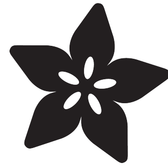
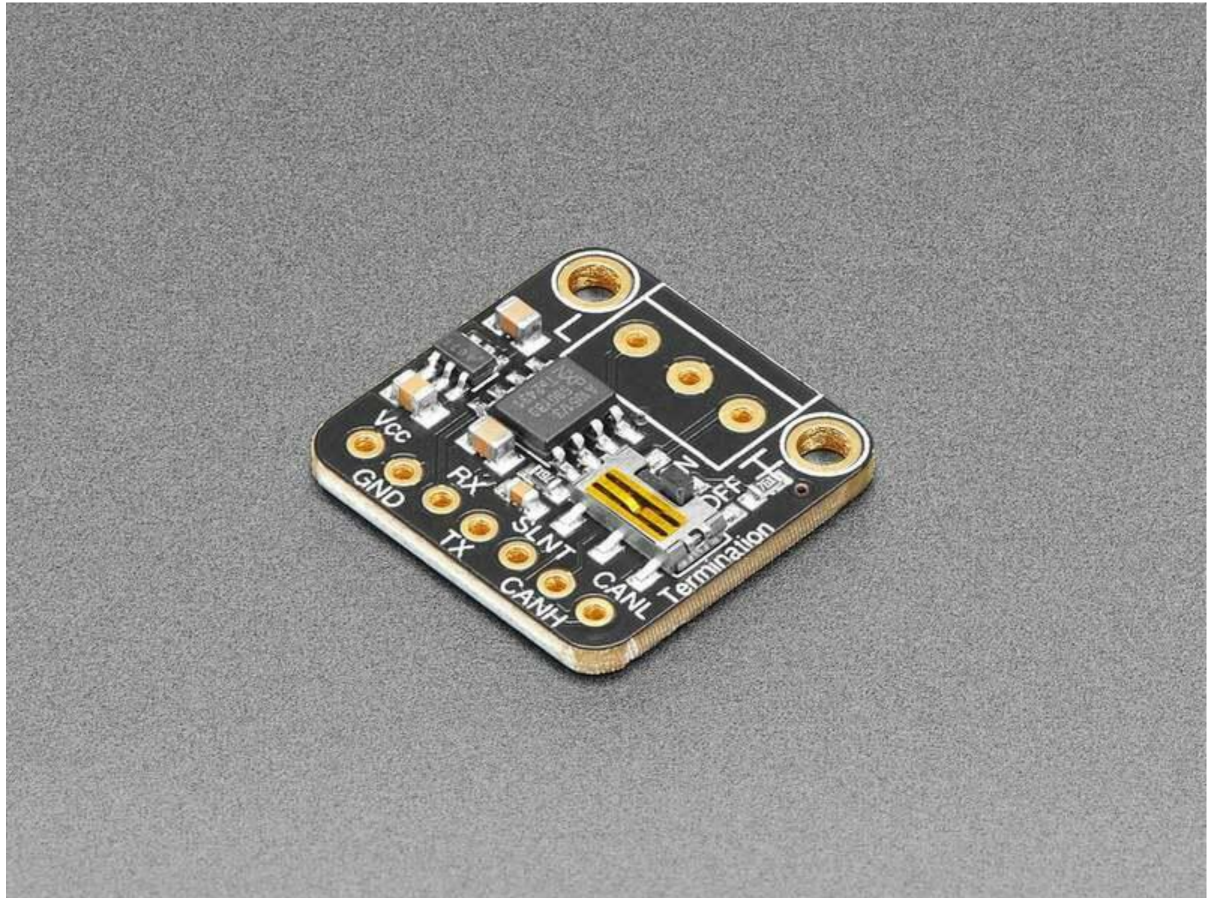


Adafruit CAN Pal

Created by Liz Clark



https://learn.adafruit.com/adafruit-can-pal

Last updated on 2023-03-28 04:24:59 PM EDT

Table of Contents

Overview 3

Pinouts 6

- Terminal Block Pins
- Power Pins
- CAN Logic Pins
- Termination On/Off Switch

CircuitPython 7

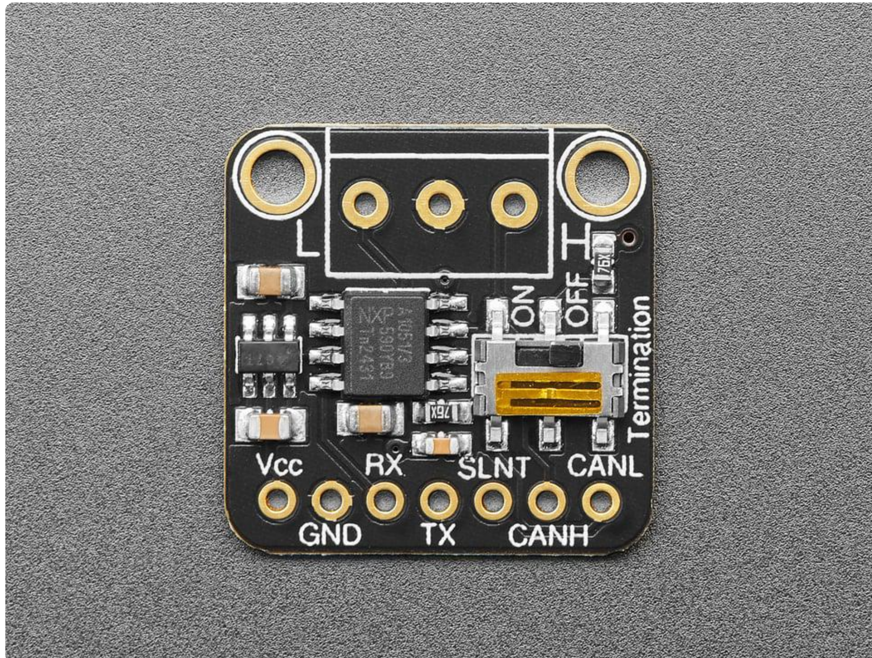
- CircuitPython Microcontroller Wiring
- CircuitPython Usage
- CAN Sender Example
- CAN Listener Example
- Going Further

Python Docs 12

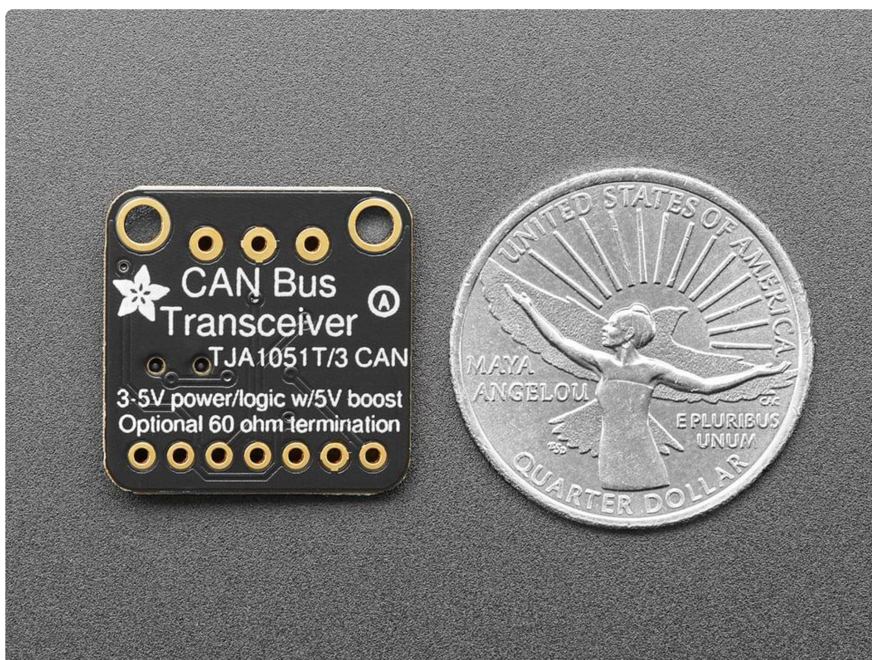
Downloads 12

- Files
- Schematic and Fab Print

Overview

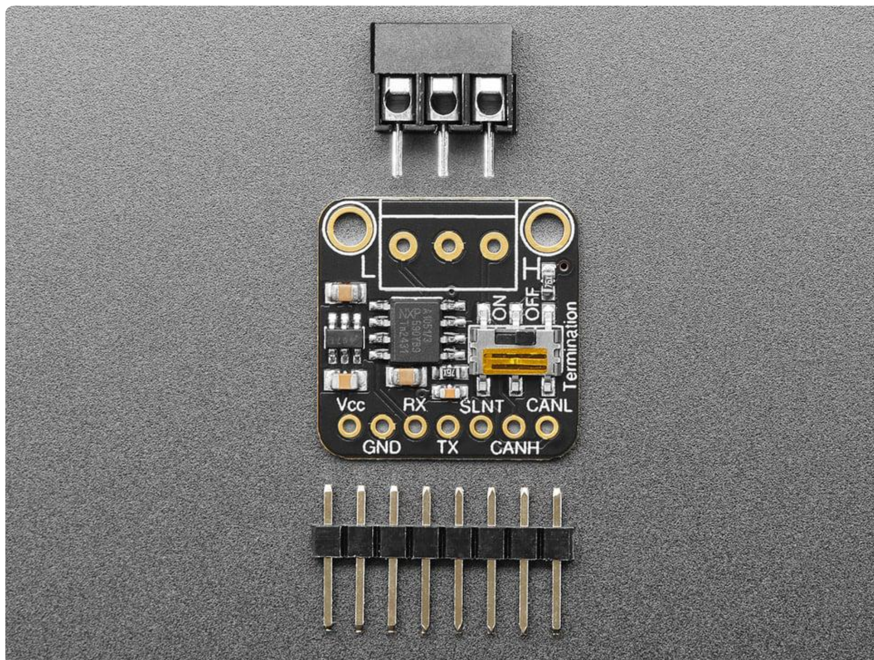


If you'd like to connect a board with native CAN Peripheral support, the Adafruit CAN Pal Transceiver will take the 3V logic level signals and convert them to CAN logic levels with the differential signaling required to communicate. Note that not all chips have a CAN peripheral! Some that we know do have it are the ESP32/ESP32-S2/ESP32-S3 () (note that ESP32 calls this interface TWAI not CAN) series of chips, SAME 51 (), STM32F405 (), and Teensy 4 ().

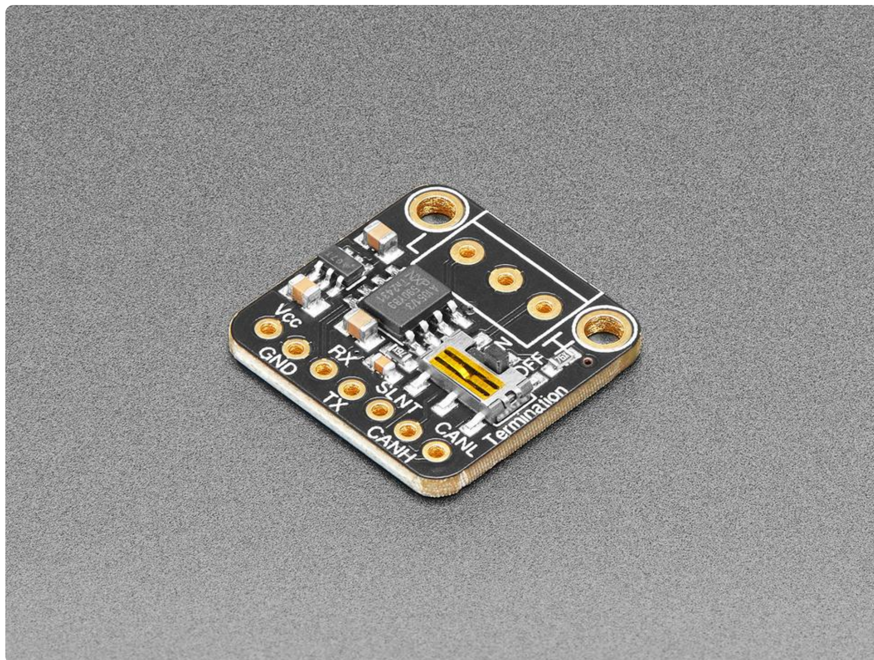


Check the product documentation for the board you are wiring this to, making sure that the chip has CAN support and the RX and TX pins are brought out for you to connect the transceiver to! Despite sharing the 'RX' and 'TX' name with UART, they're not at all the same interface.

CAN Bus is a small-scale networking standard (), originally designed for cars and, yes, busses, but is now used for many robotics or sensor networks that need better range and addressing than I2C, and don't have the pins or computational ability to talk on Ethernet. CAN is 2 wire differential, which means it's good for long distances and noisy environments.



Messages are sent at about 1Mbps rate - you set the frequency for the bus and then all 'joiners' must match it, and have an address before the packet so that each node can listen in to messages just for it. New nodes can be attached easily because they just need to connect to the two data lines anywhere in the shared net. Each CAN devices sends messages whenever it wants, and thanks to some clever data encoding, can detect if there's a message collision and retransmit later.

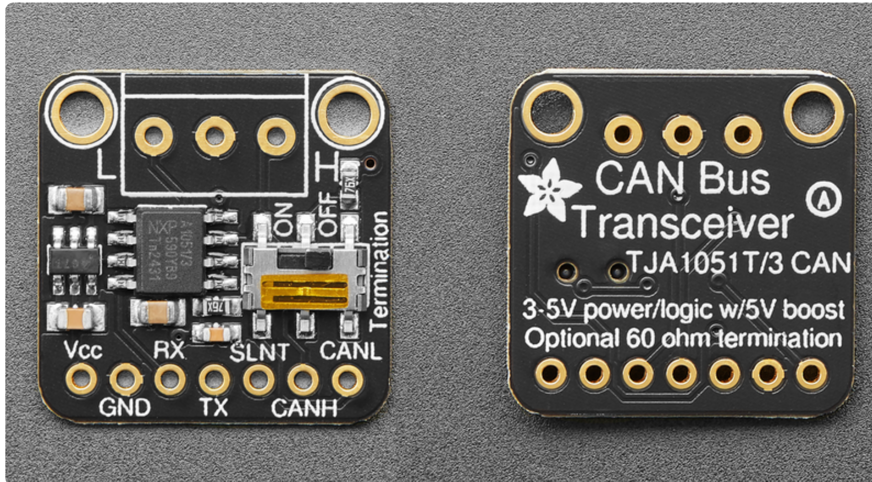


We've added a few nice extras to this breakout pal to make it useful in many common CAN scenarios:

- TJA1051/T3 () can communicate with 3.3V-5V logic for use with modern microcontrollers.
- 5V charge-pump voltage generator (), so even though you are running 3.3V power and logic on most modern microcontroller boards, it will generate a nice clean 5V as required by the transceiver. No separate 5V power required!
- 3.5mm terminal block () that can be soldered in to get quick access to the High and Low data lines as well as a ground pin.
- 2 x 60 ohm termination resistor on board (120 ohm in series), you can remove or activate the termination easily by flipping the onboard switch.

Each order comes with an assembled 'pal, terminal block and header. You will need to solder in the header yourself but it's a quick task.

Pinouts



Terminal Block Pins

On the front of the board are the three pins for the included 3.5 mm terminal block. It is outlined in white on the silk.

- L - the CAN low signal for the CAN Bus standard.
- Middle pin (unlabeled) - common ground shared between the two CAN connections
- H - the CAN high signal for the CAN Bus standard.

Both CAN L and CAN H are connected to a 5V charge-pump voltage generator (). Even if you are using 3.3V logic, it will generate a nice clean 5V as required by the CAN Bus transceiver.

Power Pins

- Vcc - this is the power pin. Since the transceiver chip uses 3-5 VDC, to power the board, give it the same power as the logic level of your microcontroller - e.g. for a 5V micro like Arduino, use 5V.
- GND - common ground for power and logic.

CAN Logic Pins

- RX - CAN receive/input pin for boards with a CAN peripheral.
- TX - CAN transmit/output pin for boards with a CAN peripheral.

Check the product documentation for the board you are wiring this to, to make sure that:

1. The chip has CAN support
2. The RX and TX pins are brought out for you to connect the transceiver to

Despite sharing the 'RX' and 'TX' name with UART, they're not at all the same interface.

- SLNT - can be pulled high to put the TJA1051/3 transceiver into silent mode. In silent mode, the transmitter is disabled, releasing the bus pins to a recessive state.
- CANH - the CAN high signal for the CAN Bus standard.
- CANL - the CAN low signal for the CAN Bus standard.

Termination On/Off Switch

The board has two 60 ohm resistors (120 ohm in series) connected between CANH and CANL. You can disable the terminator by setting the Termination switch OFF, if your bus is already terminated. Otherwise, keep the switch set to ON to enable the termination resistors.

CircuitPython

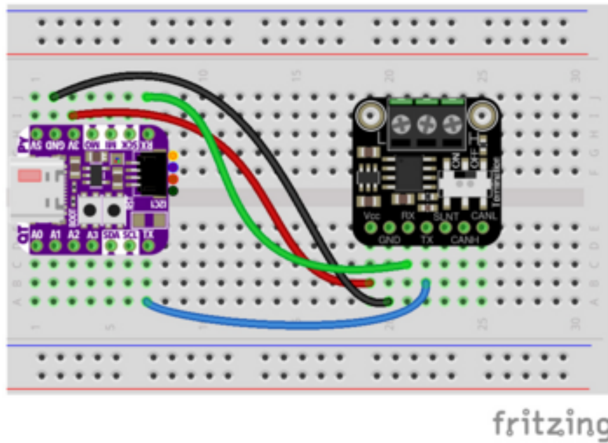
It's easy to use the CAN Pal with CircuitPython and the built-in canio () module. This module allows you to easily write Python code that lets you utilize your board's onboard CAN peripheral with the CAN Pal transceiver.

Note that not all chips have a CAN peripheral! Some that we know do have it are the ESP32/ESP32-S2/ESP32-S3 () (note that ESP32 calls this interface TWAI not CAN) series of chips, SAME51 (), STM32F405 (), and Teensy 4 ().

CircuitPython Microcontroller Wiring

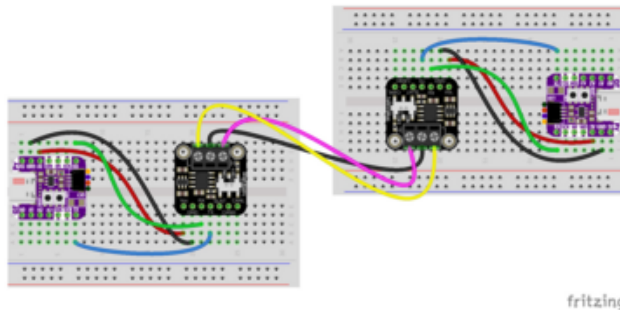
First, wire up a CAN Pal to your board exactly as shown below. You're going to do this twice so you have two CAN pals identically connected to two microcontrollers.

Here's an example of wiring a QT Py ESP32-S2 to the CAN Pal on a breadboard with 0 .100" pitch headers:



QT Py 3.3V to CAN Pal VCC (red wire)
QT Py GND to CAN Pal GND (black wire)
QT Py RX to CAN Pal RX (green wire)
QT Py TX to CAN Pal TX (blue wire)

Then, connect the two CAN Pal CAN Bus connections together via the 3.5 mm terminal blocks:

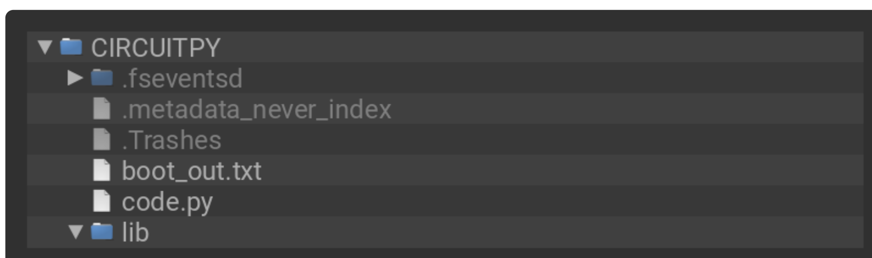


CAN Pal 0 L to CAN Pal 1 L (yellow wire)
CAN Pal 0 GND to CAN Pal 1 GND (black wire)
CAN Pal 0 H to CAN Pal 1 H (pink wire)

CircuitPython Usage

To use with CircuitPython, you need to update code.py with the example script. There are no additional libraries needed since the code is utilizing core modules.

In the examples below, click the Download Project Bundle button below to download the the code.py file in a zip file. Extract the contents of the zip file, and copy the code.py file to your CIRCUITPY drive.



CAN Sender Example

On line 22, change the RX and TX pin definitions to `board.RX` and `board.TX`:

```
can = canio.CAN(rx=board.RX, tx=board.TX, baudrate=250_000, auto_restart=True)
```

Then, upload the code to one of the QT Py ESP32-S2's.

```
# SPDX-FileCopyrightText: 2020 Jeff Epler for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import struct
import time

import board
import canio
import digitalio

# If the CAN transceiver has a standby pin, bring it out of standby mode
if hasattr(board, 'CAN_STANDBY'):
    standby = digitalio.DigitalInOut(board.CAN_STANDBY)
    standby.switch_to_output(False)

# If the CAN transceiver is powered by a boost converter, turn on its supply
if hasattr(board, 'BOOST_ENABLE'):
    boost_enable = digitalio.DigitalInOut(board.BOOST_ENABLE)
    boost_enable.switch_to_output(True)

# Use this line if your board has dedicated CAN pins. (Feather M4 CAN and Feather
STM32F405)
can = canio.CAN(rx=board.CAN_RX, tx=board.CAN_TX, baudrate=250_000,
auto_restart=True)
# On ESP32S2 most pins can be used for CAN.  Uncomment the following line to use
IO5 and IO6
#can = canio.CAN(rx=board.IO6, tx=board.IO5, baudrate=250_000, auto_restart=True)

old_bus_state = None
count = 0

while True:
    bus_state = can.state
    if bus_state != old_bus_state:
        print(f"Bus state changed to {bus_state}")
        old_bus_state = bus_state

    now_ms = (time.monotonic_ns() // 1_000_000) & 0xffffffff
    print(f"Sending message: count={count} now_ms={now_ms}")

    message = canio.Message(id=0x408, data=struct.pack("<II", count, now_ms))
    can.send(message)

    time.sleep(.5)
    count += 1
```

CAN Listener Example

On line 22, change the RX and TX pin definitions to `board.RX` and `board.TX`:

```
can = canio.CAN(rx=board.RX, tx=board.TX, baudrate=250_000, auto_restart=True)
```

Then, upload the code to the remaining QT Py ESP32-S2.

```
# SPDX-FileCopyrightText: 2020 Jeff Epler for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import struct

import board
import canio
import digitalio

# If the CAN transceiver has a standby pin, bring it out of standby mode
if hasattr(board, 'CAN_STANDBY'):
    standby = digitalio.DigitalInOut(board.CAN_STANDBY)
    standby.switch_to_output(False)

# If the CAN transceiver is powered by a boost converter, turn on its supply
if hasattr(board, 'BOOST_ENABLE'):
    boost_enable = digitalio.DigitalInOut(board.BOOST_ENABLE)
    boost_enable.switch_to_output(True)

# Use this line if your board has dedicated CAN pins. (Feather M4 CAN and Feather
STM32F405)
can = canio.CAN(rx=board.CAN_RX, tx=board.CAN_TX, baudrate=250_000,
auto_restart=True)
# On ESP32S2 most pins can be used for CAN.  Uncomment the following line to use
IO5 and IO6
#can = canio.CAN(rx=board.IO6, tx=board.IO5, baudrate=250_000, auto_restart=True)
listener = can.listen(matches=[canio.Match(0x408)], timeout=.9)

old_bus_state = None
old_count = -1

while True:
    bus_state = can.state
    if bus_state != old_bus_state:
        print(f"Bus state changed to {bus_state}")
        old_bus_state = bus_state

    message = listener.receive()
    if message is None:
        print("No messsage received within timeout")
        continue

    data = message.data
    if len(data) != 8:
        print(f"Unusual message length {len(data)}")
        continue

    count, now_ms = struct.unpack("<II", data)
    gap = count - old_count
    old_count = count
    print(f"received message: count={count} now_ms={now_ms}")
```

```
    if gap != 1:
        print(f"gap: {gap}")
```

In the REPL, you'll be able to see the message count and timestamp as they are sent and received. For the QT Py ESP32-S2 running the Sender example, the REPL will look like this:

```
CircuitPython REPL
Sending message: count=3 now_ms=1575888
Sending message: count=4 now_ms=1576390
Sending message: count=5 now_ms=1576892
Sending message: count=6 now_ms=1577394
Sending message: count=7 now_ms=1577895
Sending message: count=8 now_ms=1578397
Sending message: count=9 now_ms=1578899
Sending message: count=10 now_ms=1579401
Sending message: count=11 now_ms=1579903
Sending message: count=12 now_ms=1580405
Sending message: count=13 now_ms=1580907
Sending message: count=14 now_ms=1581409
Sending message: count=15 now_ms=1581911
```

For the QT Py ESP32-S2 running the Listener example, the REPL will look like this:

```
CircuitPython REPL
received message: count=52 now_ms=1600484
received message: count=53 now_ms=1600986
received message: count=54 now_ms=1601487
received message: count=55 now_ms=1601989
received message: count=56 now_ms=1602491
received message: count=57 now_ms=1602992
received message: count=58 now_ms=1603494
received message: count=59 now_ms=1603996
received message: count=60 now_ms=1604497
received message: count=61 now_ms=1604999
received message: count=62 now_ms=1605501
received message: count=63 now_ms=1606002
received message: count=64 now_ms=1606504
```

Going Further

For more information on using CAN Bus with CircuitPython, check out the CAN Bus with CircuitPython: Using the canio module Learn Guide ().

Python Docs

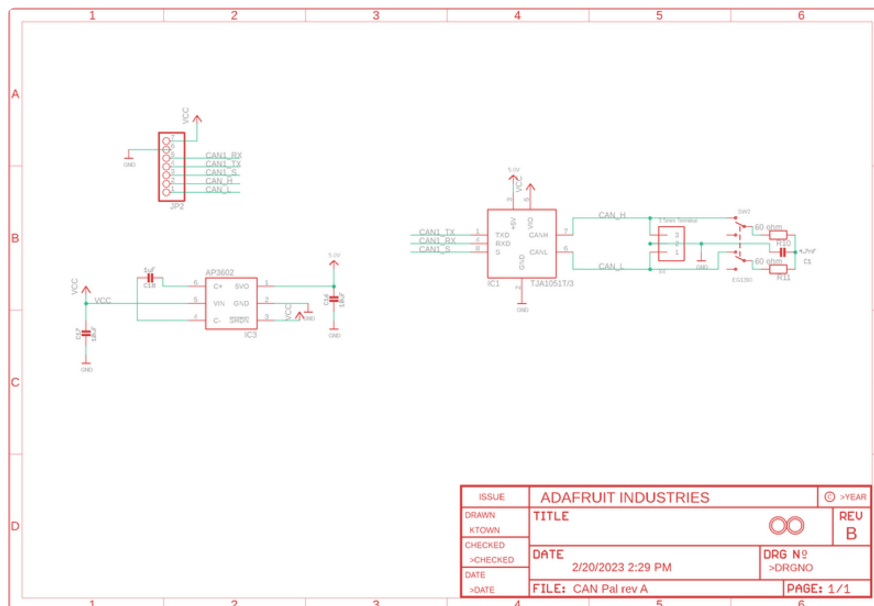
Python Docs ()

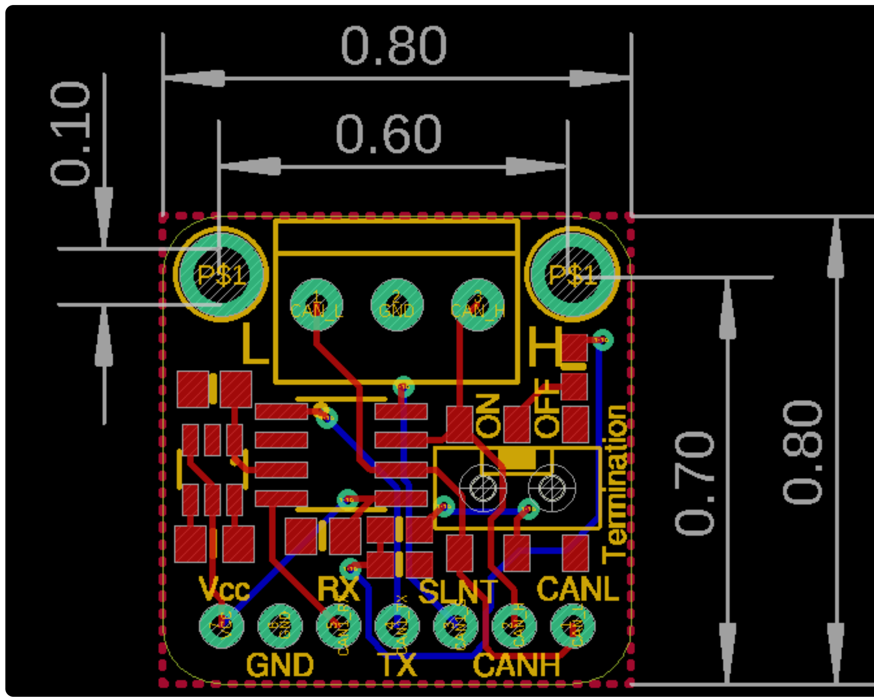
Downloads

Files

- TJA1051/3 Datasheet ()
- EagleCAD PCB files on GitHub ()
- Fritzing object in the Adafruit Fritzing Library ()

Schematic and Fab Print




0.80
0.60
0.10
0.70
0.80
P$1
P$1
L
H
ON
OFF
Termination
Vcc
RX
SLNT
CANL
GND
TX
CANH