CBUS MicroPython Library

Introduction

This document describes a set of MicroPython language modules that enables the creation of CBUS layout control modules on supported hardware. The design philosophy follows that of my Arduino CBUS library: to provide everything required to implement a generic CBUS module, leaving the designer to add the functionality specific to their module.

The initial development has been done on a Raspberry Pi Pico microcontroller board, but the code is in principle portable to any hardware supported by the MicroPython project, e.g., ESP32, ESP8266, etc. The only CAN controller supported so far is the MCP2515, although it should be possible to support other devices if MicroPython support becomes available, e.g., the ESP32's on-chip CAN controller peripheral.

There is a further module, the *pyboard*, which is sold by the developer of MicroPython. This uses an STM32 microcontroller with on-chip CAN peripheral and is fully supported by MicroPython. It is more expensive than other boards though and appears to be out-of-stock at the moment (Dec 2022).

I have also made available PCB designs ('shields') to attach the required hardware to a Pico.

MicroPython will never be available for smaller devices like the Arduino Uno or Nano (or 8-bit PICs) as they simply don't have enough memory or CPU power.

Why Python?

Python (and its MicroPython derivative) is an interpreted, rather than compiled, language. This means it is slower and requires more memory than compiled C and C++ and requires a more capable microcontroller to run. I would characterise it as 10x slower and requiring 10x the memory. However, it is much more productive from the perspective of the developer, perhaps 10x so, and using modern microcontrollers makes up for many of the performance overheads.

If your use-case requires microsecond accuracy and responsiveness, then you should probably stay with C and C++, or assembler. For anything else, Python is more than sufficient.

Python is a very popular language and frequently appears in global 'Top 3' lists from various sources. It is also an easier language for beginners to get started with and there are many tutorials available.

Because it's interpreted, you can experiment at the command line (REPL) rather than having to go through the traditional edit/compile/upload/debug cycle of compiled languages. You can also inspect variables whilst the code is running and even execute arbitrary pieces of code from the REPL.

Who is it for?

Python is often recommended for beginners to programming who might otherwise struggle getting to grips with compiled languages.

It is also a very productive language, enabling rapid prototyping and creation of useful functionality. It also includes multi-tasking functionality.

How to get started

The easiest way is to buy a Raspberry Pi Pico (or a WiFi-enabled Pico W), plug it into your PC and work through the tutorials on the Raspberry PI website. These will use the Thonny IDE, which is simple but sufficient for the time being.

https://www.raspberrypi.com/documentation/microcontrollers/micropython.html

I recommend the Pico as it is cheap, has lots of memory available to MicroPython and is easily available from UK suppliers. Other devices (ESP32, ESP8266) have less memory but are perfectly fine for initial experimentation if you already have one to hand.

Once you're happy with the basics, then you can order some PCBs and assemble a CAN bus adapter, and experiment with the CBUS libraries discussed in this document.

In addition to the Python basics, it is also worthwhile getting to grips with Python's *async/await* functionality. This provides a simple multi-tasking facility and is used extensively by the CBUS module code.

There may some confusion in terminology as Python uses the term 'module' to refer to an installable code library. As you delve deeper, you'll also find a conflict between the CBUS usage of 'event' and the Python usage. I'll try to avoid confusion wherever possible.

Using the CBUS libraries

The code is available from my GitHub repo: https://github.com/obdevel/CBUS-MicroPython-RP-Pico

As with my Arduino C/C++ libraries, I have included a basic example that has everything needed to create a fully functional, generic CBUS module. Using your IDE of choice, simply upload the various *.py* files to your Pico.

Type *Control-D* a couple of times to make sure we're starting afresh. This will produce the following:

```
MicroPython v1.19.1 on 2022-06-18; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>>
```

At the MicroPython REPL '>>>' prompt, type:

```
import module_example
```

The code will load, compile, and execute and, after a few seconds, present you with another prompt:

```
MicroPython v1.19.1 on 2022-06-18; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> import module_example
  79795381 mcp2515 device is present
  79795392 module: name = <b'PYCO   '>, mode = 1, can id = 5, node number = 77
  79795412 free memory = 123040 bytes
  79795413
  79795475 irq handler is waiting for interrupts
  79795478 run start
  79795480 asyncio is now running the module main loop and co-routines
  79795519 blink_led_coro start
  79795521 main loop coroutine start
Starting asyncio REPL...
-->
```

You'll notice that the on-board LED is blinking once a second. This serves two purposes: firstly, as an example of multi-tasking and secondly, to prove that the device is still alive and the module code is running.

You can now enter Python commands at the prompt whilst CBUS message processing is running 'in the background'.

This can be as simple as:

```
--> 2+2
4
-->
```

If you send a CBUS event from another module, or FCU/JMRI, you'll see it displayed, confirming that CBUS message processing is indeed happening in the background, e.g.

```
-->   80379736 -- user frame handler:
  80379737 [5ff] [5] [ 90 00 16 00 5f ]
```

(This output is created by the default message handling method in the *cbusmodule* class in *cbusmodule.py*, from which our application class derives. The number at the beginning of each line of output is the number of milliseconds since the device was started. This is the default behaviour of the *logging* class in *logging.py* and is useful for performance analysis).

From a ‘clean’ installation, the module will have a blank configuration and be in SLiM mode.

You can now introduce the module to FCU/JMRI in the time-honoured way. Hold down the CBUS switch for 6+ seconds and release. FCU/JMRI will prompt you for a node number and then proceed to read in the module’s configuration.

However, as we have an interactive command line, we can also use code rather than a physical switch to do this:

```
--> mod.cbus.init_flim()
  81699825 cbus: sent msg = [585] [3] [ 50 00 4d ]
```

You’ll now see the CBUS messages being sent between FCU/JMRI and the module.

How many messages?

```
--> mod.cbus.received_messages
85
--> mod.cbus.sent_messages
96
```

This shows that we can inspect the current value of any variable in any class, and also execute any method in any class, just by typing at the REPL.

The precise syntax requires a knowledge of the library’s object model, but it’s basically saying that our main module object (*mod*) has a CBUS object (*cbus*), which in turn has methods and variables we can access.

Let’s send a CBUS event. Firstly, we have to construct a new event object, by importing the required Python module and calling the class constructor:

```
--> import canmessage
-->
--> evt = canmessage.cbusevent(mod.cbus, 0, 22, 23)
-->
--> print(evt)
[585] [5] [ 00 00 16 00 17 ]
-->
```

Now we can send it (in this case, as an ‘on’ event):

```
--> evt.send_on()
  83067943 cbus: sent msg = [585] [5] [ 90 00 16 00 17 ]
-->
```

It populates the CANID field and the appropriate CBUS opcode for you.

The module class has a *module_main_loop_coro* method which is rather like the Arduino *loop*() function. You can put your module's application code there.

(In order to have your application start automatically when the device is powered on, you'll need to create a *main.py* file, with a single line to import the initial Python module. This is rather like *autoexec.bat* from the old MS-DOS days. However, it's best not to do this until you're certain your code works correctly, as it is possible to lock yourself out of the device if your code is somehow not interruptible from the command line. There are ways out of this though – known as 'unbricking' the device).

Using other MicroPython libraries

Documentation for MicroPython can be found at the official site: https://docs.micropython.org/en/latest/. This includes general documentation and that specific to each supported hardware platform.

Rather like the Arduino 'ecosystem', there is a large number of code libraries available, either built-in to the MicroPython interpreter or from 3rd parties. These cover the vast majority of things we might like a CBUS module to do, and external things we might want to control, such as servos, displays, etc.

Useful sources include:

- micropython-lib: https://github.com/micropython/micropython-lib
- Awesome MicroPython: https://awesome-micropython.com

In each case, you need to upload the library's files to your Pico, and then add code to call the various classes, methods, and variables.

For example, the Pico's pins are controlled using the *Pin* class in the built-in *machine* module. For example (from the official docs):

```
from machine import Pin

# create an output pin on pin #0
p0 = Pin(0, Pin.OUT)

# set the value low then high
p0.value(0)
p0.value(1)

# create an input pin on pin #2, with a pull up resistor
p2 = Pin(2, Pin.IN, Pin.PULL_UP)

# read and print the pin value
print(p2.value())
```

Whilst many existing Python libraries have been made available for MicroPython, the limitations of the platform mean that some cannot be. Others may be available but with reduced functionality.

A better IDE

The Thonny IDE is great for beginners, but you may find it a little too simplistic after a while. Another option is PyCharm which is a professional-level Python IDE with a plug-in to support MicroPython. It offers syntax checking, code completion and formatting, and much more. The community edition is free to use.

Moving beyond the basics

In addition to the basic CBUS module functionality, I am working on other code modules to add possibilities that would be difficult, if not impossible, to achieve in C/C++. These build upon Python's rich selection of data structure objects, such as lists, dictionaries, events, and queues.

The first is a CBUS history facility. This allows a module to keep a history of messages recently received and make queries against it. For example, one could ask: "has this group of messages been received, in this order, within the last 5 seconds?". A task would wait for new messages to be received and then issue a query, meaning it doesn't need to constantly poll the message history. The history is automatically truncated to that it only contains messages received within the last *n* seconds. This is similar to the functionality of CANCOMPUTE.

Another is a pub/sub (publish and subscribe) facility. Using this, a task can express an interest in certain types of messages or events, and only wake up when something of interest arrives.

Building on the pub/sub facility, is the idea of sensors (stolen from JMRI). This is an object that maintains state (on/off or a value) depending on messages recently received. A binary sensor would subscribe to a pair of CBUS events, whilst a value sensor would look for specific CBUS data messages and interpret them as desired. The current state (or value) of the sensor can be queried at any time.

Higher-level layout objects such as turnouts and signals are also in development. For example, a simple turnout or semaphore signal is associated with the pair of CBUS events used to operate it (throw/close or set/clear). If the CBUS module that operates it has feedback output (such as microswitch or servo position events), we can get confirmation of its commanded state by associating a sensor object with those feedback events. Colour light signals have no feedback events but, in common with semaphore signals, can be created in sets such that they cascade correctly when one signal's aspect is set.

Building on turnouts, signals, and groups thereof, we can then develop a route object that, once configured, can be acquired, locked, and set with just one line of code, presuming that no other route currently has control of any of the turnout or signal objects. The turnouts and signals are operated in the (prototypically) correct order.