Prototype to product with MicroPython

faster and happier embedded systems development

Ned Konz, Teardown 2024

About me

- Embedded HW/SW since early '80s
  - Industrial, consumer, medical product development
- Object-oriented since early '90s
  - Alpha user of Walter Bright's Zortech C++ (1988?)
  - Smalltalk 1994-2006
  - Worked for Alan Kay 2004-2006 (Squeak Smalltalk)
  - Ruby 2006-
  - Python 2019-
- Recent MicroPython projects
  - IoT Toothbrush
  - Catbox monitor (personal project)
  - Fluke 8840A/8842A continuity beeper

What we'll cover in this talk

- About MicroPython
- MicroPython overview
- A couple of my HW projects
- Development and debugging strategies
- Using Python source (and/or byte code) in files
- Building your own version of MicroPython
- Freezing modules
- Adding User C or C++ modules
- Asyncio, Interrupts, and Threading
- Optimizing speed
- Optimizing flash and RAM usage

About MicroPython

What is MicroPython?

- Open-source project hosted on Github
- Originally developed by Damien George for use with Pyboard Kickstarter in 2013
- Python 3.4 with curated features from later Python versions (including asyncio)
- Optimized to run on microcontrollers
- Even been used in space satellites!
- Subset of Python’s standard libraries
- Additional libraries to aid development
- Build system and tools (**mpremote**, **mip**, **webrepl-cli**)
- Compiles to bytecode which is interpreted at runtime by the virtual machine
- Also can compile to native code
- Includes support for REPL, compilation, networking, device access

Why MicroPython?

- Ease of Use and Learning Curve
- Rapid Development and Prototyping
- Cross-Platform Compatibility
- Rich Ecosystem and Community Support
- Performance and Efficiency
- Cost-Effectiveness
- Integration with Existing Python Ecosystem
- Educational Value
- Networking and IoT Capabilities
- Active Development and Innovation

Why NOT MicroPython?

- Lower speed (though native and viper generators or inline assembly can help)
- Memory (flash and RAM) usage is higher
- Not hard-real time (because of GC and interpreter)
- Debug support is limited (though pdb is planned for 1.24)
- Integration with C/C++ libraries requires additional C code
- You might not be as experienced in Python as in some other language

MicroPython Ports (mainline)

esp32	Espressif ESP32 SoC with Wi-Fi and Bluetooth capabilities.
esp8266	Espressif ESP8266
mimxrt	NXP i.MX RT series
nrf	Nordic Semiconductor's nRF series of BLE MCUs
renesas-ra	Renesas RA series
rp2	Raspberry Pi RP2040
samd	Atmel/Microchip SAM D series ARM Cortex-M0+
stm32	STMicroelectronics STM32 ARM Cortex-M
zephyr	for the Zephyr RTOS, on various microcontrollers. (work in progress)

Special Purpose	
embed	for embedding MicroPython inside other programs
minimal	A minimalistic port used as a template and for testing.
qemu-arm	for running in the QEMU ARM emulator.
unix	for Unix-like operating systems, useful for testing and development.
webassembly	compiles to WebAssembly for running in web browsers.
windows	for Windows operating systems, useful for testing and development.

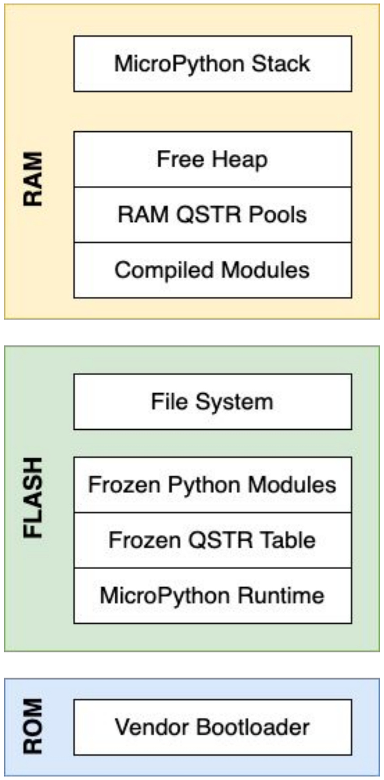
Other/Incomplete	
cc3200	Texas Instruments CC3200 Wi-Fi SoC.
pic16bit	Microchip's PIC 16-bit microcontrollers (in development, limited support).
powerpc	PowerPC architecture processors (in development, limited support).
bare-arm	for ARM Cortex-M MCUs with no operating system.

CircuitPython

- Project started in 2017; mostly funded by Adafruit
- Scott Shawcroft is primary maintainer
- Primary ports:
  - atmel-samd: Microchip SAMD21, SAMx5x
  - cxd56: Sony Spresense
  - espressif: Espressif ESP32, ESP32-S2, ESP32-S3, ESP32-C3, ESP32-C6
  - nrf: Nordic nRF52840, nRF52833
  - raspberrypi: Raspberry Pi RP2040
  - stm: ST STM32F4 family
- Emphasis is on education and consistency between ports
- Allows flexible USB device operation (CDC, MSC, MIDI, HID)
- Supports many/most of Adafruit's boards and sensors
- Adafruit `blinka` library available for running CircuitPython programs on MicroPython
- Supports native USB on most boards and BLE otherwise, allowing file editing without special tools.
- Floats (aka decimals) are enabled for all builds.
- Error messages are translated into 10+ languages.
- Concurrency within Python is not well supported. Interrupts and threading are disabled. `async/await` keywords are available on some boards for cooperative multitasking. Some concurrency is achieved with native modules for tasks that require it such as audio file playback.

MicroPython Overview

Memory and MicroPython



- MicroPython runtime, compiler and frozen modules are stored in flash
- File system(s) in flash, SD card, ...
- Heap is in RAM
  - Runtime-compiled modules and other objects
  - Allocated 16-bytes at a time
  - Garbage collected using mark/sweep
- Stack is in RAM
- Frozen modules are imported from **`.frozen`** pseudo-path
- default **`sys.path`** is `['', '.frozen', '/lib']`

stm32 port

- First port by Damien George (Pyboard STM32F405)
- Most comprehensively supported port
- Built on top of STM32 LL and HAL
- Hybrid (flash/SD card) filesystem available
- Wide variety of boards, including most of the STMicro Nucleo series
- ADAFRUIT_F405_EXPRESS, ARDUINO_GIGA, ARDUINO_NICLA_VISION, ARDUINO_PORTENTA_H7, B_L072Z_LRWAN1, B_L475E_IOT01A, CERB40, ESPRUINO_PICO, GARATRONIC_NADHAT_F405, GARATRONIC_PYBSTICK26_F411, HYDRABUS, LEGO_HUB_NO6, LEGO_HUB_NO7, LIMIFROG, MIKROE_CLICKER2_STM32, MIKROE_QUAIL, NETDUINO_PLUS_2, NUCLEO_F091RC, NUCLEO_F401RE, NUCLEO_F411RE, NUCLEO_F412ZG, NUCLEO_F413ZH, NUCLEO_F429ZI, NUCLEO_F439ZI, NUCLEO_F446RE, NUCLEO_F722ZE, NUCLEO_F746ZG, NUCLEO_F756ZG, NUCLEO_F767ZI, NUCLEO_G0B1RE, NUCLEO_G474RE, NUCLEO_H563ZI, NUCLEO_H723ZG, NUCLEO_H743ZI, NUCLEO_H743ZI2, NUCLEO_L073RZ, NUCLEO_L152RE, NUCLEO_L432KC, NUCLEO_L452RE, NUCLEO_L476RG, NUCLEO_L4A6ZG, NUCLEO_WB55, NUCLEO_WL55, OLIMEX_E407, OLIMEX_H407, PYBD_SF2, PYBD_SF3, PYBD_SF6, PYBLITEV10, PYBV10, PYBV11, PYBV3, PYBV4, SPARKFUN_MICROMOD_STM32, STM32F411DISC, STM32F429DISC, STM32F439, STM32F4DISC, STM32F769DISC, STM32F7DISC, STM32H573I_DK, STM32H7B3I_DK, STM32L476DISC, STM32L496GDISC, USBDONGLE_WB55, VCC_GND_F407VE, VCC_GND_F407ZG, VCC_GND_H743VI

esp32 port

- MicroPython runs as a FreeRTOS task on one core
- Threads created as FreeRTOS tasks on same core
- Supports esp32, esp32-s2, esp32-s3, esp32-c3 (esp32-c6 soon)
- `cmake` based build, uses esp-idf (separate installation required)
- Networking via on-chip wifi or external Ethernet PHY
- BLE and espnow RF protocols also supported (no BLE on -S2)
- ADCs on some esp32 variants have limited range (not down to 0V)
- Supports RMT, OneWire, capacitive touch, SD card
- DAC available on esp32 and esp32-s2
- Built-in JTAG over USB (but configuration is tricky)
- OTA updating possible with modified partition table

esp32 port (continued)

- Supported boards: M5STACK_ATOM, UM_TINYPICO, LOLIN_S2_PICO, SIL_WESP32, UM_PROS3, UM_TINYS3, OLIMEX_ESP32_POE, UM_NANOS3, UM_TINYS2, LOLIN_S2_MINI, ESP32_GENERIC, UM_TINYWATCHS3, ESP32_GENERIC_C3, UM_FEATHERS2, UM_FEATHERS3, ESP32_GENERIC_S3, ESP32_GENERIC_S2, LILYGO_TTGO_LORA32, ARDUINO_NANO_ESP32, LOLIN_C3_MINI, UM_FEATHERS2NEO
- OS debug messages available over UART (or USB CDC)
- Supports lightsleep and deepsleep to save power
- Wake from touch, GPIO, or timeout
- Native code generator
- REPL over USB virtual COM port for devices with USB (-S2, -S3)

rp2 (rp2040) port

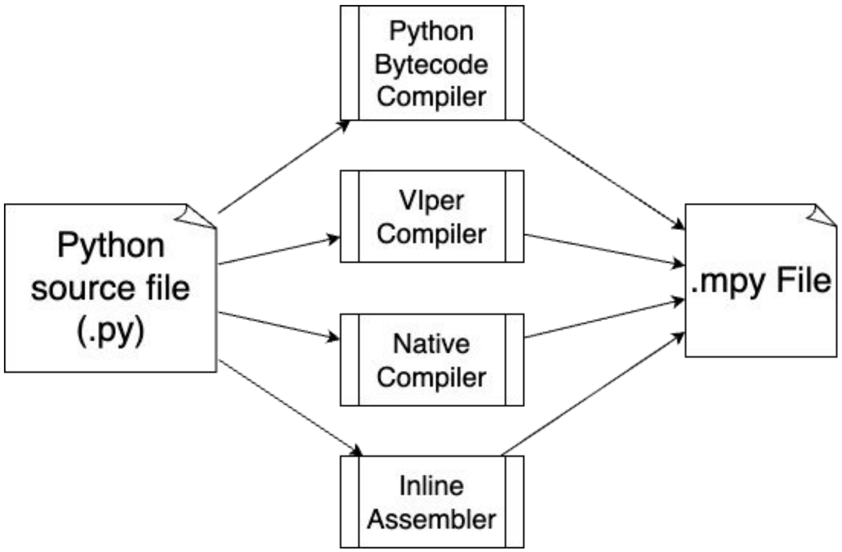
- stock boards: ADAFRUIT_FEATHER_RP2040, ADAFRUIT_ITSYBITSY_RP2040, ADAFRUIT_QTPY_RP2040, ARDUINO_NANO_RP2040_CONNECT, GARATRONIC_PYBSTICK26_RP2040, NULLBITS_BIT_C_PRO, PIMORONI_PICOLIPO_16MB, PIMORONI_PICOLIPO_4MB, PIMORONI_TINY2040, POLOLU_3PI_2040_ROBOT, POLOLU_ZUMO_2040_ROBOT, RPI_PICO, RPI_PICO_W, SIL_RP2040_SHIM, SPARKFUN_PROMICRO, SPARKFUN_THINGPLUS, W5100S_EVB_PICO, W5500_EVB_PICO, WEACTSTUDIO
- REPL over USB virtual COM port
- native code generation and inline assembler
- rp2 module provides PIO assembler and support, as well as DMA
- build produces .uf2 image that can be loaded by ROM bootloader

Startup sequence

- main.c (main() or mp_task() for FreeRTOS ports like esp32)
  - initialize clocks, some peripherals
  - .frozen/_boot (frozen .mpy file): mounts filesystem
    - first time: creates filesystem and default /boot.py
  - .frozen/boot or /boot.py from filesystem
  - .frozen/main or /main.py from filesystem

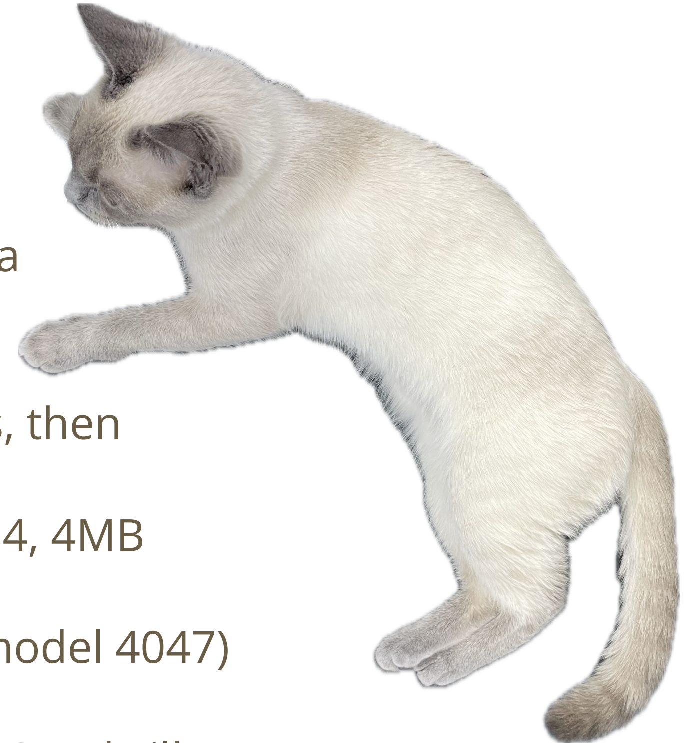
Cross Compilation

- `mpy-cross` reads `.py` files and writes `.mpy` files.
- `.mpy` files may be frozen into image binary (during build) or copied to the device filesystem
- frozen files will be run from flash so will save RAM
- `.mpy` files from filesystem get copied to RAM and run but don't require RAM and time to compile on the device



A Couple of Projects

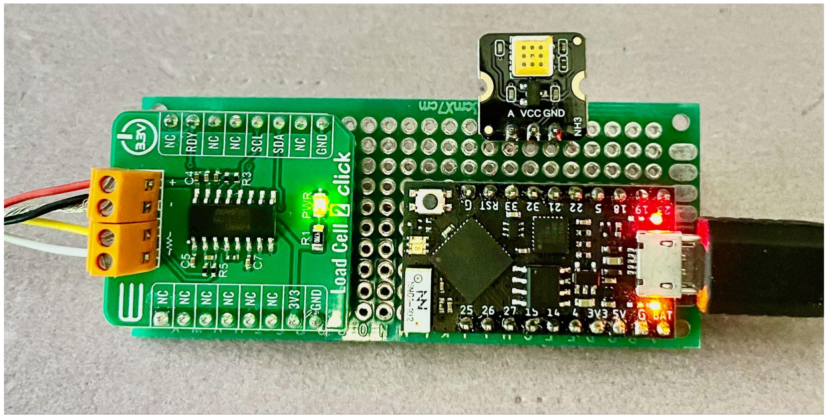
Cat litter box Monitor



- Three-cat household, including a 19 year old and a 0.75-year-old.
- Two litter boxes, two humans with ADHD
- Solution: monitor box weight and ammonia levels, then analyze it somehow and provide alerts
- TinyPICO v2 by Unexpected Maker (ESP32-PICO-D4, 4MB flash, 520+16 KB SRAM, 4MB PSRAM)
- Mikroe NAU7802 load-cell-2-click board (Mikroe model 4047)
- DFRobot Ammonia sensor board SEN0567-NH3
- Four load cells harvested from a bathroom scale (Goodwill, $1.99)
- Rigid foam board for mounting the load cells (40cm x 53cm)

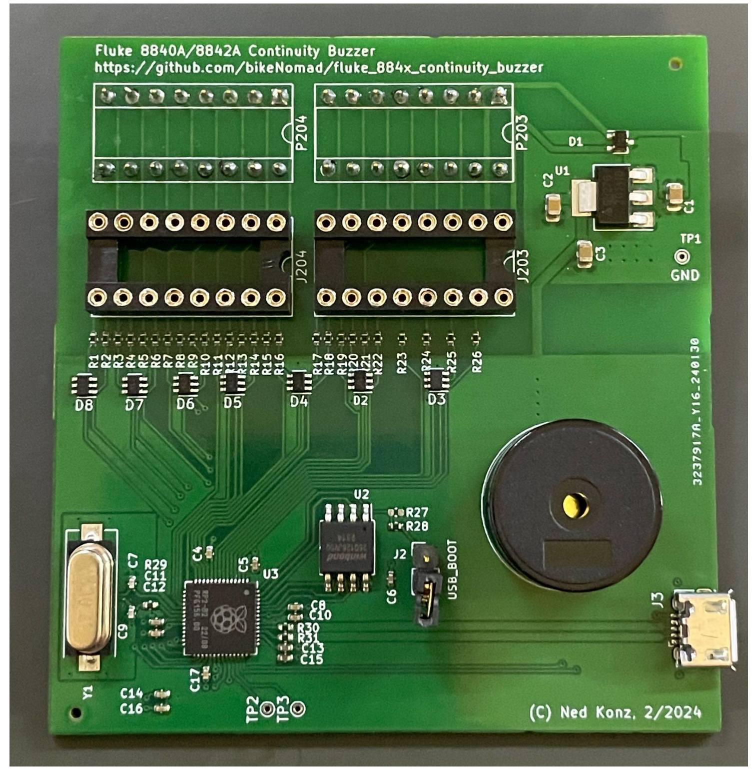
Litter box monitor (continued)

- Using micropython-mqtt library from Peter Hinch
- Publishing data to eclipse `mosquitto` broker running on Synology NAS
- Getting data from `mosquitto` with `telegraf`
- Storing in `impulsedb`
- Added aiorepl for ease of debugging
- Frozen image size: 1.56MB



Fluke 8840A/8842A Continuity Buzzer

- RP2040-based board
- Watches 30V logic level signals going to VFD display
- Interprets display segments to determine mode and reading
- Buzzes when below 10Ω
- Uses stock rp2 build of MicroPython
- Only 333 lines of Python code



Development and debugging strategies

Using the REPL

- REPL == Read-Evaluate-Print Loop
- Accessible via serial (UART, USB) on all ports
- Accessible via WebREPL (HTTP/websockets) on network-enabled ports
- Requires compiler built-in (standard)
- Allows for interactive exploration and diagnosis
- `from upysh import *`
  - `ls`, `rm('file')`, `rm('dir', True)`, `mkdir('dir')`, etc.
- Can disable REPL (for released product)

Using Thonny

- Thonny is a cross-platform Python IDE
- Knows about various MicroPython/CircuitPython ports
- Can load stock binaries and libraries onto boards
- Shows device filesystem view and provides file operations between device and host
- Offers editing of on-device files
- Provides REPL editor
- Has numeric value plotter that extracts numbers from REPL output
- Can run and debug host Python programs

Using mpremote

- `mpremote` tool comes with MicroPython, but can also install using `pipx`
  - `mpremote repl`
  - `mpremote cp main.py :main.py`
  - `mpremote mkdir :/app`
  - `mpremote cp -r app :`
  - `mpremote exec 'help("modules")'`
  - `mpremote edit :main.py`
- can create macros for your own custom commands
- `mpremote mount ./src`
  - mounts `./src` as /remote on device, does `os.chdir('/remote')`
  - easiest to use without running your app from `main.py`

Copying files to the device filesystem

- During development, `.py` files on the filesystem are easy to use
- Copy using `mpremote`, Thonny, `rshell`, `pyboard.py`
- Using `mpremote mount` avoids copying
- editing `sys.path` allows for overriding frozen modules with filesystem modules
  - Can't override frozen `boot.py` though you can override frozen `main.py`
  - After `/main.py, frozen/main` will run

Overriding frozen main.py and/or boot.py

- In your frozen main.py and/or boot.py:

```
import os
# allow for filesystem /main.py to override this one
print("start frozen main.py")
root_files = os.listdir()
if 'main.py' in root_files:
   import main
else:
   # do normal things
```

Freezing modules

- Frozen modules are compiled from Python to `.mpy` then combined into image binary.
  - Build using `make`, passing in
  - `FROZEN_MANIFEST`=<*your manifest.py*>
- `manifest.py` contains commands for freezing modules:
  - **`add_library`**(*library, library_path, prepend=False*)
    - register a library path for require
  - **`package`**(*package_path, files=None, base_path='.', opt=None*)
    - copy package_path directory to the device as frozen code
  - **`module`**(*module_path, base_path='.', opt=None*)
    - freeze a single Python file as a module
  - **`require`**(*name, library=None*)
    - Require a library (and its dependencies) from micropython-lib or from the library name registered by add_library()
  - **`include`**(*manifest_path*)
    - include another manifest file
  - **`freeze`**(*path, script=None, opt=0*)
    - freeze all the .py files under path

Adding third-party libraries to the filesystem

- `mip` (MicroPython's version of `pip`) installs libraries from micropython-lib or other sources (Github, Gitlab)
- Doesn't handle depencies
- Fetches compiled `.mpy` files to `/lib` (by default; can override)
- Run from REPL:

```
>>> import mip
>>> mip.install("pkgname")  # Installs the latest version of "pkgname" (and dependencies)
>>> mip.install("pkgname", version="x.y")  # Installs version x.y of "pkgname"
>>> mip.install("pkgname", mpy=False)  # Installs the source version (i.e. .py rather than .mpy files)
```

- Or from host PC:
  - `mpremote mip install` *<packages>*

Great 3rd-Party Libraries

- micropython-lib https://github.com/micropython/micropython-lib
  - CPython compatibility, extensions
- microdot (asyncio web/websocket server like flask)
- aiorepl (asyncio REPL)
- ulab (NumPy subset)
- micropython-esp32-ota (esp32 OTA support)
- utemplate (templating engine based on generators)
- aioble (Bluetooth Low Energy using asyncio)
- OpenMV (machine vision; STM32-based cameras)
- See curated list at https://awesome-micropython.com

Custom board definitions and partition layouts

- Each board has a "board directory"; these can be out-of-tree
- Important files:
  - `mpconfigboard.h` : macro definitions to enable/disable/configure MicroPython compilation (extends port-specific `mpconfigport.h`)
  - `partitions.csv` (esp32)
  - `manifest.py` (but can override at build time)
  - `pins.csv` (provides Python names for board-specific pins in `machine.Pin.board`)
- Examples at https://github.com/micropython/micropython-example-boards
- Github search for "*path:mpconfigboard.h*" finds 4.7k results!

C or C++ extension modules

- Compiled into micropython image binary
- Can/should be built out-of-tree
- Examples in `micropython/examples/usercmodule`
- Also look at `micropython/py/mod*.c` for examples
- Add all C files to `SRC_USERMOD` (in micropython.mk)
- Add all C++ files to `SRC_USERMOD_CXX` (in micropython.mk)
- Can add C flags to `CFLAGS_USERMOD`
- Can add C++ flags to `CXXFLAGS_USERMOD`
- For C++, also `LDFLAGS_USERMOD` += -lstdc++

Native code in .mpy files

- Allow for writing dynamically loadable modules in C or C++
- Used for I2S and bluetooth in stock MicroPython
- Must be written as position-independent code (PIC)
- Only linked against pre-defined subset of MicroPython (can't call arbitrary HAL/RTOS functions)

Object Representations

- `mp_obj_t` can have one of four representations, defined by `MICROPY_OBJ_REPR` (see `micropython/py/mpconfig.h`)
  - determines size of immediates (small integers, QSTR indexes)
  - some representations have immediate floats
  - pointers to `mp_obj_base_t` signified by low-order 0 bits
  - If you're doing lots of float math, C or D might be appropriate
  - None/False/True represented as immediates in all but D
- If you change `MICROPY_OBJ_REPR` you may also need to define `UINT_FMT`, `INT_FMT`, and typedefs for `mp_int_t` and `mp_uint_t`

Object Representations (continued)

- `MICROPY_OBJ_REPR_A` (default): one machine word, 31-bit small integer, 29-bit QSTR indexes, 30-bit pointers (add 32 bits for 64-bit machine words)
- `MICROPY_OBJ_REPR_B`: one machine word, 30-bit small integer, 29-bit QSTR indexes, 31-bit pointers (add 32 bits for 64-bit machine words)
- `MICROPY_OBJ_REPR_C`: 32-bits, 30-bit float, 31-bit small integer, 19-bit QSTR indexes, 30-bit pointers
- `MICROPY_OBJ_REPR_D`: 64-bits, 64-bit float, 47-bit small integer, 31-bit QSTR indexes

Interrupts

- Asynchronous callbacks upon events
- Two kinds:
  - hard interrupts can happen in the middle of a bytecode
  - soft interrupts run callback between bytecodes (same as `micropython.schedule()`)
  - esp32 uses only soft interrupts
- Interrupts available for some ports for some events
  - GPIO: pin change (stm32: hard or soft)
  - timers: timeout
  - bluetooth.BLE (soft), espnow.ESPNow (soft)
  - UART, rp2.DMA, rp2.PIO, machine.RTC, machine.I2S (soft)

Interrupts (continued)

- use `machine.disable_irq()`, `machine.enable_irq()` around main program access to data modified by interrupt handlers
- hard interrupt handlers (callbacks) can't allocate memory
  - runtime exception will be raised if you try
  - use pre-allocated memory or integers/booleans for return values from handlers
- keep handlers as short and simple as possible

Threading

- MicroPython has `_thread` module in some ports (cc3200, esp32, rp2)
- esp32: threads are FreeRTOS tasks on the same core
- Most ports have standard Python GIL (global interpreter lock), so threads can stall each other
- rp2: no GIL; can have two independent MicroPython runtimes, one on each core (be careful with synchronization!)
- Prefer `asyncio` to threading wherever possible

Asyncio

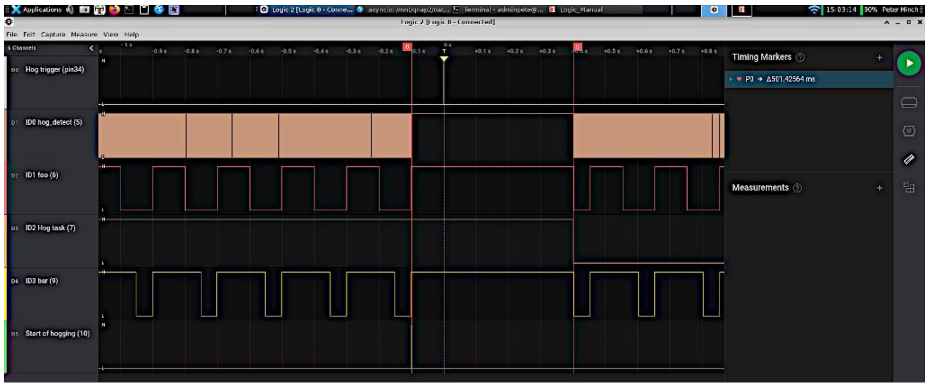
- Cooperative multi-tasking: tasks yield when awaiting events
- Can easily run hundreds of tasks at once
- MicroPython supports:
  - async functions
  - await keyword
  - async generators
  - async with …
  - asyncio.ThreadSafeFlag (for synchronizing with threads, interrupt handlers, or scheduler callbacks)
  - asyncio.Event, asyncio.Lock (for synchronizing between asyncio tasks)
  - asyncio.Stream (TCP stream connection)
  - asyncio.start_server() (start a TCP server; can use TLS)

Asyncio (continued)

- Excellent tutorial at https://github.com/peterhinch/micropython-async/blob/master/v3/docs/TUTORIAL.md
- Useful asyncio extensions from Peter Hinch:
  - Queue, RingbufQueue (communicate/synchronize between multiple tasks)
  - Semaphore, retriggerable Delay_ms
  - WaitAny, WaitAll (wait on multiple events)
  - ThreadSafeQueue, ThreadSafeEvent, Message (synchronize with threads, ISRs)
  - Drivers for switches, pushbuttons, esp32 touchbuttons, ADCs, incremental encoders

Debugging asyncio programs

- `aiorepl` by Matt Trentini provides an asyncio REPL with line-editing and history (no tab-completion)
- `aioprof` by Andrew Leech shows task statistics
- https://github.com/peterhinch/micropython-monitor uses a Raspberry Pi Pico to display task timing on a logic analyzer



Optimizing speed

- use `const()` where possible
- use local variables instead of globals
- pre-allocate variables and cache object references for use inside loops
- use pre-allocated buffers for streams, etc.
- avoid growing lists (use pre-allocated `bytearray` or `array` instances if you can)
- use integers instead of floating point
- use `memoryview` objects instead of slices
- periodically call `gc.collect()` to avoid long delays
- use the native code emitters (`@micropython.native` decorator) (~2x bytecode speed but bigger)
- use the Viper code emitter (`@micropython.viper` decorator) (not standard Python though)
- access hardware registers directly
- use the inline assembler (`@micropython.asm_thumb` decorator)
- (rp2) use the PIO and DMA for fast I/O

Optimizing size

- keep strings & names < 10 characters long so they can be shared as QSTRs
- freeze modules so they can run from flash
- use const() with variables named with initial underscores
- Pre-compile to .mpy to save compiler RAM usage
- use bytearray or array instead of list objects

Conclusion

- MicroPython speeds your development
- You may be able to use a stock build for your hardware, but if not you can easily configure and build a custom image
- Excellent community support
- Wide selection of already-supported MCUs and boards