# Using MicroPython to create a CBUS module

## Introduction

It’s nearly five years since I first released a set of code libraries to help people create MERG CBUS modules using the Arduino ecosystem. Before then, the non-expert member would have had to wait for one of the ‘experts’ to get around to designing the CBUS module they required. Furthermore, many early MERG modules were coded in assembler, making derivatives and modifications difficult for the average person.

The intention – my philosophy, if you will – was to provide everything required to create a CBUS FLiM module that would talk to FCU and JMRI, learn events and node variables, send CBUS messages, etc. This provides a platform for the designer to add the specific functionality – or personality – of their module. I’m pleased that a number of members have found these libraries useful for creating their own layout control modules and I am constantly amazed by their inventiveness.

However, technology has moved on apace during the intervening 5 years and we now have access to many more powerful and low-cost microcontrollers, and boards developed around them. So, whilst it has been possible to design CBUS modules using the Arduino system, this has still required learning the C (and C++) languages.

A significant development has been in the area of languages, specifically those that are easier for beginners to get to grips with, because the larger and faster processors do not necessarily require the efficiency of lower-level languages or assembler.

## The Python programming language

The Python programming language has been around for 20+ years and frequently appears in global top 3 lists of the most popular languages. It is seen as an easier environment for beginners, and for a number of reasons. It is an interpreted language and provides an interactive command line – think of a 1980s home computer on steroids - which promotes exploration and experimentation without the usual compile/link/upload/debug cycle that we associate with embedded system development. It also has a rich set of data structures ‘out of the box’, such as classes, lists, dictionaries, etc. For the more experienced programmer, it is a very productive language, enabling the creation of new functionality very quickly. Rarely, if ever, does it crash or hang the hardware; it just produces a (mostly) useful error message.

MicroPython is an implementation of Python for microcontrollers and supports a growing range of familiar hobbyist platforms, including the Raspberry Pi Pico, ESP8266 and ESP32, and some other Arm-based processors. It includes the vast majority of ‘big’ Python’s functionality, omitting only those elements not relevant or achievable on a microcontroller, but adding functionality for interfacing with the outside world.

Being an interpreted language, it runs slower and requires more memory than compiled languages, up to 10x in some cases, but it is in my opinion at least 10x more productive. Rather like the Arduino ecosystem, there is in-built and 3rd party support for the common things we would like to communicate with, including displays, servos, sensors, etc and anything you can attach using the I2C and SPI bus protocols, including CAN controller chips!

The are many, many learning resources available for Python (and MicroPython), including books, tutorials, videos, blogs, etc, etc.

Performance is more than adequate for all but the most time-sensitive of applications, for which C, C++ and assembler still exist as options.

(I should point out that MicroPython will never be available for the smaller microcontrollers such as 8-bit PICs and AVR-based Arduino modules. They are just not powerful enough.)

## MicroPython for CBUS

I have recently created a set of MicroPython modules to achieve a similar aim. Initially just a direct ‘port’ of my C++ code, the productivity of the Python environment has enabled me to add some new things that members may find interesting and useful. (Rather confusingly, Python uses the term ’module’ to describe a separate file or files of code that can be used to add to its built-in functionality).

At the basic level, the modules provide similar functionality to my Arduino libraries, and I have provided examples that represent a ‘blank’ module to which the user can add their specific functionality. Depending on your hardware, you may just need to change the pin numbers for the CAN transceiver chip and the CBUS switch and LEDs. (I also have a couple of CBUS shield designs; see later).

Using these code modules is even easier than Arduino, as all that is required is to copy the source (.py) files to the board of choice. Whilst there are a number of free IDEs available to make life easier, everything can be done with a text editor, a terminal app, and a couple of command line programs.

For those used to developing with my Arduino libraries, the next steps will be familiar. You can register functions to be called when CBUS messages and previously taught events are received, and then process those messages according to your program’s logic. However, there are possibly better ways to skin this particular cat as I shall explain shortly.

## Choosing your hardware

As I mentioned previously, MicroPython supports a growing range of microcontroller platforms, including some that members may already be familiar with. Code written for one platform will be mostly portable to others, and virtually 100% with a few minor changes. There is even a version for Unix and Windows if you wish to learn or test things out on your PC.

I have chosen the Raspberry Pi Pico board for my initial development work. It is cheap, fast, has a fairly large memory, and is readily available in quantity from many UK suppliers. It is now also available with Wifi capability (as the Pico W). It does require an external CAN bus device for communicating with CBUS, but these are readily available, or you can easily construct your own for a few pounds. I currently support the ubiquitous MCP2515 CAN controller chip and I have published PCBs for a couple of matching ‘shield’ designs. (\*\*\* insert photos \*\*\*)

As it stands, there is no currently-supported and -available hardware with a built-in CAN controller peripheral. The ESP32 has this peripheral but this is not (yet …) supported by MicroPython. The leader of the MicroPython project sells a board based on an STM32 microcontroller which does have CAN support (the pyboard), but this is comparatively expensive and is currently out of stock (as of January 2023) due to the global semiconductor shortage.

For more ambitious programs, the Pico has a larger available memory than the ESP32 or ESP8266.

(Note that there is also traditional Arduino support for the Pico, and my C/C++ CBUS libraries run well on it).

## Multitasking

MicroPython has a simple multi-tasking capability that enables multiple threads of execution (aka ‘tasks’) to appear to run concurrently. This enables us to move away from the traditional ‘superloop’ approach traditionally used for embedded programming (e.g., the loop() function present in every Arduino sketch), which often leads to pages of spaghetti code. By composing our code in smaller, concurrent blocks, we can make programs easier to reason about, write, read, and debug.

Multitasking in MicroPython is co-operative rather than pre-emptive, meaning that tasks must explicitly yield back to the scheduler, although this tends to happen naturally as they wait for things of interest to happen (interrupts, messages, events, timers, etc). A task that fails to do this will just block everyone else out (just like early Windows programming).

The Pico’s RP2040 Arm processor has dual cores, and this immediately invites speculation on how they might both be used. Python has historically not been able to use multiple cores, even on large systems. Suffice it to say that we can use both cores of the Pico in MicroPython … but it’s complicated.

That said, for simple functionality, you don’t need to jump into the complexity of multi-tasking as you can follow the same application approach as a familiar Arduino sketch, with the equivalent of its straightforward setup and loop functions. My example programs provide a good starting point.

One of the more powerful capabilities is the provision of an interactive command-line (aka ‘REPL’) that can be used whilst CBUS message processing happens ‘in the background’. This enables one to query variables, call library functions, and even write simple Python functions on the fly. This brings a new dimension to debugging and promotes learning and experimentation without the overhead of continual recompilation.

(The Arduino C/C++ core for the Pico includes FreeRTOS for taking advantage of the second core).

## Beyond a simple Arduino module

One of my personal interests is layout automation, which can be blamed on a thriller novel I read as a teenager that featured a massive model railway that ran itself. That idea has stuck with me all these years! I wanted to be able to implement a degree of automation without needing a PC (or single-board computer such as a Raspberry Pi) running JMRI or expensive commercial software such as TrainController.

I have taken advantage of the productivity of Python, and larger memories, to add functionality that didn’t appear in my Arduino libraries.

There are a couple of ‘abstract’ classes that provide the foundation for other, more familiar, layout objects. The first is ‘publish and subscribe’ (aka pubsub). This allows a task to say: “I’m only interested in these one or two CBUS messages. I’m going to sleep now but please wake me up when one of them arrives, at which point I’ll look at it and do something”.

The next class is a CBUS history, which extends the concept of how Ian Hogg’s CANCOMPUTE module works. A task can maintain a list of CBUS messages which is added to as selected messages arrive from the bus. The history has a finite lifetime, so only (say) the last 10 seconds’ worth of messages is kept. The task then sleeps until a message of interest is added to its history, at which point it can execute queries against the current list. For example, it might ask: “has this group of three messages been received, in this order, within the last 5 seconds? If so, I’ll do something interesting”. This is useful where we wish to apply application logic to sequences of more than one message. Any number of tasks can maintain their own individual histories, subject only to memory and processing time constraints.

A sensor is just an object that maintains its state by listening for two CBUS events (using pubsub), which might be produced by whatever hardware is used (e.g., loco spot or block detection, a microswitch connected to an input module, or servo position events from a CANMIO). A sensor may also be created with a further message which it uses to poll the feedback device, in order to establish its initial state. Once created, we can query the sensor’s state at any time. This is a binary sensor (with two distinct states), but we can also create ‘value sensors’, whose state is a value that depends on events produced by, for example, an RFID or RailCom detector. A multi-state sensor can represent the current position of a turntable with multiple exits.

We can now turn to classes that represent various ‘things’ on a layout that we may wish to operate, interrogate, and automate, including turnouts and signals, train detection, routes, clocks, and DCC locos. Some of us are also experimenting with RailCom for additional feedback and reporting.

For example, a turnout (or semaphore signal) is simply an object created with the two CBUS events needed to operate its real-life counterpart. At any point in time, it has a state (e.g., thrown or closed), even if that is initially unknown. We can then associate a sensor object with this turnout to provide positive feedback. Thus, we can operate the turnout and either wait for its state to be updated by the sensor object or query its state at any time later.

Groups of turnout and signal objects can be composed into ‘routes’, enabling us to operate them as a group in prototypical order. It is further possible to ‘lock’ a route and its component objects so that it can’t be changed whilst a train movement is in progress. The route is then released or allowed to time out. An individual object may appear in multiple different routes (e.g., a turnout in a complex station throat) but only one of those routes may be acquired and set at a time.

By adding a couple of pushbutton switches (using a history object) we can create a simple NX (eNtry/eXit) route. Thus, operating the two switches in order (as detected by a history object) will cause the route to be acquired, locked, and set. The NX route object itself can produce its own CBUS events, so we can control the mimic panel lighting correctly according to the prototype, illuminating the switches and route as it is selected. Of course, it should not be possible to set a route if any of its track sections is currently occupied by a train, and this possibility is also catered for.

It is also possible to control DCC locos, either using CBUS messages to a CANCMD or to a DCC++ command station running on an Arduino board.

Bringing all these pieces (or rather, objects) together, we eventually get most of the way to automated (or at least sequenced) train movements. A train can traverse the layout between a predefined origin and destination, using detection sensors as waypoints to determine where it is. As it approaches a signal, it can query that signal’s state and either set the turnout or route in front of itself, or wait for the human signaller to do so. Multiple movements can be in progress at the same – even human operated, so long as they obey the signals!

I have also created some clock classes, so automation sequences can be synchronised with the time of day, be that wall-clock or fast-clock time. A scheduled movement would wait until a specific time of day before departing, and then pause at en-route station stops for the desired period.

The productivity and expressiveness of the Python language and the ability to simply compose complex objects from other objects, means that a user’s program could achieve all of this in a relatively few lines of code, as the examples below show.

\*\*\* insert object diagram here \*\*\*

## Interfacing hardware

The foregoing has assumed a CBUS layout where all interactions happen through the sending and receiving of CBUS messages and events. Being a microcontroller, it is certainly possibly to connect things directly to the Pico (or another board). This covers the usual LCB requirements such as switches, LEDs, servos, displays and other types of sensors, and all of these can be integrated into a MicroPython program.

Like many contemporary microcontrollers, the Pico (and the ESPs) is a 3.3V device, meaning that some kind of logic level shifting will be required to interface with 5V devices, including most MERG modules. I am currently creating a development of my Pico CBUS shield that includes built-in level-shifting on some inputs. External level-shifting modules of various types are easily available from the usual sources.

## Other possibilities

I mentioned above the Pico W, which has a WiFi interface. I have used this to create a wireless CBUS interface, rather like a CANUSB4 or CANETHER, but supporting multiple concurrent wireless clients. With Internet access, it can also provide a synchronised time source for the layout, using CBUS FCLK messages. This would also be possible using an ESP32 or ESP8266, although these are constrained by having less memory than the Pico, and thus the size of programs that they run.

## Current status

As at the time of writing (January 2023), the basic CBUS module code is complete and has been tested using my PCB designs. It has now been released and is being evaluated by a few MERG members. I will be giving a presentation to the Raspberry Pi SIG on 17th January. By the time you read this, I hope there will be some real-world examples of other members’ work available to talk about.

I will also create a wiki page on the MERG Knowledgebase.

## Code examples

\*\*\* insert examples here \*\*\*