CLUE Text Telephone Transmitter

Created by John Park

https://learn.adafruit.com/clue-teletype-transmitter

Last updated on 2024-06-03 03:06:40 PM EDT

Table of Contents

Overview

The TTY machine (a.k.a., Teletype/textphone/Minicom) is a communications device similar to a teleprinter that is used to send text messages over the public switched telephone network.

The TTY was developed in the 1960s to assist deaf and hard-of-hearing users in communicating over the telephone system. It consists of a keyboard, display, and acoustic coupler for a phone handset (some also included a small continuous roll printer or interface for external printers). The TTY converts typed characters to audio signals which can be sent over the phone system to TTY machine on the receiving end, where those audio signals would be converted back to text for display.

The TTY was supplanted in the 1990s by modern services such as instant messaging on computers and texting on phones (as well as video calls and video relay services for sign language use) but the technology is interesting to study, and in fact still works on some phone exchanges.

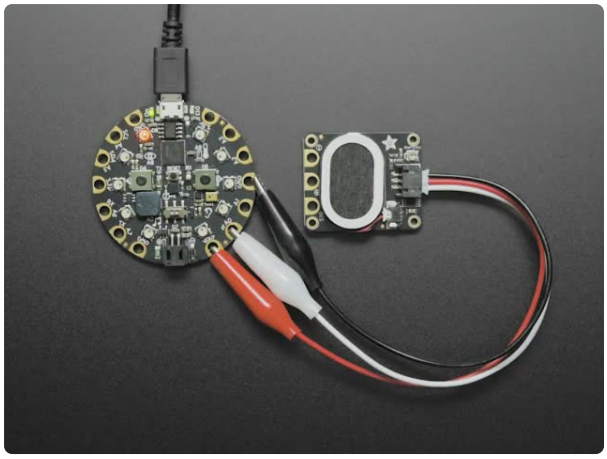
In this guide we'll take a look at how the TTY uses audio tones to communicate and build our own transmitter with a CLUE capable of sending messages to a TTY machine, both in standalone mode and in Bluetooth LE mode with messages being sent from iOS or Android.

Parts

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



Adafruit STEMMA Speaker - Plug and Play Audio Amplifier

Hey, have you heard the good news? With Adafruit STEMMA boards you can easily and safely plug sensors and devices together, like this Adafruit STEMMA Speaker - Plug and Play...

https://www.adafruit.com/product/3885

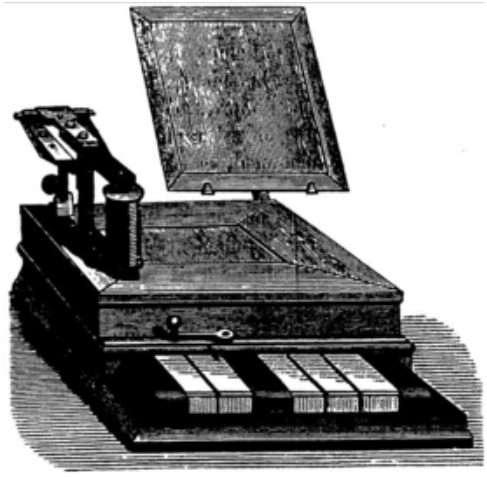
USB cable - USB A to Micro-B

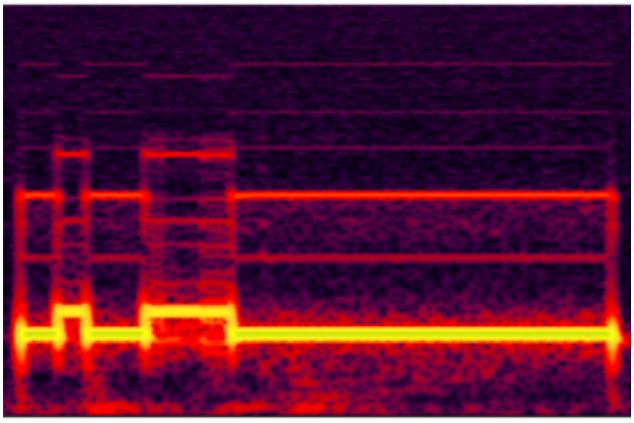
This here is your standard A to micro-B USB cable, for USB 1.1 or 2.0. Perfect for connecting a PC to your Metro, Feather, Raspberry Pi or other dev-board or...

https://www.adafruit.com/product/592

Huge thanks to Tod Kurt and Jan Goolsbey for their insights into implementing a frequency shift keyed communications protocol and to Jeff Epler and Carter Nelson for helping make the code efficient and effective.

TTY Fundamentals





5-Bit Encoding

TTY machines send and receive audio signals (usually over the phone line) which are encoded and decoded as text. They feature a small typing keyboard, display, phone handset acoustic coupling modem, and often a small printer or printer port.

TTY machines typically use a 5-bit character encoding protocol based on Baudot code (https://adafru.it/KF4) that was developed in 1870 for telegraph transmission of the Roman alphabet, numbers, and symbols.

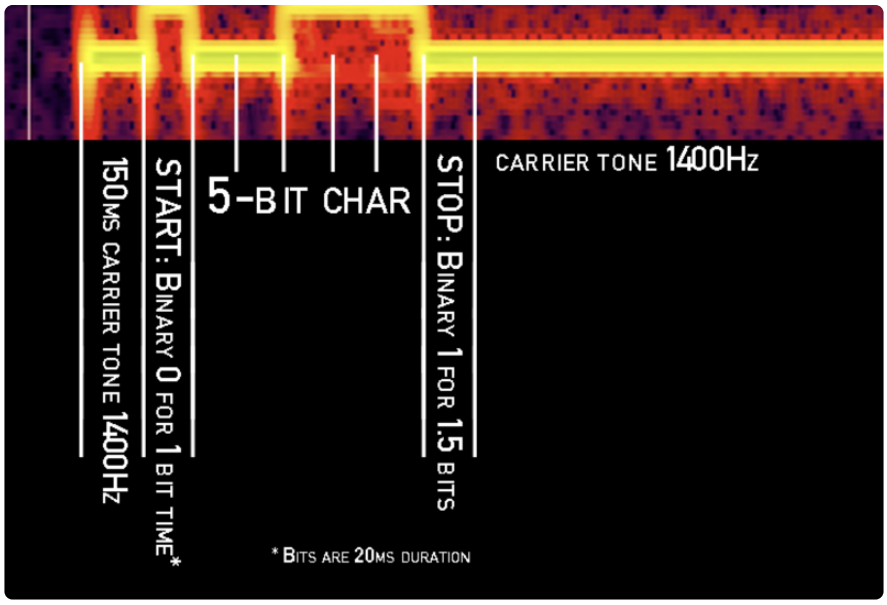
The chirping you'll hear when typing a letter on a TTY machine are 1400Hz and 1800Hz tones being shifted in 5-bit sets to represent the full character, number and symbol set. You can see the frequencies of the letter `A` visualized here -- we'll take a closer look at the specific meaning below.

The protocol is defined in this specification document (https://adafru.it/KF5). Here are some relevant selections:

> A.1 Mode of operation
> ANNEX A
> 5-bit operational mode
> The 5-bit mode is defined in ANSI TIA/EIA-825 (2000), A Frequency Shift Keyed Modem for use on the Public Switched Telephone Network.
>
> The communication channel is half-duplex with no channel turnaround. Carrier is transmitted 150 ms before the first character is transmitted. The receiver shall be disabled for 300 ms when a character is transmitted to mitigate false detection of echoes (in non-V.18 devices, the carrier may remain for up to 1 s after the last character to provide this same function).
>
> A.2 Modulation
> The modulation is frequency shift-keyed modulation (i.e. no carrier is

present when a character is not being transmitted) using 1400 Hz (±5%) for a binary 1 and 1800 Hz (±5%) for a binary 0. A bit duration of either 20 or 22.00 ± 0.40 ms is used providing either a nominal data signalling rate of 50 or 45.45 bits/s respectively.

Looking at our captured audio from the letter `A`, here's what we see:



- A bit is 20ms in duration
- The lower frequency 1400Hz tone is used both as the carrier tone and to represent a binary `1`
- The higher frequency 1800Hz tone is used to represent binary `0`
- Letter `A` (see full chart below) has a binary 5-bit encoding of `00011`
- 5-bit character codes are sent with in order of least signifigant bit (so "right-to-left"), which means the `A` bits are transmitted as `11000`
- The start bit is a binary 0 sent for 1 bit time, while the stop bit is a binary 1 sent for at least 1.5 bit time

Put all of that together and we get this:

```
carrier (for 150ms) + 0 + 11000 + 1 (for 40ms)
```

Play the file below to hear the A audio looped five times.

TTY 5-bit Baudot Encoding

Letter	Figure	5-bit code
(BACKSP)	(BACKSP)	00000
E	3	00001
LF	LF	00010
A	–	00011
SPACE	SPACE	00100
S	–	00101
I	8	00110
U	7	00111
CR	CR	01000
D	$	01001
R	4	01010
J	‘	01011
N	,	01100
F	!	01101
C	:	01110
K	(	01111
T	5	10000
Z	“	10001
L	)	10010
W	2	10011
H	=	10100
Y	6	10101
P	0	10110
Q	1	10111
O	9	11000
B	?	11001
G	+	11010
FIGS	FIGS	11011
M	.	11100
X	/	11101
V	;	11110
LTRS	LTRS	11111

One other important feature of the TTY 5-bit code is the implementation of character sets. With 5 bits we can only encode 32 characters, however there are two mode character codes reserved (sort of like a shift or control key) for switching the decoder between "letters" and "figures".

When the LTRS mode character (`11111`) is sent, all characters that follow are decoded as their alphabet letter version. When the FIGS mode character (11011) is send, all characters that follow are decoded as their numerical or symbol variant.

In practice:

- `11111` + `01101` would decode as F
- `11011` + `01101` would decode as !

The mode character should be sent right before any shift to the other mode, or every 72 characters even if you are staying in one mode.

You'll notice these mode swaps when typing on a TTY machine because some keystrokes will suddenly sound twice as long as the mode character is sent right before the letter or figure character.

Now, lets implement this in CircuitPython to send tones over a speaker and into the TTY system.

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

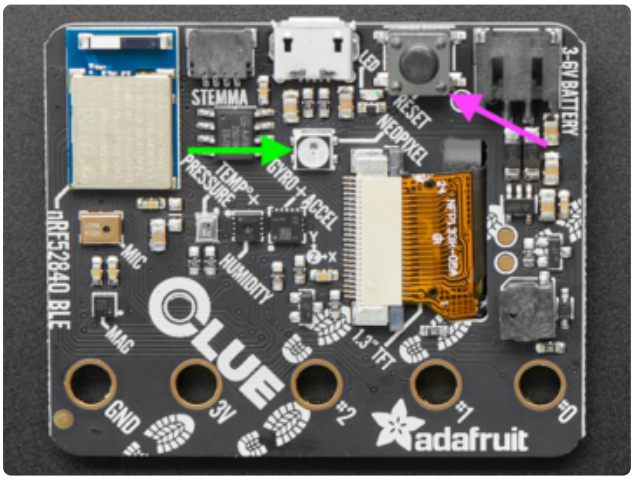
https://adafru.it/IHF

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).

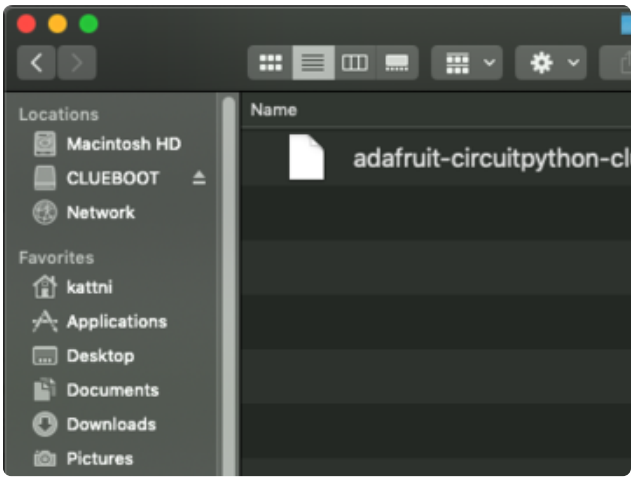
Plug your CLUE into your computer using a known-good USB cable.

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.

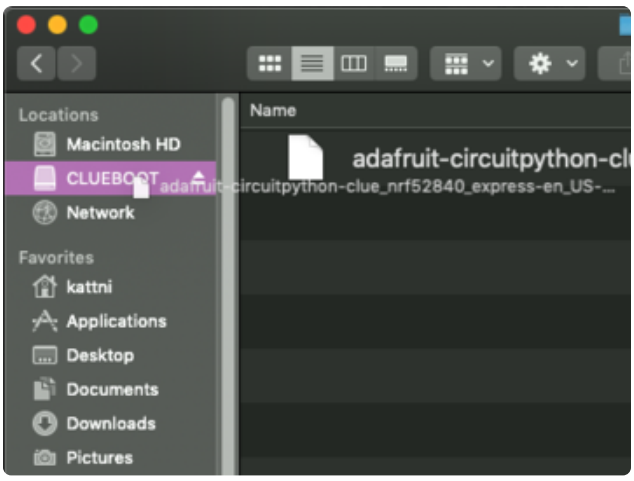


Double-click the **Reset** button on the top (magenta arrow) on your board, and you will see the NeoPixel RGB LED (green arrow) turn green. If it turns red, check the USB cable, try another USB port, etc. **Note:** The little red LED next to the USB connector will pulse red. That's ok!

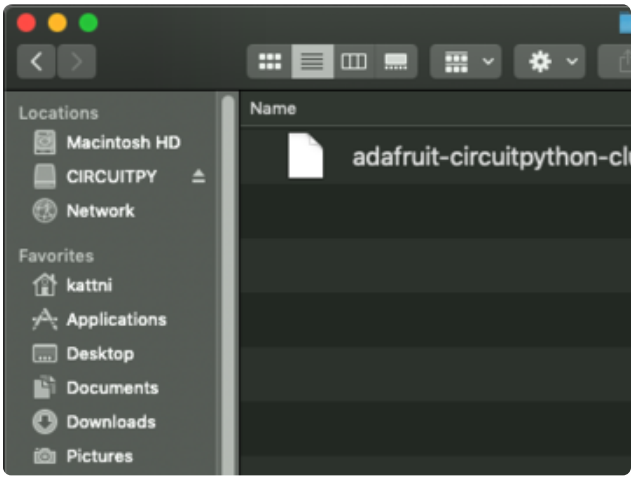
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!



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.

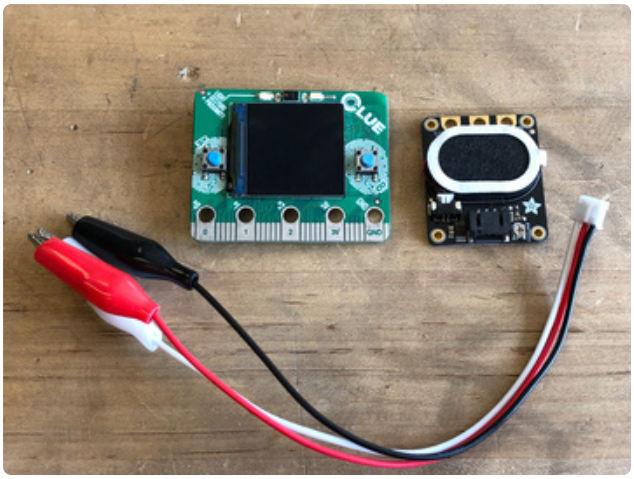
If this is the first time you're installing CircuitPython or you're doing a completely fresh install after erasing the filesystem, you will have two files - **boot_out.txt**, and **code.py**, and one folder - **lib** on your **CIRCUITPY** drive.

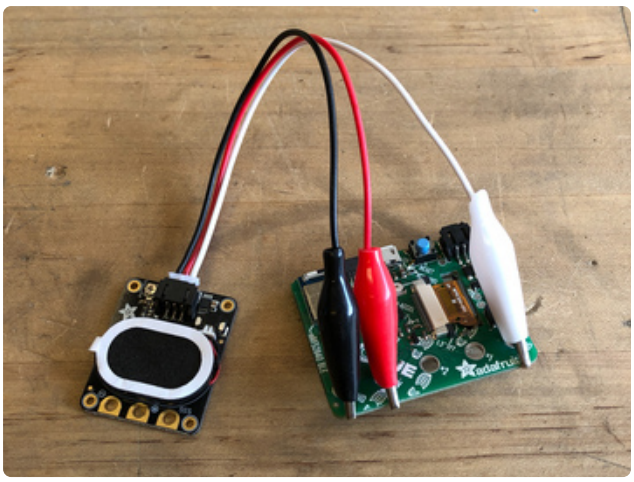
If CircuitPython was already installed, the files present before reloading CircuitPython should still be present on your **CIRCUITPY** drive. Loading CircuitPython will not create new files if there was already a CircuitPython filesystem present.

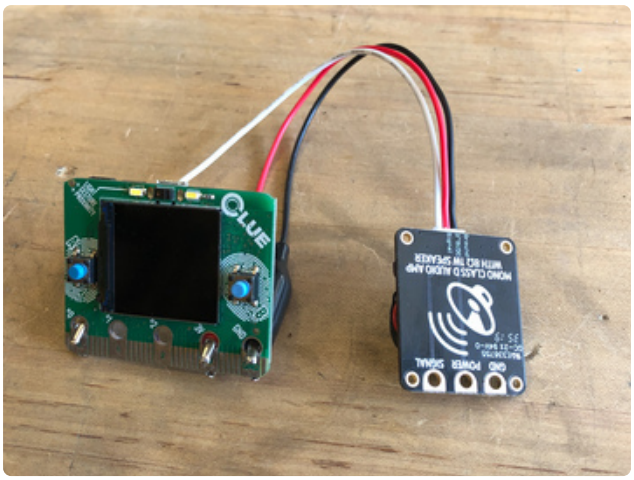
That's it, you're done! :)

Assemble the Transmitter

To get a nice, clear audio output from the CLUE, we'll add an amplifier/speaker breakout board. The STEMMA Speaker board plus JST-to-alligator clips cable makes it easy.







Hook Up

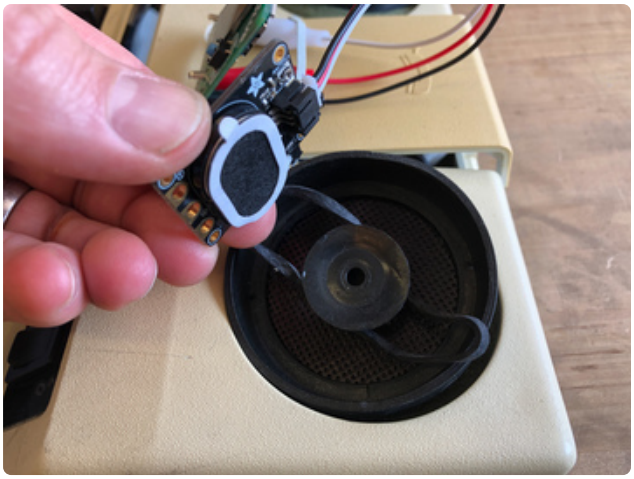
Plug the JST 3-pin cable into the STEMMA Speaker board. It is keyed to only go in one way.

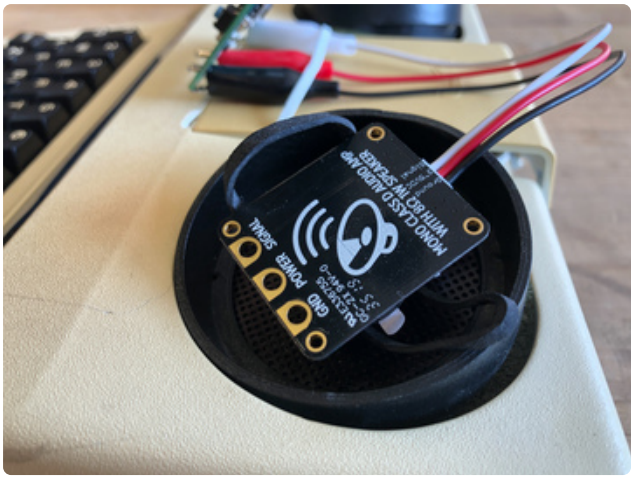
Now, connect the three alligator clips to the CLUE board from the back side. Make these connections:

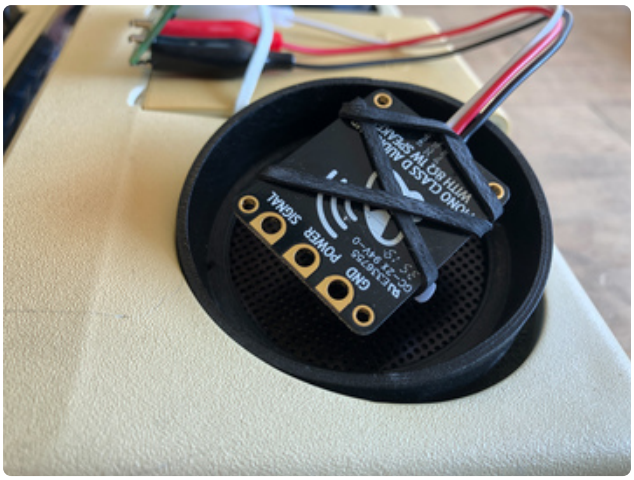
white to pin **#0**
red to pin **3V**
black to pin **GND**

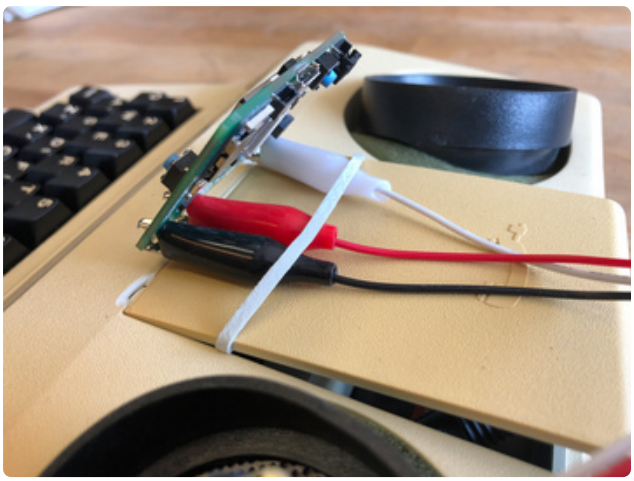
If you are sending audio TTY transmissions over a phone line to a TTY machine on the other end of the line, you'll simply hold the STEMMA speaker to the mouthpiece of your phone handset.

To test on a local machine (and see the text show up on the supremely awesome vacuum fluorescent display!) you can hold the speaker to the mic input coupler, or more permanently attach it with rubber bands as shown below.







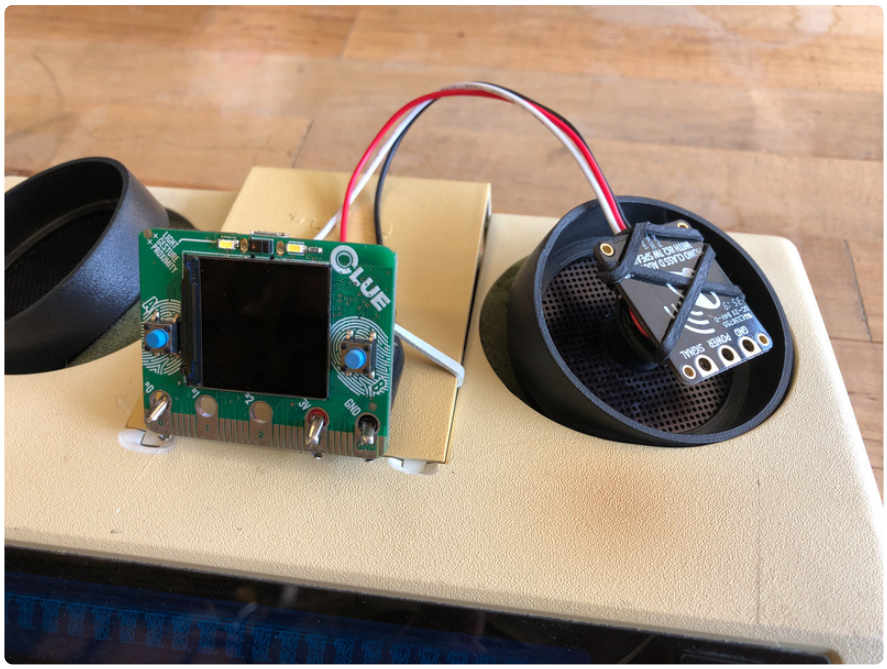


Wrap two rubber bands around the mouthpiece coupler on the TTY machine.

Place the STEMMA speaker directly over the coupler stalk.

Criss cross the rubber bands over the STEMMA speaker board to hold it snugly in place (the fewer unwanted mechanical vibrations the better).

You may also use a single rubber band to hold the CLUE board onto the battery cover as shown.



Power

Most TTY machines can be powered from batteries or a DC 9V wall adapter.

You can choose to power your CLUE from USB power or a battery, such as a LiPo or a small 3x AAA battery pack.

Code the TTY Transmitter

Text Editor

Adafruit recommends using the Mu editor for editing your CircuitPython code. You can get more info in this guide (https://adafru.it/ANO).

Alternatively, you can use any text editor that saves files.

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 **Baudot_TTY/baudot_tty/** 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:


```
# SPDX-FileCopyrightText: 2020 John Park for Adafruit Industries
#
# SPDX-License-Identifier: MIT

### Baudot TTY Message Transmitter

### The 5-bit mode is defined in ANSI TIA/EIA-825 (2000)
### "A Frequency Shift Keyed Modem for use on the Public Switched Telephone Network"

import time
import math
import array
import board
from audiocore import RawSample
import audiopwmio

# constants for sine wave generation
SIN_LENGTH = 100  # more is less choppy
SIN_AMPLITUDE = 2 ** 12  # 0 (min) to 32768 (max)  8192 is nice
SIN_OFFSET = 32767.5  # for 16bit range, (2**16 - 1) / 2
DELTA_PI = 2 * math.pi / SIN_LENGTH  # happy little constant

sine_wave = [
    int(SIN_OFFSET + SIN_AMPLITUDE * math.sin(DELTA_PI * i)) for i in
range(SIN_LENGTH)
]
tones = (
    RawSample(array.array("H", sine_wave), sample_rate=1800 * SIN_LENGTH),  # Bit 0
    RawSample(array.array("H", sine_wave), sample_rate=1400 * SIN_LENGTH),  # Bit 1
)

bit_0 = tones[0]
bit_1 = tones[1]
carrier = tones[1]


char_pause = 0.1  # pause time between chars, set to 0 for fastest rate possible

dac = audiopwmio.PWMAudioOut(
    board.A2
)  # the CLUE edge connector marked "#0" to STEMMA speaker
# The CLUE's on-board speaker works OK, not great, just crank amplitude to full
before trying.
```

```
# dac = audiopwmio.PWMAudioOut(board.SPEAKER)


LTRS = (
    "\b",
    "E",
    "\n",
    "A",
    " ",
    "S",
    "I",
    "U",
    "\r",
    "D",
    "R",
    "J",
    "N",
    "F",
    "C",
    "K",
    "T",
    "Z",
    "L",
    "W",
    "H",
    "Y",
    "P",
    "Q",
    "O",
    "B",
    "G",
    "FIGS",
    "M",
    "X",
    "V",
    "LTRS",
)

FIGS = (
    "\b",
    "3",
    "\n",
    "-",
    " ",
    "-",
    "8",
    "7",
    "\r",
    "$",
    "4",
    "'",
    ",",
    "!",
    ":",
    "(",
    "5",
    '"',
    ")",
    "2",
    "=",
    "6",
    "0",
    "1",
    "9",
    "?",
    "+",
    "FIGS",
    ".",
    "/",
```

```
    ";",
    "LTRS",
)

char_count = 0
current_mode = LTRS

#  The 5-bit Baudot text telephone (TTY) mode is a Frequency Shift Keyed modem
#  for use on the Public Switched Telephone network.
#
#   Definitions:
#       Carrier tone is a 1400Hz tone.
#       Binary 0 is an 1800Hz tone.
#       Binary 1 is a 1400Hz tone.
#       Bit duration is 20ms.

#       Two modes exist: Letters, aka LTRS, for alphabet characters
#       and Figures aka FIGS for numbers and symbols. These modes are switched by
#       sending the appropriate 5-bit LTRS or FIGS character.
#
#   Character transmission sequence:
#       Carrier tone transmits for 150ms before each character.
#       Start bit is a binary 0 (sounded for one bit duration of 20ms).
#       5-bit character code can be a combination of binary 0s and binary 1s.
#       Stop bit is a binary 1 with a minimum duration of 1-1/2 bits (30ms)
#
#


def baudot_bit(pitch=bit_1, duration=0.022):  # spec says 20ms, but adjusted as
needed
    dac.play(pitch, loop=True)
    time.sleep(duration)
    # dac.stop()


def baudot_carrier(duration=0.15):  # Carrier tone is transmitted for 150 ms before
the
    # first character is transmitted
    baudot_bit(carrier, duration)
    dac.stop()


def baudot_start():
    baudot_bit(bit_0)


def baudot_stop():
    baudot_bit(bit_1, 0.04)  # minimum duration is 30ms
    dac.stop()


def send_character(value):
    baudot_carrier()  # send carrier tone
    baudot_start()  # send start bit tone
    for i in range(5):  # send each bit of the character
        bit = (value >> i) & 0x01  # bit shift and bit mask to get value of each bit
        baudot_bit(tones[bit])  # send each bit, either 0 or 1, of a character
    baudot_stop()  # send stop bit
    baudot_carrier()  # not to spec, but works better to extend carrier


def send_message(text):
    global char_count, current_mode  # pylint: disable=global-statement
    for char in text:
        if char not in LTRS and char not in FIGS:  # just skip unknown characters
            print("Unknown character:", char)
            continue
```

```
        if char not in current_mode:  # switch mode
            if current_mode == LTRS:
                print("Switching mode to FIGS")
                current_mode = FIGS
                send_character(current_mode.index("FIGS"))
            elif current_mode == FIGS:
                print("Switching mode to LTRS")
                current_mode = LTRS
                send_character(current_mode.index("LTRS"))
        # Send char mode at beginning of message and every 72 characters
        if char_count >= 72 or char_count == 0:
            print("Resending mode")
            if current_mode == LTRS:
                send_character(current_mode.index("LTRS"))
            elif current_mode == FIGS:
                send_character(current_mode.index("FIGS"))
            # reset counter
            char_count = 0
        print(char)
        send_character(current_mode.index(char))
        time.sleep(char_pause)
        # increment counter
        char_count += 1


while True:
    send_message("\nADAFRUIT 1234567890 -$!+='()/:;?,. ")
    time.sleep(2)
    send_message("\nWELCOME TO JOHN PARK'S WORKSHOP!")
    time.sleep(3)
    send_message("\nWOULD YOU LIKE TO PLAY A GAME?")
    time.sleep(5)

    # here's an example of sending a character
    # send_character(current_mode.index("A"))
    # time.sleep(char_pause)
```

Here's how the code works:

Libraries

First, we'll import the necessary libraries, including the audiocore RawSample and audiopwmio that allow us to create an play tones over the analog output pin.

```
import time
import math
import array
import board
from audiocore import RawSample
import audiopwmio
```

Sine Waves

Next we'll create some constants and code to generate a couple of sine wave tables, one at 1400Hz and the other at 1800Hz.

```
SIN_LENGTH = 100  # more is less choppy
SIN_AMPLITUDE = 2 ** 12  # 0 (min) to 32768 (max)  8192 is nice
SIN_OFFSET = 32767.5  # for 16bit range, (2**16 - 1) / 2
DELTA_PI = 2 * math.pi / SIN_LENGTH  # happy little constant

sine_wave = [
    int(SIN_OFFSET + SIN_AMPLITUDE * math.sin(DELTA_PI * i)) for i in
range(SIN_LENGTH)
]
tones = (
    RawSample(array.array("H", sine_wave), sample_rate=1800 * SIN_LENGTH),  # Bit 0
    RawSample(array.array("H", sine_wave), sample_rate=1400 * SIN_LENGTH),  # Bit 1
)
```

Lists

We'll create a pair of lists called `LTRS` and `FIGS` that contain the full character sets we'll be able to send.

We'll also set the `current_mode` to `LTRS` for purposes of sending the mode code and switching between the modes.

Baudot Functions

A series of functions are used to create the different uses of the sine waves for carrier tone, binary 0 bit, binary 1 bit, start bit, and stop bit.

```
def baudot_bit(pitch=bit_1, duration=0.022):  # spec says 20ms, but adjusted as
needed
    dac.play(pitch, loop=True)
    time.sleep(duration)
    # dac.stop()


def baudot_carrier(duration=0.15):  # Carrier tone is transmitted for 150 ms before
the
    # first character is transmitted
    baudot_bit(carrier, duration)
    dac.stop()


def baudot_start():
    baudot_bit(bit_0)


def baudot_stop():
    baudot_bit(bit_1, 0.04)  # minimum duration is 30ms
    dac.stop()
```

Send Character

The `send_character()` function bundles up the parts into a proper TTY compliant message including the carrier tone, start bit, 5-bit character, stop bit, and carrier tone

again. It receives a value argument of a 5-bit binary code from the LTRS or FIGS list and marches through this from LSB first, using bit shifting and bit masking to grab each relevant bit and convert it to the proper tone.

```
def send_character(value):
    baudot_carrier()  # send carrier tone
    baudot_start()  # send start bit tone
    for i in range(5):  # send each bit of the character
        bit = (value &gt;&gt; i) &amp; 0x01  # bit shift and bit mask to get value
of each bit
        baudot_bit(tones[bit])  # send each bit, either 0 or 1, of a character
    baudot_stop()  # send stop bit
    baudot_carrier()  # not to spec, but works better to extend carrier
```

Send Message

The `send_message()` function is a convenience function for bundling up a whole message string and then one at a time converting the characters to proper `send_character()` commands.

This includes testing each character to see if it is a LTRS or FIGS list item, and then sending the proper mode character if needed. It also follows the spec and sends the relevant mode character after every 72 characters.

```
def send_message(text):
    global char_count, current_mode  # pylint: disable=global-statement
    for char in text:
        if char not in LTRS and char not in FIGS:  # just skip unknown characters
            print("Unknown character:", char)
            continue

        if char not in current_mode:  # switch mode
            if current_mode == LTRS:
                print("Switching mode to FIGS")
                current_mode = FIGS
                send_character(current_mode.index("FIGS"))
            elif current_mode == FIGS:
                print("Switching mode to LTRS")
                current_mode = LTRS
                send_character(current_mode.index("LTRS"))
        # Send char mode at beginning of message and every 72 characters
        if char_count &gt;= 72 or char_count == 0:
            print("Resending mode")
            if current_mode == LTRS:
                send_character(current_mode.index("LTRS"))
            elif current_mode == FIGS:
                send_character(current_mode.index("FIGS"))
            # reset counter
            char_count = 0
        print(char)
        send_character(current_mode.index(char))
        time.sleep(char_pause)
        # increment counter
        char_count += 1
```

Main Loop

The main loop of the program sends whatever messages are specified. In these demos the \n carriage return is used to add a break between messages. There is also a commented sample of sending a single character.

```
while True:
    send_message("\nADAFRUIT 1234567890 -$!+='()/:;?,. ")
    time.sleep(2)
    send_message("\nWELCOME TO JOHN PARK'S WORKSHOP!")
    time.sleep(3)
    send_message("\nWOULD YOU LIKE TO PLAY A GAME?")
    time.sleep(5)

    # here's an example of sending a character
    # send_character(current_mode.index("A"))
    # time.sleep(char_pause)
```

Code the BLE TTY Transmitter

After trying the simple example on the previous page, you can switch to this code to try out Bluetooth LE functionality using the Adafruit Bluefruit LE Connect app on your iOS or Android device.

See this guide (https://adafru.it/DNc) to get the app installed and set up.

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 **Baudot_tty/baudot_tty_ble/** 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:


```
# SPDX-FileCopyrightText: 2020 John Park for Adafruit Industries
#
# SPDX-License-Identifier: MIT

### Baudot TTY Message Transmitter
### Bluefruit Connect UART mode to send messages to CLUE for audio
### tramsission to TTY machine.

### The 5-bit mode is defined in ANSI TIA/EIA-825 (2000)
### "A Frequency Shift Keyed Modem for use on the Public Switched Telephone Network"

import time
import math
import array
import board
import audiopwmio
from audiocore import RawSample
from adafruit_ble import BLERadio
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.services.nordic import UARTService

# BLE radio setup
ble = BLERadio()
uart_server = UARTService()
advertisement = ProvideServicesAdvertisement(uart_server)
ble._adapter.name = "TTY_MACHINE"  # pylint: disable=protected-access

# constants for sine wave generation
SIN_LENGTH = 100  # more is less choppy
SIN_AMPLITUDE = 2 ** 12  # 0 (min) to 32768 (max)  8192 is nice
SIN_OFFSET = 32767.5  # for 16bit range, (2**16 - 1) / 2
DELTA_PI = 2 * math.pi / SIN_LENGTH  # happy little constant

sine_wave = [
    int(SIN_OFFSET + SIN_AMPLITUDE * math.sin(DELTA_PI * i)) for i in
range(SIN_LENGTH)
]
tones = (
    RawSample(array.array("H", sine_wave), sample_rate=1800 * SIN_LENGTH),  # Bit 0
    RawSample(array.array("H", sine_wave), sample_rate=1400 * SIN_LENGTH),  # Bit 1
)

bit_0 = tones[0]
bit_1 = tones[1]
carrier = tones[1]


char_pause = 0.0  # pause time between chars, set to 0 for fastest rate possible

dac = audiopwmio.PWMAudioOut(
    board.A2
)  # the CLUE edge connector marked "#0" to STEMMA speaker
# The CLUE's on-board speaker works OK, not great, just crank amplitude to full
before trying.
# dac = audiopwmio.PWMAudioOut(board.SPEAKER)
```

```
LTRS = (
    "\b",
    "E",
    "\n",
    "A",
    " ",
    "S",
    "I",
    "U",
    "\r",
    "D",
    "R",
    "J",
    "N",
    "F",
    "C",
    "K",
    "T",
    "Z",
    "L",
    "W",
    "H",
    "Y",
    "P",
    "Q",
    "O",
    "B",
    "G",
    "FIGS",
    "M",
    "X",
    "V",
    "LTRS",
)

FIGS = (
    "\b",
    "3",
    "\n",
    "-",
    " ",
    "-",
    "8",
    "7",
    "\r",
    "$",
    "4",
    "'",
    ",",
    "!",
    ":",
    "(",
    "5",
    '"',
    ")",
    "2",
    "=",
    "6",
    "0",
    "1",
    "9",
    "?",
    "+",
    "FIGS",
    ".",
    "/",
    ";",
```

```
    "LTRS",
)

char_count = 0
current_mode = LTRS

#  The 5-bit Baudot text telephone (TTY) mode is a Frequency Shift Keyed modem
#  for use on the Public Switched Telephone network.
#
#   Definitions:
#       Carrier tone is a 1400Hz tone.
#       Binary 0 is an 1800Hz tone.
#       Binary 1 is a 1400Hz tone.
#       Bit duration is 20ms.
#
#       Two modes exist: Letters, aka LTRS, for alphabet characters
#       and Figures aka FIGS for numbers and symbols. These modes are switched by
#       sending the appropriate 5-bit LTRS or FIGS character.
#
#   Character transmission sequence:
#       Carrier tone transmits for 150ms before each character.
#       Start bit is a binary 0 (sounded for one bit duration of 20ms).
#       5-bit character code can be a combination of binary 0s and binary 1s.
#       Stop bit is a binary 1 with a minimum duration of 1-1/2 bits (30ms)


def baudot_bit(pitch=bit_1, duration=0.022):  # spec says 20ms, but adjusted as
needed
    dac.play(pitch, loop=True)
    time.sleep(duration)
    # dac.stop()


def baudot_carrier(duration=0.15):
    # Carrier is transmitted 150 ms before first character is sent
    baudot_bit(carrier, duration)
    dac.stop()


def baudot_start():
    baudot_bit(bit_0)


def baudot_stop():
    baudot_bit(bit_1, 0.04)  # minimum duration is 30ms
    dac.stop()


def send_character(value):
    baudot_carrier()  # send carrier tone
    baudot_start()  # send start bit tone
    for i in range(5):  # send each bit of the character
        bit = (value >> i) & 0x01  # bit shift and bit mask to get value of each bit
        baudot_bit(tones[bit])  # send each bit, either 0 or 1, of a character
    baudot_stop()  # send stop bit
    baudot_carrier()  # not to spec, but works better to extend carrier


def send_message(text):
    global char_count, current_mode  # pylint: disable=global-statement
    for char in text:
        if char not in LTRS and char not in FIGS:  # just skip unknown characters
            print("Unknown character:", char)
            continue

        if char not in current_mode:  # switch mode
            if current_mode == LTRS:
                print("Switching mode to FIGS")
                current_mode = FIGS
```

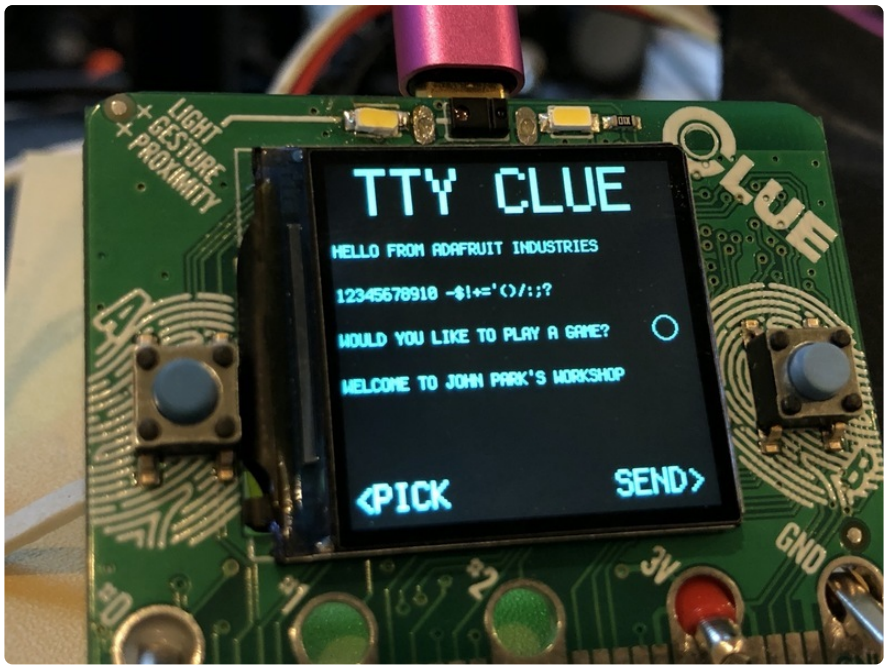
```
                send_character(current_mode.index("FIGS"))
            elif current_mode == FIGS:
                print("Switching mode to LTRS")
                current_mode = LTRS
                send_character(current_mode.index("LTRS"))
        # Send char mode at beginning of message and every 72 characters
        if char_count >= 72 or char_count == 0:
            print("Resending mode")
            if current_mode == LTRS:
                send_character(current_mode.index("LTRS"))
            elif current_mode == FIGS:
                send_character(current_mode.index("FIGS"))
            # reset counter
            char_count = 0
        print(char)
        send_character(current_mode.index(char))
        time.sleep(char_pause)
        # increment counter
        char_count += 1


while True:
    print("WAITING...")
    send_message("\nWAITING...\n")
    ble.start_advertising(advertisement)
    while not ble.connected:
        pass

    # Connected
    ble.stop_advertising()
    print("CONNECTED")
    send_message("\nCONNECTED\n")

    # Loop and read packets
    while ble.connected:
        if uart_server.in_waiting:
            raw_bytes = uart_server.read(uart_server.in_waiting)
            textmsg = raw_bytes.decode().strip()
            print("received text =", textmsg)
            send_message("\n")
            send_message(textmsg.upper())

    # Disconnected
    print("DISCONNECTED")
    send_message("\nDISCONNECTED\n")
```

Once you've copied the **code.py** code file to your CLUE, you can connect from the Bluefruit app to the CLUE device named TTY Machine and then use the UART function to send messages to the CLUE.

The letters will be received by the CLUE and then sent as audio to the TTY machine!

Code the TTY GUI

This version of the code adds in a simple GUI for selecting a phrase with the A button and sending it with the B button.

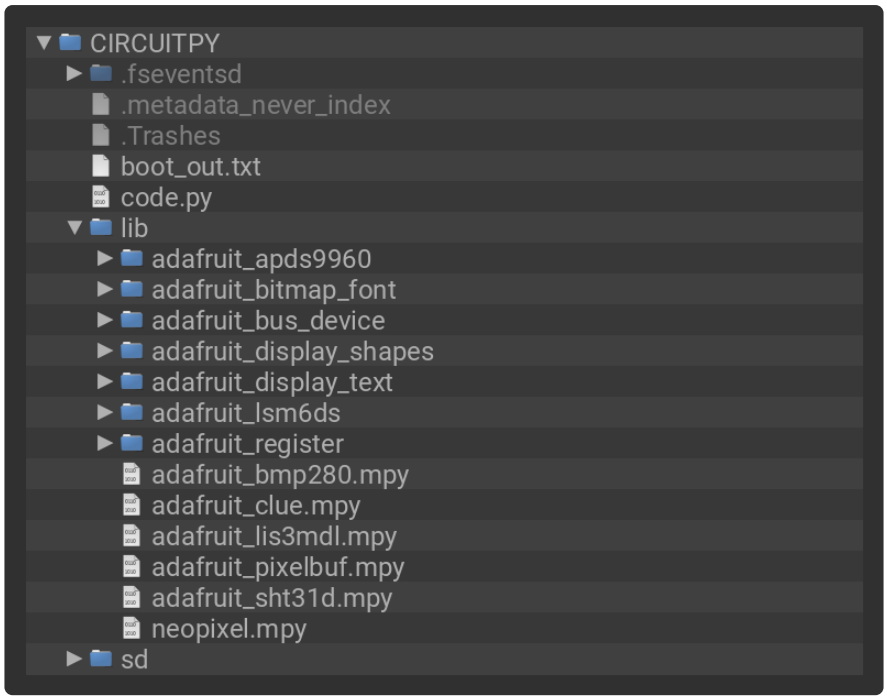


Installing Project Code

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 **Baudot_TTY/baudot_tty_gui/** 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:




```
# SPDX-FileCopyrightText: 2020 John Park for Adafruit Industries
#
# SPDX-License-Identifier: MIT

### Baudot TTY Message Transmitter with CLUE GUI
### Pick from four phrases to send from the CLUE screen with buttons

### The 5-bit mode is defined in ANSI TIA/EIA-825 (2000)
### "A Frequency Shift Keyed Modem for use on the Public Switched Telephone Network"

import time
import math
import array
import board
from audiocore import RawSample
import audiopwmio
import displayio
from adafruit_display_shapes.circle import Circle
from adafruit_clue import clue
from adafruit_display_text import label
import terminalio

# Enter your messages here no more than 34 characters including spaces per line
messages = [
    "HELLO FROM ADAFRUIT INDUSTRIES",
    "12345678910 -$!+='()/:;?",
    "WOULD YOU LIKE TO PLAY A GAME?",
    "WELCOME TO JOHN PARK'S WORKSHOP",
]


clue.display.brightness = 1.0
screen = displayio.Group()

VFD_GREEN = 0x00FFD2
VFD_BG = 0x000505

# setup screen
# BG
color_bitmap = displayio.Bitmap(240, 240, 1)
color_palette = displayio.Palette(1)
color_palette[0] = VFD_BG
```

```
bg_sprite = displayio.TileGrid(color_bitmap, x=0, y=0, pixel_shader=color_palette)
screen.append(bg_sprite)

# title
title_label = label.Label(
    terminalio.FONT, text="TTY CLUE", scale=4, color=VFD_GREEN
)
title_label.x = 20
title_label.y = 16
screen.append(title_label)

# footer
footer_label = label.Label(
    terminalio.FONT, text="<PICK         SEND>", scale=2, color=VFD_GREEN
)
footer_label.x = 4
footer_label.y = 220
screen.append(footer_label)

# message configs
messages_config = [
    (0, messages[0], VFD_GREEN, 2, 60),
    (1, messages[1], VFD_GREEN, 2, 90),
    (2, messages[2], VFD_GREEN, 2, 120),
    (3, messages[3], VFD_GREEN, 2, 150),
]

messages_labels = {}  # dictionary of configured messages_labels

message_group = displayio.Group(scale=1)

for message_config in messages_config:
    (name, textline, color, x, y) = message_config  # unpack tuple into five var
names
    message_label = label.Label(terminalio.FONT, text=textline, color=color)
    message_label.x = x
    message_label.y = y
    messages_labels[name] = message_label
    message_group.append(message_label)
screen.append(message_group)

# selection dot
dot_y = [52, 82, 112, 142]
dot = Circle(220, 60, 8, outline=VFD_GREEN, fill=VFD_BG)
screen.append(dot)

clue.display.root_group = screen

# constants for sine wave generation
SIN_LENGTH = 100  # more is less choppy
SIN_AMPLITUDE = 2 ** 12  # 0 (min) to 32768 (max)  8192 is nice
SIN_OFFSET = 32767.5  # for 16bit range, (2**16 - 1) / 2
DELTA_PI = 2 * math.pi / SIN_LENGTH  # happy little constant

sine_wave = [
    int(SIN_OFFSET + SIN_AMPLITUDE * math.sin(DELTA_PI * i)) for i in
range(SIN_LENGTH)
]
tones = (
    RawSample(array.array("H", sine_wave), sample_rate=1800 * SIN_LENGTH),  # Bit 0
    RawSample(array.array("H", sine_wave), sample_rate=1400 * SIN_LENGTH),  # Bit 1
)

bit_0 = tones[0]
bit_1 = tones[1]
carrier = tones[1]


char_pause = 0.1  # pause time between chars, set to 0 for fastest rate possible
```

```
dac = audiopwmio.PWMAudioOut(
    board.A2
)  # the CLUE edge connector marked "#0" to STEMMA speaker
# The CLUE's on-board speaker works OK, not great, just crank amplitude to full
before trying.
# dac = audiopwmio.PWMAudioOut(board.SPEAKER)


LTRS = (
    "\b",
    "E",
    "\n",
    "A",
    " ",
    "S",
    "I",
    "U",
    "\r",
    "D",
    "R",
    "J",
    "N",
    "F",
    "C",
    "K",
    "T",
    "Z",
    "L",
    "W",
    "H",
    "Y",
    "P",
    "Q",
    "O",
    "B",
    "G",
    "FIGS",
    "M",
    "X",
    "V",
    "LTRS",
)

FIGS = (
    "\b",
    "3",
    "\n",
    "-",
    " ",
    "-",
    "8",
    "7",
    "\r",
    "$",
    "4",
    "'",
    ",",
    "!",
    ":",
    "(",
    "5",
    '"',
    ")",
    "2",
    "=",
    "6",
    "0",
    "1",
```

```
    "9",
    "?",
    "+",
    "FIGS",
    ".",
    "/",
    ";",
    "LTRS",
)

char_count = 0
current_mode = LTRS

#  The 5-bit Baudot text telephone (TTY) mode is a Frequency Shift Keyed modem
#  for use on the Public Switched Telephone network.
#
#   Definitions:
#       Carrier tone is a 1400Hz tone.
#       Binary 0 is an 1800Hz tone.
#       Binary 1 is a 1400Hz tone.
#       Bit duration is 20ms.

#       Two modes exist: Letters, aka LTRS, for alphabet characters
#       and Figures aka FIGS for numbers and symbols. These modes are switched by
#       sending the appropriate 5-bit LTRS or FIGS character.
#
#   Character transmission sequence:
#       Carrier tone transmits for 150ms before each character.
#       Start bit is a binary 0 (sounded for one bit duration of 20ms).
#       5-bit character code can be a combination of binary 0s and binary 1s.
#       Stop bit is a binary 1 with a minimum duration of 1-1/2 bits (30ms)
#
#


def baudot_bit(pitch=bit_1, duration=0.022):  # spec says 20ms, but adjusted as
needed
    dac.play(pitch, loop=True)
    time.sleep(duration)
    # dac.stop()


def baudot_carrier(duration=0.15):  # Carrier tone is transmitted for 150 ms before
the
    # first character is transmitted
    baudot_bit(carrier, duration)
    dac.stop()


def baudot_start():
    baudot_bit(bit_0)


def baudot_stop():
    baudot_bit(bit_1, 0.04)  # minimum duration is 30ms
    dac.stop()


def send_character(value):
    baudot_carrier()  # send carrier tone
    baudot_start()  # send start bit tone
    for i in range(5):  # send each bit of the character
        bit = (value >> i) & 0x01  # bit shift and bit mask to get value of each bit
        baudot_bit(tones[bit])  # send each bit, either 0 or 1, of a character
    baudot_stop()  # send stop bit
    baudot_carrier()  # not to spec, but works better to extend carrier


def send_message(text):
```

```
    global char_count, current_mode  # pylint: disable=global-statement
    for char in text:
        if char not in LTRS and char not in FIGS:  # just skip unknown characters
            print("Unknown character:", char)
            continue

        if char not in current_mode:  # switch mode
            if current_mode == LTRS:
                print("Switching mode to FIGS")
                current_mode = FIGS
                send_character(current_mode.index("FIGS"))
            elif current_mode == FIGS:
                print("Switching mode to LTRS")
                current_mode = LTRS
                send_character(current_mode.index("LTRS"))
        # Send char mode at beginning of message and every 72 characters
        if char_count >= 72 or char_count == 0:
            print("Resending mode")
            if current_mode == LTRS:
                send_character(current_mode.index("LTRS"))
            elif current_mode == FIGS:
                send_character(current_mode.index("FIGS"))
            # reset counter
            char_count = 0
        print(char)
        send_character(current_mode.index(char))
        time.sleep(char_pause)
        # increment counter
        char_count += 1


message_pick = 0

while True:
    if clue.button_a:
        message_pick = (message_pick + 1) % 4  # loop through the lines
        dot.y = dot_y[message_pick]
        time.sleep(0.4)  # debounce

    if clue.button_b:
        dot.fill = VFD_GREEN
        send_message(messages[message_pick])
        dot.fill = VFD_BG
```