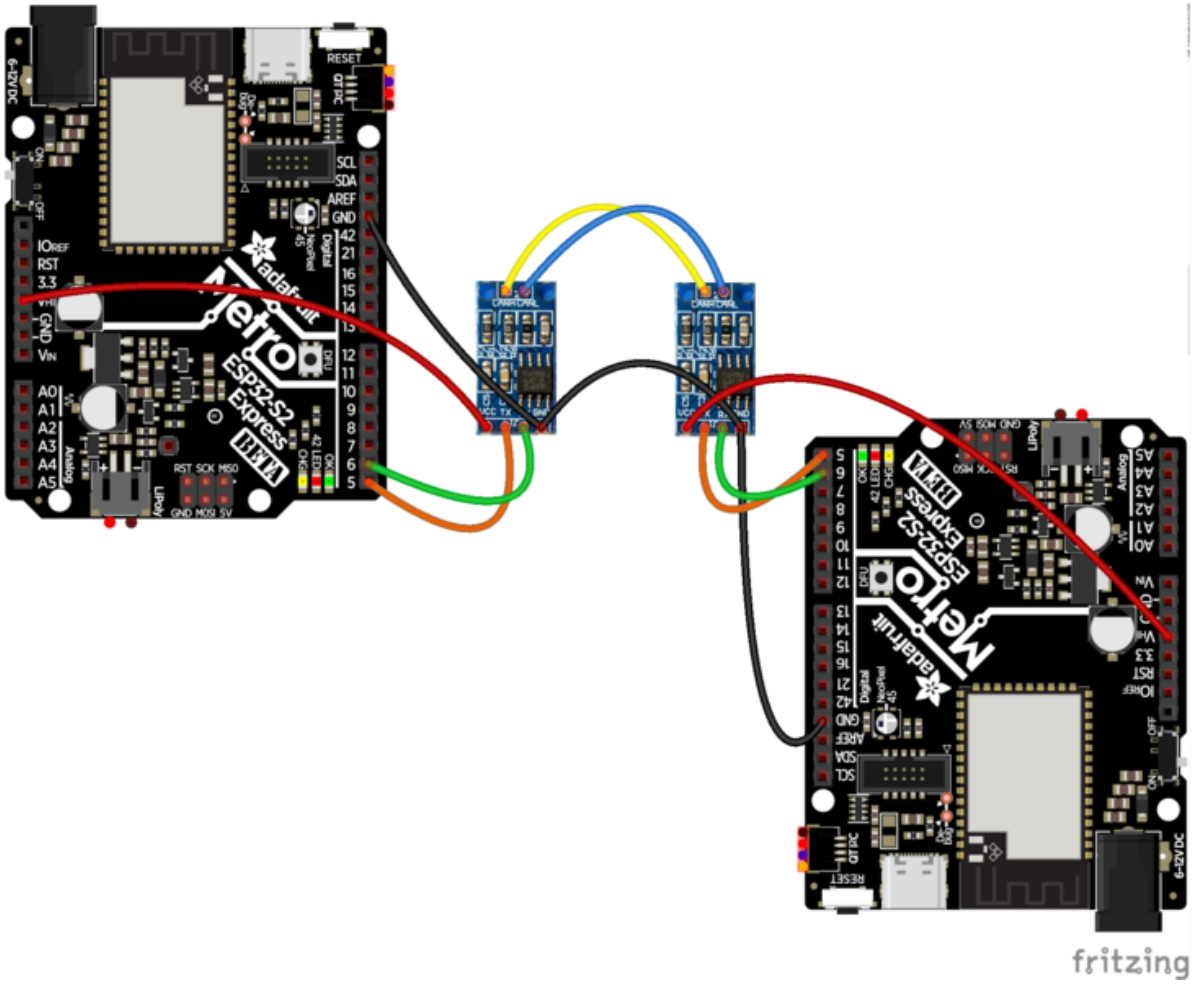
CAN Bus with CircuitPython: Using the canio module

Created by Jeff Epler

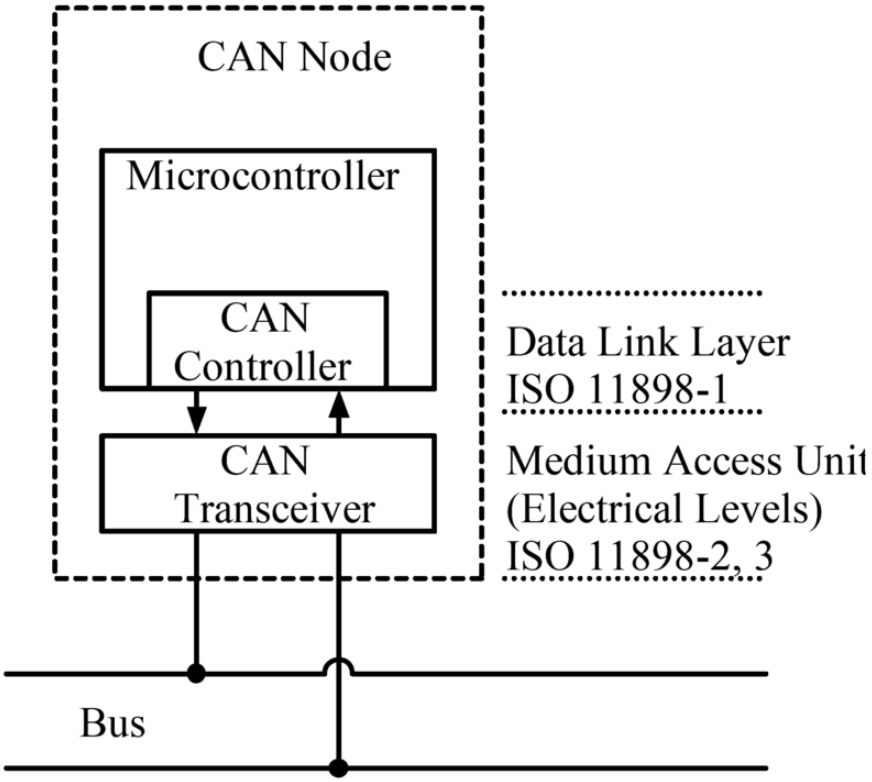


https://learn.adafruit.com/using-canio-circuitpython

Last updated on 2021-12-16 10:23:50 AM EST

Table of Contents

Overview



In this guide you'll learn how to use CircuitPython's `canio` module to send and receive data between two supported boards, such as the Feather M4 CAN.

Are you new to using CircuitPython? No worries, there is a full getting started guide here (https://adafru.it/cpy-welcome).

Adafruit suggests using the Mu editor to edit your code and have an interactive REPL in CircuitPython. You can learn about Mu and installation in this tutorial (https://adafru.it/ANO).

To use canio you need CircuitPython 6.0 or newer and a supported board. Check the list of supported modules on the downloads page to make sure canio is available.

CAN basics

According to Wikipedia,

> A Controller Area Network (CAN bus) is a robust vehicle bus (https://adafru.it/OyE) standard designed to allow microcontrollers (https://adafru.it/OyF) and devices to communicate with each other's applications without a host computer (https://adafru.it/Oza). It is a message-based protocol (https://adafru.it/Ozb), designed originally for multiplex (https://adafru.it/Ozc) electrical wiring within automobiles to save on copper, but can also be used in many other contexts.

The Controller Area Network is standardized as ISO 11898.

A CAN bus consists of 2 or more devices hooked together with a pair of wires, called H and L. Generally the devices will also share a common GND as well. We'll show networks with just 2 devices, but you can certainly have more. When you have a larger number of devices, you may have to modify the "bus termination resistors" according to the requirements of the CAN specification.

A CAN packet consists of an ID (a 11 or 29 bit number; 29 bit IDs are "extended IDs") which allows devices to listen only for message IDs they are interested in; a "Remote Transmission Request" (RTR) flag which allows one device to request data from another device; and (if it is not a Remote Transmission Request), a data payload of 0 to 8 bytes.

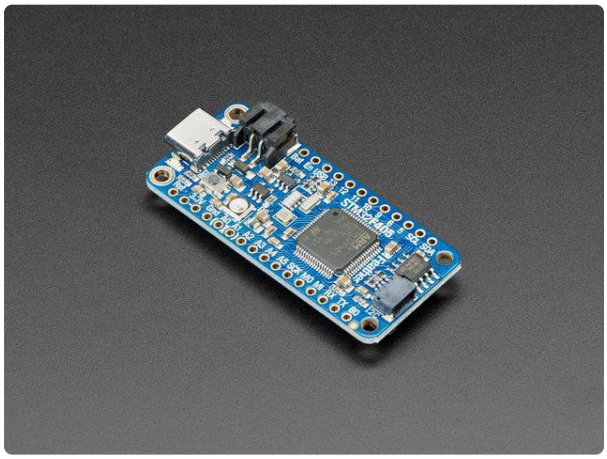
The whole packet is protected against corruption by a CRC (sometimes called a checksum). This means that while packets can sometimes go missing due to garbled transmission, almost all data transmission errors are caught and the invalid data discarded.

Since CAN does not include a way to be sure the intended recipient device has actually received a message, you may find it necessary to implement your own "reliable" transmission method by having the receiving device acknowledge that it has received a packet by sending a packet back. There are other ways of tackling potential lost packets; for instance, if you are a sensor you could just send your sensor data 10 times a second and not care whether just a few packets are lost. The right thing to do depends on your application.

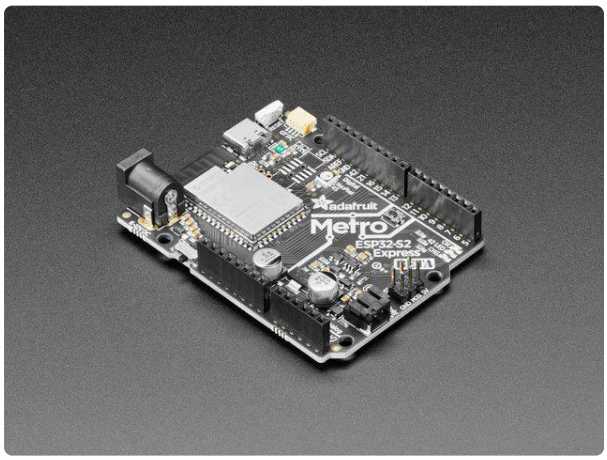
Compatible boards

Multiple Adafruit boards such as Feathers and Metros have a CAN bus peripheral. The Feather M4 CAN is the most convenient, as it also includes a CAN transceiver. Other boards need an external CAN transceiver. Supported boards include:

- Feather M4 CAN Express
- Feather STM32F405 Express (requires an external CAN transceiver)
- Metro ESP32-S2 Express (requires an external CAN transceiver)



Adafruit Feather STM32F405 Express
ST takes flight in this Feather board. The new STM32F405 Feather (video) that we designed runs CircuitPython at a blistering 168MHz –...
https://www.adafruit.com/product/4382



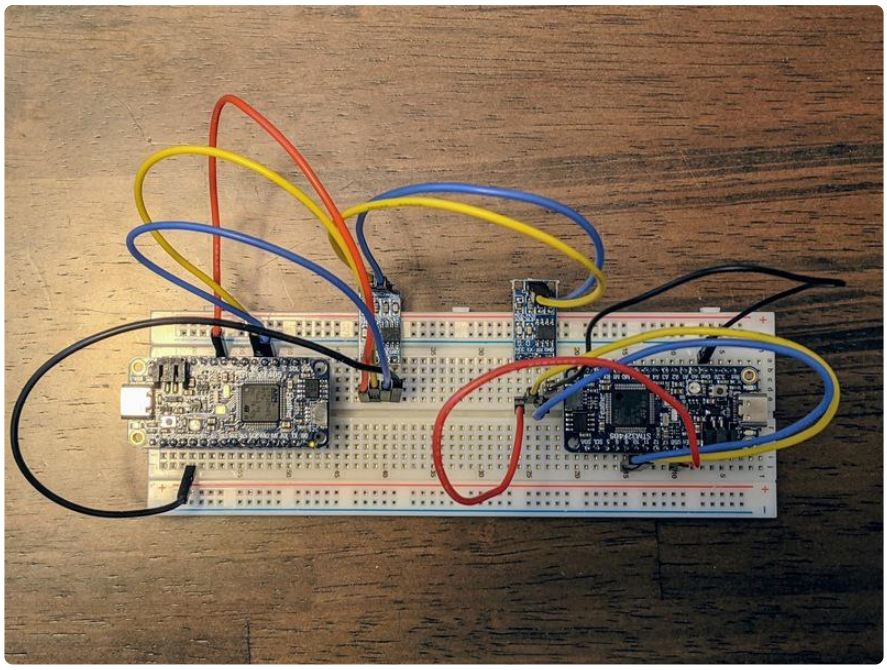
Adafruit Metro ESP32-S2
What's Metro shaped and has an ESP32-S2 WiFi module? What has a STEMMA QT connector for I2C devices, and a Lipoly charger circuit? What has your favorite Espressif WiFi...
https://www.adafruit.com/product/4775

1 x CAN Bus Module Transceiver TJA1050
5V Can Bus Transceiver modules (pack of 5)
https://www.amazon.com/gp/product/B07W4VZ2F2

CircuitPython Docs

CircuitPython Docs (https://adafru.it/OrD)

Wiring



Feather STM32F405 Express & External Transceiver

The Feather STM32F405 has a built in CAN peripheral, but it requires an external transceiver.

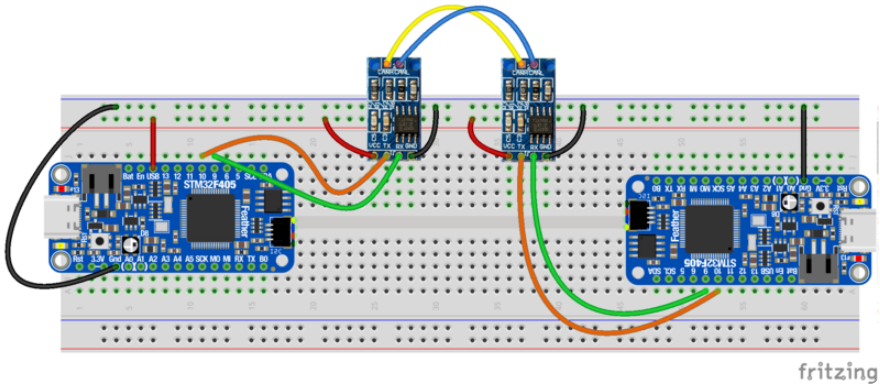
Wiring is reasonably straightforward, but you need to take note of the following:

- Whether the transceiver needs 5V (more common) or 3.3V (less common) on its power input pin. On my transceiver, 5V is required, and comes from the USB pin on the Feather
- Whether the TX and RX pins refer to the microcontroller's point of view or the transceiver point of view. On my transceiver, TX and RX refer to the transceiver's point of view.
- Whether there is an enable pin, and whether to set it `True` or `False`

On an STM32F405 Feather, the pin marked D9 is the CAN TX (data FROM feather INTO transceiver); the pin marked D10 is the CAN RX (data INTO Feather FROM transceiver) pin.

Here's how to wire up two STM32 Feathers:

Feather 1 ↔ Transceiver 1 ↔ Transceiver 2 ↔ Feather 2



After reviewing the CAN breakout board I used, I made the following connections between each Feather and its Transceiver:

- Feather USB to Transceiver VCC
- Feather D9 to Transceiver TX
- Feather D10 to Transceiver RX
- Feather GND to Transceiver GND

Make the following connections between the two transceivers:

- H to H
- L to L

Finally, we need a common GND between the two nodes on the network. If they are not already sharing a GND (for instance, plugged into the same USB hub or USB power bank, or connecting to a GND rail on a breadboard),

- either connect GND from Feather 1 to Feather 2
- or connect GND from Transceiver 1 to Transceiver 2

ESP32S2 Metro & External Transceiver

The ESP32S2 has a built in CAN-compatible peripheral (called TWAI in the documentation from Espressif). You can choose any two pins to act as the RX and TX pins, but when it comes to the sample code you'll need to change `board.CAN_RX` and `board.CAN_TX` to the pins you wired up. I arbitrarily chose `IO05` and `IO06`.

Wiring is reasonably straightforward, but you need to take note of the following:

- Whether the transceiver needs 5V (more common) or 3.3V (less common) on its power input pin. On my transceiver, 5V is required, and comes from the V_{HI} pin on the Metro
- Whether the TX and RX pins refer to the microcontroller's point of view or the transceiver's point of view. On my transceiver, TX and RX refer to the microcontroller's point of view.
- Whether there is an enable pin, and whether to set it `True` or `False`

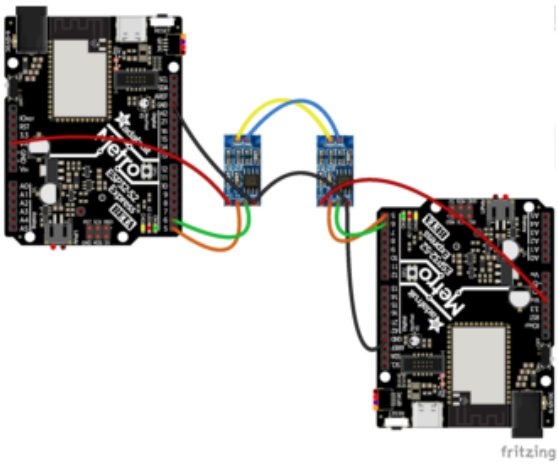
Here's how to wire up two Metro ESP32S2 Express board:

Metro 1 ↔ Transceiver 1 ↔ Transceiver 2 ↔ Metro 2

In the code samples, change the setup line for the CAN object according to the pins you chose, e.g.,:

```
can = canio.CAN(rx=board.IO6, tx=board.IO5, baudrate=250_000, auto_restart=True)
```

After reviewing the CAN breakout board I used, I made the following connections between each Feather and its Transceiver:



- Metro V_{HI} to Transceiver VCC
- Metro IO5 to Transceiver TX
- Metro IO6 to Transceiver RX
- Metro GND to Transceiver GND

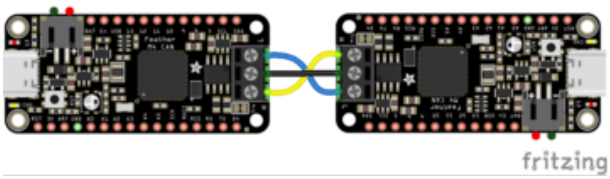
Make the following connections between the two transceivers:

- H to H
- L to L

Finally, we need a common GND between the two nodes on the network. If they are not already sharing a GND (for instance, plugged into the same USB hub or USB power bank, or connecting to a GND rail on a breadboard),

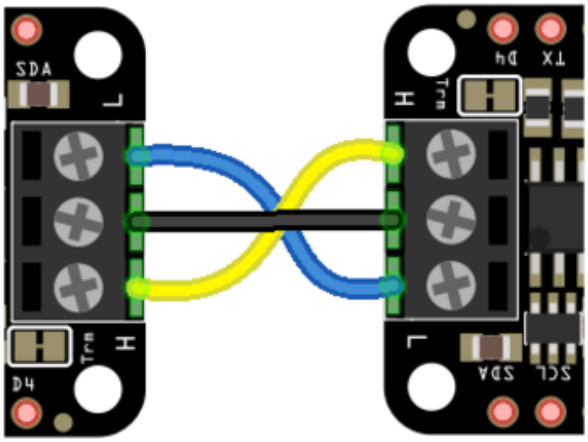
- either connect GND from Metro 1 to Metro 2
- or connect GND from Transceiver 1 to Transceiver 2

Feather M4 CAN



Because the transceiver is included, wiring a CAN bus is simple:

- Connect H to H
- Connect L to L
- Connect GND to GND (other ways of providing a common GND are also acceptable, such as powering both devices from the same computer or USB power bank)



In the case of the Feather M4 CAN, just insert wires into each screw terminal and then tighten the screw.

Make sure that H goes to H and L to L; Otherwise, the devices will not be able to communicate.

Mix and Match

Want to make a network out of different boards? Knock yourself out. Just wire each side as above, then connect the H, L, and GND wires between the nodes in the network. For example, you could wire

Metro ESP32S2 ↔ Transceiver 1 ↔ Transceiver 2 ↔ Feather STM32F405

or

Feather STM32F405 ↔ Transceiver ↔ Feather CAN

In principle, you can put more than two nodes on a network by connecting all the H wires together and all the L wires together. However, you also need to understand and may need to modify the "termination resistance" of the bus—that's beyond the scope of this guide (and indeed your humble author's experience)

Send and Receive

This demo shows how to set up one Feather M4 CAN as a sender and another as a receiver. They will print messages on the serial terminal (REPL) to show what is going on.

Before trying this demo, make sure you have the right version of CircuitPython, that i mport canio succeeds, and that you've wired the two Feathers together as shown on the Wiring page.

First, we'll set up the listening (receiving) node. Put the text below in that device's co de.py and open up the serial terminal. When the program restarts, it will display "No messsage received within timeout" until our second device is up and running.

```
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}")
```

Next, set up the transmitting (sending) node. Put the text below in that device's code.py and open up a second serial terminal. Once both programs are running, you should see the sender and receiver printing the same information.

```
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
```

In this case, the data being transmitted is a packet counter which starts at 0 and counts up 1 for each transmitted packet; and a timestamp which counts up by 1000 each second. These are sent as 4 byte values, for a total of 8 bytes—the maximum for a packet on the CAN bus.

If there are problems affecting the bus (such as a disconnected wire) then various error information will also be displayed. However, after fixing the wiring the devices should automatically recover and begin communicating again.

Typical output from sender:

```
code.py output:
Bus state changed to canio.BusState.ERROR_ACTIVE
Sending message: count=0 now_ms=372429
Sending message: count=1 now_ms=372929
Sending message: count=2 now_ms=373429
Sending message: count=3 now_ms=373929
```

Typical output from receiver:

```
code.py output:
Bus state changed to canio.BusState.ERROR_ACTIVE
received message: count=0 now_ms=372429
received message: count=1 now_ms=372929
received message: count=2 now_ms=373429
received message: count=3 now_ms=373929
```

Reliable Transmission

This demo shows one of the possible ways you can verify that the intended node has received a message.

Before trying this demo, make sure you have the right version of CircuitPython, that i mport canio succeeds, and that you've wired the two Feathers together as shown on the Wiring page.

First, we'll set up the listening (receiving) node. Put the text below in that device's co de.py and open up the serial terminal. When the program restarts, it will display "No messsage received within timeout" until our second device is up and running.

```
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: id={message.id:x} count={count} now_ms={now_ms}")
    if gap != 1:
        print(f"gap: {gap}")

    print("Sending ACK")
    can.send(canio.Message(id=0x409, data=struct.pack("<I", count)))
```

Next, set up the transmitting (sending) node. Put the text below in that device's code.py and open up a second serial terminal. Once both programs are running, you should see the sender and receiver printing the same information.

```
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)
listener = can.listen(matches=[canio.Match(0x409)], timeout=.1)

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))
    while True:
        can.send(message)

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

        data = message_in.data
        if len(data) != 4:
            print(f"Unusual message length {len(data)}")
            continue

        ack_count = struct.unpack("<I", data)[0]
        if ack_count == count:
            print("Received ACK")
            break
        print(f"Received incorrect ACK: {ack_count} should be {count}")

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

In this case, the data being transmitted is a packet counter which starts at 0 and counts up 1 for each transmitted packet; and a timestamp which counts up by 1000 each second. These are sent as 4 byte values, for a total of 8 bytes—the maximum for a packet on the CAN bus.

When the receiver receives an acknowledgement packet (an ACK), it sends back one of its own, containing just the 4 byte count value. A different message ID (0x409) is also used.

The sender waits for the ACK. If it is not received shortly, then it re-sends the original message and waits for the ACK again. In this way, the sender knows the receiver has received each message.

Typical output on sending node:

```
code.py output:
Bus state changed to canio.BusState.ERROR_ACTIVE
Sending message: count=0 now_ms=123231
Received ACK
Sending message: count=1 now_ms=123733
Received ACK
Sending message: count=2 now_ms=124235
Received ACK
```

Typical output on receiving node:

```
code.py output:
Bus state changed to canio.BusState.ERROR_ACTIVE
No messsage received within timeout
received message: id=408 count=0 now_ms=123231
Sending ACK
received message: id=408 count=1 now_ms=123733
Sending ACK
received message: id=408 count=2 now_ms=124235
Sending ACK
```

Code Walkthrough

The various programs share a lot of code, so let's look at what the building blocks are.

Begin by importing the modules that are needed by our code:

```
import struct
import time

import board
import canio
import digitalio
```

Create the necessary digital pin settings needed to enable the CAN Transceiver chip:

```
# 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)
```

Create the CAN bus object. Note that all devices on the same bus need to agree on the baudrate!

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

Construct a listener object. This listener will ONLY receive messages sent to the ID `0x408`. If no `matches=` was specified, it would receive all messages. The `timeout=` governs how long the listener will wait for a message. A `Match` object can also specify an optional mask to allow a range of related IDs to be received—see the full documentation for more details.

```
listener = can.listen(matches=[canio.Match(0x408)], timeout=.1)
```

Now we're ready for the main loop of our program:

```
while True:
    ...
```

The CAN object's state monitors the health of the bus. The confusingly-named `ERROR_ACTIVE` state actually indicates that all is well. A node that is `ERROR_PASSIVE` will not transmit messages, and one that is `BUS_OFF` will neither transmit messages nor acknowledge messages from other nodes. Because we specified `auto_restart=True` when we created our CAN object, our node will automatically restart itself a short time after entering the `BUS_OFF` state.

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

Create and send a message. In this case, we use the `struct` module to pack our integer data into a sequence of 8 bytes. Messages can range from 0 to 8 bytes of data.

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

Receive a message. One of several things can happen, and we need to deal with them:

- If no message is received before the timeout, message will be None
- If we were listening for more than one message ID, we would want to look at `message.id` and make decisions based on it.
- A message could come in, but not have the expected structure. Here, if the message is not the expected 8 bytes long, we ignore it

- If the message has the expected length, we can take the individual pieces of data out using `struct.unpack`, and act on them.

```
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("&lt;II", data)
print(f"received message: count={count} now_ms={now_ms}")
```