Cheekmate - a Wireless Haptic Communication System

Created by lady ada

https://learn.adafruit.com/cheekmate-wireless-haptic-communication

Last updated on 2023-08-29 04:51:24 PM EDT
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>3</td>
</tr>
<tr>
<td>• Parts</td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td>5</td>
</tr>
<tr>
<td>• Circuit</td>
<td></td>
</tr>
<tr>
<td>Adafruit IO Setup</td>
<td>8</td>
</tr>
<tr>
<td>• Create a New Feed</td>
<td></td>
</tr>
<tr>
<td>• Create a New Dashboard</td>
<td></td>
</tr>
<tr>
<td>• Add a Text Field</td>
<td></td>
</tr>
<tr>
<td>• Adafruit IO User Name and Key</td>
<td></td>
</tr>
<tr>
<td>CircuitPython Code</td>
<td>14</td>
</tr>
<tr>
<td>• secrets.py</td>
<td></td>
</tr>
<tr>
<td>Arduino Code</td>
<td>19</td>
</tr>
<tr>
<td>Testing and Analysis</td>
<td>22</td>
</tr>
<tr>
<td>• Meat and Greet</td>
<td></td>
</tr>
<tr>
<td>• In Action</td>
<td></td>
</tr>
<tr>
<td>• Analysis</td>
<td></td>
</tr>
</tbody>
</table>
Overview

Social media is abuzz lately over the prospect of cheating in tournament strategy games. Is it happening? How is that possible with officials watching? Could there be a hidden receiver somewhere? What can be done to rectify this? These are probing questions!

We'll get to the bottom of this by making a simple one-way hidden communicator using Adafruit parts and the Adafruit IO service. Not for actual cheating of course, that would be asinine...in brief, a stain on the sport...but to record for posterity whether this sort of backdoor intrusion is even plausible or just an internet myth.

CONTENT WARNING: last page of this guide shows raw meat.

Parts

The project requires a soldering iron and related paraphernalia, and the following Adafruit items:

Adafruit QT Py ESP32-S2 WiFi Dev Board with STEMMA QT
What has your favorite Espressif WiFi microcontroller, comes with our favorite connector - the STEMMA QT, a chainable I2C port, and has...
https://www.adafruit.com/product/5325
Adafruit DRV2605L Haptic Motor Controller - STEMMA QT / Qwiic
The DRV2605 from TI is a fancy little motor driver. Rather than controlling a stepper motor or DC motor, its designed specifically for controlling haptic motors -...
https://www.adafruit.com/product/2305

Vibrating Mini Motor Disc
*BZZZZZZZZZZZ* Feel that? That's your little buzzing motor, and for any haptic feedback project you'll want to pick up a few of them. These vibe motors are tiny discs,...
https://www.adafruit.com/product/1201

Lithium Ion Polymer Battery Ideal For Feathers - 3.7V 400mAh
Lithium-ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light, and powerful. The output ranges from 4.2V when completely charged to 3.7V. This...
https://www.adafruit.com/product/3898

Adafruit Lilon or LiPoly Charger BFF Add-On for QT Py
Is your QT Py all alone, lacking a friend to travel the wide world with? When you were a kid you may have learned...
https://www.adafruit.com/product/5397
STEMMA QT / Qwiic JST SH 4-pin Cable - 100mm Long
This 4-wire cable is a little over 100mm / 4" long and fitted with JST-SH female 4-pin connectors on both ends. Compared with the chunkier JST-PH these are 1mm pitch instead of...
https://www.adafruit.com/product/4210

Hardware

For expediency, we’ll make an assumption that only one-way communication is needed. In tournament games like chess, the current state of the board is projected for all to see. An observer accomplice in the spectator gallery (or off-site if streamed) could do the work of feeding game state to an AI engine, then relaying moves to the player. Technically there’s nothing preventing input and two-way communication for solo use, but this muddies the waters for testing the core idea.

An Adafruit QT Py ESP32-S2 provides the brains. Inexpensive, incredibly tiny, and has built-in WiFi. This can communicate with a mobile hotspot (e.g. cell phone with “WiFi tethering” feature) carried by the accomplice theorized above.
How to communicate to the player? A graphical display is right out, as are visible LEDs and audible speakers. It must be silent, but deadly to one’s opponent. So we’ll use the same sort of tiny vibration motor that’s in your mobile phone. A small driver board accompanies this, as the motor requires more current than can be driven directly from a microcontroller pin.

Such a receiver needs to be discreet...watches or jewelry are too conspicuous (and might not be allowed by tournament rules). It must be concealable, perhaps inside a shoe or under one’s armpit. These body parts are naturally prone to sweat, suggesting some kind of moisture-proof enclosure.

Social Media Internet Cops keep DEMANDING that we warn people this doesn't have a flared base. We don't know what they are imagining people are going to do with this project???

These soda bottle preforms were left over from a prior project (). They’re waterproof and practically indestructible...they’ve taken a pounding and we’ve never wrecked 'em. The smooth shape glides easily into...a back pocket. Similar capsules can be found on Amazon, eBay, etc.
Circuit

Here’s a schematic view of the parts laid out for clarity. In physical reality, the microcontroller and battery charger boards are soldered back-to-back with headers to all pins. The motor controller has identical connectors on either end…it doesn’t matter which way you stick it in.

And the actual physical circuit. Battery wires are doubled back to fit all parts down the tube:
The interior of the tube is tapered slightly, and it was necessary to sand about 1/8” width from the motor driver to make it fit down in the narrow end. Best done on the edge with the motor connections, as the other edge sits close to a PCB trace.

The vibration motor is taped to the haptic controller board, and some craft foam is inserted alongside to keep these firmly pressed against the tube body to better conduct the vibration.

A 100 mm STEMMA cable gives enough slack that the motor and controller can stay put while other parts are removable to access the power switch or for charging and uploading code.

Once capped, the whole circuit is well protected from the elements!

If expanding on this project to add outside sensor or tactile inputs, one could incorporate a cable gland () to maintain a tight seal.

Adafruit IO Setup

We’ll use Adafruit IO as a backend, its simplicity is a huge asset to this project. If you’ve not used the service, head to the Welcome to Adafruit IO () guide for an explainer and to set up an account. The basic service is free and private!

So let's assume at this point you have an account set up and are at the io.adafruit.com () home page...

Create a New Feed

Feeds provide the conduit for getting data to devices like our receiver unit.
From the navigation bar second to top, select “Feeds,” and then “New Feed.”

Give the feed a useful name (e.g. “Cheekmate” to match this project) and click the “Create” button. You'll now see it in a list of feeds (or as the sole feed, if first time using the service).

Note the “Key” name assigned to the feed; typically a lowercase version of the feed name you entered. This key is needed later when setting up the code...or return to the Feeds form later to get it when needed.

Create a New Dashboard

A dashboard provides a user interface for entering data into the above feed.
Click “Dashboards” from the navigation bar, and then “New Dashboard,” assign it a name (this can be the same as the feed if you want), and “Create.”

The dashboard now appears in a list (or as the sole dashboard to start). Click the dashboard name in the list and we’ll create a simple form for entering messages...
Add a Text Field

Our new “Cheekmate” dashboard is initially blank. Near the top right of the form, click the gear icon to pop open the Dashboard Settings menu. Select the “Create New Block” item to add a UI element...
Choose the simple Text block — it provides a single-line field for entering text, that’s all we need here.

You’ll be asked to connect this to a feed (a destination to which any text entered in the field will be sent). Select the “Cheekmate” feed created earlier (or whatever name you chose), and then the “Next step” button.

Now you can customize the look a little, like selecting the Large font so it’s easy to use the dashboard from a mobile phone. Click “Create block” when it’s all to your liking.
Optional but recommended: from the Dashboard Settings menu, select “Edit Layout” to adjust the size or position of the text field so it's easier to tap. Click “Save Layout” when done.

Adafruit IO User Name and Key

This information is needed later when setting up the code.
Click the Key icon near the top right of the main Adafruit IO page to access your Adafruit IO key.

This is a seemingly random long sequence of letters and numbers that uniquely identifies you to the system, and will be inserted into the project code to grant it access.

Never share this key. If you post project code on Github or similar, remember to strip it out before committing.

---

**CircuitPython Code**

Code for this project is available both for CircuitPython and for Arduino; you can use one or the other, whichever is more your programming style. Arduino is on the next page, CircuitPython is below.

If you’ve not used CircuitPython before, begin with the [Welcome to CircuitPython](#) guide which will walk you through downloading and installation.
Click the “Download Project Bundle” button below to get all the library files packed in along with the project’s main code.py file. You will still need to create a secrets.py file with WiFi and Adafruit IO credentials, explained later on this page.

Otherwise, if you want to assemble things manually, the project requires the following CircuitPython libraries, which can be found in the library bundle () matching the version of CircuitPython you’re using:

- adafruit_drv2605.mpy
- adafruit_io
- adafruit_minimqtt
- adafruit_requests.mpy
- neopixel.mpy

These go inside the lib folder on the CIRCUITPY drive. ".mpy" items are individual files, others require the full folder.

```python
# SPDX-FileCopyrightText: Adafruit Industries
# # SPDX-License-Identifier: MIT

""
CHEEKMATE: secret message receiver using WiFi, Adafruit IO and a haptic buzzer. Periodically polls an Adafruit IO dashboard, converting new messages to Morse code.

secrets.py file must be present and contain WiFi & Adafruit IO credentials.
""

import gc
import time
import ssl
import adafruit_drv2605
import adafruit_requests
import board
import busio
import neopixel
import socketpool
import supervisor
import wifi
from adafruit_io.adafruit_io import IO_HTTP

try:
    from secrets import secrets
except ImportError:
    print("WiFi secrets are kept in secrets.py, please add them there!")
    raise
```
# CONFIGURABLE GLOBALS

FEED_KEY = "cheekmate"  # Adafruit IO feed name
POLL = 10  # Feed polling interval in seconds
REPS = 3  # Max number of times to repeat new message
WPM = 15  # Morse code words-per-minute
BUZZ = 255  # Haptic buzzer amplitude, 0-255
LED_BRIGHTNESS = 0.2  # NeoPixel brightness 0.0-1.0, or 0 to disable
LED_COLOR = (255, 0, 0)  # NeoPixel color (R, G, B), 0-255 ea.

# These values are derived from the 'WPM' setting above and do not require
# manual editing. The dot, dash and gap times are set according to accepted
# Morse code procedure.
DOT_LENGTH = 1.2 / WPM  # Duration of one Morse dot
DASH_LENGTH = DOT_LENGTH * 3.0  # Duration of one Morse dash
SYMBOL_GAP = DOT_LENGTH  # Duration of gap between dot or dash
CHARACTER_GAP = DOT_LENGTH * 3  # Duration of gap between characters
MEDIUM_GAP = DOT_LENGTH * 7  # Duration of gap between words

# Morse code symbol-to-mark conversion dictionary. This contains the
# standard A-Z and 0-9, and extra symbols "+" and "=" sometimes used
# in chess. If other symbols are needed for this or other games, they
# can be added to the end of the list.
MORSE = {
    "A": ".-",
    "B": "-...",
    "C": "-.-.",
    "D": "-..",
    "E": ".",
    "F": "..-.",
    "G": "--.",
    "H": "....",
    "I": "..",
    "J": ".---",
    "K": "-.-",
    "L": ".-..",
    "M": "--",
    "N": "-.",
    "O": "---",
    "P": ".-.",
    "Q": "--.-",
    "R": ".-.",
    "S": "...",
    "T": "-",
    "U": "--.",
    "V": "...",
    "W": ".".",
    "X": "-..-",
    "Y": "-.--",
    "Z": "--.",
    "0": "-----",
    "1": ".----",
    "2": "..---",
    "3": "...--",
    "4": "....-",
    "5": ".....",
    "6": "-....",
    "7": "--...",
    "8": "---..",
    "9": "----.",
    "+": ".-.-.",
    "=": "-...-",
}

# SOME FUNCTIONS

def buzz_on():
    """Turn on LED and haptic motor."""
pixels[0] = LED_COLOR
drv.mode = adafruit_drv2605.MODE_REALTIME

def buzz_off():
    """Turn off LED and haptic motor."""
    pixels[0] = 0
drv.mode = adafruit_drv2605.MODE_INTTRIG

def play(string):
    """Convert a string to Morse code, output to both the onboard LED
    and the haptic motor."""
gc.collect()
for symbol in string.upper():
    if code := MORSE.get(symbol):  # find Morse code for character
        for mark in code:
            if mark == ":
                buzz_on()
            time.sleep(DASH_LENGTH if mark == ":" else DOT_LENGTH)
            buzz_off()
            time.sleep(SYMBOL_GAP)
            time.sleep(CHARACTER_GAP - SYMBOL_GAP)
        else:
            time.sleep(MEDIUM_GAP)
# NEOPIXEL INITIALIZATION ---------------------------------------------
# This assumes there is a board.NEOPIXEL, which is true for QT Py ESP32-S2
# and some other boards, but not ALL CircuitPython boards. If adapting the
# code to another board, you might use digitalio with board.LED or similar.
pixels = neopixel.NeoPixel(
    board.NEOPIXEL, 1, brightness=LED_BRIGHTNESS, auto_write=True
)

# HAPTIC MOTOR CONTROLLER INIT ---------------------------------------------
# board.SCL1 and SDA1 are the "extra" I2C interface on the QT Py ESP32-S2's
# STEMMA connector. If adapting to a different board, you might want
# board.SCL and SDA as the sole or primary I2C interface.
i2c = busio.I2C(board.SCL1, board.SDA1)
drv = adafruit_drv2605.DRV2605(i2c)

# "Real-time playback" (RTP) is an unusual mode of the DRV2605 that's not
# handled in the library by default, but is desirable here to get accurate
# Morse code timing. This requires bypassing the library for a moment and
# writing a couple of registers directly...
while not i2c.try_lock():
    pass
i2c.writeto(0x5A, bytes([0x1D, 0xA8]))  # Amplitude will be unsigned
i2c.writeto(0x5A, bytes([0x02, BUZZ]))  # Buzz amplitude
i2c.unlock()

# WIFI CONNECT ---------------------------------------------
try:
    print("Connecting to {}", end="")
    wifi.radio.connect(secrets["ssid"], secrets["password"])
    print("OK")
    print("IP:", wifi.radio.ipv4_address)
    pool = socketpool.SocketPool(wifi.radio)
    requests = adafruit_requests.Session(pool, ssl.create_default_context())
    # WiFi uses error messages, not specific exceptions, so this is "broad":
    except Exception as error:  # pylint: disable=broad-except
        print("error:", error, 
        "\nBoard will reload in 15 seconds.")
        time.sleep(15)
supervisor.reload()
# ADAFRUIT IO INITIALIZATION -----------------------------------------------

```python
aio_username = secrets["aio_username"]
aio_key = secrets["aio_key"]
io = IO_HTTP(aio_username, aio_key, requests)
```

# SUCCESSFUL STARTUP, PROCEED INTO MAIN LOOP --------------------------------

```python
buzz_on()
time.sleep(0.75)  # Long buzz indicates everything is OK
buzz_off()
```

```python
current_message = ""  # No message on startup
rep = REPS  # Act as though message is already played out
last_time = -POLL  # Force initial Adafruit IO polling
```

```python
while True:  # Repeat forever...
    now = time.monotonic()
    if now - last_time >= POLL:  # Time to poll Adafruit IO feed?
        last_time = now  # Do it! Do it now!
        feed = io.get_feed(FEED_KEY)
        new_message = feed["last_value"]
        if new_message != current_message:  # If message has changed,
            current_message = new_message  # Save it,
            rep = 0  # and reset the repeat counter

        # Play last message up to REPS times. If a new message has come along in
        # the interim, old message may repeat less than this, and new message
        # resets the count.
        if rep < REPS:
            play(current_message)
            time.sleep(MEDIUM_GAP)
            rep += 1
```

---

**secrets.py**

If you’ve previously worked with CircuitPython WiFi projects, you might already have this file on the drive, or another CircuitPython board. If not, it’s easy enough to create anew. Using your text editor of preference, create a new file on the CIRCUITPY drive, called secrets.py

Copy and paste the following exactly as it is, as a starting point:

```python
secrets = {
    'ssid': 'wifi_network_name',
    'password': 'wifi_password',
    'aio_username': 'adafruit_io_username',
    'aio_key': 'adafruit_io_key'
}
```

This is a list of Python 'key' : 'value' pairs. Do not edit the keys (the part before the colon : on each line), just the values, being careful to keep both 'quotes' around strings and the comma at the end of each line.
Replace wifi_network_name and wifi_password with the name or “SSID” of your wireless network and the password for access. If tethering from a phone, one or both might be auto-generated...this information will be somewhere in the phone settings. Only 2.4 GHz networks are supported; 5 GHz is not compatible with ESP32.

Replace adafruit_io_username and adafruit_io_key with your name and unique key as explained on the “Adafruit IO Setup” page.

## Arduino Code

The Arduino version of the code does essentially the same thing; you can use one or the other, whichever is more your programming style.

This requires the Adafruit DRV2605 and Adafruit IO libraries. Installing these using the Arduino Library Manager is recommended, as it will take care of all prerequisites: Sketch→Include Library→Manage Libraries...

There are two files in this project. One contains the bulk of the code, the other has configurable settings such as the WiFi network name and password, plus the Adafruit IO account credentials and feed name. You'll need to edit the latter file (config.h) with all your particulars...it's all named descriptively and should be clear what goes where. Make sure the correct board type is selected before uploading.

You can either download a ZIP with both files:

[Download Arduino “Cheekmate” Code](#)

Or here they are inline for your perusal:

```cpp
// SPDX-FileCopyrightText: Adafruit Industries
// SPDX-License-Identifier: MIT

/*
CHEEKMATE: secret message receiver using WiFi, Adafruit IO and a haptic buzzer. Monitors an Adafruit IO feed, converting new messages to Morse code.

WiFi & Adafruit IO credentials are in the accompanying config.h file.
*/

#include <AdafruitIO_WiFi.h>
#include <Adafruit_NeoPixel.h>
#include <Adafruit_DRV2605.h>
#include "config.h" // SET UP WIFI AND ADAFRUIT IO CREDENTIALS HERE

AdafruitIO_WiFi io(IO_USERNAME, IO_KEY, WIFI_SSID, WIFI_PASS);
```

©Adafruit Industries
AdafruitIO.Feed *feed = io.feed(FEED_NAME, FEED_OWNER);
Adafruit_NeoPixel led(1, PIN_NEOPIXEL);
Adafruit_DRV2605 drv;
char message[51];
int rep = REPS; // Act as though message is already played out

// Runs once at startup
void setup() {
    Serial.begin(115200);
    led.begin();
    led.setBrightness(LED_BRIGHTNESS);
    led.show();

    // Wire1 is the "extra" I2C interface on the QT Py ESP32-S2's
    // STEMMA connector. If adapting to a different board, you might
    // want &Wire for the sole or primary I2C interface.
    drv.begin(&Wire1);
    drv.writeRegister8(0x1D, 0xA8); // Amplitude will be unsigned
    drv.setRealtimeValue(BUZZ);
    feed->onMessage(handleMessage); // Set up message handler for feed

    Serial.print("Connecting to Adafruit IO");
    io.connect();
    while(io.status() < AIO_CONNECTED) { // Wait for connection
        Serial.write('.');
        delay(500);
    }
    Serial.println(io.statusText());

    buzz_on();
    delay(750); // Long buzz indicates everything is OK
    buzz_off();
}

// Runs repeatedly until reset or power-off
void loop() {
    io.run(); // Must periodically call Adafruit IO event manager
    // Play last message up to REPS times. If a new message has come
    // along in the interim, old message may repeat less than this,
    // and new message resets the count.
    if (rep < REPS) {
        play(message);
        delay(MEDIUM_GAP);
        rep++;
    }
}

// Turn on LED and haptic motor
void buzz_on() {
    led.setPixelColor(0, LED_COLOR);
    led.show();
    drv.setMode(DRV2605_MODE_REALTIME);
}

// Turn off LED and haptic motor
void buzz_off() {
    led.setPixelColor(0, 0);
    led.show();
    drv.setMode(DRV2605_MODE_INTTRIG);
}

// Convert a string to Morse code, output to both the onboard LED
// and the haptic motor.
void play(char *str) {
    while(char c = toupper(*str++)) { // Upper-caseify each character of string...
        int i=0;
        // Scan Morse dictionary (in config.h) for a match
for (; i<NUM_SYMBOLS && morse[i].symbol != c; i++);
if (i < NUM_SYMBOLS) { // Found one!
    char mark;
    for (int j=0; (mark = morse[i].mark[j]); j++) {
        buzz_on();
        delay(mark == '-' ? DASH_LENGTH : DOT_LENGTH);
        buzz_off();
        delay(SYMBOL_GAP);
    }
    delay(CHARACTER_GAP - SYMBOL_GAP);
} else { // Not in dictionary, prob. a space
    delay(MEDIUM_GAP);
}

// Called when feed receives a message.
void handleMessage(AdafruitIO_Data *data) {
    // Limit incoming message to fit char buffer + NUL
    strncpy(message, data->toChar(), sizeof message - 1);
    Serial.printf("Received \"%s\n\", message);
    rep = 0; // Reset the message repeat counter
}

// SPDX-FileCopyrightText: Adafruit Industries
// SPDX-License-Identifier: MIT
#define WIFI_SSID "your_wifi_ssid"
#define WIFI_PASS "your_wifi_password"

// visit io.adafruit.com if you need to create an account,
// or if you need your Adafruit IO key.
#define IO_USERNAME "your_io_username"
#define IO_KEY "your_io_key"

#define FEED_OWNER "feed_owner_name"
#define FEED_NAME  "cheekmate"

#define REPS 3 // Max number of times to repeat new message
#define WPM 15 // Morse code words-per-minute
#define BUZZ 255 // Haptic buzzer amplitude, 0-255
#define LED_BRIGHTNESS 50 // NeoPixel brightness 1-255, or 0 to disable
#define LED_COLOR 0xFF0000 // NeoPixel color (RGB hexadecimal)

// These values are derived from the 'WPM' setting above and do not require
// manual editing. The dot, dash and gap times are set according to accepted
// Morse code procedure.
#define DOT_LENGTH 1200 / WPM // Duration of one Morse dot
#define DASH_LENGTH (DOT_LENGTH * 3) // Duration of one Morse dash
#define SYMBOL_GAP DOT_LENGTH // Duration of gap between dot or dash
#define CHARACTER_GAP (DOT_LENGTH * 3) // Duration of gap between characters
#define MEDIUM_GAP (DOT_LENGTH * 7) // Duration of gap between words

// Morse code symbol-to-mark conversion dictionary. This contains the
// standard A-Z and 0-9, and extra symbols "+" and "=" sometimes used
// in chess. If other symbols are needed for this or other games, they
// can be added to the end of the list.
const struct {
    char symbol;
    const char *mark;
} morse[] = {
    {'A', ".-"},
    {'B', "-..."},
    {'C', "-.-."},
    {'D', "-.."},
    {'E', "."},
Testing and Analysis

When powered on, the device will take perhaps 20 seconds to connect to the wireless network and authenticate with Adafruit IO. On success it will emit a single long buzz and light the onboard LED.

If you do not get this buzz: there’s an issue with the WiFi or Adafruit IO credentials, or the wiring between board and motor driver. Connecting the board to USB and watching with the Arduino serial monitor or other serial tool (e.g. tio or screen) will give some indication of where the problem lies.

So let’s say at this point you’re buzzed and working...
Return to the “Dashboards” tab of Adafruit IO and pick your Cheekmate dashboard from the list.

Type a brief message in the text field and press return or click or tab out of the field.

Within a few seconds, this should be relayed to the device, which will start to flash and buzz with a Morse code version of the message.

The message will repeat up to three times, unless a new message is received during that time, in which case the current message finishes and the new one repeats three times.

Meat and Greet

So we know the code and device work in open air, but what about in a hypothetical use case? There are two things to find here:

- Bodies are mostly water, and RF energy is greatly attenuated in water. Can signals penetrate if the device is nestled in, say, one’s armpit?
- Once surrounded by flesh, is the vibration motor sufficiently muffled to avoid detection, or does it give away the gag?

Without a willing partner to test and record findings with, it seemed most objective to use a proxy with similar characteristics...like a quantity of meat. Initial plan was to shove the device between two large hams, but it turns out ham is really expensive in the off season.
In Action

The unit was powered on, sealed, and inserted. A WiFi access point was about 30 feet away, through two walls and a couple inches of meat now. The end cap did protrude slightly, so it’s not a perfect test for WiFi penetration, but fixing this would require a bigger butt roast.

Secret messages were then entered in the project’s Adafruit IO Dashboard. Here’s what happened:

Analysis

While not a thoroughly scientific test, it does shine a light on the tenable aspects of the cheat device theory:

• The circuit, and the internet dashboard, were both incredibly simple to build and code; it does not require extensive engineering skills. The hardest parts would be a bit of soldering, and memorizing Morse code.
WiFi had no problem penetrating at this distance and through this medium. If an internet connection can be established through an accomplice, and data relayed through wireless, messages can be relayed.

However, working against it...

- The vibration motor, even when muffled through pounds of flesh, is anything but subtle. Officials or other players would be immediately aware. The vibration could be dialed down to a calmer level, but risks messages not being interpreted clearly as they’re harder to sense.

Thus, a reasonable conclusion is that such an idea is plausible, but unlikely. With refinement, a more discreet device could surely be developed...but, with the risk still present of being discovered, banned from competition, and being the butt of jokes for generations to come. One’s time is likely better spent learning and practicing game strategy.

A series of escalating measures and counter-measures come to mind, and it’s not clear there’s any real endgame to this.

Metal detectors are already in use at some events, but these are usually calibrated to ignore small nuisance items like coins or keys...a well-crafted receiver might slip through.

Blocking wireless signals would seem an obvious choice...but FCC laws prevent this. A deep-pocketed tournament might manage this by hosting events offshore, beyond Federal jurisdiction. Alternately, players might compete inside a Faraday cage, Thunderdome-style.

These measures might still circumvented by eliminating the off-site component, with self-contained game AI carried on one’s person. A Raspberry Pi Zero would be a bit of a stretch...but devices are continually getting smaller and more powerful, and soon (if not already) something could tuck into one’s navel or other cavity.