                                     RpsEncDataAdvertisement,
                                                     RpsKeyDataAdvertisement,
                                                     scan_time=FIRST_MSG_TIME_S,
                                                     ad_interval=ad_interval,
                                                     receive_n=num_other_players,
                                                     seq_tx=seq_tx)

        key_data_msg = RpsKeyDataAdvertisement(key_data=short_key,
round_no=round_no)
        # All of the programs will be loosely synchronised now
        _, key_data_by_addr, _ = broadcastAndReceive(ble,
                                                     key_data_msg,
                                                     RpsEncDataAdvertisement,
                                                     RpsKeyDataAdvertisement,
                                                     RpsRoundEndAdvertisement,
                                                     scan_time=STD_MSG_TIME_S,
                                                     ad_interval=ad_interval,
                                                     receive_n=num_other_players,
                                                     seq_tx=seq_tx,
                                                     ads_by_addr=enc_data_by_addr)
        del enc_data_by_addr

        # Play end transmit sound while doing next decrypt bit
```

```
        sample.play("end-tx")

        re_msg = RpsRoundEndAdvertisement(round_no=round_no)
        # The round end message is really about acknowledging receipt of
        # the key_data_msg by sending a non-critical message with the ack
        _, re_by_addr, _ = broadcastAndReceive(ble,
                                               re_msg,
                                               RpsEncDataAdvertisement,
                                               RpsKeyDataAdvertisement,
                                               RpsRoundEndAdvertisement,
                                               scan_time=LAST_ACK_TIME_S,
                                               ad_interval=ad_interval,
                                               receive_n=num_other_players,
                                               seq_tx=seq_tx,
                                               ads_by_addr=key_data_by_addr)
        del key_data_by_addr, _  # To allow GC

        # This will have accumulated all the messages for this round
        allmsg_by_addr = re_by_addr
        del re_by_addr
```

To free up as much memory as possible any data structures not needed at this point are `del`'ed. The other players' message(s) are then decrypted.

```
# Main loop code excerpt 6/8 - right button press iii
        # Decrypt results
        # If any data is incorrect the opponent_choice is left as None
        for p_idx1 in range(1, len(players)):
            print("DECRYPT GC", p_idx1, gc.mem_free())
            opponent_name = players[p_idx1][0]
            opponent_macaddr = players[p_idx1][1]
            opponent_choice = None
            opponent_msgs = allmsg_by_addr.get(opponent_macaddr)
            if opponent_msgs is None:
                opponent_msgs = []
            cipher_ad = cipher_bytes = cipher_round = None
            key_ad = key_bytes = key_round = None
            # There should be either one or two messges per type
            # two occurs when there
            for msg_idx in range(len(opponent_msgs)):
                if (cipher_ad is None
                        and isinstance(opponent_msgs[msg_idx][0],
                                       RpsEncDataAdvertisement)):
                    cipher_ad = opponent_msgs[msg_idx][0]
                    cipher_bytes = cipher_ad.enc_data
                    cipher_round = cipher_ad.round_no
                elif (key_ad is None
                      and isinstance(opponent_msgs[msg_idx][0],
                                     RpsKeyDataAdvertisement)):
                    key_ad = opponent_msgs[msg_idx][0]
                    key_bytes = key_ad.key_data
                    key_round = key_ad.round_no

            if cipher_ad and key_ad:
                if round_no == cipher_round == key_round:
                    key = enlargeKey(key_bytes, KEY_ENLARGE)
                    plain_bytes = decrypt(cipher_bytes, key, CRYPTO_ALGO,
                                          nonce=static_nonce)
                    opponent_choice = strUnpad(plain_bytes)
                else:
                    print("Received wrong round for {:d} {:d}: {:d} {:d}",
                          opponent_name, round_no, cipher_round, key_round)
            else:
                print("Missing packets: RpsEncDataAdvertisement "
                      "and RpsKeyDataAdvertisement:", cipher_ad, key_ad)
            player_choices.append(opponent_choice)
```

```
        # Free up some memory by deleting any data that's no longer needed
        del allmsg_by_addr
        gc.collect()
        d_print(2, "GC after comms", gc.mem_free())
```

The decrypted choices of the opponents are now checked against the local player's choice to show who has won. Perhaps surprisingly, if there's more than one opponent then they are checked against each other. This is required to calculate the complete score table for all the players shown at the end of each game.

```
# Main loop code excerpt 7/8 - right button press iv
        sample.wait()  # Ensure end-tx has completed

        # Chalk up wins and losses - checks this player but also has to
        # check other players against each other to calculate all the
        # scores for the high score table at the end of game
        for p_idx0, (p0_name, _) in enumerate(players[:len(players) - 1]):
            for p_idx1, (p1_name, _) in enumerate(players[p_idx0 + 1:], p_idx0 + 1):
                # evaluateRound takes text strings for RPS
                result = evaluateRound(player_choices[p_idx0],
                                       player_choices[p_idx1])

                # this_player is used to control incrementing the summary
                # for the tally for this local player
                this_player = 0
                void = False
                if p_idx0 == 0:
                    this_player = 1
                    p0_ch_idx = None
                    p1_ch_idx = None
                    try:
                        p0_ch_idx = CHOICES.index(player_choices[p_idx0])
                        p1_ch_idx = CHOICES.index(player_choices[p_idx1])
                    except ValueError:
                        void = True  # Ensure this is marked void
                        print("ERROR", "failed to decode",
                              player_choices[p_idx0], player_choices[p_idx1])

                    # showPlayerVPlayer takes int index values for RPS
                    rps_display.showPlayerVPlayer(p0_name, p1_name, p_idx1,
                                                  p0_ch_idx, p1_ch_idx,
                                                  result == WIN,
                                                  result == DRAW,
                                                  result == INVALID or void)
```

Finally the results from the round are added to the sub-totals.

```
# Main loop code excerpt 8/8 - right button press v
               if result == INVALID or void:
                    voids += this_player
                elif result == DRAW:
                    draws += this_player
                    scores[p_idx0] += POINTS_DRAW
                    scores[p_idx1] += POINTS_DRAW
                elif result == WIN:
                    wins += this_player
                    scores[p_idx0] += POINTS_WIN
                else:
                    losses += this_player
                    scores[p_idx1] += POINTS_WIN
```

```
                d_print(2,
                        p0_name, player_choices[p_idx0], "vs",
                        p1_name, player_choices[p_idx1],
                        "result", result)

        print("Game {:d}, round {:d}, wins {:d}, losses {:d}, draws {:d}, "
              "void {:d}".format(game_no, round_no, wins, losses, draws, voids))

        round_no += 1
        new_round_init = True
```

The `evaluateRound()` function has a different style of implementation compared to the Simple version.

```
def evaluateRound(mine, yours):
    """Determine who won the round in this game based on the two strings mine and
yours.
       Returns WIN, DRAW, LOSE or INVALID for bad input."""
    # Return INVALID if any input is None
    try:
        mine_lc = mine.lower()
        yours_lc = yours.lower()
    except AttributeError:
        return INVALID

    if mine_lc not in CHOICES or yours_lc not in CHOICES:
        return INVALID

    # Both inputs are valid choices if we got this far
    if mine_lc == yours_lc:
        return DRAW
    elif (mine_lc == "rock" and yours_lc == "scissors"
          or mine_lc == "paper" and yours_lc == "rock"
          or mine_lc == "scissors" and yours_lc == "paper"):
        return WIN

    return LOSE
```

This more compact version makes use of Python's in (https://adafru.it/Nbh) operator to check if the inputs are present in the tuple of valid `CHOICES` for data validation. The `CHOICES` variable could be a list or a tuple. A tuple (https://adafru.it/Nbi) is used as form of defensive programming (https://adafru.it/Nbj) since tuples are immutable (read-only) in Python - this prevents accidental modification to this constant (https://adafru.it/Nbk) sequence.

The return type has changed and is now a form of enumerated type (https://adafru.it/Nbl) using const() (https://adafru.it/Nbm) global variables. CircuitPython's `const()` can only be used for integers but that's sufficient here. CPython offers the useful Enum (https://adafru.it/Nbn) class for enumerations but this is not currently present in CircuitPython.

The long expression in `elif` uses a mixture of `and` and `or` boolean operators. The `and` expressions are evaluated first as the programmer intended due to Python's precedence (https://adafru.it/Nbo) rules, placing `and` higher than `or`.

SampleJukebox class

Using a separate class for playing the sound samples tidies up the code a little but the main motivation for creating this class was to add a workaround to reduce the chance of fatal `MemoryError` exceptions, possibly related to memory fragmentation (https://adafru.it/Nbp). The `PWMAudioOut` library currently allocates memory dynamically under the covers. The `SampleJukebox` class attempts to use the library in a way where the 2048 byte buffer is immediately re-allocated just after it is de-allocated. This means the code is almost guaranteed to be able to reuse the previous 2048 contiguous section of memory.

RPSDisplay class

This is a large class containing all the code to send output to the display and/or to the NeoPixels for each of the screens in the game. Its constructor takes the `sample` object to let it play samples during some of the simple animations.

Dynamic Advertising Interval

The advertising interval seen in the code above is set just before the main loop using.

```
ad_interval = MIN_AD_INTERVAL if len(players) &lt;= 4 else len(players) * 0.007
```

This increases the interval for five or more players, 5 players will be 35 milliseconds, 6 players will be 42ms. This is an attempt to keep the collisions (https://adafru.it/Nbq) at a low level to maintain good efficiency.

`MIN_AD_INTERVAL` would be expected to be `0.02` seconds (20ms) for Bluetooth Low Energy. The actual value used in the code is `0.02001`. This is needed to work around a minor bug in adadfruit_ble library's start_advertising() (https://adafru.it/Nbr) which relates to the fundamental limited precision of floating-point. Some general background on this can be found in the Number Representation section of Clue Sensor Plotter in CircuitPython (https://adafru.it/Kbl).

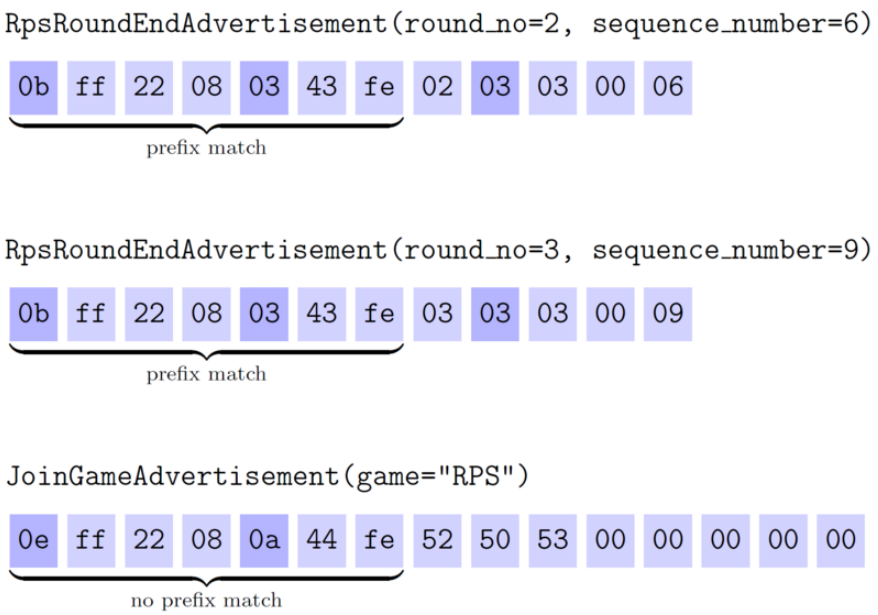
Advertisement Matching

The `adafruit_ble` library provides a feature for efficiently filtering advertising packets including any scan responses in start_scan() (https://adafru.it/Nal). If any classes are passed as arguments then only packets matching those classes will be returned. This is implemented with a simple prefix mechanism where a list of 1 or more prefix byte sequences are checked against the advertising data.

For `RpsRoundEndAdvertisement` the class sets `match_prefixes` attribute which the library code then uses to construct the prefix.

```
# match_prefixes tuple replaces deprecated prefix
    match_prefixes = (
        struct.pack(
            _PREFIX_FMT,
            MANUFACTURING_DATA_ADT,
            ADAFRUIT_COMPANY_ID,
            struct.calcsize("&lt;H" + _DATA_FMT_ROUND),
            RPS_ROUND_ID
        ),
    )
```

The `MANUFACTURING_DATA_ADT` has a value of `0xff`, the `ADAFRUIT_COMPANY_ID` is `0x0822` and the `RPS_ROUND_ID` is `0xfe43`. The prefix ends up as the bytes (in hex) `0bff22080343fe`. The `0b` is a length field and automatically prepended by the library. Some values appear "reversed", this is due to BLE values being encoded in little-endian (https://adafru.it/Ekn) order. Python's `struct` library uses `"<"` to represent this. The example below shows how this prefix will match the data in the `RpsRoundEndAdvertisement` packet regardless of the per-instance field values and correctly doesn't match a different packet.



The order of the `round_no` and `sequence_number` is critical for this prefix matching to work as the identifier number associated with `round_no` (`0xfe43`) is being used as part of the prefix to identify the class.

Current Issues

The game works well but in common with almost all large codebases there are a number of issues.

Flicker when Changing Player's Choice

There is a noticeable flicker when a player presses the left button to advance the choice between the rock, paper and scissors icons. This is a common problem with simple or naive graphics code and was left in the code on purpose to demonstrate the phenomena.

The implementation of the `showChoice()` method replaces all the screen objects and then recreates them including ones that have not been updated like the text at the top and bottom of the display. The extra display updates from these unnecessary changes are slow enough for the player to see the transition as flicker. This flicker can be reduced or eliminated by only changing the `displayio` objects which need to be updated.

In general, if lots of changes are made to `displayio` objects and if these are best displayed at once then briefly turning auto_refresh (https://adafru.it/MAr) off is an option. This will coalesce the changes into one display update.

Another occurrence of this type of flashing/flicker has been discussed in the Adafruit CircuitPython and MicroPython (https://adafru.it/ELr) forum.

Dependency on dict Key Ordering - Now Fixed

The four custom `Advertisement` sub-classes were dependent on the behaviour of CircuitPython's `dict` type for the identifier numbers to match the prefixes for each message.

A simple example on CircuitPython 5.3.0's REPL below shows the nature of this issue.

```
&gt;&gt;&gt; letters = {"a": 1, "b": 2, "c" :3}
&gt;&gt;&gt; letters
{'c': 3, 'a': 1, 'b': 2}
```

The order of the keys is not maintained in CircuitPython, `"a"` is the first key in `letters` as it is constructed but `"c"` is returned as the first one. This is a common feature for data types built upon a rudimentary hash table (https://adafru.it/Nbs). Depending on order which is not specified is a common source of pernicious, latent bugs.

CPython started returning `dict` keys in insert-order from version 3.6 (https://adafru.it/Nbt) and this formed part of the specification in version 3.7 (https://adafru.it/Nbu).

A few languages also randomly vary their hash tables to counter denial attacks, see Daniel Lemire: Use random hashing if you care about security? (https://adafru.it/Nbv)

Improvement

This was fixed with a new feature in the adafruit_ble library (https://adafru.it/Nbw) to maintain the order of fields within a `ManufacturerData` field based on the order of assignment. CircuitPython offers an `OrderedDict` data type which was used to implement this.

CPB Only Needs adafruit_display_text Library

The `RPSDisplay` class uses `Group` and `Label`. This creates the unfortunate, unnecessary dependency on having the `adafruit_display_text` library even when the Circuit Playground Bluefruit is used without a display. This could be fixed with some sub-classes that implement the relevant code for each type of output in separate files and conditional `import` statements for those sub-classes.

Sequence Number Wraparound

The sequence number used in advertising messages starts at 1 and is transmitted as an unsigned 8bit number giving it a maximum of 255. The code currently does not deal with exceeding 255 and will probably break in the 28th game.

It's common for packet identifier and sequence numbers to use a fairly small data size as the number only needs to span the maximum number of packets which are "in flight" and can wraparound back to 0. For comparison, IPv4 has a 16bit identifier (https://adafru.it/Nbx) for each packet and TCP has a 32bit sequence number (https://adafru.it/NaA) per connection.

Protocol Versioning

The messages do not have a version number. For a simple game this isn't a serious problem but for any software where there's an expectation of new features or different versions of the software in use concurrently then having a version number in the protocol/messages is useful to be able to detect or support old/new message formats, possibly concurrently.

BLE advertising packets are self-describing to some extent in the sense they have types in the fields so this is not as problematic as with some other formats.

Very Infrequent MemoryError Allocating 65536 or 42441 Bytes

A `MemoryError` exception is raised when a program runs out of memory from the heap. In CircuitPython the accompanying stack trace is printed to the serial console. There is an elusive bug somewhere that causes an allocation attempt for a relatively large amount of memory.

```
Traceback (most recent call last):
  File "code.py", line 448, in &lt;module&gt;
  File "rps_comms.py", line 340, in broadcastAndReceive
```

```
  File "rps_comms.py", line 115, in startScan
  File "adafruit_ble/__init__.py", line 263, in start_scan
MemoryError: memory allocation failed, allocating 65536 bytes
```

The program here does not do anything obvious which needs such a large amount of memory particularly around the call to `start_scan()`. The sizes are suspicious:

- both values appear far larger than any piece of data in the program;
- `42441` only has two factors, 53 * 797, suggesting it is not a simple repeated data type;
- `65536` is 2^{16}.

There's no obvious pattern so far for when this occurs although it does appear to always happen on the third call to `broadcastAndReceive()`. The frequency of occurrence is approximately 1 round in 500. It may occur more frequently when many (5+) players are playing. This is logged as issue in GitHub (https://adafru.it/Nby).

Going Further

Ideas for Areas to Explore

- Customise the game:
  - Replace the sound samples (16k, 8bit mono) with your own. Microcontroller Compatible Audio File Conversion (https://adafru.it/BvU) can help with this.
  - Replace the three sprites - the 48x16 bmp (https://adafru.it/Nbz) file can be changed without changing the code if the dimensions are preserved.
- Enhance the game:
  - Use the accelerometer to detect shaking as an alternate way to start the transmission of the player's choice.
  - Add a cumulative high score table to the Advanced game.
  - Enhance the Circuit Playground Bluefruit only version to show players joining on the NeoPixels.
  - Port the Simple game to the Circuit Playground Express using infrared for communication - this is very directional and would need careful alignment making the game finickety to play.
- Bluetooth related:
  - Update the RSSI number on the player list screen as new Bluetooth packets arrive if you are interested in this value.
  - Write a distance estimation function and add a feature for the right button to toggle between RSSI and a distance estimate.

    - Test different communication conditions to see how the RSSI varies and how the game degrades or fails. Distance, different types of obstacles and close proximity to busy Wireless Access Points are all interesting factors to experiment with.

- Try implementing another multi-player game. Note: A connection based approach is likely to be a better solution for two-player games and may also suit card games featuring a dealer.
- Investigate lots of players. The Advanced game has been tested and works well with six players. It would be interesting to see how well it works in a classroom environment with 20+ devices. The `MAX_PLAYERS` will need increasing as it's currently set to `8` but this is only because it's how many players fit on the joining screen with the chosen font size. This can be increased and they should just overspill harmlessly.

Related Projects

- CircuitPython BLE Advertising Beacons (https://adafru.it/LYC)
- Bluetooth LE Sensor Nodes to Raspberry Pi WiFi Bridge (https://adafru.it/LYD)
- Build an ML Rock Paper Scissors Game with Lobe (https://adafru.it/RiB) - object/gesture recognition with machine learning on a Raspberry Pi 4.

Further Reading

- Numberphile: Winning at Rock Paper Scissors (https://adafru.it/NbA) (YouTube) - Hannah Fry describing research and strategies against human players.
- Bluetooth SIG: Topology Options (https://adafru.it/NbB) - high level comparison of point-to-point (1:1), broadcast (1:many) and mesh (many:many with forwarding) communication.
- Introduction to Bluetooth Low Energy (https://adafru.it/DN8)
- Maxim Integrated: Bluetooth Low Energy: Understanding GAP Roles - Part 3 of 7 (https://adafru.it/NbC) (YouTube) - fairly detailed explanation of advertising in BLE GAP.
- American Scientist: Random Paths to Frequency Hopping (https://adafru.it/NbD) - the history of inventions relating to spread spectrum communication.
- Bluefruit nRF52 Feather Learning Guide (https://adafru.it/IfM) - C++/Arduino progamming for the nRF52832-based Feather nRF52 Bluefruit LE (http://adafru.it/3406)
- Bluetooth SIG: An Intro to Bluetooth Mesh part 1 (https://adafru.it/NbE)
- Yin, Yang, Cao, Liu, Zhou, Wu: A Survey on Bluetooth 5.0 and Mesh: New Milestones of IoT **(https://adafru.it/NbF)**
- Networking With The micro:bit (https://adafru.it/NbG) - an online book from Nominet using the BBC micro:bit (http://adafru.it/3530) to explain networking principles. There is also a MicroPython Edition (https://adafru.it/NbH).

- code.org: The Internet: Packets, Routing & Reliability (https://adafru.it/NbI) (YouTube) - a short, high-level introduction to TCP/IP.
- Hannah Makes: Project 3: Use two microbits to help you keep your distance (https://adafru.it/NbJ) (YouTube) - a straightforward demonstration of approximate distance estimation using the BBC micro:bit, MakeCode and radio RSSI.
- Tom Jennings' Gas Tube Random Number Generator (https://adafru.it/NbK) - using a thyratron (https://adafru.it/NbL) to produce random numbers like it's 1948.
- ERNIE (Electronic Random Number Indicator Equipment) (https://adafru.it/NbM) - the original machine ERNIE 1 resides in the Science Museum London's collection (https://adafru.it/NbN). This used a neon tube to produce random numbers (https://adafru.it/NbO) in 1957.
- General Post Office: The Importance of being E.R.N.I.E. (1964) (https://adafru.it/NbP) (YouTube) - video showing ERNIE 1 being used to choose Premium Bond winners.
- Rand Corporation: A Million Random Digits with 100,000 Normal Deviates (1955) (https://adafru.it/NbQ) - a useful book for insomniacs.
- Secura: Zerologon: Unauthenticated domain controller compromise by subverting Netlogon cryptography (CVE-2020-1472) (https://adafru.it/Nyf) - a serious flaw discovered by Tom Tervoort in the NTLM protocol (https://adafru.it/NyA) in Microsoft Windows caused by an elementary blunder setting an initialization vector (https://adafru.it/NyB).