## Getting started with MicroPython and CBUS

There are many ways to get started, depending on your existing programming skills and what you hope to achieve.

For beginners – and even experienced programmer who are new to Python – I would suggest you first get comfortable with the Python language itself, its structure and syntax. You can do this using the ‘big’ Python interpreter that is almost certainly already installed on your PC. If it isn’t, you can download it from <https://www.python.org/downloads/>. Make sure you get version 3 as version 2 is now deprecated.

The next step is to get yourself a Pico (or another supported board) and follow one of the many tutorials on how to get started with it. This will almost certainly involve downloading a simple IDE called Thonny and using this to install MicroPython on your Pico. Then get comfortable with this environment, writing simple programs, uploading them to the Pico and running them.

Once you’re ready to experiment with CBUS, simply download the .py files (and all sub-folders) from my GitHub repo (see links below) and upload them to your Pico. Upload everything even if you don’t have a need to use them all just yet. There’s plenty of storage space and they don’t consume memory until explicitly imported into a program

Depending on your CAN bus hardware and its pin configuration, you may need to edit a few lines. If you are using one of my CBUS shield designs, the example code’s defaults are already correct, and no changes are required. Otherwise, you may need to change the SPI bus pins connected to the MCP2515, and the pins for the CBUS switch and LEDs, if fitted.

I recommend you start with the file module\_example\_configurable.py.

You can also change the module’s configuration setting, including its name, module ID, and numbers of events, event variables (EVs) and node variables (NVs). Do this before setting FLiM mode as changing things later will require a reset of the config.

Upload the edited program to your Pico.

Then, just type at the >>> prompt:

>>> import module\_example\_configurable

Or whatever you have renamed it to.

After a few seconds, the program will load and run, and you will be presented with a new prompt. If you don’t see “mcp2515 device is present”, double-check your pin numbers.

At this point CBUS message processing is running ‘in the background’ but you can type Python code and commands at the interactive --> (REPL) prompt, e.g.

--> 2+2

4

--> print('hello world!')

hello world!

If you misspent your teenage Saturday mornings in the local Curry’s typing infinitely looping programs into all their microcomputers, the equivalent today is:

--> while True: print(‘hello’)

(You will probably need to power cycle the board to get out of that!)

The --> prompt is provided by a ‘mini’ REPL within the application. If you inadvertently start a long running command, you can use control-C to interrupt it.

To stop the application and return to the main >>> prompt, just type control-D at the --> prompt. Further presses of control-D will reset the board and return it to a known state. You will also need to return to the >>> prompt before uploading files as the mini-REPL doesn’t have this capability.

I’ll presume you are familiar with the basic concepts of CBUS and have a copy of the CBUS Developers’ Guide to hand.

If your module is connected to a CBUS along with other CBUS modules, just send a message or event from one of them, or from FCU or JMRI. This will be displayed, as if out of nowhere, because CBUS processing is happening in a concurrent task running ‘in the background’, e.g.

199175299 -- received message handler: [5ff] [5] [ 90 00 16 00 32 ]

The number at the beginning of each line is the number of milliseconds that have elapsed since the board was powered on and is useful for timing and performance testing.

You can now send a CBUS event, by creating a message object and then sending it, e.g., to send an ‘on’ event with node number 22 and event number 24:

--> import canmessage

-->

--> evt = canmessage.cbusevent(mod.cbus, 1, 22, 24)

-->

--> evt.send()

199379683 -- sent message handler: [585] [5] [ 90 00 16 00 18 ]

-->

­­­

The library code updates the message’s CAN ID field, including the default message priority, and calculates the correct CBUS opcode for you, depending on whether it’s a short or long event, and whether there are additional data bytes.

At this point, with everything working well, you can set your device into FLiM mode and introduce it to FCU (or JMRI). The time-honoured way is to hold down the CBUS switch for 6+ seconds until the yellow LED flashes and then release it. But, as we have a handy command line, we can instead just type:

--> mod.cbus.init\_flim()

Then, provide FCU with the desired node number for your module, and you’ll see the exchange of CBUS messages fly by as FCU interrogates the module.

Once you have completed development, you can get the Pico to execute your application code automatically at power on. To do this, create and upload a file named main.py with the single line: import my\_module, or whatever you have named your program. However, it’s best not to do this until you are certain your program can be interrupted from the command line. If you do ‘brick’ your Pico, there are ways to ‘nuke’ it and start over afresh. (Just do a Google search with these two magic words).

(Note: you’ll have noticed that it takes a few seconds for the Python code to load, compile and run. This may or may not have an impact on your layout’s start-of-day processing. It’s still significantly faster than a PC or Pi).

# A word about CBUS events

CBUS events are a specific kind of CBUS message. Most events are accessory events, indicating that something of interest has happened in the outside world, e.g., a switch has been operated, a loco has been detected, etc. They are sent by producer modules and are received by all other modules on the bus. Whether a specific consumer module does anything with this event depends on whether it has been configured to do so.

All events are messages but not all messages are events ☺. In our code, a cbusevent object is a sub-class of the canmessage class.

There are a number of ways to represent CBUS messages as Python variables. In order to send a CBUS message (or event) we need to create a full object of the class canmessage. e.g.,

--> import canmessage

--> import cbusdefs

# as a generic CBUS message

--> msg = canmessage.canmessage(0, 5, (cbusdefs.OPC\_ACON, 0, 22, 0, 23))

--> mod.cbus.send\_cbus\_message(msg)

# as a CBUS event

--> evt = canmessage.cbusevent(mod.cbus, 1, 22, 23)

--> evt.send()

However, these are fairly heavyweight objects, and we’d like to conserve memory wherever possible.

The shortest description of a CBUS event is (i) its polarity (on, off or neither), (ii) its node number and (iii) its event number, because the rest can be filled in by the library code when we come to send it. This can be represented by a Python tuple, which is a simple, read-only list type. For example, the same event could be written as (1, 22, 23). As and when we need to send this event, we can construct a full cbusevent object from it, e.g.,

--> t = (1, 22, 23)

--> evt = canmessage.event\_from\_tuple(mod.cbus, t)

# or even

--> evt = canmessage.event\_from\_tuple(mod.cbus, (1, 22, 23))

--> evt.send()

We can even create an event without a fixed polarity (polarity = -1) and then use its send\_on() or send\_off() methods. This saves having to carry around two tuples to represent the same event in two different states.

--> evt = canmessage.event\_from\_tuple(mod.cbus, (-1, 22, 23))

--> evt.send\_on()

You’ll see this approach used throughout the CBUS library code. Some methods take multiple events as arguments, represented by nested ‘tuples of tuples’, e.g.

((0,22,23),(1,22,23)) or even ((0,22,23),(0,22,24),(0,22,25)), etc.

# Message filters

A useful capability of the canmessage class is that it can determine whether a particular message matches a specific query. This is used by the pubsub and history classes as a filter to determine whether a message of interest has arrived. There are a number of pre-defined queries, or you can provide a user-defined function if none of the provided ones do what you need. The main pre-defined queries are:

QUERY\_TUPLES matches against a list of tuples, e.g. ((0,22,23),(1,22,23))  
QUERY\_OPCODES matches against a list of opcodes, e.g. (90, 91)  
QUERY\_CANID matches the message’s CAN ID against the number provided

QUERY\_NN matches the message’s node number against the number provided

QUERY\_DN matches the message’s event number against the number provided  
QUERY\_RTR matches only messages with the RTR bit set  
QUERY\_EXT matches only extended CAN messages  
QUERY\_ALL\_EVENTS matches only CBUS accessory event messages  
QUERY\_LONG\_MESSAGES matches only CBUS long messages  
QUERY\_UDF matches using a user-defined function  
QUERY\_ALL matches all messages  
QUERY\_NONE matches no messages

The code to determine whether a message matches its filter is something like this:

--> msg = canmessage.canmessage(0, 5, (144, 0, 22, 0, 23))

--> if msg.matches(QUERY\_TUPLES, ((0,22,23),(1,22,23))): print(‘match’)

match

# Adding functionality to your module

Module functionality can generally be divided into two parts:

1. Consumer modules receive CBUS events and act upon them
2. Producer modules react to events in the outside world and send CBUS events
3. Combi modules do both (that’s three parts)

## Consumers

There are a couple of ways for a consumer module to capture received events and act upon them. Similar to the approach used in my Arduino libraries, you can use the module object’s received\_message\_handler() and event\_handler() methods. The former receives all messages (by default), the latter only previously taught events.

Default implementations are provided by the base cbusmodule class (in cbusmodule.py) but you can override these and provide alternative implementations in your application class. (You can also override the default sent\_message\_handler() method if you don’t like the default behaviour). For example, you might write an event handler like:

def event\_handler(self, msg, idx: int) -> None:  
 self.logger.log(f'-- event handler: idx = {idx}: {msg}')  
 ev1 = self.cbus.config.read\_event\_ev(idx, 1)  
 self.logger.log(f'first EV = {ev1}')

(NB: don’t edit the library code. That’s not the object-oriented way of doing things and your changes will be overwritten by any library updates. Implement the same method in your application class and it will override the default implementation inherited from the base class).

A more advanced approach is to create separate concurrent tasks which use a sensor, pubsub or history object to wait for specific messages of interest and act upon them. This approach is useful if your event handler method would get unmanageably messy and long-winded, and you’d like to separate things into smaller chunks of code. The example program module\_asyncio.py shows examples of each, and the subject is addressed in more detail below.

## Producers

What a producer module does is of course specific to that module’s purpose. In this simple example, we imagine a few switches connected the Pico’s pins which it reads and then sends events based on their changed state. You can create code to do this in the module class’s module\_main\_loop\_coro() method. This is analogous to the loop() function in an Arduino sketch, except that it must yield to the scheduler each time around the loop, to allow other tasks some time to run. The example code does this by having it sleep for a few milliseconds.

For instance, you could update the example program, as follows:

*# \*\*\* user module application task - like Arduino loop()*async def module\_main\_loop\_coro(self) -> None:  
 self.logger.log('main loop coroutine start')  
 current\_pin\_state = Pin(10).value()  
 evt = canmessage.cbusevent(mod.cbus, -1, 22, 23)  
  
 while True:  
 await asyncio.sleep\_ms(25)  
 new\_pin\_state = Pin(10).value()  
 if new\_pin\_state != current\_pin\_state:  
 current\_pin\_state = new\_pin\_state  
 evt.polarity = current\_pin\_state  
 evt.send()

If you are using your module’s event table to store taught producer events, there are a couple of useful methods for looking these up, either by a single EV or multiples thereof:

def find\_event\_by\_ev(self, evnum: int, evval: int) -> int:

def find\_event\_by\_evs(self, query: tuple[tuple[int, int], ...]) -> int:

Both return the matching event table index, or -1 if not found. You can then use the canmessage.event\_from\_table() method to create an event to send:

--> mod.cbus.config.print\_events()

...

21 = 00 16 00 1b 02 04 06 08

22 = 00 16 00 1c 03 06 09 0c

23 = 00 16 00 18 02 04 08 10

...

--> idx = mod.cbus.config.find\_event\_by\_ev(1, 3)

--> idx

22

--> evt = canmessage.event\_from\_table(mod.cbus, idx)

--> print(evt)

[5] [5] [ 00 00 16 00 1c ]

To convert a canmessage or cbusevent object to a tuple, simply write:

--> t = tuple(evt)

--> t

(1, 22, 28)

-->

(Note how we use the ‘self’ object when adding code to the program’s class, but the ‘mod’ object when typing at the command line. This will make more sense as you become comfortable with object-oriented programming in Python).

(Note that printing an event will by default show its values in hex, whilst a tuple will display in decimal. It’s like that just to confuse you! Take a look at the print() and \_\_str\_\_() methods of the canmessage class in canmessage.py for options).

(The foregoing may prompt the question of why we would use the event table at all, now that we can represent a module’s configuration in code. I’ll leave the answer up to you).

(You may be wondering, as Python is a weakly-typed language, why method and function definitions show the expected argument and return types. This is known as ‘typing’ in Python and whilst the interpreter completely ignores it, this information is useful to some IDEs for code completion and error checking. It isn’t mandatory but it’s a good habit to get into).

# Using sensors and pubsub

A pubsub (publish and subscribe) object can be imagined as a subscription to a subset of messages that we are interested in. If a matching message arrives on the bus, the CBUS class publishes it to us. However, the vast majority of messages won’t be of interest, and we won’t be troubled by these. (In the real world, a company may publish numerous monthly magazines but you or I only subscribe to one or two). In software architecture terminology, this is an implementation of the observer pattern.

For example, a feedback sensor object may be interested in the pair of on and off events produced by an occupancy detector or turnout feedback module, and update its internal state accordingly. You would create a sensor which will then update itself asynchronously ‘in the background’.

If you are using the traditional superloop approach, simply create the sensor in your module’s initialise() method and then test its state each time around the main loop. e.g.,

--> sensor1 = cbusobjects.binary\_sensor('sn1', mod.cbus, ((0,22,23),(1,22,23)), None)

Now, if either of those events is received, the sensor will update its state and print a message:

3719890 binary sensor sn1, new state = 1

We can test its state explicitly at any time:

--> if sensor1.state == 1: print('sensor is active')

However, this is cumbersome, and we may wish to create an independent task that waits for changes to the sensor’s state and acts accordingly. We can create this task as method in our main application class, or in a separate Python module:

async def sensor\_test\_coro(self) -> None:  
 self.sensor1 = cbusobjects.binary\_sensor('sensor1', mod.cbus, ((0, 22, 23), (1, 22, 23)), None)  
 while True:  
 await self.sensor1.wait()  
 self.logger.log(f'sensor\_test\_coro: {self.sensor1.name} changed state to {self.sensor1.state} = {cbusobjects.sensor\_states.get(self.sensor1.state)}')

The ‘*async’* keyword introduces a function or method that is to be run as a separate concurrent task.

The ‘*await’* keyword means that the task blocks until either of the messages of interest are received.

We start this task in our application’s initialise() method, and it then happily runs in the background, e.g.,

\_ = asyncio.create\_task(self.sensor\_test\_coro())

We can also use the pubsub class directly if we our requirements are more complex that a simple sensor. This can take advantage of the message filtering described earlier.

# Using CBUS message history

Whereas most CBUS message processing logic, including sensors and pubsub, considers a single message at a time, the message history class enables us to create complex application logic that considers sequences of multiple received messages. This is similar in concept to Ian Hogg’s CANCOMPUTE module. The difference here is that we can have multiple concurrent tasks each with their own history, filters and queries (subject to memory and processing time constraints), and we express the configuration as program code.

A history is just a list of recently received messages together with their time of arrival. The history will have a limited lifespan (say, 10 seconds) and older messages are automatically removed from the list. Thus, we can imagine a sliding window in time, representing the last *n* seconds of CBUS activity

The messages that are published to the list can be controlled with one of the message filters described earlier.

A task waits until a message is added to its history, at which point it wakes up and can execute queries against it. A waiting task consumes no processor cycles.

This example shows a task waiting for two events to arrive in the sequence given, within the last 3 seconds, and over a window of 2 seconds. It then calculates the time difference between the two events:

async def history\_test\_coro(self) -> None:  
 events = ((0, 22, 23), (1, 22, 23))  
 hist = cbushistory.cbushistory(self.cbus, max\_size=1024, time\_to\_live=5\_000, query\_type=canmessage.QUERY\_TUPLES, query=events)  
 while True:  
 await hist.wait()  
 if hist.sequence\_received(events, order=cbushistory.ORDER\_GIVEN, within=3\_000, window=2\_000, which=cbushistory.WHICH\_LATEST):  
 diff = hist.time\_diff(events)  
 self.logger.log(f'history\_test\_coro: sequence {events} found, time diff = {diff}')  
 else:  
 pass

A more complex use-case is the NX r(eNtry/eXit) route class. This requires two pushbuttons on a mimic panel, placed at the start and end of the route to be set. On receipt of one of the switch events, the route object emits a CBUS event to illuminate the switch. On receipt of the second switch event, a subsequent event is produced to light the route, and then the route itself is acquired and set. This takes less than 20 lines of code to setup and execute. The route class can also monitor multiple occupancy sensors and refuse to set the route if any sensors are active.

An interlocking task might prevent a signal being cleared if the turnout it is protecting is incorrectly set. There are a couple of ways to achieve this. Either we could monitor the turnout’s control or feedback events using a sensor object, or with a history object we could ensure that multiple turnout set events were received before the signal clear event. In either case, we would only emit the event to clear the signal if it safe to do so.