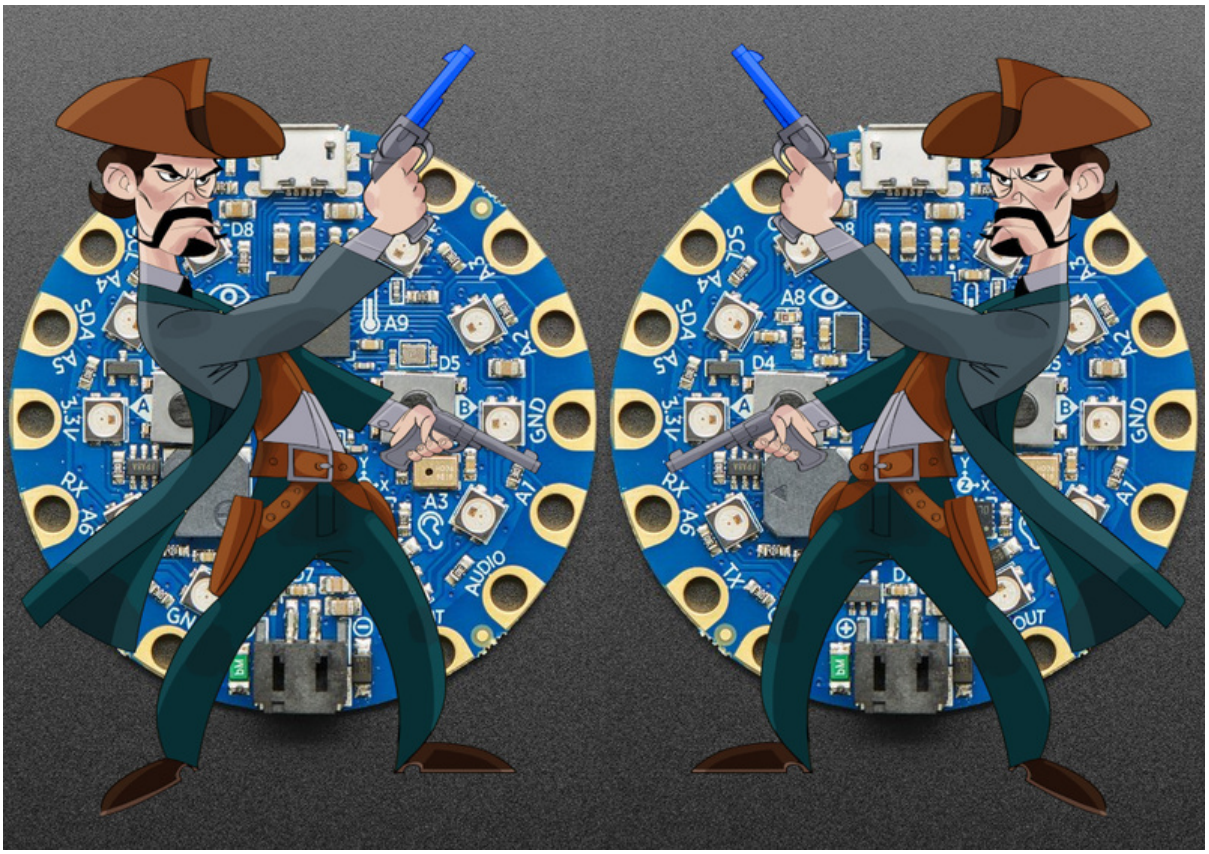
Circuit Playground Bluefruit Quick Draw Duo

Created by Kevin Walters

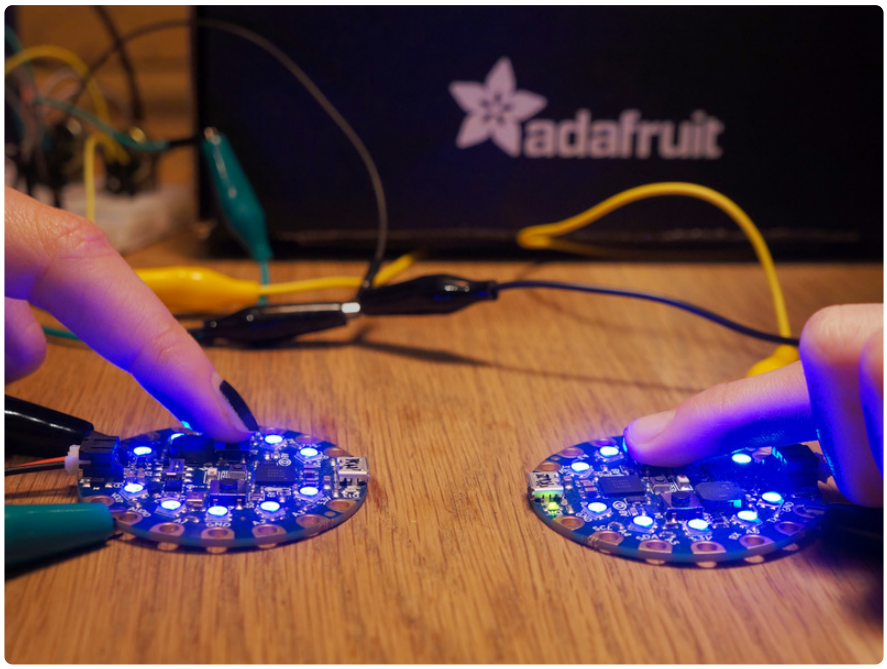


https://learn.adafruit.com/circuit-playground-bluefruit-quick-draw-duo

Last updated on 2024-06-03 02:58:02 PM EDT

Table of Contents

Overview



This project is a variant of the Circuit Playground Quick Draw (https://adafru.it/Igc) game for a pair of Circuit Playground Bluefruit (https://adafru.it/GYc) (CPB) boards using CircuitPython. Bluetooth Low Energy (LE) (https://adafru.it/Di6) is used to exchange information between the boards. This means you can play the game wirelessly!

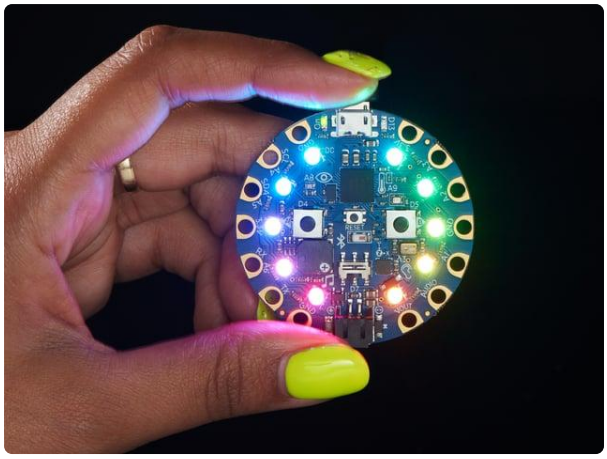
An additional CircuitPython program can be used to measure and compare visual and auditory reaction times. This can run on either a Circuit Playground Express (https://adafru.it/BeF) (CPX) or CPB board.

Thank-you to Carter Nelson (https://adafru.it/Ch4) for the original game and Megan, Izzy, Elsa and Matilda for help in testing the new game.

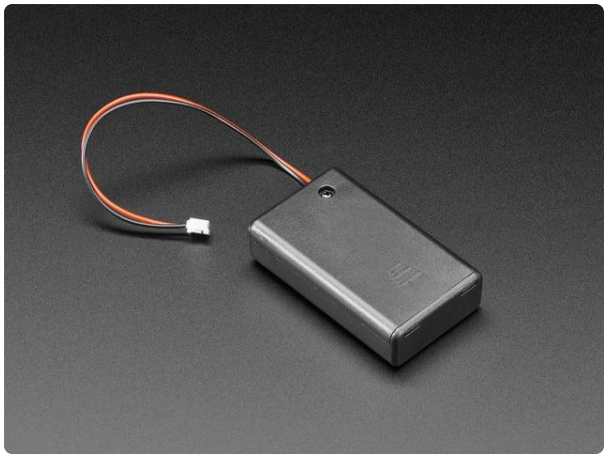
The **two** CPB boards require the latest 5.x version of CircuitPython.

Parts

You will need two each of the following parts:



[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

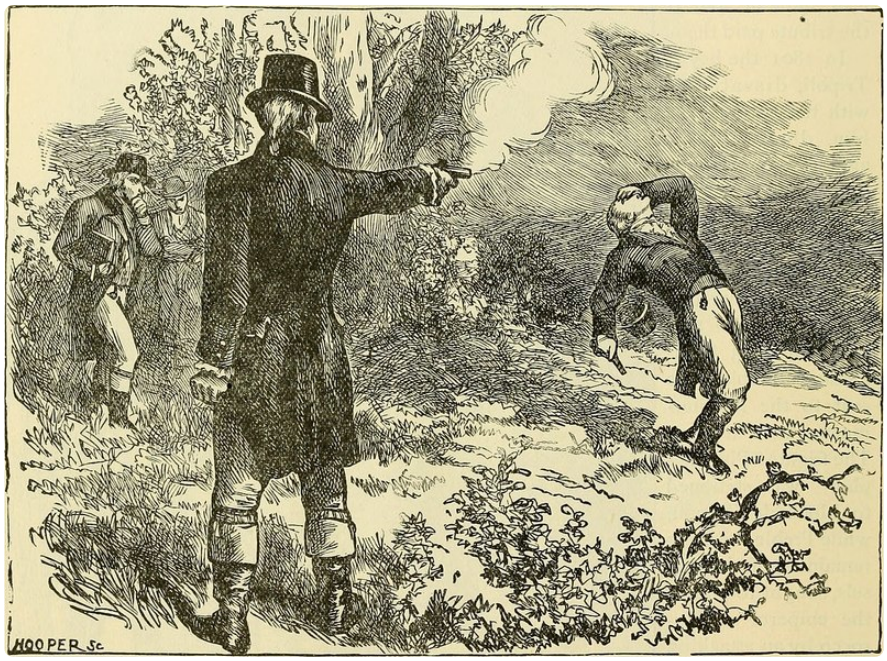


[3 x AAA Battery Holder with On/Off Switch and 2-Pin JST]
This battery holder connects 3 AAA batteries together in series for powering all kinds of projects. We spec'd these out because the box is slim, and 3 AAA's add up to about...
https://www.adafruit.com/product/727

You will need a good USB data + power cable to program things:

[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

Game Design



The design of the game is very similar to the original (https://adafru.it/lgd) with a few additions below marked with the † symbol. The additions are needed to synchronise the two CPB boards and exchange reaction times between them. There are also ten rounds to make this more fun although this doesn't strictly make sense for a duelling game!

The communication protocol between the two boards requires a different role is assigned to each board. This is set using the two-position switch (https://adafru.it/lge) on the board, i.e. the two boards must have the switch in different positions. The player's "draw" button is set to the button on the side the switch is set to.

Game Logic

1. Turn off all of the NeoPixels.
2. Determine the communication delay between the boards.
3. Synchronise the boards.†
4. Wait the determined (random) countdown time.
5. If a player presses a button during this time, they drew too soon (misdraw).
6. Once the countdown time has elapsed, turn on all of the NeoPixels (white).
7. Wait for the button presses and exchange reaction time data between boards.†
8. Look for the first (quickest) button press.
9. Whichever button was pressed first is the Quick Draw winner.
10. Repeat to step 3 until ten rounds have been completed.†
11. Display a personalised summary of wins/misdraws on the NeoPixels.†

The exchange of data at step 7 can cause a delay compared to the original single board game. The winner cannot be determined until each board has received the data from the other board.

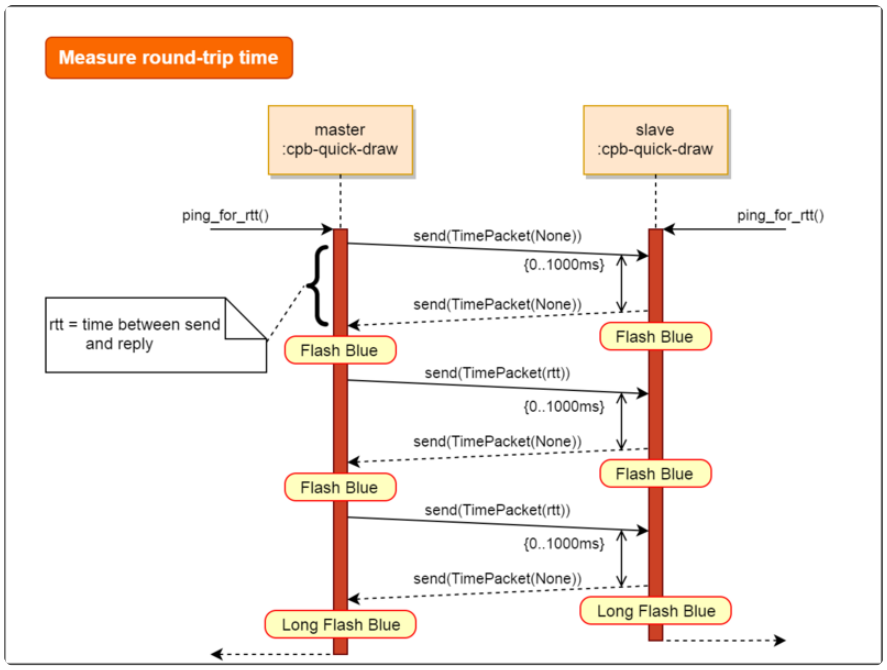
The winner at step 9 introduces a new colour (amber) to represent the unlikely outcome of both players simultaneously pressing their buttons.

Delay Calculation and Synchronisation Protocol

Sending data from one board to another takes a certain amount of time. If one board sends a "go now" command to another then the sender does not know when the receiver has received and processed that command. The receiver could send a reply but the same problem exists whereby the sender of the reply does not know when it has been received and processed. Measuring the transmission time including the time to receive and parse the data facilitates good synchronisation between the boards. This is one element of the Network Time Protocol (NTP) (https://adafru.it/lgf) which can be used to distribute accurate time from reference clocks to computers and devices across the globe.

The send time on its own cannot be measured by the sender but the time to send and receive an immediate reply can be measured by the sender. If the network propagation and processing are symmetric then exactly half of this round-trip time (rtt) (https://adafru.it/lgA) represents the time for the sender's data to propagate and be processed with the the other half representing the time for the receiver's reply to propagate and be processed.

The diagram below shows how the rtt measurement operates. Time progresses downwards in the diagram.

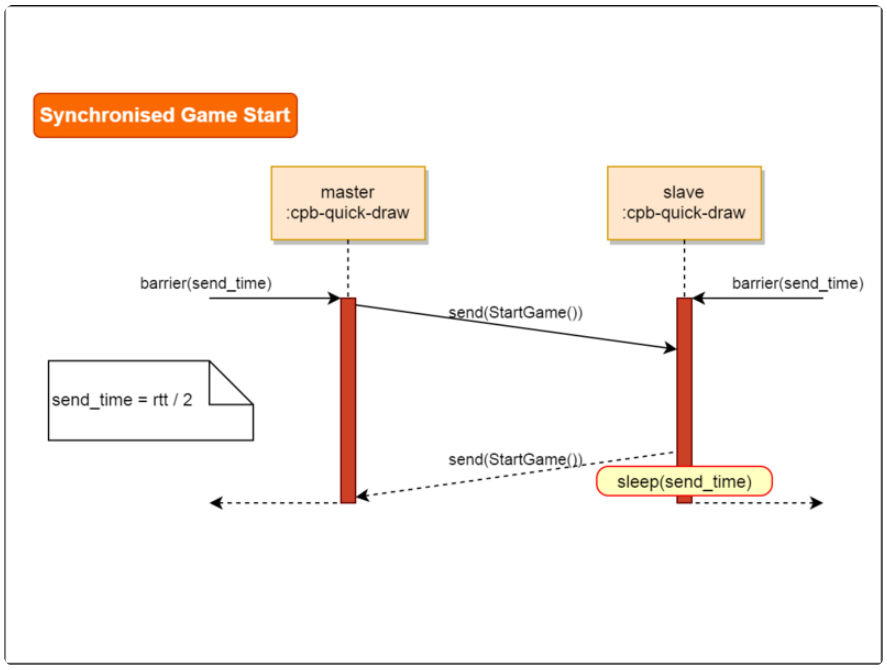


For comparison, the approach is the similar to how ping (https://adafru.it/IgB) works using the ICMP (https://adafru.it/IgC) echo service. The traditional TCP/UDP echo service (https://adafru.it/IgD) is another similar, higher-level service.

The asymmetric roles for the two boards in the measuring process are commonly referred to in computing and electronics as main and secondary (https://adafru.it/IgE). In this example only the main is measuring the rtt but it then passes the value to the secondary in subsequent packets. A simple way to set the role (or any boolean (https://adafru.it/IgF) quantity) is to use the two-position switch (https://adafru.it/Ige) on the CPB board.

In Bluetooth LE terminology (https://adafru.it/iCp), the two devices are a central device which connects to a peripheral device. This also gives the boards different roles in terms of a client/server model (https://adafru.it/Iha).

An approximate, but reasonably accurate, send time can be calculated from the measured round-trip times. This duration can then be used by one board to compensate for the send time and this allows the two boards to be synchronised, enabling the game to start on each board at the same time.



The procedure name used in the design is `barrier`, a reference to a type of synchronisation (https://adafru.it/lhb) used for multi-threaded applications. This procedure aims to complete at the same time on each CPB board causing the subsequent code on each to execute near simultaneously. In this case, the aim is to be within a few milliseconds of each other. If the player is reacting to their own white NeoPixel flash then they do not have to be perfectly synchronised.

The staccato nature of Bluetooth LE communication (see diagram in GATT Transactions (https://adafru.it/iCp)) is not accounted for in the synchronisation technique and this is one factor reducing its accuracy.

In this design, the same type of messages are used by the request and the response for simplicity.

A message (https://adafru.it/lkC) which always elicits a reply message is often referred to as a request and response. This style of interaction in this protocol could also be regarded as a remote procedure call (RPC) (https://adafru.it/lhc).

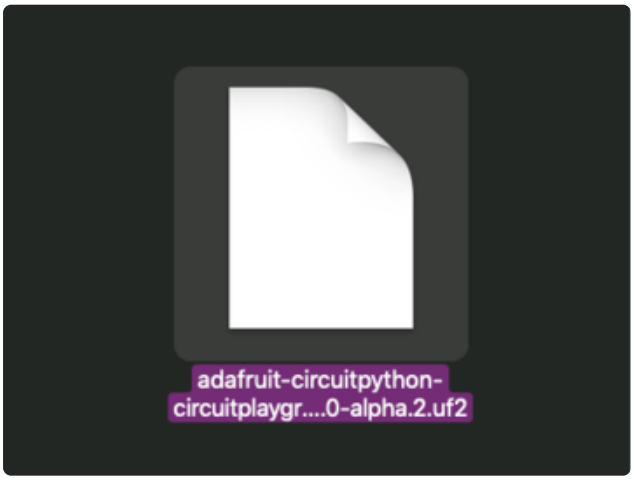
CircuitPython on Circuit Playground Bluefruit

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

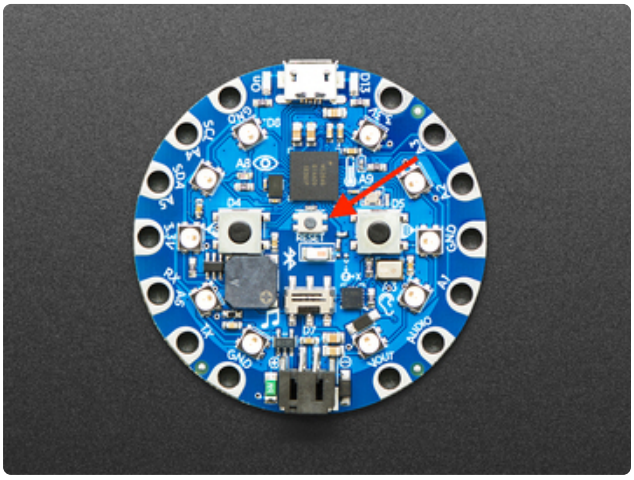


Click the link above and download the latest UF2 file

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.

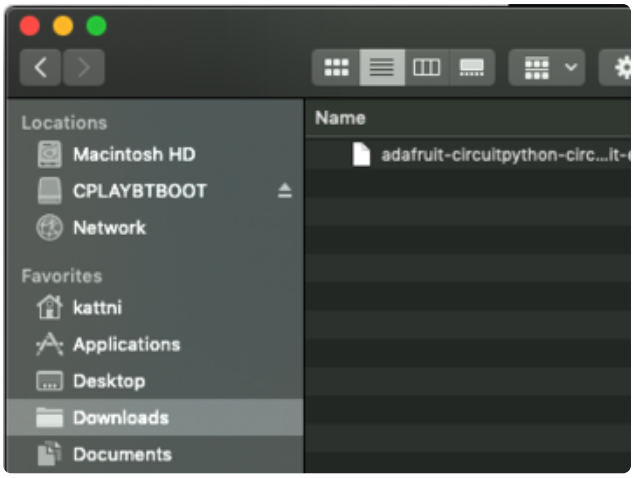
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.



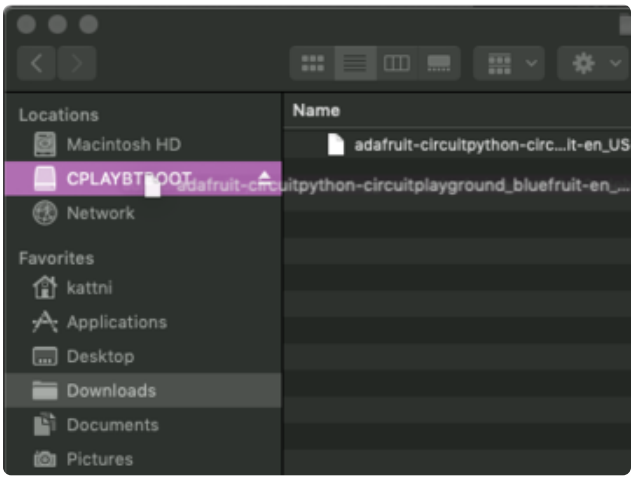
Double-click the small **Reset** button in the middle of the CPB (indicated by the red arrow in the image). The ten NeoPixel LEDs will all turn red, and then will all turn green. If they turn all red and stay red, check the USB cable, try another USB port, etc. The little red LED next to the USB connector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

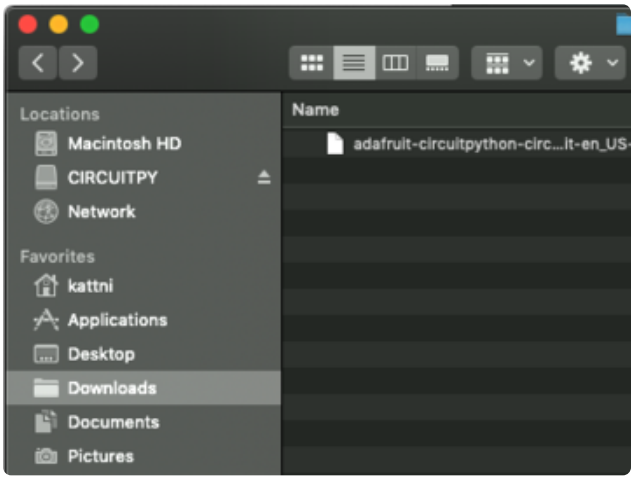
(If double-clicking doesn't do it, try a single-click!)



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 Circuit Playground Bluefruit boards, 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 Quick Draw Duo

From the library bundle you downloaded in that guide page, transfer the following libraries onto the two CPB boards' **/lib** directories:

- **adafruit_ble**
- **adafruit_bluefruit_connect**
- **adafruit_circuitplayground**
- **neopixel.mpy**

Libraries for Reaction Timer

From the library bundle you downloaded in that guide page, transfer the following libraries onto the CPB (or CPX) board's **/lib** directories:

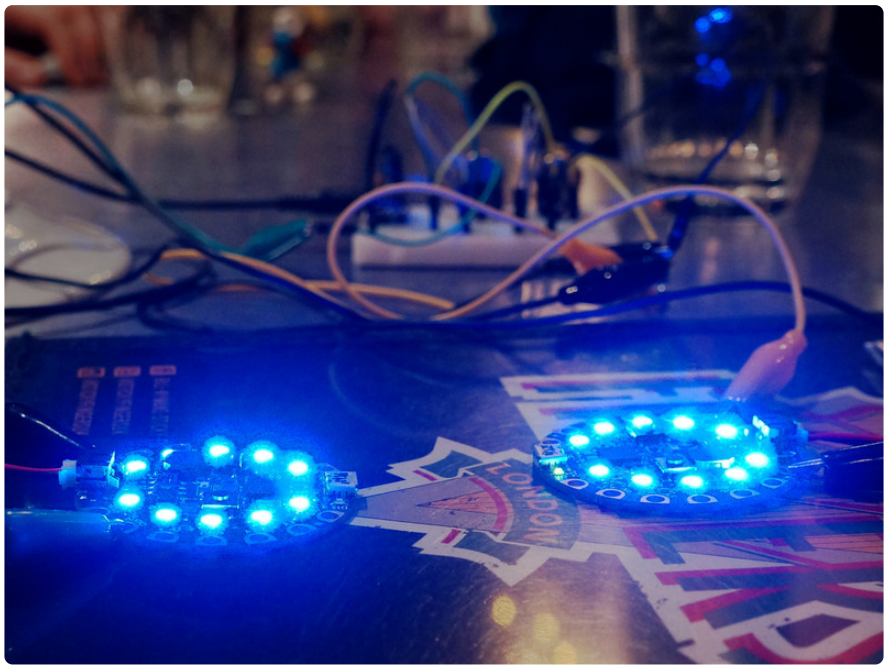
- **neopixel.mpy**

Development Testing

During development, the **Quick Draw Duo** application was tested on CPBs using CircuitPython 5.0.0-beta.2 with libraries from the adafruit-circuitpython-bundle-5.x-mpy-20200107.zip bundle. It should work on subsequent versions, the latest version is recommended (https://adafru.it/FNK).

The **Reaction Timer** code was tested on a CPX using CircuitPython 4.1.2 and on a CPB using CircuitPython 5.0.0-beta.2. The libraries were from the adafruit-circuitpython-bundle-4.x-mpy-20200107.zip and adafruit-circuitpython-bundle-5.x-mpy-20200107.zip bundles, respectively. It should work on subsequent versions, the latest version is recommended.

Quick Draw Duo

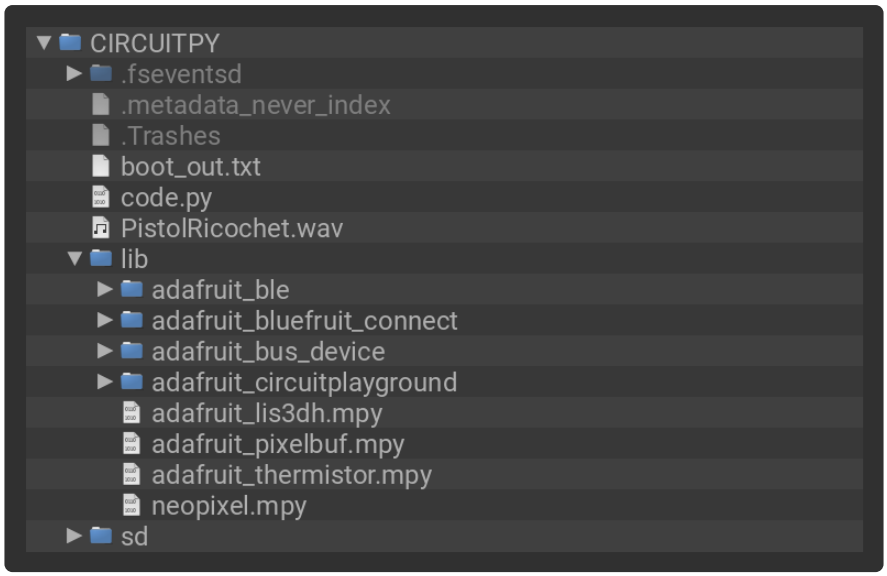


Installing Project Code

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 **CPB_Quick_Draw_Duo/quick_draw_duo/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to both of the **CIRCUITPY** drives.

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




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

# cpb-quick-draw v1.11
# CircuitPython (on CPBs) Quick Draw reaction game
# This is a two player game using two Circuit Playground Bluefruit boards
# to test the reaction time of the players in a "quick draw" with the
# synchronisation and draw times exchanged via Bluetooth Low Energy
# The switches must be set to DIFFERENT positions on the two CPBs

# Tested with Circuit Playground Bluefruit Alpha
# and CircuitPython and 5.0.0-beta.2

# Needs recent adafruit_ble and adafruit_circuitplayground.bluefruit libraries

# copy this file to 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 gc
import struct
import random  # On a CPB this seeds from a hardware RNG in the CPU

# This is the new cp object which works on CPX and CPB
from adafruit_circuitplayground import cp
```

```
from adafruit_ble import BLERadio
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.services.nordic import UARTService

from adafruit_bluefruit_connect.packet import Packet

debug = 3

# Bluetooth scanning timeout
BLESCAN_TIMEOUT = 5

TURNS = 10
# Integer number of seconds
SHORTEST_DELAY = 1
LONGEST_DELAY = 5  # This was 10 in the original game

# Misdraw time (100ms)
IMPOSSIBLE_DUR = 0.1

# The duration of the short blue flashes (in seconds) during time delay
# measurement in ping_for_rtt() and the long one at the end
SYNC_FLASH_DUR = 0.1
SYNCED_LONGFLASH_DUR = 2

# The pause between displaying each pixel in the result summary
SUMMARY_DUR = 0.5

# The number of "pings" sent by ping_for_rtt()
NUM_PINGS = 8

# Special values used to indicate failed exchange of reaction times
# and no value
ERROR_DUR = -1.0
TIME_NONE = -2.0

# A timeout value for the protocol
protocol_timeout = 14.0

# These application specific packets could be placed in an another file
# and then import'ed
class TimePacket(Packet):
    """A packet for exchanging time information,
       duration (last rtt) and time.monotonic() and lastrtt."""

    _FMT_PARSE = '<xxffx'
    PACKET_LENGTH = struct.calcsize(_FMT_PARSE)
    # _FMT_CONSTRUCT doesn't include the trailing checksum byte.
    _FMT_CONSTRUCT = '<2sff'

    # Using lower case in attempt to avoid clashing with standard packets
    _TYPE_HEADER = b'!z'

    # number of args must match _FMT_PARSE
    # for Packet.parse_private() to work, hence the sendtime parameter
    def __init__(self, duration, sendtime):
        """Construct a TimePacket."""
        self._duration = duration
        self._sendtime = sendtime  # over-written later in to_bytes()

    def to_bytes(self):
        """Return the bytes needed to send this packet.
           Unusually this also sets the sendtime to the current time indicating
           when the data was serialised.
        """
        self._sendtime = time.monotonic()  # refresh _sendtime
        partial_packet = struct.pack(self._FMT_CONSTRUCT, self._TYPE_HEADER,
                                     self._duration, self._sendtime)
        return self.add_checksum(partial_packet)
```

```
    @property
    def duration(self):
        """The last rtt value or a negative number if n/a."""
        return self._duration

    @property
    def sendtime(self):
        """The time packet was sent (when to_bytes() was last called)."""
        return self._sendtime

TimePacket.register_packet_type()


class StartGame(Packet):  # pylint: disable=too-few-public-methods
    """A packet to indicate the receiver must start the game immediately."""

    _FMT_PARSE = '<xxx'
    PACKET_LENGTH = struct.calcsize(_FMT_PARSE)
    # _FMT_CONSTRUCT doesn't include the trailing checksum byte.
    _FMT_CONSTRUCT = '<2s'

    # Using lower case in attempt to avoid clashing with standard packets
    _TYPE_HEADER = b'!y'

    def to_bytes(self):
        """Return the bytes needed to send this packet.
        """
        partial_packet = struct.pack(self._FMT_CONSTRUCT, self._TYPE_HEADER)
        return self.add_checksum(partial_packet)


StartGame.register_packet_type()

# This board's role is determine by the switch, the two CPBs must have
# the switch in different positions
# left is the master / client / central device
# right is the slave / server / peripheral device
master_device = cp.switch  # True when switch is left (near ear symbol)

# The default brightness is 1.0 - leaving at that as it
# improves performance by removing need for a second buffer in memory
# 10 is number of NeoPixels on CPX/CPB
numpixels = 10
halfnumpixels = numpixels // 2
pixels = cp.pixels

faint_red = (1, 0, 0)
red = (40, 0, 0)
green = (0, 30, 0)
blue = (0, 0, 10)
brightblue = (0, 0, 100)
yellow = (40, 20, 0)
white = (30, 30, 30)
black = (0, 0, 0)

win_colour = green
win_pixels = [win_colour] * halfnumpixels
opponent_misdraw_colour = faint_red
misdraw_colour = red
misdraw_pixels = [misdraw_colour] * halfnumpixels
draw_colour = yellow
draw_pixels = [draw_colour] * halfnumpixels
lose_colour = black

if master_device:
    # button A is on left (usb at top
    player_button = lambda: cp.button_a
    # player_button.switch_to_input(pull=digitalio.Pull.DOWN)
```

```
    player_px = (0, halfnumpixels)
    opponent_px = (halfnumpixels, numpixels)
else:
    # button B is on right
    player_button = lambda: cp.button_b
    # player_button.switch_to_input(pull=digitalio.Pull.DOWN)

    player_px = (halfnumpixels, numpixels)
    opponent_px = (0, halfnumpixels)


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 read_packet(timeout=None):
    """Read a packet with an optional locally implemented timeout.
       This is a workaround due to the timeout not being configurable."""
    if timeout is None:
        return Packet.from_stream(uart)  # Current fixed timeout is 1s

    packet = None
    read_start_t = time.monotonic()
    while packet is None and time.monotonic() - read_start_t < timeout:
        packet = Packet.from_stream(uart)
    return packet


def connect():
    """Connect two boards using the first Nordic UARTService the client
       finds over Bluetooth Low Energy.
       No timeouts, will wait forever."""
    new_conn = None
    new_uart = None
    if master_device:
        # Master code
        while new_uart is None:
            d_print("Disconnected, scanning")
            for advertisement in ble.start_scan(ProvideServicesAdvertisement,
                                                timeout=BLESCAN_TIMEOUT):
                d_print(2, advertisement.address, advertisement.rssi, "dBm")
                if UARTService not in advertisement.services:
                    continue
                d_print(1, "Connecting to", advertisement.address)
                ble.connect(advertisement)
                break
            for conns in ble.connections:
                if UARTService in conns:
                    d_print("Found UARTService")
                    new_conn = conns
                    new_uart = conns[UARTService]
                    break
            ble.stop_scan()

    else:
        # Slave code
        new_uart = UARTService()
        advertisement = ProvideServicesAdvertisement(new_uart)
        d_print("Advertising")
        ble.start_advertising(advertisement)
        # Is there a conn object somewhere here??
        while not ble.connected:
            pass
```

```
    return (new_conn, new_uart)


def ping_for_rtt():  # pylint: disable=too-many-branches,too-many-statements
    """Calculate the send time for Bluetooth Low Energy based from
       a series of round-trip time measurements and assuming that
       half of that is the send time.
       This code must be run at approximately the same time
       on each device as the timeout per packet is one second."""
    # The rtt is sent to server but for first packet client
    # sent there's no value to send, -1.0 is specal first packet value
    rtt = TIME_NONE
    rtts = []
    offsets = []

    if master_device:
        # Master code
        while True:
            gc.collect()  # an opportune moment
            request = TimePacket(rtt, TIME_NONE)
            d_print(2, "TimePacket TX")
            uart.write(request.to_bytes())
            response = Packet.from_stream(uart)
            t2 = time.monotonic()
            if isinstance(response, TimePacket):
                d_print(2, "TimePacket RX", response.sendtime)
                rtt = t2 - request.sendtime
                rtts.append(rtt)
                time_remote_cpb = response.sendtime + rtt / 2.0
                offset = time_remote_cpb - t2
                offsets.append(offset)
                d_print(3,
                        "RTT plus a bit={:f},".format(rtt),
                        "remote_time={:f},".format(time_remote_cpb),
                        "offset={:f}".format(offset))
            if len(rtts) >= NUM_PINGS:
                break

            pixels.fill(blue)
            time.sleep(SYNC_FLASH_DUR)
            pixels.fill(black)
            # This second sleep is very important to ensure that the
            # server is already awaiting the next packet before client
            # sends it to avoid server instantly reading buffered packets
            time.sleep(SYNC_FLASH_DUR)

    else:
        responses = 0
        while True:
            gc.collect()  # an opportune moment
            packet = Packet.from_stream(uart)
            if isinstance(packet, TimePacket):
                d_print(2, "TimePacket RX", packet.sendtime)
                # Send response
                uart.write(TimePacket(TIME_NONE, TIME_NONE).to_bytes())
                responses += 1
                rtts.append(packet.duration)
                pixels.fill(blue)
                time.sleep(SYNC_FLASH_DUR)
                pixels.fill(black)
            elif packet is None:
                # This could be a timeout or an indication of a disconnect
                d_print(2, "None from from_stream()")
            else:
                print("Unexpected packet type", packet)
            if responses >= NUM_PINGS:
                break

    # indicate a good rtt calculate, skip first one
```

```
    # as it's not present on slave
    if debug >= 3:
        print("RTTs:", rtts)
    if master_device:
        rtt_start = 1
        rtt_end = len(rtts) - 1
    else:
        rtt_start = 2
        rtt_end = len(rtts)

    # Use quickest ones and hope any outlier times don't reoccur!
    quicker_rtts = sorted(rtts[rtt_start:rtt_end])[0:(NUM_PINGS // 2) + 1]
    mean_rtt = sum(quicker_rtts) / len(quicker_rtts)
    # Assuming symmetry between send and receive times
    # this may not be perfectly true, parsing is one factor here
    send_time = mean_rtt / 2.0

    d_print(2, "send_time=", send_time)

    # Indicate sync with a longer 2 second blue flash
    pixels.fill(brightblue)
    time.sleep(SYNCED_LONGFLASH_DUR)
    pixels.fill(black)
    return send_time


def random_pause():
    """This is the pause before the players draw.
       It only runs on the master (BLE client) as it should be followed
       by a synchronising barrier."""
    if master_device:
        time.sleep(random.randint(SHORTEST_DELAY, LONGEST_DELAY))


def barrier(packet_send_time):
    """Master send a Start message and then waits for a reply.
       Slave waits for Start message, then sends reply, then pauses
       for packet_send_time so both master and slave return from
       barrier() at the same time."""

    if master_device:
        uart.write(StartGame().to_bytes())
        d_print(2, "StartGame TX")
        packet = read_packet(timeout=protocol_timeout)
        if isinstance(packet, StartGame):
            d_print(2, "StartGame RX")
        else:
            print("Unexpected packet type", packet)

    else:
        packet = read_packet(timeout=protocol_timeout)
        if isinstance(packet, StartGame):
            d_print(2, "StartGame RX")
            uart.write(StartGame().to_bytes())
            d_print(2, "StartGame TX")
        else:
            print("Unexpected packet type", packet)

        print("Sleeping to sync up", packet_send_time)
        time.sleep(packet_send_time)


def sync_test():
    """For testing synchronisation. Warning - this is flashes a lot!"""
    for _ in range(40):
        pixels.fill(white)
        time.sleep(0.1)
        pixels.fill(black)
        time.sleep(0.1)
```

```
def get_opponent_reactiontime(player_reaction):
    """Send reaction time data to the other player and receive theirs.
       Reusing the TimePacket() for this."""
    opponent_reaction = ERROR_DUR
    if master_device:
        uart.write(TimePacket(player_reaction,
                              TIME_NONE).to_bytes())
        print("TimePacket TX")
        packet = read_packet(timeout=protocol_timeout)
        if isinstance(packet, TimePacket):
            d_print(2, "TimePacket RX")
            opponent_reaction = packet.duration
        else:
            d_print(2, "Unexpected packet type", packet)

    else:
        packet = read_packet(timeout=protocol_timeout)
        if isinstance(packet, TimePacket):
            d_print(2, "TimePacket RX")
            opponent_reaction = packet.duration
            uart.write(TimePacket(player_reaction,
                                  TIME_NONE).to_bytes())
            d_print(2, "TimePacket TX")
        else:
            print("Unexpected packet type", packet)
    return opponent_reaction


def show_winner(player_reaction, opponent_reaction):
    """Show the winner on the appropriate set of NeoPixels.
       Returns win, misdraw, draw, colour) - 3 booleans and a result colour."""
    l_win = False
    l_misdraw = False
    l_draw = False
    l_colour = lose_colour

    if player_reaction < IMPOSSIBLE_DUR or opponent_reaction < IMPOSSIBLE_DUR:
        if opponent_reaction != ERROR_DUR and opponent_reaction < IMPOSSIBLE_DUR:
            pixels[opponent_px[0]:opponent_px[1]] = misdraw_pixels
            l_colour = opponent_misdraw_colour

        # This must come after previous if to get the most appropriate colour
        if player_reaction != ERROR_DUR and player_reaction < IMPOSSIBLE_DUR:
            l_misdraw = True
            pixels[player_px[0]:player_px[1]] = misdraw_pixels
            l_colour = misdraw_colour  # overwrite any opponent_misdraw_colour

    else:
        if player_reaction < opponent_reaction:
            l_win = True
            pixels[player_px[0]:player_px[1]] = win_pixels
            l_colour = win_colour
        elif opponent_reaction < player_reaction:
            pixels[opponent_px[0]:opponent_px[1]] = win_pixels
        else:
            # Equality! Very unlikely to reach here
            l_draw = False
            pixels[player_px[0]:player_px[1]] = draw_pixels
            pixels[opponent_px[0]:opponent_px[1]] = draw_pixels
            l_colour = draw_colour

    return (l_win, l_misdraw, l_draw, l_colour)


def show_summary(result_colours):
    """Show the results on the NeoPixels."""
    # trim anything beyond 10
```

```
    for idx, p_colour in enumerate(result_colours[0:numpixels]):
        pixels[idx] = p_colour
        time.sleep(SUMMARY_DUR)

# CPB auto-seeds from hardware random number generation on the nRF52840 chip
# Note: original code for CPX uses A4-A7 analog inputs,
#       CPB cannot use A7 for analog in

wins = 0
misdraws = 0
losses = 0
draws = 0

# default timeout is 1.0 and on latest library with UARTService this
# cannot be changed
ble = BLERadio()

# Connect the two boards over Bluetooth Low Energy
# Switch on left for master / client, switch on right for slave / server
d_print("connect()")
(conn, uart) = connect()

# Calculate round-trip time (rtt) delay between the two CPB boards
# flashing blue to indicate the packets and longer 2s flash when done
ble_send_time = None
d_print("ping_for_rtt()")
ble_send_time = ping_for_rtt()

my_results = []

# play the game for a number of TURNS then show results
for _ in range(TURNS):
    # This is an attempt to force a reconnection but may not take into
    # account all disconnection scenarios
    if uart is None:
        (conn, uart) = connect()

    # This is a good time to garbage collect
    gc.collect()

    # Random pause to stop players preempting the draw
    random_pause()

    try:
        # Synchronise the two boards by exchanging a Start message
        d_print("barrier()")
        barrier(ble_send_time)

        if debug >= 4:
            sync_test()

        # Show white on all NeoPixels to indicate draw now
        # This will execute at the same time on both boards
        pixels.fill(white)

        # Wait for and time how long it takes for player to press button
        start_t = time.monotonic()
        while not player_button():
            pass
        finish_t = time.monotonic()

        # Turn-off NeoPixels
        pixels.fill(black)

        # Play the shooting sound
        # 16k mono 8bit normalised version of
        # https://freesound.org/people/Diboz/sounds/213925/
        cp.play_file("PistolRicochet.wav")
```

```
        # The CPBs are no longer synchronised due to reaction time varying
        # per player
        # Exchange draw times
        player_reaction_dur = finish_t - start_t
        opponent_reaction_dur = get_opponent_reactiontime(player_reaction_dur)

        # Show green for winner and red for any misdraws
        (win, misdraw, draw, colour) = show_winner(player_reaction_dur,
                                                   opponent_reaction_dur)
        my_results.append(colour)
        if misdraw:
            misdraw += 1
        elif draw:
            draws += 1
        elif win:
            wins += 1
        else:
            losses += 1

        # Output reaction times to serial console in Mu friendly format
        print("({:d}, {:d}, {:f}, {:f})".format(wins, misdraws,
                                                player_reaction_dur,
                                                opponent_reaction_dur))

        # Keep NeoPixel result colour for 5 seconds then turn-off and repeat
        time.sleep(5)
    except Exception as err:  # pylint: disable=broad-except
        print("Caught exception", err)
        if conn is not None:
            conn.disconnect()
            conn = None
        uart = None

    pixels.fill(black)

# show results summary on NeoPixels
show_summary(my_results)

# infinite pause to stop the code completing which would turn off NeoPixels
while True:
    pass
```

The switch on the two CPB boards must be in a different position to assign the correct role in the communication protocol.

Playing the Game

When the code starts on both boards the NeoPixels should indicate:

1. Eight brief blue flashes - measurement of round-trip time (rtt).
2. One longer blue flash - successful conclusion of rtt measurements.
3. White - start of round one, time to press the button!

The player's button is the left button, if the switch is set to the left, and the right button if the switch is set to the right (https://adafru.it/lge). The winner of each round

is indicated by a green flash, which is on the finger side for the winner. At the end of the ten rounds, the player's board will indicate their score.

- Green - win.
- No colour - opponent won.
- Red - misdraw.
- Faint red - opponent misdrew.
- Amber - a draw, same reaction time for both players, very rare!

If one board is reset during the game, then the other must also be reset to start the game again.

The maximum Bluetooth range between two CPB boards is approximately 5m (17ft). The range will decrease if obstacles are in the path.

Example Video

The video below shows both boards being reset and the code starting with the blue flashing for synchronisation. A tense ten round game follows. As per the original game, green on the finger side indicates the winner and at the end the local summary gradually appears on the NeoPixels.

Code Discussion

For this application the core game code is the same for both boards therefore it's reasonable for the implementation to be a single program which contains the communication code for both the Bluetooth LE central device and peripheral device. There are conditionals in the code based on a `master_device` variable for when the code needs to behave differently.

The first significant part of the communication code is creating the connection and then measuring the round-trip time, as per the design, to calculate the send time.

```
ble = BLERadio()

(conn, uart) = connect()

ble_send_time = ping_for_rtt()
```

For each round of the game the following code runs.

```
# Code from the main for loop with most comments removed
    gc.collect()
```

```
    random_pause()
    try:
        barrier(ble_send_time)
        pixels.fill(white)
        start_t = time.monotonic()
        while not player_button():
            pass
        finish_t = time.monotonic()

        pixels.fill(black)

        cp.play_file("PistolRicochet.wav")

        player_reaction_dur = finish_t - start_t
        opponent_reaction_dur = get_opponent_reactiontime(player_reaction_dur)

        # Show green for winner and red for any misdraws
        (win, misdraw, draw, colour) = show_winner(player_reaction_dur,
                                                   opponent_reaction_dur)
        my_results.append(colour)
        if misdraw:
            misdraw += 1
        elif draw:
            draws += 1
        elif win:
            wins += 1
        else:
            losses += 1

        # Output reaction times to serial console in Mu friendly format
        print("({:d}, {:d}, {:f}, {:f})".format(wins, misdraws,
                                                player_reaction_dur,
                                                opponent_reaction_dur))

        time.sleep(5)
    except Exception as err:  # pylint: disable=broad-except
        print("Caught exception", err)
        if conn is not None:
            conn.disconnect()
            conn = None
        uart = None

    pixels.fill(black)
```

The code is a unusual in requesting that the memory is tidied up before the player reacts to the NeoPixels - this is an attempt to prevent it from occurring during the reaction timing. Generally, scheduling a garbage collection (https://adafru.it/FI5) (GC) is best left to the interpreter or runtime system. In this specific case, the program is designed to be idle for a period of time and then needs to accurately time a small critical section of the code. These three factors are a reasonable justification for the programmer using a carefully placed `gc.collect()` (https://adafru.it/Ihf). In some languages, like Java (https://adafru.it/InA), the programmer can only request a GC, it's not guaranteed to immediately occur.

The code between the `time.monotonic()` calls is kept to a minimum to try to ensure only the reaction time is measured. The player is reacting to the prior execution of `pixels.fill(white)`. All statements take a certain amount of time to execute and in the case of the `fill` method (https://adafru.it/IkD) it's not documented precisely when the RGB LEDs change. In this case any small advantage

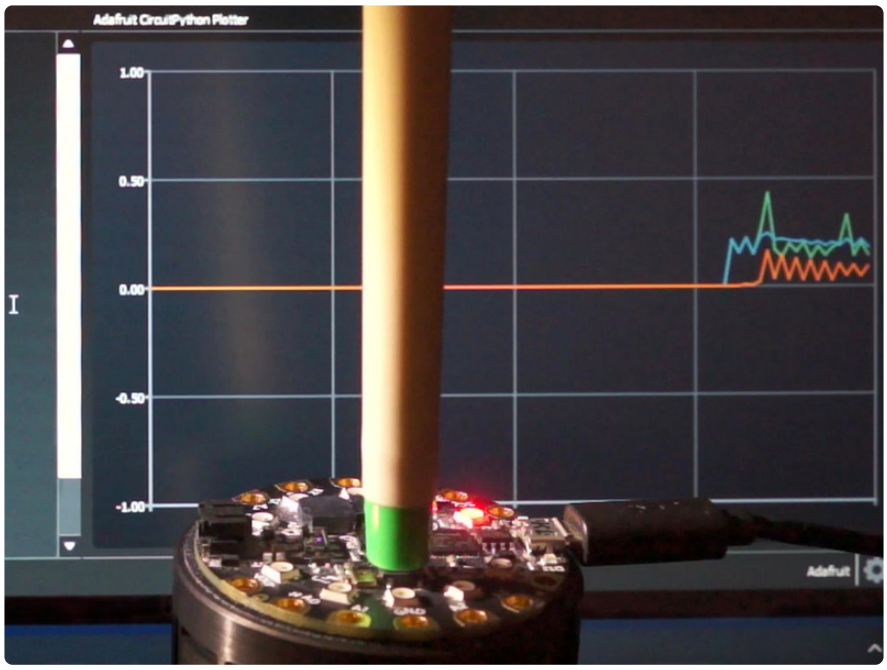
the player gets in reaction time is the same for each player so it makes no competitive difference.

The `cp.play_file("PistolRicochet.wav")` is playing a shooting sound immediately after the player has pressed the button. This statement returns when the sound has completed playing which isn't ideal here but it's the only option from the convenient `cp` object. This is a new combined object from the adafruit_circuitplayground library (https://adafru.it/C96) that can be used interchangeably on the CPB and CPX boards.

The `get_opponent_reactiontime()` function exchanges reaction times between the boards. The boards trust the reaction time is a genuine, accurate measurement. There's clearly scope here for the remote code being altered to cheat!

The boolean (https://adafru.it/IgF) variable `draw` is used to represent the unlikely outcome of both players pressing their button at the same time. This will become more likely if the boards have been powered on for several hours. Python and CircuitPython's `time.monotonic()` (https://adafru.it/Eri) returns time as a float variable. This return value increases over time reducing the resolution available to represent the fractional part of the value. The effect is far more significant for the 30bit storage representation used by CircuitPython (https://adafru.it/IhA) (based on single precision floating point (https://adafru.it/IhB)) in combination with the epoch (https://adafru.it/IhC) time of 0.0 at power-up. This lowers the precision and granularity of the millisecond portion as time passes making draws more likely.

Reaction Timer



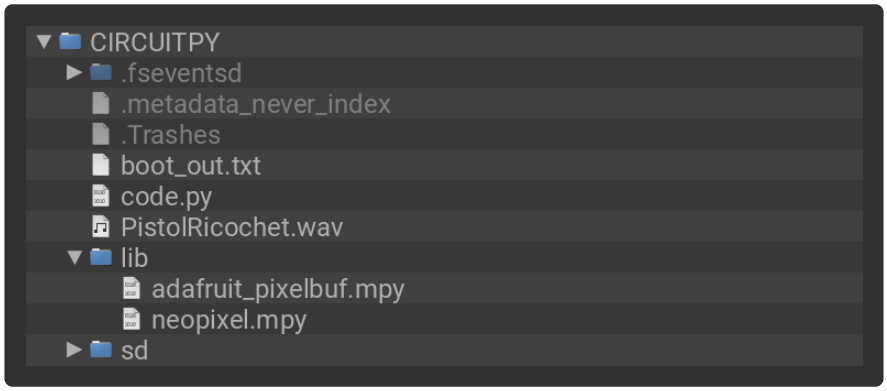
Only one board is needed for the reaction timer. A Circuit Playground Bluefruit (CPB) or a Circuit Playground Express (CPX) can be used.

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 **CPB_Quick_Draw_Duo/reaction/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to both of the **CIRCUITPY** drives.

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




```
# SPDX-FileCopyrightText: 2020 Anne Barela for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# cpx-reaction-timer v1.0.1
# A human reaction timer using light and sound

# Measures the time it takes for user to press the right button
# in response to alternate first NeoPixel and beeps from onboard speaker,
# prints times and statistics in Mu friendly format.

import os
import time
import math
import random
import array
import gc
import board
import digitalio
import analogio

# This code works on both CPB and CPX boards by bringing
# in classes with same name
try:
    from audiocore import RawSample
```

```
except ImportError:
    from audioio import RawSample
try:
    from audioio import AudioOut
except ImportError:
    from audiopwmio import PWMAudioOut as AudioOut

import neopixel

def seed_with_noise():
    """Set random seed based on four reads from analogue pads.
       Disconnected pads on CPX produce slightly noisy 12bit ADC values.
       Shuffling bits around a little to distribute that noise."""
    a2 = analogio.AnalogIn(board.A2)
    a3 = analogio.AnalogIn(board.A3)
    a4 = analogio.AnalogIn(board.A4)
    a5 = analogio.AnalogIn(board.A5)
    random_value = ((a2.value >> 4) + (a3.value << 1) +
                    (a4.value << 6) + (a5.value << 11))
    for pin in (a2, a3, a4, a5):
        pin.deinit()
    random.seed(random_value)

# Without os.urandom() the random library does not set a useful seed
try:
    os.urandom(4)
except NotImplementedError:
    seed_with_noise()

# Turn the speaker on
speaker_enable = digitalio.DigitalInOut(board.SPEAKER_ENABLE)
speaker_enable.direction = digitalio.Direction.OUTPUT
speaker_enable.value = True

audio = AudioOut(board.SPEAKER)

# Number of seconds
SHORTEST_DELAY = 3.0
LONGEST_DELAY = 7.0

red = (40, 0, 0)
black = (0, 0, 0)

A4refhz = 440
midpoint = 32768
twopi = 2 * math.pi

def sawtooth(angle):
    """A sawtooth function like math.sin(angle).
    Input of 0 returns 1.0, pi returns 0.0, 2*pi returns -1.0."""

    return 1.0 - angle % twopi / twopi * 2

# make a sawtooth wave between +/- each value in volumes
# phase shifted so it starts and ends near midpoint
vol = 32767
sample_len = 10
waveraw = array.array("H",
                      [midpoint +
                       round(vol * sawtooth((idx + 0.5) / sample_len
                                            * twopi
                                            + math.pi))
                       for idx in range(sample_len)])

beep = RawSample(waveraw, sample_rate=sample_len * A4refhz)

# play something to get things inside audio libraries initialised
audio.play(beep, loop=True)
time.sleep(0.1)
```

```
audio.stop()
audio.play(beep)

# brightness 1.0 saves memory by removing need for a second buffer
# 10 is number of NeoPixels on CPX/CPB
numpixels = 10
pixels = neopixel.NeoPixel(board.NEOPIXEL, numpixels, brightness=1.0)

# B is right (usb at top)
button_right = digitalio.DigitalInOut(board.BUTTON_B)
button_right.switch_to_input(pull=digitalio.Pull.DOWN)

def wait_finger_off_and_random_delay():
    """Ensure finger is not touching the button then execute random delay."""
    while button_right.value:
        pass
    duration = (SHORTEST_DELAY +
                random.random() * (LONGEST_DELAY - SHORTEST_DELAY))
    time.sleep(duration)


def update_stats(stats, test_type, test_num, duration):
    """Update stats dict and return data in tuple for printing."""
    stats[test_type]["values"].append(duration)
    stats[test_type]["sum"] += duration
    stats[test_type]["mean"] = stats[test_type]["sum"] / test_num

    if test_num > 1:
        # Calculate (sample) variance
        var_s = (sum([(x - stats[test_type]["mean"])**2
                      for x in stats[test_type]["values"]])
                 / (test_num - 1))
    else:
        var_s = 0.0

    stats[test_type]["sd_sample"] = var_s ** 0.5

    return ("Trial " + str(test_num), test_type, duration,
            stats[test_type]["mean"], stats[test_type]["sd_sample"])

run = 1
statistics = {"visual":    {"values": [], "sum": 0.0, "mean": 0.0,
                            "sd_sample": 0.0},
              "auditory":  {"values": [], "sum": 0.0, "mean": 0.0,
                            "sd_sample": 0.0},
              "tactile":   {"values": [], "sum": 0.0, "mean": 0.0,
                            "sd_sample": 0.0}}

print("# Trialnumber, time, mean, standarddeviation")
# serial console output is printed as tuple to allow Mu to graph it
while True:
    # Visual test using first NeoPixel
    wait_finger_off_and_random_delay()
    # do GC now to reduce likelihood of occurrence during reaction timing
    gc.collect()
    pixels[0] = red
    start_t = time.monotonic()
    while not button_right.value:
        pass
    react_t = time.monotonic()
    reaction_dur = react_t - start_t
    print(update_stats(statistics, "visual", run, reaction_dur))
    pixels[0] = black

    # Auditory test using onboard speaker and 444.4Hz beep
    wait_finger_off_and_random_delay()
    # do GC now to reduce likelihood of occurrence during reaction timing
    gc.collect()
    audio.play(beep, loop=True)
```

```
    start_t = time.monotonic()
    while not button_right.value:
        pass
    react_t = time.monotonic()
    reaction_dur = react_t - start_t
    print(update_stats(statistics, "auditory", run, reaction_dur))
    audio.stop()
    audio.play(beep)  # ensure speaker is left near midpoint

    run += 1
```

Reaction Timer with Mu Editor Video

The video below shows the reaction timer being used on a CPX board with the Mu editor (https://adafru.it/ANO) in the background. The serial console output can be seen and the numerical output is plotted. The tests alternate between **visual** and **auditory** stimulus. Since these are output on separate lines they unfortunately get plotted together by Mu.

Code Discussion

The libraries are loaded in a way that allows the same code to be used on both the CPX or CPB. The exception handling mechanism is used to load one library and if that fails, presumably due to absence, then the other will be attempted. The example below also caters for later versions of the libraries where `RawSample` has migrated to `audiocore`.

```
try:
    from audiocore import RawSample
except ImportError:
    from audioio import RawSample
```

The other conditional `import` is more interesting as it reveals the CPB's implementation of audio relies on pulse width modulation (PWM) (https://adafru.it/reK) , i.e. the audio output is digital unlike the CPX's analogue output from its DAC (https://adafru.it/EK9). The `as` syntax is useful here to rename the library's class at import time, allowing the application code to refer to a single class, `AudioOut`.

```
try:
    from audioio import AudioOut
except ImportError:
    from audiopwmio import PWMAudioOut as AudioOut
```

The CPB board uses an nRF52840 processor which includes a hardware random number generator accessible using the function `os.urandom()`. The random library uses this to seed its pseudo-random number generator (PRNG) (https://adafru.it/IhE). The CPX's ATSAMD21 processor does not have this feature which can cause the

random library to generate the same sequence each time the board is powered up. The `os.urandom()` function is always present so it must be executed to look for the exception indicating the absence of the hardware feature.

The code below uses the same seeding technique as the original Quick Draw code for the CPX. The function `seed_with_noise()` reads some analogue values from pads and uses these to seed the PRNG. These values randomly fluctuate to some degree if the pads are not connected. This will prevent a user from intentionally or unintentionally learning the sequence of pause intervals in the program.

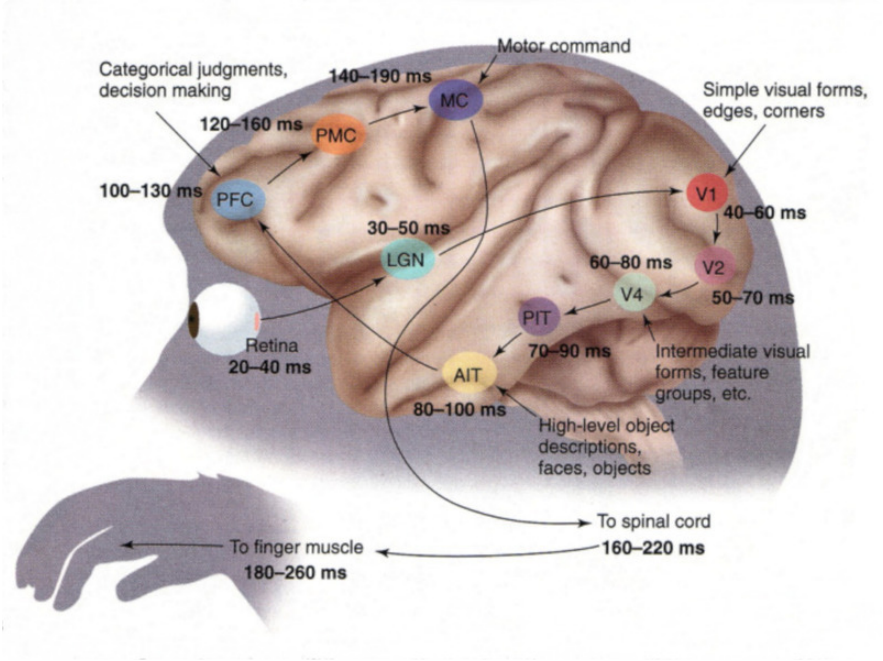
```
try:
    os.urandom(4)
except NotImplementedError:
    seed_with_noise()
```

The `update_stats()` procedure has to be cautious when calculating the standard deviation. The first step is to calculate the variance and if there is only one observation then this calculation would divide by zero. Python handles this with a `ZeroDivisionError` exception which in this case would terminate the program! A simple `if` condition avoids this disaster.

```
if test_num &gt; 1:
    var_s = (sum([(x - stats[test_type]["mean"])**2
                  for x in stats[test_type]["values"]])
             / (test_num - 1))
else:
    var_s = 0.0
```

The reaction time only turns on one NeoPixel using `pixels[0] = red`. This is an attempt to make this operation as fast as possible to make the reaction timing more accurate.

Reaction Times



The time to react can be termed a **response time** composed of:

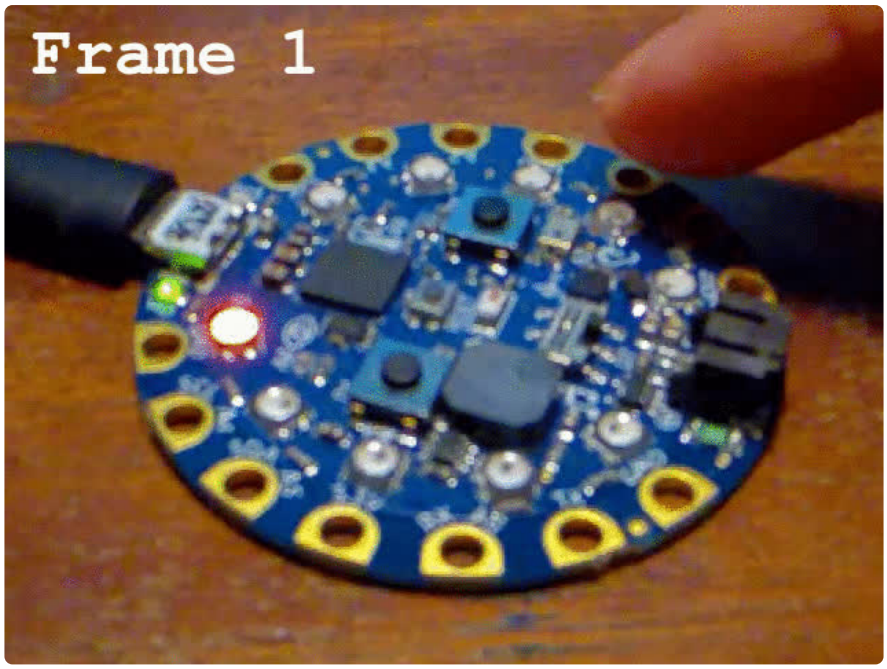
1. a **reaction** component which will vary depending on many factors including the type of stimulus, the complexity of recognition and the complexity of decision making and
2. a **movement** component.

In this guide, the term **reaction time** is used for the total. The movement time is expected to be low for pressing a touch pad or a button.

The diagram above shows some of the processing in a monkey brain with an estimation of the connectivity and **minimum-average** latencies (https://adafru.it/lhF) in milliseconds (ms). This diagram may be familiar from the Stadia Streaming Tech: A Deep Dive (Google I/O'19) (https://adafru.it/lia) which takes it and misrepresents it as human and changes the meaning and value of the times.

Movement Time

A prototype (https://adafru.it/lkF) of the Reaction Timer (https://adafru.it/lib) used a touch pad for input. A high-speed video revealed that a typical finger movement took 20-30ms. This may be partly because a cautious user needs to carefully hover about 3-4mm to avoid getting too close and accidentally triggering the capacitive input.



The input was changed to the right button to allow the user to rest their finger on the button and reduce the travel distance.

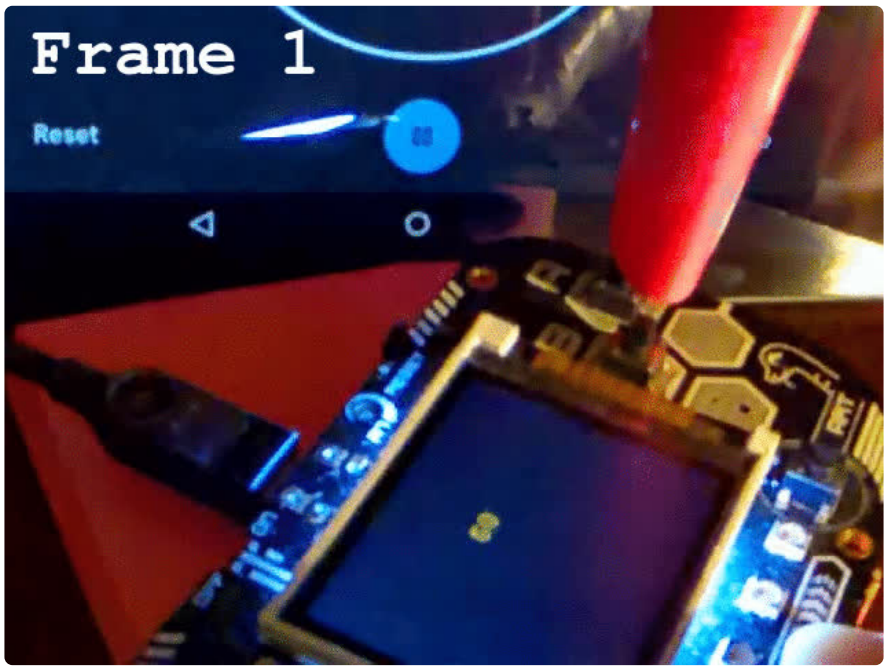
Measurement Platform

There are many reaction timing programs including ones that run in the browser. A typical, modern desktop computer is unfortunately not the best platform for measuring reaction times. Dan Luu has studied this and published a list with a great write-up on his computer latency (https://adafru.it/lic) page. His tests measure the time from a keyboard press to the character appearing on screen. The Apple IIe (https://adafru.it/lid) from 1983 is the current winner! The page explains the elements of this latency from the keyboard to the operating system (o/s) to the screen. There is also another page dedicated to keyboard latency (https://adafru.it/lie) which discusses travel time and debouncing (https://adafru.it/lif).

For basic reaction timing, this explains why the cheapest microcontroller board with an LED is superior to an expensive computer with a keyboard, general purpose operating system and an LCD screen.

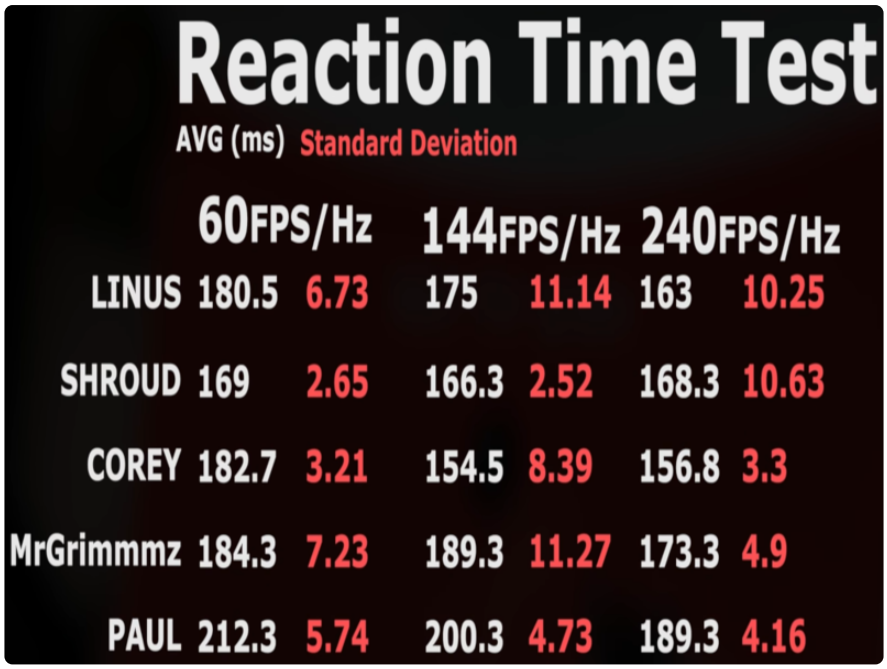
When CRT computer displays were superseded by TFT LCD panels the refresh rate (https://adafru.it/FkE) standardised from around 50-90Hz to 60Hz. This means that a modern desktop computer only displays a new image every 16.7ms and many tablets and smartphones have a similar limitation. Television models can vary in their refresh rates but often have added latency from additional image processing. This can sometimes be minimised with a "game mode" (https://adafru.it/liA) setting.

LCD displays also have a significant response time. The small PyGamer's screen can be seen below taking around 6 frames (26ms) to reach full brightness for a red letter **A**.



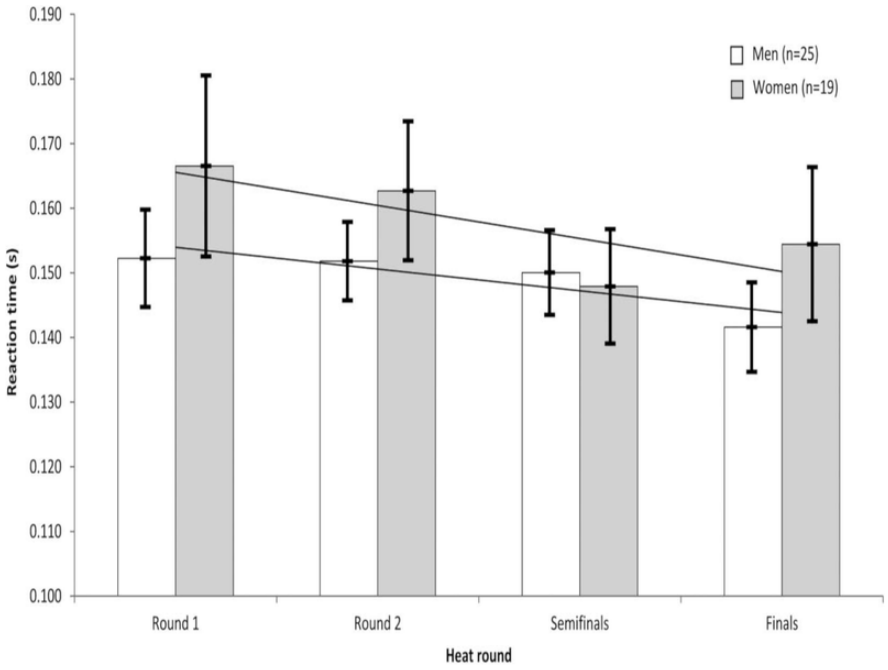
Most computer displays would be slightly better than the PyGamer's screen but there is no standard way to measure and summarise the response time. Some caution is required when comparing manufacturers' published numbers. Blur Busters have some interesting techniques to visually compare displays (https://adafru.it/liB).

Latency in all its forms is an important factor in games. particularly fast-paced, distributed, multi-player games. Linus Tech Tips conducted a set of tests on 60, 144 and 240Hz refresh rate displays (https://adafru.it/liC) to see how frame rate affects the gaming experience. This also included some simple reaction time tests using a mouse and reacting to a (visual) red/green banner. The summary of their results is shown below.



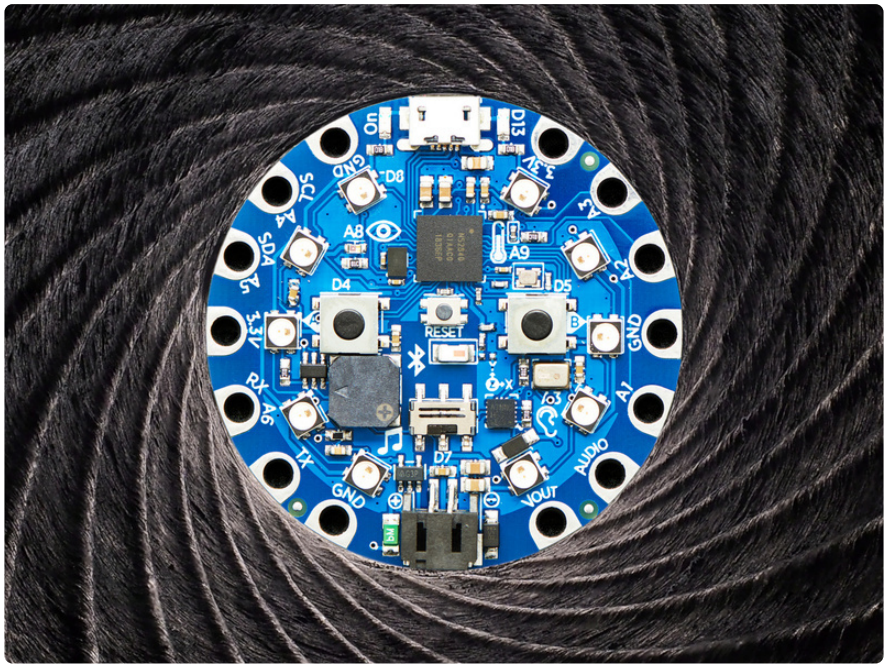
In the video there is mention of "run them until we see statistical convergence" - for a basic reaction time test, this sounds like cherry picking (https://adafru.it/liD) of the test results. This explains the low but plausible mean (**AVG**) reaction times. The standard deviation (**sd**) numbers look less plausible based on calculating **sd** from the low values actually shown in the video (**sd**=15.7ms) and from trying to reproduce low **sd** results for a visual stimulus.

On a similar theme, the reaction times for the finalist sprinters from the World Athletics Championships are shown below. These particular results are highly selected in two ways: the elite nature of the athletes; and their success in reaching the final. The graph starts at 0.1 seconds (100ms) because IAAF rules regard lower values as impossible, i.e. false starts (https://adafru.it/liE). These results cannot be directly compared to the previous visual results as athletics uses sound to start the race (from equidistant speakers) and pressure from the starting blocks.



Some results from the Reaction Timer in this guide are shown on the next page.

Reaction Timer Results

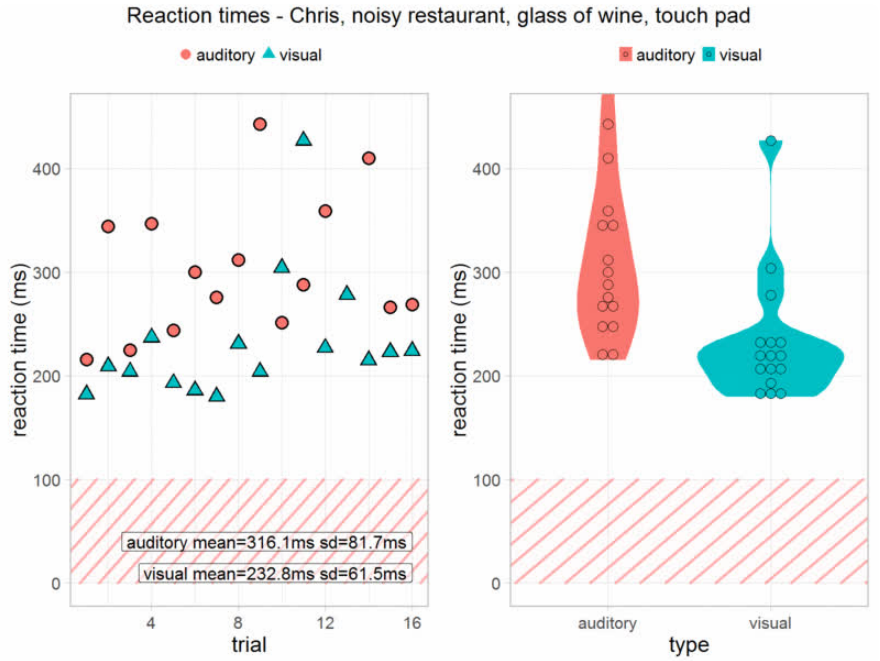


CPX Results

This is a set of graphs showing the results from various trials on a Circuit Playground Express. The graphs only show values between 0ms and 450ms. Any values less than 100ms are not used in the calculations for arithmetic **mean** and sample standard deviation (https://adafru.it/liF) (**sd**). The graph on the right shows the same data in the

form of a violin plot (https://adafru.it/lja). This shows the distribution and provides some indication if there are values above 450ms.

The test conditions vary wildy - these are described in the title on each graph. The degree of practice by the test subject varies too. The later tests were conducted with a button rather than the original use of a touch pad. These three factors make comparisons between the test subjects hard.



See results (https://adafru.it/ljb) for the individual graphs.

Conclusions

- Reaction to an auditory stimulus is faster than a visual one - this is a common and well-known result. The variance may also be less but more data is needed to prove this.
- Minimum reaction times do not vary considerably with age but very young children may struggle with consistency.
- Auditory reaction time decreases with increase in background noise even when there are no sounds which mimic the test sound.
- The auditory reaction time test is much harder if another test is being performed at the same time even if the sound's pitch is different.
- The absence of a penalty for false reactions probably decreases reaction times but this was not studied explicitly.
- Distribution does not follow the normal distribution (https://adafru.it/ljc) reducing the significance of the **mean**. It's generally asymmetric unless the subject can maintain a very high level of attention during test.

- The **movement time** appears to be less for a button compared to cautious use of a capacitive touch pad.
- Very tempting to cherry pick results and discard slow times leading to a form of sampling bias (https://adafru.it/ljd).

Going Further

Ideas for Areas to Explore

Quick Draw Duo

- Add some winning and losing sounds.
- Use the accelerometer to change the game to one with a physical draw.
- Find some other uses for a pair of synchronised CPB, e.g. light shows, stereo/spatial audio playback.
- Technical challenge 1: increase the robustness of the communication (https://adafru.it/ELr), e.g. make it recover from transient connectivity issues; match request and response with a suitable identifier.
- Technical challenge 2: understand and set the connection interval for Blutetooth LE GATT transactions and factor this into the design of the board syncronisation process.

Reaction Timer

- Add a cheat detector to the reaction timer to highlight impossible times and early presses.
- Use the NeoPixels to show the reaction time graphically.
- Increase the complexity of the visual (e.g. different colours, odd vs even) and auditory (e.g. different pitches) stimuli to investigate the additional delays from discrimination.
- Investigate tactile reaction time with some sort of actuator controlled by the CPX/CPB board.
- Examine any effect of using peripheral vision and lower brightness settings on reaction time.

Both

- Add an announcement of each time or the fastest time using sound samples for each digit.

- Independently verify the reaction timing - many smartphones offer high fps video recording in the form of a slow motion mode.

Related Projects

- Circuit Playground Quick Draw (https://adafru.it/Igc) - the original project that this one is based on.
- Color Remote with Circuit Playground Bluefruit (https://adafru.it/Ije) and Circuit Playground Bluefruit NeoPixel Animation and Color Remote Control (https://adafru.it/HE0) - two other projects showing Bluetooth communication between two CPB boards.
- De Montfort University Leicester: CTEC1802 Arduino lab work: reaction timers. (https://adafru.it/Ijf)
- BBC Bitesize: Coordination and control - The nervous system: Investigating human reaction times (https://adafru.it/IjA) - this is the remarkably simple but effective "ruler drop test".
- Reaction! (https://adafru.it/ID6) game in MakeCode for CPX.
- Reaction Time (https://adafru.it/ID7) game in MakeCode for BBC micro:bit.

Further Reading

- Introduction to Bluetooth Low Energy (https://adafru.it/iCS)
- MagicLight Bulb Color Mixer with Circuit Playground Bluefruit: Understanding BLE (https://adafru.it/IjB)
- All the Internet of Things - Epsiode One: Bluetooth & BTLE (https://adafru.it/Dik)
- Nordic Semiconductor: Wireless timer synchronization among nRF5 devices (https://adafru.it/LJd) - an interesting, very low-level approach in C for synchonising devices accurately.
- Timothy Jordan: Characteristics of Visual and Proprioceptive Response Times in the Learning of a Motor Skill. 1972. - discussed on ResearchGate: Questions: Do you react faster to a haptic stimulus than to a visual one? (https://adafru.it/IjC)
- Simon Thorpe, Michele Fabre-Thorpe: Seeking Categories in the Brain. 2001. (https://adafru.it/IjD)
- Espen Tonnessen, Thomas Haugen, Shaher A.I. Shalfawi: Reaction Time Aspects of Elite Sprinters In Athletics World Championships. 2013. (https://adafru.it/IjE)