MicroPython Documentation

Release latest

MicroPython authors and contributors

Sep 29, 2026

CONTENTS

1 MicroPython libraries **1**

1.1 Python standard libraries and micro-libraries . . . 2
- 1.1.1 `array` – arrays of numeric data . . . 2
- 1.1.2 `asyncio` — asynchronous I/O scheduler . . . 3
- 1.1.3 `binascii` – binary/ASCII conversions . . . 8
- 1.1.4 `builtins` – builtin functions and exceptions . . . 8
- 1.1.5 `cmath` – mathematical functions for complex numbers . . . 13
- 1.1.6 `collections` – collection and container types . . . 14
- 1.1.7 `errno` – system error codes . . . 15
- 1.1.8 `gc` – control the garbage collector . . . 16
- 1.1.9 `gzip` – gzip compression & decompression . . . 18
- 1.1.10 `hashlib` – hashing algorithms . . . 19
- 1.1.11 `heapq` – heap queue algorithm . . . 20
- 1.1.12 `io` – input/output streams . . . 21
- 1.1.13 `json` – JSON encoding and decoding . . . 25
- 1.1.14 `marshal` – Python object serialization . . . 25
- 1.1.15 `math` – mathematical functions . . . 26
- 1.1.16 `os` – basic “operating system” services . . . 28
- 1.1.17 `platform` – access to underlying platform’s identifying data . . . 31
- 1.1.18 `random` – generate random numbers . . . 32
- 1.1.19 `re` – simple regular expressions . . . 33
- 1.1.20 `select` – wait for events on a set of streams . . . 36
- 1.1.21 `socket` – socket module . . . 38
- 1.1.22 `ssl` – SSL/TLS module . . . 43
- 1.1.23 `string.templatelib` – Template String Support . . . 46
- 1.1.24 `struct` – pack and unpack primitive data types . . . 50
- 1.1.25 `sys` – system specific functions . . . 52
- 1.1.26 `time` – time related functions . . . 55
- 1.1.27 `weakref` – Python object lifetime management . . . 58
- 1.1.28 `zlib` – zlib compression & decompression . . . 59
- 1.1.29 `_thread` – multithreading support . . . 61

1.2 MicroPython-specific libraries . . . 61
- 1.2.1 `bluetooth` — low-level Bluetooth . . . 61
- 1.2.2 `btree` – simple BTree database . . . 73
- 1.2.3 `cryptolib` – cryptographic ciphers . . . 75
- 1.2.4 `deflate` – deflate compression & decompression . . . 76
- 1.2.5 `framebuf` — frame buffer manipulation . . . 78
- 1.2.6 `machine` — functions related to the hardware . . . 81
- 1.2.7 `micropython` – access and control MicroPython internals . . . 142
- 1.2.8 `neopixel` — control of WS2812 / NeoPixel LEDs . . . 146

1.2.9 network — network configuration . . . 147
1.2.10 openamp – provides standard Asymmetric Multiprocessing (AMP) support . . . 164
1.2.11 uctypes – access binary data in a structured way . . . 166
1.2.12 vfs – virtual filesystem control . . . 171
1.2.13 WM8960 – Driver for the WM8960 codec . . . 175
1.3 Port-specific libraries . . . 181
1.3.1 Libraries specific to the pyboard . . . 181
1.3.2 Libraries specific to the WiPy . . . 232
1.3.3 Libraries specific to the ESP8266 and ESP32 . . . 236
1.3.4 Libraries specific to NXP i.MXRT . . . 263
1.3.5 Libraries specific to the RP2040 . . . 264
1.3.6 Libraries specific to Zephyr . . . 278
1.4 Extending built-in libraries from Python . . . 284

2 MicroPython language and implementation 285
2.1 Glossary . . . 285
2.2 The MicroPython Interactive Interpreter Mode (aka REPL) . . . 288
2.2.1 Auto-indent . . . 288
2.2.2 Auto-completion . . . 289
2.2.3 Interrupting a running program . . . 289
2.2.4 Paste mode . . . 290
2.2.5 Soft reset . . . 290
2.2.6 The special variable _ (underscore) . . . 291
2.2.7 Raw mode and raw-paste mode . . . 291
2.3 Reset and Boot Sequence . . . 293
2.3.1 Hard reset . . . 293
2.3.2 Soft Reset . . . 293
2.3.3 Boot Sequence . . . 294
2.3.4 Soft Bricking (failure to boot) . . . 296
2.4 MicroPython remote control: mpremote . . . 297
2.4.1 Commands . . . 297
2.4.2 ROMFS commands . . . 302
2.4.3 Auto connection and soft-reset . . . 303
2.4.4 Shortcuts . . . 304
2.4.5 Examples . . . 305
2.5 MicroPython .mpy files . . . 307
2.5.1 Versioning and compatibility of .mpy files . . . 308
2.5.2 Binary encoding of .mpy files . . . 309
2.6 Writing interrupt handlers . . . 311
2.6.1 Tips and recommended practices . . . 311
2.6.2 MicroPython issues . . . 312
2.6.3 Exceptions . . . 315
2.6.4 Interfacing to asyncio . . . 315
2.6.5 General issues . . . 315
2.7 Maximising MicroPython speed . . . 318
2.7.1 Designing for speed . . . 319
2.7.2 Identifying the slowest section of code . . . 321
2.7.3 MicroPython code improvements . . . 321
2.7.4 The Native code emitter . . . 322
2.7.5 The Viper code emitter . . . 322
2.7.6 Accessing hardware directly . . . 324
2.8 MicroPython on microcontrollers . . . 324
2.8.1 Flash memory . . . 324
2.8.2 RAM . . . 325

2.8.3 The heap . . . 329
2.8.4 String operations . . . 330
2.8.5 Postscript . . . 330
2.9 MicroPython manifest files . . . 331
2.9.1 Summary . . . 331
2.9.2 Writing manifest files . . . 331
2.9.3 Examples . . . 334
2.10 Package management . . . 335
2.10.1 Installing packages with `mip` . . . 335
2.10.2 Installing packages with `mpremote` . . . 336
2.10.3 Installing packages manually . . . 337
2.10.4 Writing & publishing packages . . . 337
2.10.5 Freezing packages . . . 338
2.11 Inline assembler for Thumb2 architectures . . . 338
2.11.1 Document conventions . . . 339
2.11.2 Instruction categories . . . 339
2.11.3 Usage examples . . . 348
2.11.4 References . . . 352
2.12 Working with filesystems . . . 353
2.12.1 VFS . . . 353
2.12.2 Block devices . . . 354
2.12.3 Filesystems . . . 356
2.13 Working with ROMFS . . . 358
2.13.1 Overview . . . 359
2.13.2 Port support . . . 359
2.13.3 Enabling ROMFS for a port or board . . . 360
2.13.4 Workflow . . . 360
2.13.5 Automatic mounting at boot . . . 360
2.13.6 Using mpremote to manage ROMFS . . . 360
2.13.7 ROMFS usage examples . . . 361
2.14 ROMFS filesystem format . . . 362
2.15 Unicode Support . . . 363
2.15.1 Terminology . . . 363
2.15.2 Character Encoding . . . 363
2.15.3 Encoding and Decoding . . . 364
2.15.4 String Methods . . . 365
2.15.5 String Formatting . . . 365
2.15.6 Build Configuration . . . 366
2.15.7 Limitations . . . 367
2.16 The pyboard.py tool . . . 367
2.16.1 Running a command on the device . . . 368
2.16.2 Running a script on the device . . . 368
2.16.3 Filesystem access . . . 368
2.16.4 Using the pyboard library . . . 369
2.17 MicroPython 2.0 Migration Guide . . . 369
2.17.1 Hardware and peripherals . . . 370
2.17.2 OS & filesystem . . . 370
2.17.3 CPython compatibility . . . 370

3 MicroPython differences from CPython **371**
3.1 Python 3.5 . . . 371
3.2 Python 3.6 . . . 373
3.3 Python 3.7 . . . 375
3.4 Python 3.8 . . . 377

3.5 Python 3.9 . . . 379
3.6 Python 3.10 . . . 381
3.7 Syntax . . . 385
3.7.1 Annotations . . . 385
3.7.2 Literals . . . 386
3.7.3 Operators . . . 387
3.7.4 Unicode . . . 388
3.7.5 Unpacking . . . 388
3.8 Core language . . . 389
3.8.1 Classes . . . 389
3.8.2 Exceptions . . . 397
3.8.3 Functions . . . 397
3.8.4 Generator . . . 399
3.8.5 Runtime . . . 400
3.8.6 f-strings . . . 401
3.8.7 import . . . 402
3.9 Builtin types . . . 403
3.9.1 Exception . . . 403
3.9.2 OSError . . . 405
3.9.3 bytearray . . . 407
3.9.4 bytes . . . 408
3.9.5 complex . . . 411
3.9.6 dict . . . 411
3.9.7 float . . . 412
3.9.8 int . . . 412
3.9.9 list . . . 413
3.9.10 memoryview . . . 414
3.9.11 range . . . 415
3.9.12 str . . . 416
3.9.13 tuple . . . 421
3.10 Modules . . . 421
3.10.1 Positional-only Parameters . . . 421
3.10.2 array . . . 422
3.10.3 errno . . . 424
3.10.4 json . . . 424
3.10.5 os . . . 424
3.10.6 random . . . 425
3.10.7 struct . . . 426
3.10.8 sys . . . 427

4 MicroPython Internals **429**
4.1 Getting Started . . . 429
4.1.1 Source control with git . . . 429
4.1.2 Get the code . . . 429
4.1.3 Compile and build the code . . . 430
4.1.4 Building the documentation . . . 432
4.1.5 Running the tests . . . 433
4.1.6 Additional make targets for developers . . . 433
4.1.7 Using ci.sh locally . . . 433
4.1.8 Folder structure . . . 434
4.2 Writing tests . . . 435
4.2.1 Using run-tests.py . . . 436
4.3 The Compiler . . . 437
4.3.1 Adding a grammar rule . . . 437

4.3.2 Adding a lexical token . . . 438
4.3.3 Parsing . . . 439
4.3.4 Compiler passes . . . 439
4.3.5 Emitting bytecode . . . 441
4.3.6 Emitting native code . . . 442
4.4 Memory Management . . . 442
4.4.1 MicroPython Memory from C code . . . 442
4.4.2 The object model . . . 443
4.4.3 Allocation of objects . . . 444
4.5 Implementing a Module . . . 445
4.5.1 Implementing a core module . . . 446
4.6 Optimizations . . . 447
4.6.1 Frozen bytecode . . . 447
4.6.2 Variables . . . 447
4.6.3 Allocation of memory . . . 448
4.7 MicroPython string interning . . . 448
4.7.1 Compile-time QSTR generation . . . 448
4.7.2 Run-time QSTR generation . . . 449
4.8 Maps and Dictionaries . . . 450
4.8.1 Open addressing . . . 450
4.8.2 Linear probing . . . 450
4.9 The public C API . . . 451
4.10 Extending MicroPython in C . . . 451
4.10.1 MicroPython external C modules . . . 452
4.10.2 Native machine code in .mpy files . . . 457
4.11 Porting MicroPython . . . 461
4.11.1 Minimal MicroPython firmware . . . 462
4.11.2 MicroPython Configurations . . . 464
4.11.3 Support for standard input/output . . . 464
4.11.4 Building and running . . . 465
4.11.5 Adding a module to the port . . . 466
4.12 MicroPython Support Tiers . . . 467

5 MicroPython license information 469

6 Quick reference for the pyboard 471
6.1 General information about the pyboard . . . 472
6.1.1 Local filesystem and SD card . . . 472
6.1.2 Boot modes . . . 472
6.1.3 Errors: flashing LEDs . . . 473
6.1.4 Guide for using the pyboard with Windows . . . 473
6.1.5 The pyboard hardware . . . 473
6.1.6 Datasheets for the components on the pyboard . . . 473
6.1.7 Datasheets for other components . . . 474
6.2 MicroPython tutorial for the pyboard . . . 474
6.2.1 Introduction to the pyboard . . . 474
6.2.2 Running your first script . . . 475
6.2.3 Getting a MicroPython REPL prompt . . . 478
6.2.4 Turning on LEDs and basic Python concepts . . . 480
6.2.5 Switches, callbacks and interrupts . . . 482
6.2.6 The accelerometer . . . 483
6.2.7 Safe mode and factory reset . . . 485
6.2.8 Making the pyboard act as a USB mouse . . . 486
6.2.9 The Timers . . . 488

6.2.10 Inline assembler . . . 490
6.2.11 Power control . . . 492
6.2.12 Tutorials requiring extra components . . . 492
6.2.13 Tips, tricks and useful things to know . . . 504
6.3 General board control . . . 505
6.4 Delay and timing . . . 505
6.5 Internal LEDs . . . 506
6.6 Internal switch . . . 506
6.7 Pins and GPIO . . . 506
6.8 Servo control . . . 506
6.9 External interrupts . . . 507
6.10 Timers . . . 507
6.11 RTC (real time clock) . . . 507
6.12 PWM (pulse width modulation) . . . 507
6.13 ADC (analog to digital conversion) . . . 508
6.14 DAC (digital to analog conversion) . . . 508
6.15 UART (serial bus) . . . 508
6.16 SPI bus . . . 508
6.17 I2C bus . . . 509
6.18 I2S bus . . . 509
6.19 CAN bus (controller area network) . . . 509
6.20 Internal accelerometer . . . 510

7 Quick reference for the ESP8266 511
7.1 General information about the ESP8266 port . . . 512
7.1.1 Multitude of boards . . . 512
7.1.2 Technical specifications and SoC datasheets . . . 512
7.1.3 Scarcity of runtime resources . . . 513
7.1.4 Boot process . . . 513
7.1.5 Known Issues . . . 513
7.2 MicroPython tutorial for ESP8266 . . . 514
7.2.1 Getting started with MicroPython on the ESP8266 . . . 515
7.2.2 Getting a MicroPython REPL prompt . . . 519
7.2.3 The internal filesystem . . . 522
7.2.4 Network basics . . . 523
7.2.5 Network - TCP sockets . . . 525
7.2.6 GPIO Pins . . . 527
7.2.7 Pulse Width Modulation . . . 528
7.2.8 Analog to Digital Conversion . . . 529
7.2.9 Power control . . . 530
7.2.10 Controlling 1-wire devices . . . 531
7.2.11 Controlling NeoPixels . . . 531
7.2.12 Controlling APA102 LEDs . . . 533
7.2.13 Temperature and Humidity . . . 534
7.2.14 Using a SSD1306 OLED display . . . 535
7.2.15 Next steps . . . 537
7.3 Installing MicroPython . . . 537
7.4 General board control . . . 537
7.5 Networking . . . 538
7.6 Delay and timing . . . 538
7.7 Timers . . . 539
7.8 Pins and GPIO . . . 539
7.9 UART (serial bus) . . . 539
7.10 PWM (pulse width modulation) . . . 540

7.11 ADC (analog to digital conversion) . . . 540
7.12 Software SPI bus . . . 541
7.13 Hardware SPI bus . . . 541
7.14 I2C bus . . . 541
7.15 Real time clock (RTC) . . . 542
7.16 WDT (Watchdog timer) . . . 542
7.17 Deep-sleep mode . . . 542
7.18 OneWire driver . . . 543
7.19 NeoPixel driver . . . 543
7.20 APA102 driver . . . 544
7.21 DHT driver . . . 544
7.22 SSD1306 driver . . . 545
7.23 WebREPL (web browser interactive prompt) . . . 545

8 Quick reference for the ESP32 547
8.1 General information about the ESP32 port . . . 548
8.1.1 Multitude of boards . . . 548
8.1.2 Technical specifications and SoC datasheets . . . 548
8.2 MicroPython tutorial for ESP32 . . . 549
8.2.1 Getting started with MicroPython on the ESP32 . . . 549
8.2.2 Pulse Width Modulation . . . 551
8.2.3 Accessing peripherals directly via registers . . . 555
8.2.4 Factory reset . . . 558
8.3 Installing MicroPython . . . 558
8.4 General board control . . . 558
8.5 Networking . . . 559
8.5.1 WLAN . . . 559
8.5.2 LAN . . . 560
8.6 Delay and timing . . . 562
8.7 Timers . . . 562
8.8 Pins and GPIO . . . 563
8.9 UART (serial bus) . . . 564
8.10 PWM (pulse width modulation) . . . 564
8.11 DAC (digital to analog conversion) . . . 565
8.12 ADC (analog to digital conversion) . . . 566
8.13 Pulse Counter (pin pulse/edge counting) . . . 567
8.14 Software SPI bus . . . 568
8.15 Hardware SPI bus . . . 569
8.16 Software I2C bus . . . 569
8.17 Hardware I2C bus . . . 569
8.18 I2S bus . . . 570
8.19 Real time clock (RTC) . . . 570
8.20 WDT (Watchdog timer) . . . 570
8.21 Deep-sleep mode . . . 571
8.22 SD card . . . 572
8.23 RMT . . . 572
8.24 OneWire driver . . . 572
8.25 NeoPixel and APA106 driver . . . 573
8.26 Capacitive touch . . . 573
8.27 DHT driver . . . 574
8.28 WebREPL (web browser interactive prompt) . . . 574

9 Quick reference for the RP2 577
9.1 General information about the RP2xxx port . . . 578

9.1.1 Technical specifications and SoC datasheets . . . 578
9.2 Getting started with MicroPython on the RP2xxx . . . 579
9.2.1 Factory reset . . . 579
9.2.2 Programmable IO . . . 579
9.3 Installing MicroPython . . . 582
9.4 General board control . . . 582
9.5 Networking . . . 583
9.5.1 WLAN . . . 583
9.6 Delay and timing . . . 584
9.7 Timers . . . 584
9.8 Pins and GPIO . . . 584
9.9 Programmable IO (PIO) . . . 585
9.10 UART (serial bus) . . . 585
9.11 PWM (pulse width modulation) . . . 586
9.12 ADC (analog to digital conversion) . . . 586
9.13 Software SPI bus . . . 587
9.14 Hardware SPI bus . . . 587
9.15 Software I2C bus . . . 587
9.16 Hardware I2C bus . . . 588
9.17 I2S bus . . . 588
9.18 Real time clock (RTC) . . . 588
9.19 WDT (Watchdog timer) . . . 589
9.20 OneWire driver . . . 589
9.21 NeoPixel and APA106 driver . . . 589

10 Quick reference for the i.MXRT family **591**
10.1 General information about the MIMXRT port . . . 592
10.1.1 Multitude of boards . . . 592
10.1.2 Supported MCUs . . . 592
10.1.3 Technical specifications and SoC datasheets . . . 593
10.2 Getting started with MicroPython on the i.MXRT . . . 593
10.2.1 Requirements . . . 594
10.2.2 Powering the board . . . 594
10.2.3 Getting the firmware . . . 594
10.2.4 Deploying the firmware . . . 594
10.2.5 Serial prompt . . . 595
10.2.6 Troubleshooting installation problems . . . 595
10.3 Pinout for the i.MXRT machine modules . . . 595
10.3.1 UART pin assignment . . . 596
10.3.2 PWM pin assignment . . . 597
10.3.3 Hardware SPI pin assignment . . . 603
10.3.4 Hardware I2C pin assignment . . . 605
10.3.5 Hardware I2S pin assignment . . . 606
10.3.6 Pin Assignment . . . 606
10.3.7 Hardware CAN pin assignment . . . 607
10.4 Installing MicroPython . . . 608
10.5 General board control . . . 608
10.6 Delay and timing . . . 608
10.7 Timers . . . 608
10.8 Pins and GPIO . . . 609
10.9 UART (serial bus) . . . 609
10.10 PWM (pulse width modulation) . . . 610
10.10.1 PWM Constructor . . . 610
10.10.2 PWM Methods . . . 611

10.10.3 PWM Pin Assignment . . . 612
10.11 ADC (analog to digital conversion) . . . 612
10.12 Software SPI bus . . . 612
10.13 Hardware SPI bus . . . 613
10.14 Software I2C bus . . . 613
10.15 Hardware I2C bus . . . 613
10.16 I2S bus . . . 614
10.17 Real time clock (RTC) . . . 615
10.18 SD card . . . 615
10.19 OneWire driver . . . 616
10.20 DHT driver . . . 616
10.21 Ethernet driver . . . 617
10.22 class Counter – Signal counter for i.MXRT MCUs . . . 617
10.23 class Encoder – Quadrature Encoder for i.MXRT MCUs . . . 617
10.24 Transferring files . . . 618

11 Quick reference for the PSOC™ Edge **619**
11.1 General information about the PSOC™ Edge port . . . 619
11.1.1 Supported boards . . . 620
11.2 Installing MicroPython . . . 622
11.2.1 Prerequisites . . . 622
11.2.2 Flashing . . . 622
11.3 Pins and GPIO . . . 623
11.3.1 The constructor . . . 623
11.3.2 Methods . . . 625
11.3.3 Signal . . . 625
11.4 Real time clock (RTC) . . . 625
11.5 UART . . . 626
11.5.1 Constructor . . . 627
11.5.2 Methods . . . 627

12 Quick reference for the WiPy **629**
12.1 General information about the WiPy . . . 629
12.1.1 No floating point support . . . 629
12.1.2 Before applying power . . . 630
12.1.3 WLAN default behaviour . . . 630
12.1.4 Telnet REPL . . . 630
12.1.5 Local file system and FTP access . . . 630
12.1.6 FileZilla settings . . . 631
12.1.7 Upgrading the firmware Over The Air . . . 631
12.1.8 Boot modes and safe boot . . . 631
12.1.9 The heartbeat LED . . . 632
12.1.10 Details on sleep modes . . . 632
12.1.11 Additional details for machine.Pin . . . 632
12.1.12 Additional details for machine.I2C . . . 633
12.1.13 Known issues . . . 634
12.2 WiPy tutorials and examples . . . 636
12.2.1 Introduction to the WiPy . . . 636
12.2.2 Getting a MicroPython REPL prompt . . . 637
12.2.3 Getting started with Blynk and the WiPy . . . 639
12.2.4 WLAN step by step . . . 639
12.2.5 Hardware timers . . . 640
12.2.6 Reset and boot modes . . . 642
12.3 General board control (including sleep modes) . . . 643

12.4 Pins and GPIO . . . 643
12.5 Timers . . . 644
12.6 PWM (pulse width modulation) . . . 644
12.7 ADC (analog to digital conversion) . . . 644
12.8 UART (serial bus) . . . 644
12.9 SPI bus . . . 645
12.10 I2C bus . . . 645
12.11 Watchdog timer (WDT) . . . 645
12.12 Real time clock (RTC) . . . 645
12.13 SD card . . . 646
12.14 WLAN (WiFi) . . . 646
12.15 Telnet and FTP server . . . 647
12.16 Heart beat LED . . . 647

13 Quick reference for the UNIX and Windows ports 649
13.1 Command line options . . . 649
13.2 Environment variables . . . 650

14 Quick reference for the Zephyr port 651
14.1 General information about the Zephyr port . . . 651
14.1.1 Multitude of boards . . . 651
14.2 MicroPython tutorial for the Zephyr port . . . 651
14.2.1 Getting started with MicroPython on the Zephyr port . . . 651
14.2.2 Getting a MicroPython REPL prompt . . . 652
14.2.3 Filesystems and Storage . . . 653
14.2.4 GPIO Pins . . . 655
14.3 Running MicroPython . . . 656
14.4 Delay and timing . . . 656
14.5 Pins and GPIO . . . 656
14.6 PWM . . . 657
14.7 Hardware I2C bus . . . 657
14.8 Hardware SPI bus . . . 657
14.9 Analog to Digital Converter (ADC) . . . 658
14.10 Disk Access . . . 658
14.11 Flash Area . . . 659
14.12 Sensor . . . 659
14.13 Display . . . 660

15 Quick reference for the Renesas RA 663
15.1 General information about Renesas RA port . . . 663
15.1.1 Overview . . . 663
15.2 MicroPython tutorial for Renesas RA port . . . 664
15.2.1 Getting started with MicroPython on the Renesas RA . . . 664
15.2.2 Using peripherals . . . 665
15.2.3 Write a program in internal file system . . . 666
15.2.4 Reset and boot mode . . . 667
15.2.5 Trouble Shooting . . . 668
15.3 Installing MicroPython . . . 669
15.4 General board control . . . 669
15.5 Delay and timing . . . 669
15.6 Timers . . . 670
15.7 Pins and GPIO . . . 670
15.8 UART (serial bus) . . . 671
15.9 Real time clock (RTC) . . . 672

15.10 ADC (analog to digital conversion) . . . 672
15.11 SPI bus . . . 673
15.12 I2C bus . . . 674
15.13 PWM (pulse width modulation) . . . 675
15.14 WDT (Watchdog timer) . . . 675
15.15 SDCard . . . 675
15.16 OneWire driver . . . 675
15.17 NeoPixel and APA106 driver . . . 676

16 Quick reference for the SAMD21/SAMD51 family 677
16.1 General information about the SAMD port . . . 678
16.1.1 Multitude of boards . . . 678
16.1.2 Technical specifications and SoC data sheets . . . 678
16.2 Getting started with MicroPython on the SAMD . . . 679
16.2.1 Requirements . . . 679
16.2.2 Powering the board . . . 680
16.2.3 Getting the firmware . . . 680
16.2.4 Deploying the firmware . . . 680
16.2.5 Serial prompt . . . 680
16.2.6 Troubleshooting installation problems . . . 680
16.3 Pinout for the SAMD machine modules . . . 681
16.3.1 Adafruit ItsyBitsy M0 Express pin assignment table . . . 681
16.3.2 Adafruit ItsyBitsy M4 Express pin assignment table . . . 684
16.3.3 Adafruit Feather M4 Express pin assignment table . . . 688
16.3.4 Adafruit Metro M4 Airlift pin assignment table . . . 691
16.3.5 SEEED XIAO pin assignment table . . . 694
16.3.6 Adafruit Feather M0 Express pin assignment table . . . 696
16.3.7 Adafruit Trinket M0 pin assignment table . . . 698
16.3.8 Adafruit QT PY pin assignment table . . . 701
16.3.9 Adafruit NeoKey Trinkey pin assignment table . . . 702
16.3.10 SparkFun Redboard Turbo assignment table . . . 702
16.3.11 SparkFun SAMD21 Dev Breakout assignment table . . . 705
16.3.12 SAMD21 Xplained PRO pin assignment table . . . 707
16.3.13 Minisam M4 pin assignment table . . . 709
16.3.14 Seeed WIO Terminal pin assignment table . . . 711
16.3.15 SparkFun SAMD51 Thing Plus pin assignment table . . . 717
16.3.16 Generic SAMD21x18 pin assignment table . . . 720
16.3.17 Generic SAMD51x19 and SAM51x20 pin assignment table . . . 722
16.3.18 Scripts for creating the pin assignment tables . . . 727
16.4 Installing MicroPython . . . 728
16.5 General board control . . . 729
16.6 Delay and timing . . . 729
16.7 Clock and time . . . 729
16.8 Timers . . . 730
16.9 Pins and GPIO . . . 730
16.10 UART (serial bus) . . . 731
16.11 PWM (pulse width modulation) . . . 731
16.11.1 PWM Constructor . . . 731
16.11.2 PWM Methods . . . 732
16.11.3 PWM Pin Assignment . . . 732
16.12 ADC (analog to digital conversion) . . . 732
16.12.1 ADC Constructor . . . 732
16.12.2 ADC Methods . . . 733
16.13 DAC (digital to analogue conversion) . . . 734

16.13.1 DAC Constructor . . . 734
16.13.2 DAC Methods . . . 734
16.14 Software SPI bus . . . 735
16.15 Hardware SPI bus . . . 736
16.16 Software I2C bus . . . 736
16.17 Hardware I2C bus . . . 736
16.18 OneWire driver . . . 737
16.19 DHT driver . . . 737
16.20 Driving an APA102 LED . . . 738
16.21 Driving a Neopixel LED . . . 738
16.22 Transferring files . . . 739

Python Module Index **741**

CHAPTER

ONE

MICROPYTHON LIBRARIES

Warning: Important summary of this section

- MicroPython provides built-in modules that mirror the functionality of the *Python standard library* (e.g. `os`, `time`), as well as *MicroPython-specific modules* (e.g. `bluetooth`, `machine`).
- Most Python standard library modules implement a subset of the functionality of the equivalent Python module, and in a few cases provide some MicroPython-specific extensions (e.g. `array`, `os`)
- Due to resource constraints or other limitations, some ports or firmware versions may not include all the functionality documented here.
- To allow for extensibility, some built-in modules can be *extended from Python code* loaded onto the device filesystem.

This chapter describes modules (function and class libraries) which are built into MicroPython. This documentation in general aspires to describe all modules and functions/classes which are implemented in the MicroPython project. However, MicroPython is highly configurable, and each port to a particular board/embedded system may include only a subset of the available MicroPython libraries.

With that in mind, please be warned that some functions/classes in a module (or even the entire module) described in this documentation **may be unavailable** in a particular build of MicroPython on a particular system. The best place to find general information of the availability/non-availability of a particular feature is the "General Information" section which contains information pertaining to a specific *MicroPython port*.

On some ports you are able to discover the available, built-in libraries that can be imported by entering the following at the *REPL*:

```
help('modules')
```

Beyond the built-in libraries described in this documentation, many more modules from the Python standard library, as well as further MicroPython extensions to it, can be found in *micropython-lib*.

1.1 Python standard libraries and micro-libraries

The following standard Python libraries have been "micro-ified" to fit in with the philosophy of MicroPython. They provide the core functionality of that module and are intended to be a drop-in replacement for the standard Python library.

1.1.1 `array` – arrays of numeric data

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `array`.

Supported format codes: `b`, `B`, `h`, `H`, `i`, `I`, `l`, `L`, `q`, `Q`, `f`, `d` (the latter 2 depending on the floating-point support).

Classes

class array.**array**(*typecode*[, *iterable*])

Create array with elements of given type. Initial contents of the array are given by *iterable*. If it is not provided, an empty array is created.

In addition to the methods below, array objects also implement the buffer protocol. This means the contents of the entire array can be accessed as raw bytes via a *memoryview* or other interfaces which use this protocol.

append(*val*)

Append new element *val* to the end of array, growing it.

extend(*iterable*)

Append new elements as contained in *iterable* to the end of array, growing it.

__getitem__(*index*)

Indexed read of the array, called as `a[index]` (where `a` is an `array`). Returns a value if *index* is an `int` and an `array` if *index* is a slice. Negative indices count from the end and `IndexError` is thrown if the index is out of range.

Note: `__getitem__` cannot be called directly (`a.__getitem__(index)` fails) and is not present in `__dict__`, however `a[index]` does work.

__setitem__(*index*, *value*)

Indexed write into the array, called as `a[index] = value` (where `a` is an `array`). `value` is a single value if *index* is an `int` and an `array` if *index* is a slice. Negative indices count from the end and `IndexError` is thrown if the index is out of range.

Note: `__setitem__` cannot be called directly (`a.__setitem__(index, value)` fails) and is not present in `__dict__`, however `a[index] = value` does work.

__len__()

Returns the number of items in the array, called as `len(a)` (where `a` is an `array`).

Note: `__len__` cannot be called directly (`a.__len__()` fails) and the method is not present in `__dict__`, however `len(a)` does work.

__add__(*other*)

Return a new `array` that is the concatenation of the array with *other*, called as `a + other` (where `a` and *other* are both `arrays`).

Note: `__add__` cannot be called directly (`a.__add__(other)` fails) and is not present in `__dict__`, however `a + other` does work.

__iadd__(*other*)

Concatenates the array with *other* in-place, called as `a += other` (where `a` and *other* are both `arrays`). Equivalent to `extend(other)`.

Note: `__iadd__` cannot be called directly (`a.__iadd__(other)` fails) and is not present in `__dict__`, however `a += other` does work.

__repr__()

Returns the string representation of the array, called as `str(a)` or `repr(a)` ` (where `a` is an `array`). Returns the string `"array(<type>, [<elements>])"`, where `<type>` is the type code letter for the array and `<elements>` is a comma separated list of the elements of the array.

Note: `__repr__` cannot be called directly (`a.__repr__()` fails) and is not present in `__dict__`, however `str(a)` and `repr(a)` both work.

1.1.2 `asyncio` — asynchronous I/O scheduler

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: asyncio

Example:

```
import asyncio

async def blink(led, period_ms):
    while True:
        led.on()
        await asyncio.sleep_ms(5)
        led.off()
        await asyncio.sleep_ms(period_ms)

async def main(led1, led2):
    asyncio.create_task(blink(led1, 700))
    asyncio.create_task(blink(led2, 400))
    await asyncio.sleep_ms(10_000)

# Running on a pyboard
from pyb import LED
asyncio.run(main(LED(1), LED(2)))

# Running on a generic board
from machine import Pin
asyncio.run(main(Pin(1), Pin(2)))
```

Core functions

`asyncio.`**create_task**(*coro*)

Create a new task from the given coroutine and schedule it to run.

Returns the corresponding *Task* object.

`asyncio.`**current_task**()

Return the *Task* object associated with the currently running task.

asyncio.**run**(*coro*)

Create a new task from the given coroutine and run it until it completes.

Returns the value returned by *coro*.

asyncio.**sleep**(*t*)

Sleep for *t* seconds (can be a float).

This is a coroutine.

asyncio.**sleep_ms**(*t*)

Sleep for *t* milliseconds.

This is a coroutine, and a MicroPython extension.

Additional functions

asyncio.**wait_for**(*awaitable*, *timeout*)

Wait for the *awaitable* to complete, but cancel it if it takes longer than *timeout* seconds. If *awaitable* is not a task then a task will be created from it.

If a timeout occurs, it cancels the task and raises `asyncio.TimeoutError`: this should be trapped by the caller. The task receives `asyncio.CancelledError` which may be ignored or trapped using `try...except` or `try...finally` to run cleanup code.

Returns the return value of *awaitable*.

This is a coroutine.

asyncio.**wait_for_ms**(*awaitable*, *timeout*)

Similar to *wait_for* but *timeout* is an integer in milliseconds.

This is a coroutine, and a MicroPython extension.

asyncio.**gather**(**awaitables*, *return_exceptions=False*)

Run all *awaitables* concurrently. Any *awaitables* that are not tasks are promoted to tasks.

Returns a list of return values of all *awaitables*.

This is a coroutine.

class Task

class asyncio.**Task**

This object wraps a coroutine into a running task. Tasks can be waited on using `await task`, which will wait for the task to complete and return the return value of the task.

Tasks should not be created directly, rather use *create_task* to create them.

Task.**cancel**()

Cancel the task by injecting `asyncio.CancelledError` into it. The task may ignore this exception. Cleanup code may be run by trapping it, or via `try ... finally`.

class Event

class asyncio.**Event**

Create a new event which can be used to synchronise tasks. Events start in the cleared state.

Event.**is_set**()

Returns `True` if the event is set, `False` otherwise.

Event.**set**()

Set the event. Any tasks waiting on the event will be scheduled to run.

Note: This must be called from within a task. It is not safe to call this from an IRQ, scheduler callback, or other thread. See *ThreadSafeFlag*.

Event.**clear**()

Clear the event.

Event.**wait**()

Wait for the event to be set. If the event is already set then it returns immediately.

This is a coroutine.

class ThreadSafeFlag

class asyncio.**ThreadSafeFlag**

Create a new flag which can be used to synchronise a task with code running outside the asyncio loop, such as other threads, IRQs, or scheduler callbacks. Flags start in the cleared state.

ThreadSafeFlag.**set**()

Set the flag. If there is a task waiting on the flag, it will be scheduled to run.

ThreadSafeFlag.**clear**()

Clear the flag. This may be used to ensure that a possibly previously-set flag is clear before waiting for it.

ThreadSafeFlag.**wait**()

Wait for the flag to be set. If the flag is already set then it returns immediately. The flag is automatically reset upon return from `wait`.

A flag may only be waited on by a single task at a time.

This is a coroutine.

class Lock

class asyncio.**Lock**

Create a new lock which can be used to coordinate tasks. Locks start in the unlocked state.

In addition to the methods below, locks can be used in an `async with` statement.

Lock.**locked**()

Returns `True` if the lock is locked, otherwise `False`.

Lock.**acquire**()

Wait for the lock to be in the unlocked state and then lock it in an atomic way. Only one task can acquire the lock at any one time.

This is a coroutine.

Lock.**release**()

Release the lock. If any tasks are waiting on the lock then the next one in the queue is scheduled to run and the lock remains locked. Otherwise, no tasks are waiting an the lock becomes unlocked.

TCP stream connections

asyncio.**open_connection**(*host*, *port*, *ssl=None*)

Open a TCP connection to the given *host* and *port*. The *host* address will be resolved using *socket.getaddrinfo*, which is currently a blocking call. If *ssl* is a *ssl.SSLContext* object, this context is used to create the transport; if *ssl* is True, a default context is used.

Returns a pair of streams: a reader and a writer stream. Will raise a socket-specific OSError if the host could not be resolved or if the connection could not be made.

This is a coroutine.

asyncio.**start_server**(*callback*, *host*, *port*, *backlog=5*, *ssl=None*)

Start a TCP server on the given *host* and *port*. The *callback* will be called with incoming, accepted connections, and be passed 2 arguments: reader and writer streams for the connection.

If *ssl* is a *ssl.SSLContext* object, this context is used to create the transport.

Returns a *Server* object.

This is a coroutine.

class asyncio.**Stream**

This represents a TCP stream connection. To minimise code this class implements both a reader and a writer, and both StreamReader and StreamWriter alias to this class.

Stream.**get_extra_info**(*v*)

Get extra information about the stream, given by *v*. The valid values for *v* are: peername.

Stream.**close**()

Close the stream.

Stream.**wait_closed**()

Wait for the stream to close.

This is a coroutine.

Stream.**read**(*n=-1*)

Read up to *n* bytes and return them. If *n* is not provided or -1 then read all bytes until EOF. The returned value will be an empty bytes object if EOF is encountered before any bytes are read.

This is a coroutine.

Stream.**readinto**(*buf*)

Read up to n bytes into *buf* with n being equal to the length of *buf*.

Return the number of bytes read into *buf*.

This is a coroutine, and a MicroPython extension.

Stream.**readexactly**(*n*)

Read exactly *n* bytes and return them as a bytes object.

Raises an EOFError exception if the stream ends before reading *n* bytes.

This is a coroutine.

`Stream.`**`readline`**`()`

Read a line and return it.

This is a coroutine.

`Stream.`**`write`**`(`*buf*`)`

Accumulated *buf* to the output buffer. The data is only flushed when *Stream.drain* is called. It is recommended to call *Stream.drain* immediately after calling this function.

`Stream.`**`drain`**`()`

Drain (write) all buffered output data out to the stream.

This is a coroutine.

`class` `asyncio.`**`Server`**

This represents the server class returned from *start_server*. It can be used in an `async with` statement to close the server upon exit.

`Server.`**`close`**`()`

Close the server.

`Server.`**`wait_closed`**`()`

Wait for the server to close.

This is a coroutine.

Event Loop

`asyncio.`**`get_event_loop`**`()`

Return the event loop used to schedule and run tasks. See *Loop*.

`asyncio.`**`new_event_loop`**`()`

Reset the event loop and return it.

Note: since MicroPython only has a single event loop this function just resets the loop's state, it does not create a new one.

`class` `asyncio.`**`Loop`**

This represents the object which schedules and runs tasks. It cannot be created, use *get_event_loop* instead.

`Loop.`**`create_task`**`(`*coro*`)`

Create a task from the given *coro* and return the new *Task* object.

`Loop.`**`run_forever`**`()`

Run the event loop until *stop()* is called.

`Loop.`**`run_until_complete`**`(`*awaitable*`)`

Run the given *awaitable* until it completes. If *awaitable* is not a task then it will be promoted to one.

`Loop.`**`stop`**`()`

Stop the event loop.

`Loop.`**`close`**`()`

Close the event loop.

`Loop.`**`set_exception_handler`**`(`*handler*`)`

Set the exception handler to call when a Task raises an exception that is not caught. The *handler* should accept two arguments: `(loop, context)`.

Loop.**get_exception_handler**()

Get the current exception handler. Returns the handler, or None if no custom handler is set.

Loop.**default_exception_handler**(*context*)

The default exception handler that is called.

Loop.**call_exception_handler**(*context*)

Call the current exception handler. The argument *context* is passed through and is a dictionary containing keys: 'message', 'exception', 'future'.

1.1.3 binascii – binary/ASCII conversions

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: binascii.

This module implements conversions between binary data and various encodings of it in ASCII form (in both directions).

Functions

binascii.**hexlify**(*data*[, *sep*])

Convert the bytes in the *data* object to a hexadecimal representation. Returns a bytes object.

If the additional argument *sep* is supplied it is used as a separator between hexadecimal values.

binascii.**unhexlify**(*data*)

Convert hexadecimal data to binary representation. Returns bytes string. (i.e. inverse of hexlify)

binascii.**a2b_base64**(*data*)

Decode base64-encoded data, ignoring invalid characters in the input. Conforms to RFC 2045 s.6.8. Returns a bytes object.

binascii.**b2a_base64**(*data*, *, *newline=True*)

Encode binary data in base64 format, as in RFC 3548. Returns the encoded data followed by a newline character if newline is true, as a bytes object.

binascii.**crc32**(*data*[, *value*])

Compute CRC-32, the 32-bit checksum of *data*, starting with an initial CRC of *value*. The default initial CRC is zero. The algorithm is consistent with the ZIP file checksum.

1.1.4 builtins – builtin functions and exceptions

All builtin functions and exceptions are described here. They are also available via builtins module.

Functions and types

abs()

all()

any()

bin()

class bool

class bytearray

See CPython documentation: `bytearray`.

class bytes

See CPython documentation: `bytes`.

decode(*encoding='utf-8'*, *errors='strict'*)

Decode the bytes object to a string using the specified *encoding*.

MicroPython supports the following encodings:

- `'utf-8'` or `'utf8'` - UTF-8 encoding (default)
- `'ascii'` - ASCII encoding (subset of UTF-8)

The *errors* parameter controls how decoding errors are handled:

- `'strict'` - Raise a `UnicodeError` on invalid UTF-8 (default)
- `'ignore'` - Skip invalid bytes (requires `MICROPY_PY_BUILTINS_BYTES_DECODE_ERRORS`)
- `'replace'` - Replace invalid bytes with U+FFFD '' (requires `MICROPY_PY_BUILTINS_BYTES_DECODE_ERRORS`)

Note: Error handler support depends on build configuration. On constrained systems, only `'strict'` mode may be available.

Example:

```
>>> b'\xc2\xa9 2024'.decode('utf-8')  # © symbol
'© 2024'
>>> b'hello\xffworld'.decode('utf-8', 'ignore')  # Skip invalid bytes
'helloworld'
```

Raises `LookupError` if the encoding is not supported, or `UnicodeError` if the data contains invalid UTF-8 and `errors='strict'`.

callable()

chr()

classmethod()

compile()

class complex

delattr(*obj*, *name*)

The argument *name* should be a string, and this function deletes the named attribute from the object given by *obj*.

class dict

dir()

divmod()

enumerate()

eval()

exec()

filter()

class float

class frozenset

getattr()

globals()

hasattr()

hash()

hex()

id()

input()

class int

classmethod from_bytes(*bytes*, *byteorder*)

In MicroPython, `byteorder` parameter must be positional (this is compatible with CPython).

to_bytes(*size*, *byteorder*, */*, ***, *signed=False*)

In MicroPython, `byteorder` parameter must be positional (this is compatible with CPython).

isinstance()

issubclass()

iter()

len()

class list

locals()

map()

max()

class memoryview

See CPython documentation: `memoryview`.

min()

next()

class object

oct()

open()

ord()

pow()

print()

property()

range()

repr()

reversed()

round()

class set

setattr()

class slice

The *slice* builtin is the type that slice objects have.

sorted()

staticmethod()

class str

encode(*encoding='utf-8'*)

Encode the string to bytes using the specified *encoding*.

MicroPython supports the following encodings:

- `'utf-8'` or `'utf8'` - UTF-8 encoding (default)
- `'ascii'` - ASCII encoding (subset of UTF-8)

Example:

```
>>> '© 2024'.encode('utf-8')  # Copyright symbol
b'\xc2\xa9 2024'
```

Raises `LookupError` if the encoding is not supported.

center(*width*)

Return a centered string of length *width*. Padding is done using spaces.

When Unicode support is enabled (`MICROPY_PY_BUILTINS_STR_UNICODE`), this method counts Unicode characters rather than bytes, ensuring proper alignment for multi-byte UTF-8 characters.

Example:

```
>>> 'café'.center(10)  # é is 2 bytes in UTF-8
'   café   '
```

sum()

super()

class tuple

type()

zip()

Exceptions

exception AssertionError

exception AttributeError

exception Exception

exception ImportError

exception IndexError

exception KeyboardInterrupt

See CPython documentation: `KeyboardInterrupt`.

See also in the context of *Soft Bricking (failure to boot)*.

exception KeyError

exception MemoryError

exception NameError

exception NotImplementedError

exception OSError

exception RuntimeError

exception StopIteration

exception SyntaxError

exception SystemExit

See CPython documentation: `SystemExit`.

On non-embedded ports (i.e. Windows and Unix), an unhandled `SystemExit` exits the MicroPython process in a similar way to CPython.

On embedded ports, an unhandled `SystemExit` currently causes a *Soft Reset* of MicroPython.

exception TypeError

See CPython documentation: `TypeError`.

exception ValueError

exception ZeroDivisionError

1.1.5 `cmath` – mathematical functions for complex numbers

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `cmath`.

The `cmath` module provides some basic mathematical functions for working with complex numbers.

Availability: not available on WiPy and ESP8266. Floating point support required for this module.

Functions

`cmath.`**`cos`**`(z)`
Return the cosine of `z`.

`cmath.`**`exp`**`(z)`
Return the exponential of `z`.

`cmath.`**`log`**`(z)`
Return the natural logarithm of `z`. The branch cut is along the negative real axis.

`cmath.`**`log10`**`(z)`
Return the base-10 logarithm of `z`. The branch cut is along the negative real axis.

`cmath.`**`phase`**`(z)`
Returns the phase of the number `z`, in the range (-pi, +pi].

`cmath.`**`polar`**`(z)`
Returns, as a tuple, the polar form of `z`.

`cmath.`**`rect`**`(r, phi)`
Returns the complex number with modulus `r` and phase `phi`.

`cmath.`**`sin`**`(z)`
Return the sine of `z`.

`cmath.`**`sqrt`**`(z)`
Return the square-root of `z`.

Constants

`cmath.`**`e`**
base of the natural logarithm

`cmath.`**`pi`**
the ratio of a circle's circumference to its diameter

1.1.6 `collections` – collection and container types

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `collections`.

This module implements advanced collection and container types to hold/accumulate various objects.

Classes

class collections.**deque**(*iterable*, *maxlen*[, *flags*])

Deques (double-ended queues) are a list-like container that support O(1) appends and pops from either side of the deque. New deques are created using the following arguments:

- *iterable* is an iterable used to populate the deque when it is created. It can be an empty tuple or list to create a deque that is initially empty.
- *maxlen* must be specified and the deque will be bounded to this maximum length. Once the deque is full, any new items added will discard items from the opposite end.
- The optional *flags* can be 1 to check for overflow when adding items.

Deque objects support *bool*, *len*, iteration and subscript load and store. They also have the following methods:

append(*x*)

Add *x* to the right side of the deque. Raises `IndexError` if overflow checking is enabled and there is no more room in the queue.

appendleft(*x*)

Add *x* to the left side of the deque. Raises `IndexError` if overflow checking is enabled and there is no more room in the queue.

pop()

Remove and return an item from the right side of the deque. Raises `IndexError` if no items are present.

popleft()

Remove and return an item from the left side of the deque. Raises `IndexError` if no items are present.

extend(*iterable*)

Extend the deque by appending all the items from *iterable* to the right of the deque. Raises `IndexError` if overflow checking is enabled and there is no more room in the deque.

collections.**namedtuple**(*name*, *fields*)

This is factory function to create a new namedtuple type with a specific name and set of fields. A namedtuple is a subclass of tuple which allows to access its fields not just by numeric index, but also with an attribute access syntax using symbolic field names. Fields is a sequence of strings specifying field names. For compatibility with CPython it can also be a a string with space-separated field named (but this is less efficient). Example of use:

```
from collections import namedtuple

MyTuple = namedtuple("MyTuple", ("id", "name"))
t1 = MyTuple(1, "foo")
t2 = MyTuple(2, "bar")
print(t1.name)
assert t2.name == t2[1]
```

class collections.**OrderedDict**(...)

dict type subclass which remembers and preserves the order of keys added. When ordered dict is iterated over, keys/items are returned in the order they were added:

```
from collections import OrderedDict

# To make benefit of ordered keys, OrderedDict should be initialized
# from sequence of (key, value) pairs.
d = OrderedDict([("z", 1), ("a", 2)])
# More items can be added as usual
d["w"] = 5
d["b"] = 3
for k, v in d.items():
    print(k, v)
```

Output:

```
z 1
a 2
w 5
b 3
```

popitem()

Remove and return a (key, value) pair from the dictionary. Pairs are returned in LIFO order.

Difference to CPython

`OrderedDict.popitem()` does not support the `last=False` argument and will always remove and return the last item if present.

A workaround for this is to use `pop(<first_key>)` to remove the first item:

```
first_key = next(iter(d))
d.pop(first_key)
```

1.1.7 errno – system error codes

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `errno`.

This module provides access to symbolic error codes for *OSError* exception. A particular inventory of codes depends on *MicroPython port*.

Constants

EEXIST, EAGAIN, etc.

Error codes, based on ANSI C/POSIX standard. All error codes start with "E". As mentioned above, inventory of the codes depends on *MicroPython port*. Errors are usually accessible as `exc.errno` where `exc` is an instance of *OSError*. Usage example:

```
try:
    os.mkdir("my_dir")
except OSError as exc:
```

(continues on next page)

(continued from previous page)

```
    if exc.errno == errno.EEXIST:
        print("Directory already exists")
```

errno.**errorcode**

Dictionary mapping numeric error codes to strings with symbolic error code (see above):

```
>>> print(errno.errorcode[errno.EEXIST])
EEXIST
```

1.1.8 gc – control the garbage collector

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: gc.

Functions

gc.**enable**()

Enable automatic garbage collection.

gc.**disable**()

Disable automatic garbage collection. Heap memory can still be allocated, and garbage collection can still be initiated manually using *gc.collect()*.

gc.**isenabled**()

Returns True if automatic garbage collection is enabled, and False otherwise.

gc.**collect**()

Run a garbage collection.

gc.**mem_alloc**()

Return the number of bytes of heap RAM that are allocated by Python code.

Difference to CPython

This function is MicroPython extension.

---

gc.**mem_free**()

Return the number of bytes of heap RAM that is available for Python code to allocate, or -1 if this amount is not known.

Difference to CPython

This function is MicroPython extension.

---

gc.**threshold**([*amount*])

Set or query the additional GC allocation threshold. Normally, a collection is triggered only when a new allocation cannot be satisfied, i.e. on an out-of-memory (OOM) condition. If this function is called, in addition to OOM, a collection will be triggered each time after *amount* bytes have been allocated (in total, since the previous time such an amount of bytes have been allocated). *amount* is usually specified as less than the full heap size, with the intention to trigger a collection earlier than when the heap becomes exhausted, and in the hope that an early

collection will prevent excessive memory fragmentation. This is a heuristic measure, the effect of which will vary from application to application, as well as the optimal value of the *amount* parameter.

Calling the function without argument will return the current value of the threshold. A value of -1 means a disabled allocation threshold.

Difference to CPython

This function is a MicroPython extension. CPython has a similar function - `set_threshold()`, but due to different GC implementations, its signature and semantics are different.

Examples

To trigger a garbage collection each time 32768 bytes of RAM have been allocated in total:

```
gc.threshold(32768)
```

To restore the default behaviour, only triggering garbage collection when out of memory:

```
gc.threshold(-1)
```

Example

```
>>> import gc
>>> gc.mem_free() # Gets number of bytes free in memory
8192
>>> gc.mem_alloc() # Gets number of bytes allocated in memory
1024
>>> foo = bytearray(1000) # Create a big array of data
>>> gc.mem_free() # Show that there's less memory available
7168
>>> gc.mem_alloc() # Show that there's more memory used
2048
>>> del foo # Delete the object
>>> gc.mem_free() # Show that collection hasn't run yet
7168
>>> gc.mem_alloc() # That memory is still allocated
2048
>>> gc.collect() # Manually run the collection
>>> gc.mem_free() # Now we have reclaimed that memory
8192
>>> gc.mem_alloc() # That memory is no longer allocated
1024
```

1.1.9 `gzip` – gzip compression & decompression

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `gzip`.

This module allows compression and decompression of binary data with the DEFLATE algorithm used by the gzip file format.

Note: Prefer to use `deflate.DeflateIO` instead of the functions in this module as it provides a streaming interface to compression and decompression which is convenient and more memory efficient when working with reading or writing compressed data to a file, socket, or stream.

Availability:

- This module is **not present by default** in official MicroPython firmware releases as it duplicates functionality available in the `deflate` module.
- A copy of this module can be installed (or frozen) from *micropython-lib* (source). See *Package management* for more information. This documentation describes that module.
- Compression support will only be available if compression support is enabled in the built-in `deflate` module.

Functions

`gzip.`**`open`**`(`*filename*, *mode*, `/)`

Wrapper around built-in `open()` returning a GzipFile instance.

`gzip.`**`decompress`**`(`*data*, `/)`

Decompresses *data* into a bytes object.

`gzip.`**`compress`**`(`*data*, `/)`

Compresses *data* into a bytes object.

Classes

`class` `gzip.`**`GzipFile`**`(*, `*fileobj*, *mode*`)`

This class can be used to wrap a *fileobj* which is any *stream-like* object such as a file, socket, or stream (including `io.BytesIO`). It is itself a stream and implements the standard read/readinto/write/close methods.

When the *mode* argument is `"rb"`, reads from the GzipFile instance will decompress the data in the underlying stream and return decompressed data.

If compression support is enabled then the *mode* argument can be set to `"wb"`, and writes to the GzipFile instance will be compressed and written to the underlying stream.

By default the GzipFile class will read and write data using the gzip file format, including a header and footer with checksum and a window size of 512 bytes.

The **file**, **compresslevel**, and **mtime** arguments are not supported. **fileobj** and **mode** must always be specified as keyword arguments.

Examples

A typical use case for *gzip.GzipFile* is to read or write a compressed file from storage:

```
import gzip

# Reading:
with open("data.gz", "rb") as f:
    with gzip.GzipFile(fileobj=f, mode="rb") as g:
        # Use g.read(), g.readinto(), etc.

 # Same, but using gzip.open:
with gzip.open("data.gz", "rb") as f:
     # Use f.read(), f.readinto(), etc.

# Writing:
with open("data.gz", "wb") as f:
    with gzip.GzipFile(fileobj=f, mode="wb") as g:
        # Use g.write(...) etc

# Same, but using gzip.open:
with gzip.open("data.gz", "wb") as f:
    # Use f.write(...) etc

# Write a dictionary as JSON in gzip format, with a
# small (64 byte) window size.
config = { ... }
with gzip.open("config.gz", "wb") as f:
    json.dump(config, f)
```

For guidance on working with gzip sources and choosing the window size see the note at the *end of the deflate documentation*.

1.1.10 `hashlib` – hashing algorithms

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `hashlib`.

This module implements binary data hashing algorithms. The exact inventory of available algorithms depends on a board. Among the algorithms which may be implemented:

- SHA256 - The current generation, modern hashing algorithm (of SHA2 series). It is suitable for cryptographically-secure purposes. Included in the MicroPython core and any board is recommended to provide this, unless it has particular code size constraints.
- SHA1 - A previous generation algorithm. Not recommended for new usages, but SHA1 is a part of number of Internet standards and existing applications, so boards targeting network connectivity and interoperability will try to provide this.
- MD5 - A legacy algorithm, not considered cryptographically secure. Only selected boards, targeting interoperability with legacy applications, will offer this.

Constructors

class hashlib.**sha256**([*data*])

Create an SHA256 hasher object and optionally feed `data` into it.

class hashlib.**sha1**([*data*])

Create an SHA1 hasher object and optionally feed `data` into it.

class hashlib.**md5**([*data*])

Create an MD5 hasher object and optionally feed `data` into it.

Methods

hash.**update**(*data*)

Feed more binary data into hash.

hash.**digest**()

Return hash for all data passed through hash, as a bytes object. After this method is called, more data cannot be fed into the hash any longer.

hash.**hexdigest**()

This method is NOT implemented. Use `binascii.hexlify(hash.digest())` to achieve a similar effect.

1.1.11 heapq – heap queue algorithm

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `heapq`.

This module implements the min heap queue algorithm.

A heap queue is essentially a list that has its elements stored in such a way that the first item of the list is always the smallest.

Functions

heapq.**heappush**(*heap*, *item*)

Push the `item` onto the `heap`.

heapq.**heappop**(*heap*)

Pop the first item from the `heap`, and return it. Raise `IndexError` if `heap` is empty.

The returned item will be the smallest item in the `heap`.

heapq.**heapify**(*x*)

Convert the list `x` into a heap. This is an in-place operation.

1.1.12 `io` – input/output streams

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `io`.

This module contains additional types of *`stream`* (file-like) objects and helper functions.

Conceptual hierarchy

Difference to CPython

Conceptual hierarchy of stream base classes is simplified in MicroPython, as described in this section.

(Abstract) base stream classes, which serve as a foundation for behaviour of all the concrete classes, adhere to few dichotomies (pair-wise classifications) in CPython. In MicroPython, they are somewhat simplified and made implicit to achieve higher efficiencies and save resources.

An important dichotomy in CPython is unbuffered vs buffered streams. In MicroPython, all streams are currently unbuffered. This is because all modern OSes, and even many RTOSes and filesystem drivers already perform buffering on their side. Adding another layer of buffering is counter- productive (an issue known as "bufferbloat") and takes precious memory. Note that there still cases where buffering may be useful, so we may introduce optional buffering support at a later time.

But in CPython, another important dichotomy is tied with "bufferedness" - it's whether a stream may incur short read/writes or not. A short read is when a user asks e.g. 10 bytes from a stream, but gets less, similarly for writes. In CPython, unbuffered streams are automatically short operation susceptible, while buffered are guarantee against them. The no short read/writes is an important trait, as it allows to develop more concise and efficient programs - something which is highly desirable for MicroPython. So, while MicroPython doesn't support buffered streams, it still provides for no-short-operations streams. Whether there will be short operations or not depends on each particular class' needs, but developers are strongly advised to favour no-short-operations behaviour for the reasons stated above. For example, MicroPython sockets are guaranteed to avoid short read/writes. Actually, at this time, there is no example of a short-operations stream class in the core, and one would be a port-specific class, where such a need is governed by hardware peculiarities.

The no-short-operations behaviour gets tricky in case of non-blocking streams, blocking vs non-blocking behaviour being another CPython dichotomy, fully supported by MicroPython. Non-blocking streams never wait for data either to arrive or be written - they read/write whatever possible, or signal lack of data (or ability to write data). Clearly, this conflicts with "no-short-operations" policy, and indeed, a case of non-blocking buffered (and this no-short-ops) streams is convoluted in CPython - in some places, such combination is prohibited, in some it's undefined or just not documented, in some cases it raises verbose exceptions. The matter is much simpler in MicroPython: non-blocking stream are important for efficient asynchronous operations, so this property prevails on the "no-short-ops" one. So, while blocking streams will avoid short reads/writes whenever possible (the only case to get a short read is if end of file is reached, or in case of error (but errors don't return short data, but raise exceptions)), non-blocking streams may produce short data to avoid blocking the operation.

The final dichotomy is binary vs text streams. MicroPython of course supports these, but while in CPython text streams are inherently buffered, they aren't in MicroPython. (Indeed, that's one of the cases for which we may introduce buffering support.)

Note that for efficiency, MicroPython doesn't provide abstract base classes corresponding to the hierarchy above. However, the *`IOBase`* class can be subclassed to implement custom stream objects in pure Python.

Functions

io.**open**(*name*, *mode='r'*, ***kwargs*)

Open a file. Builtin `open()` function is aliased to this function. All ports (which provide access to file system) are required to support *mode* parameter, but support for other arguments vary by port.

Classes

class io.**IOBase**

Base class for implementing custom stream objects in Python. Subclasses can override `readinto`, `write`, and `ioctl` to create objects that work with `print()`, `json.dump()`, `select.poll()`, `open()` via a user filesystem, and other stream consumers.

Difference to CPython

In CPython, `io.IOBase` has a much larger API surface. MicroPython's `IOBase` is minimal: the C stream infrastructure provides standard methods like `read()`, `readline()`, `seek()`, `close()`, and `flush()` automatically. Subclasses only need to implement the methods below.

Subclasses implement some or all of the following methods. The C stream layer calls these internally when user code calls standard stream functions.

readinto(*buf*)

Read data into *buf* (a `bytearray` sized by the caller). Return the number of bytes read, 0 at EOF, or `None` if no data is available on a non-blocking stream. Return a negative errno value (e.g. `-errno.EIO` or `-1`) to signal an error.

write(*buf*)

Write *buf* (a `bytearray`) to the stream. Return the number of bytes written, or `None` if a non-blocking stream cannot accept data. Return a negative errno value to signal an error.

ioctl(*op*, *arg*)

Control the stream and query its properties. The operation to perform is given by *op* which is one of the following integers:

- 1 – flush write buffers (*arg* is unused)
- 3 – poll for readiness; *arg* is a bitmask of events to check, return a bitmask of ready events. Poll flags:
  - `0x0001` – data available for reading
  - `0x0004` – stream ready for writing
  - `0x0008` – error condition
  - `0x0010` – hang up (e.g. connection closed)
  - `0x0020` – invalid request
- 4 – close the stream (*arg* is unused)
- 11 – return the preferred read buffer size, or 0 (*arg* is unused)

As a minimum `ioctl(4, ...)` should be handled to support stream closure. Implement `ioctl(3, ...)` if the stream will be used with `select.poll()` or `asyncio`.

Other operations exist for advanced use cases (2 = seek, 5 = timeout, 10 = fileno); see `py/stream.h` for the full list.

Must always return an integer. Return 0 for success, or `-1` for unsupported operations. (Returning 0 for an unhandled operation tells the C layer the operation was processed successfully, which may cause incorrect behaviour.)

class io.**StringIO**([*string*])

class io.**BytesIO**([*string*])

In-memory file-like objects for input/output. *StringIO* is used for text-mode I/O (similar to a normal file opened with "t" modifier). *BytesIO* is used for binary-mode I/O (similar to a normal file opened with "b" modifier). Initial contents of file-like objects can be specified with *string* parameter (should be normal string for *StringIO* or bytes object for *BytesIO*). All the usual file methods like `read()`, `write()`, `seek()`, `flush()`, `close()` are available on these objects, and additionally, a following method:

getvalue()

Get the current contents of the underlying buffer which holds data.

class io.**StringIO**(*alloc_size*)

class io.**BytesIO**(*alloc_size*)

Create an empty *StringIO*/*BytesIO* object, preallocated to hold up to *alloc_size* number of bytes. That means that writing that amount of bytes won't lead to reallocation of the buffer, and thus won't hit out-of-memory situation or lead to memory fragmentation. These constructors are a MicroPython extension and are recommended for usage only in special cases and in system-level libraries, not for end-user applications.

Difference to CPython

These constructors are a MicroPython extension.

IOBase Examples

A minimal write-only stream that collects output into a buffer:

```
import io

class MyOutput(io.IOBase):
    def __init__(self):
        self.data = bytearray()

    def write(self, buf):
        self.data.extend(buf)
        return len(buf)

    def ioctl(self, op, arg):
        if op == 4:  # close
            return 0
        return -1

s = MyOutput()
print("hello", file=s)
print(s.data)  # bytearray(b'hello\n')
```

A readable stream that can be used with `select.poll()`:

```
import io, select
from micropython import const

_MP_STREAM_POLL = const(3)
_MP_STREAM_POLL_RD = const(0x0001)
_MP_STREAM_CLOSE = const(4)

class RingBuffer(io.IOBase):
    def __init__(self, size):
        self._buf = bytearray(size)
        self._size = size
        self._wpos = 0
        self._rpos = 0

    def _available(self):
        return (self._wpos - self._rpos) % self._size

    def put(self, data):
        for b in data:
            self._buf[self._wpos % self._size] = b
            self._wpos = (self._wpos + 1) % self._size

    def readinto(self, buf):
        n = min(len(buf), self._available())
        for i in range(n):
            buf[i] = self._buf[self._rpos % self._size]
            self._rpos = (self._rpos + 1) % self._size
        return n

    def ioctl(self, op, arg):
        if op == _MP_STREAM_POLL:
            if arg & _MP_STREAM_POLL_RD and self._available() > 0:
                return _MP_STREAM_POLL_RD
            return 0
        if op == _MP_STREAM_CLOSE:
            return 0
        return -1

rb = RingBuffer(64)
rb.put(b"test data")

poller = select.poll()
poller.register(rb, select.POLLIN)
for obj, flags in poller.poll(0):
    buf = bytearray(16)
    n = obj.readinto(buf)
    print(buf[:n])  # bytearray(b'test data')
```

1.1.13 `json` – JSON encoding and decoding

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `json`.

This modules allows to convert between Python objects and the JSON data format.

Functions

`json.`**`dump`**(*obj*, *stream*, *separators=None*)

Serialise *obj* to a JSON string, writing it to the given *stream*.

If specified, separators should be an `(item_separator, key_separator)` tuple. The default is `(', ', ': ')`. To get the most compact JSON representation, you should specify `(',', ':')` to eliminate whitespace.

`json.`**`dumps`**(*obj*, *separators=None*)

Return *obj* represented as a JSON string.

The arguments have the same meaning as in *dump*.

`json.`**`load`**(*stream*)

Parse the given *stream*, interpreting it as a JSON string and deserialising the data to a Python object. The resulting object is returned.

Parsing continues until end-of-file is encountered. A *ValueError* is raised if the data in *stream* is not correctly formed.

`json.`**`loads`**(*str*)

Parse the JSON *str* and return an object. Raises *ValueError* if the string is not correctly formed.

1.1.14 `marshal` – Python object serialization

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `marshal`.

This module implements conversion between Python objects and a binary format. The format is specific to MicroPython but does not depend on the machine architecture, so the data can be transferred and used on a different MicroPython instance, as long as the version of the binary data matches (it's currently versioned as the mpy file version, see *MicroPython .mpy files*).

Functions

`marshal.`**`dumps`**(*value*, `/`)

Convert the given *value* to binary format and return a corresponding `bytes` object.

Currently, code objects are the only supported values that can be converted.

`marshal.`**`loads`**(*data*, `/`)

Convert the given bytes-like *data* to its corresponding Python object, and return it.

1.1.15 `math` – mathematical functions

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `math`.

The `math` module provides some basic mathematical functions for working with floating-point numbers.

Note: On the pyboard, floating-point numbers have 32-bit precision.

Availability: not available on WiPy. Floating point support required for this module.

Functions

`math.`**`acos`**(*x*)
Return the inverse cosine of `x`.

`math.`**`acosh`**(*x*)
Return the inverse hyperbolic cosine of `x`.

`math.`**`asin`**(*x*)
Return the inverse sine of `x`.

`math.`**`asinh`**(*x*)
Return the inverse hyperbolic sine of `x`.

`math.`**`atan`**(*x*)
Return the inverse tangent of `x`.

`math.`**`atan2`**(*y*, *x*)
Return the principal value of the inverse tangent of `y/x`.

`math.`**`atanh`**(*x*)
Return the inverse hyperbolic tangent of `x`.

`math.`**`ceil`**(*x*)
Return an integer, being `x` rounded towards positive infinity.

`math.`**`copysign`**(*x*, *y*)
Return `x` with the sign of `y`.

`math.`**`cos`**(*x*)
Return the cosine of `x`.

`math.`**`cosh`**(*x*)
Return the hyperbolic cosine of `x`.

`math.`**`degrees`**(*x*)
Return radians `x` converted to degrees.

`math.`**`erf`**(*x*)
Return the error function of `x`.

`math.`**`erfc`**(*x*)
Return the complementary error function of `x`.

`math.`**`exp`**(*x*)
Return the exponential of `x`.

math.**expm1**(*x*)

Return `exp(x) - 1`.

math.**fabs**(*x*)

Return the absolute value of `x`.

math.**floor**(*x*)

Return an integer, being `x` rounded towards negative infinity.

math.**fmod**(*x*, *y*)

Return the remainder of `x/y`.

math.**frexp**(*x*)

Decomposes a floating-point number into its mantissa and exponent. The returned value is the tuple `(m, e)` such that `x == m * 2**e` exactly. If `x == 0` then the function returns `(0.0, 0)`, otherwise the relation `0.5 <= abs(m) < 1` holds.

math.**gamma**(*x*)

Return the gamma function of `x`.

math.**isfinite**(*x*)

Return `True` if `x` is finite.

math.**isinf**(*x*)

Return `True` if `x` is infinite.

math.**isnan**(*x*)

Return `True` if `x` is not-a-number

math.**ldexp**(*x*, *exp*)

Return `x * (2**exp)`.

math.**lgamma**(*x*)

Return the natural logarithm of the gamma function of `x`.

math.**log**(*x*)

math.**log**(*x*, *base*)

With one argument, return the natural logarithm of *x*.

With two arguments, return the logarithm of *x* to the given *base*.

math.**log10**(*x*)

Return the base-10 logarithm of `x`.

math.**log2**(*x*)

Return the base-2 logarithm of `x`.

math.**modf**(*x*)

Return a tuple of two floats, being the fractional and integral parts of `x`. Both return values have the same sign as `x`.

math.**pow**(*x*, *y*)

Returns `x` to the power of `y`.

math.**radians**(*x*)

Return degrees `x` converted to radians.

math.**sin**(*x*)
: Return the sine of x.

math.**sinh**(*x*)
: Return the hyperbolic sine of x.

math.**sqrt**(*x*)
: Return the square root of x.

math.**tan**(*x*)
: Return the tangent of x.

math.**tanh**(*x*)
: Return the hyperbolic tangent of x.

math.**trunc**(*x*)
: Return an integer, being x rounded towards 0.

Constants

math.**e**
: base of the natural logarithm

math.**pi**
: the ratio of a circle's circumference to its diameter

1.1.16 os – basic "operating system" services

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: os.

The os module contains functions for filesystem access and mounting, terminal redirection and duplication, and the uname and urandom functions.

General functions

os.**uname**()
: Return a tuple (possibly a named tuple) containing information about the underlying machine and/or its operating system. The tuple has five fields in the following order, each of them being a string:

  - sysname – the name of the underlying system
  - nodename – the network name (can be the same as sysname)
  - release – the version of the underlying system
  - version – the MicroPython version and build date
  - machine – an identifier for the underlying hardware (eg board, CPU)

os.**urandom**(*n*)
: Return a bytes object with *n* random bytes. Whenever possible, it is generated by the hardware random number generator.

Filesystem access

os.**chdir**(*path*)

Change current directory.

os.**getcwd**()

Get the current directory.

os.**ilistdir**([*dir*])

This function returns an iterator which then yields tuples corresponding to the entries in the directory that it is listing. With no argument it lists the current directory, otherwise it lists the directory given by *dir*.

The tuples have the form *(name, type, inode[, size])*:

- *name* is a string (or bytes if *dir* is a bytes object) and is the name of the entry;
- *type* is an integer that specifies the type of the entry, with 0x4000 for directories and 0x8000 for regular files;
- *inode* is an integer corresponding to the inode of the file, and may be 0 for filesystems that don't have such a notion.
- *size* is an integer that may be included depending on the filesystem type. For file entries, *size* represents the size of the file or -1 if unknown. Its meaning is currently undefined for directory entries.

os.**listdir**([*dir*])

With no argument, list the current directory. Otherwise list the given directory.

os.**mkdir**(*path*)

Create a new directory.

os.**remove**(*path*)

Remove a file.

os.**rmdir**(*path*)

Remove a directory.

os.**rename**(*old_path*, *new_path*)

Rename a file.

os.**stat**(*path*)

Get the status of a file or directory.

os.**statvfs**(*path*)

Get the status of a filesystem.

Returns a tuple with the filesystem information in the following order:

- `f_bsize` – file system block size
- `f_frsize` – fragment size
- `f_blocks` – size of fs in f_frsize units
- `f_bfree` – number of free blocks
- `f_bavail` – number of free blocks for unprivileged users
- `f_files` – number of inodes
- `f_ffree` – number of free inodes
- `f_favail` – number of free inodes for unprivileged users
- `f_flag` – mount flags

- `f_namemax` – maximum filename length

Parameters related to inodes: `f_files`, `f_ffree`, `f_avail` and the `f_flags` parameter may return `0` as they can be unavailable in a port-specific implementation.

`os.`**`sync`**`()`

Sync all filesystems. On some ports this function isn't present because it isn't necessary to sync after writes to the file-system.

Terminal redirection and duplication

`os.`**`dupterm`**`(`*stream_object*, *index=0*, `/)`

Duplicate or switch the MicroPython terminal (the REPL) on the given *stream*-like object. The *stream_object* argument must be a native stream object, or derive from `io.IOBase` and implement the `readinto()` and `write()` methods. The stream should be in non-blocking mode and `readinto()` should return `None` if there is no data available for reading.

After calling this function all terminal output is repeated on this stream, and any input that is available on the stream is passed on to the terminal input.

The *index* parameter should be a non-negative integer and specifies which duplication slot is set. A given port may implement more than one slot (slot 0 will always be available) and in that case terminal input and output is duplicated on all the slots that are set.

If `None` is passed as the *stream_object* then duplication is cancelled on the slot given by *index*.

The function returns the previous stream-like object in the given slot.

`os.`**`dupterm_notify`**`(`*obj_in*, `/)`

Notify the MicroPython REPL that input is available on a stream-like object previously registered via *os.dupterm()*.

This function should be called by custom stream implementations (e.g., UART, Bluetooth, or other non-USB REPL streams) to inform the REPL that input is ready to be read. Proper use ensures that special characters such as Ctrl+C (used to trigger KeyboardInterrupt) are processed promptly by the REPL, enabling expected interruption behavior for user code.

The *obj_in* parameter is ignored by *os.dupterm_notify()*, but is required to allow calling dupterm_notify from an interrupt handler such as *UART.irq()*.

Example:

```
from machine import UART
import os
uart = UART(0)
os.dupterm(uart, 0)
uart.irq(os.dupterm_notify, machine.UART.IRQ_RX)
```

Note: If the `dupterm_notify()` function is not called, input from the custom stream may not be detected or processed until the next REPL poll, potentially delaying KeyboardInterrupts or other control signals. This is especially important for UART, Bluetooth and other non-standard REPL connections, where automatic notification is not guaranteed.

Filesystem mounting

The following functions and classes have been moved to the *vfs* module. They are provided in this module only for backwards compatibility and will be removed in version 2 of MicroPython.

os.**mount**(*fsobj*, *mount_point*, ***, *readonly*)

See *vfs.mount*.

os.**umount**(*mount_point*)

See *vfs.umount*.

class os.**VfsFat**(*block_dev*)

See *vfs.VfsFat*.

class os.**VfsLfs1**(*block_dev*, *readsize=32*, *progsize=32*, *lookahead=32*)

See *vfs.VfsLfs1*.

class os.**VfsLfs2**(*block_dev*, *readsize=32*, *progsize=32*, *lookahead=32*, *mtime=True*)

See *vfs.VfsLfs2*.

class os.**VfsPosix**(*root=None*)

See *vfs.VfsPosix*.

1.1.17 platform – access to underlying platform's identifying data

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: platform.

This module tries to retrieve as much platform-identifying data as possible. It makes this information available via function APIs.

Functions

platform.**platform**()

Returns a string identifying the underlying platform. This string is composed of several substrings in the following order, delimited by dashes (−):

- the name of the platform system (e.g. Unix, Windows or MicroPython)
- the MicroPython version
- the architecture of the platform
- the version of the underlying platform
- the concatenation of the name of the libc that MicroPython is linked to and its corresponding version.

For example, this could be "MicroPython-1.20.0-xtensa-IDFv4.2.4-with-newlib3.0.0".

platform.**python_compiler**()

Returns a string identifying the compiler used for compiling MicroPython.

platform.**libc_ver**()

Returns a tuple of strings *(lib, version)*, where *lib* is the name of the libc that MicroPython is linked to, and *version* the corresponding version of this libc.

platform.**processor**()

Returns a string with a detailed name of the processor, if one is available. If no name for the processor is known, it will return an empty string instead.

This is currently available only on RISC-V targets (both 32 and 64 bits).

1.1.18 random – generate random numbers

This module implements a pseudo-random number generator (PRNG).

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: random .

Note: The following notation is used for intervals:

- () are open interval brackets and do not include their endpoints. For example, (0, 1) means greater than 0 and less than 1. In set notation: (0, 1) = {x | 0 < x < 1}.
- [] are closed interval brackets which include all their limit points. For example, [0, 1] means greater than or equal to 0 and less than or equal to 1. In set notation: [0, 1] = {x | 0 <= x <= 1}.

Note: The *randrange()*, *randint()* and *choice()* functions are only available if the MICROPY_PY_RANDOM_EXTRA_FUNCS configuration option is enabled.

Functions for integers

random.**getrandbits**(*n*)

Return an integer with *n* random bits (0 <= n <= 32).

random.**randint**(*a*, *b*)

Return a random integer in the range [*a*, *b*].

random.**randrange**(*stop*)

random.**randrange**(*start*, *stop*)

random.**randrange**(*start*, *stop*[, *step*])

The first form returns a random integer from the range [0, *stop*). The second form returns a random integer from the range [*start*, *stop*). The third form returns a random integer from the range [*start*, *stop*) in steps of *step*. For instance, calling `randrange(1, 10, 2)` will return odd numbers between 1 and 9 inclusive.

Functions for floats

random.**random**()

Return a random floating point number in the range [0.0, 1.0).

random.**uniform**(*a*, *b*)

Return a random floating point number N such that *a* <= N <= *b* for *a* <= *b*, and *b* <= N <= *a* for *b* < *a*.

Other Functions

`random.`**`seed`**(*n=None*, */*)

Initialise the random number generator module with the seed *n* which should be an integer. When no argument (or `None`) is passed in it will (if supported by the port) initialise the PRNG with a true random number (usually a hardware generated random number).

The `None` case only works if `MICROPY_PY_RANDOM_SEED_INIT_FUNC` is enabled by the port, otherwise it raises `ValueError`.

`random.`**`choice`**(*sequence*)

Chooses and returns one item at random from *sequence* (tuple, list or any object that supports the subscript operation).

1.1.19 `re` – simple regular expressions

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `re`.

This module implements regular expression operations. Regular expression syntax supported is a subset of CPython `re` module (and actually is a subset of POSIX extended regular expressions).

Supported operators and special sequences are:

`.`
Match any character.

`[...]`
Match set of characters. Individual characters and ranges are supported, including negated sets (e.g. `[^a-c]`).

`^`
Match the start of the string.

`$`
Match the end of the string.

`?`
Match zero or one of the previous sub-pattern.

`*`
Match zero or more of the previous sub-pattern.

`+`
Match one or more of the previous sub-pattern.

`??`
Non-greedy version of `?`, match zero or one, with the preference for zero.

`*?`
Non-greedy version of `*`, match zero or more, with the preference for the shortest match.

`+?`
Non-greedy version of `+`, match one or more, with the preference for the shortest match.

`|`
Match either the left-hand side or the right-hand side sub-patterns of this operator.

`(...)`
Grouping. Each group is capturing (a substring it captures can be accessed with *`match.group()`* method).

(?:...)
Non-capturing grouping. Each group is matched using the same rules as regular grouping, but will not be part of the match object.

\d
Matches digit. Equivalent to `[0-9]`.

\D
Matches non-digit. Equivalent to `[^0-9]`.

\s
Matches whitespace. Equivalent to `[ \t-\r]`.

\S
Matches non-whitespace. Equivalent to `[^ \t-\r]`.

\w
Matches "word characters" (ASCII only). Equivalent to `[A-Za-z0-9_]`.

\W
Matches non "word characters" (ASCII only). Equivalent to `[^A-Za-z0-9_]`.

****
Escape character. Any other character following the backslash, except for those listed above, is taken literally. For example, `\*` is equivalent to literal `*` (not treated as the `*` operator). Note that `\r`, `\n`, etc. are not handled specially, and will be equivalent to literal letters `r`, `n`, etc. Due to this, it's not recommended to use raw Python strings (`r""`) for regular expressions. For example, `r"\r\n"` when used as the regular expression is equivalent to `"rn"`. To match CR character followed by LF, use `"\r\n"`.

NOT SUPPORTED:

- counted repetitions (`{m,n}`)
- named groups (`(?P<name>...)`)
- more advanced assertions (`\b`, `\B`)
- special character escapes like `\r`, `\n` - use Python's own escaping instead
- etc.

Example:

```
import re

# As re doesn't support escapes itself, use of r"" strings is not
# recommended.
regex = re.compile("[\r\n]")

regex.split("line1\rline2\nline3\r\n")

# Result:
# ['line1', 'line2', 'line3', '', '']
```

Functions

re.**compile**(*regex_str*[, *flags*])

Compile regular expression, return *regex* object.

re.**match**(*regex_str*, *string*)

Compile *regex_str* and match against *string*. Match always happens from starting position in a string.

re.**search**(*regex_str*, *string*)

Compile *regex_str* and search it in a *string*. Unlike `match`, this will search string for first position which matches regex (which still may be 0 if regex is anchored).

re.**sub**(*regex_str*, *replace*, *string*, *count=0*, *flags=0*, /)

Compile *regex_str* and search for it in *string*, replacing all matches with *replace*, and returning the new string.

replace can be a string or a function. If it is a string then escape sequences of the form `\<number>` and `\g<number>` can be used to expand to the corresponding group (or an empty string for unmatched groups). If *replace* is a function then it must take a single argument (the match) and should return a replacement string.

If *count* is specified and non-zero then substitution will stop after this many substitutions are made. The *flags* argument is ignored.

Note: availability of this function depends on *MicroPython port*.

re.**DEBUG**

Flag value, display debug information about compiled expression. (Availability depends on *MicroPython port*.)

Regex objects

Compiled regular expression. Instances of this class are created using `re.compile()`.

regex.**match**(*string*[, *pos*[, *endpos*]])

regex.**search**(*string*[, *pos*[, *endpos*]])

regex.**sub**(*replace*, *string*, *count=0*, *flags=0*, /)

Similar to the module-level functions `match()`, `search()` and `sub()`. Using methods is (much) more efficient if the same regex is applied to multiple strings.

The optional second parameter *pos* gives an index in the string where the search is to start; it defaults to `0`. This is not completely equivalent to slicing the string; the `'^'` pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter *endpos* limits how far the string will be searched; it will be as if the string is *endpos* characters long, so only the characters from *pos* to `endpos - 1` will be searched for a match.

regex.**split**(*string*, *max_split=-1*, /)

Split a *string* using regex. If *max_split* is given, it specifies maximum number of splits to perform. Returns list of strings (there may be up to *max_split+1* elements if it's specified).

Match objects

Match objects as returned by `match()` and `search()` methods, and passed to the replacement function in `sub()`.

`match.`**`group`**`(`*index*`)`

Return matching (sub)string. *index* is 0 for entire match, 1 and above for each capturing group. Only numeric groups are supported.

`match.`**`groups`**`()`

Return a tuple containing all the substrings of the groups of the match.

Note: availability of this method depends on *MicroPython port*.

`match.`**`start`**`(`[*index*]`)`

`match.`**`end`**`(`[*index*]`)`

Return the index in the original string of the start or end of the substring group that was matched. *index* defaults to the entire group, otherwise it will select a group.

Note: availability of these methods depends on *MicroPython port*.

`match.`**`span`**`(`[*index*]`)`

Returns the 2-tuple `(match.start(index), match.end(index))`.

Note: availability of this method depends on *MicroPython port*.

1.1.20 `select` – wait for events on a set of streams

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `select`.

This module provides functions to efficiently wait for events on multiple `streams` (select streams which are ready for operations).

Functions

`select.`**`poll`**`()`

Create an instance of the Poll class.

`select.`**`select`**`(`*rlist*, *wlist*, *xlist*[, *timeout*]`)`

Wait for activity on a set of objects.

This function is provided by some MicroPython ports for compatibility and is not efficient. Usage of `Poll` is recommended instead.

class `Poll`

Methods

`poll.`**`register`**`(`*obj*[, *eventmask*]`)`

Register `stream` *obj* for polling. *eventmask* is logical OR of:

- `select.POLLIN` - data available for reading
- `select.POLLOUT` - more data can be written

Note that flags like `select.POLLHUP` and `select.POLLERR` are *not* valid as input eventmask (these are unsolicited events which will be returned from *poll()* regardless of whether they are asked for). This semantics is per POSIX.

eventmask defaults to `select.POLLIN | select.POLLOUT`.

It is OK to call this function multiple times for the same *obj*. Successive calls will update *obj*'s eventmask to the value of *eventmask* (i.e. will behave as *modify()*).

`poll.`**`unregister`**(*obj*)

Unregister *obj* from polling.

`poll.`**`modify`**(*obj*, *eventmask*)

Modify the *eventmask* for *obj*. If *obj* is not registered, *OSError* is raised with error of ENOENT.

`poll.`**`poll`**(*timeout=-1*, */*)

Wait for at least one of the registered objects to become ready or have an exceptional condition, with optional timeout in milliseconds (if *timeout* arg is not specified or -1, there is no timeout).

Returns list of (`obj`, `event`, ...) tuples. There may be other elements in tuple, depending on a platform and version, so don't assume that its size is 2. The `event` element specifies which events happened with a stream and is a combination of `select.POLL*` constants described above. Note that flags `select.POLLHUP` and `select.POLLERR` can be returned at any time (even if were not asked for), and must be acted on accordingly (the corresponding stream unregistered from poll and likely closed), because otherwise all further invocations of *poll()* may return immediately with these flags set for this stream again.

In case of timeout, an empty list is returned.

Calling `poll.poll` is guaranteed to call pending callback functions before entering the polling loop.

Difference to CPython

Tuples returned may contain more than 2 elements as described above.

`poll.`**`ipoll`**(*timeout=-1*, *flags=0*, */*)

Like *poll.poll()*, but instead returns an iterator which yields a *callee-owned tuple*. This function provides an efficient, allocation-free way to poll on streams.

If *flags* is 1, one-shot behaviour for events is employed: streams for which events happened will have their event masks automatically reset (equivalent to `poll.modify(obj, 0)`), so new events for such a stream won't be processed until new mask is set with *poll.modify()*. This behaviour is useful for asynchronous I/O schedulers.

Calling `poll.ipoll` is guaranteed to call pending callback functions before entering the polling loop.

Difference to CPython

This function is a MicroPython extension.

1.1.21 `socket` – socket module

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `socket`.

This module provides access to the BSD socket interface.

Difference to CPython

For efficiency and consistency, socket objects in MicroPython implement a *`stream`* (file-like) interface directly. In CPython, you need to convert a socket to a file-like object using *`makefile()`* method. This method is still supported by MicroPython (but is a no-op), so where compatibility with CPython matters, be sure to use it.

Socket address format(s)

The native socket address format of the `socket` module is an opaque data type returned by *`getaddrinfo`* function, which must be used to resolve textual address (including numeric addresses):

```
sockaddr = socket.getaddrinfo('www.micropython.org', 80)[0][-1]
# You must use getaddrinfo() even for numeric addresses
sockaddr = socket.getaddrinfo('127.0.0.1', 80)[0][-1]
# Now you can use that address
sock.connect(sockaddr)
```

Using *`getaddrinfo`* is the most efficient (both in terms of memory and processing power) and portable way to work with addresses.

However, `socket` module (note the difference with native MicroPython `socket` module described here) provides CPython-compatible way to specify addresses using tuples, as described below. Note that depending on a *MicroPython port*, `socket` module can be builtin or need to be installed from *`micropython-lib`* (as in the case of *MicroPython Unix port*), and some ports still accept only numeric addresses in the tuple format, and require to use *`getaddrinfo`* function to resolve domain names.

Summing up:

- Always use *`getaddrinfo`* when writing portable applications.
- Tuple addresses described below can be used as a shortcut for quick hacks and interactive use, if your port supports them.

Tuple address format for `socket` module:

- IPv4: *(ipv4_address, port)*, where *ipv4_address* is a string with dot-notation numeric IPv4 address, e.g. `"8.8.8.8"`, and *port* is and integer port number in the range 1-65535. Note the domain names are not accepted as *ipv4_address*, they should be resolved first using *`socket.getaddrinfo()`*.
- IPv6: *(ipv6_address, port, flowinfo, scopeid)*, where *ipv6_address* is a string with colon-notation numeric IPv6 address, e.g. `"2001:db8::1"`, and *port* is an integer port number in the range 1-65535. *flowinfo* must be 0. *scopeid* is the interface scope identifier for link-local addresses. Note the domain names are not accepted as *ipv6_address*, they should be resolved first using *`socket.getaddrinfo()`*. Availability of IPv6 support depends on a *MicroPython port*.

Functions

socket.**getaddrinfo**(*host*, *port*, *af=0*, *type=0*, *proto=0*, *flags=0*, */*)

Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. Arguments *af*, *type*, and *proto* (which have the same meaning as for the *socket()* function) can be used to filter which kind of addresses are returned. If a parameter is not specified or zero, all combinations of addresses can be returned (requiring filtering on the user side).

The resulting list of 5-tuples has the following structure:

```
(family, type, proto, canonname, sockaddr)
```

The following example shows how to connect to a given url:

```
s = socket.socket()
# This assumes that if "type" is not specified, an address for
# SOCK_STREAM will be returned, which may be not true
s.connect(socket.getaddrinfo('www.micropython.org', 80)[0][-1])
```

Recommended use of filtering params:

```
s = socket.socket()
# Guaranteed to return an address which can be connect'ed to for
# stream operation.
s.connect(socket.getaddrinfo('www.micropython.org', 80, 0, SOCK_STREAM)[0][-1])
```

Difference to CPython

CPython raises a `socket.gaierror` exception (*OSError* subclass) in case of error in this function. MicroPython doesn't have `socket.gaierror` and raises OSError directly. Note that error numbers of *getaddrinfo()* form a separate namespace and may not match error numbers from the *errno* module. To distinguish *getaddrinfo()* errors, they are represented by negative numbers, whereas standard system errors are positive numbers (error numbers are accessible using `e.args[0]` property from an exception object). The use of negative values is a provisional detail which may change in the future.

---

socket.**inet_ntop**(*af*, *bin_addr*)

Convert a binary network address *bin_addr* of the given address family *af* to a textual representation:

```
>>> socket.inet_ntop(socket.AF_INET, b"\x7f\0\0\1")
'127.0.0.1'
```

socket.**inet_pton**(*af*, *txt_addr*)

Convert a textual network address *txt_addr* of the given address family *af* to a binary representation:

```
>>> socket.inet_pton(socket.AF_INET, "1.2.3.4")
b'\x01\x02\x03\x04'
```

Constants

socket.**AF_INET**

socket.**AF_INET6**

Address family types. Availability depends on a particular *MicroPython port*.

socket.**SOCK_STREAM**

socket.**SOCK_DGRAM**

Socket types.

socket.**IPPROTO_UDP**

socket.**IPPROTO_TCP**

IP protocol numbers. Availability depends on a particular *MicroPython port*. Note that you don't need to specify these in a call to *socket.socket()*, because *SOCK_STREAM* socket type automatically selects *IPPROTO_TCP*, and *SOCK_DGRAM* - *IPPROTO_UDP*. Thus, the only real use of these constants is as an argument to *setsockopt()*.

socket.SOL_*

Socket option levels (an argument to *setsockopt()*). The exact inventory depends on a *MicroPython port*.

socket.SO_*

Socket options (an argument to *setsockopt()*). The exact inventory depends on a *MicroPython port*.

Constants specific to WiPy:

socket.**IPPROTO_SEC**

Special protocol value to create SSL-compatible socket.

class socket

class socket.**socket**(*af=AF_INET*, *type=SOCK_STREAM*, *proto=IPPROTO_TCP*, */*)

Create a new socket using the given address family, socket type and protocol number. Note that specifying *proto* in most cases is not required (and not recommended, as some MicroPython ports may omit `IPPROTO_*` constants). Instead, *type* argument will select needed protocol automatically:

```
# Create STREAM TCP socket
socket(AF_INET, SOCK_STREAM)
# Create DGRAM UDP socket
socket(AF_INET, SOCK_DGRAM)
```

Methods

socket.**close**()

Mark the socket closed and release all resources. Once that happens, all future operations on the socket object will fail. The remote end will receive EOF indication if supported by protocol.

Sockets are automatically closed when they are garbage-collected, but it is recommended to *close()* them explicitly as soon you finished working with them.

socket.**bind**(*address*)

Bind the socket to *address*. The socket must not already be bound.

socket.**listen**([*backlog*])

Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it's lower, it will be set to 0); and specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

socket.**accept**()

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

socket.**connect**(*address*)

Connect to a remote socket at *address*.

socket.**send**(*bytes*)

Send data to the socket. The socket must be connected to a remote socket. Returns number of bytes sent, which may be smaller than the length of data ("short write").

socket.**sendall**(*bytes*)

Send all data to the socket. The socket must be connected to a remote socket. Unlike *send()*, this method will try to send all of data, by sending data chunk by chunk consecutively.

The behaviour of this method on non-blocking sockets is undefined. Due to this, on MicroPython, it's recommended to use *write()* method instead, which has the same "no short writes" policy for blocking sockets, and will return number of bytes sent on non-blocking sockets.

socket.**recv**(*bufsize*[, *flags*])

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by bufsize.

Most ports support the optional *flags* argument. Available *flags* are defined as constants in the socket module and have the same meaning as in CPython. MSG_PEEK and MSG_DONTWAIT are supported on all ports which accept the *flags* argument.

socket.**sendto**(*bytes*, *address*)

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*.

socket.**recvfrom**(*bufsize*[, *flags*])

Receive data from the socket. The return value is a pair *(bytes, address)* where *bytes* is a bytes object representing the data received and *address* is the address of the socket sending the data.

See the *recv* function for an explanation of the optional *flags* argument.

socket.**setsockopt**(*level*, *optname*, *value*)

Set the value of the given socket option. The needed symbolic constants are defined in the socket module (SO_* etc.). The *value* can be an integer or a bytes-like object representing a buffer.

socket.**settimeout**(*value*)

Note: Not every port supports this method, see below.

Set a timeout on blocking socket operations. The value argument can be a nonnegative floating point number expressing seconds, or None. If a non-zero value is given, subsequent socket operations will raise an *OSError* exception if the timeout period value has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If None is given, the socket is put in blocking mode.

Not every *MicroPython port* supports this method. A more portable and generic solution is to use *select.poll* object. This allows to wait on multiple objects at the same time (and not just on sockets, but on generic *stream* objects which support polling). Example:

```
# Instead of:
s.settimeout(1.0)  # time in seconds
s.read(10)  # may timeout

# Use:
poller = select.poll()
poller.register(s, select.POLLIN)
res = poller.poll(1000)  # time in milliseconds
if not res:
    # s is still not ready for input, i.e. operation timed out
```

Difference to CPython

CPython raises a `socket.timeout` exception in case of timeout, which is an *`OSError`* subclass. MicroPython raises an OSError directly instead. If you use `except OSError:` to catch the exception, your code will work both in MicroPython and CPython.

`socket.`**`setblocking`**(*flag*)

Set blocking or non-blocking mode of the socket: if flag is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain *`settimeout()`* calls:

- `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
- `sock.setblocking(False)` is equivalent to `sock.settimeout(0)`

`socket.`**`makefile`**(*mode='rb'*, *buffering=0*, */*)

Return a file object associated with the socket. The exact returned type depends on the arguments given to makefile(). The support is limited to binary modes only ('rb', 'wb', and 'rwb'). CPython's arguments: *encoding*, *errors* and *newline* are not supported.

Difference to CPython

As MicroPython doesn't support buffered streams, values of *buffering* parameter is ignored and treated as if it was 0 (unbuffered).

Difference to CPython

Closing the file object returned by makefile() WILL close the original socket as well.

`socket.`**`read`**(*[size]*)

Read up to size bytes from the socket. Return a bytes object. If *size* is not given, it reads all data available from the socket until EOF; as such the method will not return until the socket is closed. This function tries to read as much data as requested (no "short reads"). This may be not possible with non-blocking socket though, and then less data will be returned.

`socket.`**`readinto`**(*buf*[, *nbytes*])

Read bytes into the *buf*. If *nbytes* is specified then read at most that many bytes. Otherwise, read at most *len(buf)* bytes. Just as *`read()`*, this method follows "no short reads" policy.

Return value: number of bytes read and stored into *buf*.

socket.**readline**()

Read a line, ending in a newline character.

Return value: the line read.

socket.**write**(*buf*)

Write the buffer of bytes to the socket. This function will try to write all data to a socket (no "short writes"). This may be not possible with a non-blocking socket though, and returned value will be less than the length of *buf*.

Return value: number of bytes written.

exception socket.**error**

MicroPython does NOT have this exception.

Difference to CPython

CPython used to have a `socket.error` exception which is now deprecated, and is an alias of *OSError*. In MicroPython, use *OSError* directly.

1.1.22 ssl – SSL/TLS module

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `ssl`.

This module provides access to Transport Layer Security (previously and widely known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side.

Functions

ssl.**wrap_socket**(*sock*, *server_side=False*, *key=None*, *cert=None*, *cert_reqs=CERT_NONE*, *cadata=None*, *server_hostname=None*, *do_handshake=True*)

Wrap the given *sock* and return a new wrapped-socket object. The implementation of this function is to first create an *SSLContext* and then call the *SSLContext.wrap_socket* method on that context object. The arguments *sock*, *server_side* and *server_hostname* are passed through unchanged to the method call. The argument *do_handshake* is passed through as *do_handshake_on_connect*. The remaining arguments have the following behaviour:

- *cert_reqs* determines whether the peer (server or client) must present a valid certificate. Note that for mbedtls based ports, `ssl.CERT_NONE` and `ssl.CERT_OPTIONAL` will not validate any certificate, only `ssl.CERT_REQUIRED` will.
- *cadata* is a str or bytes object containing the CA certificate chain that will validate the peer's certificate. It can be one or more certificates in PEM format, or single DER-encoded certificate.

Depending on the underlying module implementation in a particular *MicroPython port*, some or all keyword arguments above may be not supported.

class SSLContext

class ssl.**SSLContext**(*protocol*, /)

Create a new SSLContext instance. The *protocol* argument must be one of the PROTOCOL_* constants.

SSLContext.**load_cert_chain**(*certfile*, *keyfile*)

Load a private key and the corresponding certificate. The *certfile* is a string with the file path of the certificate. The *keyfile* is a string with the file path of the private key.

Difference to CPython

MicroPython extension: *certfile* and *keyfile* can be bytes objects instead of strings, in which case they are interpreted as the actual certificate/key data.

SSLContext.**load_verify_locations**(*cafile=None*, *cadata=None*)

Load the CA certificate chain that will validate the peer's certificate. *cafile* is the file path of the CA certificates. *cadata* is a bytes object containing the CA certificates. Only one of these arguments should be provided.

SSLContext.**get_ciphers**()

Get a list of enabled ciphers, returned as a list of strings.

SSLContext.**set_ciphers**(*ciphers*)

Set the available ciphers for sockets created with this context. *ciphers* should be a list of strings in the IANA cipher suite format .

SSLContext.**psk_identity**

SSLContext.**psk_key**

The pre-shared key (PSK) identity and key to authenticate with as a client. Set both to use PSK: *psk_identity* is the identity sent to the server and *psk_key* is the shared key, both as *bytes* objects.

While PSK is configured the context offers only PSK cipher suites, so the connection cannot fall back to a non-PSK (e.g. certificate-based) suite.

Availability depends on the port's mbedTLS being built with PSK support.

SSLContext.**server_psk_keys**

A mapping used by a server to look up the key for the identity presented by a connecting client. Set it to accept PSK clients:

```
ctx.server_psk_keys = {b"my-identity": b"my-key"}
```

Its get() method is called with the client's identity (a *bytes* object) and should return the corresponding key as a *bytes* object, or None to reject an unknown identity. Any object providing such a get() method may be used, so keys can be computed or fetched on demand. As with the client, the context is restricted to PSK cipher suites while this is set.

SSLContext.**wrap_socket**(*sock*, *, *server_side=False*, *do_handshake_on_connect=True*, *server_hostname=None*, *client_id=None*)

Takes a *stream* *sock* (usually socket.socket instance of SOCK_STREAM type), and returns an instance of ssl.SSLSocket, wrapping the underlying stream. The returned object has the usual *stream* interface methods like read(), write(), etc.

- *server_side* selects whether the wrapped socket is on the server or client side. A server-side SSL socket should be created from a normal socket returned from *accept()* on a non-SSL listening server socket.

- *do_handshake_on_connect* determines whether the handshake is done as part of the `wrap_socket` or whether it is deferred to be done as part of the initial reads or writes For blocking sockets doing the handshake immediately is standard. For non-blocking sockets (i.e. when the *sock* passed into `wrap_socket` is in non-blocking mode) the handshake should generally be deferred because otherwise `wrap_socket` blocks until it completes. Note that in AXTLS the handshake can be deferred until the first read or write but it then blocks until completion.
- *server_hostname* is for use as a client, and sets the hostname to check against the received server certificate. It also sets the name for Server Name Indication (SNI), allowing the server to present the proper certificate.
- *client_id* is a MicroPython-specific extension argument used only when implementing a DTLS Server. See *DTLS support* for details.

Warning: Some implementations of `ssl` module do NOT validate server certificates, which makes an SSL connection established prone to man-in-the-middle attacks.

CPython's `wrap_socket` returns an `SSLSocket` object which has methods typical for sockets, such as `send`, `recv`, etc. MicroPython's `wrap_socket` returns an object more similar to CPython's `SSLObject` which does not have these socket methods.

`SSLContext.`**`verify_mode`**

Set or get the behaviour for verification of peer certificates. Must be one of the `CERT_*` constants.

Note: `ssl.CERT_REQUIRED` requires the device's date/time to be properly set, e.g. using *mpremote rtc --set* or `ntptime`, and `server_hostname` must be specified when on the client side.

Exceptions

`ssl.`**`SSLError`**

This exception does NOT exist. Instead its base class, OSError, is used.

DTLS support

Difference to CPython

This is a MicroPython extension.

On most ports, this module supports DTLS in client and server mode via the *PROTOCOL_DTLS_CLIENT* and *PROTOCOL_DTLS_SERVER* constants that can be used as the `protocol` argument of *SSLContext*.

In this case the underlying socket is expected to behave as a datagram socket (i.e. like the socket opened with `socket.socket` with `socket.AF_INET` as `af` and `socket.SOCK_DGRAM` as `type`).

DTLS is only supported on ports that use mbedTLS, and it is enabled by default in most configurations but can be manually disabled by defining `MICROPY_PY_SSL_DTLS` to 0.

DTLS server support

MicroPython's DTLS server support is configured with "Hello Verify" as required for DTLS 1.2. This is transparent for DTLS clients, but there are relevant considerations when implementing a DTLS server in MicroPython:

- The server should pass an additional argument *client_id* when calling *SSLContext.wrap_socket()*. This ID must be a *bytes* object (or similar) with a transport-specific identifier representing the client.

  The simplest approach is to convert the tuple of `(client_ip, client_port)` returned from `socket.recv_from()` into a byte string, i.e.:

```
_, client_addr = sock.recvfrom(1, socket.MSG_PEEK)
sock.connect(client_addr)  # Connect back to the client
sock = ssl_ctx.wrap_socket(sock, server_side=True,
                           client_id=repr(client_addr).encode())
```

- The first time a client connects, the server call to `wrap_socket` will fail with a *OSError* error "Hello Verify Required". This is because the DTLS "Hello Verify" cookie is not yet known by the client. If the same client connects a second time then `wrap_socket` will succeed.
- DTLS cookies for "Hello Verify" are associated with the *SSLContext* object, so the same *SSLContext* object should be used to wrap a subsequent connection from the same client. The cookie implementation includes a timeout and has constant memory use regardless of how many clients connect, so it's OK to reuse the same *SSLContext* object for the lifetime of the server.

Constants

`ssl.`**PROTOCOL_TLS_CLIENT**

`ssl.`**PROTOCOL_TLS_SERVER**

`ssl.`**PROTOCOL_DTLS_CLIENT** (*when DTLS support is enabled*)

`ssl.`**PROTOCOL_DTLS_SERVER** (*when DTLS support is enabled*)

Supported values for the *protocol* parameter.

`ssl.`**CERT_NONE**

`ssl.`**CERT_OPTIONAL**

`ssl.`**CERT_REQUIRED**

Supported values for *cert_reqs* parameter, and the *SSLContext.verify_mode* attribute.

1.1.23 `string.templatelib` – Template String Support

This module provides support for template strings (t-strings) as defined in PEP 750. Template strings are created using the `t` prefix and provide access to both the literal string parts and interpolated values before they are combined.

Availability: template strings require `MICROPY_PY_TSTRINGS` to be enabled at compile time. They are enabled by default at the full feature level, which includes the alif, mimxrt and samd (SAMD51 only) ports, the unix coverage variant and the webassembly pyscript variant.

Classes

class string.templatelib.**Template**(**args*)

Represents a template string. Template objects are typically created by t-string syntax (`t"..."`) but can also be constructed directly using the constructor.

strings

A tuple of string literals that appear between interpolations.

interpolations

A tuple of *Interpolation* objects representing the interpolated expressions.

values

A read-only property that returns a tuple containing the `value` attribute from each interpolation in the template.

__iter__()

Iterate over the template contents, yielding string parts and *Interpolation* objects in the order they appear. Empty strings are omitted.

__add__(*other*)

Concatenate two templates. Returns a new *Template* combining the strings and interpolations from both templates.

Raises

TypeError – if *other* is not a *Template*

Template concatenation with `str` is prohibited to avoid ambiguity about whether the string should be treated as a literal or interpolation:

```
t1 = t"Hello "
t2 = t"World"
result = t1 + t2  # Valid

# TypeError: cannot concatenate str to Template
result = t1 + "World"
```

class string.templatelib.**Interpolation**(*value*, *expression=''*, *conversion=None*, *format_spec=''*)

Represents an interpolated expression within a template string. All arguments can be passed as keyword arguments.

value

The evaluated value of the interpolated expression.

expression

The string representation of the expression as it appeared in the template string.

conversion

The conversion specifier (`'s'` or `'r'`) if present, otherwise `None`. Note that MicroPython does not support the `'a'` conversion.

format_spec

The format specification string if present, otherwise an empty string.

Template String Syntax

Template strings use the same syntax as f-strings but with a `t` prefix:

```
name = "World"
template = t"Hello {name}!"

# Access template components
print(template.strings)         # ('Hello ', '!')
print(template.values)          # ('World',)
print(template.interpolations[0].expression)  # 'name'
```

Conversion Specifiers

Template strings store conversion specifiers as metadata. Unlike f-strings, the conversion is not applied automatically:

```
value = "test"
t = t"{value!r}"
# t.interpolations[0].value == "test" (not repr(value))
# t.interpolations[0].conversion == "r"
```

Processing code must explicitly apply conversions when needed.

Format Specifications

Format specifications are stored as metadata in the `Interpolation` object. Unlike f-strings, formatting is not applied automatically:

```
pi = 3.14159
t = t"{pi:.2f}"
# t.interpolations[0].value == 3.14159 (not formatted)
# t.interpolations[0].format_spec == ".2f"
```

Per PEP 750, processing code is not required to use format specifications, but when present they should be respected and match f-string behavior where possible.

Debug Format

The debug format `{expr=}` is supported:

```
x = 42
t = t"{x=}"
# t.strings == ("x=", "")
# t.interpolations[0].expression == "x"
# t.interpolations[0].conversion == "r"
```

Important

As per PEP 750, unlike f-strings, template strings do not automatically apply conversions or format specifications. This is by design to allow processing code to control how these are handled. Processing code must explicitly handle these attributes.

MicroPython does not provide the `format()` built-in function. Use string formatting methods like `str.format()` instead.

Example Usage

Basic processing without format support:

```
def simple_process(template):
    """Simple template processing"""
    parts = []
    for item in template:
        if isinstance(item, str):
            parts.append(item)
        else:
            parts.append(str(item.value))
    return "".join(parts)
```

Processing template with format support:

```
from string.templatelib import Template, Interpolation

def convert(value, conversion):
    """Apply conversion specifier to value"""
    if conversion == "r":
        return repr(value)
    elif conversion == "s":
        return str(value)
    return value

def process_template(template):
    """Process template with conversion and format support"""
    result = []
    for part in template:
        if isinstance(part, str):
            result.append(part)
        else:  # Interpolation
            value = convert(part.value, part.conversion)
            if part.format_spec:
                # Apply format specification using str.format
                value = ("{:" + part.format_spec + "}").format(value)
            else:
                value = str(value)
            result.append(value)
    return "".join(result)

pi = 3.14159
name = "Alice"
t = t"{name!r}: {pi:.2f}"
print(process_template(t))
# Output: "'Alice': 3.14"

# Other format specifications work too
value = 42
print(process_template(t"{value:>10}"))  # "        42"
print(process_template(t"{value:04d}"))  # "0042"
```

HTML escaping example:

```
def html_escape(value):
    """Escape HTML special characters"""
    if not isinstance(value, str):
        value = str(value)
    return value.replace("&", "&amp;").replace("<", "&lt;").replace(">", "&gt;")

def safe_html(template):
    """Convert template to HTML-safe string"""
    result = []
    for part in template:
        if isinstance(part, str):
            result.append(part)
        else:
            result.append(html_escape(part.value))
    return "".join(result)

user_input = "<script>alert('xss')</script>"
t = t"User said: {user_input}"
print(safe_html(t))
# Output: "User said: &lt;script&gt;alert('xss')&lt;/script&gt;"
```

See Also

- PEP 750 - Template Strings specification
- Format String Syntax - Format string syntax
- Formatted string literals - f-strings in Python

1.1.24 `struct` – pack and unpack primitive data types

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `struct`.

The following byte orders are supported:

Character	Byte order	Size	Alignment
@	native	native	native
<	little-endian	standard	none
>	big-endian	standard	none
!	network (= big-endian)	standard	none

The following data types are supported:

Format	C Type	Python type	Standard size
b	signed char	integer	1
B	unsigned char	integer	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer (*1*)	4
I	unsigned int	integer (*1*)	4
l	long	integer (*1*)	4
L	unsigned long	integer (*1*)	4
q	long long	integer (*1*)	8
Q	unsigned long long	integer (*1*)	8
e	n/a (half-float)	float (*2*)	2
f	float	float (*2*)	4
d	double	float (*2*)	8
s	char[]	bytes	
P	void *	integer	

(1) Requires long support when used with values larger than 30 bits.

(2) Requires floating point support.

Difference to CPython

Whitespace is not supported in format strings.

Functions

`struct.`**`calcsize`**(*fmt*)

Return the number of bytes needed to store the given *fmt*.

`struct.`**`pack`**(*fmt*, *v1*, *v2*, ...)

Pack the values *v1*, *v2*, … according to the format string *fmt*. The return value is a bytes object encoding the values.

`struct.`**`pack_into`**(*fmt*, *buffer*, *offset*, *v1*, *v2*, ...)

Pack the values *v1*, *v2*, … according to the format string *fmt* into a *buffer* starting at *offset*. *offset* may be negative to count from the end of *buffer*.

`struct.`**`unpack`**(*fmt*, *data*)

Unpack from the *data* according to the format string *fmt*. The return value is a tuple of the unpacked values.

`struct.`**`unpack_from`**(*fmt*, *data*, *offset=0*, */*)

Unpack from the *data* starting at *offset* according to the format string *fmt*. *offset* may be negative to count from the end of *data*. The return value is a tuple of the unpacked values.

1.1.25 `sys` – system specific functions

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `sys`.

Functions

`sys.`**`exit`**(*retval=0*, */*)

Terminate current program with a given exit code. Underlyingly, this function raises a *`SystemExit`* exception. If an argument is given, its value given as an argument to *`SystemExit`*.

On embedded ports (i.e. all ports but Windows and Unix), an unhandled *`SystemExit`* currently causes a *Soft Reset* of MicroPython.

`sys.`**`atexit`**(*func*)

Register *func* to be called upon termination. *func* must be a callable that takes no arguments, or `None` to disable the call. The `atexit` function will return the previous value set by this function, which is initially `None`.

Difference to CPython

This function is a MicroPython extension intended to provide similar functionality to the `atexit` module in CPython.

`sys.`**`print_exception`**(*exc*, *file=sys.stdout*, */*)

Print exception with a traceback to a file-like object *file* (or *`sys.stdout`* by default).

Difference to CPython

This is simplified version of a function which appears in the `traceback` module in CPython. Unlike `traceback.print_exception()`, this function takes just exception value instead of exception type, exception value, and traceback object; *file* argument should be positional; further arguments are not supported. CPython-compatible `traceback` module can be found in *`micropython-lib`*.

`sys.`**`settrace`**(*tracefunc*)

Enable tracing of bytecode execution. For details see the CPython documentation.

This function requires a custom MicroPython build as it is typically not present in pre-built firmware (due to it affecting performance). The relevant configuration option is *MICROPY_PY_SYS_SETTRACE*.

Constants

`sys.`**`argv`**

A mutable list of arguments the current program was started with.

`sys.`**`byteorder`**

The byte order of the system (`"little"` or `"big"`).

`sys.`**`implementation`**

Object with information about the current Python implementation. For MicroPython, it has following attributes:

- *name* - string "micropython"
- *version* - tuple (major, minor, micro, releaselevel), e.g. (1, 22, 0, '')

- *_machine* - string describing the underlying machine
- *_mpy* - supported mpy file-format version (optional attribute)
- *_build* - string that can help identify the configuration that MicroPython was built with
- *_thread* - optional string attribute, exists if the target has threading and is either "GIL" or "unsafe"

This object is the recommended way to distinguish MicroPython from other Python implementations (note that it still may not exist in the very minimal ports).

Starting with version 1.22.0-preview, the fourth node *releaselevel* in *implementation.version* is either an empty string or `"preview"`.

The *_build* entry was added in version 1.25.0 and is a hyphen-separated set of elements. New elements may be appended in the future so it's best to access this field using `sys.implementation._build.split("-")`. The elements that are currently used are:

- On the unix, webassembly and windows ports the first element is the variant name, for example `'standard'`.
- On microcontroller targets, the first element is the board name and the second element (if present) is the board variant, for example `'RPI_PICO2-RISCV'`

The *_thread* entry was added in version 1.26.0 and if it exists then the target has the `_thread` module. If the target enables the GIL (global interpreter lock) then this attribute is `"GIL"`. Otherwise the attribute is `"unsafe"` and the target has threading but does not enable the GIL, and mutable Python objects (such as *bytearray*, *list* and *dict*) that are shared amongst threads must be protected explicitly by locks such as `_thread.allocate_lock`.

Difference to CPython

CPython mandates more attributes for this object, but the actual useful bare minimum is implemented in MicroPython.

`sys.`**`maxsize`**

Maximum value which a native integer type can hold on the current platform, or maximum value representable by MicroPython integer type, if it's smaller than platform max value (that is the case for MicroPython ports without long int support).

This attribute is useful for detecting "bitness" of a platform (32-bit vs 64-bit, etc.). It's recommended to not compare this attribute to some value directly, but instead count number of bits in it:

```
bits = 0
v = sys.maxsize
while v:
    bits += 1
    v >>= 1
if bits > 32:
    # 64-bit (or more) platform
    ...
else:
    # 32-bit (or less) platform
    # Note that on 32-bit platform, value of bits may be less than 32
    # (e.g. 31) due to peculiarities described above, so use "> 16",
    # "> 32", "> 64" style of comparisons.
```

`sys.`**`modules`**

Dictionary of loaded modules. On some ports, it may not include builtin modules.

sys.**path**

A mutable list of directories to search for imported modules.

Difference to CPython

On MicroPython, an entry with the value `".frozen"` will indicate that import should search *frozen modules* at that point in the search. If no frozen module is found then search will *not* look for a directory called `.frozen`, instead it will continue with the next entry in `sys.path`.

---

sys.**platform**

The platform that MicroPython is running on. For OS/RTOS ports, this is usually an identifier of the OS, e.g. `"linux"`. For baremetal ports it is an identifier of a board, e.g. `"pyboard"` for the original MicroPython reference board. It thus can be used to distinguish one board from another. If you need to check whether your program runs on MicroPython (vs other Python implementation), use *sys.implementation* instead.

sys.**ps1**

sys.**ps2**

Mutable attributes holding strings, which are used for the REPL prompt. The defaults give the standard Python prompt of >>> and `...`.

sys.**stderr**

Standard error *stream*.

sys.**stdin**

Standard input *stream*.

sys.**stdout**

Standard output *stream*.

sys.**tracebacklimit**

A mutable attribute holding an integer value which is the maximum number of traceback entries to store in an exception. Set to 0 to disable adding tracebacks. Defaults to 1000.

Note: this is not available on all ports.

sys.**version**

Python language version that this implementation conforms to, as a string.

sys.**version_info**

Python language version that this implementation conforms to, as a tuple of ints.

Difference to CPython

Only the first three version numbers (major, minor, micro) are supported and they can be referenced only by index, not by name.

1.1.26 `time` – time related functions

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `time`.

The `time` module provides functions for getting the current time and date, measuring time intervals, and for delays.

Time Epoch: The unix, windows, webassembly, alif, mimxrt and rp2 ports use the standard for POSIX systems epoch of 1970-01-01 00:00:00 UTC. The other embedded ports use an epoch of 2000-01-01 00:00:00 UTC. Epoch year may be determined with `gmtime(0)[0]`.

Maintaining actual calendar date/time: This requires a Real Time Clock (RTC). On systems with underlying OS (including some RTOS), an RTC may be implicit. Setting and maintaining actual calendar time is responsibility of OS/RTOS and is done outside of MicroPython, it just uses OS API to query date/time. On baremetal ports however system time depends on `machine.RTC()` object. The current calendar time may be set using `machine.RTC().datetime(tuple)` function, and maintained by following means:

- By a backup battery (which may be an additional, optional component for a particular board).
- Using networked time protocol (requires setup by a port/user).
- Set manually by a user on each power-up (many boards then maintain RTC time across hard resets, though some may require setting it again in such case).

If actual calendar time is not maintained with a system/MicroPython RTC, functions below which require reference to current absolute time may behave not as expected.

Functions

`time.`**`gmtime`**`(`[*secs*]`)`

`time.`**`localtime`**`(`[*secs*]`)`

Convert the time *secs* expressed in seconds since the Epoch (see above) into an 8-tuple which contains: `(year, month, mday, hour, minute, second, weekday, yearday)` If *secs* is not provided or None, then the current time from the RTC is used.

The *`gmtime()`* function returns a date-time tuple in UTC, and *`localtime()`* returns a date-time tuple in local time.

The format of the entries in the 8-tuple are:

- year includes the century (for example 2014).
- month is 1-12
- mday is 1-31
- hour is 0-23
- minute is 0-59
- second is 0-59
- weekday is 0-6 for Mon-Sun
- yearday is 1-366

`time.`**`mktime`**`(`*date_time_tuple*`)`

This is inverse function of localtime. It's argument is a full 8-tuple which expresses a time as per localtime. It returns an integer which is the number of seconds since the time epoch.

time.**sleep**(*seconds*)

Sleep for the given number of seconds. Some boards may accept *seconds* as a floating-point number to sleep for a fractional number of seconds. Note that other boards may not accept a floating-point argument, for compatibility with them use *sleep_ms()* and *sleep_us()* functions.

Calling sleep, including sleep(0) is guaranteed to call pending callback functions.

time.**sleep_ms**(*ms*)

Delay for given number of milliseconds, should be positive or 0.

This function will delay for at least the given number of milliseconds, but may take longer than that if other processing must take place, for example interrupt handlers or other threads. Passing in 0 for *ms* will still allow this other processing to occur. Use *sleep_us()* for more precise delays.

Calling sleep_ms, including sleep_ms(0) is guaranteed to call pending callback functions.

time.**sleep_us**(*us*)

Delay for given number of microseconds, should be positive or 0.

This function attempts to provide an accurate delay of at least *us* microseconds, but it may take longer if the system has other higher priority processing to perform.

time.**ticks_ms**()

Returns an increasing millisecond counter with an arbitrary reference point, that wraps around after some value.

The wrap-around value is not explicitly exposed, but we will refer to it as *TICKS_MAX* to simplify discussion. Period of the values is *TICKS_PERIOD = TICKS_MAX + 1*. *TICKS_PERIOD* is guaranteed to be a power of two, but otherwise may differ from port to port. The same period value is used for all of *ticks_ms()*, *ticks_us()*, *ticks_cpu()* functions (for simplicity). Thus, these functions will return a value in range [*0* .. *TICKS_MAX*], inclusive, total *TICKS_PERIOD* values. Note that only non-negative values are used. For the most part, you should treat values returned by these functions as opaque. The only operations available for them are *ticks_diff()* and *ticks_add()* functions described below.

Note: Performing standard mathematical operations (+, -) or relational operators (<, <=, >, >=) directly on these value will lead to invalid result. Performing mathematical operations and then passing their results as arguments to *ticks_diff()* or *ticks_add()* will also lead to invalid results from the latter functions.

time.**ticks_us**()

Just like *ticks_ms()* above, but in microseconds.

time.**ticks_cpu**()

Similar to *ticks_ms()* and *ticks_us()*, but with the highest possible resolution in the system. This is usually CPU clocks, and that's why the function is named that way. But it doesn't have to be a CPU clock, some other timing source available in a system (e.g. high-resolution timer) can be used instead. The exact timing unit (resolution) of this function is not specified on time module level, but documentation for a specific port may provide more specific information. This function is intended for very fine benchmarking or very tight real-time loops. Avoid using it in portable code.

Availability: Not every port implements this function.

time.**ticks_add**(*ticks*, *delta*)

Offset ticks value by a given number, which can be either positive or negative. Given a *ticks* value, this function allows to calculate ticks value *delta* ticks before or after it, following modular-arithmetic definition of tick values (see *ticks_ms()* above). *ticks* parameter must be a direct result of call to *ticks_ms()*, *ticks_us()*, or *ticks_cpu()* functions (or from previous call to *ticks_add()*). However, *delta* can be an arbitrary integer number or numeric expression. *ticks_add()* is useful for calculating deadlines for events/tasks. (Note: you must use *ticks_diff()* function to work with deadlines.)

Examples:

```
# Find out what ticks value there was 100ms ago
print(ticks_add(time.ticks_ms(), -100))

# Calculate deadline for operation and test for it
deadline = ticks_add(time.ticks_ms(), 200)
while ticks_diff(deadline, time.ticks_ms()) > 0:
    do_a_little_of_something()

# Find out TICKS_MAX used by this port
print(ticks_add(0, -1))
```

time.**ticks_diff**(*ticks1*, *ticks2*)

Measure ticks difference between values returned from *ticks_ms()*, *ticks_us()*, or *ticks_cpu()* functions, as a signed value which may wrap around.

The argument order is the same as for subtraction operator, `ticks_diff(ticks1, ticks2)` has the same meaning as `ticks1 - ticks2`. However, values returned by *ticks_ms()*, etc. functions may wrap around, so directly using subtraction on them will produce incorrect result. That is why *ticks_diff()* is needed, it implements modular (or more specifically, ring) arithmetic to produce correct result even for wrap-around values (as long as they not too distant in between, see below). The function returns **signed** value in the range [-*TICKS_PERIOD/2* .. *TICKS_PERIOD/2-1*] (that's a typical range definition for two's-complement signed binary integers). If the result is negative, it means that *ticks1* occurred earlier in time than *ticks2*. Otherwise, it means that *ticks1* occurred after *ticks2*. This holds **only** if *ticks1* and *ticks2* are apart from each other for no more than *TICKS_PERIOD/2-1* ticks. If that does not hold, incorrect result will be returned. Specifically, if two tick values are apart for *TICKS_PERIOD/2-1* ticks, that value will be returned by the function. However, if *TICKS_PERIOD/2* of real-time ticks has passed between them, the function will return *-TICKS_PERIOD/2* instead, i.e. result value will wrap around to the negative range of possible values.

Informal rationale of the constraints above: Suppose you are locked in a room with no means to monitor passing of time except a standard 12-notch clock. Then if you look at dial-plate now, and don't look again for another 13 hours (e.g., if you fall for a long sleep), then once you finally look again, it may seem to you that only 1 hour has passed. To avoid this mistake, just look at the clock regularly. Your application should do the same. "Too long sleep" metaphor also maps directly to application behaviour: don't let your application run any single task for too long. Run tasks in steps, and do time-keeping in between.

ticks_diff() is designed to accommodate various usage patterns, among them:

- Polling with timeout. In this case, the order of events is known, and you will deal only with positive results of *ticks_diff()*:

```
# Wait for GPIO pin to be asserted, but at most 500us
start = time.ticks_us()
while pin.value() == 0:
    if time.ticks_diff(time.ticks_us(), start) > 500:
        raise TimeoutError
```

- Scheduling events. In this case, *ticks_diff()* result may be negative if an event is overdue:

```
# This code snippet is not optimized
now = time.ticks_ms()
scheduled_time = task.scheduled_time()
if ticks_diff(scheduled_time, now) > 0:
    print("Too early, let's nap")
    sleep_ms(ticks_diff(scheduled_time, now))
    task.run()
elif ticks_diff(scheduled_time, now) == 0:
    print("Right at time!")
```

(continues on next page)

(continued from previous page)

```
    task.run()
elif ticks_diff(scheduled_time, now) < 0:
    print("Oops, running late, tell task to run faster!")
    task.run(run_faster=true)
```

Note: Do not pass *time()* values to *ticks_diff()*, you should use normal mathematical operations on them. But note that *time()* may (and will) also overflow. This is known as https://en.wikipedia.org/wiki/Year_2038_problem .

time.**time**()

Returns the number of seconds, as an integer, since the Epoch, assuming that underlying RTC is set and maintained as described above. If an RTC is not set, this function returns number of seconds since a port-specific reference point in time (for embedded boards without a battery-backed RTC, usually since power up or reset). If you want to develop portable MicroPython application, you should not rely on this function to provide higher than second precision. If you need higher precision, absolute timestamps, use *time_ns()*. If relative times are acceptable then use the *ticks_ms()* and *ticks_us()* functions. If you need calendar time, *gmtime()* or *localtime()* without an argument is a better choice.

Difference to CPython

In CPython, this function returns number of seconds since Unix epoch, 1970-01-01 00:00 UTC, as a floating-point, usually having microsecond precision. With MicroPython, only Unix port uses the same Epoch, and if floating-point precision allows, returns sub-second precision. Embedded hardware usually doesn't have floating-point precision to represent both long time ranges and subsecond precision, so they use integer value with second precision. Some embedded hardware also lacks battery-powered RTC, so returns number of seconds since last power-up or from other relative, hardware-specific point (e.g. reset).

---

time.**time_ns**()

Similar to *time()* but returns nanoseconds since the Epoch, as an integer (usually a big integer, so will allocate on the heap).

1.1.27 weakref – Python object lifetime management

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: weakref.

This module allows creation of weak references to Python objects. A weak reference is a non-traceable reference to a heap-allocated Python object, so the garbage collector can still reclaim the object even though the weak reference refers to it.

Python callbacks can be registered to be called when an object is reclaimed by the garbage collector. This provides a safe way to clean up when objects are no longer needed.

Availability: the weakref module requires MICROPY_PY_WEAKREF to be enabled at compile time. It is enabled on the unix coverage variant and the webassembly pyscript variant.

ref objects

A ref object is the simplest way to make a weak reference.

class weakref.**ref**(*object*, [*callback*,]/)

Return a weak reference to the given *object*.

If *callback* is given and is not `None` then, when *object* is reclaimed by the garbage collector and if the weak reference object is still alive, the *callback* will be called. The *callback* will be passed the weak reference object as its single argument.

ref.**__call__**()

Calling the weak reference object will return its referenced object if that object is still alive. Otherwise `None` will be returned.

finalize objects

A finalize object is an extended version of a ref object that is more convenient to use, and allows more control over the callback.

class weakref.**finalize**(*object*, *callback*, /, **args*, ***kwargs*)

Return a weak reference to the given *object*. In contrast to *weakref.ref* objects, finalize objects are held onto internally and will not be collected until *object* is collected.

A finalize object starts off alive. It transitions to the dead state when the finalize object is called, either explicitly or when *object* is collected. It also transitions to dead if the *`finalize.detach()`* method is called.

When *object* is reclaimed by the garbage collector (or the finalize object is explicitly called by user code) and the finalize object is still in the alive state, the *callback* will be called. The *callback* will be passed arguments as: `callback(*args, **kwargs)`.

finalize.**__call__**()

If the finalize object is alive then it transitions to the dead state and returns the value of `callback(*args, **kwargs)`. Otherwise `None` will be returned.

finalize.**alive**()

Read-only boolean attribute that indicates if the finalizer is in the alive state.

finalize.**peek**()

If the finalize object is alive then return `(object, callback, args, kwargs)`. Otherwise return `None`.

finalize.**detach**()

If the finalize object is alive then it transitions to the dead state and returns `(object, callback, args, kwargs)`. Otherwise `None` will be returned.

1.1.28 `zlib` – zlib compression & decompression

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `zlib`.

This module allows compression and decompression of binary data with the DEFLATE algorithm (commonly used in the zlib library and gzip archiver).

Note: Prefer to use *deflate.DeflateIO* instead of the functions in this module as it provides a streaming interface to compression and decompression which is convenient and more memory efficient when working with reading or writing compressed data to a file, socket, or stream.

Availability:

- From MicroPython v1.21 onwards, this module may not be present by default on all MicroPython firmware as it duplicates functionality available in the *deflate* module.
- A copy of this module can be installed (or frozen) from *micropython-lib* (source). See *Package management* for more information. This documentation describes that module.
- Requires the built-in *deflate* module (available since MicroPython v1.21)
- Compression support will only be available if compression support is enabled in the built-in *deflate* module.

Functions

zlib.**decompress**(*data*, *wbits=15*, /)

Decompresses *data* into a bytes object.

The *wbits* parameter works the same way as for *zlib.compress()* with the following additional valid values:

- 0: Automatically determine the window size from the zlib header (*data* must be in zlib format).
- 35 to 47: Auto-detect either the zlib or gzip format.

As for *zlib.compress()*, see the CPython documentation for zlib for more information about the *wbits* parameter. As for *zlib.compress()*, MicroPython also supports smaller window sizes than CPython. See more *MicroPython-specific details* in the *deflate* module documentation.

If the data to be decompressed requires a larger window size, it will fail during decompression.

zlib.**compress**(*data*, *wbits=15*, /)

Compresses *data* into a bytes object.

wbits allows you to configure the DEFLATE dictionary window size and the output format. The window size allows you to trade-off memory usage for compression level. A larger window size will allow the compressor to reference fragments further back in the input. The output formats are "raw" DEFLATE (no header/footer), zlib, and gzip, where the latter two include a header and checksum.

The low four bits of the absolute value of *wbits* set the base-2 logarithm of the DEFLATE dictionary window size. So for example, wbits=10, wbits=-10, and wbits=26 all set the window size to 1024 bytes. Valid window sizes are 5 to 15 inclusive (corresponding to 32 to 32k bytes).

Negative values of *wbits* between -5 and -15 correspond to "raw" output mode, positive values between 5 and 15 correspond to zlib output mode, and positive values between 21 and 31 correspond to gzip output mode.

See the CPython documentation for zlib for more information about the *wbits* parameter. Note that MicroPython allows for smaller window sizes, which is useful when memory is constrained while still achieving a reasonable level of compression. It also speeds up the compressor. See more *MicroPython-specific details* in the *deflate* module documentation.

1.1.29 `_thread` – multithreading support

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `_thread`.

This module implements multithreading support.

This module is highly experimental and its API is not yet fully settled and not yet described in this documentation.

1.2 MicroPython-specific libraries

Functionality specific to the MicroPython implementation is available in the following libraries.

1.2.1 `bluetooth` — low-level Bluetooth

This module provides an interface to a Bluetooth controller on a board. Currently this supports Bluetooth Low Energy (BLE) in Central, Peripheral, Broadcaster, and Observer roles, as well as GATT Server and Client and L2CAP connection-oriented-channels. A device may operate in multiple roles concurrently. Pairing (and bonding) is supported on some ports.

This API is intended to match the low-level Bluetooth protocol and provide building-blocks for higher-level abstractions such as specific device types.

Note: For most applications, we recommend using the higher-level aioble library.

Note: This module is still under development and its classes, functions, methods and constants are subject to change.

class BLE

Constructor

class bluetooth.**BLE**

Returns the singleton BLE object.

Configuration

BLE.**active**([*active*,]/)

Optionally changes the active state of the BLE radio, and returns the current state.

The radio must be made active before using any other methods on this class.

BLE.**config**(*'param'*, /)

BLE.**config**(*, *param=value*, ...)

Get or set configuration values of the BLE interface. To get a value the parameter name should be quoted as a string, and just one parameter is queried at a time. To set values use the keyword syntax, and one or more parameter can be set at a time.

Currently supported values are:

- `'mac'`: The current address in use, depending on the current address mode. This returns a tuple of `(addr_type, addr)`.

  See *gatts_write* for details about address type.

  This may only be queried while the interface is currently active.

- `'addr_mode'`: Sets the address mode. Values can be:
  - 0x00 - PUBLIC - Use the controller's public address.
  - 0x01 - RANDOM - Use a generated static address.
  - 0x02 - RPA - Use resolvable private addresses.
  - 0x03 - NRPA - Use non-resolvable private addresses.

  By default the interface mode will use a PUBLIC address if available, otherwise it will use a RANDOM address.

- `'gap_name'`: Get/set the GAP device name used by service 0x1800, characteristic 0x2a00. This can be set at any time and changed multiple times.
- `'rxbuf'`: Get/set the size in bytes of the internal buffer used to store incoming events. This buffer is global to the entire BLE driver and so handles incoming data for all events, including all characteristics. Increasing this allows better handling of bursty incoming data (for example scan results) and the ability to receive larger characteristic values.
- `'mtu'`: Get/set the MTU that will be used during a ATT MTU exchange. The resulting MTU will be the minimum of this and the remote device's MTU. ATT MTU exchange will not happen automatically (unless the remote device initiates it), and must be manually initiated with *gattc_exchange_mtu*. Use the `_IRQ_MTU_EXCHANGED` event to discover the MTU for a given connection.
- `'bond'`: Sets whether bonding will be enabled during pairing. When enabled, pairing requests will set the "bond" flag and the keys will be stored by both devices.
- `'mitm'`: Sets whether MITM-protection is required for pairing.
- `'io'`: Sets the I/O capabilities of this device.

  Available options are:

```
_IO_CAPABILITY_DISPLAY_ONLY = const(0)
_IO_CAPABILITY_DISPLAY_YESNO = const(1)
_IO_CAPABILITY_KEYBOARD_ONLY = const(2)
_IO_CAPABILITY_NO_INPUT_OUTPUT = const(3)
_IO_CAPABILITY_KEYBOARD_DISPLAY = const(4)
```

- `'le_secure'`: Sets whether "LE Secure" pairing is required. Default is false (i.e. allow "Legacy Pairing").

Event Handling

`BLE.`**`irq`**`(`*handler*`, /)`

Registers a callback for events from the BLE stack. The *handler* takes two arguments, `event` (which will be one of the codes below) and `data` (which is an event-specific tuple of values).

Note: As an optimisation to prevent unnecessary allocations, the `addr`, `adv_data`, `char_data`, `notify_data`, and `uuid` entries in the tuples are read-only memoryview instances pointing to *bluetooth*'s internal ringbuffer, and are only valid during the invocation of the IRQ handler function. If your program needs to save one of these values to access after the IRQ handler has returned (e.g. by saving it in a class instance or global variable), then it needs to take a copy of the data, either by using `bytes()` or `bluetooth.UUID()`, like this:

```
connected_addr = bytes(addr)  # equivalently: adv_data, char_data, or notify_data
matched_uuid = bluetooth.UUID(uuid)
```

For example, the IRQ handler for a scan result might inspect the `adv_data` to decide if it's the correct device, and only then copy the address data to be used elsewhere in the program. And to print data from within the IRQ handler, `print(bytes(addr))` will be needed.

An event handler showing all possible events:

```
def bt_irq(event, data):
    if event == _IRQ_CENTRAL_CONNECT:
        # A central has connected to this peripheral.
        conn_handle, addr_type, addr = data
    elif event == _IRQ_CENTRAL_DISCONNECT:
        # A central has disconnected from this peripheral.
        conn_handle, addr_type, addr = data
    elif event == _IRQ_GATTS_WRITE:
        # A client has written to this characteristic or descriptor.
        conn_handle, attr_handle = data
    elif event == _IRQ_GATTS_READ_REQUEST:
        # A client has issued a read. Note: this is only supported on STM32.
        # Return a non-zero integer to deny the read (see below), or zero (or
↪None)
        # to accept the read.
        conn_handle, attr_handle = data
    elif event == _IRQ_SCAN_RESULT:
        # A single scan result.
        addr_type, addr, adv_type, rssi, adv_data = data
    elif event == _IRQ_SCAN_DONE:
        # Scan duration finished or manually stopped.
        pass
    elif event == _IRQ_PERIPHERAL_CONNECT:
        # A successful gap_connect().
        conn_handle, addr_type, addr = data
    elif event == _IRQ_PERIPHERAL_DISCONNECT:
        # Connected peripheral has disconnected.
        conn_handle, addr_type, addr = data
    elif event == _IRQ_GATTC_SERVICE_RESULT:
        # Called for each service found by gattc_discover_services().
        conn_handle, start_handle, end_handle, uuid = data
    elif event == _IRQ_GATTC_SERVICE_DONE:
        # Called once service discovery is complete.
        # Note: Status will be zero on success, implementation-specific value
↪otherwise.
        conn_handle, status = data
    elif event == _IRQ_GATTC_CHARACTERISTIC_RESULT:
        # Called for each characteristic found by gattc_discover_services().
        conn_handle, end_handle, value_handle, properties, uuid = data
    elif event == _IRQ_GATTC_CHARACTERISTIC_DONE:
        # Called once service discovery is complete.
        # Note: Status will be zero on success, implementation-specific value
↪otherwise.
        conn_handle, status = data
    elif event == _IRQ_GATTC_DESCRIPTOR_RESULT:
        # Called for each descriptor found by gattc_discover_descriptors().
        conn_handle, dsc_handle, uuid = data
    elif event == _IRQ_GATTC_DESCRIPTOR_DONE:
        # Called once service discovery is complete.
```

(continues on next page)

(continued from previous page)

```
        # Note: Status will be zero on success, implementation-specific value␣
↪otherwise.
        conn_handle, status = data
    elif event == _IRQ_GATTC_READ_RESULT:
        # A gattc_read() has completed.
        conn_handle, value_handle, char_data = data
    elif event == _IRQ_GATTC_READ_DONE:
        # A gattc_read() has completed.
        # Note: Status will be zero on success, implementation-specific value␣
↪otherwise.
        conn_handle, value_handle, status = data
    elif event == _IRQ_GATTC_WRITE_DONE:
        # A gattc_write() has completed.
        # Note: Status will be zero on success, implementation-specific value␣
↪otherwise.
        conn_handle, value_handle, status = data
    elif event == _IRQ_GATTC_NOTIFY:
        # A server has sent a notify request.
        conn_handle, value_handle, notify_data = data
    elif event == _IRQ_GATTC_INDICATE:
        # A server has sent an indicate request.
        conn_handle, value_handle, notify_data = data
    elif event == _IRQ_GATTS_INDICATE_DONE:
        # A client has acknowledged the indication.
        # Note: Status will be zero on successful acknowledgment, implementation-
↪specific value otherwise.
        conn_handle, value_handle, status = data
    elif event == _IRQ_MTU_EXCHANGED:
        # ATT MTU exchange complete (either initiated by us or the remote device).
        conn_handle, mtu = data
    elif event == _IRQ_L2CAP_ACCEPT:
        # A new channel has been accepted.
        # Return a non-zero integer to reject the connection, or zero (or None)␣
↪to accept.
        conn_handle, cid, psm, our_mtu, peer_mtu = data
    elif event == _IRQ_L2CAP_CONNECT:
        # A new channel is now connected (either as a result of connecting or␣
↪accepting).
        conn_handle, cid, psm, our_mtu, peer_mtu = data
    elif event == _IRQ_L2CAP_DISCONNECT:
        # Existing channel has disconnected (status is zero), or a connection␣
↪attempt failed (non-zero status).
        conn_handle, cid, psm, status = data
    elif event == _IRQ_L2CAP_RECV:
        # New data is available on the channel. Use l2cap_recvinto to read.
        conn_handle, cid = data
    elif event == _IRQ_L2CAP_SEND_READY:
        # A previous l2cap_send that returned False has now completed and the␣
↪channel is ready to send again.
        # If status is non-zero, then the transmit buffer overflowed and the␣
↪application should re-send the data.
        conn_handle, cid, status = data
    elif event == _IRQ_CONNECTION_UPDATE:
        # The remote device has updated connection parameters.
        conn_handle, conn_interval, conn_latency, supervision_timeout, status =␣
↪data
    elif event == _IRQ_ENCRYPTION_UPDATE:
```

(continues on next page)

(continued from previous page)

```
        # The encryption state has changed (likely as a result of pairing or␣
↪bonding).
        conn_handle, encrypted, authenticated, bonded, key_size = data
    elif event == _IRQ_GET_SECRET:
        # Return a stored secret.
        # If key is None, return the index'th value of this sec_type.
        # Otherwise return the corresponding value for this sec_type and key.
        sec_type, index, key = data
        return value
    elif event == _IRQ_SET_SECRET:
        # Save a secret to the store for this sec_type and key.
        sec_type, key, value = data
        return True
    elif event == _IRQ_PASSKEY_ACTION:
        # Respond to a passkey request during pairing.
        # See gap_passkey() for details.
        # action will be an action that is compatible with the configured "io"␣
↪config.
        # passkey will be non-zero if action is "numeric comparison".
        conn_handle, action, passkey = data
```

The event codes are:

```
from micropython import const
_IRQ_CENTRAL_CONNECT = const(1)
_IRQ_CENTRAL_DISCONNECT = const(2)
_IRQ_GATTS_WRITE = const(3)
_IRQ_GATTS_READ_REQUEST = const(4)
_IRQ_SCAN_RESULT = const(5)
_IRQ_SCAN_DONE = const(6)
_IRQ_PERIPHERAL_CONNECT = const(7)
_IRQ_PERIPHERAL_DISCONNECT = const(8)
_IRQ_GATTC_SERVICE_RESULT = const(9)
_IRQ_GATTC_SERVICE_DONE = const(10)
_IRQ_GATTC_CHARACTERISTIC_RESULT = const(11)
_IRQ_GATTC_CHARACTERISTIC_DONE = const(12)
_IRQ_GATTC_DESCRIPTOR_RESULT = const(13)
_IRQ_GATTC_DESCRIPTOR_DONE = const(14)
_IRQ_GATTC_READ_RESULT = const(15)
_IRQ_GATTC_READ_DONE = const(16)
_IRQ_GATTC_WRITE_DONE = const(17)
_IRQ_GATTC_NOTIFY = const(18)
_IRQ_GATTC_INDICATE = const(19)
_IRQ_GATTS_INDICATE_DONE = const(20)
_IRQ_MTU_EXCHANGED = const(21)
_IRQ_L2CAP_ACCEPT = const(22)
_IRQ_L2CAP_CONNECT = const(23)
_IRQ_L2CAP_DISCONNECT = const(24)
_IRQ_L2CAP_RECV = const(25)
_IRQ_L2CAP_SEND_READY = const(26)
_IRQ_CONNECTION_UPDATE = const(27)
_IRQ_ENCRYPTION_UPDATE = const(28)
_IRQ_GET_SECRET = const(29)
_IRQ_SET_SECRET = const(30)
```

For the `_IRQ_GATTS_READ_REQUEST` event, the available return codes are:

```
_GATTS_NO_ERROR = const(0x00)
_GATTS_ERROR_READ_NOT_PERMITTED = const(0x02)
_GATTS_ERROR_WRITE_NOT_PERMITTED = const(0x03)
_GATTS_ERROR_INSUFFICIENT_AUTHENTICATION = const(0x05)
_GATTS_ERROR_INSUFFICIENT_AUTHORIZATION = const(0x08)
_GATTS_ERROR_INSUFFICIENT_ENCRYPTION = const(0x0f)
```

For the `_IRQ_PASSKEY_ACTION` event, the available actions are:

```
_PASSKEY_ACTION_NONE = const(0)
_PASSKEY_ACTION_INPUT = const(2)
_PASSKEY_ACTION_DISPLAY = const(3)
_PASSKEY_ACTION_NUMERIC_COMPARISON = const(4)
```

In order to save space in the firmware, these constants are not included on the *bluetooth* module. Add the ones that you need from the list above to your program.

Broadcaster Role (Advertiser)

BLE.**gap_advertise**(*interval_us*, *adv_data=None*, ***, *resp_data=None*, *connectable=True*)

Starts advertising at the specified interval (in **micro**seconds). This interval will be rounded down to the nearest 625us. To stop advertising, set *interval_us* to `None`.

adv_data and *resp_data* can be any type that implements the buffer protocol (e.g. `bytes`, `bytearray`, `str`). *adv_data* is included in all broadcasts, and *resp_data* is send in reply to an active scan.

Note: if *adv_data* (or *resp_data*) is `None`, then the data passed to the previous call to `gap_advertise` will be reused. This allows a broadcaster to resume advertising with just `gap_advertise(interval_us)`. To clear the advertising payload pass an empty `bytes`, i.e. `b''`.

Observer Role (Scanner)

BLE.**gap_scan**(*duration_ms*, *interval_us=1280000*, *window_us=11250*, *active=False*, */*)

Run a scan operation lasting for the specified duration (in **milli**seconds).

To scan indefinitely, set *duration_ms* to `0`.

To stop scanning, set *duration_ms* to `None`.

Use *interval_us* and *window_us* to optionally configure the duty cycle. The scanner will run for *window_us* **micro**seconds every *interval_us* **micro**seconds for a total of *duration_ms* **milli**seconds. The default interval and window are 1.28 seconds and 11.25 milliseconds respectively (background scanning).

For each scan result the `_IRQ_SCAN_RESULT` event will be raised, with event data `(addr_type, addr, adv_type, rssi, adv_data)`.

`addr_type` values indicate public or random addresses:

- 0x00 - PUBLIC
- 0x01 - RANDOM (either static, RPA, or NRPA, the type is encoded in the address itself)

`adv_type` values correspond to the Bluetooth Specification:

- 0x00 - ADV_IND - connectable and scannable undirected advertising
- 0x01 - ADV_DIRECT_IND - connectable directed advertising
- 0x02 - ADV_SCAN_IND - scannable undirected advertising

- 0x03 - ADV_NONCONN_IND - non-connectable undirected advertising
- 0x04 - SCAN_RSP - scan response

`active` can be set `True` if you want to receive scan responses in the results.

When scanning is stopped (either due to the duration finishing or when explicitly stopped), the `_IRQ_SCAN_DONE` event will be raised.

Central Role

A central device can connect to peripherals that it has discovered using the observer role (see *gap_scan*) or with a known address.

`BLE.`**`gap_connect`**`(`*addr_type*, *addr*, *scan_duration_ms=2000*, *min_conn_interval_us=None*, *max_conn_interval_us=None*, `/`)

Connect to a peripheral.

See *gap_scan* for details about address types.

To cancel an outstanding connection attempt early, call `gap_connect(None)`.

On success, the `_IRQ_PERIPHERAL_CONNECT` event will be raised. If cancelling a connection attempt, the `_IRQ_PERIPHERAL_DISCONNECT` event will be raised.

The device will wait up to *scan_duration_ms* to receive an advertising payload from the device.

The connection interval can be configured in **micro**seconds using either or both of *min_conn_interval_us* and *max_conn_interval_us*. Otherwise a default interval will be chosen, typically between 30000 and 50000 microseconds. A shorter interval will increase throughput, at the expense of power usage.

Peripheral Role

A peripheral device is expected to send connectable advertisements (see *gap_advertise*). It will usually be acting as a GATT server, having first registered services and characteristics using *gatts_register_services*.

When a central connects, the `_IRQ_CENTRAL_CONNECT` event will be raised.

Central & Peripheral Roles

`BLE.`**`gap_disconnect`**`(`*conn_handle*, `/`)

Disconnect the specified connection handle. This can either be a central that has connected to this device (if acting as a peripheral) or a peripheral that was previously connected to by this device (if acting as a central).

On success, the `_IRQ_PERIPHERAL_DISCONNECT` or `_IRQ_CENTRAL_DISCONNECT` event will be raised.

Returns `False` if the connection handle wasn't connected, and `True` otherwise.

GATT Server

A GATT server has a set of registered services. Each service may contain characteristics, which each have a value. Characteristics can also contain descriptors, which themselves have values.

These values are stored locally, and are accessed by their "value handle" which is generated during service registration. They can also be read from or written to by a remote client device. Additionally, a server can "notify" a characteristic to a connected client via a connection handle.

A device in either central or peripheral roles may function as a GATT server, however in most cases it will be more common for a peripheral device to act as the server.

Characteristics and descriptors have a default maximum size of 20 bytes. Anything written to them by a client will be truncated to this length. However, any local write will increase the maximum size, so if you want to allow larger writes from a client to a given characteristic, use *gatts_write* after registration. e.g. `gatts_write(char_handle, bytes(100))`.

BLE.**gatts_register_services**(*services_definition*, /)

Configures the server with the specified services, replacing any existing services.

services_definition is a list of **services**, where each **service** is a two-element tuple containing a UUID and a list of **characteristics**.

Each **characteristic** is a two-or-three-element tuple containing a UUID, a **flags** value, and optionally a list of *descriptors*.

Each **descriptor** is a two-element tuple containing a UUID and a **flags** value.

The **flags** are a bitwise-OR combination of the flags defined below. These set both the behaviour of the characteristic (or descriptor) as well as the security and privacy requirements.

The return value is a list (one element per service) of tuples (each element is a value handle). Characteristics and descriptor handles are flattened into the same tuple, in the order that they are defined.

The following example registers two services (Heart Rate, and Nordic UART):

```
HR_UUID = bluetooth.UUID(0x180D)
HR_CHAR = (bluetooth.UUID(0x2A37), bluetooth.FLAG_READ | bluetooth.FLAG_NOTIFY,)
HR_SERVICE = (HR_UUID, (HR_CHAR,),)
UART_UUID = bluetooth.UUID('6E400001-B5A3-F393-E0A9-E50E24DCCA9E')
UART_TX = (bluetooth.UUID('6E400003-B5A3-F393-E0A9-E50E24DCCA9E'), bluetooth.FLAG_
↪READ | bluetooth.FLAG_NOTIFY,)
UART_RX = (bluetooth.UUID('6E400002-B5A3-F393-E0A9-E50E24DCCA9E'), bluetooth.FLAG_
↪WRITE,)
UART_SERVICE = (UART_UUID, (UART_TX, UART_RX,),)
SERVICES = (HR_SERVICE, UART_SERVICE,)
( (hr,), (tx, rx,), ) = bt.gatts_register_services(SERVICES)
```

The three value handles (`hr`, `tx`, `rx`) can be used with *gatts_read*, *gatts_write*, *gatts_notify*, and *gatts_indicate*.

Note: Advertising must be stopped before registering services.

Available flags for characteristics and descriptors are:

```
from micropython import const
_FLAG_BROADCAST = const(0x0001)
_FLAG_READ = const(0x0002)
_FLAG_WRITE_NO_RESPONSE = const(0x0004)
_FLAG_WRITE = const(0x0008)
_FLAG_NOTIFY = const(0x0010)
```

(continues on next page)

(continued from previous page)

```
_FLAG_INDICATE = const(0x0020)
_FLAG_AUTHENTICATED_SIGNED_WRITE = const(0x0040)

_FLAG_AUX_WRITE = const(0x0100)
_FLAG_READ_ENCRYPTED = const(0x0200)
_FLAG_READ_AUTHENTICATED = const(0x0400)
_FLAG_READ_AUTHORIZED = const(0x0800)
_FLAG_WRITE_ENCRYPTED = const(0x1000)
_FLAG_WRITE_AUTHENTICATED = const(0x2000)
_FLAG_WRITE_AUTHORIZED = const(0x4000)
```

As for the IRQs above, any required constants should be added to your Python code.

BLE.**gatts_read**(*value_handle*, /)

Reads the local value for this handle (which has either been written by *gatts_write* or by a remote client).

BLE.**gatts_write**(*value_handle*, *data*, *send_update=False*, /)

Writes the local value for this handle, which can be read by a client.

If *send_update* is `True`, then any subscribed clients will be notified (or indicated, depending on what they're subscribed to and which operations the characteristic supports) about this write.

BLE.**gatts_notify**(*conn_handle*, *value_handle*, *data=None*, /)

Sends a notification request to a connected client.

If *data* is `None` (the default), then the current local value (as set with *gatts_write*) will be sent.

Otherwise, if *data* is not `None`, then that value is sent to the client as part of the notification. The local value will not be modified.

Note: The notification will be sent regardless of the subscription status of the client to this characteristic.

BLE.**gatts_indicate**(*conn_handle*, *value_handle*, *data=None*, /)

Sends a indication request to a connected client.

If *data* is `None` (the default), then the current local value (as set with *gatts_write*) will be sent.

Otherwise, if *data* is not `None`, then that value is sent to the client as part of the indication. The local value will not be modified.

On acknowledgment (or failure, e.g. timeout), the `_IRQ_GATTS_INDICATE_DONE` event will be raised.

Note: The indication will be sent regardless of the subscription status of the client to this characteristic.

BLE.**gatts_set_buffer**(*value_handle*, *len*, *append=False*, /)

Sets the internal buffer size for a value in bytes. This will limit the largest possible write that can be received. The default is 20.

Setting *append* to `True` will make all remote writes append to, rather than replace, the current value. At most *len* bytes can be buffered in this way. When you use *gatts_read*, the value will be cleared after reading. This feature is useful when implementing something like the Nordic UART Service.

GATT Client

A GATT client can discover and read/write characteristics on a remote GATT server.

It is more common for a central role device to act as the GATT client, however it's also possible for a peripheral to act as a client in order to discover information about the central that has connected to it (e.g. to read the device name from the device information service).

`BLE.`**`gattc_discover_services`**(*conn_handle*, *uuid=None*, */*)

Query a connected server for its services.

Optionally specify a service *uuid* to query for that service only.

For each service discovered, the `_IRQ_GATTC_SERVICE_RESULT` event will be raised, followed by `_IRQ_GATTC_SERVICE_DONE` on completion.

`BLE.`**`gattc_discover_characteristics`**(*conn_handle*, *start_handle*, *end_handle*, *uuid=None*, */*)

Query a connected server for characteristics in the specified range.

Optionally specify a characteristic *uuid* to query for that characteristic only.

You can use `start_handle=1`, `end_handle=0xffff` to search for a characteristic in any service.

For each characteristic discovered, the `_IRQ_GATTC_CHARACTERISTIC_RESULT` event will be raised, followed by `_IRQ_GATTC_CHARACTERISTIC_DONE` on completion.

`BLE.`**`gattc_discover_descriptors`**(*conn_handle*, *start_handle*, *end_handle*, */*)

Query a connected server for descriptors in the specified range.

For each descriptor discovered, the `_IRQ_GATTC_DESCRIPTOR_RESULT` event will be raised, followed by `_IRQ_GATTC_DESCRIPTOR_DONE` on completion.

`BLE.`**`gattc_read`**(*conn_handle*, *value_handle*, */*)

Issue a remote read to a connected server for the specified characteristic or descriptor handle.

When a value is available, the `_IRQ_GATTC_READ_RESULT` event will be raised. Additionally, the `_IRQ_GATTC_READ_DONE` will be raised.

`BLE.`**`gattc_write`**(*conn_handle*, *value_handle*, *data*, *mode=0*, */*)

Issue a remote write to a connected server for the specified characteristic or descriptor handle.

The argument *mode* specifies the write behaviour, with the currently supported values being:

- `mode=0` (default) is a write-without-response: the write will be sent to the remote server but no confirmation will be returned, and no event will be raised.
- `mode=1` is a write-with-response: the remote server is requested to send a response/acknowledgement that it received the data.

If a response is received from the remote server the `_IRQ_GATTC_WRITE_DONE` event will be raised.

`BLE.`**`gattc_exchange_mtu`**(*conn_handle*, */*)

Initiate MTU exchange with a connected server, using the preferred MTU set using `BLE.config(mtu=value)`.

The `_IRQ_MTU_EXCHANGED` event will be raised when MTU exchange completes.

Note: MTU exchange is typically initiated by the central. When using the BlueKitchen stack in the central role, it does not support a remote peripheral initiating the MTU exchange. NimBLE works for both roles.

L2CAP connection-oriented-channels

This feature allows for socket-like data exchange between two BLE devices. Once the devices are connected via GAP, either device can listen for the other to connect on a numeric PSM (Protocol/Service Multiplexer).

Note: This is currently only supported when using the NimBLE stack on STM32 and Unix (not ESP32). Only one L2CAP channel may be active at a given time (i.e. you cannot connect while listening).

Active L2CAP channels are identified by the connection handle that they were established on and a CID (channel ID).

Connection-oriented channels have built-in credit-based flow control. Unlike ATT, where devices negotiate a shared MTU, both the listening and connecting devices each set an independent MTU which limits the maximum amount of outstanding data that the remote device can send before it is fully consumed in *l2cap_recvinto*.

BLE.**l2cap_listen**(*psm*, *mtu*, /)

Start listening for incoming L2CAP channel requests on the specified *psm* with the local MTU set to *mtu*.

When a remote device initiates a connection, the `_IRQ_L2CAP_ACCEPT` event will be raised, which gives the listening server a chance to reject the incoming connection (by returning a non-zero integer).

Once the connection is accepted, the `_IRQ_L2CAP_CONNECT` event will be raised, allowing the server to obtain the channel id (CID) and the local and remote MTU.

Note: It is not currently possible to stop listening.

BLE.**l2cap_connect**(*conn_handle*, *psm*, *mtu*, /)

Connect to a listening peer on the specified *psm* with local MTU set to *mtu*.

On successful connection, the `_IRQ_L2CAP_CONNECT` event will be raised, allowing the client to obtain the CID and the local and remote (peer) MTU.

An unsuccessful connection will raise the `_IRQ_L2CAP_DISCONNECT` event with a non-zero status.

BLE.**l2cap_disconnect**(*conn_handle*, *cid*, /)

Disconnect an active L2CAP channel with the specified *conn_handle* and *cid*.

BLE.**l2cap_send**(*conn_handle*, *cid*, *buf*, /)

Send the specified *buf* (which must support the buffer protocol) on the L2CAP channel identified by *conn_handle* and *cid*.

The specified buffer cannot be larger than the remote (peer) MTU, and no more than twice the size of the local MTU.

This will return `False` if the channel is now "stalled", which means that *l2cap_send* must not be called again until the `_IRQ_L2CAP_SEND_READY` event is received (which will happen when the remote device grants more credits, typically after it has received and processed the data).

BLE.**l2cap_recvinto**(*conn_handle*, *cid*, *buf*, /)

Receive data from the specified *conn_handle* and *cid* into the provided *buf* (which must support the buffer protocol, e.g. bytearray or memoryview).

Returns the number of bytes read from the channel.

If *buf* is None, then returns the number of bytes available.

Note: After receiving the `_IRQ_L2CAP_RECV` event, the application should continue calling *l2cap_recvinto* until no more bytes are available in the receive buffer (typically up to the size of the remote (peer) MTU).

Until the receive buffer is empty, the remote device will not be granted more channel credits and will be unable to send any more data.

Pairing and bonding

Pairing allows a connection to be encrypted and authenticated via exchange of secrets (with optional MITM protection via passkey authentication).

Bonding is the process of storing those secrets into non-volatile storage. When bonded, a device is able to resolve a resolvable private address (RPA) from another device based on the stored identity resolving key (IRK). To support bonding, an application must implement the `_IRQ_GET_SECRET` and `_IRQ_SET_SECRET` events.

Note: This is currently only supported when using the NimBLE stack on ESP32, STM32 and Unix.

`BLE.`**`gap_pair`**`(`*conn_handle*`, /)`

Initiate pairing with the remote device.

Before calling this, ensure that the `io`, `mitm`, `le_secure`, and `bond` configuration options are set (via *`config`*).

On successful pairing, the `_IRQ_ENCRYPTION_UPDATE` event will be raised.

`BLE.`**`gap_passkey`**`(`*conn_handle*`, `*action*`, `*passkey*`, /)`

Respond to a `_IRQ_PASSKEY_ACTION` event for the specified *conn_handle* and *action*.

The *passkey* is a numeric value and will depend on on the *action* (which will depend on what I/O capability has been set):

- When the *action* is `_PASSKEY_ACTION_INPUT`, then the application should prompt the user to enter the passkey that is shown on the remote device.
- When the *action* is `_PASSKEY_ACTION_DISPLAY`, then the application should generate a random 6-digit passkey and show it to the user.
- When the *action* is `_PASSKEY_ACTION_NUMERIC_COMPARISON`, then the application should show the passkey that was provided in the `_IRQ_PASSKEY_ACTION` event and then respond with either `0` (cancel pairing), or `1` (accept pairing).

class UUID

Constructor

`class` `bluetooth.`**`UUID`**`(`*value*`, /)`

Creates a UUID instance with the specified **value**.

The **value** can be either:

- A 16-bit integer. e.g. `0x2908`.
- An object with the buffer protocol and that is 2, 4 or 16 bytes long, e.g. `b'\x08\x29'`.
- A 128-bit UUID string. e.g. `'6E400001-B5A3-F393-E0A9-E50E24DCCA9E'`.

1.2.2 btree – simple BTree database

The btree module implements a simple key-value database using external storage (disk files, or in general case, a random-access *stream*). Keys are stored sorted in the database, and besides efficient retrieval by a key value, a database also supports efficient ordered range scans (retrieval of values with the keys in a given range). On the application interface side, BTree database work as close a possible to a way standard *dict* type works, one notable difference is that both keys and values must be *bytes*-like objects (so, if you want to store objects of other types, you need to first serialize them to *str* or *bytes* or another type that supports the buffer protocol).

The module is based on the well-known BerkelyDB library, version 1.xx.

Example:

```
import btree

# First, we need to open a stream which holds a database
# This is usually a file, but can be in-memory database
# using io.BytesIO, a raw flash partition, etc.
# Oftentimes, you want to create a database file if it doesn't
# exist and open if it exists. Idiom below takes care of this.
# DO NOT open database with "a+b" access mode.
try:
    f = open("mydb", "r+b")
except OSError:
    f = open("mydb", "w+b")

# Now open a database itself
db = btree.open(f)

# The keys you add will be sorted internally in the database
db[b"3"] = b"three"
db[b"1"] = b"one"
db[b"2"] = b"two"

# Assume that any changes are cached in memory unless
# explicitly flushed (or database closed). Flush database
# at the end of each "transaction".
db.flush()

# Prints b'two'
print(db[b"2"])

# Iterate over sorted keys in the database, starting from b"2"
# until the end of the database, returning only values.
# Mind that arguments passed to values() method are *key* values.
# Prints:
#   b'two'
#   b'three'
for word in db.values(b"2"):
    print(word)

del db[b"2"]

# No longer true, prints False
print(b"2" in db)

# Prints:
#  b"1"
```

(continues on next page)

(continued from previous page)

```
#  b"3"
for key in db:
    print(key)

db.close()

# Don't forget to close the underlying stream!
f.close()
```

Functions

btree.**open**(*stream*, *, *flags=0*, *pagesize=0*, *cachesize=0*, *minkeypage=0*)

Open a database from a random-access *stream* (like an open file). All other parameters are optional and keyword-only, and allow to tweak advanced parameters of the database operation (most users will not need them):

- *flags* - Currently unused.
- *pagesize* - Page size used for the nodes in BTree. Acceptable range is 512-65536. If 0, a port-specific default will be used, optimized for port's memory usage and/or performance.
- *cachesize* - Suggested memory cache size in bytes. For a board with enough memory using larger values may improve performance. Cache policy is as follows: entire cache is not allocated at once; instead, accessing a new page in database will allocate a memory buffer for it, until value specified by *cachesize* is reached. Then, these buffers will be managed using LRU (least recently used) policy. More buffers may still be allocated if needed (e.g., if a database contains big keys and/or values). Allocated cache buffers aren't reclaimed.
- *minkeypage* - Minimum number of keys to store per page. Default value of 0 equivalent to 2.

Returns a BTree object, which implements a dictionary protocol (set of methods), and some additional methods described below.

Methods

btree.**close**()

Close the database. It's mandatory to close the database at the end of processing, as some unwritten data may be still in the cache. Note that this does not close underlying stream with which the database was opened, it should be closed separately (which is also mandatory to make sure that data flushed from buffer to the underlying storage).

btree.**flush**()

Flush any data in cache to the underlying stream.

btree.**__getitem__**(*key*)

btree.**get**(*key*, *default=None*, */*)

btree.**__setitem__**(*key*, *val*)

btree.**__delitem__**(*key*)

btree.**__contains__**(*key*)

Standard dictionary methods.

btree.**__iter__**()

A BTree object can be iterated over directly (similar to a dictionary) to get access to all keys in order.

btree.**keys**([*start_key*[, *end_key*[, *flags*]]])

btree.**values**([*start_key*[, *end_key*[, *flags*]]])

btree.**items**(*start_key*[, *end_key*[, *flags*]])

These methods are similar to standard dictionary methods, but also can take optional parameters to iterate over a key sub-range, instead of the entire database. Note that for all 3 methods, *start_key* and *end_key* arguments represent key values. For example, *values()* method will iterate over values corresponding to they key range given. None values for *start_key* means "from the first key", no *end_key* or its value of None means "until the end of database". By default, range is inclusive of *start_key* and exclusive of *end_key*, you can include *end_key* in iteration by passing *flags* of *btree.INCL*. You can iterate in descending key direction by passing *flags* of *btree.DESC*. The flags values can be ORed together.

Constants

btree.**INCL**

A flag for *btree.keys()*, *btree.values()*, *btree.items()* methods to specify that scanning should be inclusive of the end key.

btree.**DESC**

A flag for *btree.keys()*, *btree.values()*, *btree.items()* methods to specify that scanning should be in descending direction of keys.

1.2.3 cryptolib – cryptographic ciphers

Classes

class cryptolib.**aes**

classmethod **__init__**(*key*, *mode*[, *IV*])

Initialize cipher object, suitable for encryption/decryption. Note: after initialization, cipher object can be use only either for encryption or decryption. Running decrypt() operation after encrypt() or vice versa is not supported.

Parameters are:

- *key* is an encryption/decryption key (bytes-like).
- *mode* is:
  - 1 (or cryptolib.MODE_ECB if it exists) for Electronic Code Book (ECB).
  - 2 (or cryptolib.MODE_CBC if it exists) for Cipher Block Chaining (CBC).
  - 6 (or cryptolib.MODE_CTR if it exists) for Counter mode (CTR).
- *IV* is an initialization vector for CBC mode.
- For Counter mode, *IV* is the initial value for the counter.

encrypt(*in_buf*[, *out_buf*])

Encrypt *in_buf*. If no *out_buf* is given result is returned as a newly allocated *bytes* object. Otherwise, result is written into mutable buffer *out_buf*. *in_buf* and *out_buf* can also refer to the same mutable buffer, in which case data is encrypted in-place.

decrypt(*in_buf*[, *out_buf*])

Like *encrypt()*, but for decryption.

1.2.4 `deflate` – deflate compression & decompression

This module allows compression and decompression of binary data with the DEFLATE algorithm (commonly used in the zlib library and gzip archiver).

Availability:

- Added in MicroPython v1.21.
- Decompression: Enabled via the `MICROPY_PY_DEFLATE` build option, on by default on ports with the "extra features" level or higher (which is most boards).
- Compression: Enabled via the `MICROPY_PY_DEFLATE_COMPRESS` build option, on by default on ports with the "full features" level or higher (generally this means you need to build your own firmware to enable this).

Classes

class deflate.**DeflateIO**(*stream*, *format=AUTO*, *wbits=0*, *close=False*, */*)

This class can be used to wrap a *stream* which is any *stream-like* object such as a file, socket, or stream (including *io.BytesIO*). It is itself a stream and implements the standard read/readinto/write/close methods.

The *stream* must be a blocking stream. Non-blocking streams are currently not supported.

The *format* can be set to any of the constants defined below, and defaults to `AUTO` which for decompressing will auto-detect gzip or zlib streams, and for compressing it will generate a raw stream.

The *wbits* parameter sets the base-2 logarithm of the DEFLATE dictionary window size. So for example, setting *wbits* to `10` sets the window size to 1024 bytes. Valid values are `5` to `15` inclusive (corresponding to window sizes of 32 to 32k bytes).

If *wbits* is set to `0` (the default), then for compression a window size of 256 bytes will be used (as if *wbits* was set to 8). For decompression, it depends on the format:

- `RAW` will use 256 bytes (corresponding to *wbits* set to 8).
- `ZLIB` (or `AUTO` with zlib detected) will use the value from the zlib header.
- `GZIP` (or `AUTO` with gzip detected) will use 32 kilobytes (corresponding to *wbits* set to 15).

See the *window size* notes below for more information about the window size, zlib, and gzip streams.

If *close* is set to `True` then the underlying stream will be closed automatically when the *deflate.DeflateIO* stream is closed. This is useful if you want to return a *deflate.DeflateIO* stream that wraps another stream and not have the caller need to know about managing the underlying stream.

If compression is enabled, a given *deflate.DeflateIO* instance supports both reading and writing. For example, a bidirectional stream like a socket can be wrapped, which allows for compression/decompression in both directions.

Constants

deflate.**AUTO**

deflate.**RAW**

deflate.**ZLIB**

deflate.**GZIP**

Supported values for the *format* parameter.

Examples

A typical use case for *deflate.DeflateIO* is to read or write a compressed file from storage:

```
import deflate

# Writing a zlib-compressed stream (uses the default window size of 256 bytes).
with open("data.gz", "wb") as f:
    with deflate.DeflateIO(f, deflate.ZLIB) as d:
        # Use d.write(...) etc

# Reading a zlib-compressed stream (auto-detect window size).
with open("data.z", "rb") as f:
    with deflate.DeflateIO(f, deflate.ZLIB) as d:
        # Use d.read(), d.readinto(), etc.
```

Because *deflate.DeflateIO* is a stream, it can be used for example with *json.dump()* and *json.load()* (and any other places streams can be used):

```
import deflate, json

# Write a dictionary as JSON in gzip format, with a
# small (64 byte) window size.
config = { ... }
with open("config.gz", "wb") as f:
    with deflate.DeflateIO(f, deflate.GZIP, 6) as f:
        json.dump(config, f)

# Read back that dictionary.
with open("config.gz", "rb") as f:
    with deflate.DeflateIO(f, deflate.GZIP, 6) as f:
        config = json.load(f)
```

If your source data is not in a stream format, you can use *io.BytesIO* to turn it into a stream suitable for use with *deflate.DeflateIO*:

```
import deflate, io

# Decompress a bytes/bytearray value.
compressed_data = get_data_z()
with deflate.DeflateIO(io.BytesIO(compressed_data), deflate.ZLIB) as d:
    decompressed_data = d.read()

# Compress a bytes/bytearray value.
uncompressed_data = get_data()
stream = io.BytesIO()
with deflate.DeflateIO(stream, deflate.ZLIB) as d:
    d.write(uncompressed_data)
compressed_data = stream.getvalue()
```

Deflate window size

The window size limits how far back in the stream the (de)compressor can reference. Increasing the window size will improve compression, but will require more memory and make the compressor slower.

If an input stream was compressed a given window size, then *DeflateIO* using a smaller window size will fail mid-way during decompression with *OSError*, but only if a back-reference actually refers back further than the decompressor's window size. This means it may be possible to decompress with a smaller window size. For example, this would trivially be the case if the original uncompressed data is shorter than the window size.

Decompression

The zlib format includes a header which specifies the window size that was used to compress the data. This indicates the maximum window size required to decompress this stream. If this header value is less than the specified *wbits* value (or if *wbits* is unset), then the header value will be used.

The gzip format does not include the window size in the header, and assumes that all gzip compressors (e.g. the `gzip` utility, or CPython's implementation of *gzip.GzipFile*) use the maximum window size of 32kiB. For this reason, if the *wbits* parameter is not set, the decompressor will use a 32 kiB window size (corresponding to *wbits* set to 15). This means that to be able to decompress an arbitrary gzip stream, you must have at least this much RAM available. If you control the source data, consider instead using the zlib format with a smaller window size.

The raw format has no header and therefore does not include any information about the window size. If *wbits* is not set, then it will default to a window size of 256 bytes, which may not be large enough for a given stream. Therefore it is recommended that you should always explicitly set *wbits* if using the raw format.

Compression

For compression, MicroPython will default to a window size of 256 bytes for all formats. This provides a reasonable amount of compression with minimal memory usage and fast compression time, and will generate output that will work with any decompressor.

1.2.5 `framebuf` — frame buffer manipulation

This module provides a general frame buffer which can be used to create bitmap images, which can then be sent to a display.

class FrameBuffer

The FrameBuffer class provides a pixel buffer which can be drawn upon with pixels, lines, rectangles, ellipses, polygons, text and even other FrameBuffers. It is useful when generating output for displays.

For example:

```
import framebuf

# FrameBuffer needs 2 bytes for every RGB565 pixel
fbuf = framebuf.FrameBuffer(bytearray(100 * 10 * 2), 100, 10, framebuf.RGB565)

fbuf.fill(0)
fbuf.text('MicroPython!', 0, 0, 0xffff)
fbuf.hline(0, 9, 96, 0xffff)
```

Constructors

class framebuf.**FrameBuffer**(*buffer*, *width*, *height*, *format*, *stride=width*, */*)

Construct a FrameBuffer object. The parameters are:

- *buffer* is an object with a buffer protocol which must be large enough to contain every pixel defined by the width, height and format of the FrameBuffer.
- *width* is the width of the FrameBuffer in pixels
- *height* is the height of the FrameBuffer in pixels
- *format* specifies the type of pixel used in the FrameBuffer; permissible values are listed under Constants below. These set the number of bits used to encode a color value and the layout of these bits in *buffer*. Where a color value c is passed to a method, c is a small integer with an encoding that is dependent on the format of the FrameBuffer.
- *stride* is the number of pixels between each horizontal line of pixels in the FrameBuffer. This defaults to *width* but may need adjustments when implementing a FrameBuffer within another larger FrameBuffer or screen. The *buffer* size must accommodate an increased step size.

One must specify valid *buffer*, *width*, *height*, *format* and optionally *stride*. Invalid *buffer* size or dimensions may lead to unexpected errors.

Drawing primitive shapes

The following methods draw shapes onto the FrameBuffer.

FrameBuffer.**fill**(*c*)

Fill the entire FrameBuffer with the specified color.

FrameBuffer.**pixel**(*x*, *y*[, *c*])

If *c* is not given, get the color value of the specified pixel. If *c* is given, set the specified pixel to the given color.

FrameBuffer.**hline**(*x*, *y*, *w*, *c*)

FrameBuffer.**vline**(*x*, *y*, *h*, *c*)

FrameBuffer.**line**(*x1*, *y1*, *x2*, *y2*, *c*)

Draw a line from a set of coordinates using the given color and a thickness of 1 pixel. The *line* method draws the line up to a second set of coordinates whereas the *hline* and *vline* methods draw horizontal and vertical lines respectively up to a given length.

FrameBuffer.**rect**(*x*, *y*, *w*, *h*, *c*[, *f*])

Draw a rectangle at the given location, size and color.

The optional *f* parameter can be set to True to fill the rectangle. Otherwise just a one pixel outline is drawn.

FrameBuffer.**ellipse**(*x*, *y*, *xr*, *yr*, *c*[, *f*, *m*])

Draw an ellipse at the given location. Radii *xr* and *yr* define the geometry; equal values cause a circle to be drawn. The *c* parameter defines the color.

The optional *f* parameter can be set to True to fill the ellipse. Otherwise just a one pixel outline is drawn.

The optional *m* parameter enables drawing to be restricted to certain quadrants of the ellipse. The LS four bits determine which quadrants are to be drawn, with bit 0 specifying Q1, b1 Q2, b2 Q3 and b3 Q4. Quadrants are numbered counterclockwise with Q1 being top right.

FrameBuffer.**poly**(*x*, *y*, *coords*, *c*[, *f*])

Given a list of coordinates, draw an arbitrary (convex or concave) closed polygon at the given x, y location using the given color.

The *coords* must be specified as a *array* of integers, e.g. `array('h', [x0, y0, x1, y1, ... xn, yn])`.

The optional *f* parameter can be set to `True` to fill the polygon. Otherwise just a one pixel outline is drawn.

Drawing text

FrameBuffer.**text**(*s*, *x*, *y*[, *c*])

Write text to the FrameBuffer using the coordinates as the upper-left corner of the text. The color of the text can be defined by the optional argument but is otherwise a default value of 1. All characters have dimensions of 8x8 pixels and there is currently no way to change the font.

Other methods

FrameBuffer.**scroll**(*xstep*, *ystep*)

Shift the contents of the FrameBuffer by the given vector. This may leave a footprint of the previous colors in the FrameBuffer.

FrameBuffer.**blit**(*fbuf*, *x*, *y*, *key=-1*, *palette=None*)

Draw another FrameBuffer on top of the current one at the given coordinates. If *key* is specified then it should be a color integer and the corresponding color will be considered transparent: all pixels with that color value will not be drawn. (If the *palette* is specified then the *key* is compared to the value from *palette*, not to the value directly from *fbuf*.)

fbuf can be another FrameBuffer instance, or a tuple or list of the form:

```
(buffer, width, height, format)
```

or:

```
(buffer, width, height, format, stride)
```

This matches the signature of the FrameBuffer constructor, and the elements of the tuple/list are the same as the arguments to the constructor except that the *buffer* here can be read-only.

The *palette* argument enables blitting between FrameBuffers with differing formats. Typical usage is to render a monochrome or grayscale glyph/icon to a color display. The *palette* is a FrameBuffer instance whose format is that of the current FrameBuffer. The *palette* height is one pixel and its pixel width is the number of colors in the source FrameBuffer. The *palette* for an N-bit source needs 2**N pixels; the *palette* for a monochrome source would have 2 pixels representing background and foreground colors. The application assigns a color to each pixel in the *palette*. The color of the current pixel will be that of that *palette* pixel whose x position is the color of the corresponding source pixel.

Constants

framebuf.**MONO_VLSB**

Monochrome (1-bit) color format This defines a mapping where the bits in a byte are vertically mapped with bit 0 being nearest the top of the screen. Consequently each byte occupies 8 vertical pixels. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered at locations starting at the leftmost edge, 8 pixels lower.

framebuf.**MONO_HLSB**

Monochrome (1-bit) color format This defines a mapping where the bits in a byte are horizontally mapped. Each byte occupies 8 horizontal pixels with bit 7 being the leftmost. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered on the next row, one pixel lower.

framebuf.**MONO_HMSB**

Monochrome (1-bit) color format This defines a mapping where the bits in a byte are horizontally mapped. Each byte occupies 8 horizontal pixels with bit 0 being the leftmost. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered on the next row, one pixel lower.

framebuf.**RGB565**

Red Green Blue (16-bit, 5+6+5) color format

framebuf.**GS2_HMSB**

Grayscale (2-bit) color format

framebuf.**GS4_HMSB**

Grayscale (4-bit) color format

framebuf.**GS8**

Grayscale (8-bit) color format

1.2.6 machine — functions related to the hardware

The machine module contains specific functions related to the hardware on a particular board. Most functions in this module allow to achieve direct and unrestricted access to and control of hardware blocks on a system (like CPU, timers, buses, etc.). Used incorrectly, this can lead to malfunction, lockups, crashes of your board, and in extreme cases, hardware damage.

Memory access

The module exposes three objects used for raw memory access.

machine.**mem8**

Read/write 8 bits of memory.

machine.**mem16**

Read/write 16 bits of memory.

machine.**mem32**

Read/write 32 bits of memory.

Use subscript notation [...] to index these objects with the address of interest. Note that the address is the byte address, regardless of the size of memory being accessed.

Example use (registers are specific to an stm32 microcontroller):

```
import machine
from micropython import const

GPIOA = const(0x48000000)
GPIO_BSRR = const(0x18)
GPIO_IDR = const(0x10)

# set PA2 high
machine.mem32[GPIOA + GPIO_BSRR] = 1 << 2

# read PA3
value = (machine.mem32[GPIOA + GPIO_IDR] >> 3) & 1
```

Note: the returned values are signed integers. Example: reading the cpuid register on esp8266

```
value = mem32[0x40001000]
```

will return a negative value, that could be counter-intuitive.

To always read a positive integer

```
value = mem32[0x40001000] & 0xffffffff
```

machine.**mem_backup**(*region=0*)

Return a writable *memoryview* over a persistent hardware memory region that survives at least *Soft Reset* on all ports; battery-backed ports also survive power-off. Per-port persistence guarantees vary, see the table below.

region selects which backup region to access (default 0, the primary region). Pass `-1` to get a tuple of all available regions instead.

The element type depends on the port's hardware alignment requirements: `'B'` (unsigned byte) on ports with byte-addressable backup memory, `'I'` (unsigned 32-bit) on ports backed by word-sized registers. Use `mem.itemsize` to discover the access granularity at runtime.

The total size in bytes is `len(mem) * mem.itemsize`, where `len(mem)` is the number of elements and `mem.itemsize` is the size of each element. For example, on a port with 4 word-sized registers, `len(mem)` is 4 and `mem.itemsize` is 4, giving 16 bytes total. On a port with 4096 bytes of byte-addressable backup SRAM, `len(mem)` is 4096 and `mem.itemsize` is 1.

Cross-port guarantees for portable code: `mem.itemsize` is either `1` or `4`; valid indices are `0..len(mem)-1`; out-of-range access raises `IndexError`; values are stored in host-native byte order. Region index semantics are not portable, see notes below for `stm32` in particular.

Usage:

```
import machine

mem = machine.mem_backup()
mem[0] = 0x12345678                # write element 0
print(hex(mem[0]))                 # read element 0
print(len(mem))                    # number of elements
print(mem.itemsize)                # bytes per element
print(len(mem) * mem.itemsize)     # total bytes available

# Discover all available regions
for i, r in enumerate(machine.mem_backup(-1)):
    print(i, len(r), r.itemsize)
```

The total byte size and backing hardware vary by port:

Port	Backing storage	Total bytes	Battery-backed
alif	Backup SRAM	4080	yes
esp32	RTC slow memory	2048	no
mimxrt	SNVS LPGPR registers (4 per chip)	12-16	yes
nrf	POWER GPREGRET registers	1-2	no
rp2	Watchdog scratch registers	28-60	no
samd	Backup RAM (SAMD51 only)	8192	yes
stm32	Backup SRAM + BKP registers (F4/F7/H5/H7/U5/N6)	2048-8192	yes
stm32	RTC BKP registers (other families)	20-128	yes

Note: On esp32 and rp2, data persists across *Soft Reset*, `machine.reset()` and `machine.deepsleep()` wake but is lost on power-off and on poweron-style resets. On esp32 in particular this includes pressing the EN/RESET button on most dev boards, which the chip reports as a power-on reset.

Some ports split backup storage across multiple regions, or exclude registers reserved by the bootloader or system firmware:

Port	Register(s)	Note
mimxrt	LPGPR[3]	Excluded; used by TinyUF2 (when used)
rp2	scratch[4]	Excluded; used by pico-sdk on reset
rp2	powman scratch[0..7]	Region 2 on RP2350 only
stm32	BKP registers	Region 1 on BKPSRAM families (F4/F7/H5/H7/U5/N6)

Use `machine.mem_backup(-1)` to discover available regions and their sizes.

On stm32 the region index does not have a uniform meaning across boards: region 0 is BKPSRAM (`itemsize=1`) on BKPSRAM families and BKP registers (`itemsize=4`) on others. Portable code should branch on `mem.itemsize` before structuring data.

Some registers within a region are accessible but reserved by convention and should not be overwritten. The BKP register file is region 1 on BKPSRAM families and region 0 on the others:

Port	Register(s)	Used by
stm32	BKP0R	Arduino bootloader (Portenta H7, Giga, Opta, Nicla)
stm32	BKP16R-BKP18R	`rfcore_firmware.py` on STM32WB
stm32	last BKP reg	clock frequency (`MICROPY_HW_CLK_LAST_FREQ`)
stm32	BKP31R (N6)	mboot bootloader entry

The buffer allows direct register access and can be combined with `uctypes` for structured layouts:

```
import machine, uctypes

mem = machine.mem_backup()

# Structured access via uctypes (check len(mem) for your board)
layout = {
    "flags": (0 * 4, uctypes.UINT32),     # register 0
    "counter": (1 * 4, uctypes.UINT32),   # register 1
}
```

(continues on next page)

(continued from previous page)

```
regs = uctypes.struct(uctypes.addressof(mem), layout)
regs.flags = 0x01
print(regs.counter)
```

Availability: alif, esp32, mimxrt, nrf, rp2, samd, stm32 ports.

Reset related functions

machine.**reset**()

Hard resets the device in a manner similar to pushing the external RESET button.

machine.**soft_reset**()

Performs a *soft reset* of the interpreter, deleting all Python objects and resetting the Python heap.

machine.**reset_cause**()

Get the reset cause. See *constants* for the possible return values.

machine.**bootloader**([*value*])

Reset the device and enter its bootloader. This is typically used to put the device into a state where it can be programmed with new firmware.

Some ports support passing in an optional *value* argument which can control which bootloader to enter, what to pass to it, or other things.

Interrupt related functions

The following functions allow control over interrupts. Some systems require interrupts to operate correctly so disabling them for long periods may compromise core functionality, for example watchdog timers may trigger unexpectedly. Interrupts should only be disabled for a minimum amount of time and then re-enabled to their previous state. For example:

```
import machine

# Disable interrupts
state = machine.disable_irq()

# Do a small amount of time-critical work here

# Enable interrupts
machine.enable_irq(state)
```

machine.**disable_irq**()

Disable interrupt requests. Returns the previous IRQ state which should be considered an opaque value. This return value should be passed to the *enable_irq()* function to restore interrupts to their original state, before *disable_irq()* was called.

machine.**enable_irq**(*state*)

Re-enable interrupt requests. The *state* parameter should be the value that was returned from the most recent call to the *disable_irq()* function.

Power related functions

machine.**freq**([*hz*])

Returns the CPU frequency in hertz.

On some ports this can also be used to set the CPU frequency by passing in *hz*.

machine.**idle**()

Gates the clock to the CPU, useful to reduce power consumption at any time during short or long periods. Peripherals continue working and execution resumes as soon as any interrupt is triggered, or at most one millisecond after the CPU was paused.

It is recommended to call this function inside any tight loop that is continuously checking for an external change (i.e. polling). This will reduce power consumption without significantly impacting performance. To reduce power consumption further then see the *lightsleep()*, *time.sleep()* and *time.sleep_ms()* functions.

machine.**sleep**()

Note: This function is deprecated, use *lightsleep()* instead with no arguments.

machine.**lightsleep**([*time_ms*])

machine.**deepsleep**([*time_ms*])

Stops execution in an attempt to enter a low power state.

If *time_ms* is specified then this will be the maximum time in milliseconds that the sleep will last for. Otherwise the sleep can last indefinitely.

With or without a timeout, execution may resume at any time if there are events that require processing. Such events, or wake sources, should be configured before sleeping, like *Pin* change or *RTC* timeout.

The precise behaviour and power-saving capabilities of lightsleep and deepsleep is highly dependent on the underlying hardware, but the general properties are:

- A lightsleep has full RAM and state retention. Upon wake execution is resumed from the point where the sleep was requested, with all subsystems operational.
- A deepsleep may not retain RAM or any other state of the system (for example peripherals or network interfaces). Upon wake execution is resumed from the main script, similar to a hard or power-on reset. The *reset_cause()* function will return *machine.DEEPSLEEP* and this can be used to distinguish a deepsleep wake from other resets.

machine.**wake_reason**()

Get the wake reason. See *constants* for the possible return values.

Availability: ESP32, WiPy.

machine.**wake_pins**()

Returns the GPIO pin numbers of those pins which caused wakeup from deep sleep as a tuple of integers.

Availability: ESP32.

Miscellaneous functions

machine.**unique_id**()

Returns a byte string with a unique identifier of a board/SoC. It will vary from a board/SoC instance to another, if underlying hardware allows. Length varies by hardware (so use substring of a full value if you expect a short ID). In some MicroPython ports, ID corresponds to the network MAC address.

machine.**time_pulse_us**(*pin*, *pulse_level*, *timeout_us=1000000*, /)

Time a pulse on the given *pin*, and return the duration of the pulse in microseconds. The *pulse_level* argument should be 0 to time a low pulse or 1 to time a high pulse.

If the current input value of the pin is different to *pulse_level*, the function first (*) waits until the pin input becomes equal to *pulse_level*, then (**) times the duration that the pin is equal to *pulse_level*. If the pin is already equal to *pulse_level* then timing starts straight away.

The function will return -2 if there was timeout waiting for condition marked (*) above, and -1 if there was timeout during the main measurement, marked (**) above. The timeout is the same for both cases and given by *timeout_us* (which is in microseconds).

machine.**bitstream**(*pin*, *encoding*, *timing*, *data*, /)

Transmits *data* by bit-banging the specified *pin*. The *encoding* argument specifies how the bits are encoded, and *timing* is an encoding-specific timing specification.

The supported encodings are:

- `0` for "high low" pulse duration modulation. This will transmit 0 and 1 bits as timed pulses, starting with the most significant bit. The *timing* must be a four-tuple of nanoseconds in the format `(high_time_0, low_time_0, high_time_1, low_time_1)`. For example, `(400, 850, 800, 450)` is the timing specification for WS2812 RGB LEDs at 800kHz.

The accuracy of the timing varies between ports. On Cortex M0 at 48MHz, it is at best +/- 120ns, however on faster MCUs (ESP8266, ESP32, STM32, Pyboard), it will be closer to +/-30ns.

Note: For controlling WS2812 / NeoPixel strips, see the *neopixel* module for a higher-level API.

---

machine.**rng**()

Return a 24-bit software generated random number.

Availability: WiPy.

Constants

machine.**IDLE**

machine.**SLEEP**

machine.**DEEPSLEEP**

IRQ wake values.

machine.**PWRON_RESET**

machine.**HARD_RESET**

machine.**WDT_RESET**

machine.**DEEPSLEEP_RESET**

machine.**SOFT_RESET**

Reset causes.

machine.**WLAN_WAKE**

machine.**PIN_WAKE**

machine.**RTC_WAKE**

Wake-up reasons.

Classes

class Pin – control I/O pins

A pin object is used to control I/O pins (also known as GPIO - general-purpose input/output). Pin objects are commonly associated with a physical pin that can drive an output voltage and read input voltages. The pin class has methods to set the mode of the pin (IN, OUT, etc) and methods to get and set the digital logic level. For analog control of a pin, see the *ADC* class.

A pin object is constructed by using an identifier which unambiguously specifies a certain I/O pin. The allowed forms of the identifier and the physical pin that the identifier maps to are port-specific. Possibilities for the identifier are an integer, a string or a tuple with port and pin number.

Usage Model:

```
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())

# reconfigure pin #0 in input mode with a pull down resistor
p0.init(p0.IN, p0.PULL_DOWN)

# configure an irq callback
p0.irq(lambda p:print(p))
```

Constructors

class machine.**Pin**(*id*, *mode=-1*, *pull=-1*, ***, *value=None*, *drive=0*, *alt=-1*)

Access the pin peripheral (GPIO pin) associated with the given `id`. If additional arguments are given in the constructor then they are used to initialise the pin. Any settings that are not specified will remain in their previous state.

The arguments are:

- `id` is mandatory and can be an arbitrary object. Among possible value types are: int (an internal Pin identifier), str (a Pin name), and tuple (pair of [port, pin]).

- `mode` specifies the pin mode, which can be one of:
  - `Pin.IN` - Pin is configured for input. If viewed as an output the pin is in high-impedance state.
  - `Pin.OUT` - Pin is configured for (normal) output.
  - `Pin.OPEN_DRAIN` - Pin is configured for open-drain output. Open-drain output works in the following way: if the output value is set to 0 the pin is active at a low level; if the output value is 1 the pin is in a high-impedance state. Not all ports implement this mode, or some might only on certain pins.
  - `Pin.ALT` - Pin is configured to perform an alternative function, which is port specific. For a pin configured in such a way any other Pin methods (except *`Pin.init()`*) are not applicable (calling them will lead to undefined, or a hardware-specific, result). Not all ports implement this mode.
  - `Pin.ALT_OPEN_DRAIN` - The Same as `Pin.ALT`, but the pin is configured as open-drain. Not all ports implement this mode.
  - `Pin.ANALOG` - Pin is configured for analog input, see the *`ADC`* class.
- `pull` specifies if the pin has a (weak) pull resistor attached, and can be one of:
  - `None` - No pull up or down resistor.
  - `Pin.PULL_UP` - Pull up resistor enabled.
  - `Pin.PULL_DOWN` - Pull down resistor enabled.
- `value` is valid only for Pin.OUT and Pin.OPEN_DRAIN modes and specifies initial output pin value if given, otherwise the state of the pin peripheral remains unchanged.
- `drive` specifies the output power of the pin and can be one of: `Pin.DRIVE_0`, `Pin.DRIVE_1`, etc., increasing in drive strength. The actual current driving capabilities are port dependent. Not all ports implement this argument.
- `alt` specifies an alternate function for the pin and the values it can take are port dependent. This argument is valid only for `Pin.ALT` and `Pin.ALT_OPEN_DRAIN` modes. It may be used when a pin supports more than one alternate function. If only one pin alternate function is supported the this argument is not required. Not all ports implement this argument.

As specified above, the Pin class allows to set an alternate function for a particular pin, but it does not specify any further operations on such a pin. Pins configured in alternate-function mode are usually not used as GPIO but are instead driven by other hardware peripherals. The only operation supported on such a pin is re-initialising, by calling the constructor or *`Pin.init()`* method. If a pin that is configured in alternate-function mode is re-initialised with `Pin.IN`, `Pin.OUT`, or `Pin.OPEN_DRAIN`, the alternate function will be removed from the pin.

Methods

`Pin.`**`init`**(*mode=-1*, *pull=-1*, ***, *value=None*, *drive=0*, *alt=-1*)

Re-initialise the pin using the given parameters. Only those arguments that are specified will be set. The rest of the pin peripheral state will remain unchanged. See the constructor documentation for details of the arguments.

Returns `None`.

`Pin.`**`value`**([*x*])

This method allows to set and get the value of the pin, depending on whether the argument `x` is supplied or not.

If the argument is omitted then this method gets the digital logic level of the pin, returning 0 or 1 corresponding to low and high voltage signals respectively. The behaviour of this method depends on the mode of the pin:

- `Pin.IN` - The method returns the actual input value currently present on the pin.
- `Pin.OUT` - The behaviour and return value of the method is undefined.

- `Pin.OPEN_DRAIN` - If the pin is in state '0' then the behaviour and return value of the method is undefined. Otherwise, if the pin is in state '1', the method returns the actual input value currently present on the pin.

If the argument is supplied then this method sets the digital logic level of the pin. The argument `x` can be anything that converts to a boolean. If it converts to `True`, the pin is set to state '1', otherwise it is set to state '0'. The behaviour of this method depends on the mode of the pin:

- `Pin.IN` - The value is stored in the output buffer for the pin. The pin state does not change, it remains in the high-impedance state. The stored value will become active on the pin as soon as it is changed to `Pin.OUT` or `Pin.OPEN_DRAIN` mode.
- `Pin.OUT` - The output buffer is set to the given value immediately.
- `Pin.OPEN_DRAIN` - If the value is '0' the pin is set to a low voltage state. Otherwise the pin is set to high-impedance state.

When setting the value this method returns `None`.

`Pin.__call__`([*x*])

Pin objects are callable. The call method provides a (fast) shortcut to set and get the value of the pin. It is equivalent to Pin.value([x]). See *Pin.value()* for more details.

`Pin.on`()

Set pin to "1" output level.

`Pin.off`()

Set pin to "0" output level.

`Pin.irq`(*handler=None*, *trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING*, ***, *priority=1*, *wake=None*, *hard=False*)

Configure an interrupt handler to be called when the trigger source of the pin is active. If the pin mode is `Pin.IN` then the trigger source is the external value on the pin. If the pin mode is `Pin.OUT` then the trigger source is the output buffer of the pin. Otherwise, if the pin mode is `Pin.OPEN_DRAIN` then the trigger source is the output buffer for state '0' and the external pin value for state '1'.

The arguments are:

- `handler` is an optional function to be called when the interrupt triggers. The handler must take exactly one argument which is the `Pin` instance.
- `trigger` configures the event which can generate an interrupt. Possible values are:
  - `Pin.IRQ_FALLING` interrupt on falling edge.
  - `Pin.IRQ_RISING` interrupt on rising edge.
  - `Pin.IRQ_LOW_LEVEL` interrupt on low level.
  - `Pin.IRQ_HIGH_LEVEL` interrupt on high level.

  These values can be OR'ed together to trigger on multiple events.
- `priority` sets the priority level of the interrupt. The values it can take are port-specific, but higher values always represent higher priorities.
- `wake` selects the power mode in which this interrupt can wake up the system. It can be `machine.IDLE`, `machine.SLEEP` or `machine.DEEPSLEEP`. These values can also be OR'ed together to make a pin generate interrupts in more than one power mode.
- `hard` if true a hardware interrupt is used. This reduces the delay between the pin change and the handler being called. Hard interrupt handlers may not allocate memory; see *Writing interrupt handlers*. Not all ports support this argument.

This method returns a callback object.

The following methods are not part of the core Pin API and only implemented on certain ports.

`Pin.`**`low`**`()`

Set pin to "0" output level.

Availability: mimxrt, nrf, psoc-edge, renesas-ra, rp2, samd, stm32 ports.

`Pin.`**`high`**`()`

Set pin to "1" output level.

Availability: mimxrt, nrf, psoc-edge, renesas-ra, rp2, samd, stm32 ports.

`Pin.`**`mode`**`(`[*mode*]`)`

Get or set the pin mode. See the constructor documentation for details of the `mode` argument.

Availability: cc3200, psoc-edge, stm32 ports.

`Pin.`**`pull`**`(`[*pull*]`)`

Get or set the pin pull state. See the constructor documentation for details of the `pull` argument.

Availability: cc3200, psoc-edge, stm32 ports.

`Pin.`**`drive`**`(`[*drive*]`)`

Get or set the pin drive strength. See the constructor documentation for details of the `drive` argument.

Availability: cc3200, psoc-edge ports.

`Pin.`**`toggle`**`()`

Toggle output pin from "0" to "1" or vice-versa.

Availability: cc3200, esp32, esp8266, mimxrt, psoc-edge, rp2, samd ports.

Attributes

`Pin.`**`board`**

Contains pins named after the board's silkscreen or schematic labels. For example `Pin.board.X1` or `Pin.board.LED`.

Availability: alif, esp32, mimxrt, nrf, renesas-ra, rp2, samd, stm32 ports.

`Pin.`**`cpu`**

Contains the MCU pin names as given in the datasheet. For example `Pin.cpu.A0` or `Pin.cpu.GPIO0`.

Availability: alif, mimxrt, nrf, renesas-ra, rp2, samd, stm32 ports.

Multiple board pins can refer to the same CPU pin. Not all ports provide both attributes, and the available names depend on the board definition. On the esp32 port only `Pin.board` is provided, and only for boards whose definition includes named pins (e.g. `UM_TINYS3`, `M5STACK_NANOC6`). Generic ESP32 boards do not define any board pin names.

When constructing a `Pin` from a string name, the board pins are searched first, then the cpu pins:

```
from machine import Pin

# On a Pyboard v1.0, these all refer to the same physical pin:
p = Pin(Pin.board.X1, Pin.OUT)
p = Pin(Pin.cpu.A0, Pin.OUT)
p = Pin("X1", Pin.OUT)          # searches Pin.board, then Pin.cpu

# On a Raspberry Pi Pico W:
```

(continues on next page)

(continued from previous page)

```
p = Pin(Pin.board.LED, Pin.OUT)
p = Pin("LED", Pin.OUT)
```

Use `help(Pin.board)` or `help(Pin.cpu)` to list the pin names available on a particular board.

Constants

The following constants are used to configure the pin objects. Note that not all constants are available on all ports.

`Pin.`**`IN`**

`Pin.`**`OUT`**

`Pin.`**`OPEN_DRAIN`**

`Pin.`**`ALT`**

`Pin.`**`ALT_OPEN_DRAIN`**

`Pin.`**`ANALOG`**

Selects the pin mode.

`Pin.`**`PULL_UP`**

`Pin.`**`PULL_DOWN`**

Selects whether there is a pull up/down resistor. Use the value `None` for no pull.

Some ports have a different constants set that can be used to select hardware-specific behaviour:

- The esp8266 port does not have pull-down resistors on GPIO pins, hence `Pin.PULL_DOWN` is not supported.
- The mimxrt port has several extra constants to enable different pull modes: `Pin.PULL_UP_22K` enables a 22KΩ pull-up on the pin, `Pin.PULL_UP_47K` enables a 47KΩ pull-up on the pin, and `Pin.PULL_HOLD` that puts the pin into high-impedance mode. The `Pin.PULL_UP` and `Pin.PULL_DOWN` constants will use a 100KΩ internal resistor.

`Pin.`**`DRIVE_0`**

`Pin.`**`DRIVE_1`**

`Pin.`**`DRIVE_2`**

Selects the pin drive strength. A port may define additional drive constants with increasing number corresponding to increasing drive strength.

`Pin.`**`IRQ_FALLING`**

`Pin.`**`IRQ_RISING`**

`Pin.`**`IRQ_LOW_LEVEL`**

`Pin.`**`IRQ_HIGH_LEVEL`**

Selects the IRQ trigger type.

class Signal – control and sense external I/O devices

The Signal class is a simple extension of the *Pin* class. Unlike Pin, which can be only in "absolute" 0 and 1 states, a Signal can be in "asserted" (on) or "deasserted" (off) states, while being inverted (active-low) or not. In other words, it adds logical inversion support to Pin functionality. While this may seem a simple addition, it is exactly what is needed to support wide array of simple digital devices in a way portable across different boards, which is one of the major MicroPython goals. Regardless of whether different users have an active-high or active-low LED, a normally open or normally closed relay - you can develop a single, nicely looking application which works with each of them, and capture hardware configuration differences in few lines in the config file of your app.

Example:

```
from machine import Pin, Signal

# Suppose you have an active-high LED on pin 0
led1_pin = Pin(0, Pin.OUT)
# ... and active-low LED on pin 1
led2_pin = Pin(1, Pin.OUT)

# Now to light up both of them using Pin class, you'll need to set
# them to different values
led1_pin.value(1)
led2_pin.value(0)

# Signal class allows to abstract away active-high/active-low
# difference
led1 = Signal(led1_pin, invert=False)
led2 = Signal(led2_pin, invert=True)

# Now lighting up them looks the same
led1.value(1)
led2.value(1)

# Even better:
led1.on()
led2.on()
```

Following is the guide when Signal vs Pin should be used:

- Use Signal: If you want to control a simple on/off (including software PWM!) devices like LEDs, multi-segment indicators, relays, buzzers, or read simple binary sensors, like normally open or normally closed buttons, pulled high or low, Reed switches, moisture/flame detectors, etc. etc. Summing up, if you have a real physical device/sensor requiring GPIO access, you likely should use a Signal.
- Use Pin: If you implement a higher-level protocol or bus to communicate with more complex devices.

The split between Pin and Signal come from the use cases above and the architecture of MicroPython: Pin offers the lowest overhead, which may be important when bit-banging protocols. But Signal adds additional flexibility on top of Pin, at the cost of minor overhead (much smaller than if you implemented active-high vs active-low device differences in Python manually!). Also, Pin is a low-level object which needs to be implemented for each support board, while Signal is a high-level object which comes for free once Pin is implemented.

If in doubt, give the Signal a try! Once again, it is offered to save developers from the need to handle unexciting differences like active-low vs active-high signals, and allow other users to share and enjoy your application, instead of being frustrated by the fact that it doesn't work for them simply because their LEDs or relays are wired in a slightly different way.

Constructors

class machine.**Signal**(*pin_obj*, *invert=False*)

class machine.**Signal**(*pin_arguments...*, ***, *invert=False*)

Create a Signal object. There're two ways to create it:

- By wrapping existing Pin object - universal method which works for any board.
- By passing required Pin parameters directly to Signal constructor, skipping the need to create intermediate Pin object. Available on many, but not all boards.

The arguments are:

- `pin_obj` is existing Pin object.
- `pin_arguments` are the same arguments as can be passed to Pin constructor.
- `invert` - if True, the signal will be inverted (active low).

Note: The value of the pin can be set in the Pin constructor *and/or* the Signal constructor. If the Signal is also *inverted* then a value set in the *Pin* constructor will be in the opposite sense.

Example:

```
>>> c0 = Signal(Pin(0, Pin.OUT, value=0), invert=True)
>>> c0()
1
>>> c1 = Signal(1, Pin.OUT, value=0, invert=True)
>>> c1()
0
```

The first creates the pin and sets it's initial value and then Signal inverts the logic. Whereas, the second sets the pin to the inverted value.

This behavior is only different after construction and before a call to a 'set' method.

Example:

```
>>> c0.off()
>>> c0()
0
>>> c1.off()
>>> c1()
0
```

Methods

`Signal.`**`value`**`([x])`

This method allows to set and get the value of the signal, depending on whether the argument `x` is supplied or not.

If the argument is omitted then this method gets the signal level, 1 meaning signal is asserted (active) and 0 - signal inactive.

If the argument is supplied then this method sets the signal level. The argument `x` can be anything that converts to a boolean. If it converts to `True`, the signal is active, otherwise it is inactive.

Correspondence between signal being active and actual logic level on the underlying pin depends on whether signal is inverted (active-low) or not. For non-inverted signal, active status corresponds to logical 1, inactive - to logical 0. For inverted/active-low signal, active status corresponds to logical 0, while inactive - to logical 1.

`Signal.`**`__call__`**`([x])`

Signal objects are callable. The call method provides a (fast) shortcut to set and get the value of the pin. It is equivalent to Signal.value([x]). See *`Signal.value()`* for more details.

`Signal.`**`on`**`()`

Activate signal.

`Signal.`**`off`**`()`

Deactivate signal.

class ADC – analog to digital conversion

The ADC class provides an interface to analog-to-digital converters, and represents a single endpoint that can sample a continuous voltage and convert it to a discretised value.

For extra control over ADC sampling see *machine.ADCBlock*.

Example usage:

```
from machine import ADC

adc = ADC(pin)         # create an ADC object acting on a pin
val = adc.read_u16()   # read a raw analog value in the range 0-65535
val = adc.read_uv()    # read an analog value in microvolts
```

Constructors

class machine.**ADC**(*id*, *, *sample_ns*, *atten*)

Access the ADC associated with a source identified by *id*. This *id* may be an integer (usually specifying a channel number), a *Pin* object, or other value supported by the underlying machine.

If additional keyword-arguments are given then they will configure various aspects of the ADC. If not given, these settings will take previous or default values. The settings are:

- *sample_ns* is the sampling time in nanoseconds.
- *atten* specifies the input attenuation.

Methods

ADC.**init**(*, *sample_ns*, *atten*)

Apply the given settings to the ADC. Only those arguments that are specified will be changed. See the ADC constructor above for what the arguments are.

ADC.**block**()

Return the *ADCBlock* instance associated with this ADC object.

This method only exists if the port supports the *ADCBlock* class.

ADC.**read_u16**()

Take an analog reading and return an integer in the range 0-65535. The return value represents the raw reading taken by the ADC, scaled such that the minimum value is 0 and the maximum value is 65535.

ADC.**read_uv**()

Take an analog reading and return an integer value with units of microvolts. It is up to the particular port whether or not this value is calibrated, and how calibration is done.

class ADCBlock – control ADC peripherals

The ADCBlock class provides access to an ADC peripheral which has a number of channels that can be used to sample analog values. It allows finer control over configuration of *machine.ADC* objects, which do the actual sampling.

This class is not always available.

Example usage:

```
from machine import ADCBlock

block = ADCBlock(id, bits=12)  # create an ADCBlock with 12-bit resolution
adc = block.connect(4, pin)    # connect channel 4 to the given pin
val = adc.read_uv()            # read an analog value
```

Constructors

class machine.**ADCBlock**(*id*, *, *bits*)

Access the ADC peripheral identified by *id*, which may be an integer or string.

The *bits* argument, if given, sets the resolution in bits of the conversion process. If not specified then the previous or default resolution is used.

Methods

ADCBlock.**init**(*, *bits*)

Configure the ADC peripheral. *bits* will set the resolution of the conversion process.

ADCBlock.**connect**(*channel*, *, ...)

ADCBlock.**connect**(*source*, *, ...)

ADCBlock.**connect**(*channel*, *source*, *, ...)

Connect up a channel on the ADC peripheral so it is ready for sampling, and return an *ADC* object that represents that connection.

The *channel* argument must be an integer, and *source* must be an object (for example a *Pin*) which can be connected up for sampling.

If only *channel* is given then it is configured for sampling.

If only *source* is given then that object is connected to a default channel ready for sampling.

If both *channel* and *source* are given then they are connected together and made ready for sampling.

Any additional keyword arguments are used to configure the returned ADC object, via its `init` method.

class DAC – digital to analog conversion

The DAC is used to output an analog voltage based on a digital value.

The output voltage will be between 0 and 3.3V.

DAC is currently supported on ESP32[1], SAMD and Renesas RA.

Note: The STM32 port has similar functionality to `machine.DAC`. See *pyb.DAC* for details.

Example usage (ESP32):

```
from machine import DAC

dac = DAC(pin)     # create a DAC object acting on a pin
dac.write(128)     # write a value to the DAC
dac.write(255)     # output maximum value, 3.3V
```

Constructors

class machine.**DAC**(*id*)

Construct a new DAC object.

`id` is a pin object (ESP32 and Renesas RA) or an index to a DAC resource (SAMD).

Note: On the ESP32, DAC functionality is available on pins 25 and 26. On the ESP32-S2, pins 17 and 18. See *ESP32 Quickref* for more details.

Note: SAMD21 has one DAC resource, SAMD51 has two. See *SAMD Quickref* for more details.

Methods

DAC.**write**(*value*)

Output an analog voltage to the pin connected to the DAC.

`value` is a representation of the desired output; a linear interpolation of 0-3.3V, though the range differs depending on the port and micro, see below:

Port/micro	Bits	Range
ESP32	8	0-255
SAMD21	10	0-1023
SAMD51	12	0-4095
Renesas RA	12	0-4095

[1] The original ESP32 and ESP32-S2 *only*, since DAC hardware is not present on other microcontrollers in the family.

class PWM – pulse width modulation

This class provides pulse width modulation output.

Example usage:

```
from machine import PWM

pwm = PWM(pin, freq=50, duty_u16=8192)  # create a PWM object on a pin
                                        # and set freq 50 Hz and duty 12.5%
pwm.duty_u16(32768)                     # set duty to 50%

# reinitialise with a period of 200us, duty of 5us
pwm.init(freq=5000, duty_ns=5000)

pwm.duty_ns(3000)                       # set pulse width to 3us

pwm.deinit()
```

Constructors

class machine.**PWM**(*dest*, ***, *freq*, *duty_u16*, *duty_ns*, *invert=False*)

Construct and return a new PWM object using the following parameters:

- *dest* is the entity on which the PWM is output, which is usually a *machine.Pin* object, but a port may allow other values, like integers.
- *freq* should be an integer which sets the frequency in Hz for the PWM cycle.
- *duty_u16* sets the duty cycle as a ratio `duty_u16 / 65535`.
- *duty_ns* sets the pulse width in nanoseconds.
- *invert* inverts the respective output if the value is True

Setting *freq* may affect other PWM objects if the objects share the same underlying PWM generator (this is hardware specific). Only one of *duty_u16* and *duty_ns* should be specified at a time. *invert* is available only on the alif, esp32, mimxrt, nrf, rp2, samd, stm32 and zephyr ports.

Methods

PWM.**init**(***, *freq*, *duty_u16*, *duty_ns*)

Modify settings for the PWM object. See the above constructor for details about the parameters.

PWM.**deinit**()

Disable the PWM output.

PWM.**freq**([*value*])

Get or set the current frequency of the PWM output.

With no arguments the frequency in Hz is returned.

With a single *value* argument the frequency is set to that value in Hz. The method may raise a `ValueError` if the frequency is outside the valid range.

PWM.**duty_u16**([*value*])

Get or set the current duty cycle of the PWM output, as an unsigned 16-bit value in the range 0 to 65535 inclusive.

With no arguments the duty cycle is returned.

With a single *value* argument the duty cycle is set to that value, measured as the ratio `value / 65535`.

PWM.**duty_ns**([*value*])

Get or set the current pulse width of the PWM output, as a value in nanoseconds.

With no arguments the pulse width in nanoseconds is returned.

With a single *value* argument the pulse width is set to that value.

Specific PWM class implementations

On the alif port there are 11 independent PWM blocks with independent frequencies, and they have 2 outputs each. The underlying counter is 32-bits wide for all 11 PWM blocks.

On the rp2 port there are 8 independent PWM blocks on RP2040 and 12 on RP2350, each with independent frequencies, and each with 2 outputs. The underlying counter is 16-bits wide for all PWM blocks.

On the stm32 port the number of independent PWM blocks depends on the MCU and can range between 4 and 19. TIM2 and TIM5 blocks (also TIM3 and TIM4 blocks on STM32U5 and STM32N6) are 32-bits wide, and the others are 16-bits wide. All MCUs supported by MicroPython have at least one 32-bit block available, and most have two. MCUs will have pins PA0 through PA3 assigned to a 32-bit PWM block (except STM32N6 which has a 16-bit PWM block on PA3). PWM blocks have up to 4 outputs each.

The following concrete class(es) implement enhancements to the PWM class.

pyb.Timer for PyBoard

Limitations of PWM

- Not all frequencies can be generated with absolute accuracy due to the discrete nature of the computing hardware. Typically the PWM frequency is obtained by dividing some integer base frequency by an integer divider. For example, if the base frequency is 80MHz and the required PWM frequency is 300kHz the divider must be a non-integer number 80000000 / 300000 = 266.67. After rounding the divider is set to 267 and the PWM frequency will be 80000000 / 267 = 299625.5 Hz, not 300kHz. If the divider is set to 266 then the PWM frequency will be 80000000 / 266 = 300751.9 Hz, but again not 300kHz.

  Some ports like the RP2040 one use a fractional divider, which allow a finer granularity of the frequency at higher frequencies by switching the PWM pulse duration between two adjacent values, such that the resulting average frequency is more close to the intended one, at the cost of spectral purity.

- The duty cycle has the same discrete nature and its absolute accuracy is not achievable. On most hardware platforms the duty will be applied at the next frequency period. Therefore, you should wait more than "1/frequency" before measuring the duty.

- The frequency and the duty cycle resolution are usually interdependent. The higher the PWM frequency the lower the duty resolution which is available, and vice versa. For example, a 300kHz PWM frequency can have a duty cycle resolution of 8 bit, not 16-bit as may be expected. In this case, the lowest 8 bits of *duty_u16* are insignificant. So:

```
pwm=PWM(Pin(13), freq=300_000, duty_u16=65536//2)
```

  and:

```
pwm=PWM(Pin(13), freq=300_000, duty_u16=65536//2 + 255)
```

will generate PWM with the same 50% duty cycle.

class UART – duplex serial communication bus

UART implements the standard UART/USART duplex serial communications protocol. At the physical level it consists of 2 lines: RX and TX. The unit of communication is a character (not to be confused with a string character) which can be 8 or 9 bits wide.

UART objects can be created and initialised using:

```
from machine import UART

uart = UART(1, 9600)                         # init with given baudrate
uart.init(9600, bits=8, parity=None, stop=1) # init with given parameters
```

Supported parameters differ on a board:

Pyboard: Bits can be 7, 8 or 9. Stop can be 1 or 2. With *parity=None*, only 8 and 9 bits are supported. With parity enabled, only 7 and 8 bits are supported.

WiPy/CC3200: Bits can be 5, 6, 7, 8. Stop can be 1 or 2.

A UART object acts like a *stream* object and reading and writing is done using the standard stream methods:

```
uart.read(10)       # read 10 characters, returns a bytes object
uart.read()         # read all available characters
uart.readline()     # read a line
uart.readinto(buf)  # read and store into the given buffer
uart.write('abc')   # write the 3 characters
```

Constructors

class machine.**UART**(*id*, ...)

Construct a UART object of the given id.

Methods

UART.**init**(*baudrate=9600*, *bits=8*, *parity=None*, *stop=1*, *, ...)

Initialise the UART bus with the given parameters:

- *baudrate* is the clock rate.
- *bits* is the number of bits per character, 7, 8 or 9.
- *parity* is the parity, `None`, 0 (even) or 1 (odd).
- *stop* is the number of stop bits, 1 or 2.

Additional keyword-only parameters that may be supported by a port are:

- *tx* specifies the TX pin to use.
- *rx* specifies the RX pin to use.
- *rts* specifies the RTS (output) pin to use for hardware receive flow control.

- *cts* specifies the CTS (input) pin to use for hardware transmit flow control.
- *txbuf* specifies the length in characters of the TX buffer.
- *rxbuf* specifies the length in characters of the RX buffer.
- *timeout* specifies the time to wait for the first character (in ms).
- *timeout_char* specifies the time to wait between characters (in ms).
- *invert* specifies which lines to invert.
  - `0` will not invert lines (idle state of both lines is logic high).
  - `UART.INV_TX` will invert TX line (idle state of TX line now logic low).
  - `UART.INV_RX` will invert RX line (idle state of RX line now logic low).
  - `UART.INV_TX | UART.INV_RX` will invert both lines (idle state at logic low).
- *flow* specifies which hardware flow control signals to use. The value is a bitmask.
  - `0` will ignore hardware flow control signals.
  - `UART.RTS` will enable receive flow control by using the RTS output pin to signal if the receive FIFO has sufficient space to accept more data.
  - `UART.CTS` will enable transmit flow control by pausing transmission when the CTS input pin signals that the receiver is running low on buffer space.
  - `UART.RTS | UART.CTS` will enable both, for full hardware flow control.

On the WiPy only the following keyword-only parameter is supported:

- *pins* is a 4 or 2 item list indicating the TX, RX, RTS and CTS pins (in that order). Any of the pins can be None if one wants the UART to operate with limited functionality. If the RTS pin is given the RX pin must be given as well. The same applies to CTS. When no pins are given, then the default set of TX and RX pins is taken, and hardware flow control will be disabled. If *pins* is `None`, no pin assignment will be made.

Note: It is possible to call `init()` multiple times on the same object in order to reconfigure UART on the fly. That allows using single UART peripheral to serve different devices attached to different GPIO pins. Only one device can be served at a time in that case. Also do not call `deinit()` as it will prevent calling `init()` again.

`UART.`**`deinit`**`()`

Turn off the UART bus.

Note: You will not be able to call `init()` on the object after `deinit()`. A new instance needs to be created in that case.

`UART.`**`any`**`()`

Returns an integer counting the number of characters that can be read without blocking. It will return 0 if there are no characters available and a positive number if there are characters. The method may return 1 even if there is more than one character available for reading.

For more sophisticated querying of available characters use select.poll:

```
poll = select.poll()
poll.register(uart, select.POLLIN)
poll.poll(timeout)
```

UART.**read**([*nbytes*])

Read characters. If `nbytes` is specified then read at most that many bytes, otherwise read as much data as possible. It may return sooner if a timeout is reached. The timeout is configurable in the constructor.

Return value: a bytes object containing the bytes read in. Returns `None` on timeout.

UART.**readinto**(*buf*[, *nbytes*])

Read bytes into the `buf`. If `nbytes` is specified then read at most that many bytes. Otherwise, read at most `len(buf)` bytes. It may return sooner if a timeout is reached. The timeout is configurable in the constructor.

Return value: number of bytes read and stored into `buf` or `None` on timeout.

UART.**readline**()

Read a line, ending in a newline character. It may return sooner if a timeout is reached. The timeout is configurable in the constructor.

Return value: the line read or `None` on timeout.

UART.**write**(*buf*)

Write the buffer of bytes to the bus.

Return value: number of bytes written or `None` on timeout.

UART.**sendbreak**()

Send a break condition on the bus. This drives the bus low for a duration longer than required for a normal transmission of a character.

UART.**flush**()

Waits until all data has been sent. In case of a timeout, an exception is raised. The timeout duration depends on the tx buffer size and the baud rate. Unless flow control is enabled, a timeout should not occur.

Note: For the esp8266 and nrf ports the call returns while the last byte is sent. If required, a one character wait time has to be added in the calling script.

Availability: rp2, esp32, esp8266, mimxrt, cc3200, stm32, nrf ports, psoc-edge, renesas-ra

UART.**txdone**()

Tells whether all data has been sent or no data transfer is happening. In this case, it returns `True`. If a data transmission is ongoing it returns `False`.

Note: For the esp8266 and nrf ports the call may return `True` even if the last byte of a transfer is still being sent. If required, a one character wait time has to be added in the calling script.

Availability: rp2, esp32, esp8266, mimxrt, cc3200, stm32, nrf ports, psoc-edge, renesas-ra

UART.**irq**(*handler=None*, *trigger=0*, *hard=False*)

Configure an interrupt handler to be called when a UART event occurs.

The arguments are:

- *handler* is an optional function to be called when the interrupt event triggers. The handler must take exactly one argument which is the `UART` instance.
- *trigger* configures the event(s) which can generate an interrupt. Possible values are a mask of one or more of the following:
  - `UART.IRQ_RXIDLE` interrupt after receiving at least one character and then the RX line goes idle.

  - `UART.IRQ_RX` interrupt after each received character.
  - `UART.IRQ_TXIDLE` interrupt after or while the last character(s) of a message are or have been sent.
  - `UART.IRQ_BREAK` interrupt when a break state is detected at RX

- *hard* if true a hardware interrupt is used. This reduces the delay between the pin change and the handler being called. Hard interrupt handlers may not allocate memory; see *Writing interrupt handlers*.

Returns an irq object.

Due to limitations of the hardware not all trigger events are available on all ports.

Table 1: Availability of triggers

Port / Trigger	IRQ_RXIDLE	IRQ_RX	IRQ_TXIDLE	IRQ_BREAK
CC3200		yes		
ESP32	yes	yes		yes
MIMXRT	yes		yes	
NRF		yes	yes	
PSOC-EDGE	yes		yes	yes
RENESAS-RA	yes	yes		
RP2	yes		yes	yes
SAMD	yes	yes	yes	
STM32	yes	yes		

Note:

- The ESP32 port does not support the option hard=True.
- The rp2 port's UART.IRQ_TXIDLE is only triggered when the message is longer than 5 characters and the trigger happens when still 5 characters are to be sent.
- The rp2 port's UART.IRQ_BREAK needs receiving valid characters for triggering again.
- The SAMD port's UART.IRQ_TXIDLE is triggered while the last character is sent.
- On STM32F4xx MCU's, using the trigger UART.IRQ_RXIDLE the handler will be called once after the first character and then after the end of the message, when the line is idle.

Availability: cc3200, esp32, mimxrt, nrf, psoc-edge, renesas-ra, rp2, samd, stm32.

Constants

`UART.RTS`

`UART.CTS`

Flow control options.

Availability: esp32, mimxrt, renesas-ra, rp2, stm32.

`UART.IRQ_RXIDLE`

`UART.IRQ_RX`

`UART.IRQ_TXIDLE`

UART.**IRQ_BREAK**

IRQ trigger sources.

Availability: renesas-ra, stm32, esp32, rp2040, mimxrt, samd, cc3200, psoc-edge.

class SPI – a Serial Peripheral Interface bus protocol (controller side)

SPI is a synchronous serial protocol that is driven by a controller. At the physical level, a bus consists of 3 lines: SCK, MOSI, MISO. Multiple devices can share the same bus. Each device should have a separate, 4th signal, CS (Chip Select), to select a particular device on a bus with which communication takes place. Management of a CS signal should happen in user code (via machine.Pin class).

Both hardware and software SPI implementations exist via the *machine.SPI* and *machine.SoftSPI* classes. Hardware SPI uses underlying hardware support of the system to perform the reads/writes and is usually efficient and fast but may have restrictions on which pins can be used. Software SPI is implemented by bit-banging and can be used on any pin but is not as efficient. These classes have the same methods available and differ primarily in the way they are constructed.

Example usage:

```
from machine import SPI, Pin

spi = SPI(0, baudrate=400000)           # Create SPI peripheral 0 at frequency of␣
↪400kHz.
                                        # Depending on the use case, extra parameters␣
↪may be required
                                        # to select the bus characteristics and/or␣
↪pins to use.
cs = Pin(4, mode=Pin.OUT, value=1)      # Create chip-select on pin 4.

try:
    cs(0)                               # Select peripheral.
    spi.write(b"12345678")              # Write 8 bytes, and don't care about␣
↪received data.
finally:
    cs(1)                               # Deselect peripheral.

try:
    cs(0)                               # Select peripheral.
    rxdata = spi.read(8, 0x42)          # Read 8 bytes while writing 0x42 for each␣
↪byte.
finally:
    cs(1)                               # Deselect peripheral.

rxdata = bytearray(8)
try:
    cs(0)                               # Select peripheral.
    spi.readinto(rxdata, 0x42)          # Read 8 bytes inplace while writing 0x42 for␣
↪each byte.
finally:
    cs(1)                               # Deselect peripheral.

txdata = b"12345678"
rxdata = bytearray(len(txdata))
try:
    cs(0)                               # Select peripheral.
    spi.write_readinto(txdata, rxdata)  # Simultaneously write and read bytes.
```

(continues on next page)

(continued from previous page)

```
finally:
    cs(1)                                    # Deselect peripheral.
```

Constructors

class machine.**SPI**(*id*, ...)

Construct an SPI object on the given bus, *id*. Values of *id* depend on a particular port and its hardware. Values 0, 1, etc. are commonly used to select hardware SPI block #0, #1, etc.

With no additional parameters, the SPI object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See `init` for parameters of initialisation.

class machine.**SoftSPI**(*baudrate=500000*, ***, *polarity=0*, *phase=0*, *bits=8*, *firstbit=MSB*, *sck=None*, *mosi=None*, *miso=None*)

Construct a new software SPI object. Additional parameters must be given, usually at least *sck*, *mosi* and *miso*, and these are used to initialise the bus. See *SPI.init* for a description of the parameters.

Methods

SPI.**init**(*baudrate=1000000*, ***, *polarity=0*, *phase=0*, *bits=8*, *firstbit=SPI.MSB*, *sck=None*, *mosi=None*, *miso=None*, *pins=(SCK, MOSI, MISO)*)

Initialise the SPI bus with the given parameters:

- `baudrate` is the SCK clock rate.
- `polarity` can be 0 or 1, and is the level the idle clock line sits at.
- `phase` can be 0 or 1 to sample data on the first or second clock edge respectively.
- `bits` is the width in bits of each transfer. Only 8 is guaranteed to be supported by all hardware.
- `firstbit` can be `SPI.MSB` or `SPI.LSB`.
- `sck`, `mosi`, `miso` are pins (machine.Pin) objects to use for bus signals. For most hardware SPI blocks (as selected by `id` parameter to the constructor), pins are fixed and cannot be changed. In some cases, hardware blocks allow 2-3 alternative pin sets for a hardware SPI block. Arbitrary pin assignments are possible only for a bitbanging SPI driver (`id` = -1).
- `pins` - WiPy port doesn't `sck`, `mosi`, `miso` arguments, and instead allows to specify them as a tuple of `pins` parameter.

In the case of hardware SPI the actual clock frequency may be lower than the requested baudrate. This is dependent on the platform hardware. The actual rate may be determined by printing the SPI object.

SPI.**deinit**()

Turn off the SPI bus.

SPI.**read**(*nbytes*, *write=0x00*)

Read a number of bytes specified by `nbytes` while continuously writing the single byte given by `write`. Returns a `bytes` object with the data that was read.

SPI.**readinto**(*buf*, *write=0x00*)

Read into the buffer specified by `buf` while continuously writing the single byte given by `write`. Returns `None`.

Note: on WiPy this function returns the number of bytes read.

SPI.**write**(*buf*)

Write the bytes contained in `buf`. Returns `None`.

Note: on WiPy this function returns the number of bytes written.

SPI.**write_readinto**(*write_buf*, *read_buf*)

Write the bytes from `write_buf` while reading into `read_buf`. The buffers can be the same or different, but both buffers must have the same length. Returns `None`.

Note: on WiPy this function returns the number of bytes written.

Constants

SPI.**CONTROLLER**

for initialising the SPI bus to controller; this is only used for the WiPy

SPI.**MSB**

SoftSPI.**MSB**

set the first bit to be the most significant bit

SPI.**LSB**

SoftSPI.**LSB**

set the first bit to be the least significant bit

class I2C – a two-wire serial protocol

I2C is a two-wire protocol for communicating between devices. At the physical level it consists of 2 wires: SCL and SDA, the clock and data lines respectively.

I2C objects are created attached to a specific bus. They can be initialised when created, or initialised later on.

Printing the I2C object gives you information about its configuration.

Both hardware and software I2C implementations exist via the *machine.I2C* and *machine.SoftI2C* classes. Hardware I2C uses underlying hardware support of the system to perform the reads/writes and is usually efficient and fast but may have restrictions on which pins can be used. Software I2C is implemented by bit-banging and can be used on any pin but is not as efficient. These classes have the same methods available and differ primarily in the way they are constructed.

Note: The I2C bus requires pull-up circuitry on both SDA and SCL for it's operation. Usually these are resistors in the range of 1 - 10 kOhm, connected from each SDA/SCL to Vcc. Without these, the behaviour is undefined and may range from blocking, unexpected watchdog reset to just wrong values. Often, this pull-up circuitry is built-in already to the MCU board or sensor breakout boards, but there is no rule for that. So please check in case of trouble. See also this excellent learning guide by Adafruit about I2C wiring.

Example usage:

```
from machine import I2C

i2c = I2C(freq=400000)          # create I2C peripheral at frequency of 400kHz
                                # depending on the port, extra parameters may be␣
↪required
                                # to select the peripheral and/or pins to use
```

(continues on next page)

(continued from previous page)

```
i2c.scan()                      # scan for peripherals, returning a list of 7-bit␣
↪addresses

i2c.writeto(42, b'123')         # write 3 bytes to peripheral with 7-bit address 42
i2c.readfrom(42, 4)             # read 4 bytes from peripheral with 7-bit address 42

i2c.readfrom_mem(42, 8, 3)      # read 3 bytes from memory of peripheral 42,
                                #   starting at memory-address 8 in the peripheral
i2c.writeto_mem(42, 2, b'\x10') # write 1 byte to memory of peripheral 42
                                #   starting at address 2 in the peripheral
```

Constructors

class machine.**I2C**(*id*, *, *scl*, *sda*, *freq=400000*, *timeout=50000*)

Construct and return a new I2C object using the following parameters:

- *id* identifies a particular I2C peripheral. Allowed values for depend on the particular port/board
- *scl* should be a pin object specifying the pin to use for SCL.
- *sda* should be a pin object specifying the pin to use for SDA.
- *freq* should be an integer which sets the maximum frequency for SCL.
- *timeout* is the maximum time in microseconds to allow for I2C transactions. This parameter is not allowed on some ports.

Note that some ports/boards will have default values of *scl* and *sda* that can be changed in this constructor. Others will have fixed values of *scl* and *sda* that cannot be changed.

class machine.**SoftI2C**(*scl*, *sda*, *, *freq=400000*, *timeout=50000*)

Construct a new software I2C object. The parameters are:

- *scl* should be a pin object specifying the pin to use for SCL.
- *sda* should be a pin object specifying the pin to use for SDA.
- *freq* should be an integer which sets the maximum frequency for SCL.
- *timeout* is the maximum time in microseconds to wait for clock stretching (SCL held low by another device on the bus), after which an `OSError(ETIMEDOUT)` exception is raised.

General Methods

I2C.**init**(*scl*, *sda*, *, *freq=400000*)

Initialise the I2C bus with the given arguments:

- *scl* is a pin object for the SCL line
- *sda* is a pin object for the SDA line
- *freq* is the SCL clock rate

In the case of hardware I2C the actual clock frequency may be lower than the requested frequency. This is dependent on the platform hardware. The actual rate may be determined by printing the I2C object.

I2C.**deinit**()

Turn off the I2C bus.

I2C.**scan**()

Scan all I2C addresses between 0x08 and 0x77 inclusive and return a list of those that respond. A device responds if it pulls the SDA line low after its address (including a write bit) is sent on the bus.

Primitive I2C operations

The following methods implement the primitive I2C controller bus operations and can be combined to make any I2C transaction. They are provided if you need more control over the bus, otherwise the standard methods (see below) can be used.

These methods are only available on the *machine.SoftI2C* class.

I2C.**start**()

Generate a START condition on the bus (SDA transitions to low while SCL is high).

I2C.**stop**()

Generate a STOP condition on the bus (SDA transitions to high while SCL is high).

I2C.**readinto**(*buf*, *nack=True*, */*)

Reads bytes from the bus and stores them into *buf*. The number of bytes read is the length of *buf*. An ACK will be sent on the bus after receiving all but the last byte. After the last byte is received, if *nack* is true then a NACK will be sent, otherwise an ACK will be sent (and in this case the peripheral assumes more bytes are going to be read in a later call).

I2C.**write**(*buf*)

Write the bytes from *buf* to the bus. Checks that an ACK is received after each byte and stops transmitting the remaining bytes if a NACK is received. The function returns the number of ACKs that were received.

Standard bus operations

The following methods implement the standard I2C controller read and write operations that target a given peripheral device.

I2C.**readfrom**(*addr*, *nbytes*, *stop=True*, */*)

Read *nbytes* from the peripheral specified by *addr*. If *stop* is true then a STOP condition is generated at the end of the transfer. Returns a *bytes* object with the data read.

I2C.**readfrom_into**(*addr*, *buf*, *stop=True*, */*)

Read into *buf* from the peripheral specified by *addr*. The number of bytes read will be the length of *buf*. If *stop* is true then a STOP condition is generated at the end of the transfer.

The method returns `None`.

I2C.**writeto**(*addr*, *buf*, *stop=True*, */*)

Write the bytes from *buf* to the peripheral specified by *addr*. If a NACK is received following the write of a byte from *buf* then the remaining bytes are not sent. If *stop* is true then a STOP condition is generated at the end of the transfer, even if a NACK is received. The function returns the number of ACKs that were received.

I2C.**writevto**(*addr*, *vector*, *stop=True*, */*)

Write the bytes contained in *vector* to the peripheral specified by *addr*. *vector* should be a tuple or list of objects with the buffer protocol. The *addr* is sent once and then the bytes from each object in *vector* are written out sequentially. The objects in *vector* may be zero bytes in length in which case they don't contribute to the output.

If a NACK is received following the write of a byte from one of the objects in *vector* then the remaining bytes, and any remaining objects, are not sent. If *stop* is true then a STOP condition is generated at the end of the transfer, even if a NACK is received. The function returns the number of ACKs that were received.

Memory operations

Some I2C devices act as a memory device (or set of registers) that can be read from and written to. In this case there are two addresses associated with an I2C transaction: the peripheral address and the memory address. The following methods are convenience functions to communicate with such devices.

I2C.**readfrom_mem**(*addr*, *memaddr*, *nbytes*, ***, *addrsize=8*)

Read *nbytes* from the peripheral specified by *addr* starting from the memory address specified by *memaddr*. The argument *addrsize* specifies the address size in bits. Returns a *bytes* object with the data read.

I2C.**readfrom_mem_into**(*addr*, *memaddr*, *buf*, ***, *addrsize=8*)

Read into *buf* from the peripheral specified by *addr* starting from the memory address specified by *memaddr*. The number of bytes read is the length of *buf*. The argument *addrsize* specifies the address size in bits (on ESP8266 this argument is not recognised and the address size is always 8 bits).

The method returns `None`.

I2C.**writeto_mem**(*addr*, *memaddr*, *buf*, ***, *addrsize=8*)

Write *buf* to the peripheral specified by *addr* starting from the memory address specified by *memaddr*. The argument *addrsize* specifies the address size in bits (on ESP8266 this argument is not recognised and the address size is always 8 bits).

The method returns `None`.

class I2CTarget – an I2C target device

An I2C target is a device which connects to an I2C bus and is controlled by an I2C controller. I2C targets can take many forms. The *machine.I2CTarget* class implements an I2C target that can be configured as a memory/register device, or as an arbitrary I2C device by using callbacks (if supported by the port).

Example usage for the case of a memory device:

```
from machine import I2CTarget

# Create the backing memory for the I2C target.
mem = bytearray(8)

# Create an I2C target.  Depending on the port, extra parameters
# may be required to select the peripheral and/or pins to use.
i2c = I2CTarget(addr=67, mem=mem)

# At this point an I2C controller can read and write `mem`.
...

# Deinitialise the I2C target.
i2c.deinit()
```

Note that some ports require an `id`, and maybe `scl` and `sda` pins, to be passed to the *I2CTarget* constructor, to select the hardware I2C instance and pins that it connects to.

When configured as a memory device, it's also possible to register to receive events. For example to be notified when the memory is read/written:

```
from machine import I2CTarget

# Define an IRQ handler, for I2C events.
```

(continues on next page)

(continued from previous page)

```
def irq_handler(i2c_target):
    flags = i2c_target.irq().flags()
    if flags & I2CTarget.IRQ_END_READ:
        print("controller read target at addr", i2c_target.memaddr)
    if flags & I2CTarget.IRQ_END_WRITE:
        print("controller wrote target at addr", i2c_target.memaddr)

# Create the I2C target and register to receive default events.
mem = bytearray(8)
i2c = I2CTarget(addr=67, mem=mem)
i2c.irq(irq_handler)
```

More complicated I2C devices can be implemented using the full set of events. For example, to see the raw events as they are triggered:

```
from machine import I2CTarget

# Define an IRQ handler that prints the event id and responds to reads/writes.
def irq_handler(i2c_target, buf=bytearray(1)):
    flags = i2c_target.irq().flags()
    print(flags)
    if flags & I2CTarget.IRQ_READ_REQ:
        i2c_target.write(buf)
    if flags & I2CTarget.IRQ_WRITE_REQ:
        i2c_target.readinto(buf)

# Create the I2C target and register to receive all events.
i2c = I2CTarget(addr=67)
all_triggers = (
    I2CTarget.IRQ_ADDR_MATCH_READ
    | I2CTarget.IRQ_ADDR_MATCH_WRITE
    | I2CTarget.IRQ_READ_REQ
    | I2CTarget.IRQ_WRITE_REQ
    | I2CTarget.IRQ_END_READ
    | I2CTarget.IRQ_END_WRITE
)
i2c.irq(irq_handler, trigger=all_triggers, hard=True)
```

Availability: **Alif, ESP32, MIMXRT, RP2, SAMD, STM32, Zephyr**

Constructors

class machine.**I2CTarget**(*id*, *addr*, *, *addrsize=7*, *mem=None*, *mem_addrsize=8*, *scl=None*, *sda=None*)

Construct and return a new I2CTarget object using the following parameters:

- *id* identifies a particular I2C peripheral. Allowed values depend on the particular port/board. Some ports have a default in which case this parameter can be omitted.
- *addr* is the I2C address of the target.
- *addrsize* is the number of bits in the I2C target address. Valid values are 7 and 10.
- *mem* is an object with the buffer protocol that is writable. If not specified then there is no backing memory and data must be read/written using the *I2CTarget.readinto()* and *I2CTarget.write()* methods.
- *mem_addrsize* is the number of bits in the memory address. Valid values are 0, 8, 16, 24 and 32.

- *scl* is a pin object specifying the pin to use for SCL.
- *sda* is a pin object specifying the pin to use for SDA.

Note that some ports/boards will have default values of *scl* and *sda* that can be changed in this constructor. Others will have fixed values of *scl* and *sda* that cannot be changed.

General Methods

I2CTarget.**deinit**()

Deinitialise the I2C target. After this method is called the hardware will no longer respond to requests on the I2C bus, and no other methods can be called.

I2CTarget.**readinto**(*buf*)

Read into the given buffer any pending bytes written by the I2C controller. Returns the number of bytes read.

I2CTarget.**write**(*buf*)

Write out the bytes from the given buffer, to be passed to the I2C controller after it sends a read request. Returns the number of bytes written. Most ports only accept one byte at a time to this method.

I2CTarget.**irq**(*handler=None*, *trigger=IRQ_END_READ | IRQ_END_WRITE*, *hard=False*)

Configure an IRQ *handler* to be called when an event occurs. The possible events are given by the following constants, which can be or'd together and passed to the *trigger* argument:

- IRQ_ADDR_MATCH_READ indicates that the target was addressed by a controller for a read transaction.
- IRQ_ADDR_MATCH_WRITE indicates that the target was addressed by a controller for a write transaction.
- IRQ_READ_REQ indicates that the controller is requesting data, and this request must be satisfied by calling *I2CTarget.write* with the data to be passed back to the controller.
- IRQ_WRITE_REQ indicates that the controller has written data, and the data must be read by calling *I2CTarget.readinto*.
- IRQ_END_READ indicates that the controller has finished a read transaction.
- IRQ_END_WRITE indicates that the controller has finished a write transaction.

Not all triggers are available on all ports. If a port has the constant then that event is available.

Note the following restrictions:

- IRQ_ADDR_MATCH_READ, IRQ_ADDR_MATCH_WRITE, IRQ_READ_REQ and IRQ_WRITE_REQ must be handled by a hard IRQ callback (with the *hard* argument set to True). This is because these events have very strict timing requirements and must usually be satisfied synchronously with the hardware event.
- IRQ_END_READ and IRQ_END_WRITE may be handled by either a soft or hard IRQ callback (although note that all events must be registered with the same handler, so if any events need a hard callback then all events must be hard).
- If a memory buffer has been supplied in the constructor then IRQ_END_WRITE is not emitted for the transaction that writes the memory address. This is to allow IRQ_END_READ and IRQ_END_WRITE to function correctly as soft IRQ callbacks, where the IRQ handler may be called quite some time after the actual hardware event.

I2CTarget.**memaddr**

The integer value of the most recent memory address that was selected by the I2C controller (only valid if mem was specified in the constructor).

Constants

I2CTarget.**IRQ_ADDR_MATCH_READ**

I2CTarget.**IRQ_ADDR_MATCH_WRITE**

I2CTarget.**IRQ_READ_REQ**

I2CTarget.**IRQ_WRITE_REQ**

I2CTarget.**IRQ_END_READ**

I2CTarget.**IRQ_END_WRITE**

IRQ trigger sources.

class I2S – Inter-IC Sound bus protocol

I2S is a synchronous serial protocol used to connect digital audio devices. At the physical level, a bus consists of 3 lines: SCK, WS, SD. The I2S class supports controller operation. Peripheral operation is not supported.

The I2S class is currently available as a Technical Preview. During the preview period, feedback from users is encouraged. Based on this feedback, the I2S class API and implementation may be changed.

I2S objects can be created and initialized using:

```
from machine import I2S
from machine import Pin

# ESP32
sck_pin = Pin(14)   # Serial clock output
ws_pin = Pin(13)    # Word clock output
sd_pin = Pin(12)    # Serial data output

or

# PyBoards
sck_pin = Pin("Y6")   # Serial clock output
ws_pin = Pin("Y5")    # Word clock output
sd_pin = Pin("Y8")    # Serial data output

audio_out = I2S(2,
                sck=sck_pin, ws=ws_pin, sd=sd_pin,
                mode=I2S.TX,
                bits=16,
                format=I2S.MONO,
                rate=44100,
                ibuf=20000)

audio_in = I2S(2,
               sck=sck_pin, ws=ws_pin, sd=sd_pin,
               mode=I2S.RX,
               bits=32,
               format=I2S.STEREO,
               rate=22050,
               ibuf=20000)
```

3 modes of operation are supported:

- blocking
- non-blocking

- asyncio

blocking:

```
num_written = audio_out.write(buf) # blocks until buf emptied

num_read = audio_in.readinto(buf) # blocks until buf filled
```

non-blocking:

```
audio_out.irq(i2s_callback)          # i2s_callback is called when buf is emptied
num_written = audio_out.write(buf)   # returns immediately

audio_in.irq(i2s_callback)           # i2s_callback is called when buf is filled
num_read = audio_in.readinto(buf)    # returns immediately
```

asyncio:

```
swriter = asyncio.StreamWriter(audio_out)
swriter.write(buf)
await swriter.drain()

sreader = asyncio.StreamReader(audio_in)
num_read = await sreader.readinto(buf)
```

Some codec devices like the WM8960 or SGTL5000 require separate initialization before they can operate with the I2S class. For these, separate drivers are supplied, which also offer methods for controlling volume, audio processing and other things. For these drivers see:

- *WM8960 – Driver for the WM8960 codec*

Availability: **ESP32, MIMXRT, RP2, STM32**

Constructor

class machine.**I2S**(*id*, ***, *sck*, *ws*, *sd*, *mck=None*, *mode*, *bits*, *format*, *rate*, *ibuf*)

Construct an I2S object of the given id:

- `id` identifies a particular I2S bus; it is board and port specific

Keyword-only parameters that are supported on all ports:

- `sck` is a pin object for the serial clock line
- `ws` is a pin object for the word select line
- `sd` is a pin object for the serial data line
- `mck` is a pin object for the master clock line; master clock frequency is sampling rate * 256
- `mode` specifies receive or transmit
- `bits` specifies sample size (bits), 16 or 32
- `format` specifies channel format, STEREO or MONO
- `rate` specifies audio sampling rate (Hz); this is the frequency of the `ws` signal
- `ibuf` specifies internal buffer length (bytes)

For all ports, DMA runs continuously in the background and allows user applications to perform other operations while sample data is transferred between the internal buffer and the I2S peripheral unit. Increasing the size of the internal buffer has the potential to increase the time that user applications can perform non-I2S operations before underflow (e.g. `write` method) or overflow (e.g. `readinto` method).

Methods

I2S.**init**(*sck*, ...)

see Constructor for argument descriptions

I2S.**deinit**()

Deinitialize the I2S bus

I2S.**readinto**(*buf*)

Read audio samples into the buffer specified by `buf`. `buf` must support the buffer protocol, such as bytearray or array. "buf" byte ordering is little-endian. For Stereo format, left channel sample precedes right channel sample. For Mono format, the left channel sample data is used. Returns number of bytes read

I2S.**write**(*buf*)

Write audio samples contained in `buf`. `buf` must support the buffer protocol, such as bytearray or array. "buf" byte ordering is little-endian. For Stereo format, left channel sample precedes right channel sample. For Mono format, the sample data is written to both the right and left channels. Returns number of bytes written

I2S.**irq**(*handler*)

Set a callback. `handler` is called when `buf` is emptied (`write` method) or becomes full (`readinto` method). Setting a callback changes the `write` and `readinto` methods to non-blocking operation. `handler` is called in the context of the MicroPython scheduler.

static I2S.**shift**(*, *buf*, *bits*, *shift*)

bitwise shift of all samples contained in `buf`. `bits` specifies sample size in bits. `shift` specifies the number of bits to shift each sample. Positive for left shift, negative for right shift. Typically used for volume control. Each bit shift changes sample volume by 6dB.

Constants

I2S.**RX**

for initialising the I2S bus `mode` to receive

I2S.**TX**

for initialising the I2S bus `mode` to transmit

I2S.**STEREO**

for initialising the I2S bus `format` to stereo

I2S.**MONO**

for initialising the I2S bus `format` to mono

class CAN – Controller Area Network protocol

CAN is a two-wire serial protocol used for reliable real-time message delivery between one or more nodes connected to a common bus. CAN 2.0 was standardised in ISO-11898, and is now also known as CAN Classic.

There is also a newer, backwards compatible, protocol named CAN FD (CAN with Flexible Data-Rate). *The machine.CAN driver does not currently support CAN FD features, use `pyb.CAN` on stm32 if you need CAN FD.*

CAN support requires a controller (often an internal microcontroller peripheral), and an external transceiver to level-shift the signals onto the CAN bus.

The `machine.CAN` interface is a *low level basic* CAN messaging interface that abstracts a CAN controller as an outgoing priority queue for sending messages, an incoming queue for receiving messages, and mechanisms for reporting errors.

Note: The planned `can` and `aiocan` micropython-lib modules will be the recommended way to use CAN with MicroPython.

Availability: **STM32, MIMXRT, Alif**

Constructor

class machine.**CAN**(*id*, **args*, ***kwargs*)

Construct a CAN controller object of the given id:

- `id` identifies a particular CAN controller object; it is board and port specific.
- All other arguments are passed to *CAN.init()*. At least one argument (`bitrate`) must be provided.

Future versions of this class may also accept port-specific keyword arguments here which configure the hardware. Currently no such keyword arguments are implemented.

Example

Construct and initialise CAN controller 1 with bitrate 500kbps:

```
from machine import CAN
can = CAN(1, 500_000)
```

Methods

CAN.**init**(*bitrate*, *mode=CAN.MODE_NORMAL*, *sample_point=75*, *sjw=1*, *tseg1=None*, *tseg2=None*)

Initialise the CAN bus with the given parameters:

- *bitrate* is the desired bus bit rate in bits per second.
- *mode* is one of the values shown under *Modes*, indicating the desired mode of operation. The default is "normal" operation on the bus.

The next parameters are optional and relate to CAN bit timings. In most cases you can leave these parameters set to the default values:

- *sample_point* is an integer percentage of the data bit time. It specifies the position of the bit sample with respect to the whole nominal bit time. The CAN driver will calculate parameters accordingly. This parameter is ignored if *tseg1* and *tseg2* are set.

- *sjw* is the resynchronisation jump width in units of time quanta for nominal bits; it can be a value between 1 and 4 inclusive for classic CAN.
- *tseg1* defines the location of the sample point in units of time quanta for nominal bits; it can be a value between 1 and 16 inclusive for classic CAN. This is the sum of the `Prop_Seg` and `Phase_Seg1` phases as defined in the ISO-11898 standard. If this value is set then *tseg2* must also be set and *sample_point* is ignored.
- *tseg2* defines the location of the transmit point in units of the time quanta for nominal bits; it can be a value between 1 and 8 inclusive for classic CAN. This corresponds to `Phase_Seg2` in the ISO-11898 standard. If this value is set then *tseg1* must also be set.

If these arguments are specified then the CAN controller is configured correctly for the desired *bitrate* and the specified total number of time quanta per bit. The *tseg1* and *tseg2* values override the *sample_point* argument if all of these are supplied.

Note: Individual controller hardware may have additional restrictions on valid values for these parameters, and will raise a `ValueError` if a given value is not supported.

Note: Specific controller hardware may accept additional optional keyword parameters for hardware-specific features such as oversampling.

`CAN.`**`set_filters`**(*filters*)

Set receive filters in the CAN controller. *filters* can be:

- `None` to accept all incoming messages, or
- `[]` or `()` to disable all message receiving, or
- An iterable of one or more items defining the filter criteria. Each item should be a tuple or list with three elements:
  - `identifier` is a CAN identifier (int).
  - `bit_mask` is a bit mask for bits in the CAN identifier field (int).
  - `flags` is an integer with zero or more of the bits defined in *Message Flags* set. This specifies properties that the incoming message needs to match. Not all controllers support filtering on all flags, a `ValueError` is raised if an unsupported flag is requested.

Incoming messages are accepted if the bits masked in `bit_mask` match between the message identifier and the filter `identifier` value, and flags set in the filter match the incoming message.

If the *CAN.FLAG_EXT_ID* bit is set in flags, the filter matches Extended CAN IDs only. If the *CAN.FLAG_EXT_ID* bit is not set, the filter matches Standard CAN IDs only.

All filters are ORed together in the controller. Passing an empty list or tuple for the filters argument means that no messages will be received.

Some CAN controllers require each filter to be associated with only one receive FIFO. In these cases, the filter items in the argument are allocated round-robin to the available FIFOs. This driver does not distinguish between FIFOs in the receive IRQ.

Note: If the caller passes an iterable with more items than *CAN.FILTERS_MAX*, `ValueError` will be raised.

Note: If either the `identifier` or the `bit_mask` is out of range for the specified ID type, a `ValueError` with reason "invalid id" will be raised.

Examples

Receive all incoming messages:

```
can.set_filters(None)
```

Receive messages with Standard ID values 0x301 and 0x700 only:

```
can.set_filters(((0x301, 0x7FF, 0),
                 (0x700, 0x7FF, 0)))
```

Receive messages with Standard ID values in range 0x300-0x3FF, and Extended ID value 0x50700 only:

```
can.set_filters(((0x300, 0x700, 0),
                 (0x50700, 0x1FFF_FFFF, CAN.FLAG_EXT_ID)))
```

CAN.**FILTERS_MAX**

Constant value that reads the maximum number of supported receive filters for this hardware controller.

Note that some controllers may have more complex hardware restrictions on the number of filters in use (for example, counting Standard and Extended ID filters independently.) In these cases *CAN.set_filters* may raise a `ValueError` even when the `FILTERS_MAX` limit is not exceeded.

CAN.**send**(*id*, *data*, *flags=0*)

Copy a new CAN message into the controller's hardware transmit queue to be sent onto the bus. The transmit queue is a priority queue sorted on CAN identifier priority (lower numeric identifiers have higher priority).

- *id* is an integer CAN identifier value.
- *data* is a bytes object (or similar) containing the CAN message data, or describing a Remote Transmission Request (see below).
- *flags* is an integer with zero or more of the bits defined in *Message Flags* set, specifying properties of the outgoing CAN message (Extended ID, Remote Transmission Request, etc.)

If the message is successfully queued for transmit onto the bus, the function returns an integer in the range `0` to *CAN.TX_QUEUE_LEN* (exclusive). This value is the transmit buffer index where the message is queued to send, and can be used by the *CAN.cancel_send* function and in *CAN.IRQ_TX* events.

If the queue is full then the send will fail and `None` is returned.

The send can also fail and return `None` if the provided *id* value has equal priority to an existing message in the transmit queue and the CAN controller hardware cannot guarantee that messages with the same ID will be sent onto the bus in the same order they were added to the queue. To queue the message anyway, pass the value *CAN.FLAG_UNORDERED* flag in the *flags* argument. This flag indicates that it's OK to send messages with the same CAN ID onto the bus in any order.

If the controller is in the "Bus Off" error state or disabled then calling this function will raise an `OSError`.

Note: This intentionally low-level implementation is designed so the caller can establish a software queue of outgoing messages.

Important: The CAN "transmit queue" is not a FIFO queue, it is priority ordered, and although it can hold up to *CAN.TX_QUEUE_LEN* items there may be other hardware restrictions on messages which can be queued at the same time.

Remote Transmission Requests

If the bit *CAN.FLAG_RTR* is set in the *flags* argument then the controller will send a Remote Transmission Request instead of a message. In this case the contents of the *data* argument is ignored. The controller will send a request where the `DLC` length field is equal to the length of the *data* argument.

Examples

Attempt to send a message with three byte payload `0a0b0c` and Standard ID 0x200:

```
can.send(0x200, b"\x0a\x0b\x0c", 0)
```

Attempt to send a message with an empty payload and Extended ID 0x180008. Indicate that the controller can send messages with this ID in any order, in case other messages are already queued to send with the same ID:

```
can.send(0x180008, b"", can.FLAG_EXT_ID | can.FLAG_UNORDERED)
```

Attempt to send a Remote Transmission Request with length 8 bytes and Standard ID 0x555:

```
can.send(0x555, b" " * 8, can.FLAG_RTR)
```

CAN.**recv**(*arg=None*)

Return a CAN message that has been received by the controller, according to filters set by *CAN.set_filters()*.

This function takes a single optional argument, if provided then it must be a list of at least 4 elements where the second element is a *memoryview* object that refers to a *bytearray* or similar object that has enough capacity to hold any received CAN message (8 bytes for CAN Classic, 64 bytes for CAN FD). The provided list will be returned as a successful result, and avoids memory allocation inside the function.

If no messages have been received by the CAN controller, this function returns `None`.

Note: *CAN.set_filters* must be called before any messages can be received by the controller. To receive all messages, call `set_filters(None)`.

If a message has been received by the CAN controller, this function returns a list with 4 elements:

- Index 0 is the CAN ID of the received message, as an integer.
- Index 1 is a memoryview that provides access to the received message data.
  - If *arg* is not provided then this is a *memoryview* holding the bytes which were received. This *memoryview* is backed by a newly allocated *bytearray* large enough to hold any received CAN message. This allows the result to be safely reused as a future *arg*, to save memory allocations.
  - If *arg* is provided then the provided *memoryview* will be resized to hold exactly the bytes which were received. The caller is responsible for making sure the backing object for the *memoryview* can hold a CAN message of any length.

- Index 2 is an integer with zero or more of the bits defined in *Message Flags* set. It indicates metadata about the received message.
- Index 3 is an integer with zero or more of the bits defined in *Receive Error Flags* set. Any non-zero value indicates potential issues when receiving CAN messages. These flags are reset inside the controller each time this function returns.

Remote Transmission Requests

If a Remote Transmission Request is received then the bit *CAN.FLAG_RTR* will be set in Index 2 and the memoryview at Index 1 will contain all zeroes, with a length equal to the DLC field of the received request.

Example

```
can.set_filters(None)    # receive all
while True:
    res = can.recv()
    if res:
        can_id, data, flags, errs = res
        print("Received", hex(can_id), data.hex(), hex(flags), hex(errs))
    else:
        time.sleep_ms(1)  # not a good pattern, use the irq instead!
```

CAN.**irq**(*handler=None*, *trigger=0*, *hard=False*)

Sets an interrupt *handler* function to be called when one or more of the events flagged in *trigger* has occurred.

- *handler* is a function to be called when the interrupt event triggers. The handler must take exactly one argument which is the *CAN* instance.
- *trigger* configures the event(s) which can generate an interrupt. Possible values are a mask of one or more of the following:
  - *CAN.IRQ_RX* event occurs after the CAN controller has received at least one message into its RX FIFO (meaning that *CAN.recv()* will return successfully).
  - *CAN.IRQ_TX* event occurs after the CAN controller has either successfully sent a message onto the CAN bus or failed to send a message. This trigger has additional requirements for the handler, see *IRQ flags* for details.
  - *CAN.IRQ_STATE* event occurs when the CAN controller has transitioned into a more severe error state. Call *CAN.state()* to get the updated state.
- *hard* if True, a hard interrupt is used. This reduces the delay between the CAN controller event and the handler being called. Hard interrupt handlers may not allocate memory; see *Writing interrupt handlers*.

Returns an irq object. If called with no arguments then a previously-configured irq object is returned.

See *IRQ flags* for an example.

CAN.**cancel_send**(*index*)

Request the CAN controller to cancel sending a message onto the bus.

Argument *index* identifies a single transmit buffer. It should be an integer in the range 0 to CAN.TX_QUEUE_LEN (exclusive). Generally this will be a value previously returned by *CAN.send()*.

The result is True if a message was pending transmission in this buffer and transmission was cancelled.

The result is `False` otherwise (either no message was pending transmission in this buffer, or transmission succeeded already).

The IRQ event *CAN.IRQ_TX* should be used to determine if a message was definitely sent or not, but note there are potential race conditions if a transmission is cancelled and then the same buffer is used to send another message (especially if the CAN controller IRQ is not "hard").

CAN.**state**()

Returns an integer value indicating the current state of the controller. The value will be one of the values defined in *States*.

Lower severity error states may automatically clear if the bus recovers, but the *CAN.STATE_BUS_OFF* state can only be recovered by calling *CAN.restart()*.

CAN.**get_counters**(*list=None /*)

Returns controller's error counter values. The result is a list of eight values. If the optional *list* parameter is specified then the provided list object is updated and returned as the result, to avoid an allocation.

The list items are:

- TEC (Transmit Error Counter) value
- REC (Receive Error Counter) value
- Number of times the controller entered the Warning state from the Active state.
- Number of times the controller entered the Error Passive state from the Warning state.
- Number of times the controller entered the Bus Off state from the Error Passive state.
- Total number of pending TX messages in the hardware queue.
- Total number of pending RX messages in the hardware queue.
- Number of times an RX overrun occurred.

Note: Depending on the controller, these values may overflow back to 0 after a certain value.

Note: If a controller doesn't support a particular counter, it will return `None` for that list element.

Note: The **Alif** and **MIMXRT** ports cannot report the exact number of pending RX messages. They will report a number > 0, if messages are pending.

CAN.**get_timings**(*list=None /*)

Returns a list of elements indicating the current timings configured in the CAN controller. This can be used to verify timings for debugging purposes. The result is a list of six values. If the optional *list* parameter is specified then the provided list object is updated and returned as the result, to avoid an allocation.

The list items are:

- Exact bitrate used by the controller. May vary from *bitrate* argument passed to *CAN.init()* due to quantisation to meet hardware constraints.
- Resynchronisation jump width (SJW) in units of time quanta for nominal bits. Has the same meaning as the *sjw* parameter of *CAN.init()*.

- Location of the sample point in units of time quanta for nominal bits. Has the same meaning as the *tseg1* parameter of *CAN.init()*.
- Location of the transmit point in units of time quanta for nominal bits. Has the same meaning as the *tseg2* parameter of *CAN.init()*.
- CAN FD timing information. `None` for controllers which don't support CAN FD, or if CAN FD is not initialised. Otherwise, a nested list of four elements corresponding to the items above but applicable to the CAN FD BRS feature.
- Optional controller-specific timing information. Depending on the controller this will either be `None` if controller doesn't report any, or it will be a constant length list whose elements are specific to a particular hardware controller.

Note: If *CAN.init()* has not been called then this function still returns a result, but the result depends on the controller internals and may not be accurate.

CAN.**restart**()

Causes the controller to exit *STATE_BUS_OFF* without clearing any other internal state. Also clears some of the error counters (always the number of times each error state has been entered, possibly TEC and REC depending on the controller.)

Calling this function also cancels any messages waiting to be sent. No *IRQ_TX* interrupts are delivered for these messages.

Note that this function may or may not cause the controller to exit the "Error Passive" state, depending whether the controller hardware zeroes TEC and REC or not.

CAN.**deinit**()

De-initialises a previously active CAN instance. All pending messages (transmit and receive) are dropped and the controller stops interacting on the bus. To use this instance again, call *CAN.init()*.

No *IRQ_TX* or *IRQ_RX* interrupts are called in response to calling this function.

See also *CAN.restart()*.

Constants

CAN.**TX_QUEUE_LEN**

Maximum number of CAN messages which can be queued in the outgoing hardware message queue of the controller. The "transmit buffer indexes" used by *CAN.send()*, *CAN.cancel_send()* and *IRQ flags* will be in this range.

Modes

These values represent controller modes of operation, as passed to *CAN.init()*. Not all controllers may support all modes.

Changing the mode of a running controller requires calling *CAN.deinit()* and then calling *CAN.init()* again with the new mode.

CAN.**MODE_NORMAL**

The controller is active as a standard CAN network node (will acknowledge valid messages and may transmit errors depending on its current *State*).

CAN.**MODE_SLEEP**

CAN controller is asleep in a low power mode. Depending on the controller, this may support waking the controller and transitioning to *CAN.MODE_NORMAL* if CAN traffic is received.

CAN.**MODE_LOOPBACK**

A testing mode. The CAN controller is still connected to the external bus, but will also receive its own transmitted messages and ignore any ACK errors.

CAN.**MODE_SILENT**

CAN controller receives messages but does not interact with the CAN bus (including sending ACKs, errors, etc.)

CAN.**MODE_SILENT_LOOPBACK**

A testing mode that does not require a CAN transceiver to be connected at all. The CAN controller receives its own transmitted messages without interacting with the CAN bus at all. The CAN TX and RX pins remain idle.

States

These values are returned by *CAN.state()* and reflect the error state of the CAN controller:

CAN.**STATE_STOPPED**

The controller has not been initialised.

CAN.**STATE_ACTIVE**

The controller is active and TEC and REC error counters are both below the warning threshold of 96. See *CAN.get_counters()*.

CAN.**STATE_WARNING**

The controller is active but at last one of the TEC and REC error counters are between 96 and 127. See *CAN.get_counters()*.

CAN.**STATE_PASSIVE**

The controller is in the "Error Passive" state meaning it no longer transmits active errors to the bus, but it is otherwise functional. This state is entered when at least one of the TEC and REC error counters is 128 or greater, but TEC is less than 255. See *CAN.get_counters()*.

CAN.**STATE_BUS_OFF**

The controller is in the Bus-Off state, meaning TEC error counter is greater than 255. The CAN controller will not interact with the bus in this state, and needs to be restarted via *CAN.restart()* to continue.

Message Flags

These values represent metadata about a CAN message. Functions *CAN.send()*, *CAN.recv()*, and *CAN.set_filters()* either accept or return an integer value made up of zero or more of these flags bitwise ORed together.

CAN.**FLAG_RTR**

Indicates a message is a remote transmission request.

CAN.**FLAG_EXT_ID**

If set, indicates a Message identifier is Extended (29-bit). If not set, indicates a message identifier is Standard (11-bit).

CAN.**FLAG_UNORDERED**

If set in the `flags` argument of *CAN.send()*, indicates that it's OK if messages with the same CAN ID are sent in any order onto the bus.

Otherwise trying to queue multiple messages with the same ID may result in *CAN.send()* failing if the controller hardware can't enforce ordering.

This flag is never set on received messages, and is ignored by *CAN.set_filters()*.

Receive Error Flags

The result of *CAN.recv()* includes an integer value made up of zero or more of these flags bitwise ORed together. If set, these flags indicate potential general issues with receiving CAN messages.

CAN.**RECV_ERR_FULL**

The hardware FIFO where this message was received is full, and additional incoming messages may be lost.

CAN.**RECV_ERR_OVERRUN**

The hardware FIFO where this message was received is full, and one or more incoming messages has been lost.

IRQ values

CAN.**IRQ_RX**

CAN.**IRQ_TX**

CAN.**IRQ_STATE**

IRQ event triggers. Used with *CAN.irq()* and *IRQ flags*.

CAN.**IRQ_TX_FAILED**

CAN.**IRQ_TX_IDX_SHIFT**

CAN.**IRQ_TX_IDX_MASK**

Additional IRQ event flags for *CAN.IRQ_TX*. See *IRQ flags*.

IRQ flags

Calling *CAN.irq()* registers an interrupt handler with one or more of the triggers *CAN.IRQ_RX*, *CAN.IRQ_TX* and *CAN.IRQ_STATE*.

The function returns an IRQ object, and calling the `flags()` function on this object returns an integer indicating which trigger event(s) triggered the interrupt. A CAN IRQ handler should call the `flags()` function repeatedly until it returns `0`.

When the `flags()` function returns with *CAN.IRQ_TX* bit set, the handler can also check the following flag bits in the result for additional information about the TX event:

- `CAN.IRQ_TX_FAILED` bit is set if the transmit failed. Usually this will only happen if *CAN.cancel_send()* was called, although it may also happen if the controller enters an error state.
- `CAN.IRQ_TX_MASK << CAN.IRQ_TX_SHIFT` is a bitmasked region of the flags value that holds the index of the transmit buffer which generated the event. This will be an integer in the range `0` to *CAN.TX_QUEUE_LEN* (exclusive), and will match the result of a previous call to *CAN.send()*.

IRQ_TX Example

```
from machine import CAN

def irq_send(can):
    while flags := can.irq().flags():
        if flags & can.IRQ_TX:
            idx = (flags >> can.IRQ_TX_IDX_SHIFT) & can.IRQ_TX_IDX_MASK
            success = not (flags & can.IRQ_TX_FAILED)
            print("irq_send", idx, success)

can = CAN(1, 500_000)
can.irq(irq_send, trigger=can.IRQ_TX, hard=True)
```

Important: If the *CAN.IRQ_TX* trigger is set then the handler **must** call `flags()` repeatedly until it returns `0`, as shown in this example. Otherwise, CAN interrupts may not be correctly re-enabled.

class RTC – real time clock

The RTC is an independent clock that keeps track of the date and time.

Example usage:

```
rtc = machine.RTC()
rtc.datetime((2020, 1, 21, 2, 10, 32, 36, 0))
print(rtc.datetime())
```

Availability: **Alif, ESP32, ESP8266, MIMXRT, Renesas-RA, RP2, SAMD, STM32**

Constructors

class machine.**RTC**(*id=0, ...*)

Create an RTC object. See init for parameters of initialization.

Methods

RTC.**datetime**([*datetimetuple*])

Get or set the date and time of the RTC.

With no arguments, this method returns an 8-tuple with the current date and time. With 1 argument (being an 8-tuple) it sets the date and time.

The 8-tuple has the following format:

(year, month, day, weekday, hours, minutes, seconds, subseconds)

The meaning of the `subseconds` field is hardware dependent.

RTC.**init**(*datetime*)

Initialise the RTC. Datetime is a tuple of the form:

```
(year, month, day, hour, minute, second, microsecond, tzinfo)
```

All eight arguments must be present. The `microsecond` and `tzinfo` values are currently ignored but might be used in the future.

Availability: CC3200, ESP32, MIMXRT, SAMD. The rtc.init() method on the stm32 and renesas-ra ports just (re-)starts the RTC and does not accept arguments.

`RTC.`**`now`**`()`

Get get the current datetime tuple.

Availability: WiPy.

`RTC.`**`deinit`**`()`

Resets the RTC to the time of January 1, 2015 and starts running it again.

`RTC.`**`alarm`**`(`*id*, *time*, *, *repeat=False*`)`

Set the RTC alarm. Time might be either a millisecond value to program the alarm to current time + time_in_ms in the future, or a datetimetuple. If the time passed is in milliseconds, repeat can be set to `True` to make the alarm periodic.

`RTC.`**`alarm_left`**`(`*alarm_id=0*`)`

Get the number of milliseconds left before the alarm expires.

`RTC.`**`alarm_cancel`**`(`*alarm_id=0*`)`

Cancel a running alarm.

The mimxrt port also exposes this function as `RTC.cancel(alarm_id=0)`, but this is scheduled to be removed in MicroPython 2.0.

`RTC.`**`irq`**`(`*, *trigger*, *handler=None*, *wake=machine.IDLE*`)`

Create an irq object triggered by a real time clock alarm.

- `trigger` must be `RTC.ALARM0`
- `handler` is the function to be called when the callback is triggered.
- `wake` specifies the sleep mode from where this interrupt can wake up the system.

`RTC.`**`memory`**`(`[*data*]`)`

`RTC.memory(data)` will write *data* to the RTC memory, where *data* is any object which supports the buffer protocol (including *bytes*, *bytearray*, *memoryview* and *array.array*). `RTC.memory()` reads RTC memory and returns a *bytes* object.

Data written to RTC user memory is persistent across restarts, including *Soft Reset* and *machine.deepsleep()*.

The maximum length of RTC user memory is 2048 bytes by default on esp32, and 492 bytes on esp8266.

Availability: esp32, esp8266 ports.

Note: For cross-port persistent storage, see *machine.mem_backup()* which is available on more ports and provides direct memoryview access.

Warning: On esp32, `RTC.memory()` and *machine.mem_backup()* share the same backing buffer but track length independently. Writes through one API are not reflected in the length seen by the other. Avoid mixing the two in the same application.

Constants

RTC.**ALARM0**
irq trigger source

class Timer – control hardware timers

Timer class provides the ability to trigger a Python callback function after an expiry time, or periodically at a regular interval.

The available features and restrictions of Timer objects vary depending on the MicroPython board and port.

If you are using a WiPy board please refer to *machine.TimerWiPy* instead of this class.

Timer Types

There are two types of Timer in MicroPython, but not all ports support both:

- Virtual timers. These are managed in software, and are generally more flexible. Multiple virtual timers can be constructed and active at once. The `id` of a virtual timer is `-1`.
- Hardware timers. Hardware timers have integer `id` values starting at `0`. The number of available `id` values is determined by the hardware. Hardware timers may be more accurate for very fine sub-millisecond timing (especially when `hard=True` is supported and set, see *Writing interrupt handlers*.) Most microcontroller ports support hardware timers, except Zephyr and RP2 which only support virtual timers.

Constructors

class machine.**Timer**(*id*, */*, ...)
Construct a new Timer object with the given `id`.

On ports which support virtual timers the `id` parameter is optional - the default value is `-1` which constructs a virtual timer.

On ports which support hardware timers, setting the `id` parameter to a non-negative integer determines which timer to use.

`id` shall not be passed as a keyword argument.

Any additional parameters are handled the same as *Timer.init()*.

Methods

Timer.**init**(*, *mode=Timer.PERIODIC*, *freq=-1*, *period=-1*, *callback=None*, *hard=True*)
Initialise the timer. Example:

```
def mycallback(t):
    pass

# periodic at 1kHz
tim.init(mode=Timer.PERIODIC, freq=1000, callback=mycallback)

# periodic with 100ms period
```

(continues on next page)

(continued from previous page)

```
tim.init(period=100, callback=mycallback)

# one shot firing after 1000ms
tim.init(mode=Timer.ONE_SHOT, period=1000, callback=mycallback)
```

Keyword arguments:

- `mode` can be one of:
  - `Timer.ONE_SHOT` - The timer runs once until the configured period of the channel expires.
  - `Timer.PERIODIC` - The timer runs periodically at the configured frequency of the channel.
- `freq` - The timer frequency, in units of Hz. The upper bound of the frequency is dependent on the port. When both the `freq` and `period` arguments are given, `freq` has a higher priority and `period` is ignored.
- `period` - The timer period, in milliseconds.
- `callback` - The callable to call upon expiration of the timer period. The callback must take one argument, which is passed the Timer object.

  The `callback` argument shall be specified. Otherwise an exception will occur upon timer expiration: `TypeError: 'NoneType' object isn't callable`
- `hard` can be one of:
  - `True` - The callback will be executed in hard interrupt context, which minimises delay and jitter but is subject to the limitations described in *Writing interrupt handlers*. Not all ports support hard interrupts, see the port documentation for more information.
  - `False` - The callback will be scheduled as a soft interrupt, allowing it to allocate but possibly also introducing garbage-collection delays and jitter.

  The default value of this parameter is port-specific for historical reasons.

`Timer.`**`deinit`**`()`

Deinitialises the timer. Stops the timer, and disables the timer peripheral.

Constants

`Timer.`**`ONE_SHOT`**

`Timer.`**`PERIODIC`**

Timer operating mode.

class Counter – pulse counter

Counter implements pulse counting by monitoring an input signal and counting rising or falling edges.

Minimal ESP32 example usage:

```
from machine import Pin, Counter

counter = Counter(0, Pin(0, Pin.IN))  # create Counter for pin 0 and begin counting
value = counter.value()               # retrieve current pulse count
```

Availability: **ESP32, MIMXRT**

Constructors

class machine.**Counter**(*id*, ...)

Returns the singleton Counter object for the the given *id*. Values of *id* depend on a particular port and its hardware. Values 0, 1, etc. are commonly used to select hardware block #0, #1, etc.

Additional arguments are passed to the `init()` method described below, and will cause the Counter instance to be re-initialised and reset.

On ESP32, the *id* corresponds to a *PCNT unit*.

Methods

Counter.**init**(*src*, *, ...)

Initialise and reset the Counter with the given parameters:

- *src* specifies the input pin as a *machine.Pin* object. May be omitted on ports that have a predefined pin for a given hardware block.

Additional keyword-only parameters that may be supported by a port are:

- *edge* specifies the edge to count. Either `Counter.RISING` (the default) or `Counter.FALLING`. *(Supported on ESP32)*
- *direction* specifies the direction to count. Either `Counter.UP` (the default) or `Counter.DOWN`. *(Supported on ESP32 and MIMXRT)* A *machine.Pin* object as parameter argument specifies a pin which controls the counting direction. Low: Count up, High: Count down. *(Supported on MIMXRT)*
- *filter_ns* specifies a minimum period of time in nanoseconds that the source signal needs to be stable for a pulse to be counted. Implementations should use the longest filter supported by the hardware that is less than or equal to this value. The default is 0 (no filter). *(Supported on ESP32 and MIMXRT)*
- *max* Specify the upper counting range. The position counter will count up from a *min* start value up to *max*, then roll over to the init value and increase the cycles counter by one. When counting down, the cycles counter decreases at the transition from *min* to *max*. The range is reset by defining both *max* and *min* to 0. The default value is the hardware's counter range. *(Supported by MIMXRT and the ESP32 PCNT module)*
- *min* Specify the lower counting range. The default value is 0. *(Supported by MIMXRT and the ESP32 PCNT module)*
- *index* A Pin specifier telling to which pin the index pulse is connected. At a rising slope of the index pulse the pulse counter is reset to the min value and the cycles counter is increased or decreased by one, depending on the counting direction. A *value* of *None* disables the index input. *(Supported on MIMXRT)*
- *reset* A Pin specifier telling to which pin the reset pulse is connected. At a rising slope of the reset pulse the counter is set to the init value, but the cycles counter is not changed. A *value* of *None* disables the reset input. *(Supported on MIMXRT)*
- *match* Set the counter value at which the interrupt IRQ_MATCH shall trigger. The value is not checked for being in the bounds of the counter range. This option if equivalent to the *threshold* options of the ESP32 PCNT module. A *value* of *None* resets the match value and disables the IRQ_MATCH interrupt. *(Supported on MIMXRT)*
- *match_pin* A Pin specifier telling to which pin the match output is connected. This output will have a high level as long as the counter matches the match value. The signal is generated by the encoder logic and requires no further software support. The pulse width is defined by the input signal frequency and can be very short, like 20ns, or stay, if the counter stops at the match value. A *value* of *None* disables the match output. *(Supported on MIMXRT)*

`Counter.`**`deinit`**`()`

Stops the Counter, disabling any interrupts and releasing hardware resources. A Soft Reset should deinitialize all Counter objects.

`Counter.`**`value`**`(`[*value*]`)`

Get, and optionally set, the counter value as a signed integer. Implementations must aim to do the get and set atomically (i.e. without leading to skipped counts).

This counter value could exceed the range of a *small integer*, which means that calling `Counter.value()` could cause a heap allocation, but implementations should aim to ensure that internal state only uses small integers and therefore will not allocate until the user calls `Counter.value()`.

For example, on ESP32, the internal state counts overflows of the hardware counter (every 32000 counts), which means that it will not exceed the small integer range until `2**30 * 32000` counts (slightly over 1 year at 1MHz).

In general, it is recommended that you should use `Counter.value(0)` to reset the counter (i.e. to measure the counts since the last call), and this will avoid this problem.

`Counter.`**`cycles`**`(`[*value*]`)`

Get or set the current cycles counter of the counter as signed 16 bit integer. The value represents the overflow or underflow events of the count range. With no arguments the actual cycles counter value is returned. With a single *value* argument the cycles counter is set to that value. The base counter is not changed. The method returns the previous value. *(Supported on MIMXRT)*

`Counter.`**`irq`**`(`*handler=None*, *trigger=0*, *hard=False*`)`

Specifies, that the *handler* is called when the respective *event* happens.

***event* may be:**

- Counter.IRQ_RESET Triggered with a transition at the *reset* input.
- Counter.IRQ_INDEX Triggered with a transition at the *index* input.
- Counter.IRQ_MATCH Triggered when the positions counter matches the match value. For fast signals, the actual position counter value when retrieved in the callback may be different from the trigger value.
- Counter.IRQ_ROLL_OVER Triggered when the position counter rolls over from the highest to the lowest value.
- Counter.IRQ_ROLL_UNDER Triggered when the position counter rolls under from the lowest to the highest value.

The callback function *handler* receives a single argument, which is the Counter object. All events share the same callback. The event which triggers the callback can be identified with the irq.flags() method. The argument *hard* specifies, whether the callback is called as a hard interrupt or as regular scheduled function. Hard interrupts have always a short latency, but are limited in that they must not allocate memory. Regular scheduled functions are not limited in what can be used, but depending on the load of the device execution may be delayed. Under low load, the difference in latency is minor.

The default arguments values are handler=None, trigger=0, hard=False. The callback will be disabled, when called with handler=None.

The position match event is triggered as long as the position and match value are identical. Therefore the position match callback is run in a one-shot fashion, and has to be enabled again when the position has changed. It will be enabled by re-defining the trigger with either `Counter.irq()` or `irq().trigger()`. For ESP32, Counter interrupts are handled by the *PCNT*. *(Supported on MIMXRT)*

Constants

`Counter.`**`RISING`**

`Counter.`**`FALLING`**

Select the pulse edge. *(Supported on ESP32)*

`Counter.`**`UP`**

`Counter.`**`DOWN`**

Select the counting direction.

`Counter.`**`IRQ_RESET`**

`Counter.`**`IRQ_INDEX`**

`Counter.`**`IRQ_MATCH`**

`Counter.`**`IRQ_ROLL_OVER`**

`Counter.`**`IRQ_ROLL_UNDER`**

Select the IRQ trigger event. *(Supported on MIMXRT)*

class Encoder – quadrature decoding

Encoder implements decoding of quadrature signals as commonly output from rotary encoders, by counting either up or down depending on the order of two input pulses.

Minimal ESP32 example usage:

```
from machine import Pin, Encoder

encoder = Encoder(0, Pin(0, Pin.IN), Pin(1, Pin.IN))    # create Encoder for pins 0, 1␣
↪and begin counting
value = encoder.value()                                 # retrieve current count
```

Availability: **ESP32, MIMXRT**

Constructors

`class` `machine.`**`Encoder`**(*id*, ...)

Returns the singleton Encoder object for the the given *id*. Values of *id* depend on a particular port and its hardware. Values 0, 1, etc. are commonly used to select hardware block #0, #1, etc.

Additional arguments are passed to the *`init()`* method described below, and will cause the Encoder instance to be re-initialised and reset.

On ESP32, the *id* corresponds to a *PCNT unit*.

Methods

Encoder.**init**(*phase_a*, *phase_b*, *, ...)

Initialise and reset the Encoder with the given parameters:

- *phase_a* specifies the first input pin as a *machine.Pin* object.
- *phase_b* specifies the second input pin as a *machine.Pin* object.

These pins may be omitted on ports that have predefined pins for a given hardware block.

Additional keyword-only parameters that may be supported by a port are:

- *filter_ns* specifies a minimum period of time in nanoseconds that the source signal needs to be stable for a pulse to be counted. Implementations should use the longest filter supported by the hardware that is less than or equal to this value. The default is 0 (no filter). *(Supported on ESP32 and MIMXRT)*
- *phases* specifies the number of signal edges to count and thus the granularity of the decoding. e.g. 4 phases corresponds to "4x quadrature decoding", and will result in four counts per pulse. Ports may support either 1, 2, or 4 phases and the default is 1 phase. *(Supported on ESP32 and MIMXRT)*
- *max* Specify the upper counting range. The position counter will count up from a *min* start value up to *max*, then roll over to the init value and increase the cycles counter by one. When counting down, the cycles counter decreases at the transition from *min* to *max*. The range is reset by defining both *max* and *min* to 0. The default value is the hardware's counter range. *(Supported by MIMXRT and the ESP32 PCNT module)*
- *min*. Specify the lower counting range. The default value is 0. *(Supported by MIMXRT and the ESP32 PCNT module)*
- *index* A Pin specifier telling to which pin the index pulse is connected. At a rising slope of the index pulse the encoder counter is set to the min value and the cycles counter is increased or decreased by one, depending on the input levels. A *value* of *None* disables the index input. *(Supported on MIMXRT)*
- *reset* A Pin specifier telling to which pin the reset pulse is connected. At a rising slope of the reset pulse the position counter is set to the init value, but the cycles counter is not changed. A *value* of *None* disables the reset input. *(Supported on MIMXRT)*
- *match* Set the counter value at which the interrupt IRQ_MATCH shall trigger. The value is not checked for being in the bounds of the counter range. This option if equivalent to the *threshold* options of the ESP32 PCNT module. A *value* of *None* resets the match value and disables the IRQ_MATCH interrupt. *(Supported on MIMXRT)*
- *match_pin* A Pin specifier telling to which pin the match output is connected. This output will have a high level as long as the position counter matches the match value. The signal is generated by the encoder logic and requires no further software support. The pulse width is defined by the input signal frequency and can be very short, like 20ns, or stay, if the counter stops at the match position. A *value* of *None* disables the match output. *(Supported on MIMXRT)*

Encoder.**deinit**()

Stops the Encoder, disabling any interrupts and releasing hardware resources. A Soft Reset should deinitialize all Encoder objects.

Encoder.**value**([*value*])

Get, and optionally set, the encoder value as a signed integer. Implementations should aim to do the get and set atomically.

See *machine.Counter.value()* for details about overflow of this value.

Encoder.**cycles**([*value*])

Get or set the current cycles counter of the counter as signed 16 bit integer. The value represents the overflow or underflow events of the count range. With no arguments the actual cycles counter value is returned. With a single

value argument the cycles counter is set to that value. The base counter is not changed. The method returns the previous value. *(Supported on MIMXRT)*

`Encoder.`**`irq`**(*handler=None*, *trigger=0*, *hard=False*)

Specifies, that the *handler* is called when the respective *event* happens.

***event* may be:**

- Encoder.IRQ_RESET Triggered with a transition at the *reset* input.
- Encoder.IRQ_INDEX Triggered with a transition at the *index* input.
- Encoder.IRQ_MATCH Triggered when the position counter matches the *match* value. For fast signals, the actual position counter value when retrieved in the callback may be different from the trigger value.
- Encoder.IRQ_ROLL_OVER Triggered when the position counter rolls over from the highest to the lowest value.
- Encoder.IRQ_ROLL_UNDER Triggered when the position counter rolls under from the lowest to the highest value.

The callback function *handler* receives a single argument, which is the Encoder object. All events share the same callback. The event which triggers the callback can be identified with the irq.flags() method. The argument *hard* specifies, whether the callback is called as a hard interrupt or as regular scheduled function. Hard interrupts have always a short latency, but are limited in that they must not allocate memory. Regular scheduled functions are not limited in what can be used, but depending on the load of the device execution may be delayed. Under low load, the difference in latency is minor.

The default arguments values are trigger=0, handler=None, hard=False. The callback will be disabled, when called with handler=None.

The position match event is triggered as long as the position and match value are identical. Therefore the position match callback is run in a one-shot fashion, and has to be enabled again when the position has changed. It will be enabled by re-defining the trigger with either *`Encoder.irq()`* or `irq().trigger()`. For ESP32, Encoder interrupts are handled by the *PCNT unit*.

(Supported on MIMXRT)

Constants

`Encoder.`**`IRQ_RESET`**

`Encoder.`**`IRQ_INDEX`**

`Encoder.`**`IRQ_MATCH`**

`Encoder.`**`IRQ_ROLL_OVER`**

`Encoder.`**`IRQ_ROLL_UNDER`**

Select the IRQ trigger event. *(Supported on MIMXRT)*

class WDT – watchdog timer

The WDT is used to restart the system when the application crashes and ends up into a non recoverable state. Once started it cannot be stopped or reconfigured in any way. After enabling, the application must "feed" the watchdog periodically to prevent it from expiring and resetting the system.

Example usage:

```
from machine import WDT
wdt = WDT(timeout=2000)  # enable it with a timeout of 2s
wdt.feed()
```

Availability: **Alif, ESP32, ESP8266, MIMXRT, RP2, SAMD, STM32, Zephyr**

Constructors

class machine.**WDT**(*id=0*, *timeout=5000*)

Create a WDT object and start it. The timeout must be given in milliseconds. Once it is running the timeout cannot be changed and the WDT cannot be stopped either.

Notes:

- On the alif port the HP and HE cores have independent watchdogs, both accessed by the default `id=0`. The maximum timeout on the HP core is 10737ms. The watchdog does not run during deepsleep.
- On the esp8266 port a timeout cannot be specified, it is determined by the underlying system.
- On rp2040 devices the maximum timeout is 8388 ms.
- On the stm32 port the default `id=0` is the IWDG, which can also be specified by an id of `"IWDG"`. Use an id of `"WWDG"` to access the WWDG peripheral. For dual-core STM32H7 MCUs there are also `"IWDG2"` and `"WWDG2"`. The WWDG has a very limited maximum timeout across all MCUs, of around 100ms (but it depends heavily on the APB clock).

Methods

WDT.**feed**()

Feed the WDT to prevent it from resetting the system. The application should place this call in a sensible place ensuring that the WDT is only fed after verifying that everything is functioning correctly.

class SD – secure digital memory card (cc3200 port only)

Warning: This is a non-standard class and is only available on the cc3200 port.

The SD card class allows to configure and enable the memory card module of the WiPy and automatically mount it as `/sd` as part of the file system. There are several pin combinations that can be used to wire the SD card socket to the WiPy and the pins used can be specified in the constructor. Please check the pinout and alternate functions table. for more info regarding the pins which can be remapped to be used with a SD card.

Example usage:

```
from machine import SD
import vfs
# clk cmd and dat0 pins must be passed along with
# their respective alternate functions
sd = machine.SD(pins=('GP10', 'GP11', 'GP15'))
vfs.mount(sd, '/sd')
# do normal file operations
```

Constructors

class machine.**SD**(*id*, ...)

Create a SD card object. See `init()` for parameters if initialization.

Methods

`SD`.**init**(*id=0*, *pins=('GP10', 'GP11', 'GP15')*)

Enable the SD card. In order to initialize the card, give it a 3-tuple: `(clk_pin, cmd_pin, dat0_pin)`.

`SD`.**deinit**()

Disable the SD card.

class SDCard – secure digital memory card

SD cards are one of the most common small form factor removable storage media. SD cards come in a variety of sizes and physical form factors. MMC cards are similar removable storage devices while eMMC devices are electrically similar storage devices designed to be embedded into other systems. All three form share a common protocol for communication with their host system and high-level support looks the same for them all. As such in MicroPython they are implemented in a single class called `machine.SDCard`.

Both SD and MMC interfaces support being accessed with a variety of bus widths. When being accessed with a 1-bit wide interface they can be accessed using the SPI protocol. Different MicroPython hardware platforms support different widths and pin configurations but for most platforms there is a standard configuration for any given hardware. In general constructing an `SDCard` object with without passing any parameters will initialise the interface to the default card slot for the current hardware. The arguments listed below represent the common arguments that might need to be set in order to use either a non-standard slot or a non-standard pin assignment. The exact subset of arguments supported will vary from platform to platform.

SDCard(slot=1, width=1, cd=None, wp=None, sck=None, miso=None, mosi=None, cs=None, cmd=None, data=None, ldo=None, freq=20000000)

This class provides access to SD or MMC storage cards using either a dedicated SD/MMC interface hardware or through an SPI channel. The class implements the block protocol defined by *vfs.AbstractBlockDev*. This allows the mounting of an SD card to be as simple as:

```
vfs.mount(machine.SDCard(), "/sd")
```

The constructor takes the following parameters:

- *slot* selects which of the available interfaces to use. Leaving this unset will select the default interface.
- *width* selects the bus width for the SD/MMC interface. This many data pins must be connected to the SD card.
- *cd* can be used to specify a card-detect pin.

- *wp* can be used to specify a write-protect pin.
- *sck* can be used to specify an SPI clock pin.
- *miso* can be used to specify an SPI miso pin.
- *mosi* can be used to specify an SPI mosi pin.
- *cs* can be used to specify an SPI chip select pin.

The following additional parameters are only present on ESP32 port:

- *cmd* can be used to specify the SD CMD pin (ESP32-S3 only).
- *data* can be used to specify a list or tuple of SD data bus pins (ESP32-S3 only).
- *ldo* can be used to specify the internal LDO channel used for SD card logic level for SDIO 3.0 (ESP32-P4 only).
- *freq* selects the SD/MMC interface frequency in Hz.

Implementation-specific details

Different implementations of the `SDCard` class on different hardware support varying subsets of the options above.

PyBoard

The standard PyBoard has just one slot. No arguments are necessary or supported.

ESP32

SD cards support access in both SD/MMC mode and the simpler (but slower) SPI mode.

SPI mode makes use of a *SPI* host peripheral, which cannot concurrently be used for other SPI interactions.

The `slot` argument determines which mode is used. Different values are supported on different chips:

Chip	Slot 0	Slot 1	Slot 2	Slot 3
ESP32		SD/MMC	SPI (id=1)	SPI (id=0)
ESP32-C3			SPI (id=0)	
ESP32-C6			SPI (id=0)	
ESP32-S2			SPI (id=1)	SPI (id=0)
ESP32-S3	SD/MMC	SD/MMC	SPI (id=1)	SPI (id=0)

Different slots support different data bus widths (number of data pins):

Slot	Type	Supported data widths
0	SD/MMC	1, 4, 8
1	SD/MMC	1, 4
2	SPI	1
3	SPI	1

Note: Most ESP32 modules that provide an SD card slot using the dedicated hardware only wire up 1 data pin, so the default value for `width` is 1.

Additional details depend on which ESP32 family chip is in use:

Original ESP32

In SD/MMC mode (slot 1), pin assignments in SD/MMC mode are fixed on the original ESP32. The SPI mode slots (2 & 3) allow pins to be set to different values in the constructor.

The default pin assignments are as follows:

Slot Signal	1 Pin	2 Pin	3 Pin	Can be set
CLK	14			No
CMD	15			No
D0	2			No
D1	4			No
D2	12			No
D3	13			No
sck		18	14	Yes
cs		5	15	Yes
miso		19	12	Yes
mosi		23	13	Yes

The `cd` and `wp` pins are not fixed in either mode and default to disabled, unless set.

ESP32-S3

The ESP32-S3 chip allows pins to be set to different values for both SD/MMC and SPI mode access.

If not set, default pin assignments are as follows:

Slot Signal	0 Pin	1 Pin	2 Pin	3 Pin
CLK	14	14		
CMD	15	15		
D0	2	2		
D1	4	4		
D2	12	12		
D3	13	13		
D4	33*			
D5	34*			
D6	35*			
D7	36*			
sck			37*	14
cs			34*	13
miso			37*	2
mosi			35*	15

Note: Slots 0 and 1 cannot both be in use at the same time.

Note: Pins marked with an asterisk * in the table must be changed from the default if the ESP32-S3 board is configured for Octal SPI Flash or PSRAM.

To access a card in SD/MMC mode, set `slot` parameter value 0 or 1 and parameters `sck` (for CLK), `cmd` and `data` as needed to assign pins. If the `data` argument is passed then it should be a list or tuple of data pins or pin numbers with length equal to the `width` argument. For example:

```
sd = SDCard(slot=0, width=4, sck=8, cmd=9, data=(10, 11, 12, 13))
```

To access a card in SPI mode, set `slot` parameter value 2 or 3 and pass parameters `sck`, `cs`, `miso`, `mosi` as needed to assign pins.

In either mode the `cd` and `wp` pins default to disabled, unless set in the constructor.

ESP32-P4

The ESP32-P4 has multiple internal adjustable LDO regulators, and some boards use one of LDOs to control the SD card logic level required by SDIO 3.0. Most boards will automatically select the correct LDO channel, but it may be necessary to manually specify the `ldo` parameter as an integer (1 through 4). For example:

```
sd = SDCard(ldo=4)
```

Other ESP32 chips

Other ESP32 family chips do not have hardware SD/MMC host controllers and can only access SD cards in SPI mode.

To access a card in SPI mode, set `slot` parameter value 2 or 3 and pass parameters `sck`, `cs`, `miso`, `mosi` to assign pins.

Note: ESP32-C3 and ESP32-C6 only have one available *SPI* bus, so the only valid `slot` parameter value is 2. Using this bus for the SD card will prevent also using it for *machine.SPI*.

cc3200

You can set the pins used for SPI access by passing a tuple as the *pins* argument.

Note: The current cc3200 SD card implementation names the this class *machine.SD* rather than `machine.SDCard` .

mimxrt

The SDCard module for the mimxrt port only supports access via dedicated SD/MMC peripheral (USDHC) in 4-bit mode with 50MHz clock frequency exclusively. Unfortunately the MIMXRT1011 controller does not support the USDHC peripheral. Hence this controller does not feature the `machine.SDCard` module.

Due to the decision to only support 4-bit mode with 50MHz clock frequency the interface has been simplified, and the constructor signature is:

class machine.**SDCard**(*slot=1*)

The pins used for the USDHC peripheral have to be configured in `mpconfigboard.h`. Most of the controllers supported by the mimxrt port provide up to two USDHC peripherals. Therefore the pin configuration is performed using the macro `MICROPY_USDHCx` with x being 1 or 2 respectively.

The following shows an example configuration for USDHC1:

```
#define MICROPY_USDHC1 \
  { \
        .cmd   = { GPIO_SD_B0_02_USDHC1_CMD}, \
        .clk   = { GPIO_SD_B0_03_USDHC1_CLK }, \
        .cd_b  = { GPIO_SD_B0_06_USDHC1_CD_B },\
        .data0 = { GPIO_SD_B0_04_USDHC1_DATA0 },\
        .data1 = { GPIO_SD_B0_05_USDHC1_DATA1 },\
        .data2 = { GPIO_SD_B0_00_USDHC1_DATA2 },\
        .data3 = { GPIO_SD_B0_01_USDHC1_DATA3 },\
  }
```

If the card detect pin is not used (cb_b pin) then the respective entry has to be filled with the following dummy value:

```
#define USDHC_DUMMY_PIN NULL , 0
```

Based on the definition of macro `MICROPY_USDHC1` and/or `MICROPY_USDHC2` the `machine.SDCard` module either supports one or two slots. If only one of the defines is provided, calling `machine.SDCard()` or `machine.SDCard(1)` will return an instance using the respective USDHC peripheral. When both macros are defined, calling `machine.SDCard(2)` returns an instance using USDHC2.

class USBDevice – USB Device driver

Availability: **ESP32, RP2, SAMD**

Note: Native USB support is required, and not every board supports native USB.

USBDevice provides a low-level Python API for implementing USB device functions using Python code.

Warning: This low-level API assumes familiarity with the USB standard. There are high-level usb driver modules in micropython-lib which provide a simpler interface and more built-in functionality.

Terminology

- A "Runtime" USB device interface or driver is one which is defined using this Python API after MicroPython initially starts up.
- A "Built-in" USB device interface or driver is one that is compiled into the MicroPython firmware, and is always available. Examples are USB-CDC (serial port) which is usually enabled by default. Built-in USB-MSC (Mass Storage) is an option on some ports.

Lifecycle

Managing a runtime USB interface can be tricky, especially if you are communicating with MicroPython over a built-in USB-CDC serial port that's part of the same USB device.

- A MicroPython *soft reset* will always clear all runtime USB interfaces, which results in the entire USB device disconnecting from the host. If MicroPython is also providing a built-in USB-CDC serial port then this will re-appear after the soft reset.

  This means some functions (like `mpremote run`) that target the USB-CDC serial port will immediately fail if a runtime USB interface is active, because the port goes away when `mpremote` triggers a soft reset. The operation should succeed on the second try, as after the soft reset there is no more runtime USB interface.
- To configure a runtime USB device on every boot, it's recommended to place the configuration code in the *boot.py* file on the *device VFS*. On each reset this file is executed before the USB subsystem is initialised (and before *main.py*), so it allows the board to come up with the runtime USB device immediately.
- For development or debugging, it may be convenient to connect a hardware serial REPL and disable the built-in USB-CDC serial port entirely. Not all ports support this (currently only `rp2`). The custom build should be configured with `#define MICROPY_HW_USB_CDC (0)` and `#define MICROPY_HW_ENABLE_UART_REPL (1)`.

Constructors

class machine.**USBDevice**

Construct a USBDevice object.

Note: This object is a singleton, each call to this constructor returns the same object reference.

Methods

USBDevice.**config**(*desc_dev*, *desc_cfg*, *desc_strs=None*, *open_itf_cb=None*, *reset_cb=None*, *control_xfer_cb=None*, *xfer_cb=None*)

Configures the `USBDevice` singleton object with the USB runtime device state and callback functions:

- `desc_dev` - A bytes-like object containing the new USB device descriptor.
- `desc_cfg` - A bytes-like object containing the new USB configuration descriptor.
- **`desc_strs` - Optional object holding strings or bytes objects** containing USB string descriptor values. Can be a list, a dict, or any object which supports subscript indexing with integer keys (USB string descriptor index).

Strings are an optional USB feature, and this parameter can be unset (default) if no strings are referenced in the device and configuration descriptors, or if only built-in strings should be used.

Apart from index 0, all the string values should be plain ASCII. Index 0 is the special "languages" USB descriptor, represented as a bytes object with a custom format defined in the USB standard. `None` can be returned at index 0 in order to use a default "English" language descriptor.

To fall back to providing a built-in string value for a given index, a subscript lookup can return `None`, raise `KeyError`, or raise `IndexError`.

- `open_itf_cb` - This callback is called once for each interface or Interface Association Descriptor in response to a Set Configuration request from the USB Host (the final stage before the USB device is available to the host).

  The callback takes a single argument, which is a memoryview of the interface or IAD descriptor that the host is accepting (including all associated descriptors). It is a view into the same `desc_cfg` object that was provided as a separate argument to this function. The memoryview is only valid until the callback function returns.

- `reset_cb` - This callback is called when the USB host performs a bus reset. The callback takes no arguments. Any in-progress transfers will never complete. The USB host will most likely proceed to re-enumerate the USB device by calling the descriptor callbacks and then `open_itf_cb()`.

- `control_xfer_cb` - This callback is called one or more times for each USB control transfer (device Endpoint 0). It takes two arguments.

  The first argument is the control transfer stage. It is one of:

  - `1` for SETUP stage.
  - `2` for DATA stage.
  - `3` for ACK stage.

  Second argument is a memoryview to read the USB control request data for this stage. The memoryview is only valid until the callback function returns. Data in this memoryview will be the same across each of the three stages of a single transfer.

  A successful transfer consists of this callback being called in sequence for the three stages. Generally speaking, if a device wants to do something in response to a control request then it's best to wait until the ACK stage to confirm the host controller completed the transfer as expected.

  The callback should return one of the following values:

  - `False` to stall the endpoint and reject the transfer. It won't proceed to any remaining stages.
  - `True` to continue the transfer to the next stage.
  - A buffer object can be returned at the SETUP stage when the transfer will send or receive additional data. Typically this is the case when the `wLength` field in the request has a non-zero value. This should be a writable buffer for an `OUT` direction transfer, or a readable buffer with data for an `IN` direction transfer.

- `xfer_cb` - This callback is called whenever a non-control transfer submitted by calling *`USBDevice.submit_xfer()`* completes.

  The callback has three arguments:

  1. The Endpoint number for the completed transfer.
  2. Result value. This is an integer which is `0` (*`XFER_SUCCESS`*) on success, or one of the non-zero values *`XFER_FAILED`* or *`XFER_STALLED`* if the transfer failed.

3. Number of bytes successfully transferred. In the case of a "short" transfer, the result is `0` (*XFER_SUCCESS*) and `xferred_bytes` will be smaller than the length of the buffer submitted for the transfer.

Note: If a bus reset occurs (see `USBDevice.reset()`), `xfer_cb` is not called for any transfers that have not already completed.

`USBDevice.`**`active`**(*self, [value] /*)

Returns the current active state of this runtime USB device as a boolean. The runtime USB device is "active" when it is available to interact with the host, it doesn't mean that a USB Host is actually present.

If the optional `value` argument is set to a truthy value, then the USB device will be activated.

If the optional `value` argument is set to a falsey value, then the USB device is deactivated. While the USB device is deactivated, it will not be detected by the USB Host.

To simulate a disconnect and a reconnect of the USB device, call `active(False)` followed by `active(True)`. This may be necessary if the runtime device configuration has changed, so that the host sees the new device.

`USBDevice.`**`builtin_driver`**

This attribute holds the current built-in driver configuration, and must be set to one of the `USBDevice.BUILTIN_` named constants defined on this object.

By default it holds the value *USBDevice.BUILTIN_NONE*.

Runtime USB device must be inactive when setting this field. Call the *USBDevice.active()* function to deactivate before setting if necessary (and again to activate after setting).

If this value is set to any value other than *USBDevice.BUILTIN_NONE* then the following restrictions apply to the *USBDevice.config()* arguments:

- `desc_cfg` should begin with the built-in USB interface descriptor data accessible via *USBDevice.builtin_driver* attribute `desc_cfg`. Descriptors appended after the built-in configuration descriptors should use interface, string and endpoint numbers starting from the max built-in values defined in *USBDevice.builtin_driver* attributes `itf_max`, `str_max` and `ep_max`.
- The `bNumInterfaces` field in the built-in configuration descriptor will also need to be updated if any new interfaces are appended to the end of `desc_cfg`.
- `desc_strs` should either be `None` or a list/dictionary where index values less than `USBDevice.builtin_driver.str_max` are missing or have value `None`. This reserves those string indexes for the built-in drivers. Placing a different string at any of these indexes overrides that string in the built-in driver.

`USBDevice.`**`remote_wakeup`**(*self*)

Wake up host if we are in suspend mode and the REMOTE_WAKEUP feature is enabled by the host. This has to be enabled in the USB attributes, and on the host. Returns `True` if remote wakeup was enabled and active and the host was woken up.

`USBDevice.`**`submit_xfer`**(*self*, *ep*, *buffer /*)

Submit a USB transfer on endpoint number `ep`. `buffer` must be an object implementing the buffer interface, with read access for `IN` endpoints and write access for `OUT` endpoints.

Note: `ep` cannot be the control Endpoint number 0. Control transfers are built up through successive executions of `control_xfer_cb`, see above.

Returns `True` if successful, `False` if the transfer could not be queued (as USB device is not configured by host, or because another transfer is queued on this endpoint.)

When the USB host completes the transfer, the `xfer_cb` callback is called (see above).

Raises `OSError` with reason `MP_EINVAL` If the USB device is not active.

`USBDevice.`**`stall`**(*self, ep, [stall]*`/`)

Calling this function gets or sets the STALL state of a device endpoint.

`ep` is the number of the endpoint.

If the optional `stall` parameter is set, this is a boolean flag for the STALL state.

The return value is the current stall state of the endpoint (before any change made by this function).

An endpoint that is set to STALL may remain stalled until this function is called again, or STALL may be cleared automatically by the USB host.

Raises `OSError` with reason `MP_EINVAL` If the USB device is not active.

Constants

`USBDevice.`**`BUILTIN_NONE`**

`USBDevice.`**`BUILTIN_DEFAULT`**

`USBDevice.`**`BUILTIN_CDC`**

`USBDevice.`**`BUILTIN_MSC`**

`USBDevice.`**`BUILTIN_CDC_MSC`**

These constant objects hold the built-in descriptor data which is compiled into the MicroPython firmware. `USBDevice.BUILTIN_NONE` and `USBDevice.BUILTIN_DEFAULT` are always present. Additional objects may be present depending on the firmware build configuration and the actual built-in drivers.

Note: Currently at most one of `USBDevice.BUILTIN_CDC`, `USBDevice.BUILTIN_MSC` and `USBDevice.BUILTIN_CDC_MSC` is defined and will be the same object as `USBDevice.BUILTIN_DEFAULT`. These constants are defined to allow run-time detection of the built-in driver (if any). Support for selecting one of multiple built-in driver configurations may be added in the future.

These values are assigned to *`USBDevice.builtin_driver`* to get/set the built-in configuration.

Each object contains the following read-only fields:

- `itf_max` - One more than the highest bInterfaceNumber value used in the built-in configuration descriptor.
- `ep_max` - One more than the highest bEndpointAddress value used in the built-in configuration descriptor. Does not include any `IN` flag bit (0x80).
- `str_max` - One more than the highest string descriptor index value used by any built-in descriptor.
- `desc_dev` - `bytes` object containing the built-in USB device descriptor.
- `desc_cfg` - `bytes` object containing the complete built-in USB configuration descriptor.

`USBDevice.`**`XFER_SUCCESS`**

`USBDevice.`**`XFER_FAILED`**

USBDevice.**XFER_STALLED**

These are integer constants that represent the possible transfer result values passed to the `xfer_cb` callback (see *USBDevice.config*).

- `XFER_SUCCESS` has value `0` and indicates the transfer was successful.
- `XFER_FAILED` indicates the transfer failed due to low-level integrity errors.
- `XFER_STALLED` indicates that the host has stalled this endpoint.

All failure values are non-zero integers.

1.2.7 `micropython` – access and control MicroPython internals

Functions

micropython.**const**(*expr*)

Used to declare that the expression is a constant so that the compiler can optimise it. The use of this function should be as follows:

```
from micropython import const

CONST_X = const(123)
CONST_Y = const(2 * CONST_X + 1)
```

When the parser encounters `NAME = const(expr)`, it evaluates the expression at compile time and substitutes the resulting value directly into the bytecode at every use of `NAME`, avoiding a global dictionary lookup each time.

The `const` name is recognised directly by the parser so no import is actually required in MicroPython. However `from micropython import const` is recommended so the script also runs on CPython where `const` is provided as an identity function. Note that the parser only recognises the bare name `const` – using `micropython.const()` (with module prefix) or an alias will not trigger the optimisation.

Constants declared this way are still stored as global variables in the module's dictionary, so other modules can access them (e.g. `import mymodule; print(mymodule.X)`). Reassigning the global from another module does not affect the inlined values within the defining module. This global entry costs at least two machine words of RAM.

Module-private constants: To avoid this RAM cost, prefix the name with an underscore (e.g. `_X = const(1)`). This prevents the variable from being added to the module dictionary and hides it from other modules.

The expression passed to `const()` must be evaluable at compile time. Supported types are:

- `int` (including expressions with arithmetic and bitwise operators)
- `float`
- `str`
- `bytes`
- `bool` (`True`, `False`), `None`, `...` (Ellipsis)
- `tuple` of constants

The expression can reference previously defined constants. Using runtime values or function calls raises `SyntaxError: not a constant`.

Examples:

```
BUFFER_SIZE = const(1024)
BUFFER_MASK = const(BUFFER_SIZE - 1)
FLAGS = const(0x01 | 0x02)
_SCALE = const(0.001)
_PREFIX = const("data_")
_HEADER = const(b"\x00\xff")
_MODES = const(("read", "write"))
```

Because the compiler evaluates boolean constants at compile time, `const()` can be used for conditional compilation. Code guarded by a false constant is eliminated from the bytecode entirely, and when frozen via `mpy-cross` the unreachable code is stripped from the output:

```
FEATURE_X = const(True)

if FEATURE_X:
    def feature_x_handler():
        ...
```

For cross-platform compatibility with CPython, the typical pattern is:

```
try:
    from micropython import const
except ImportError:
    const = lambda x: x
```

See also *MicroPython on microcontrollers* and *Maximising MicroPython speed* for practical guidance on using constants to reduce memory usage and improve performance.

`micropython.`**`opt_level`**`(`[*level*]`)`

If *level* is given then this function sets the optimisation level for subsequent compilation of scripts, and returns `None`. Otherwise it returns the current optimisation level.

The optimisation level controls the following compilation features:

- Assertions: at level 0 assertion statements are enabled and compiled into the bytecode; at levels 1 and higher assertions are not compiled.
- Built-in `__debug__` variable: at level 0 this variable expands to `True`; at levels 1 and higher it expands to `False`.
- Source-code line numbers: at levels 0, 1 and 2 source-code line number are stored along with the bytecode so that exceptions can report the line number they occurred at; at levels 3 and higher line numbers are not stored.

The default optimisation level is usually level 0.

`micropython.`**`alloc_emergency_exception_buf`**`(`*size*`)`

Allocate *size* bytes of RAM for the emergency exception buffer (a good size is around 100 bytes). The buffer is used to create exceptions in cases when normal RAM allocation would fail (eg within an interrupt handler) and therefore give useful traceback information in these situations.

A good way to use this function is to put it at the start of your main script (eg `boot.py` or `main.py`) and then the emergency exception buffer will be active for all the code following it.

`micropython.`**`mem_info`**`(`[*verbose*]`)`

Print information about currently used memory. If the *verbose* argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the amount of stack and heap used. In verbose mode it prints out a summary of the entire heap indicating which blocks are used and which are free.

The exact output of verbose mode varies between ports, but in general each letter represents a single 16 byte block of memory. Each line of output represents 0x400 bytes or 1KiB of RAM.

The meaning of each letter:

Symbol	Meaning
.	free block
h	head block
=	tail block
m	marked head block
T	tuple
L	list
D	dict
F	float
B	byte code
M	module
S	string or bytes
A	bytearray

micropython.**qstr_info**([*verbose*])

Print information about currently interned strings. If the *verbose* argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the number of interned strings and the amount of RAM they use. In verbose mode it prints out the names of all RAM-interned strings.

micropython.**stack_use**()

Return an integer representing the current amount of stack that is being used. The absolute value of this is not particularly useful, rather it should be used to compute differences in stack usage at different points.

micropython.**heap_lock**()

micropython.**heap_unlock**()

micropython.**heap_locked**()

Lock or unlock the heap. When locked no memory allocation can occur and a *MemoryError* will be raised if any heap allocation is attempted. *heap_locked()* returns a true value if the heap is currently locked.

These functions can be nested, ie *heap_lock()* can be called multiple times in a row and the lock-depth will increase, and then *heap_unlock()* must be called the same number of times to make the heap available again.

Both *heap_unlock()* and *heap_locked()* return the current lock depth (after unlocking for the former) as a non-negative integer, with 0 meaning the heap is not locked.

If the REPL becomes active with the heap locked then it will be forcefully unlocked.

Note: *heap_locked()* is not enabled on most ports by default, requires MICROPY_PY_MICROPYTHON_HEAP_LOCKED.

micropython.**kbd_intr**(*chr*)

Set the character that will raise a *KeyboardInterrupt* exception. By default this is set to 3 during script execution, corresponding to Ctrl-C. Passing -1 to this function will disable capture of Ctrl-C, and passing 3 will restore it.

This function can be used to prevent the capturing of Ctrl-C on the incoming stream of characters that is usually used for the REPL, in case that stream is used for other purposes.

micropython.**schedule**(*func*, *arg*)

Schedule the function *func* to be executed "very soon". The function is passed the value *arg* as its single argument. "Very soon" means that the MicroPython runtime will do its best to execute the function at the earliest possible time, given that it is also trying to be efficient, and that the following conditions hold:

- A scheduled function will never preempt another scheduled function.
- Scheduled functions are always executed "between opcodes" which means that all fundamental Python operations (such as appending to a list) are guaranteed to be atomic.
- A given port may define "critical regions" within which scheduled functions will never be executed. Functions may be scheduled within a critical region but they will not be executed until that region is exited. An example of a critical region is a preempting interrupt handler (an IRQ).
- Inside native code functions, scheduled functions are not called unless the native code calls a function that specifically does so.
- Certain functions including `poll.poll`, `poll.ipoll`, `time.sleep` and `time.sleep_ms` (including zero-duration sleeps) will call scheduled functions.

A use for this function is to schedule a callback from a preempting IRQ. Such an IRQ puts restrictions on the code that runs in the IRQ (for example the heap may be locked) and scheduling a function to call later will lift those restrictions.

On multi-threaded ports, the scheduled function's behaviour depends on whether the Global Interpreter Lock (GIL) is enabled for the specific port:

- If GIL is enabled, the function can preempt any thread and run in its context.
- If GIL is disabled, the function will only preempt the main thread and run in its context.

Note: If *schedule()* is called from a preempting IRQ, when memory allocation is not allowed and the callback to be passed to *schedule()* is a bound method, passing this directly will fail. This is because creating a reference to a bound method causes memory allocation. A solution is to create a reference to the method in the class constructor and to pass that reference to *schedule()*. This is discussed in detail here *reference documentation* under "Creation of Python objects".

There is a finite queue to hold the scheduled functions and *schedule()* will raise a *RuntimeError* if the queue is full.

As a special case, it's possible to pass *micropython.kbd_intr* to this function as the first argument (and `None` as the second argument), and that will schedule a *KeyboardInterrupt* to be raised "very soon" in the main thread.

Classes

class micropython.**RingIO**(*size*)

class micropython.**RingIO**(*buffer*)

Provides a fixed-size ringbuffer for bytes with a stream interface. Can be considered like a fifo queue variant of *io.BytesIO*.

When created with integer size a suitable buffer will be allocated. Alternatively a *bytearray* or similar buffer protocol object can be provided to the constructor for in-place use.

The classic ringbuffer algorithm is used which allows for any size buffer to be used however one byte will be consumed for tracking. If initialised with an integer size this will be accounted for, for example `RingIO(16)` will allocate a 17 byte buffer internally so it can hold 16 bytes of data. When passing in a pre-allocated buffer however one byte less than its original length will be available for storage, eg. `RingIO(bytearray(16))` will only hold 15 bytes of data.

A RingIO instance can be IRQ / thread safe when used to pass data in a single direction eg. when written to in an IRQ and read from in a non-IRQ function (or vice versa). This does not hold if you try to eg. write to a single instance from both IRQ and non-IRQ code, this would often cause data corruption.

any()

Returns an integer counting the number of characters that can be read.

read(*[nbytes]*)

Read available characters. This is a non-blocking function. If `nbytes` is specified then read at most that many bytes, otherwise read as much data as possible.

Return value: a bytes object containing the bytes read. Will be zero-length bytes object if no data is available.

readline(*[nbytes]*)

Read a line, ending in a newline character or return if one exists in the buffer, else return available bytes in buffer. If `nbytes` is specified then read at most that many bytes.

Return value: a bytes object containing the line read.

readinto(*buf[, nbytes]*)

Read available bytes into the provided `buf`. If `nbytes` is specified then read at most that many bytes. Otherwise, read at most `len(buf)` bytes.

Return value: Integer count of the number of bytes read into `buf`.

write(*buf*)

Non-blocking write of bytes from `buf` into the ringbuffer, limited by the available space in the ringbuffer.

Return value: Integer count of bytes written.

close()

No-op provided as part of standard *stream* interface. Has no effect on data in the ringbuffer.

1.2.8 `neopixel` — control of WS2812 / NeoPixel LEDs

This module provides a driver for WS2818 / NeoPixel LEDs.

Note: This module is only included by default on the ESP8266, ESP32 and RP2 ports. On STM32 / Pyboard and others, you can either install the `neopixel` package using *mip*, or you can download the module directly from *micropython-lib* and copy it to the filesystem.

class NeoPixel

This class stores pixel data for a WS2812 LED strip connected to a pin. The application should set pixel data and then call *NeoPixel.write()* when it is ready to update the strip.

For example:

```
import neopixel

# 32 LED strip connected to X8.
p = machine.Pin.board.X8
n = neopixel.NeoPixel(p, 32)
```

(continues on next page)

(continued from previous page)

```
# Draw a red gradient.
for i in range(32):
    n[i] = (i * 8, 0, 0)

# Update the strip.
n.write()
```

Constructors

class neopixel.**NeoPixel**(*pin*, *n*, *, *bpp=3*, *timing=1*)

Construct an NeoPixel object. The parameters are:

- *pin* is a machine.Pin instance.
- *n* is the number of LEDs in the strip.
- *bpp* is 3 for RGB LEDs, and 4 for RGBW LEDs.
- *timing* is 0 for 400KHz, and 1 for 800kHz LEDs (most are 800kHz). You may also supply a timing tuple as accepted by *machine.bitstream()*.

Pixel access methods

NeoPixel.**fill**(*pixel*)

Sets the value of all pixels to the specified *pixel* value (i.e. an RGB/RGBW tuple).

NeoPixel.**__len__**()

Returns the number of LEDs in the strip.

NeoPixel.**__setitem__**(*index*, *val*)

Set the pixel at *index* to the value, which is an RGB/RGBW tuple.

NeoPixel.**__getitem__**(*index*)

Returns the pixel at *index* as an RGB/RGBW tuple.

Output methods

NeoPixel.**write**()

Writes the current pixel data to the strip.

1.2.9 network — network configuration

This module provides network drivers and routing configuration. To use this module, a MicroPython variant/build with network capabilities must be installed. Network drivers for specific hardware are available within this module and are used to configure hardware network interface(s). Network services provided by configured interfaces are then available for use via the *socket* module.

For example:

```
# connect/ show IP config a specific network interface
# see below for examples of specific drivers
import network
import time
nic = network.Driver(...)
if not nic.isconnected():
    nic.connect()
    print("Waiting for connection...")
    while not nic.isconnected():
        time.sleep(1)
print(nic.ipconfig("addr4"))

# now use socket as usual
import socket
addr = socket.getaddrinfo('micropython.org', 80)[0][-1]
s = socket.socket()
s.connect(addr)
s.send(b'GET / HTTP/1.1\r\nHost: micropython.org\r\n\r\n')
data = s.recv(1000)
s.close()
```

Common network adapter interface

This section describes an (implied) abstract base class for all network interface classes implemented by *MicroPython ports* for different hardware. This means that MicroPython does not actually provide `AbstractNIC` class, but any actual NIC class, as described in the following sections, implements methods as described here.

class network.**AbstractNIC**(*id=None, ...*)

Instantiate a network interface object. Parameters are network interface dependent. If there are more than one interface of the same type, the first parameter should be `id`.

AbstractNIC.**active**([*is_active*])

Activate ("up") or deactivate ("down") the network interface, if a boolean argument is passed. Otherwise, query current state if no argument is provided. Most other methods require an active interface (behaviour of calling them on inactive interface is undefined).

AbstractNIC.**connect**([*service_id*, *key=None*, *, ...])

Connect the interface to a network. This method is optional, and available only for interfaces which are not "always connected". If no parameters are given, connect to the default (or the only) service. If a single parameter is given, it is the primary identifier of a service to connect to. It may be accompanied by a key (password) required to access said service. There can be further arbitrary keyword-only parameters, depending on the networking medium type and/or particular device. Parameters can be used to: a) specify alternative service identifier types; b) provide additional connection parameters. For various medium types, there are different sets of predefined/recommended parameters, among them:

- WiFi: *bssid* keyword to connect to a specific BSSID (MAC address)

AbstractNIC.**disconnect**()

Disconnect from network.

AbstractNIC.**isconnected**()

Returns `True` if connected to network, otherwise returns `False`.

AbstractNIC.**scan**(*, ...)

Scan for the available network services/connections. Returns a list of tuples with discovered service parameters. For various network media, there are different variants of predefined/ recommended tuple formats, among them:

- WiFi: (ssid, bssid, channel, RSSI, security, hidden). There may be further fields, specific to a particular device.

The function may accept additional keyword arguments to filter scan results (e.g. scan for a particular service, on a particular channel, for services of a particular set, etc.), and to affect scan duration and other parameters. Where possible, parameter names should match those in connect().

`AbstractNIC.`**`status`**`(`[*param*]`)`

Query dynamic status information of the interface. When called with no argument the return value describes the network link status. Otherwise *param* should be a string naming the particular status parameter to retrieve.

The return types and values are dependent on the network medium/technology. Some of the parameters that may be supported are:

- WiFi STA: use `'rssi'` to retrieve the RSSI of the AP signal
- WiFi AP: use `'stations'` to retrieve a list of all the STAs connected to the AP. The list contains tuples of the form (MAC, RSSI).

`AbstractNIC.`**`ipconfig`**`(`*'param'*`)`

`AbstractNIC.`**`ipconfig`**`(`*param*=*value*, ...`)`

Get or set interface-specific IP-configuration interface parameters. Supported parameters are the following (availability of a particular parameter depends on the port and the specific network interface):

- `dhcp4` (`True/False`) obtain an IPv4 address, gateway and dns server via DHCP. This method does not block and wait for an address to be obtained. To check if an address was obtained, use the read-only property `has_dhcp4`.
- `gw4` Get/set the IPv4 default-gateway.
- `dhcp6` (`True/False`) obtain a DNS server via stateless DHCPv6. Obtaining IP Addresses via DHCPv6 is currently not implemented.
- `autoconf6` (`True/False`) obtain a stateless IPv6 address via the network prefix shared in router advertisements. To check if a stateless address was obtained, use the read-only property `has_autoconf6`.
- `addr4` (e.g. `192.168.0.4/24`) obtain the current IPv4 address and network mask as `(ip, subnet)`-tuple, regardless of how this address was obtained. This method can be used to set a static IPv4 address either as `(ip, subnet)`-tuple or in CIDR-notation.
- `addr6` (e.g. `fe80::1234:5678`) obtain a list of current IPv6 addresses as `(ip, state, preferred_lifetime, valid_lifetime)`-tuple. This include link-local, slaac and static addresses. `preferred_lifetime` and `valid_lifetime` represent the remaining valid and preferred lifetime of each IPv6 address, in seconds. `state` indicates the current state of the address:
  - `0x08` - `0x0f` indicates the address is tentative, counting the number of probes sent.
  - `0x10` The address is deprecated (but still valid)
  - `0x30` The address is preferred (and valid)
  - `0x40` The address is duplicated and can not be used.

  This method can be used to set a static IPv6 address, by setting this parameter to the address, like `fe80::1234:5678`.

`AbstractNIC.`**`ifconfig`**`(`[*(ip, subnet, gateway, dns)*]`)`

Note: This function is deprecated, use *`ipconfig()`* instead.

Get/set IP-level network interface parameters: IP address, subnet mask, gateway and DNS server. When called with no arguments, this method returns a 4-tuple with the above information. To set the above values, pass a 4-tuple with the required information. For example:

```
nic.ifconfig(('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
```

AbstractNIC.**config**(*'param'*)

AbstractNIC.**config**(*param=value, ...*)

Get or set general network interface parameters. These methods allow to work with additional parameters beyond standard IP configuration (as dealt with by *ipconfig()*). These include network-specific and hardware-specific parameters. For setting parameters, the keyword argument syntax should be used, and multiple parameters can be set at once. For querying, a parameter name should be quoted as a string, and only one parameter can be queried at a time:

```
# Set WiFi access point name (formally known as SSID) and WiFi channel
ap.config(ssid='My AP', channel=11)
# Query params one by one
print(ap.config('ssid'))
print(ap.config('channel'))
```

Specific network class implementations

The following concrete classes implement the AbstractNIC interface and provide a way to control networking interfaces of various kinds.

class WLAN – control built-in WiFi interfaces

This class provides a driver for WiFi network processors. Example usage:

```
import network
# enable station interface and connect to WiFi access point
nic = network.WLAN(network.WLAN.IF_STA)
nic.active(True)
nic.connect('your-ssid', 'your-key')
# now use sockets as usual
```

Constructors

class network.**WLAN**(*interface_id*)

Create a WLAN network interface object. Supported interfaces are `network.WLAN.IF_STA` (station aka client, connects to upstream WiFi access points) and `network.WLAN.IF_AP` (access point, allows other WiFi clients to connect). Availability of the methods below depends on interface type. For example, only STA interface may *WLAN.connect()* to an access point.

Methods

WLAN.**active**([*is_active*])

Activate ("up") or deactivate ("down") network interface, if boolean argument is passed. Otherwise, query current state if no argument is provided. Most other methods require active interface.

WLAN.**connect**(*ssid=None*, *key=None*, ***, *bssid=None*)

Connect to the specified wireless network, using the specified key. If *bssid* is given then the connection will be restricted to the access-point with that MAC address (the *ssid* must also be specified in this case).

WLAN.**disconnect**()

Disconnect from the currently connected wireless network.

WLAN.**scan**()

Scan for the available wireless networks. Hidden networks – where the SSID is not broadcast – will also be scanned if the WLAN interface allows it.

Scanning is only possible on STA interface. Returns list of tuples with the information about WiFi access points:

(ssid, bssid, channel, RSSI, security, hidden)

bssid is hardware address of an access point, in binary form, returned as bytes object. You can use *binascii.hexlify()* to convert it to ASCII form.

There are five values for security:

- 0 – open
- 1 – WEP
- 2 – WPA-PSK
- 3 – WPA2-PSK
- 4 – WPA/WPA2-PSK

and two for hidden:

- 0 – visible
- 1 – hidden

WLAN.**status**([*param*])

Return the current status of the wireless connection.

When called with no argument the return value describes the network link status. The possible statuses are defined as constants in the *network* module:

- STAT_IDLE – no connection and no activity,
- STAT_CONNECTING – connecting in progress,
- STAT_WRONG_PASSWORD – failed due to incorrect password,
- STAT_NO_AP_FOUND – failed because no access point replied,
- STAT_CONNECT_FAIL – failed due to other problems,
- STAT_GOT_IP – connection successful.

When called with one argument *param* should be a string naming the status parameter to retrieve, and different parameters are supported depending on the mode the WiFi is in.

In STA mode, passing `'rssi'` returns a signal strength indicator value, whose format varies depending on the port (this is available on all ports that support WiFi network interfaces, except for CC3200).

In AP mode, passing `'stations'` returns a list of connected WiFi stations (this is available on all ports that support WiFi network interfaces, except for CC3200). The format of the station information entries varies across ports, providing either the raw BSSID of the connected station, the IP address of the connected station, or both.

`WLAN.`**`isconnected`**`()`

In case of STA mode, returns `True` if connected to a WiFi access point and has a valid IP address. In AP mode returns `True` when a station is connected. Returns `False` otherwise.

`WLAN.`**`ifconfig`**`(`[*(ip, subnet, gateway, dns)*]`)`

Get/set IP-level network interface parameters: IP address, subnet mask, gateway and DNS server. When called with no arguments, this method returns a 4-tuple with the above information. To set the above values, pass a 4-tuple with the required information. For example:

```
nic.ifconfig(('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
```

`WLAN.`**`config`**`(`*'param'*`)`

`WLAN.`**`config`**`(`*param=value, ...*`)`

Get or set general network interface parameters. These methods allow to work with additional parameters beyond standard IP configuration (as dealt with by *`AbstractNIC.ipconfig()`*). These include network-specific and hardware-specific parameters. For setting parameters, keyword argument syntax should be used, multiple parameters can be set at once. For querying, parameters name should be quoted as a string, and only one parameter can be queries at time:

```
# Set WiFi access point name (formally known as SSID) and WiFi channel
ap.config(ssid='My AP', channel=11)
# Query params one by one
print(ap.config('ssid'))
print(ap.config('channel'))
```

Following are commonly supported parameters (availability of a specific parameter depends on network technology type, driver, and *MicroPython port*).

Parameter	Description
mac	MAC address (bytes)
ssid	WiFi access point name (string)
channel	WiFi channel (integer). Depending on the port this may only be supported on the AP interface.
hidden	Whether SSID is hidden (boolean)
security	Security protocol supported (enumeration, see module constants)
key	Access key (string)
hostname	The hostname that will be sent to DHCP (STA interfaces) and mDNS (if supported, both STA and AP). (Deprecated, use *`network.hostname()`* instead)
reconnects	Number of reconnect attempts to make (integer, 0=none, -1=unlimited)
txpower	Maximum transmit power in dBm (integer or float)
pm	WiFi Power Management setting (see below for allowed values)
protocol	(ESP32 Only.) WiFi Low level 802.11 protocol. See *`WLAN.PROTOCOL_DEFAULT`*.
bandwidth	(ESP32 Only.) WiFi channel bandwidth. See *`WLAN.BANDWIDTH_20`* and others.

CSI Methods (ESP32 only)

Note: These methods are only available on ESP32 builds with CSI support enabled. The standard generic ESP32, ESP32-C3, ESP32-C5, ESP32-C6, and ESP32-S3 board definitions enable this in their default configuration. Other builds need `CONFIG_ESP_WIFI_CSI_ENABLED=y` in the ESP-IDF configuration.

Channel State Information (CSI) provides per-packet physical layer channel data derived from received Wi-Fi frames. CSI capture requires an active Wi-Fi connection and incoming traffic to the device. Without traffic, no CSI frames will be captured.

Other Espressif CSI options are hard-coded to defaults intended for connected station capture.

`WLAN.`**`csi_enable`**`(`*buffer_size=16*`)`

Enable CSI capture and allocate a circular buffer for received frames.

The optional `buffer_size` argument sets the number of frames stored before new incoming frames are dropped. Larger values reduce drops at the cost of RAM. The exact maximum depends on the build, but it is limited by the underlying ringbuffer implementation to roughly 100 frames.

Raises `OSError` if CSI cannot be enabled, for example if Wi-Fi is not active or the ESP-IDF rejects the configuration.

Example:

```
import network
import time

wlan = network.WLAN(network.WLAN.IF_STA)
wlan.active(True)
wlan.config(protocol=network.MODE_11B | network.MODE_11G | network.MODE_11N)
wlan.config(pm=wlan.PM_NONE)
wlan.connect("SSID", "password")

while not wlan.isconnected():
    time.sleep_ms(100)

wlan.csi_enable(buffer_size=32)
```

`WLAN.`**`csi_disable`**`()`

Disable CSI capture and clean up resources.

`WLAN.`**`csi_read`**`(`[*result*]`)`

Read a CSI frame from the buffer.

Returns: A list containing CSI frame data, or `None` if no frames are available.

If the optional `result` argument is provided, it must be a previous list returned by *`WLAN.csi_read()`*. The list will be updated in place and returned again. This reduces heap churn in busy read loops by reusing the existing list object and, when the captured frame fits, the existing CSI data `bytearray`.

Frame list fields (in order):

- **0 - rssi** (int): Received signal strength in dBm
- **1 - channel** (int): Wi-Fi channel number
- **2 - mac** (bytes): Source MAC address (6 bytes)
- **3 - timestamp** (int): Timestamp in microseconds

- **4 - local_timestamp** (int): Local timestamp from Wi-Fi hardware
- **5 - data** (bytearray): CSI raw data (I/Q components as int8_t values)
- **6 - rate** (int): Data rate
- **7 - sig_mode** (int): Signal mode (legacy, HT, VHT)
- **8 - mcs** (int): Modulation and Coding Scheme index
- **9 - cwb** (int): Channel bandwidth
- **10 - smoothing** (int): Smoothing applied
- **11 - not_sounding** (int): Not sounding frame
- **12 - aggregation** (int): Aggregation
- **13 - stbc** (int): STBC
- **14 - fec_coding** (int): FEC coding
- **15 - sgi** (int): Short GI
- **16 - noise_floor** (int): Background noise level in dBm
- **17 - ampdu_cnt** (int): AMPDU count
- **18 - secondary_channel** (int): Secondary channel
- **19 - ant** (int): Antenna
- **20 - sig_len** (int): Signal length
- **21 - rx_state** (int): RX state

Some metadata fields may be `0` on targets where ESP-IDF does not provide the corresponding value in the public CSI receive structure.

`WLAN.`**`csi_available`**`()`

Get the number of CSI frames available in the buffer.

`WLAN.`**`csi_dropped`**`()`

Get the number of CSI frames dropped due to buffer overflow. Frames are dropped when the buffer is full and new frames arrive faster than they can be read. Increase `buffer_size` in `csi_enable()` to reduce drops.

Constants

`WLAN.`**`PM_PERFORMANCE`**

`WLAN.`**`PM_POWERSAVE`**

`WLAN.`**`PM_NONE`**

Allowed values for the `WLAN.config(pm=...)` network interface parameter:

- `PM_PERFORMANCE`: enable WiFi power management to balance power savings and WiFi performance
- `PM_POWERSAVE`: enable WiFi power management with additional power savings and reduced WiFi performance
- `PM_NONE`: disable wifi power management

ESP32 Protocol Constants

The following ESP32-only constants relate to the `WLAN.config(protocol=...)` network interface parameter:

`WLAN.`**`PROTOCOL_DEFAULT`**

A bitmap representing all of the default 802.11 Wi-Fi modes supported by the chip. Consult ESP-IDF Wi-Fi Protocols documentation for details.

`WLAN.`**`PROTOCOL_LR`**

This value corresponds to the Espressif proprietary "long-range" mode, which is not compatible with standard Wi-Fi devices. By setting this protocol it's possible for an ESP32 STA in long-range mode to connect to an ESP32 AP in long-range mode, or to use *ESP-NOW long range modes*.

This mode can be bitwise ORed with some standard 802.11 protocol bits (including *`WLAN.PROTOCOL_DEFAULT`*) in order to support a mix of standard Wi-Fi modes as well as LR mode, consult the Espressif long-range documentation for more details.

Long range mode is not supported on ESP32-C2.

`WLAN.`**`BANDWIDTH_20`**

`WLAN.`**`BANDWIDTH_40`**

`WLAN.`**`BANDWIDTH_80`**

`WLAN.`**`BANDWIDTH_160`**

`WLAN.`**`BANDWIDTH_80_80`**

Allowed values for the `WLAN.config(bandwidth=...)` network interface parameter:

- `BANDWIDTH_20`: specifies a 20MHz wide WiFi channel when in STA and AP mode
- `BANDWIDTH_40`: specifies a 40MHz wide WiFi channel when in STA and AP mode
- `BANDWIDTH_80`: specifies a 80MHz wide WiFi channel when in AP mode, may not be available on all ESP32 models
- `BANDWIDTH_160`: specifies a 160MHz wide WiFi channel when in AP mode, may not be available on all ESP32 models
- `BANDWIDTH_80_80`: specifies a multi-antenna 80MHz + 80MHz wide WiFi channel setup when in AP mode, may not be available on all ESP32 models.

When in STA mode, bandwidth can only be changed when the adapter is not connected to a network. In AP mode it can be changed at any time.

class WLANWiPy – WiPy specific WiFi control

Note: This class is a non-standard WLAN implementation for the WiPy. It is available simply as `network.WLAN` on the WiPy but is named in the documentation below as `network.WLANWiPy` to distinguish it from the more general *network.WLAN* class.

This class provides a driver for the WiFi network processor in the WiPy. Example usage:

```
import network
import time
# setup as a station
wlan = network.WLAN(mode=WLAN.STA)
wlan.connect('your-ssid', auth=(WLAN.WPA2, 'your-key'))
```

(continues on next page)

(continued from previous page)

```
while not wlan.isconnected():
    time.sleep_ms(50)
print(wlan.ipconfig("addr4"))

# now use socket as usual
...
```

Constructors

class network.**WLANWiPy**(*id=0*, ...)

Create a WLAN object, and optionally configure it. See *init()* for params of configuration.

Note: The WLAN constructor is special in the sense that if no arguments besides the id are given, it will return the already existing WLAN instance without re-configuring it. This is because WLAN is a system feature of the WiPy. If the already existing instance is not initialized it will do the same as the other constructors an will initialize it with default values.

Methods

WLANWiPy.**init**(*mode*, *, *ssid*, *auth*, *channel*, *antenna*)

Set or get the WiFi network processor configuration.

Arguments are:

- *mode* can be either WLAN.STA or WLAN.AP.
- *ssid* is a string with the ssid name. Only needed when mode is WLAN.AP.
- *auth* is a tuple with (sec, key). Security can be None, WLAN.WEP, WLAN.WPA or WLAN.WPA2. The key is a string with the network password. If sec is WLAN.WEP the key must be a string representing hexadecimal values (e.g. 'ABC1DE45BF'). Only needed when mode is WLAN.AP.
- *channel* a number in the range 1-11. Only needed when mode is WLAN.AP.
- *antenna* selects between the internal and the external antenna. Can be either WLAN.INT_ANT or WLAN.EXT_ANT.

For example, you can do:

```
# create and configure as an access point
wlan.init(mode=WLAN.AP, ssid='wipy-wlan', auth=(WLAN.WPA2,'www.wipy.io'),␣
↪channel=7, antenna=WLAN.INT_ANT)
```

or:

```
# configure as an station
wlan.init(mode=WLAN.STA)
```

WLANWiPy.**connect**(*ssid*, *, *auth=None*, *bssid=None*, *timeout=None*)

Connect to a WiFi access point using the given SSID, and other security parameters.

- *auth* is a tuple with (sec, key). Security can be None, WLAN.WEP, WLAN.WPA or WLAN.WPA2. The key is a string with the network password. If sec is WLAN.WEP the key must be a string representing hexadecimal values (e.g. 'ABC1DE45BF').

- *bssid* is the MAC address of the AP to connect to. Useful when there are several APs with the same ssid.
- *timeout* is the maximum time in milliseconds to wait for the connection to succeed.

`WLANWiPy.`**`scan`**`()`

Performs a network scan and returns a list of named tuples with (ssid, bssid, sec, channel, rssi). Note that channel is always `None` since this info is not provided by the WiPy.

`WLANWiPy.`**`disconnect`**`()`

Disconnect from the WiFi access point.

`WLANWiPy.`**`isconnected`**`()`

In case of STA mode, returns `True` if connected to a WiFi access point and has a valid IP address. In AP mode returns `True` when a station is connected, `False` otherwise.

`WLANWiPy.`**`ipconfig`**`(`*'param'*`)`

`WLANWiPy.`**`ipconfig`**`(`*param=value, ...*`)`

See *`AbstractNIC.ipconfig`*. Supported parameters are: `dhcp4`, `addr4`, `gw4`.

`WLANWiPy.`**`mode`**`(`[*mode*]`)`

Get or set the WLAN mode.

`WLANWiPy.`**`ssid`**`(`[*ssid*]`)`

Get or set the SSID when in AP mode.

`WLANWiPy.`**`auth`**`(`[*auth*]`)`

Get or set the authentication type when in AP mode.

`WLANWiPy.`**`channel`**`(`[*channel*]`)`

Get or set the channel (only applicable in AP mode).

`WLANWiPy.`**`antenna`**`(`[*antenna*]`)`

Get or set the antenna type (external or internal).

`WLANWiPy.`**`mac`**`(`[*mac_addr*]`)`

Get or set a 6-byte long bytes object with the MAC address.

`WLANWiPy.`**`irq`**`(*, `*handler, wake*`)`

Create a callback to be triggered when a WLAN event occurs during `machine.SLEEP` mode. Events are triggered by socket activity or by WLAN connection/disconnection.

- *handler* is the function that gets called when the IRQ is triggered.
- *wake* must be `machine.SLEEP`.

Returns an IRQ object.

Constants

`WLANWiPy.`**`STA`**

`WLANWiPy.`**`AP`**

selects the WLAN mode

`WLANWiPy.`**`WEP`**

`WLANWiPy.`**`WPA`**

WLANWiPy.**WPA2**

selects the network security

WLANWiPy.**INT_ANT**

WLANWiPy.**EXT_ANT**

selects the antenna type

class WIZNET5K – control WIZnet5x00 Ethernet modules

This class allows you to control WIZnet5x00 Ethernet adaptors based on the W5200 and W5500 chipsets. The particular chipset that is supported by the firmware is selected at compile-time via the MICROPY_PY_NETWORK_WIZNET5K option.

Note: The esp32 port also supports WIZnet W5500 chipsets, but this port uses the *network.LAN interface*.

Example usage:

```
import network
nic = network.WIZNET5K(pyb.SPI(1), pyb.Pin.board.X5, pyb.Pin.board.X4)
print(nic.ipconfig("addr4"))

# now use socket as usual
...
```

For this example to work the WIZnet5x00 module must have the following connections:

- MOSI connected to X8
- MISO connected to X7
- SCLK connected to X6
- nSS connected to X5
- nRESET connected to X4

It is possible to use other SPI buses and other pins for nSS and nRESET.

Constructors

class network.**WIZNET5K**(*spi*, *pin_cs*, *pin_rst*)

Create a WIZNET5K driver object, initialise the WIZnet5x00 module using the given SPI bus and pins, and return the WIZNET5K object.

Arguments are:

- *spi* is an *SPI object* which is the SPI bus that the WIZnet5x00 is connected to (the MOSI, MISO and SCLK pins).
- *pin_cs* is a *Pin object* which is connected to the WIZnet5x00 nSS pin.
- *pin_rst* is a *Pin object* which is connected to the WIZnet5x00 nRESET pin.

All of these objects will be initialised by the driver, so there is no need to initialise them yourself. For example, you can use:

```
nic = network.WIZNET5K(pyb.SPI(1), pyb.Pin.board.X5, pyb.Pin.board.X4)
```

Methods

This class implements most methods from *AbstractNIC*, which are documented there. Additional methods are:

WIZNET5K.**regs**()

Dump the WIZnet5x00 registers. Useful for debugging.

class LAN – control an Ethernet module

This class allows you to control the Ethernet interface. The PHY hardware type is board-specific.

Example usage, for a board with built-in LAN support:

```
import network
nic = network.LAN(0)
print(nic.ipconfig("addr4"))

# now use socket as usual
...
```

Constructors

class network.**LAN**(*id*, *, *phy_type=<board_default>*, *phy_addr=<board_default>*, *ref_clk_mode=<board_default>*)

Create a LAN driver object, initialise the LAN module using the given PHY driver name, and return the LAN object.

Arguments are:

- *id* is the number of the Ethernet port, either 0 or 1.
- *phy_type* is the name of the PHY driver. For most board the on-board PHY has to be used and is the default. Suitable values are port specific.
- *phy_addr* specifies the address of the PHY interface. As with *phy_type*, the hardwired value has to be used for most boards and that value is the default.
- *ref_clk_mode* specifies, whether the data clock is provided by the Ethernet controller or the PHY interface. The default value is the one that matches the board. If set to `LAN.OUT` or `Pin.OUT` or `True`, the clock is driven by the Ethernet controller, if set to `LAN.IN` or `Pin.IN` or `False`, the clock is driven by the PHY interface.

For example, with the Seeed Arch Mix board you can use:

```
nic = LAN(0, phy_type=LAN.PHY_LAN8720, phy_addr=1, ref_clk_mode=Pin.IN)
```

Note: On esp32 port the constructor requires different arguments. See *esp32 port reference*.

Methods

LAN.**active**([*state*])

With a parameter, it sets the interface active if *state* is true, otherwise it sets it inactive. Without a parameter, it returns the state.

LAN.**isconnected**()

Returns `True` if the physical Ethernet link is connected and up. Returns `False` otherwise.

LAN.**status**()

Returns the LAN status.

LAN.**ifconfig**([*(ip, subnet, gateway, dns)*])

Get/set IP address, subnet mask, gateway and DNS.

When called with no arguments, this method returns a 4-tuple with the above information.

To set the above values, pass a 4-tuple with the required information. For example:

```
nic.ifconfig(('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
```

LAN.**config**(*config_parameters*)

Sets or gets parameters of the LAN interface. The only parameter that can be retrieved is the MAC address, using:

```
mac = LAN.config("mac")
```

The parameters that can be set are:

- `trace=n` sets trace levels; suitable values are:
  - 2: trace TX
  - 4: trace RX
  - 8: full trace
- `low_power=bool` sets or clears low power mode, valid values being `False` or `True`.

Specific LAN class implementations

On the mimxrt port, suitable values for the *phy_type* constructor argument are: `PHY_KSZ8081`, `PHY_DP83825`, `PHY_DP83848`, `PHY_LAN8720`, `PHY_RTL8211F`.

class PPP – create network connections over serial PPP

This class allows you to create a network connection over a serial port using the PPP protocol.

Note: Currently only the esp32 port has PPP support enabled in the default firmware build. PPP support can be enabled in custom builds of the stm32 and rp2 ports by enabling networking support and setting `MICROPY_PY_NETWORK_PPP_LWIP` to 1.

Example usage:

```
import network

ppp = network.PPP(uart)
ppp.connect()

while not ppp.isconnected():
    pass

print(ppp.ipconfig("addr4"))

# use the socket module as usual, etc

ppp.disconnect()
```

Constructors

class network.**PPP**(*stream*)

Create a PPP driver object.

Arguments are:

- *stream* is any object that supports the stream protocol, but is most commonly a *machine.UART* instance. This stream object must have an `irq()` method and an `IRQ_RXIDLE` constant, for use by *PPP.connect*.

Methods

PPP.**connect**(*security=SEC_NONE*, *user=None*, *key=None*)

Initiate a PPP connection with the given parameters:

- *security* is the type of security, either `PPP.SEC_NONE`, `PPP.SEC_PAP`, or `PPP.SEC_CHAP`.
- *user* is an optional user name to use with the security mode.
- *key* is an optional password to use with the security mode.

When this method is called the underlying stream has its interrupt configured to call *PPP.poll* via `stream.irq(ppp.poll, stream.IRQ_RXIDLE)`. This makes sure the stream is polled, and data passed up the PPP stack, wheverver data becomes available on the stream.

The connection proceeds asynchronously, in the background.

PPP.**disconnect**()

Terminate the connection. This must be called to cleanly close the PPP connection.

PPP.**isconnected**()

Returns `True` if the PPP link is connected and up. Returns `False` otherwise.

PPP.**status**()

Returns the PPP status.

PPP.**config**(*config_parameters*)

Sets or gets parameters of the PPP interface. The only parameter that can be retrieved and set is the underlying stream, using:

```
stream = PPP.config("stream")
PPP.config(stream=stream)
```

PPP.**ipconfig**(*'param'*)

PPP.**ipconfig**(*param=value, ...*)

See *AbstractNIC.ipconfig*.

PPP.**ifconfig**([*(ip, subnet, gateway, dns)*])

See *AbstractNIC.ifconfig*.

PPP.**poll**([*irq_arg*])

Poll the underlying stream for data, and pass it up the PPP stack. This is called automatically if the stream is a UART with a RXIDLE interrupt, so it's not usually necessary to call it manually.

The optional *irq_arg* argument is ignored, this argument exists only so this function is compatible with the *machine.UART.irq() handler* argument.

Constants

PPP.**SEC_NONE**

PPP.**SEC_PAP**

PPP.**SEC_CHAP**

The type of connection security.

class USBD_NCM – USB NCM network interface

This class provides a network interface over USB using the NCM (Network Control Model) protocol. The host computer sees this device as a USB Ethernet adapter and assigns it an IP address via DHCP (served by the MicroPython device).

Note: `network.USBD_NCM` requires a port with TinyUSB and NCM support, enabled at build time by defining `MICROPY_PY_NETWORK_USBD_NCM` (off by default).

Example usage:

```
import network

nic = network.USBD_NCM()
nic.active(True)
# wait for USB host to configure the NCM interface
while not nic.isconnected():
    pass

print(nic.ipconfig("addr4"))
```

Constructors

class network.**USBD_NCM**

Create and return a USBD_NCM object. This initialises the NCM network interface if it has not already been initialised. Only one instance exists (singleton).

Methods

USBD_NCM.**active**([*is_active*])

Activate or deactivate the network interface. Without argument returns current state as a bool.

The interface is brought up automatically before USB enumeration, so this returns `True` from boot.

USBD_NCM.**isconnected**()

Returns `True` if the USB host has configured the NCM interface, `False` otherwise.

When USB is disconnected, this returns `False` and network traffic stops. The interface remains registered with lwIP and can resume when the host reconnects and re-enumerates the device.

USBD_NCM.**status**()

Returns the link status as an integer: `1` if the interface is up, `0` otherwise.

USBD_NCM.**ipconfig**(*'param'*)

USBD_NCM.**ipconfig**(*param=value, ...*)

See *AbstractNIC.ipconfig*.

USBD_NCM.**ifconfig**([*(ip, subnet, gateway, dns)*])

See *AbstractNIC.ifconfig*.

Notes

Link-local IP address: The device IP (169.254.x.1) is derived deterministically from the device MAC address. RFC 3927 ARP probe/announce (conflict detection) is not implemented, so if two devices happen to derive the same address on the same network segment, the conflict will go undetected.

MAC address uniqueness: The device and host-side MAC addresses are derived from the value returned by `mp_hal_get_mac()`. If two boards have the same hardware MAC (e.g. the port does not use a hardware UID), they will present the same network addresses and cause ARP conflicts.

Network functions

The following are functions available in the network module.

network.**country**([*code*])

Get or set the two-letter ISO 3166-1 Alpha-2 country code to be used for radio compliance.

If the *code* parameter is provided, the country will be set to this value. If the function is called without parameters, it returns the current country.

The default code `"XX"` represents the "worldwide" region.

network.**hostname**([*name*])

Get or set the hostname that will identify this device on the network. It will be used by all interfaces.

This hostname is used for:

- Sending to the DHCP server in the client request. (If using DHCP)
- Broadcasting via mDNS. (If enabled)

If the *name* parameter is provided, the hostname will be set to this value. If the function is called without parameters, it returns the current hostname.

A change in hostname is typically only applied during connection. For DHCP this is because the hostname is part of the DHCP client request, and the implementation of mDNS in most ports only initialises the hostname once during connection. For this reason, you must set the hostname before activating/connecting your network interfaces.

The length of the hostname is limited to 32 characters. *MicroPython ports* may choose to set a lower limit for memory reasons. If the given name does not fit, a `ValueError` is raised.

The default hostname is typically the name of the board.

`network.`**`ipconfig`**`('param')`

`network.`**`ipconfig`**`(param=value, ...)`

Get or set global IP-configuration parameters. Supported parameters are the following (availability of a particular parameter depends on the port and the specific network interface):

- `dns` Get/set DNS server. This method can support both, IPv4 and IPv6 addresses.
- `prefer(4/6)` Specify which address type to return, if a domain name has both A and AAAA records. Note, that this does not clear the local DNS cache, so that any previously obtained addresses might not change.

`network.`**`phy_mode`**`([mode])`

Get or set the PHY mode.

If the *mode* parameter is provided, the PHY mode will be set to this value. If the function is called without parameters, it returns the current PHY mode.

The possible modes are defined as constants:

- `MODE_11B` – IEEE 802.11b,
- `MODE_11G` – IEEE 802.11g,
- `MODE_11N` – IEEE 802.11n.

Availability: ESP8266.

1.2.10 `openamp` – provides standard Asymmetric Multiprocessing (AMP) support

The `openamp` module provides a standard inter-processor communications infrastructure for MicroPython. The module handles all of the details of OpenAMP, such as setting up the shared resource table, initializing vrings, etc. It provides an API for using the RPMsg bus infrastructure with the `Endpoint` class, and provides processor Life Cycle Management (LCM) support, such as loading firmware and starting and stopping a remote core, via the `RemoteProc` class.

Example usage:

```
import openamp

def ept_recv_callback(src, data):
    print("Received message on endpoint", data)

# Create a new RPMsg endpoint to communicate with the remote core.
ept = openamp.Endpoint("vuart-channel", callback=ept_recv_callback)

# Create a RemoteProc object, load its firmware and start it.
rproc = openamp.RemoteProc("virtual_uart.elf") # Or entry point address (ex␣
↪0x081E0000)
rproc.start()

while True:
    if ept.is_ready():
        ept.send("data")
```

Functions

openamp.**new_service_callback**(*ns_callback*)

Set the new service callback.

The *ns_callback* argument is a function that will be called when the remote processor announces new services. At that point the host processor can choose to create the announced endpoint, if this particular service is supported, or ignore it if it's not. If this function is not set, the host processor should first register the endpoint locally, and it will be automatically bound when the remote announces the service.

Endpoint class

class openamp.**Endpoint**(*name*, *callback*, *src=ENDPOINT_ADDR_ANY*, *dest=ENDPOINT_ADDR_ANY*)

Construct a new RPMsg Endpoint. An endpoint is a bidirectional communication channel between two cores.

Arguments are:

- *name* is the name of the endpoint.
- *callback* is a function that is called when the endpoint receives data with the source address of the remote point, and the data as bytes passed by reference.
- *src* is the endpoint source address. If none is provided one will be assigned to the endpoint by the library.
- *dest* is the endpoint destination address. If the endpoint is created from the new_service_callback, this must be provided and it must match the remote endpoint's source address. If the endpoint is registered locally, before the announcement, the destination address will be assigned by the library when the endpoint is bound.

Endpoint.**deinit**()

Destroy the endpoint and release all of its resources.

Endpoint.**is_ready**()

Returns True if the endpoint is ready to send (i.e., has both a source and destination addresses)

Endpoint.**send**(*src=-1*, *dest=-1*, *timeout=-1*)

Send a message to the remote processor over this endpoint.

Arguments are:

- *src* is the source endpoint address of the message. If none is provided, the source address the endpoint is bound to is used.
- *dest* is the destination endpoint address of the message. If none is provided, the destination address the endpoint is bound to is used.
- *timeout* specifies the time in milliseconds to wait for a free buffer. By default the function is blocking.

RemoteProc class

class openamp.**RemoteProc**(*entry*)

The RemoteProc object provides processor Life Cycle Management (LCM) support, such as loading firmware, starting and stopping a remote core.

The *entry* argument can be a path to firmware image, in which case the firmware is loaded from file to its target memory, or an entry point address, in which case the firmware must be loaded already at the given address.

RemoteProc.**start**()

Starts the remote processor.

RemoteProc.**stop**()

Stops the remote processor. The exact behavior is platform-dependent. On the STM32H7 for example it's not possible to stop and then restart the Cortex-M4 core, so a complete system reset is performed on a call to this function.

RemoteProc.**shutdown**()

Shutdown stops the remote processor and releases all of its resources. The exact behavior is platform-dependent, however typically it disables power and clocks to the remote core. This function is also used as the finaliser (i.e., called when RemoteProc object is collected). Note that on the STM32H7, it's not possible to stop and then restart the Cortex-M4 core, so a complete system reset is performed on a call to this function.

1.2.11 uctypes – access binary data in a structured way

This module implements "foreign data interface" for MicroPython. The idea behind it is similar to CPython's ctypes modules, but the actual API is different, streamlined and optimized for small size. The basic idea of the module is to define data structure layout with about the same power as the C language allows, and then access it using familiar dot-syntax to reference sub-fields.

Warning: uctypes module allows access to arbitrary memory addresses of the machine (including I/O and control registers). Uncareful usage of it may lead to crashes, data loss, and even hardware malfunction.

See also:

Module *struct*
Standard Python way to access binary data structures (doesn't scale well to large and complex structures).

Usage examples:

```
import uctypes

# Example 1: Subset of ELF file header
# https://wikipedia.org/wiki/Executable_and_Linkable_Format#File_header
ELF_HEADER = {
    "EI_MAG": (0x0 | uctypes.ARRAY, 4 | uctypes.UINT8),
    "EI_DATA": 0x5 | uctypes.UINT8,
    "e_machine": 0x12 | uctypes.UINT16,
}

# "f" is an ELF file opened in binary mode
buf = f.read(uctypes.sizeof(ELF_HEADER, uctypes.LITTLE_ENDIAN))
header = uctypes.struct(uctypes.addressof(buf), ELF_HEADER, uctypes.LITTLE_ENDIAN)
assert header.EI_MAG == b"\x7fELF"
assert header.EI_DATA == 1, "Oops, wrong endianness. Could retry with uctypes.BIG_
↪ENDIAN."
print("machine:", hex(header.e_machine))


# Example 2: In-memory data structure, with pointers
COORD = {
    "x": 0 | uctypes.FLOAT32,
    "y": 4 | uctypes.FLOAT32,
}

STRUCT1 = {
```

(continues on next page)

(continued from previous page)

```
    "data1": 0 | uctypes.UINT8,
    "data2": 4 | uctypes.UINT32,
    "ptr": (8 | uctypes.PTR, COORD),
}

# Suppose you have address of a structure of type STRUCT1 in "addr"
# uctypes.NATIVE is optional (used by default)
struct1 = uctypes.struct(addr, STRUCT1, uctypes.NATIVE)
print("x:", struct1.ptr[0].x)


# Example 3: Access to CPU registers. Subset of STM32F4xx WWDG block
WWDG_LAYOUT = {
    "WWDG_CR": (0, {
        # BFUINT32 here means size of the WWDG_CR register
        "WDGA": 7 << uctypes.BF_POS | 1 << uctypes.BF_LEN | uctypes.BFUINT32,
        "T": 0 << uctypes.BF_POS | 7 << uctypes.BF_LEN | uctypes.BFUINT32,
    }),
    "WWDG_CFR": (4, {
        "EWI": 9 << uctypes.BF_POS | 1 << uctypes.BF_LEN | uctypes.BFUINT32,
        "WDGTB": 7 << uctypes.BF_POS | 2 << uctypes.BF_LEN | uctypes.BFUINT32,
        "W": 0 << uctypes.BF_POS | 7 << uctypes.BF_LEN | uctypes.BFUINT32,
    }),
}

WWDG = uctypes.struct(0x40002c00, WWDG_LAYOUT)

WWDG.WWDG_CFR.WDGTB = 0b10
WWDG.WWDG_CR.WDGA = 1
print("Current counter:", WWDG.WWDG_CR.T)
```

Defining structure layout

Structure layout is defined by a "descriptor" - a Python dictionary which encodes field names as keys and other properties required to access them as associated values:

```
{
    "field1": <properties>,
    "field2": <properties>,
    ...
}
```

Currently, `uctypes` requires explicit specification of offsets for each field. Offset are given in bytes from the structure start.

Following are encoding examples for various field types:

- Scalar types:

  ```
  "field_name": offset | uctypes.UINT32
  ```

  in other words, the value is a scalar type identifier ORed with a field offset (in bytes) from the start of the structure.

- Recursive structures:

```
"sub": (offset, {
    "b0": 0 | uctypes.UINT8,
    "b1": 1 | uctypes.UINT8,
})
```

i.e. value is a 2-tuple, first element of which is an offset, and second is a structure descriptor dictionary (note: offsets in recursive descriptors are relative to the structure it defines). Of course, recursive structures can be specified not just by a literal dictionary, but by referring to a structure descriptor dictionary (defined earlier) by name.

- Arrays of primitive types:

```
"arr": (offset | uctypes.ARRAY, size | uctypes.UINT8),
```

i.e. value is a 2-tuple, first element of which is ARRAY flag ORed with offset, and second is scalar element type ORed number of elements in the array.

- Arrays of aggregate types:

```
"arr2": (offset | uctypes.ARRAY, size, {"b": 0 | uctypes.UINT8}),
```

i.e. value is a 3-tuple, first element of which is ARRAY flag ORed with offset, second is a number of elements in the array, and third is a descriptor of element type.

- Pointer to a primitive type:

```
"ptr": (offset | uctypes.PTR, uctypes.UINT8),
```

i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, and second is a scalar element type.

- Pointer to an aggregate type:

```
"ptr2": (offset | uctypes.PTR, {"b": 0 | uctypes.UINT8}),
```

i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, second is a descriptor of type pointed to.

- Bitfields:

```
"bitf0": offset | uctypes.BFUINT16 | lsbit << uctypes.BF_POS | bitsize << uctypes.
↪BF_LEN,
```

i.e. value is a type of scalar value containing given bitfield (typenames are similar to scalar types, but prefixes with `BF`), ORed with offset for scalar value containing the bitfield, and further ORed with values for bit position and bit length of the bitfield within the scalar value, shifted by BF_POS and BF_LEN bits, respectively. A bitfield position is counted from the least significant bit of the scalar (having position of 0), and is the number of right-most bit of a field (in other words, it's a number of bits a scalar needs to be shifted right to extract the bitfield).

In the example above, first a UINT16 value will be extracted at offset 0 (this detail may be important when accessing hardware registers, where particular access size and alignment are required), and then bitfield whose rightmost bit is *lsbit* bit of this UINT16, and length is *bitsize* bits, will be extracted. For example, if *lsbit* is 0 and *bitsize* is 8, then effectively it will access least-significant byte of UINT16.

Note that bitfield operations are independent of target byte endianness, in particular, example above will access least-significant byte of UINT16 in both little- and big-endian structures. But it depends on the least significant bit being numbered 0. Some targets may use different numbering in their native ABI, but `uctypes` always uses the normalized numbering described above.

Module contents

class uctypes.**struct**(*addr*, *descriptor*, *layout_type=NATIVE*, */*)

Instantiate a "foreign data structure" object based on structure address in memory, descriptor (encoded as a dictionary), and layout type (see below).

uctypes.**LITTLE_ENDIAN**

Layout type for a little-endian packed structure. (Packed means that every field occupies exactly as many bytes as defined in the descriptor, i.e. the alignment is 1).

uctypes.**BIG_ENDIAN**

Layout type for a big-endian packed structure.

uctypes.**NATIVE**

Layout type for a native structure - with data endianness and alignment conforming to the ABI of the system on which MicroPython runs.

uctypes.**sizeof**(*struct*, *layout_type=NATIVE*, */*)

Return size of data structure in bytes. The *struct* argument can be either a structure class or a specific instantiated structure object (or its aggregate field).

uctypes.**addressof**(*obj*)

Return address of an object. Argument should be bytes, bytearray or other object supporting buffer protocol (and address of this buffer is what actually returned).

uctypes.**bytes_at**(*addr*, *size*)

Capture memory at the given address and size as bytes object. As bytes object is immutable, memory is actually duplicated and copied into bytes object, so if memory contents change later, created object retains original value.

uctypes.**bytearray_at**(*addr*, *size*)

Capture memory at the given address and size as bytearray object. Unlike bytes_at() function above, memory is captured by reference, so it can be both written too, and you will access current value at the given memory address.

uctypes.**UINT8**

uctypes.**INT8**

uctypes.**UINT16**

uctypes.**INT16**

uctypes.**UINT32**

uctypes.**INT32**

uctypes.**UINT64**

uctypes.**INT64**

Integer types for structure descriptors. Constants for 8, 16, 32, and 64 bit types are provided, both signed and unsigned.

uctypes.**FLOAT32**

uctypes.**FLOAT64**

Floating-point types for structure descriptors.

uctypes.**VOID**

`VOID` is an alias for `UINT8`, and is provided to conveniently define C's void pointers: `(uctypes.PTR, uctypes.VOID)`.

uctypes.**PTR**

uctypes.**ARRAY**

Type constants for pointers and arrays. Note that there is no explicit constant for structures, it's implicit: an aggregate type without `PTR` or `ARRAY` flags is a structure.

Structure descriptors and instantiating structure objects

Given a structure descriptor dictionary and its layout type, you can instantiate a specific structure instance at a given memory address using *uctypes.struct()* constructor. Memory address usually comes from following sources:

- Predefined address, when accessing hardware registers on a baremetal system. Lookup these addresses in datasheet for a particular MCU/SoC.
- As a return value from a call to some FFI (Foreign Function Interface) function.
- From *uctypes.addressof()*, when you want to pass arguments to an FFI function, or alternatively, to access some data for I/O (for example, data read from a file or network socket).

Structure objects

Structure objects allow accessing individual fields using standard dot notation: `my_struct.substruct1.field1`. If a field is of scalar type, getting it will produce a primitive value (Python integer or float) corresponding to the value contained in a field. A scalar field can also be assigned to.

If a field is an array, its individual elements can be accessed with the standard subscript operator `[]` - both read and assigned to.

If a field is a pointer, it can be dereferenced using `[0]` syntax (corresponding to C `*` operator, though `[0]` works in C too). Subscripting a pointer with other integer values but 0 are also supported, with the same semantics as in C.

Summing up, accessing structure fields generally follows the C syntax, except for pointer dereference, when you need to use `[0]` operator instead of `*`.

Limitations

1. Accessing non-scalar fields leads to allocation of intermediate objects to represent them. This means that special care should be taken to layout a structure which needs to be accessed when memory allocation is disabled (e.g. from an interrupt). The recommendations are:

- Avoid accessing nested structures. For example, instead of `mcu_registers.peripheral_a.register1`, define separate layout descriptors for each peripheral, to be accessed as `peripheral_a.register1`. Or just cache a particular peripheral: `peripheral_a = mcu_registers.peripheral_a`. If a register consists of multiple bitfields, you would need to cache references to a particular register: `reg_a = mcu_registers.peripheral_a.reg_a`.
- Avoid other non-scalar data, like arrays. For example, instead of `peripheral_a.register[0]` use `peripheral_a.register0`. Again, an alternative is to cache intermediate values, e.g. `register0 = peripheral_a.register[0]`.

2. Range of offsets supported by the `uctypes` module is limited. The exact range supported is considered an implementation detail, and the general suggestion is to split structure definitions to cover from a few kilobytes to a few dozen of kilobytes maximum. In most cases, this is a natural situation anyway, e.g. it doesn't make sense to define all registers of an MCU (spread over 32-bit address space) in one structure, but rather a peripheral block by peripheral block. In some extreme cases, you may need to split a structure in several parts artificially (e.g. if accessing native data structure with multi-megabyte array in the middle, though that would be a very synthetic case).

1.2.12 `vfs` – virtual filesystem control

The `vfs` module contains functions for creating filesystem objects and mounting/unmounting them in the Virtual Filesystem.

Filesystem mounting

Some ports provide a Virtual Filesystem (VFS) and the ability to mount multiple "real" filesystems within this VFS. Filesystem objects can be mounted at either the root of the VFS, or at a subdirectory that lives in the root. This allows dynamic and flexible configuration of the filesystem that is seen by Python programs. Ports that have this functionality provide the *mount()* and *umount()* functions, and possibly various filesystem implementations represented by VFS classes.

`vfs.`**`mount`**(*fsobj*, *mount_point*, *, *readonly*)

Mount the filesystem object *fsobj* at the location in the VFS given by the *mount_point* string. *fsobj* can be a a VFS object that has a `mount()` method, or a block device. If it's a block device then the filesystem type is automatically detected (an exception is raised if no filesystem was recognised). *mount_point* may be `'/'` to mount *fsobj* at the root, or `'/<name>'` to mount it at a subdirectory under the root.

If *readonly* is `True` then the filesystem is mounted read-only.

During the mount process the method `mount()` is called on the filesystem object.

Will raise `OSError(EPERM)` if *mount_point* is already mounted.

`vfs.`**`mount`**()

With no arguments to *mount()*, return a list of tuples representing all active mountpoints.

The returned list has the form *[(fsobj, mount_point), ...]*.

`vfs.`**`umount`**(*mount_point*)

Unmount a filesystem. *mount_point* can be a string naming the mount location, or a previously-mounted filesystem object. During the unmount process the method `umount()` is called on the filesystem object.

Will raise `OSError(EINVAL)` if *mount_point* is not found.

Filesystem Types

`class` `vfs.`**`VfsFat`**(*block_dev*)

Create a filesystem object that uses the FAT filesystem format. Storage of the FAT filesystem is provided by *block_dev*. Objects created by this constructor can be mounted using *mount()*.

`static mkfs`(*block_dev*)

Build a FAT filesystem on *block_dev*.

`class` `vfs.`**`VfsLfs1`**(*block_dev*, *readsize=32*, *progsize=32*, *lookahead=32*)

Create a filesystem object that uses the littlefs v1 filesystem format. Storage of the littlefs filesystem is provided by *block_dev*, which must support the *extended interface*. Objects created by this constructor can be mounted using *mount()*.

See *Working with filesystems* for more information.

`static mkfs`(*block_dev*, *readsize=32*, *progsize=32*, *lookahead=32*)

Build a Lfs1 filesystem on *block_dev*.

Note: There are reports of littlefs v1 failing in certain situations, for details see littlefs issue 347.

class vfs.**VfsLfs2**(*block_dev*, *readsize=32*, *progsize=32*, *lookahead=32*, *mtime=True*)

Create a filesystem object that uses the littlefs v2 filesystem format. Storage of the littlefs filesystem is provided by *block_dev*, which must support the *extended interface*. Objects created by this constructor can be mounted using `mount()`.

The *mtime* argument enables modification timestamps for files, stored using littlefs attributes. This option can be disabled or enabled differently each mount time and timestamps will only be added or updated if *mtime* is enabled, otherwise the timestamps will remain untouched. Littlefs v2 filesystems without timestamps will work without reformatting and timestamps will be added transparently to existing files once they are opened for writing. When *mtime* is enabled `os.stat` on files without timestamps will return 0 for the timestamp.

See *Working with filesystems* for more information.

static mkfs(*block_dev*, *readsize=32*, *progsize=32*, *lookahead=32*)

Build a Lfs2 filesystem on *block_dev*.

Note: There are reports of littlefs v2 failing in certain situations, for details see littlefs issue 295.

class vfs.**VfsPosix**(*root=None*)

Create a filesystem object that accesses the host POSIX filesystem. If *root* is specified then it should be a path in the host filesystem to use as the root of the `VfsPosix` object. Otherwise the current directory of the host filesystem is used.

class vfs.**VfsRom**(*buffer*)

This class is only available when the firmware is built with `MICROPY_VFS_ROM` enabled.

Create a filesystem object that uses the *ROMFS read-only filesystem format*. *buffer* must be an object supporting the buffer protocol (e.g. `bytes`, `bytearray`, or `memoryview`) that contains a valid ROMFS image. The constructor validates that *buffer* begins with the ROMFS magic bytes (`b"\xd2\xcd\x31"`). If the buffer is too small or not a valid ROMFS then `OSError(ENODEV)` is raised.

See *Working with ROMFS* for more information on the ROMFS filesystem and how to deploy images using *mpremote*.

open(*path*, *mode*)

Open a file from the ROMFS. Only read modes (`''`, `'r'`, `'rt'`, `'rb'`) are supported. For binary files opened in read mode, the returned object also supports the buffer protocol so that a `memoryview` of the file data can be obtained, which refers directly into the ROMFS memory (zero-copy).

statvfs(*path*)

The block size is reported as 1 and the block count represents the total size of the ROMFS image in bytes.

chdir(*path*)

Change directory within the ROMFS. Only the root (`'/'`) is supported; changing to any subdirectory raises `OSError(EOPNOTSUPP)`.

Miscellaneous Functions

vfs.**rom_ioctl**(*op*, ...)

Low-level interface for accessing the read-only memory (ROM) partition(s) of the device. This function is only available on ports that support ROMFS (i.e. where MICROPY_VFS_ROM_IOCTL is enabled).

The supported operations are:

- `vfs.rom_ioctl(1)` – Return the number of available ROM partitions.
- `vfs.rom_ioctl(2, id)` – Return ROM partition *id* as an object that supports the buffer protocol. Depending on the port, this is either a block device or a `memoryview`. A block device supports the standard block protocol, including erase and write operations. If a `memoryview` is returned, the port must provide erase and write operations through `vfs.rom_ioctl()` operations 3, 4, and 5 below.
- `vfs.rom_ioctl(3, id, length)` – Prepare the first *length* bytes of ROM partition *id* for writing (for example, by erasing flash). Returns the minimum write size in bytes (the alignment required for subsequent writes).
- `vfs.rom_ioctl(3, id, offset, length)` – Prepare *length* bytes of ROM partition *id*, starting at byte *offset*, for writing. This form allows a partition to be prepared incrementally. Returns the minimum write size in bytes.
- `vfs.rom_ioctl(4, id, offset, buf)` – Write *buf* (a bytes-like object) to the ROM partition with index *id* at byte *offset*.
- `vfs.rom_ioctl(5, id)` – Complete a write sequence to partition *id* (performs any finalisation needed after writing, such as cache flushing).
- `vfs.rom_ioctl(6, id)` – Return the minimum number of bytes of ROM partition *id* that can be prepared at once. A positive return value indicates that the four-argument form of operation 3 is supported; *offset* and *length* must be aligned to this value. Otherwise, only the three-argument form of operation 3 is supported.

These operations are used internally by `mpremote` to deploy ROMFS images. Most users do not need to call `vfs.rom_ioctl()` directly.

See *Working with ROMFS* for more information.

Block devices

A block device is an object which implements the block protocol. This enables a device to support MicroPython filesystems. The physical hardware is represented by a user defined class. The *AbstractBlockDev* class is a template for the design of such a class: MicroPython does not actually provide that class, but an actual block device class must implement the methods described below.

A concrete implementation of this class will usually allow access to the memory-like functionality of a piece of hardware (like flash memory). A block device can be formatted to any supported filesystem and mounted using `os` methods.

See *Working with filesystems* for example implementations of block devices using the two variants of the block protocol described below.

Simple and extended interface

There are two compatible signatures for the `readblocks` and `writeblocks` methods (see below), in order to support a variety of use cases. A given block device may implement one form or the other, or both at the same time. The second form (with the offset parameter) is referred to as the "extended interface".

Some filesystems (such as littlefs) that require more control over write operations, for example writing to sub-block regions without erasing, may require that the block device supports the extended interface.

class vfs.**AbstractBlockDev**(...)

Construct a block device object. The parameters to the constructor are dependent on the specific block device.

readblocks(*block_num*, *buf*)

readblocks(*block_num*, *buf*, *offset*)

The first form reads aligned, multiples of blocks. Starting at the block given by the index *block_num*, read blocks from the device into *buf* (an array of bytes). The number of blocks to read is given by the length of *buf*, which will be a multiple of the block size.

The second form allows reading at arbitrary locations within a block, and arbitrary lengths. Starting at block index *block_num*, and byte offset within that block of *offset*, read bytes from the device into *buf* (an array of bytes). The number of bytes to read is given by the length of *buf*.

Upon success the method should return `None` or 0. Upon failure it should return a negative integer corresponding to an `OSError` errno code.

writeblocks(*block_num*, *buf*)

writeblocks(*block_num*, *buf*, *offset*)

The first form writes aligned, multiples of blocks, and requires that the blocks that are written to be first erased (if necessary) by this method. Starting at the block given by the index *block_num*, write blocks from *buf* (an array of bytes) to the device. The number of blocks to write is given by the length of *buf*, which will be a multiple of the block size.

The second form allows writing at arbitrary locations within a block, and arbitrary lengths. Only the bytes being written should be changed, and the caller of this method must ensure that the relevant blocks are erased via a prior `ioctl` call. Starting at block index *block_num*, and byte offset within that block of *offset*, write bytes from *buf* (an array of bytes) to the device. The number of bytes to write is given by the length of *buf*.

Note that implementations must never implicitly erase blocks if the offset argument is specified, even if it is zero.

Upon success the method should return `None` or 0. Upon failure it should return a negative integer corresponding to an `OSError` errno code.

ioctl(*op*, *arg*)

Control the block device and query its parameters. The operation to perform is given by *op* which is one of the following integers:

- 1 – initialise the device (*arg* is unused)
- 2 – shutdown the device (*arg* is unused)
- 3 – sync the device (*arg* is unused)
- 4 – get a count of the number of blocks, should return an integer (*arg* is unused)
- 5 – get the number of bytes in a block, should return an integer, or `None` in which case the default value of 512 is used (*arg* is unused)
- 6 – erase a block, *arg* is the block number to erase

As a minimum `ioctl(4, ...)` must be intercepted; for littlefs `ioctl(6, ...)` must also be intercepted. The need for others is hardware dependent.

Prior to any call to `writeblocks(block, ...)` littlefs issues `ioctl(6, block)`. This enables a device driver to erase the block prior to a write if the hardware requires it. Alternatively a driver might intercept `ioctl(6, block)` and return 0 (success). In this case the driver assumes responsibility for detecting the need for erasure.

Unless otherwise stated `ioctl(op, arg)` can return `None`. Consequently an implementation can ignore unused values of `op`. Where `op` is intercepted, the return value for operations 4 and 5 are as detailed above. Other operations should return 0 on success and non-zero for failure, with the value returned being an `OSError` errno code.

The following libraries provide drivers for hardware components.

1.2.13 `WM8960` – Driver for the WM8960 codec

This driver is used to control a WM8960 codec chip. It is a Python translation of the C-Code provided by NXP/Freescale for their i.MX RT series of MCUs. Very little has been added, and just a few API related names were changed or added to cope with the naming style of MicroPython.

The primary purpose of the driver is initialization and setting operation modes of the codec. It does not do the audio data processing for the codec. That is the task of a separate driver.

The WM8960 supports an I2C interface, in addition to the audio interface. The connection depends on the interface used and the number of devices in the system. For the I2C interface, SCL and SDA have to be connected, and of course GND and Vcc. The I2C default address is `0x1A`.

Constructor

class WM8960(*i2c*, *sample_rate*, ***, *bits=16*, *swap=SWAP_NONE*, *route=ROUTE_PLAYBACK_RECORD*, *left_input=INPUT_MIC3*, *right_input=INPUT_MIC2*, *sysclk_source=SYSCLK_MCLK*, *mclk_freq=None*, *primary=False*, *adc_sync=SYNC_DAC*, *protocol=BUS_I2S*, *i2c_address=WM8960_I2C_ADDR*)

Create a WM8960 driver object, initialize the device with default settings and return the WM8960 object.

Only the first two arguments are mandatory. All others are optional. The arguments are:

- *i2c* is the I2C bus object.
- *sample_rate* is the audio sample rate. Acceptable values are 8000, 11025, 12000, 16000, 22050, 24000, 32000, 44100, 48000, 96000, 192000 and 384000. Note that not every I2S hardware will support all values.
- *bits* is the number of bits per audio word. Acceptable value are 16, 20, 24, and 32.
- *swap* swaps the left & right channel, if set; see below for options.
- *route* Setting the audio path in the codec; see below for options.
- *left_input* sets the audio source for the left input channel; see below for options.
- *right_input* sets the audio source for the right input channel; see below for options.
- *play_source* sets the audio target for the output audio; see below for options.
- *sysclk_source* controls whether the internal master clock called “sysclk” is directly taken from the MCLK input or derived from it using an internal PLL. It is usually not required to change this.
- *mclk_freq* sets the mclk frequency applied to the MCLK pin of the codec. If not set, default values are used.

- *primary* lets the WM8960 act as primary or secondary device. The default setting is `False`. When set to `False`, *sample_rate* and *bits* are controlled by the MCU.
- *adc_sync* sets which input is used for the ADC sync signal. The default is using the DACLRC pin.
- *protocol* sets the communication protocol. The default is I2S. See below for all options.
- *i2c_address* sets the I2C address of the WM8960, with default `0x1A`.

If *mclk_freq* is not set the following default values are used:

- sysclk_source == SYSCLK_PLL: 11.2896 MHz for sample rates of 44100, 22050 and 11015 Hz, and 12.288 Mhz for sample rates < 48000, otherwise sample_rate * 256.
- sysclk_source == SYSCLK_MCLK: sample_rate * 256.

If the MCLK signal is applied using, for example,. a separate oscillator, it must be specified for proper operation.

Tables of parameter constants

Table 2: **Swap Parameter**

Value	Name
0	SWAP_NONE
1	SWAP_INPUT
2	SWAP_OUTPUT

Table 3: **Protocol Parameter**

Value	Name
2	BUS_I2S
1	BUS_LEFT_JUSTIFIED
0	BUS_RIGHT_JUSTIFIED
3	BUS_PCMA
19	BUS_PCMB

Table 4: **Input Source Parameter**

Value	Name	Type
0	INPUT_CLOSED	
1	INPUT_MIC1	Single ended
2	INPUT_MIC2	Differential
3	INPUT_MIC3	Differential
4	INPUT_LINE2	
5	INPUT_LINE3	

Table 5: **Route Parameter**

Value	Name
0	ROUTE_BYPASS
1	ROUTE_PLAYBACK
2	ROUTE_PLAYBACK_RECORD
5	ROUTE_RECORD

Table 6: **Master Clock Source Parameter**

Value	Name
0	SYSCLK_MCLK
1	SYSCLK_PLL

Table 7: **Module Names**

Value	Name
0	MODULE_ADC
1	MODULE_DAC
2	MODULE_VREF
3	MODULE_HEADPHONE
4	MODULE_MIC_BIAS
5	MODULE_MIC
6	MODULE_LINE_IN
7	MODULE_LINE_OUT
8	MODULE_SPEAKER
9	MODULE_OMIX
10	MODULE_MONO_OUT

Table 8: **Play Channel Names**

Value	Name
1	PLAY_HEADPHONE_LEFT
2	PLAY_HEADPHONE_RIGHT
4	PLAY_SPEAKER_LEFT
8	PLAY_SPEAKER_RIGHT

Table 9: **adc_sync Parameters**

Value	Name
0	SYNC_ADC
1	SYNC_DAC

Methods

In addition to initialization, the driver provides some useful methods for controlling its operation:

`WM8960.`**`set_left_input`**(*input_source*)

Specify the source for the left input. The input source names are listed above.

`WM8960.`**`set_right_input`**(*input_source*)

Specify the source for the right input. The input source names are listed above.

`WM8960.`**`volume`**(*module*, *volume_l=None*, *volume_r=None*)

Sets or gets the volume of a certain module.

If no volume values are supplied, the actual volume tuple is returned.

If one or two values are supplied, it sets the volume of a certain module. If two values are provided, the first one is used for the left channel, the second for the right channel. If only one value is supplied, it is used for both channels. The value range is normalized to 0.0-100.0 with a logarithmic scale.

For a list of suitable modules and db/step, see the table below.

Table 10: **Module Names and dB steps**

dB/Step	Name
1.28	MODULE_ADC
1.28	MODULE_DAC
0.8	MODULE_HEADPHONE
0.475	MODULE_LINE_IN
0.8	MODULE_SPEAKER

WM8960.**mute**(*module*, *mute*, *soft=True*, *ramp=wm8960.MUTE_FAST*)

Mute or unmute the output. If *mute* is True, the output is muted, if `False` it is unmuted.

If *soft* is set as True, muting will happen as a soft transition. The time for the transition is defined by *ramp*, which is either `MUTE_FAST` or `MUTE_SLOW`.

WM8960.**set_data_route**(*route*)

Set the audio data route. For the parameter value/names, see the table above.

WM8960.**set_module**(*module*, *active*)

Enable or disable a module, with *active* being `False` or `True`. For the list of module names, see the table above.

Note that enabling `MODULE_MONO_OUT` is different from the *WM8960.mono* method. The first enables output 3, while the *WM8960.mono* method sends a mono mix to the left and right output.

WM8960.**enable_module**(*module*)

Enable a module. For the list of module names, see the table above.

WM8960.**disable_module**(*module*)

Disable a module. For the list of module names, see the table above.

WM8960.**expand_3d**(*level*)

Enable Stereo 3D expansion. *level* is a number between 0 and 15. A value of 0 disables the expansion.

WM8960.**mono**(*active*)

If *active* is `True`, a Mono mix is sent to the left and right output channel. This is different from enabling the `MODULE_MONO_MIX`, which enables output 3.

WM8960.**alc_mode**(*channel*, *mode=ALC_MODE*)

Enables or disables ALC mode. Parameters are:

- *channel* enables and sets the channel for ALC. The parameter values are:
  - ALC_OFF: Switch ALC off
  - ALS_RIGHT: Use the right input channel
  - ALC_LEFT: Use the left input channel
  - ALC_STEREO: Use both input channels.
- *mode* sets the ALC mode. Input values are:
  - ALC_MODE: act as ALC

- ALC_LIMITER: act as limiter.

WM8960.**alc_gain**(*target=-12*, *max_gain=30*, *min_gain=-17.25*, *noise_gate=-78*)

Set the target level, highest and lowest gain levels and the noise gate as dB level. Permitted ranges are:

- *target*: -22.5 to -1.5 dB
- *max_gain*: -12 to 30 dB
- *min_gain*: -17 to 25 dB
- *noise_gate*: -78 to -30 dB

Excess values are limited to the permitted ranges. A value of -78 or less for *noise_gate* disables the noise gate function.

WM8960.**alc_time**(*attack=24*, *decay=192*, *hold=0*)

Set the dynamic characteristic of ALC. The times are given as millisecond values. Permitted ranges are:

- *attack*: 6 to 6140
- *decay*: 24 to 24580
- *hold*: 0 to 43000

Excess values are limited within the permitted ranges.

WM8960.**deemphasis**(*active*)

Enables or disables a deemphasis filter for playback, with *active* being `False` or `True`. This filter is applied only for sample rates of 32000, 44100 and 48000. For other sample rates, the filter setting is silently ignored.

WM8960.**deinit**()

Disable all modules.

Examples

Run WM8960 in secondary mode (default):

```
# Micro_python WM8960 Codec driver
#
# Setting the driver to Slave mode using the default settings
#
from machine import Pin, I2C
import wm8960
i2c = I2C(0)
wm=wm8960.WM8960(i2c, 32000, left_input=wm8960.INPUT_MIC1)
wm.set_volume(wm8960.MODULE_HEADPHONE, 100)
```

Run WM8960 in primary mode:

```
# Micro_python WM8960 Codec driver
#
# Setting the driver to Master mode using specific audio format settings
#
from machine import Pin, I2C
import wm8960

i2c = I2C(0)
wm=wm8960.WM8960(i2c, 44100, primary=True, bits=16)
```

Run WM8960 on a MIMXRT10xx_DEV board in secondary mode (default):

```
# Micro_python WM8960 Codec driver
#
# Setting the driver to Slave mode using the default settings
# swap the input channels such that a MIMXRT Dev board mic, which
# is connected to the right input, is assigned to the left audio channel.
#
from machine import Pin, I2C
import wm8960
i2c = I2C(0)
wm=wm8960.WM8960(i2c, sample_rate=16_000,
    adc_sync=wm8960.SYNC_DAC,
    swap=wm8960.SWAP_INPUT,
    sysclk_source=wm8960.SYSCLK_MCLK)
```

Record with a SparkFun WM8960 breakout board with Teensy in secondary mode (default):

```
# Micro_python WM8960 Codec driver
#
# The breakout board uses a fixed 24MHz MCLK. Therefore the internal
# PLL must be used as sysclk, which is the master audio clock.
# The SparkFun board has the WS pins for RX and TX connected on the
# board. Therefore adc_sync must be set to sync_adc, to configure
# it's ADCLRC pin as input.
#
from machine import Pin, I2C
import wm8960
i2c = I2C(0)
wm=wm8960.WM8960(i2c, sample_rate=16_000,
    adc_sync=wm8960.SYNC_ADC,
    sysclk_source=wm8960.SYSCLK_PLL,
    mclk_freq=24_000_000,
    left_input=wm8960.INPUT_MIC1,
    right_input=wm8960.INPUT_CLOSED)
```

Play with a SparkFun WM8960 breakout board with Teensy in secondary mode (default):

```
# The breakout board uses a fixed 24MHz MCLK. Therefore the internal
# PLL must be used as sysclk, which is the master audio clock.
# The SparkFun board has the WS pins for RX and TX connected on the
# board. Therefore adc_sync must be set to sync_adc, to configure
# it's ADCLRC pin as input.

from machine import I2C
i2c=I2C(0)
import wm8960
wm=wm8960.WM8960(i2c, sample_rate=44_100,
    adc_sync=wm8960.SYNC_ADC,
    sysclk_source=wm8960.SYSCLK_PLL,
    mclk_freq=24_000_000)
wm.set_volume(wm8960.MODULE_HEADPHONE, 100)
```

1.3 Port-specific libraries

In some cases the following port/board-specific libraries have functions or classes similar to those in the *machine* library. Where this occurs, the entry in the port specific library exposes hardware functionality unique to that platform.

To write portable code use functions and classes from the *machine* module. To access platform-specific hardware use the appropriate library, e.g. *pyb* in the case of the Pyboard.

1.3.1 Libraries specific to the pyboard

The following libraries are specific to the pyboard.

pyb — functions related to the board

The pyb module contains specific functions related to the board.

Time related functions

pyb.**delay**(*ms*)

Delay for the given number of milliseconds.

pyb.**udelay**(*us*)

Delay for the given number of microseconds.

pyb.**millis**()

Returns the number of milliseconds since the board was last reset.

The result is always a MicroPython smallint (31-bit signed number), so after 2^30 milliseconds (about 12.4 days) this will start to return negative numbers.

Note that if *pyb.stop()* is issued the hardware counter supporting this function will pause for the duration of the "sleeping" state. This will affect the outcome of *pyb.elapsed_millis()*.

pyb.**micros**()

Returns the number of microseconds since the board was last reset.

The result is always a MicroPython smallint (31-bit signed number), so after 2^30 microseconds (about 17.8 minutes) this will start to return negative numbers.

Note that if *pyb.stop()* is issued the hardware counter supporting this function will pause for the duration of the "sleeping" state. This will affect the outcome of *pyb.elapsed_micros()*.

pyb.**elapsed_millis**(*start*)

Returns the number of milliseconds which have elapsed since start.

This function takes care of counter wrap, and always returns a positive number. This means it can be used to measure periods up to about 12.4 days.

Example:

```
start = pyb.millis()
while pyb.elapsed_millis(start) < 1000:
    # Perform some operation
```

pyb.**elapsed_micros**(*start*)

Returns the number of microseconds which have elapsed since `start`.

This function takes care of counter wrap, and always returns a positive number. This means it can be used to measure periods up to about 17.8 minutes.

Example:

```
start = pyb.micros()
while pyb.elapsed_micros(start) < 1000:
    # Perform some operation
    pass
```

Reset related functions

pyb.**hard_reset**()

Resets the pyboard in a manner similar to pushing the external RESET button.

pyb.**bootloader**()

Activate the bootloader without BOOT* pins.

pyb.**fault_debug**(*value*)

Enable or disable hard-fault debugging. A hard-fault is when there is a fatal error in the underlying system, like an invalid memory access.

If the *value* argument is `False` then the board will automatically reset if there is a hard fault.

If *value* is `True` then, when the board has a hard fault, it will print the registers and the stack trace, and then cycle the LEDs indefinitely.

The default value is disabled, i.e. to automatically reset.

Interrupt related functions

pyb.**disable_irq**()

Disable interrupt requests. Returns the previous IRQ state: `False`/`True` for disabled/enabled IRQs respectively. This return value can be passed to enable_irq to restore the IRQ to its original state.

pyb.**enable_irq**(*state=True*)

Enable interrupt requests. If `state` is `True` (the default value) then IRQs are enabled. If `state` is `False` then IRQs are disabled. The most common use of this function is to pass it the value returned by `disable_irq` to exit a critical section.

Power related functions

pyb.**freq**([*sysclk*[, *hclk*[, *pclk1*[, *pclk2*]]]])

If given no arguments, returns a tuple of clock frequencies: (sysclk, hclk, pclk1, pclk2). These correspond to:

- sysclk: frequency of the CPU
- hclk: frequency of the AHB bus, core memory and DMA
- pclk1: frequency of the APB1 bus
- pclk2: frequency of the APB2 bus

If given any arguments then the function sets the frequency of the CPU, and the buses if additional arguments are given. Frequencies are given in Hz. Eg freq(120000000) sets sysclk (the CPU frequency) to 120MHz. Note that not all values are supported and the largest supported frequency not greater than the given value will be selected.

Supported sysclk frequencies are (in MHz): 8, 16, 24, 30, 32, 36, 40, 42, 48, 54, 56, 60, 64, 72, 84, 96, 108, 120, 144, 168.

The maximum frequency of hclk is 168MHz, of pclk1 is 42MHz, and of pclk2 is 84MHz. Be sure not to set frequencies above these values.

The hclk, pclk1 and pclk2 frequencies are derived from the sysclk frequency using a prescaler (divider). Supported prescalers for hclk are: 1, 2, 4, 8, 16, 64, 128, 256, 512. Supported prescalers for pclk1 and pclk2 are: 1, 2, 4, 8. A prescaler will be chosen to best match the requested frequency.

A sysclk frequency of 8MHz uses the HSE (external crystal) directly and 16MHz uses the HSI (internal oscillator) directly. The higher frequencies use the HSE to drive the PLL (phase locked loop), and then use the output of the PLL.

Note that if you change the frequency while the USB is enabled then the USB may become unreliable. It is best to change the frequency in *boot.py*, before the USB peripheral is started. Also note that sysclk frequencies below 36MHz do not allow the USB to function correctly.

pyb.**wfi**()

Wait for an internal or external interrupt.

This executes a `wfi` instruction which reduces power consumption of the MCU until any interrupt occurs (be it internal or external), at which point execution continues. Note that the system-tick interrupt occurs once every millisecond (1000Hz) so this function will block for at most 1ms.

pyb.**stop**()

Put the pyboard in a "sleeping" state.

This reduces power consumption to less than 500 uA. To wake from this sleep state requires an external interrupt or a real-time-clock event. Upon waking execution continues where it left off.

See `rtc.wakeup()` to configure a real-time-clock wakeup event.

pyb.**standby**()

Put the pyboard into a "deep sleep" state.

This reduces power consumption to less than 50 uA. To wake from this sleep state requires a real-time-clock event, or an external interrupt on X1 (PA0=WKUP) or X18 (PC13=TAMP1). Upon waking the system undergoes a hard reset.

See `rtc.wakeup()` to configure a real-time-clock wakeup event.

Miscellaneous functions

pyb.**have_cdc**()

Return True if USB is connected as a serial device, False otherwise.

Note: This function is deprecated. Use pyb.USB_VCP().isconnected() instead.

---

pyb.**hid**(*(buttons, x, y, z)*)

Takes a 4-tuple (or list) and sends it to the USB host (the PC) to signal a HID mouse-motion event.

Note: This function is deprecated. Use *pyb.USB_HID.send()* instead.

pyb.**info**([*dump_alloc_table*])

Print out lots of information about the board.

pyb.**main**(*filename*)

Set the filename of the main script to run after *boot.py* is finished. If this function is not called then the default file *main.py* will be executed.

It only makes sense to call this function from within boot.py.

pyb.**mount**(*device*, *mountpoint*, *, *readonly=False*, *mkfs=False*)

Note: This function is deprecated. Mounting and unmounting devices should be performed by *vfs.mount()* and *vfs.umount()* instead.

Mount a block device and make it available as part of the filesystem. `device` must be an object that provides the block protocol. (The following is also deprecated. See *vfs.AbstractBlockDev* for the correct way to create a block device.)

- `readblocks(self, blocknum, buf)`
- `writeblocks(self, blocknum, buf)` (optional)
- `count(self)`
- `sync(self)` (optional)

`readblocks` and `writeblocks` should copy data between `buf` and the block device, starting from block number `blocknum` on the device. `buf` will be a bytearray with length a multiple of 512. If `writeblocks` is not defined then the device is mounted read-only. The return value of these two functions is ignored.

`count` should return the number of blocks available on the device. `sync`, if implemented, should sync the data on the device.

The parameter `mountpoint` is the location in the root of the filesystem to mount the device. It must begin with a forward-slash.

If `readonly` is `True`, then the device is mounted read-only, otherwise it is mounted read-write.

If `mkfs` is `True`, then a new filesystem is created if one does not already exist.

pyb.**repl_uart**(*uart*)

Get or set the UART object where the REPL is repeated on.

pyb.**rng**()

Return a 30-bit hardware generated random number.

pyb.**sync**()

Sync all file systems.

pyb.**unique_id**()

Returns a string of 12 bytes (96 bits), which is the unique ID of the MCU.

pyb.**usb_mode**([*modestr*,]*port=-1*, *vid=0xf055*, *pid=-1*, *msc=()*, *hid=pyb.hid_mouse*, *high_speed=False*)

If called with no arguments, return the current USB mode as a string.

If called with *modestr* provided, attempts to configure the USB mode. The following values of *modestr* are understood:

- `None`: disables USB
- `'VCP'`: enable with VCP (Virtual COM Port) interface
- `'MSC'`: enable with MSC (mass storage device class) interface
- `'VCP+MSC'`: enable with VCP and MSC
- `'VCP+HID'`: enable with VCP and HID (human interface device)
- `'VCP+MSC+HID'`: enabled with VCP, MSC and HID (only available on PYBD boards)

For backwards compatibility, `'CDC'` is understood to mean `'VCP'` (and similarly for `'CDC+MSC'` and `'CDC+HID'`).

The *port* parameter should be an integer (0, 1, ...) and selects which USB port to use if the board supports multiple ports. A value of -1 uses the default or automatically selected port.

The *vid* and *pid* parameters allow you to specify the VID (vendor id) and PID (product id). A *pid* value of -1 will select a PID based on the value of *modestr*.

If enabling MSC mode, the *msc* parameter can be used to specify a list of SCSI LUNs to expose on the mass storage interface. For example `msc=(pyb.Flash(), pyb.SDCard())`.

If enabling HID mode, you may also specify the HID details by passing the *hid* keyword parameter. It takes a tuple of (subclass, protocol, max packet length, polling interval, report descriptor). By default it will set appropriate values for a USB mouse. There is also a `pyb.hid_keyboard` constant, which is an appropriate tuple for a USB keyboard.

The *high_speed* parameter, when set to `True`, enables USB HS mode if it is supported by the hardware.

Constants

`pyb.`**`hid_mouse`**

`pyb.`**`hid_keyboard`**

A tuple of (subclass, protocol, max packet length, polling interval, report descriptor) to set appropriate values for a USB mouse or keyboard.

Classes

class Accel – accelerometer control

Accel is an object that controls the accelerometer. Example usage:

```
accel = pyb.Accel()
for i in range(10):
    print(accel.x(), accel.y(), accel.z())
```

Raw values are between -32 and 31.

Constructors

class pyb.**Accel**

Create and return an accelerometer object.

Methods

Accel.**filtered_xyz**()

Get a 3-tuple of filtered x, y and z values.

Implementation note: this method is currently implemented as taking the sum of 4 samples, sampled from the 3 previous calls to this function along with the sample from the current call. Returned values are therefore 4 times the size of what they would be from the raw x(), y() and z() calls.

Accel.**tilt**()

Get the tilt register.

Accel.**x**()

Get the x-axis value.

Accel.**y**()

Get the y-axis value.

Accel.**z**()

Get the z-axis value.

Hardware Note

The accelerometer uses I2C bus 1 to communicate with the processor. Consequently when readings are being taken pins X9 and X10 should be unused (other than for I2C). Other devices using those pins, and which therefore cannot be used concurrently, are UART 1 and Timer 4 channels 1 and 2.

class ADC – analog to digital conversion

Usage:

```
import pyb

adc = pyb.ADC(pin)                  # create an analog object from a pin
val = adc.read()                    # read an analog value

adc = pyb.ADCAll(resolution)        # create an ADCAll object
adc = pyb.ADCAll(resolution, mask)  # create an ADCAll object for selected analog␣
↪channels
val = adc.read_channel(channel)     # read the given channel
val = adc.read_core_temp()          # read MCU temperature
val = adc.read_core_vbat()          # read MCU VBAT
val = adc.read_core_vref()          # read MCU VREF
val = adc.read_vref()               # read MCU supply voltage
```

Constructors

class pyb.**ADC**(*pin*)

Create an ADC object associated with the given pin. This allows you to then read analog values on that pin.

Methods

ADC.**read**()

Read the value on the analog pin and return it. The returned value will be between 0 and 4095.

ADC.**read_timed**(*buf*, *timer*)

Read analog values into `buf` at a rate set by the `timer` object.

`buf` can be bytearray or array.array for example. The ADC values have 12-bit resolution and are stored directly into `buf` if its element size is 16 bits or greater. If `buf` has only 8-bit elements (eg a bytearray) then the sample resolution will be reduced to 8 bits.

`timer` should be a Timer object, and a sample is read each time the timer triggers. The timer must already be initialised and running at the desired sampling frequency.

To support previous behaviour of this function, `timer` can also be an integer which specifies the frequency (in Hz) to sample at. In this case Timer(6) will be automatically configured to run at the given frequency.

Example using a Timer object (preferred way):

```
adc = pyb.ADC(pyb.Pin.board.X19)     # create an ADC on pin X19
tim = pyb.Timer(6, freq=10)          # create a timer running at 10Hz
buf = bytearray(100)                 # creat a buffer to store the samples
adc.read_timed(buf, tim)             # sample 100 values, taking 10s
```

Example using an integer for the frequency:

```
adc = pyb.ADC(pyb.Pin.board.X19)     # create an ADC on pin X19
buf = bytearray(100)                 # create a buffer of 100 bytes
adc.read_timed(buf, 10)              # read analog values into buf at 10Hz
                                     #   this will take 10 seconds to finish
for val in buf:                      # loop over all values
    print(val)                       # print the value out
```

This function does not allocate any heap memory. It has blocking behaviour: it does not return to the calling program until the buffer is full.

ADC.**read_timed_multi**((*adcx*, *adcy*, ...), (*bufx*, *bufy*, ...), *timer*)

This is a static method. It can be used to extract relative timing or phase data from multiple ADC's.

It reads analog values from multiple ADC's into buffers at a rate set by the *timer* object. Each time the timer triggers a sample is rapidly read from each ADC in turn.

ADC and buffer instances are passed in tuples with each ADC having an associated buffer. All buffers must be of the same type and length and the number of buffers must equal the number of ADC's.

Buffers can be `bytearray` or `array.array` for example. The ADC values have 12-bit resolution and are stored directly into the buffer if its element size is 16 bits or greater. If buffers have only 8-bit elements (eg a `bytearray`) then the sample resolution will be reduced to 8 bits.

timer must be a Timer object. The timer must already be initialised and running at the desired sampling frequency.

Example reading 3 ADC's:

```
adc0 = pyb.ADC(pyb.Pin.board.X1)     # Create ADC's
adc1 = pyb.ADC(pyb.Pin.board.X2)
adc2 = pyb.ADC(pyb.Pin.board.X3)
tim = pyb.Timer(8, freq=100)         # Create timer
rx0 = array.array('H', (0 for i in range(100))) # ADC buffers of
rx1 = array.array('H', (0 for i in range(100))) # 100 16-bit words
rx2 = array.array('H', (0 for i in range(100)))
# read analog values into buffers at 100Hz (takes one second)
pyb.ADC.read_timed_multi((adc0, adc1, adc2), (rx0, rx1, rx2), tim)
for n in range(len(rx0)):
    print(rx0[n], rx1[n], rx2[n])
```

This function does not allocate any heap memory. It has blocking behaviour: it does not return to the calling program until the buffers are full.

The function returns `True` if all samples were acquired with correct timing. At high sample rates the time taken to acquire a set of samples can exceed the timer period. In this case the function returns `False`, indicating a loss of precision in the sample interval. In extreme cases samples may be missed.

The maximum rate depends on factors including the data width and the number of ADC's being read. In testing two ADC's were sampled at a timer rate of 210kHz without overrun. Samples were missed at 215kHz. For three ADC's the limit is around 140kHz, and for four it is around 110kHz. At high sample rates disabling interrupts for the duration can reduce the risk of sporadic data loss.

The ADCAll Object

Instantiating this changes all masked ADC pins to analog inputs. The preprocessed MCU temperature, VREF and VBAT data can be accessed on ADC channels 16, 17 and 18 respectively. Appropriate scaling is handled according to reference voltage used (usually 3.3V). The temperature sensor on the chip is factory calibrated and allows to read the die temperature to +/- 1 degree centigrade. Although this sounds pretty accurate, don't forget that the MCU's internal temperature is measured. Depending on processing loads and I/O subsystems active the die temperature may easily be tens of degrees above ambient temperature. On the other hand a pyboard woken up after a long standby period will show correct ambient temperature within limits mentioned above.

The `ADCAll` `read_core_vbat()`, `read_vref()` and `read_core_vref()` methods read the backup battery voltage, reference voltage and the (1.21V nominal) reference voltage using the actual supply as a reference. All results are floating point numbers giving direct voltage values.

`read_core_vbat()` returns the voltage of the backup battery. This voltage is also adjusted according to the actual supply voltage. To avoid analog input overload the battery voltage is measured via a voltage divider and scaled according to the divider value. To prevent excessive loads to the backup battery, the voltage divider is only active during ADC conversion.

`read_vref()` is evaluated by measuring the internal voltage reference and backscale it using factory calibration value of the internal voltage reference. In most cases the reading would be close to 3.3V. If the pyboard is operated from a battery, the supply voltage may drop to values below 3.3V. The pyboard will still operate fine as long as the operating conditions are met. With proper settings of MCU clock, flash access speed and programming mode it is possible to run the pyboard down to 2 V and still get useful ADC conversion.

It is very important to make sure analog input voltages never exceed actual supply voltage.

Other analog input channels (0..15) will return unscaled integer values according to the selected precision.

To avoid unwanted activation of analog inputs (channel 0..15) a second parameter can be specified. This parameter is a binary pattern where each requested analog input has the corresponding bit set. The default value is 0xffffffff which means all analog inputs are active. If just the internal channels (16..18) are required, the mask value should be 0x70000.

Example:

```
adcall = pyb.ADCAll(12, 0x70000) # 12 bit resolution, internal channels
temp = adcall.read_core_temp()
```

class CAN – controller area network communication bus

CAN implements support for classic CAN (available on F4, F7 MCUs) and CAN FD (H7 series) controllers. At the physical level CAN bus consists of 2 lines: RX and TX. Note that to connect the pyboard to a CAN bus you must use a CAN transceiver to convert the CAN logic signals from the pyboard to the correct voltage levels on the bus.

Example usage for classic CAN controller in Loopback (transceiver-less) mode:

```
from pyb import CAN
can = CAN(1, CAN.LOOPBACK)
can.setfilter(0, CAN.LIST16, 0, (123, 124, 125, 126))  # set a filter to receive␣
↪messages with id=123, 124, 125 and 126
can.send('message!', 123)    # send a message with id 123
can.recv(0)                  # receive message on FIFO 0
```

Example usage for CAN FD controller with all of the possible options enabled:

```
# FD frame + BRS mode + Extended frame ID. 500 Kbit/s for arbitration phase, 1Mbit/s␣
↪for data phase.
can = CAN(1, CAN.NORMAL, baudrate=500_000, brs_baudrate=1_000_000, sample_point=80)
can.setfilter(0, CAN.RANGE, 0, (0xFFF0, 0xFFFF))
can.send('a'*64, 0xFFFF, fdf=True, brs=True, extframe=True)
can.recv(0)
```

The following CAN module functions and their arguments are available for both classic and FD CAN controllers, unless otherwise stated.

Constructors

class pyb.**CAN**(*bus*, ...)

Construct a CAN object on the given bus. *bus* can be 1-2, or `'YA'` or `'YB'`. With no additional parameters, the CAN object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See `CAN.init()` for parameters of initialisation.

The physical pins of the CAN buses are:

- `CAN(1)` is on `YA`: `(RX, TX) = (Y3, Y4) = (PB8, PB9)`
- `CAN(2)` is on `YB`: `(RX, TX) = (Y5, Y6) = (PB12, PB13)`

Methods

CAN.init(mode, prescaler=100, *, sjw=1, bs1=6, bs2=8, auto_restart=False, baudrate=0, sample_point=75, num_filter_banks=14, brs_sjw=1, brs_bs1=8, brs_bs2=3, brs_baudrate=0, brs_sample_point=75)

Initialise the CAN bus with the given parameters:

- *mode* is one of: NORMAL, LOOPBACK, SILENT, SILENT_LOOPBACK

- *prescaler* is the value by which the CAN input clock is divided to generate the nominal bit time quanta. The prescaler can be a value between 1 and 1024 inclusive for classic CAN, and between 1 and 512 inclusive for CAN FD.
- *sjw* is the resynchronisation jump width in units of time quanta for nominal bits; it can be a value between 1 and 4 inclusive for classic CAN, and between 1 and 128 inclusive for CAN FD.
- *bs1* defines the location of the sample point in units of the time quanta for nominal bits; it can be a value between 1 and 16 inclusive for classic CAN, and between 2 and 256 inclusive for CAN FD.
- *bs2* defines the location of the transmit point in units of the time quanta for nominal bits; it can be a value between 1 and 8 inclusive for classic CAN, and between 2 and 128 inclusive for CAN FD.
- *auto_restart* sets whether the controller will automatically try and restart communications after entering the bus-off state; if this is disabled then `restart()` can be used to leave the bus-off state
- *baudrate* if a baudrate other than 0 is provided, this function will try to automatically calculate the CAN nominal bit time (overriding *prescaler*, *bs1* and *bs2*) that satisfies both the *baudrate* (within .1%) and the desired *sample_point* (to the nearest 1%). For more precise control over the CAN timing, set the *prescaler*, *bs1* and *bs2* parameters directly.
- *sample_point* specifies the position of the bit sample with respect to the whole nominal bit time, expressed as an integer percentage of the nominal bit time. The default *sample_point* is 75%. This parameter is ignored unless *baudrate* is set.
- *num_filter_banks* for classic CAN, this is the number of banks that will be assigned to CAN(1), the rest of the 28 are assigned to CAN(2).

The remaining parameters are only present on boards with CAN FD support, and configure the optional CAN FD Bit Rate Switch (BRS) feature:

- *brs_prescaler* is the value by which the CAN FD input clock is divided to generate the data bit time quanta. The prescaler can be a value between 1 and 32 inclusive.
- *brs_sjw* is the resynchronisation jump width in units of time quanta for data bits; it can be a value between 1 and 16 inclusive
- *brs_bs1* defines the location of the sample point in units of the time quanta for data bits; it can be a value between 1 and 32 inclusive
- *brs_bs2* defines the location of the transmit point in units of the time quanta for data bits; it can be a value between 1 and 16 inclusive
- *brs_baudrate* if a baudrate other than 0 is provided, this function will try to automatically calculate the CAN data bit time (overriding *brs_prescaler*, *brs_bs1* and *brs_bs2*) that satisfies both the *brs_baudrate* (within .1%) and the desired *brs_sample_point* (to the nearest 1%). For more precise control over the BRS timing, set the *brs_prescaler*, *brs_bs1* and *brs_bs2* parameters directly.
- *brs_sample_point* specifies the position of the bit sample with respect to the whole nominal bit time, expressed as an integer percentage of the nominal bit time. The default *brs_sample_point* is 75%. This parameter is ignored unless *brs_baudrate* is set.

The time quanta tq is the basic unit of time for the CAN bus. tq is the CAN prescaler value divided by PCLK1 (the frequency of internal peripheral bus 1); see `pyb.freq()` to determine PCLK1.

A single bit is made up of the synchronisation segment, which is always 1 tq. Then follows bit segment 1, then bit segment 2. The sample point is after bit segment 1 finishes. The transmit point is after bit segment 2 finishes. The baud rate will be 1/bittime, where the bittime is 1 + BS1 + BS2 multiplied by the time quanta tq.

For example, with PCLK1=42MHz, prescaler=100, sjw=1, bs1=6, bs2=8, the value of tq is 2.38 microseconds. The bittime is 35.7 microseconds, and the baudrate is 28kHz.

See page 680 of the STM32F405 datasheet for more details.

CAN.**deinit**()

Turn off the CAN bus.

CAN.**restart**()

Force a software restart of the CAN controller without resetting its configuration.

If the controller enters the bus-off state then it will no longer participate in bus activity. If the controller is not configured to automatically restart (see `init()`) then this method can be used to trigger a restart, and the controller will follow the CAN protocol to leave the bus-off state and go into the error active state.

CAN.**state**()

Return the state of the controller. The return value can be one of:

- `CAN.STOPPED` – the controller is completely off and reset;
- `CAN.ERROR_ACTIVE` – the controller is on and in the Error Active state (both TEC and REC are less than 96);
- `CAN.ERROR_WARNING` – the controller is on and in the Error Warning state (at least one of TEC or REC is 96 or greater);
- `CAN.ERROR_PASSIVE` – the controller is on and in the Error Passive state (at least one of TEC or REC is 128 or greater);
- `CAN.BUS_OFF` – the controller is on but not participating in bus activity (TEC overflowed beyond 255).

CAN.**info**([*list*])

Get information about the controller's error states and TX and RX buffers. If *list* is provided then it should be a list object with at least 8 entries, which will be filled in with the information. Otherwise a new list will be created and filled in. In both cases the return value of the method is the populated list.

The values in the list are:

- TEC value
- REC value
- number of times the controller enterted the Error Warning state (wrapped around to 0 after 65535)
- number of times the controller enterted the Error Passive state (wrapped around to 0 after 65535)
- number of times the controller enterted the Bus Off state (wrapped around to 0 after 65535)
- number of pending TX messages
- number of pending RX messages on fifo 0
- number of pending RX messages on fifo 1

CAN.**setfilter**(*bank*, *mode*, *fifo*, *params*, *, *rtr*, *extframe=False*)

Configure a filter bank:

- *bank* is the classic CAN controller filter bank, or CAN FD filter index, to configure.
- *mode* is the mode the filter should operate in, see the tables below.
- *fifo* is which fifo (0 or 1) a message should be stored in, if it is accepted by this filter.
- *params* is an array of values the defines the filter. The contents of the array depends on the *mode* argument.

mode	Contents of *params* array for classic CAN controller
CAN.LIST16	Four 16 bit ids that will be accepted
CAN.LIST32	Two 32 bit ids that will be accepted
CAN.MASK16	**Two 16 bit id/mask pairs. E.g. (1, 3, 4, 4)** The first pair, 1 and 3 will accept all ids that have bit 0 = 1 and bit 1 = 0. The second pair, 4 and 4, will accept all ids that have bit 2 = 1.
CAN.MASK32	As with CAN.MASK16 but with only one 32 bit id/mask pair.

mode	Contents of *params* array for CAN FD controller
CAN.RANGE	Two ids that represent a range of accepted ids.
CAN.DUAL	Two ids that will be accepted. For example (1, 2)
CAN.MASK	One filter ID and a mask. For example (0x111, 0x7FF)

- *rtr* For classic CAN controllers, this is an array of booleans that states if a filter should accept a remote transmission request message. If this argument is not given then it defaults to `False` for all entries. The length of the array depends on the *mode* argument. For CAN FD, this argument is ignored.

mode	length of *rtr* array
CAN.LIST16	4
CAN.LIST32	2
CAN.MASK16	2
CAN.MASK32	1

- *extframe* If True the frame will have an extended identifier (29 bits), otherwise a standard identifier (11 bits) is used.

`CAN.`**`clearfilter`**(*bank*, *extframe=False*)

Clear and disables a filter bank:

- *bank* is the classic CAN controller filter bank, or CAN FD filter index, to clear.
- *extframe* For CAN FD controllers, if True, clear an extended filter (configured with extframe=True), otherwise the clear a standard identifier (configured with extframe=False).

`CAN.`**`any`**(*fifo*)

Return `True` if any message waiting on the FIFO, else `False`.

`CAN.`**`recv`**(*fifo*, *list=None*, ***, *timeout=5000*)

Receive data on the bus:

- *fifo* is an integer, which is the FIFO to receive on
- *list* is an optional list object to be used as the return value

- *timeout* is the timeout in milliseconds to wait for the receive.

Return value: A list containing five values.

- The id of the message.
- A boolean that indicates if the message ID is standard or extended.
- A boolean that indicates if the message is an RTR message.
- The FMI (Filter Match Index) value.
- An array containing the data.

If *list* is `None` then a new list will be allocated, as well as a new bytes object to contain the data (as the fifth element in the list).

If *list* is not `None` then it should be a list object with a least five elements. The fifth element should be a memoryview object which is created from either a bytearray or an array of type 'B' or 'b', and this array must have enough room for at least 8 bytes. The list object will then be populated with the first four return values above, and the memoryview object will be resized inplace to the size of the data and filled in with that data. The same list and memoryview objects can be reused in subsequent calls to this method, providing a way of receiving data without using the heap. For example:

```
buf = bytearray(8)
lst = [0, 0, 0, 0, memoryview(buf)]
# No heap memory is allocated in the following call
can.recv(0, lst)
```

`CAN.`**`send`**(*data*, *id*, *, *timeout=0*, *rtr=False*, *extframe=False*, *fdf=False*, *brs=False*)

Send a message on the bus:

- *data* is the data to send (an integer to send, or a buffer object).
- *id* is the id of the message to be sent.
- *timeout* is the timeout in milliseconds to wait for the send.
- *rtr* is a boolean that specifies if the message shall be sent as a remote transmission request. If *rtr* is True then only the length of *data* is used to fill in the DLC slot of the frame; the actual bytes in *data* are unused.
- *extframe* if True the frame will have an extended identifier (29 bits), otherwise a standard identifier (11 bits) is used.
- *fdf* for CAN FD controllers, if set to True, the frame will have an FD frame format, which supports data payloads up to 64 bytes.
- *brs* for CAN FD controllers, if set to True, the bitrate switching mode is enabled, in which the data phase is transmitted at a different bitrate. See `CAN.init()` for the data bit timing configuration parameters.

If timeout is 0 the message is placed in a buffer in one of three hardware buffers and the method returns immediately. If all three buffers are in use an exception is thrown. If timeout is not 0, the method waits until the message is transmitted. If the message can't be transmitted within the specified time an exception is thrown.

Return value: `None`.

`CAN.`**`rxcallback`**(*fifo*, *fun*)

Register a function to be called when a message is accepted into a empty fifo:

- *fifo* is the receiving fifo.

- *fun* is the function to be called when the fifo becomes non empty.

The callback function takes two arguments the first is the can object it self the second is a integer that indicates the reason for the callback.

Reason	
0	A message has been accepted into a empty FIFO.
1	The FIFO is full
2	A message has been lost due to a full FIFO

Example use of rxcallback:

```
def cb0(bus, reason):
  print('cb0')
  if reason == 0:
      print('pending')
  if reason == 1:
      print('full')
  if reason == 2:
      print('overflow')

can = CAN(1, CAN.LOOPBACK)
can.rxcallback(0, cb0)
```

Constants

CAN.**NORMAL**

CAN.**LOOPBACK**

CAN.**SILENT**

CAN.**SILENT_LOOPBACK**

The mode of the CAN bus used in `init()`.

CAN.**STOPPED**

CAN.**ERROR_ACTIVE**

CAN.**ERROR_WARNING**

CAN.**ERROR_PASSIVE**

CAN.**BUS_OFF**

Possible states of the CAN controller returned from *state()*.

CAN.**LIST16**

CAN.**MASK16**

CAN.**LIST32**

CAN.**MASK32**

The operation mode of a filter used in *setfilter()* for classic CAN.

CAN.**DUAL**

CAN.**RANGE**

CAN.**MASK**

The operation mode of a filter used in *setfilter()* for CAN FD.

class DAC – digital to analog conversion

The DAC is used to output analog values (a specific voltage) on pin X5 or pin X6. The voltage will be between 0 and 3.3V.

This module will undergo changes to the API.

Example usage:

```
from pyb import DAC

dac = DAC(1)            # create DAC 1 on pin X5
dac.write(128)          # write a value to the DAC (makes X5 1.65V)

dac = DAC(1, bits=12)   # use 12 bit resolution
dac.write(4095)         # output maximum value, 3.3V
```

To output a continuous sine-wave:

```
import math
from pyb import DAC

# create a buffer containing a sine-wave
buf = bytearray(100)
for i in range(len(buf)):
    buf[i] = 128 + int(127 * math.sin(2 * math.pi * i / len(buf)))

# output the sine-wave at 400Hz
dac = DAC(1)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

To output a continuous sine-wave at 12-bit resolution:

```
import math
from array import array
from pyb import DAC

# create a buffer containing a sine-wave, using half-word samples
buf = array('H', 2048 + int(2047 * math.sin(2 * math.pi * i / 128)) for i in␣
↪range(128))

# output the sine-wave at 400Hz
dac = DAC(1, bits=12)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

Constructors

class pyb.**DAC**(*port*, *bits=8*, *, *buffering=None*)

Construct a new DAC object.

`port` can be a pin object, or an integer (1 or 2). DAC(1) is on pin X5 and DAC(2) is on pin X6.

`bits` is an integer specifying the resolution, and can be 8 or 12. The maximum value for the write and write_timed methods will be 2**``bits``-1.

The *buffering* parameter selects the behaviour of the DAC op-amp output buffer, whose purpose is to reduce the output impedance. It can be `None` to select the default (buffering enabled for *DAC.noise()*, *DAC.triangle()*

and *DAC.write_timed()*, and disabled for *DAC.write()*), `False` to disable buffering completely, or `True` to enable output buffering.

When buffering is enabled the DAC pin can drive loads down to 5KΩ. Otherwise it has an output impedance of 15KΩ maximum: consequently to achieve a 1% accuracy without buffering requires the applied load to be less than 1.5MΩ. Using the buffer incurs a penalty in accuracy, especially near the extremes of range.

Methods

DAC.**init**(*bits=8*, ***, *buffering=None*)

Reinitialise the DAC. *bits* can be 8 or 12. *buffering* can be `None`, `False` or `True`; see above constructor for the meaning of this parameter.

DAC.**deinit**()

De-initialise the DAC making its pin available for other uses.

DAC.**noise**(*freq*)

Generate a pseudo-random noise signal. A new random sample is written to the DAC output at the given frequency.

DAC.**triangle**(*freq*)

Generate a triangle wave. The value on the DAC output changes at the given frequency and ramps through the full 12-bit range (up and down). Therefore the frequency of the repeating triangle wave itself is 8192 times smaller.

DAC.**write**(*value*)

Direct access to the DAC output. The minimum value is 0. The maximum value is 2**``bits``-1, where `bits` is set when creating the DAC object or by using the `init` method.

DAC.**write_timed**(*data*, *freq*, ***, *mode=DAC.NORMAL*)

Initiates a burst of RAM to DAC using a DMA transfer. The input data is treated as an array of bytes in 8-bit mode, and an array of unsigned half-words (array typecode 'H') in 12-bit mode.

`freq` can be an integer specifying the frequency to write the DAC samples at, using Timer(6). Or it can be an already-initialised Timer object which is used to trigger the DAC sample. Valid timers are 2, 4, 5, 6, 7 and 8.

`mode` can be `DAC.NORMAL` or `DAC.CIRCULAR`.

Example using both DACs at the same time:

```
dac1 = DAC(1)
dac2 = DAC(2)
dac1.write_timed(buf1, pyb.Timer(6, freq=100), mode=DAC.CIRCULAR)
dac2.write_timed(buf2, pyb.Timer(7, freq=200), mode=DAC.CIRCULAR)
```

Constants

DAC.**NORMAL**

NORMAL mode does a single transmission of the waveform in the data buffer,

DAC.**CIRCULAR**

CIRCULAR mode does a transmission of the waveform in the data buffer, and wraps around to the start of the data buffer every time it reaches the end of the table.

class ExtInt – configure I/O pins to interrupt on external events

There are a total of 22 interrupt lines. 16 of these can come from GPIO pins and the remaining 6 are from internal sources.

For lines 0 through 15, a given line can map to the corresponding line from an arbitrary port. So line 0 can map to Px0 where x is A, B, C, … and line 1 can map to Px1 where x is A, B, C, …

```
def callback(line):
    print("line =", line)
```

Note: ExtInt will automatically configure the gpio line as an input.

```
extint = pyb.ExtInt(pin, pyb.ExtInt.IRQ_FALLING, pyb.Pin.PULL_UP, callback)
```

Now every time a falling edge is seen on the X1 pin, the callback will be called. Caution: mechanical pushbuttons have "bounce" and pushing or releasing a switch will often generate multiple edges. See: http://www.eng.utah.edu/~cs5780/debouncing.pdf for a detailed explanation, along with various techniques for debouncing.

Trying to register 2 callbacks onto the same pin will throw an exception.

If pin is passed as an integer, then it is assumed to map to one of the internal interrupt sources, and must be in the range 16 through 22.

All other pin objects go through the pin mapper to come up with one of the gpio pins.

```
extint = pyb.ExtInt(pin, mode, pull, callback)
```

Valid modes are pyb.ExtInt.IRQ_RISING, pyb.ExtInt.IRQ_FALLING, pyb.ExtInt.IRQ_RISING_FALLING, pyb.ExtInt.EVT_RISING, pyb.ExtInt.EVT_FALLING, and pyb.ExtInt.EVT_RISING_FALLING.

Only the IRQ_xxx modes have been tested. The EVT_xxx modes have something to do with sleep mode and the WFE instruction.

Valid pull values are pyb.Pin.PULL_UP, pyb.Pin.PULL_DOWN, pyb.Pin.PULL_NONE.

There is also a C API, so that drivers which require EXTI interrupt lines can also use this code. See extint.h for the available functions and usrsw.h for an example of using this.

Constructors

class pyb.**ExtInt**(*pin*, *mode*, *pull*, *callback*)

Create an ExtInt object:

- `pin` is the pin on which to enable the interrupt (can be a pin object or any valid pin name).
- `mode` can be one of: - `ExtInt.IRQ_RISING` - trigger on a rising edge; - `ExtInt.IRQ_FALLING` - trigger on a falling edge; - `ExtInt.IRQ_RISING_FALLING` - trigger on a rising or falling edge.
- `pull` can be one of: - `pyb.Pin.PULL_NONE` - no pull up or down resistors; - `pyb.Pin.PULL_UP` - enable the pull-up resistor; - `pyb.Pin.PULL_DOWN` - enable the pull-down resistor.
- `callback` is the function to call when the interrupt triggers. The callback function must accept exactly 1 argument, which is the line that triggered the interrupt.

Class methods

classmethod ExtInt.**regs**()

Dump the values of the EXTI registers.

Methods

ExtInt.**disable**()

Disable the interrupt associated with the ExtInt object. This could be useful for debouncing.

ExtInt.**enable**()

Enable a disabled interrupt.

ExtInt.**line**()

Return the line number that the pin is mapped to.

ExtInt.**swint**()

Trigger the callback from software.

Constants

ExtInt.**IRQ_FALLING**

interrupt on a falling edge

ExtInt.**IRQ_RISING**

interrupt on a rising edge

ExtInt.**IRQ_RISING_FALLING**

interrupt on a rising or falling edge

class Flash – access to built-in flash storage

The Flash class allows direct access to the primary flash device on the pyboard.

In most cases, to store persistent data on the device, you'll want to use a higher-level abstraction, for example the filesystem via Python's standard file API, but this interface is useful to *customise the filesystem configuration* or implement a low-level storage system for your application.

Constructors

class pyb.**Flash**

Create and return a block device that represents the flash device presented to the USB mass storage interface.

It includes a virtual partition table at the start, and the actual flash starts at block `0x100`.

This constructor is deprecated and will be removed in a future version of MicroPython.

class pyb.**Flash**(*, *start=-1*, *len=-1*)

Create and return a block device that accesses the flash at the specified offset. The length defaults to the remaining size of the device.

The *start* and *len* offsets are in bytes, and must be a multiple of the block size (typically 512 for internal flash).

Methods

Flash.**readblocks**(*block_num*, *buf*)

Flash.**readblocks**(*block_num*, *buf*, *offset*)

Flash.**writeblocks**(*block_num*, *buf*)

Flash.**writeblocks**(*block_num*, *buf*, *offset*)

Flash.**ioctl**(*cmd*, *arg*)

These methods implement the simple and *extended* block protocol defined by `vfs.AbstractBlockDev`.

Hardware Note

On boards with external spiflash (e.g. Pyboard D), the MicroPython firmware will be configured to use that as the primary flash storage. On all other boards, the internal flash inside the *MCU* will be used.

class I2C – a two-wire serial protocol

I2C is a two-wire protocol for communicating between devices. At the physical level it consists of 2 wires: SCL and SDA, the clock and data lines respectively.

I2C objects are created attached to a specific bus. They can be initialised when created, or initialised later on.

Example:

```
from pyb import I2C

i2c = I2C(1)                               # create on bus 1
i2c = I2C(1, I2C.CONTROLLER)               # create and init as a controller
i2c.init(I2C.CONTROLLER, baudrate=20000) # init as a controller
i2c.init(I2C.PERIPHERAL, addr=0x42)      # init as a peripheral with given address
i2c.deinit()                               # turn off the I2C unit
```

Printing the i2c object gives you information about its configuration.

The basic methods are send and recv:

```
i2c.send('abc')      # send 3 bytes
i2c.send(0x42)       # send a single byte, given by the number
data = i2c.recv(3)   # receive 3 bytes
```

To receive inplace, first create a bytearray:

```
data = bytearray(3)  # create a buffer
i2c.recv(data)       # receive 3 bytes, writing them into data
```

You can specify a timeout (in ms):

```
i2c.send(b'123', timeout=2000)   # timeout after 2 seconds
```

A controller must specify the recipient's address:

```
i2c.init(I2C.CONTROLLER)
i2c.send('123', 0x42)        # send 3 bytes to peripheral with address 0x42
i2c.send(b'456', addr=0x42)  # keyword for address
```

Master also has other methods:

```
i2c.is_ready(0x42)           # check if peripheral 0x42 is ready
i2c.scan()                   # scan for peripherals on the bus, returning
                             #   a list of valid addresses
i2c.mem_read(3, 0x42, 2)     # read 3 bytes from memory of peripheral 0x42,
                             #   starting at address 2 in the peripheral
i2c.mem_write('abc', 0x42, 2, timeout=1000) # write 'abc' (3 bytes) to memory of␣
↪peripheral 0x42
                                            # starting at address 2 in the peripheral,
↪ timeout after 1 second
```

Constructors

class pyb.**I2C**(*bus*, ...)

Construct an I2C object on the given bus. `bus` can be 1 or 2, 'X' or 'Y'. With no additional parameters, the I2C object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See `init` for parameters of initialisation.

The physical pins of the I2C buses on Pyboards V1.0 and V1.1 are:

- `I2C(1)` is on the X position: `(SCL, SDA) = (X9, X10) = (PB6, PB7)`
- `I2C(2)` is on the Y position: `(SCL, SDA) = (Y9, Y10) = (PB10, PB11)`

On the Pyboard Lite:

- `I2C(1)` is on the X position: `(SCL, SDA) = (X9, X10) = (PB6, PB7)`
- `I2C(3)` is on the Y position: `(SCL, SDA) = (Y9, Y10) = (PA8, PB8)`

Calling the constructor with 'X' or 'Y' enables portability between Pyboard types.

Methods

I2C.**deinit**()

Turn off the I2C bus.

I2C.**init**(*mode*, *, *addr=0x12*, *baudrate=400000*, *gencall=False*, *dma=False*)

Initialise the I2C bus with the given parameters:

- `mode` must be either `I2C.CONTROLLER` or `I2C.PERIPHERAL`
- `addr` is the 7-bit address (only sensible for a peripheral)
- `baudrate` is the SCL clock rate (only sensible for a controller)
- `gencall` is whether to support general call mode
- `dma` is whether to allow the use of DMA for the I2C transfers (note that DMA transfers have more precise timing but currently do not handle bus errors properly)

The actual clock frequency may be lower than the requested frequency. This is dependent on the platform hardware. The actual rate may be determined by printing the I2C object.

I2C.**is_ready**(*addr*)

Check if an I2C device responds to the given address. Only valid when in controller mode.

I2C.**mem_read**(*data*, *addr*, *memaddr*, *, *timeout=5000*, *addr_size=8*)

Read from the memory of an I2C device:

- `data` can be an integer (number of bytes to read) or a buffer to read into
- `addr` is the I2C device address
- `memaddr` is the memory location within the I2C device
- `timeout` is the timeout in milliseconds to wait for the read
- `addr_size` selects width of memaddr: 8 or 16 bits

Returns the read data. This is only valid in controller mode.

I2C.**mem_write**(*data*, *addr*, *memaddr*, *, *timeout=5000*, *addr_size=8*)

Write to the memory of an I2C device:

- `data` can be an integer or a buffer to write from
- `addr` is the I2C device address
- `memaddr` is the memory location within the I2C device
- `timeout` is the timeout in milliseconds to wait for the write
- `addr_size` selects width of memaddr: 8 or 16 bits

Returns `None`. This is only valid in controller mode.

I2C.**recv**(*recv*, *addr=0x00*, *, *timeout=5000*)

Receive data on the bus:

- `recv` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes
- `addr` is the address to receive from (only required in controller mode)
- `timeout` is the timeout in milliseconds to wait for the receive

Return value: if `recv` is an integer then a new buffer of the bytes received, otherwise the same buffer that was passed in to `recv`.

I2C.**send**(*send*, *addr=0x00*, *, *timeout=5000*)

Send data on the bus:

- `send` is the data to send (an integer to send, or a buffer object)
- `addr` is the address to send to (only required in controller mode)
- `timeout` is the timeout in milliseconds to wait for the send

Return value: `None`.

I2C.**scan**()

Scan all I2C addresses from 0x01 to 0x7f and return a list of those that respond. Only valid when in controller mode.

Constants

I2C.**CONTROLLER**

for initialising the bus to controller mode

I2C.**PERIPHERAL**

for initialising the bus to peripheral mode

class LCD – LCD control for the LCD touch-sensor pyskin

The LCD class is used to control the LCD on the LCD touch-sensor pyskin, LCD32MKv1.0. The LCD is a 128x32 pixel monochrome screen, part NHD-C12832A1Z.

The pyskin must be connected in either the X or Y positions, and then an LCD object is made using:

```
lcd = pyb.LCD('X')      # if pyskin is in the X position
lcd = pyb.LCD('Y')      # if pyskin is in the Y position
```

Then you can use:

```
lcd.light(True)                 # turn the backlight on
lcd.write('Hello world!\n')     # print text to the screen
```

This driver implements a double buffer for setting/getting pixels. For example, to make a bouncing dot, try:

```
x = y = 0
dx = dy = 1
while True:
    # update the dot's position
    x += dx
    y += dy

    # make the dot bounce of the edges of the screen
    if x <= 0 or x >= 127: dx = -dx
    if y <= 0 or y >= 31: dy = -dy

    lcd.fill(0)                 # clear the buffer
    lcd.pixel(x, y, 1)          # draw the dot
    lcd.show()                  # show the buffer
    pyb.delay(50)               # pause for 50ms
```

Constructors

class pyb.**LCD**(*skin_position*)

Construct an LCD object in the given skin position. `skin_position` can be 'X' or 'Y', and should match the position where the LCD pyskin is plugged in.

Methods

LCD.**command**(*instr_data*, *buf*)

Send an arbitrary command to the LCD. Pass 0 for `instr_data` to send an instruction, otherwise pass 1 to send data. `buf` is a buffer with the instructions/data to send.

LCD.**contrast**(*value*)

Set the contrast of the LCD. Valid values are between 0 and 47.

LCD.**fill**(*colour*)

Fill the screen with the given colour (0 or 1 for white or black).

This method writes to the hidden buffer. Use `show()` to show the buffer.

LCD.**get**(*x*, *y*)

Get the pixel at the position `(x, y)`. Returns 0 or 1.

This method reads from the visible buffer.

LCD.**light**(*value*)

Turn the backlight on/off. True or 1 turns it on, False or 0 turns it off.

LCD.**pixel**(*x*, *y*, *colour*)

Set the pixel at `(x, y)` to the given colour (0 or 1).

This method writes to the hidden buffer. Use `show()` to show the buffer.

LCD.**show**()

Show the hidden buffer on the screen.

LCD.**text**(*str*, *x*, *y*, *colour*)

Draw the given text to the position `(x, y)` using the given colour (0 or 1).

This method writes to the hidden buffer. Use `show()` to show the buffer.

LCD.**write**(*str*)

Write the string `str` to the screen. It will appear immediately.

class LED – LED object

The LED object controls an individual LED (Light Emitting Diode).

Constructors

class pyb.**LED**(*id*)

Create an LED object associated with the given LED:

- `id` is the LED number, 1-4.

Methods

LED.**intensity**([*value*])

Get or set the LED intensity. Intensity ranges between 0 (off) and 255 (full on). If no argument is given, return the LED intensity. If an argument is given, set the LED intensity and return `None`.

Note: Only LED(3) and LED(4) can have a smoothly varying intensity, and they use timer PWM to implement it. LED(3) uses Timer(2) and LED(4) uses Timer(3). These timers are only configured for PWM if the intensity of the relevant LED is set to a value between 1 and 254. Otherwise the timers are free for general purpose use.

LED.**off**()

Turn the LED off.

LED.**on**()

Turn the LED on, to maximum intensity.

LED.**toggle**()

Toggle the LED between on (maximum intensity) and off. If the LED is at non-zero intensity then it is considered "on" and toggle will turn it off.

class Pin – control I/O pins

A pin is the basic object to control I/O pins. It has methods to set the mode of the pin (input, output, etc) and methods to get and set the digital logic level. For analog control of a pin, see the ADC class.

Usage Model:

All Board Pins are predefined as pyb.Pin.board.Name:

```
x1_pin = pyb.Pin.board.X1

g = pyb.Pin(pyb.Pin.board.X1, pyb.Pin.IN)
```

CPU pins which correspond to the board pins are available as `pyb.Pin.cpu.Name`. For the CPU pins, the names are the port letter followed by the pin number. On the PYBv1.0, `pyb.Pin.board.X1` and `pyb.Pin.cpu.A0` are the same pin.

You can also use strings:

```
g = pyb.Pin('X1', pyb.Pin.OUT_PP)
```

Users can add their own names:

```
MyMapperDict = { 'LeftMotorDir' : pyb.Pin.cpu.C12 }
pyb.Pin.dict(MyMapperDict)
g = pyb.Pin("LeftMotorDir", pyb.Pin.OUT_OD)
```

and can query mappings:

```
pin = pyb.Pin("LeftMotorDir")
```

Users can also add their own mapping function:

```
def MyMapper(pin_name):
   if pin_name == "LeftMotorDir":
       return pyb.Pin.cpu.A0
```

(continues on next page)

(continued from previous page)

```
pyb.Pin.mapper(MyMapper)
```

So, if you were to call: `pyb.Pin("LeftMotorDir", pyb.Pin.OUT_PP)` then `"LeftMotorDir"` is passed directly to the mapper function.

To summarise, the following order determines how things get mapped into an ordinal pin number:

1. Directly specify a pin object
2. User supplied mapping function
3. User supplied mapping (object must be usable as a dictionary key)
4. Supply a string which matches a board pin
5. Supply a string which matches a CPU port/pin

You can set `pyb.Pin.debug(True)` to get some debug information about how a particular object gets mapped to a pin.

All pin objects go through the pin mapper to come up with one of the gpio pins.

Constructors

class pyb.**Pin**(*id*, ...)

Create a new Pin object associated with the id. If additional arguments are given, they are used to initialise the pin. See `pin.init()`.

Class methods

classmethod Pin.**debug**([*state*])

Get or set the debugging state (`True` or `False` for on or off).

classmethod Pin.**dict**([*dict*])

Get or set the pin mapper dictionary.

classmethod Pin.**mapper**([*fun*])

Get or set the pin mapper function.

Methods

Pin.**init**(*mode*, *pull=Pin.PULL_NONE*, ***, *value=None*, *alt=-1*)

Initialise the pin:

- *mode* can be one of:
  - `Pin.IN` - configure the pin for input;
  - `Pin.OUT_PP` - configure the pin for output, with push-pull control;
  - `Pin.OUT_OD` - configure the pin for output, with open-drain control;
  - `Pin.ALT` - configure the pin for alternate function, input or output;
  - `Pin.AF_PP` - configure the pin for alternate function, push-pull;

  - `Pin.AF_OD` - configure the pin for alternate function, open-drain;
  - `Pin.ANALOG` - configure the pin for analog.
- *pull* can be one of:
  - `Pin.PULL_NONE` - no pull up or down resistors;
  - `Pin.PULL_UP` - enable the pull-up resistor;
  - `Pin.PULL_DOWN` - enable the pull-down resistor.

  When a pin has the `Pin.PULL_UP` or `Pin.PULL_DOWN` pull-mode enabled, that pin has an effective 40k Ohm resistor pulling it to 3V3 or GND respectively (except pin Y5 which has 11k Ohm resistors).
- *value* if not None will set the port output value before enabling the pin.
- *alt* can be used when mode is `Pin.ALT` , `Pin.AF_PP` or `Pin.AF_OD` to set the index or name of one of the alternate functions associated with a pin. This arg was previously called *af* which can still be used if needed.

Returns: `None`.

`Pin.`**`value`**`(`[*value*]`)`

Get or set the digital logic level of the pin:

- With no argument, return 0 or 1 depending on the logic level of the pin.
- With `value` given, set the logic level of the pin. `value` can be anything that converts to a boolean. If it converts to `True`, the pin is set high, otherwise it is set low.

`Pin.`**`__str__`**`()`

Return a string describing the pin object.

`Pin.`**`af`**`()`

Returns the currently configured alternate-function of the pin. The integer returned will match one of the allowed constants for the af argument to the init function.

`Pin.`**`af_list`**`()`

Returns an array of alternate functions available for this pin.

`Pin.`**`gpio`**`()`

Returns the base address of the GPIO block associated with this pin.

`Pin.`**`mode`**`()`

Returns the currently configured mode of the pin. The integer returned will match one of the allowed constants for the mode argument to the init function.

`Pin.`**`name`**`()`

Get the pin name.

`Pin.`**`names`**`()`

Returns the cpu and board names for this pin.

`Pin.`**`pin`**`()`

Get the pin number.

`Pin.`**`port`**`()`

Get the pin port.

`Pin.`**`pull`**`()`

Returns the currently configured pull of the pin. The integer returned will match one of the allowed constants for the pull argument to the init function.

Constants

`Pin.`**`ALT`**
initialise the pin to alternate-function mode for input or output

`Pin.`**`AF_OD`**
initialise the pin to alternate-function mode with an open-drain drive

`Pin.`**`AF_PP`**
initialise the pin to alternate-function mode with a push-pull drive

`Pin.`**`ANALOG`**
initialise the pin to analog mode

`Pin.`**`IN`**
initialise the pin to input mode

`Pin.`**`OUT_OD`**
initialise the pin to output mode with an open-drain drive

`Pin.`**`OUT_PP`**
initialise the pin to output mode with a push-pull drive

`Pin.`**`PULL_DOWN`**
enable the pull-down resistor on the pin

`Pin.`**`PULL_NONE`**
don't enable any pull up or down resistors on the pin

`Pin.`**`PULL_UP`**
enable the pull-up resistor on the pin

class PinAF – Pin Alternate Functions

A Pin represents a physical pin on the microprocessor. Each pin can have a variety of functions (GPIO, I2C SDA, etc). Each PinAF object represents a particular function for a pin.

Usage Model:

```
x3 = pyb.Pin.board.X3
x3_af = x3.af_list()
```

x3_af will now contain an array of PinAF objects which are available on pin X3.

For the pyboard, x3_af would contain:
[Pin.AF1_TIM2, Pin.AF2_TIM5, Pin.AF3_TIM9, Pin.AF7_USART2]

Normally, each peripheral would configure the alternate function automatically, but sometimes the same function is available on multiple pins, and having more control is desired.

To configure X3 to expose TIM2_CH3, you could use:

```
pin = pyb.Pin(pyb.Pin.board.X3, mode=pyb.Pin.ALT, alt=pyb.Pin.AF1_TIM2)
```

or:

```
pin = pyb.Pin(pyb.Pin.board.X3, mode=pyb.Pin.ALT, alt=1)
```

Methods

pinaf.**__str__**()

Return a string describing the alternate function.

pinaf.**index**()

Return the alternate function index.

pinaf.**name**()

Return the name of the alternate function.

pinaf.**reg**()

Return the base register associated with the peripheral assigned to this alternate function. For example, if the alternate function were TIM2_CH3 this would return stm.TIM2

class RTC – real time clock

The RTC is an independent clock that keeps track of the date and time.

Example usage:

```
rtc = pyb.RTC()
rtc.datetime((2014, 5, 1, 4, 13, 0, 0, 0))
print(rtc.datetime())
```

Constructors

class pyb.**RTC**

Create an RTC object.

Methods

RTC.**datetime**([*datetimetuple*])

Get or set the date and time of the RTC.

With no arguments, this method returns an 8-tuple with the current date and time. With 1 argument (being an 8-tuple) it sets the date and time (and `subseconds` is reset to 255).

The 8-tuple has the following format:

(year, month, day, weekday, hours, minutes, seconds, subseconds)

`weekday` is 1-7 for Monday through Sunday.

`subseconds` counts down from 255 to 0

RTC.**wakeup**(*timeout*, *callback=None*)

Set the RTC wakeup timer to trigger repeatedly at every `timeout` milliseconds. This trigger can wake the pyboard from both the sleep states: *pyb.stop()* and *pyb.standby()*.

If `timeout` is `None` then the wakeup timer is disabled.

If `callback` is given then it is executed at every trigger of the wakeup timer. `callback` must take exactly one argument.

RTC.**info**()

Get information about the startup time and reset source.

- The lower 0xffff are the number of milliseconds the RTC took to start up.
- Bit 0x10000 is set if a power-on reset occurred.
- Bit 0x20000 is set if an external reset occurred

RTC.**calibration**(*cal*)

Get or set RTC calibration.

With no arguments, `calibration()` returns the current calibration value, which is an integer in the range [-511 : 512]. With one argument it sets the RTC calibration.

The RTC Smooth Calibration mechanism adjusts the RTC clock rate by adding or subtracting the given number of ticks from the 32768 Hz clock over a 32 second period (corresponding to 2^20 clock ticks.) Each tick added will speed up the clock by 1 part in 2^20, or 0.954 ppm; likewise the RTC clock it slowed by negative values. The usable calibration range is: (-511 * 0.954) ~= -487.5 ppm up to (512 * 0.954) ~= 488.5 ppm

class Servo – 3-wire hobby servo driver

Servo objects control standard hobby servo motors with 3-wires (ground, power, signal). There are 4 positions on the pyboard where these motors can be plugged in: pins X1 through X4 are the signal pins, and next to them are 4 sets of power and ground pins.

Example usage:

```
import pyb

s1 = pyb.Servo(1)    # create a servo object on position X1
s2 = pyb.Servo(2)    # create a servo object on position X2

s1.angle(45)         # move servo 1 to 45 degrees
s2.angle(0)          # move servo 2 to 0 degrees

# move servo1 and servo2 synchronously, taking 1500ms
s1.angle(-60, 1500)
s2.angle(30, 1500)
```

Note: The Servo objects use Timer(5) to produce the PWM output. You can use Timer(5) for Servo control, or your own purposes, but not both at the same time.

Constructors

class pyb.**Servo**(*id*)

Create a servo object. `id` is 1-4, and corresponds to pins X1 through X4.

Methods

Servo.**angle**([*angle*, *time=0*])

If no arguments are given, this function returns the current angle.

If arguments are given, this function sets the angle of the servo:

- `angle` is the angle to move to in degrees.
- `time` is the number of milliseconds to take to get to the specified angle. If omitted, then the servo moves as quickly as possible to its new position.

Servo.**speed**([*speed*, *time=0*])

If no arguments are given, this function returns the current speed.

If arguments are given, this function sets the speed of the servo:

- `speed` is the speed to change to, between -100 and 100.
- `time` is the number of milliseconds to take to get to the specified speed. If omitted, then the servo accelerates as quickly as possible.

Servo.**pulse_width**([*value*])

If no arguments are given, this function returns the current raw pulse-width value.

If an argument is given, this function sets the raw pulse-width value.

Servo.**calibration**([*pulse_min*, *pulse_max*, *pulse_centre*[, *pulse_angle_90*, *pulse_speed_100*]])

If no arguments are given, this function returns the current calibration data, as a 5-tuple.

If arguments are given, this function sets the timing calibration:

- `pulse_min` is the minimum allowed pulse width.
- `pulse_max` is the maximum allowed pulse width.
- `pulse_centre` is the pulse width corresponding to the centre/zero position.
- `pulse_angle_90` is the pulse width corresponding to 90 degrees.
- `pulse_speed_100` is the pulse width corresponding to a speed of 100.

class SPI – a controller-driven serial protocol

SPI is a serial protocol that is driven by a controller. At the physical level there are 3 lines: SCK, MOSI, MISO.

See usage model of I2C; SPI is very similar. Main difference is parameters to init the SPI bus:

```
from pyb import SPI
spi = SPI(1, SPI.CONTROLLER, baudrate=600000, polarity=1, phase=0, crc=0x7)
```

Only required parameter is mode, SPI.CONTROLLER or SPI.PERIPHERAL. Polarity can be 0 or 1, and is the level the idle clock line sits at. Phase can be 0 or 1 to sample data on the first or second clock edge respectively. Crc can be None for no CRC, or a polynomial specifier.

Additional methods for SPI:

```
data = spi.send_recv(b'1234')        # send 4 bytes and receive 4 bytes
buf = bytearray(4)
spi.send_recv(b'1234', buf)          # send 4 bytes and receive 4 into buf
spi.send_recv(buf, buf)              # send/recv 4 bytes from/to buf
```

Constructors

class pyb.**SPI**(*bus*, ...)

Construct an SPI object on the given bus. `bus` can be 1 or 2, or 'X' or 'Y'. With no additional parameters, the SPI object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See `init` for parameters of initialisation.

The physical pins of the SPI buses are:

- `SPI(1)` is on the X position: `(NSS, SCK, MISO, MOSI) = (X5, X6, X7, X8) = (PA4, PA5, PA6, PA7)`
- `SPI(2)` is on the Y position: `(NSS, SCK, MISO, MOSI) = (Y5, Y6, Y7, Y8) = (PB12, PB13, PB14, PB15)`

At the moment, the NSS pin is not used by the SPI driver and is free for other use.

Methods

SPI.**deinit**()

Turn off the SPI bus.

SPI.**init**(*mode*, *baudrate=328125*, ***, *prescaler=-1*, *polarity=1*, *phase=0*, *bits=8*, *firstbit=SPI.MSB*, *ti=False*, *crc=None*)

Initialise the SPI bus with the given parameters:

- `mode` must be either `SPI.CONTROLLER` or `SPI.PERIPHERAL`.
- `baudrate` is the SCK clock rate (only sensible for a controller).
- `prescaler` is the prescaler to use to derive SCK from the APB bus frequency; use of `prescaler` overrides `baudrate`.
- `polarity` can be 0 or 1, and is the level the idle clock line sits at.
- `phase` can be 0 or 1 to sample data on the first or second clock edge respectively.
- `bits` can be 8 or 16, and is the number of bits in each transferred word.
- `firstbit` can be `SPI.MSB` or `SPI.LSB`.
- `ti` True indicates Texas Instruments, as opposed to Motorola, signal conventions.
- `crc` can be None for no CRC, or a polynomial specifier.

Note that the SPI clock frequency will not always be the requested baudrate. The hardware only supports baudrates that are the APB bus frequency (see *pyb.freq()*) divided by a prescaler, which can be 2, 4, 8, 16, 32, 64, 128

or 256. SPI(1) is on AHB2, and SPI(2) is on AHB1. For precise control over the SPI clock frequency, specify `prescaler` instead of `baudrate`.

Printing the SPI object will show you the computed baudrate and the chosen prescaler.

`SPI.`**`recv`**(*recv*, *, *timeout=5000*)

Receive data on the bus:

- `recv` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: if `recv` is an integer then a new buffer of the bytes received, otherwise the same buffer that was passed in to `recv`.

`SPI.`**`send`**(*send*, *, *timeout=5000*)

Send data on the bus:

- `send` is the data to send (an integer to send, or a buffer object).
- `timeout` is the timeout in milliseconds to wait for the send.

Return value: `None`.

`SPI.`**`send_recv`**(*send*, *recv=None*, *, *timeout=5000*)

Send and receive data on the bus at the same time:

- `send` is the data to send (an integer to send, or a buffer object).
- `recv` is a mutable buffer which will be filled with received bytes. It can be the same as `send`, or omitted. If omitted, a new buffer will be created.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: the buffer with the received bytes.

Constants

`SPI.`**`CONTROLLER`**

`SPI.`**`PERIPHERAL`**

for initialising the SPI bus to controller or peripheral mode

`SPI.`**`LSB`**

`SPI.`**`MSB`**

set the first bit to be the least or most significant bit

class Switch – switch object

A Switch object is used to control a push-button switch.

Usage:

```
sw = pyb.Switch()       # create a switch object
sw.value()              # get state (True if pressed, False otherwise)
sw()                    # shorthand notation to get the switch state
sw.callback(f)          # register a callback to be called when the
```

(continues on next page)

(continued from previous page)

```
                        #   switch is pressed down
sw.callback(None)       # remove the callback
```

Example:

```
pyb.Switch().callback(lambda: pyb.LED(1).toggle())
```

Constructors

class pyb.**Switch**

Create and return a switch object.

Methods

Switch.**__call__**()

Call switch object directly to get its state: `True` if pressed down, `False` otherwise.

Switch.**value**()

Get the switch state. Returns `True` if pressed down, otherwise `False`.

Switch.**callback**(*fun*)

Register the given function to be called when the switch is pressed down. If `fun` is `None`, then it disables the callback.

class Timer – control internal timers

Timers can be used for a great variety of tasks. At the moment, only the simplest case is implemented: that of calling a function periodically.

Each timer consists of a counter that counts up at a certain rate. The rate at which it counts is the peripheral clock frequency (in Hz) divided by the timer prescaler. When the counter reaches the timer period it triggers an event, and the counter resets back to zero. By using the callback method, the timer event can call a Python function.

Example usage to toggle an LED at a fixed frequency:

```
tim = pyb.Timer(4)              # create a timer object using timer 4
tim.init(freq=2)                # trigger at 2Hz
tim.callback(lambda t:pyb.LED(1).toggle())
```

Example using named function for the callback:

```
def tick(timer):                # we will receive the timer object when being called
    print(timer.counter())      # show current timer's counter value
tim = pyb.Timer(4, freq=1)      # create a timer object using timer 4 - trigger at 1Hz
tim.callback(tick)              # set the callback to our tick function
```

Further examples:

```
tim = pyb.Timer(4, freq=100)    # freq in Hz
tim = pyb.Timer(4, prescaler=0, period=99)
tim.counter()                   # get counter (can also set)
```

(continues on next page)

(continued from previous page)

```
tim.prescaler(2)                # set prescaler (can also get)
tim.period(199)                 # set period (can also get)
tim.callback(lambda t: ...)     # set callback for update interrupt (t=tim instance)
tim.callback(None)              # clear callback
```

Note: Timer(2) and Timer(3) are used for PWM to set the intensity of LED(3) and LED(4) respectively. But these timers are only configured for PWM if the intensity of the relevant LED is set to a value between 1 and 254. If the intensity feature of the LEDs is not used then these timers are free for general purpose use. Similarly, Timer(5) controls the servo driver, and Timer(6) is used for timed ADC/DAC reading/writing. It is recommended to use the other timers in your programs.

Note: Memory can't be allocated during a callback (an interrupt) and so exceptions raised within a callback don't give much information. See `micropython.alloc_emergency_exception_buf()` for how to get around this limitation.

Constructors

class pyb.**Timer**(*id*, ...)

Construct a new timer object of the given id. If additional arguments are given, then the timer is initialised by `init(...)`. `id` can be 1 to 14.

Methods

Timer.**init**(*, *freq*, *prescaler*, *period*, *mode=Timer.UP*, *div=1*, *callback=None*, *deadtime=0*, *brk=Timer.BRK_OFF*, *hard=True*)

Initialise the timer. Initialisation must be either by frequency (in Hz) or by prescaler and period:

```
tim.init(freq=100)                  # set the timer to trigger at 100Hz
tim.init(prescaler=83, period=999)  # set the prescaler and period directly
```

Keyword arguments:

- `freq` — specifies the periodic frequency of the timer. You might also view this as the frequency with which the timer goes through one complete cycle.
- `prescaler` [0-0xffff] - specifies the value to be loaded into the timer's Prescaler Register (PSC). The timer clock source is divided by (`prescaler + 1`) to arrive at the timer clock. Timers 2-7 and 12-14 have a clock source of 84 MHz (pyb.freq()[2] * 2), and Timers 1, and 8-11 have a clock source of 168 MHz (pyb.freq()[3] * 2).
- `period` [0-0xffff] for timers 1, 3, 4, and 6-15. [0-0x3fffffff] for timers 2 & 5. Specifies the value to be loaded into the timer's AutoReload Register (ARR). This determines the period of the timer (i.e. when the counter cycles). The timer counter will roll-over after `period + 1` timer clock cycles.
- `mode` can be one of:
  - `Timer.UP` - configures the timer to count from 0 to ARR (default)
  - `Timer.DOWN` - configures the timer to count from ARR down to 0.
  - `Timer.CENTER` - configures the timer to count from 0 to ARR and then back down to 0.
- `div` can be one of 1, 2, or 4. Divides the timer clock to determine the sampling clock used by the digital filters.
- `callback` - as per Timer.callback()

- `deadtime` - specifies the amount of "dead" or inactive time between transitions on complimentary channels (both channels will be inactive) for this time). `deadtime` may be an integer between 0 and 1008, with the following restrictions: 0-128 in steps of 1. 128-256 in steps of 2, 256-512 in steps of 8, and 512-1008 in steps of 16. `deadtime` measures ticks of `source_freq` divided by `div` clock ticks. `deadtime` is only available on timers 1 and 8.
- `brk` - specifies if the break mode is used to kill the output of the PWM when the `BRK_IN` input is asserted. The value of this argument determines if break is enabled and what the polarity is, and can be one of `Timer.BRK_OFF`, `Timer.BRK_LOW` or `Timer.BRK_HIGH`. To select the `BRK_IN` pin construct a Pin object with `mode=Pin.ALT, alt=Pin.AFn_TIMx`. The pin's GPIO input features are available in alt mode - `pull=` , `value()` and `irq()`.
- `hard` can be one of:
  - `True` - The callback will be executed in hard interrupt context, which minimises delay and jitter but is subject to the limitations described in *Writing interrupt handlers* including being unable to allocate on the heap.
  - `False` - The callback will be scheduled as a soft interrupt, allowing it to allocate but possibly also introducing garbage-collection delays and jitter.

  The default value of this option is True.

You must either specify freq or both of period and prescaler.

`Timer.`**`deinit`**`()`

Deinitialises the timer.

Disables the callback (and the associated irq).

Disables any channel callbacks (and the associated irq). Stops the timer, and disables the timer peripheral.

`Timer.`**`callback`**`(`*`fun`*`)`

Set the function to be called when the timer triggers. `fun` is passed 1 argument, the timer object. If `fun` is `None` then the callback will be disabled.

`Timer.`**`channel`**`(`*`channel`*`, `*`mode`*`, ...)`

If only a channel number is passed, then a previously initialized channel object is returned (or `None` if there is no previous channel).

Otherwise, a TimerChannel object is initialized and returned.

Each channel can be configured to perform pwm, output compare, or input capture. All channels share the same underlying timer, which means that they share the same timer clock.

Keyword arguments:

- `mode` can be one of:
  - `Timer.PWM` — configure the timer in PWM mode (active high).
  - `Timer.PWM_INVERTED` — configure the timer in PWM mode (active low).
  - `Timer.OC_TIMING` — indicates that no pin is driven.
  - `Timer.OC_ACTIVE` — the pin will be made active when a compare match occurs (active is determined by polarity)
  - `Timer.OC_INACTIVE` — the pin will be made inactive when a compare match occurs.
  - `Timer.OC_TOGGLE` — the pin will be toggled when an compare match occurs.
  - `Timer.OC_FORCED_ACTIVE` — the pin is forced active (compare match is ignored).
  - `Timer.OC_FORCED_INACTIVE` — the pin is forced inactive (compare match is ignored).

  - `Timer.IC` — configure the timer in Input Capture mode.
  - `Timer.ENC_A` — configure the timer in Encoder mode. The counter only changes when CH1 changes.
  - `Timer.ENC_B` — configure the timer in Encoder mode. The counter only changes when CH2 changes.
  - `Timer.ENC_AB` — configure the timer in Encoder mode. The counter changes when CH1 or CH2 changes.
- `callback` - as per TimerChannel.callback()
- `pin` None (the default) or a Pin object. If specified (and not None) this will cause the alternate function of the indicated pin to be configured for this timer channel. An error will be raised if the pin doesn't support any alternate functions for this timer channel.

Keyword arguments for Timer.PWM modes:

- `pulse_width` - determines the initial pulse width value to use.
- `pulse_width_percent` - determines the initial pulse width percentage to use.
- `pulse_width_us` - determines the initial pulse width in microseconds.
- `pulse_width_ns` - determines the initial pulse width in nanoseconds.

Keyword arguments for Timer.OC modes:

- `compare` - determines the initial value of the compare register.
- `polarity` can be one of:
  - `Timer.HIGH` - output is active high
  - `Timer.LOW` - output is active low

Optional keyword arguments for Timer.IC modes:

- `polarity` can be one of:
  - `Timer.RISING` - captures on rising edge.
  - `Timer.FALLING` - captures on falling edge.
  - `Timer.BOTH` - captures on both edges.

Note that capture only works on the primary channel, and not on the complimentary channels.

Notes for Timer.ENC modes:

- Requires 2 pins, so one or both pins will need to be configured to use the appropriate timer AF using the Pin API.
- Read the encoder value using the timer.counter() method.
- Only works on CH1 and CH2 (and not on CH1N or CH2N)
- The channel number is ignored when setting the encoder mode.

PWM Example:

```
timer = pyb.Timer(2, freq=1000)
ch2 = timer.channel(2, pyb.Timer.PWM, pin=pyb.Pin.board.X2, pulse_width=8000)
ch3 = timer.channel(3, pyb.Timer.PWM, pin=pyb.Pin.board.X3, pulse_width=16000)
```

PWM Motor Example with complementary outputs, dead time, break input and break callback:

```
from pyb import Timer
from machine import Pin # machine.Pin supports alt mode and irq on the same pin.
pin_t8_1 = Pin(Pin.board.Y1, mode=Pin.ALT, af=Pin.AF3_TIM8)   # Pin PC6, TIM8_CH1
pin_t8_1n = Pin(Pin.board.X8, mode=Pin.ALT, af=Pin.AF3_TIM8)  # Pin PA7, TIM8_CH1N
pin_bkin = Pin(Pin.board.X7, mode=Pin.ALT, af=Pin.AF3_TIM8)   # Pin PA6, TIM8_BKIN
pin_bkin.irq(handler=break_callabck, trigger=Pin.IRQ_FALLING)
timer = pyb.Timer(8, freq=1000, deadtime=1008, brk=Timer.BRK_LOW)
ch1 = timer.channel(1, pyb.Timer.PWM, pulse_width_percent=30)
```

`Timer.`**`counter`**`(`[*value*]`)`

Get or set the timer counter.

`Timer.`**`freq`**`(`[*value*]`)`

Get or set the frequency for the timer (changes prescaler and period if set).

`Timer.`**`period`**`(`[*value*]`)`

Get or set the period of the timer.

`Timer.`**`prescaler`**`(`[*value*]`)`

Get or set the prescaler for the timer.

`Timer.`**`source_freq`**`()`

Get the frequency of the source of the timer.

class TimerChannel — setup a channel for a timer

Timer channels are used to generate/capture a signal using a timer.

TimerChannel objects are created using the Timer.channel() method.

Methods

`timerchannel.`**`callback`**`(`*fun*`)`

Set the function to be called when the timer channel triggers. `fun` is passed 1 argument, the timer object. If `fun` is `None` then the callback will be disabled.

`timerchannel.`**`capture`**`(`[*value*]`)`

Get or set the capture value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. capture is the logical name to use when the channel is in input capture mode.

`timerchannel.`**`compare`**`(`[*value*]`)`

Get or set the compare value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. compare is the logical name to use when the channel is in output compare mode.

`timerchannel.`**`pulse_width`**`(`[*value*]`)`

Get or set the pulse width value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. pulse_width is the logical name to use when the channel is in PWM mode.

In edge aligned mode, a pulse_width of `period + 1` corresponds to a duty cycle of 100% In center aligned mode, a pulse width of `period` corresponds to a duty cycle of 100%

`timerchannel.`**`pulse_width_percent`**`(`[*value*]`)`

Get or set the pulse width percentage associated with a channel. The value is a number between 0 and 100 and sets the percentage of the timer period for which the pulse is active. The value can be an integer or floating-point number for more accuracy. For example, a value of 25 gives a duty cycle of 25%.

timerchannel.**pulse_width_us**([*value*])

Get or set the pulse width in microseconds associated with a channel. The value is converted to/from timer ticks using the timer's clock and prescaler. Conversions are performed assuming simple up-counting and do not account for center-aligned mode. For example, passing 1000 sets the pulse width to 1 ms.

timerchannel.**pulse_width_ns**([*value*])

Get or set the pulse width in nanoseconds associated with a channel. The value is converted to/from timer ticks using the timer's clock and prescaler. Conversions are performed assuming simple up-counting and do not account for center-aligned mode. This method offers finer resolution than `pulse_width_us()` and matches the interface of *machine.PWM.duty_ns()*.

Constants

Timer.**UP**

Timer.**DOWN**

Timer.**CENTER**

Configures the timer to count Up, Down, or from 0 to ARR and then back down to 0.

Timer.**BRK_OFF**

Timer.**BRK_LOW**

Timer.**BRK_HIGH**

Configures the break mode when passed to the `brk` keyword argument.

class UART – duplex serial communication bus

UART implements the standard UART/USART duplex serial communications protocol. At the physical level it consists of 2 lines: RX and TX. The unit of communication is a character (not to be confused with a string character) which can be 8 or 9 bits wide.

UART objects can be created and initialised using:

```
from pyb import UART

uart = UART(1, 9600)                         # init with given baudrate
uart.init(9600, bits=8, parity=None, stop=1) # init with given parameters
```

Bits can be 7, 8 or 9. Parity can be None, 0 (even) or 1 (odd). Stop can be 1 or 2.

Note: with parity=None, only 8 and 9 bits are supported. With parity enabled, only 7 and 8 bits are supported.

A UART object acts like a *stream* object and reading and writing is done using the standard stream methods:

```
uart.read(10)       # read 10 characters, returns a bytes object
uart.read()         # read all available characters
uart.readline()     # read a line
uart.readinto(buf)  # read and store into the given buffer
uart.write('abc')   # write the 3 characters
```

Individual characters can be read/written using:

```
uart.readchar()     # read 1 character and returns it as an integer
uart.writechar(42)  # write 1 character
```

To check if there is anything to be read, use:

```
uart.any()          # returns the number of characters waiting
```

Note: The stream functions `read`, `write`, etc. are new in MicroPython v1.3.4. Earlier versions use `uart.send` and `uart.recv`.

Constructors

class pyb.**UART**(*bus*, ...)

Construct a UART object on the given bus. For Pyboard `bus` can be 1-4, 6, 'XA', 'XB', 'YA', or 'YB'. For Pyboard Lite `bus` can be 1, 2, 6, 'XB', or 'YA'. For Pyboard D `bus` can be 1-4, 'XA', 'YA' or 'YB'. With no additional parameters, the UART object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See `init` for parameters of initialisation.

The physical pins of the UART buses on Pyboard are:

- `UART(4)` is on `XA`: `(TX, RX) = (X1, X2) = (PA0, PA1)`
- `UART(1)` is on `XB`: `(TX, RX) = (X9, X10) = (PB6, PB7)`
- `UART(6)` is on `YA`: `(TX, RX) = (Y1, Y2) = (PC6, PC7)`
- `UART(3)` is on `YB`: `(TX, RX) = (Y9, Y10) = (PB10, PB11)`
- `UART(2)` is on: `(TX, RX) = (X3, X4) = (PA2, PA3)`

The Pyboard Lite supports UART(1), UART(2) and UART(6) only, pins are:

- `UART(1)` is on `XB`: `(TX, RX) = (X9, X10) = (PB6, PB7)`
- `UART(6)` is on `YA`: `(TX, RX) = (Y1, Y2) = (PC6, PC7)`
- `UART(2)` is on: `(TX, RX) = (X1, X2) = (PA2, PA3)`

The Pyboard D supports UART(1), UART(2), UART(3) and UART(4) only, pins are:

- `UART(4)` is on `XA`: `(TX, RX) = (X1, X2) = (PA0, PA1)`
- `UART(1)` is on `YA`: `(TX, RX) = (Y1, Y2) = (PA9, PA10)`
- `UART(3)` is on `YB`: `(TX, RX) = (Y9, Y10) = (PB10, PB11)`
- `UART(2)` is on: `(TX, RX) = (X3, X4) = (PA2, PA3)`

Note: Pyboard D has `UART(1)` on `YA`, unlike Pyboard and Pyboard Lite that both have `UART(1)` on `XB` and `UART(6)` on `YA`.

Methods

UART.**init**(*baudrate*, *bits=8*, *parity=None*, *stop=1*, ***, *timeout=0*, *flow=0*, *timeout_char=0*, *read_buf_len=64*)

Initialise the UART bus with the given parameters:

- `baudrate` is the clock rate.
- `bits` is the number of bits per character, 7, 8 or 9.
- `parity` is the parity, `None`, 0 (even) or 1 (odd).
- `stop` is the number of stop bits, 1 or 2.
- `flow` sets the flow control type. Can be 0, `UART.RTS`, `UART.CTS` or `UART.RTS | UART.CTS`.
- `timeout` is the timeout in milliseconds to wait for writing/reading the first character.

- `timeout_char` is the timeout in milliseconds to wait between characters while writing or reading.
- `read_buf_len` is the character length of the read buffer (0 to disable).

This method will raise an exception if the baudrate could not be set within 5% of the desired value. The minimum baudrate is dictated by the frequency of the bus that the UART is on; UART(1) and UART(6) are APB2, the rest are on APB1. The default bus frequencies give a minimum baudrate of 1300 for UART(1) and UART(6) and 650 for the others. Use *pyb.freq* to reduce the bus frequencies to get lower baudrates.

Note: with parity=None, only 8 and 9 bits are supported. With parity enabled, only 7 and 8 bits are supported.

`UART.`**`deinit`**`()`

Turn off the UART bus.

`UART.`**`any`**`()`

Returns the number of bytes waiting (may be 0).

`UART.`**`read`**`(`[*nbytes*]`)`

Read characters. If `nbytes` is specified then read at most that many bytes. If `nbytes` are available in the buffer, returns immediately, otherwise returns when sufficient characters arrive or the timeout elapses.

If `nbytes` is not given then the method reads as much data as possible. It returns after the timeout has elapsed.

Note: for 9 bit characters each character takes two bytes, `nbytes` must be even, and the number of characters is `nbytes/2`.

Return value: a bytes object containing the bytes read in. Returns `None` on timeout.

`UART.`**`readchar`**`()`

Receive a single character on the bus.

Return value: The character read, as an integer. Returns -1 on timeout.

`UART.`**`readinto`**`(`*buf*[, *nbytes*]`)`

Read bytes into the `buf`. If `nbytes` is specified then read at most that many bytes. Otherwise, read at most `len(buf)` bytes.

Return value: number of bytes read and stored into `buf` or `None` on timeout.

`UART.`**`readline`**`()`

Read a line, ending in a newline character. If such a line exists, return is immediate. If the timeout elapses, all available data is returned regardless of whether a newline exists.

Return value: the line read or `None` on timeout if no data is available.

`UART.`**`write`**`(`*buf*`)`

Write the buffer of bytes to the bus. If characters are 7 or 8 bits wide then each byte is one character. If characters are 9 bits wide then two bytes are used for each character (little endian), and `buf` must contain an even number of bytes.

Return value: number of bytes written. If a timeout occurs and no bytes were written returns `None`.

`UART.`**`writechar`**`(`*char*`)`

Write a single character on the bus. `char` is an integer to write. Return value: `None`. See note below if CTS flow control is used.

`UART.`**`sendbreak`**`()`

Send a break condition on the bus. This drives the bus low for a duration of 13 bits. Return value: `None`.

Constants

`UART.RTS`

`UART.CTS`

to select the flow control type.

Flow Control

On Pyboards V1 and V1.1 `UART(2)` and `UART(3)` support RTS/CTS hardware flow control using the following pins:

- `UART(2)` is on: `(TX, RX, nRTS, nCTS) = (X3, X4, X2, X1) = (PA2, PA3, PA1, PA0)`
- `UART(3)` is on :`(TX, RX, nRTS, nCTS) = (Y9, Y10, Y7, Y6) = (PB10, PB11, PB14, PB13)`

On the Pyboard Lite only `UART(2)` supports flow control on these pins:

```
(TX, RX, nRTS, nCTS) = (X1, X2, X4, X3) = (PA2, PA3, PA1, PA0)
```

In the following paragraphs the term "target" refers to the device connected to the UART.

When the UART's `init()` method is called with `flow` set to one or both of `UART.RTS` and `UART.CTS` the relevant flow control pins are configured. `nRTS` is an active low output, `nCTS` is an active low input with pullup enabled. To achieve flow control the Pyboard's `nCTS` signal should be connected to the target's `nRTS` and the Pyboard's `nRTS` to the target's `nCTS`.

CTS: target controls Pyboard transmitter

If CTS flow control is enabled the write behaviour is as follows:

If the Pyboard's `UART.write(buf)` method is called, transmission will stall for any periods when `nCTS` is `False`. This will result in a timeout if the entire buffer was not transmitted in the timeout period. The method returns the number of bytes written, enabling the user to write the remainder of the data if required. In the event of a timeout, a character will remain in the UART pending `nCTS`. The number of bytes composing this character will be included in the return value.

If `UART.writechar()` is called when `nCTS` is `False` the method will time out unless the target asserts `nCTS` in time. If it times out `OSError 116` will be raised. The character will be transmitted as soon as the target asserts `nCTS`.

RTS: Pyboard controls target's transmitter

If RTS flow control is enabled, behaviour is as follows:

If buffered input is used (`read_buf_len` > 0), incoming characters are buffered. If the buffer becomes full, the next character to arrive will cause `nRTS` to go `False`: the target should cease transmission. `nRTS` will go `True` when characters are read from the buffer.

Note that the `any()` method returns the number of bytes in the buffer. Assume a buffer length of `N` bytes. If the buffer becomes full, and another character arrives, `nRTS` will be set False, and `any()` will return the count `N`. When characters are read the additional character will be placed in the buffer and will be included in the result of a subsequent `any()` call.

If buffered input is not used (`read_buf_len` == 0) the arrival of a character will cause `nRTS` to go `False` until the character is read.

class USB_HID – USB Human Interface Device (HID)

The USB_HID class allows creation of an object representing the USB Human Interface Device (HID) interface. It can be used to emulate a peripheral such as a mouse or keyboard.

Before you can use this class, you need to use *pyb.usb_mode()* to set the USB mode to include the HID interface.

Constructors

class pyb.**USB_HID**

Create a new USB_HID object.

Methods

USB_HID.**recv**(*data*, *, *timeout=5000*)

Receive data on the bus:

- `data` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: if `data` is an integer then a new buffer of the bytes received, otherwise the number of bytes read into `data` is returned.

USB_HID.**send**(*data*)

Send data over the USB HID interface:

- `data` is the data to send (a tuple/list of integers, or a bytearray).

class USB_VCP – USB virtual comm port

The USB_VCP class allows creation of a *stream*-like object representing the USB virtual comm port. It can be used to read and write data over USB to the connected host.

Constructors

class pyb.**USB_VCP**(*id=0*)

Create a new USB_VCP object. The *id* argument specifies which USB VCP port to use.

Methods

USB_VCP.**init**(*, *flow=-1*)

Configure the USB VCP port. If the *flow* argument is not -1 then the value sets the flow control, which can be a bitwise-or of `USB_VCP.RTS` and `USB_VCP.CTS`. RTS is used to control read behaviour and CTS, to control write behaviour.

USB_VCP.**setinterrupt**(*chr*)

Set the character which interrupts running Python code. This is set to 3 (CTRL-C) by default, and when a CTRL-C character is received over the USB VCP port, a KeyboardInterrupt exception is raised.

Set to -1 to disable this interrupt feature. This is useful when you want to send raw bytes over the USB VCP port.

USB_VCP.**isconnected**()

Return `True` if USB is connected as a serial device, else `False`.

USB_VCP.**any**()

Return `True` if any characters waiting, else `False`.

USB_VCP.**close**()

This method does nothing. It exists so the USB_VCP object can act as a file.

USB_VCP.**read**([*nbytes*])

Read at most `nbytes` from the serial device and return them as a bytes object. If `nbytes` is not specified then the method reads all available bytes from the serial device. USB_VCP *stream* implicitly works in non-blocking mode, so if no pending data available, this method will return immediately with `None` value.

USB_VCP.**readinto**(*buf*[, *maxlen*])

Read bytes from the serial device and store them into `buf`, which should be a buffer-like object. At most `len(buf)` bytes are read. If `maxlen` is given and then at most `min(maxlen, len(buf))` bytes are read.

Returns the number of bytes read and stored into `buf` or `None` if no pending data available.

USB_VCP.**readline**()

Read a whole line from the serial device.

Returns a bytes object containing the data, including the trailing newline character or `None` if no pending data available.

USB_VCP.**readlines**()

Read as much data as possible from the serial device, breaking it into lines.

Returns a list of bytes objects, each object being one of the lines. Each line will include the newline character.

USB_VCP.**write**(*buf*)

Write the bytes from `buf` to the serial device.

Returns the number of bytes written.

USB_VCP.**recv**(*data*, *, *timeout=5000*)

Receive data on the bus:

- `data` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: if `data` is an integer then a new buffer of the bytes received, otherwise the number of bytes read into `data` is returned.

USB_VCP.**send**(*data*, *, *timeout=5000*)

Send data over the USB VCP:

- `data` is the data to send (an integer to send, or a buffer object).
- `timeout` is the timeout in milliseconds to wait for the send.

Return value: number of bytes sent.

USB_VCP.**irq**(*handler=None*, *trigger=IRQ_RX*, *hard=False*)

Register *handler* to be called whenever an event specified by *trigger* occurs. The *handler* function must take exactly one argument, which will be the USB VCP object. Pass in `None` to disable the callback.

Valid values for *trigger* are:

- `USB_VCP.IRQ_RX`: new data is available for reading from the USB VCP object.

Constants

USB_VCP.**RTS**

USB_VCP.**CTS**

to select the flow control type.

USB_VCP.**IRQ_RX**

IRQ trigger values for *USB_VCP.irq()*.

stm — functionality specific to STM32 MCUs

This module provides functionality specific to STM32 microcontrollers, including direct access to peripheral registers.

Memory access

The module exposes three objects used for raw memory access.

stm.**mem8**

Read/write 8 bits of memory.

stm.**mem16**

Read/write 16 bits of memory.

stm.**mem32**

Read/write 32 bits of memory.

Use subscript notation `[...]` to index these objects with the address of interest.

These memory objects can be used in combination with the peripheral register constants to read and write registers of the MCU hardware peripherals, as well as all other areas of address space.

Peripheral register constants

The module defines constants for registers which are generated from CMSIS header files, and the constants available depend on the microcontroller series that is being compiled for. Examples of some constants include:

stm.**GPIOA**

Base address of the GPIOA peripheral.

stm.**GPIOB**

Base address of the GPIOB peripheral.

stm.**GPIO_BSRR**

Offset of the GPIO bit set/reset register.

stm.**GPIO_IDR**

Offset of the GPIO input data register.

stm.**GPIO_ODR**

Offset of the GPIO output data register.

Constants that are named after a peripheral, like `GPIOA`, are the absolute address of that peripheral. Constants that have a prefix which is the name of a peripheral, like `GPIO_BSRR`, are relative offsets of the register. Accessing peripheral registers requires adding the absolute base address of the peripheral and the relative register offset. For example `GPIOA + GPIO_BSRR` is the full, absolute address of the `GPIOA->BSRR` register.

Example use:

```
# set PA2 high
stm.mem32[stm.GPIOA + stm.GPIO_BSRR] = 1 << 2

# read PA3
value = (stm.mem32[stm.GPIOA + stm.GPIO_IDR] >> 3) & 1
```

Functions specific to STM32WBxx MCUs

These functions are available on STM32WBxx microcontrollers, and interact with the second CPU, the RF core.

stm.**rfcore_status**()

Returns the status of the second CPU as an integer (the first word of device info table).

stm.**rfcore_fw_version**(*id*)

Get the version of the firmware running on the second CPU. Pass in 0 for *id* to get the FUS version, and 1 to get the WS version.

Returns a 5-tuple with the full version number.

stm.**rfcore_sys_hci**(*ogf*, *ocf*, *data*, *timeout_ms=0*)

Execute a HCI command on the SYS channel. The execution is synchronous.

Returns a bytes object with the result of the SYS command.

Functions specific to STM32WLxx MCUs

These functions are available on STM32WLxx microcontrollers, and interact with the integrated "SUBGHZ" radio modem peripheral.

stm.**subghz_cs**(*level*)

Sets the internal SPI CS pin attached to the radio peripheral. The `level` argument is active-low: a truthy value means "CS pin high" and de-asserts the signal, a falsey value means "CS pin low" and asserts the signal.

The internal-only SPI bus corresponding to this CS signal can be instantiated using *machine.SPI()* `id` value `"SUBGHZ"`.

stm.**subghz_irq**(*handler*)

Sets the internal SUBGHZ radio interrupt handler to the provided function. The handler function is called as a "hard" interrupt in response to radio peripheral interrupts. See *Writing interrupt handlers* for more information about interrupt handlers in MicroPython.

Calling this function with the handler argument set to None disables the IRQ.

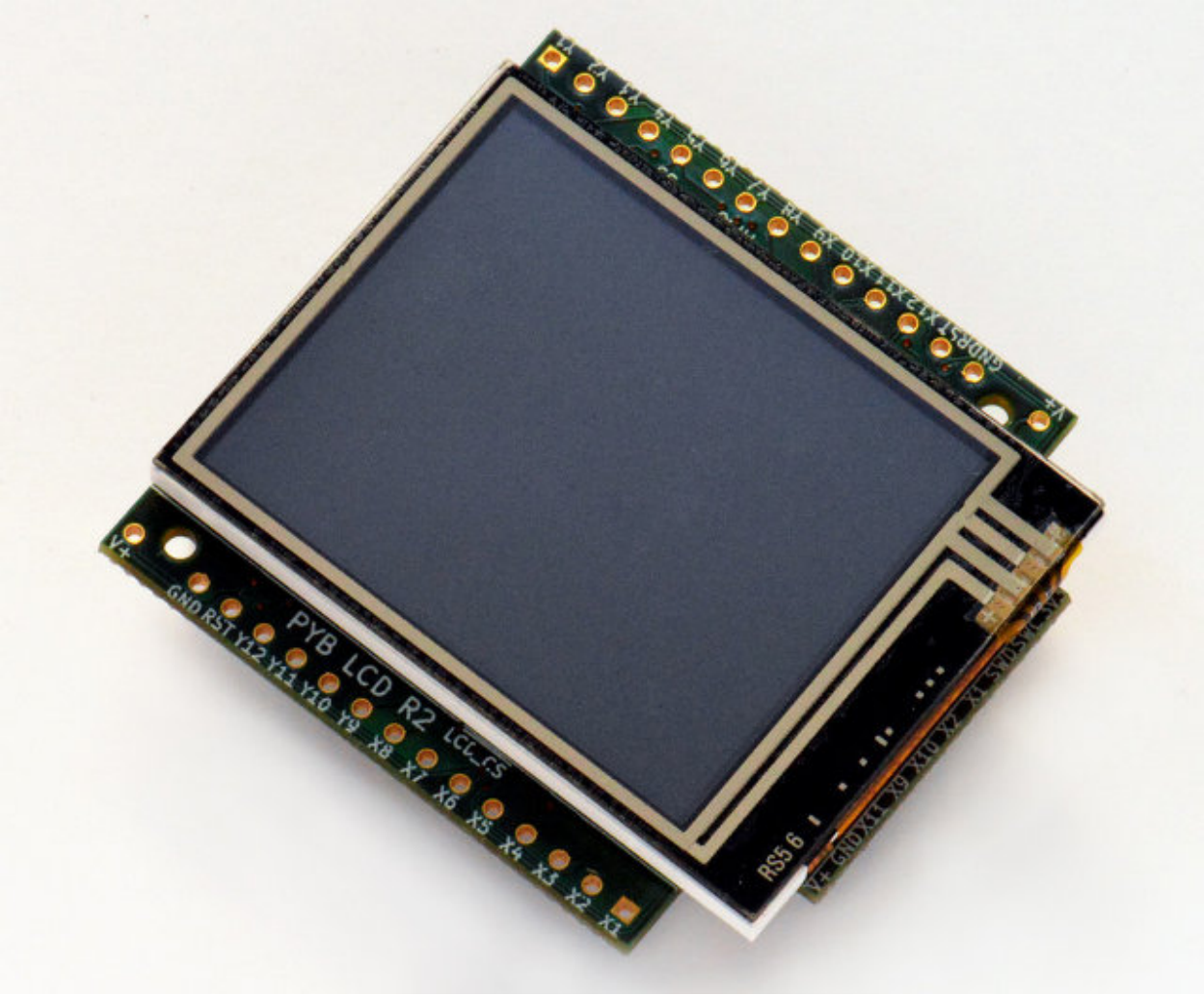
Due to a hardware limitation, each time this IRQ fires MicroPython disables it before calling the handler. In order to receive another interrupt, Python code should call `subghz_irq()` to set the handler again. This has the side effect of re-enabling the IRQ.

`stm.`**`subghz_is_busy`**`()`

Return a `bool` corresponding to the internal "RFBUSYS" signal from the radio peripheral. Before sending a new command to the radio over SPI then this function should be polled until it returns `False`, to confirm the busy signal is de-asserted.

`lcd160cr` — control of LCD160CR display

This module provides control of the MicroPython LCD160CR display.



Further resources are available via the following links:

- LCD160CRv1.0 reference manual (100KiB PDF)
- LCD160CRv1.0 schematics (1.6MiB PDF)

class LCD160CR

The LCD160CR class provides an interface to the display. Create an instance of this class and use its methods to draw to the LCD and get the status of the touch panel.

For example:

```
import lcd160cr

lcd = lcd160cr.LCD160CR('X')
lcd.set_orient(lcd160cr.PORTRAIT)
lcd.set_pos(0, 0)
lcd.set_text_color(lcd.rgb(255, 0, 0), lcd.rgb(0, 0, 0))
lcd.set_font(1)
lcd.write('Hello MicroPython!')
print('touch:', lcd.get_touch())
```

Constructors

class lcd160cr.**LCD160CR**(*connect=None*, ***, *pwr=None*, *i2c=None*, *spi=None*, *i2c_addr=98*)

Construct an LCD160CR object. The parameters are:

- *connect* is a string specifying the physical connection of the LCD display to the board; valid values are "X", "Y", "XY", "YX". Use "X" when the display is connected to a pyboard in the X-skin position, and "Y" when connected in the Y-skin position. "XY" and "YX" are used when the display is connected to the right or left side of the pyboard, respectively.
- *pwr* is a Pin object connected to the LCD's power/enabled pin.
- *i2c* is an I2C object connected to the LCD's I2C interface.
- *spi* is an SPI object connected to the LCD's SPI interface.
- *i2c_addr* is the I2C address of the display.

One must specify either a valid *connect* or all of *pwr*, *i2c* and *spi*. If a valid *connect* is given then any of *pwr*, *i2c* or *spi* which are not passed as parameters (i.e. they are `None`) will be created based on the value of *connect*. This allows to override the default interface to the display if needed.

The default values are:

- "X" is for the X-skin and uses: `pwr=Pin("X4")`, `i2c=I2C("X")`, `spi=SPI("X")`
- "Y" is for the Y-skin and uses: `pwr=Pin("Y4")`, `i2c=I2C("Y")`, `spi=SPI("Y")`
- "XY" is for the right-side and uses: `pwr=Pin("X4")`, `i2c=I2C("Y")`, `spi=SPI("X")`
- "YX" is for the left-side and uses: `pwr=Pin("Y4")`, `i2c=I2C("X")`, `spi=SPI("Y")`

See this image for how the display can be connected to the pyboard.

Static methods

static LCD160CR.**rgb**(*r*, *g*, *b*)

Return a 16-bit integer representing the given rgb color values. The 16-bit value can be used to set the font color (see *LCD160CR.set_text_color()*) pen color (see *LCD160CR.set_pen()*) and draw individual pixels.

LCD160CR.clip_line(data, w, h):

Clip the given line data. This is for internal use.

Instance members

The following instance members are publicly accessible.

LCD160CR.**w**

LCD160CR.**h**

The width and height of the display, respectively, in pixels. These members are updated when calling *LCD160CR.set_orient()* and should be considered read-only.

Setup commands

LCD160CR.**set_power**(*on*)

Turn the display on or off, depending on the given value of *on*: 0 or `False` will turn the display off, and 1 or `True` will turn it on.

LCD160CR.**set_orient**(*orient*)

Set the orientation of the display. The *orient* parameter can be one of *PORTRAIT*, *LANDSCAPE*, *PORTRAIT_UPSIDEDOWN*, *LANDSCAPE_UPSIDEDOWN*.

LCD160CR.**set_brightness**(*value*)

Set the brightness of the display, between 0 and 31.

LCD160CR.**set_i2c_addr**(*addr*)

Set the I2C address of the display. The *addr* value must have the lower 2 bits cleared.

LCD160CR.**set_uart_baudrate**(*baudrate*)

Set the baudrate of the UART interface.

LCD160CR.**set_startup_deco**(*value*)

Set the start-up decoration of the display. The *value* parameter can be a logical or of *STARTUP_DECO_NONE*, *STARTUP_DECO_MLOGO*, *STARTUP_DECO_INFO*.

LCD160CR.**save_to_flash**()

Save the following parameters to flash so they persist on restart and power up: initial decoration, orientation, brightness, UART baud rate, I2C address.

Pixel access methods

The following methods manipulate individual pixels on the display.

LCD160CR.**set_pixel**(*x*, *y*, *c*)

Set the specified pixel to the given color. The color should be a 16-bit integer and can be created by *LCD160CR.rgb()*.

LCD160CR.**get_pixel**(*x*, *y*)

Get the 16-bit value of the specified pixel.

LCD160CR.**get_line**(*x*, *y*, *buf*)

Low-level method to get a line of pixels into the given buffer. To read *n* pixels *buf* should be *2*n+1* bytes in length. The first byte is a dummy byte and should be ignored, and subsequent bytes represent the pixels in the line starting at coordinate *(x, y)*.

LCD160CR.**screen_dump**(*buf*, *x=0*, *y=0*, *w=None*, *h=None*)

Dump the contents of the screen to the given buffer. The parameters *x* and *y* specify the starting coordinate, and *w* and *h* the size of the region. If *w* or *h* are `None` then they will take on their maximum values, set by the size of the screen minus the given *x* and *y* values. *buf* should be large enough to hold `2*w*h` bytes. If it's smaller then only the initial horizontal lines will be stored.

LCD160CR.**screen_load**(*buf*)

Load the entire screen from the given buffer.

Drawing text

To draw text one sets the position, color and font, and then uses *LCD160CR.write* to draw the text.

LCD160CR.**set_pos**(*x*, *y*)

Set the position for text output using *LCD160CR.write()*. The position is the upper-left corner of the text.

LCD160CR.**set_text_color**(*fg*, *bg*)

Set the foreground and background color of the text.

LCD160CR.**set_font**(*font*, *scale=0*, *bold=0*, *trans=0*, *scroll=0*)

Set the font for the text. Subsequent calls to *write* will use the newly configured font. The parameters are:

- *font* is the font family to use, valid values are 0, 1, 2, 3.
- *scale* is a scaling value for each character pixel, where the pixels are drawn as a square with side length equal to *scale* + *1*. The value can be between 0 and 63.
- *bold* controls the number of pixels to overdraw each character pixel, making a bold effect. The lower 2 bits of *bold* are the number of pixels to overdraw in the horizontal direction, and the next 2 bits are for the vertical direction. For example, a *bold* value of 5 will overdraw 1 pixel in both the horizontal and vertical directions.
- *trans* can be either 0 or 1 and if set to 1 the characters will be drawn with a transparent background.
- *scroll* can be either 0 or 1 and if set to 1 the display will do a soft scroll if the text moves to the next line.

LCD160CR.**write**(*s*)

Write text to the display, using the current position, color and font. As text is written the position is automatically incremented. The display supports basic VT100 control codes such as newline and backspace.

Drawing primitive shapes

Primitive drawing commands use a foreground and background color set by the *set_pen* method.

LCD160CR.**set_pen**(*line*, *fill*)

Set the line and fill color for primitive shapes.

LCD160CR.**erase**()

Erase the entire display to the pen fill color.

LCD160CR.**dot**(*x*, *y*)

Draw a single pixel at the given location using the pen line color.

LCD160CR.**rect**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_outline**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_interior**(*x*, *y*, *w*, *h*)

Draw a rectangle at the given location and size using the pen line color for the outline, and the pen fill color for the interior. The *rect* method draws the outline and interior, while the other methods just draw one or the other.

LCD160CR.**line**(*x1*, *y1*, *x2*, *y2*)

Draw a line between the given coordinates using the pen line color.

LCD160CR.**dot_no_clip**(*x*, *y*)

LCD160CR.**rect_no_clip**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_outline_no_clip**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_interior_no_clip**(*x*, *y*, *w*, *h*)

LCD160CR.**line_no_clip**(*x1*, *y1*, *x2*, *y2*)

These methods are as above but don't do any clipping on the input coordinates. They are faster than the clipping versions and can be used when you know that the coordinates are within the display.

LCD160CR.**poly_dot**(*data*)

Draw a sequence of dots using the pen line color. The *data* should be a buffer of bytes, with each successive pair of bytes corresponding to coordinate pairs (x, y).

LCD160CR.**poly_line**(*data*)

Similar to *LCD160CR.poly_dot()* but draws lines between the dots.

Touch screen methods

LCD160CR.**touch_config**(*calib=False*, *save=False*, *irq=None*)

Configure the touch panel:

- If *calib* is `True` then the call will trigger a touch calibration of the resistive touch sensor. This requires the user to touch various parts of the screen.
- If *save* is `True` then the touch parameters will be saved to NVRAM to persist across reset/power up.
- If *irq* is `True` then the display will be configured to pull the IRQ line low when a touch force is detected. If *irq* is `False` then this feature is disabled. If *irq* is `None` (the default value) then no change is made to this setting.

LCD160CR.**is_touched**()

Returns a boolean: `True` if there is currently a touch force on the screen, `False` otherwise.

LCD160CR.**get_touch**()

Returns a 3-tuple of: *(active, x, y)*. If there is currently a touch force on the screen then *active* is 1, otherwise it is 0. The *x* and *y* values indicate the position of the current or most recent touch.

Advanced commands

LCD160CR.**set_spi_win**(*x*, *y*, *w*, *h*)

Set the window that SPI data is written to.

LCD160CR.**fast_spi**(*flush=True*)

Ready the display to accept RGB pixel data on the SPI bus, resetting the location of the first byte to go to the top-left corner of the window set by *LCD160CR.set_spi_win()*. The method returns an SPI object which can be used to write the pixel data.

Pixels should be sent as 16-bit RGB values in the 5-6-5 format. The destination counter will increase as data is sent, and data can be sent in arbitrary sized chunks. Once the destination counter reaches the end of the window specified by *LCD160CR.set_spi_win()* it will wrap around to the top-left corner of that window.

LCD160CR.**show_framebuf**(*buf*)

Show the given buffer on the display. *buf* should be an array of bytes containing the 16-bit RGB values for the pixels, and they will be written to the area specified by *LCD160CR.set_spi_win()*, starting from the top-left corner.

The framebuf module can be used to construct frame buffers and provides drawing primitives. Using a frame buffer will improve performance of animations when compared to drawing directly to the screen.

LCD160CR.**set_scroll**(*on*)

Turn scrolling on or off. This controls globally whether any window regions will scroll.

LCD160CR.**set_scroll_win**(*win*, *x=-1*, *y=0*, *w=0*, *h=0*, *vec=0*, *pat=0*, *fill=0x07e0*, *color=0*)

Configure a window region for scrolling:

- *win* is the window id to configure. There are 0..7 standard windows for general purpose use. Window 8 is the text scroll window (the ticker).
- *x*, *y*, *w*, *h* specify the location of the window in the display.
- *vec* specifies the direction and speed of scroll: it is a 16-bit value of the form `0bF.ddSSSSSSSSSSSS`. *dd* is 0, 1, 2, 3 for +x, +y, -x, -y scrolling. *F* sets the speed format, with 0 meaning that the window is shifted *S % 256* pixel every frame, and 1 meaning that the window is shifted 1 pixel every *S* frames.
- *pat* is a 16-bit pattern mask for the background.
- *fill* is the fill color.
- *color* is the extra color, either of the text or pattern foreground.

LCD160CR.**set_scroll_win_param**(*win*, *param*, *value*)

Set a single parameter of a scrolling window region:

- *win* is the window id, 0..8.
- *param* is the parameter number to configure, 0..7, and corresponds to the parameters in the *set_scroll_win* method.
- *value* is the value to set.

LCD160CR.**set_scroll_buf**(*s*)

Set the string for scrolling in window 8. The parameter *s* must be a string with length 32 or less.

LCD160CR.**jpeg**(*buf*)

Display a JPEG. *buf* should contain the entire JPEG data. JPEG data should not include EXIF information. The following encodings are supported: Baseline DCT, Huffman coding, 8 bits per sample, 3 color components, YCbCr4:2:2. The origin of the JPEG is set by *LCD160CR.set_pos()*.

LCD160CR.**jpeg_start**(*total_len*)

LCD160CR.**jpeg_data**(*buf*)

Display a JPEG with the data split across multiple buffers. There must be a single call to *jpeg_start* to begin with, specifying the total number of bytes in the JPEG. Then this number of bytes must be transferred to the display using one or more calls to the *jpeg_data* command.

LCD160CR.**feed_wdt**()

The first call to this method will start the display's internal watchdog timer. Subsequent calls will feed the watchdog. The timeout is roughly 30 seconds.

LCD160CR.**reset**()

Reset the display.

Constants

lcd160cr.**PORTRAIT**

lcd160cr.**LANDSCAPE**

lcd160cr.**PORTRAIT_UPSIDEDOWN**

lcd160cr.**LANDSCAPE_UPSIDEDOWN**

Orientations of the display, used by *LCD160CR.set_orient()*.

lcd160cr.**STARTUP_DECO_NONE**

lcd160cr.**STARTUP_DECO_MLOGO**

lcd160cr.**STARTUP_DECO_INFO**

Types of start-up decoration, can be OR'ed together, used by *LCD160CR.set_startup_deco()*.

1.3.2 Libraries specific to the WiPy

The following libraries and classes are specific to the WiPy.

wipy – WiPy specific features

The wipy module contains functions to control specific features of the WiPy, such as the heartbeat LED.

Functions

wipy.**heartbeat**([*enable*])

Get or set the state (enabled or disabled) of the heartbeat LED. Accepts and returns boolean values (`True` or `False`).

class ADCWiPy – analog to digital conversion

Note: This class is a non-standard ADC implementation for the WiPy. It is available simply as `machine.ADC` on the WiPy but is named in the documentation below as `machine.ADCWiPy` to distinguish it from the more general *machine.ADC* class.

Usage:

```
import machine

adc = machine.ADC()             # create an ADC object
apin = adc.channel(pin='GP3')   # create an analog pin on GP3
val = apin()                    # read an analog value
```

Constructors

class machine.**ADCWiPy**(*id=0*, ***, *bits=12*)

Create an ADC object associated with the given pin. This allows you to then read analog values on that pin. For more info check the pinout and alternate functions table.

Warning: ADC pin input range is 0-1.4V (being 1.8V the absolute maximum that it can withstand). When GP2, GP3, GP4 or GP5 are remapped to the ADC block, 1.8 V is the maximum. If these pins are used in digital mode, then the maximum allowed input is 3.6V.

Methods

ADCWiPy.**channel**(*id*, ***, *pin*)

Create an analog pin. If only channel ID is given, the correct pin will be selected. Alternatively, only the pin can be passed and the correct channel will be selected. Examples:

```
# all of these are equivalent and enable ADC channel 1 on GP3
apin = adc.channel(1)
apin = adc.channel(pin='GP3')
apin = adc.channel(id=1, pin='GP3')
```

ADCWiPy.**init**()

Enable the ADC block.

ADCWiPy.**deinit**()

Disable the ADC block.

class ADCChannel — read analog values from internal or external sources

ADC channels can be connected to internal points of the MCU or to GPIO pins. ADC channels are created using the ADC.channel method.

`machine.`**`adcchannel`**`()`

Fast method to read the channel value.

`adcchannel.`**`value`**`()`

Read the channel value.

`adcchannel.`**`init`**`()`

Re-init (and effectively enable) the ADC channel.

`adcchannel.`**`deinit`**`()`

Disable the ADC channel.

class TimerWiPy – control hardware timers

Note: This class is a non-standard Timer implementation for the WiPy. It is available simply as `machine.Timer` on the WiPy but is named in the documentation below as `machine.TimerWiPy` to distinguish it from the more general *machine.Timer* class.

Hardware timers deal with timing of periods and events. Timers are perhaps the most flexible and heterogeneous kind of hardware in MCUs and SoCs, differently greatly from a model to a model. MicroPython's Timer class defines a baseline operation of executing a callback with a given period (or once after some delay), and allow specific boards to define more non-standard behaviour (which thus won't be portable to other boards).

Constructors

`class` `machine.`**`TimerWiPy`**`(`*id, ...*`)`

Construct a new timer object of the given id. Id of -1 constructs a virtual timer (if supported by a board).

Methods

`TimerWiPy.`**`init`**`(`*mode, *, width=16*`)`

Initialise the timer. Example:

```
tim.init(Timer.PERIODIC)              # periodic 16-bit timer
tim.init(Timer.ONE_SHOT, width=32)    # one shot 32-bit timer
```

Keyword arguments:

- `mode` can be one of:
  - `TimerWiPy.ONE_SHOT` - The timer runs once until the configured period of the channel expires.
  - `TimerWiPy.PERIODIC` - The timer runs periodically at the configured frequency of the channel.
  - `TimerWiPy.PWM` - Output a PWM signal on a pin.
- `width` must be either 16 or 32 (bits). For really low frequencies < 5Hz (or large periods), 32-bit timers should be used. 32-bit mode is only available for `ONE_SHOT` AND `PERIODIC` modes.

`TimerWiPy.deinit()`

Deinitialises the timer. Stops the timer, and disables the timer peripheral.

`TimerWiPy.channel(channel, **, freq, period, polarity=TimerWiPy.POSITIVE, duty_cycle=0)`

If only a channel identifier passed, then a previously initialized channel object is returned (or `None` if there is no previous channel).

Otherwise, a TimerChannel object is initialized and returned.

The operating mode is the one configured to the Timer object that was used to create the channel.

- `channel` if the width of the timer is 16-bit, then must be either `TIMER.A`, `TIMER.B`. If the width is 32-bit then it **must be** `TIMER.A | TIMER.B`.

Keyword only arguments:

- `freq` sets the frequency in Hz.
- `period` sets the period in microseconds.

Note: Either `freq` or `period` must be given, never both.

- `polarity` this is applicable for `PWM`, and defines the polarity of the duty cycle
- `duty_cycle` only applicable to `PWM`. It's a percentage (0.00-100.00). Since the WiPy doesn't support floating point numbers the duty cycle must be specified in the range 0-10000, where 10000 would represent 100.00, 5050 represents 50.50, and so on.

Note: When the channel is in PWM mode, the corresponding pin is assigned automatically, therefore there's no need to assign the alternate function of the pin via the `Pin` class. The pins which support PWM functionality are the following:

- `GP24` on Timer 0 channel A.
- `GP25` on Timer 1 channel A.
- `GP9` on Timer 2 channel B.
- `GP10` on Timer 3 channel A.
- `GP11` on Timer 3 channel B.

---

class TimerChannel — setup a channel for a timer

Timer channels are used to generate/capture a signal using a timer.

TimerChannel objects are created using the Timer.channel() method.

Methods

timerchannel.**irq**(*, *trigger*, *priority=1*, *handler=None*)

The behaviour of this callback is heavily dependent on the operating mode of the timer channel:

- If mode is TimerWiPy.PERIODIC the callback is executed periodically with the configured frequency or period.
- If mode is TimerWiPy.ONE_SHOT the callback is executed once when the configured timer expires.
- If mode is TimerWiPy.PWM the callback is executed when reaching the duty cycle value.

The accepted params are:

- priority level of the interrupt. Can take values in the range 1-7. Higher values represent higher priorities.
- handler is an optional function to be called when the interrupt is triggered.
- trigger must be TimerWiPy.TIMEOUT when the operating mode is either TimerWiPy.PERIODIC or TimerWiPy.ONE_SHOT. In the case that mode is TimerWiPy.PWM then trigger must be equal to TimerWiPy.MATCH.

Note that callback handlers are hard interrupts, and the constraints described in *Writing interrupt handlers* apply when they are executed.

Returns a callback object.

timerchannel.**freq**([*value*])

Get or set the timer channel frequency (in Hz).

timerchannel.**period**([*value*])

Get or set the timer channel period (in microseconds).

timerchannel.**duty_cycle**([*value*])

Get or set the duty cycle of the PWM signal. It's a percentage (0.00-100.00). Since the WiPy doesn't support floating point numbers the duty cycle must be specified in the range 0-10000, where 10000 would represent 100.00, 5050 represents 50.50, and so on.

Constants

TimerWiPy.**ONE_SHOT**

TimerWiPy.**PERIODIC**

Timer operating mode.

1.3.3 Libraries specific to the ESP8266 and ESP32

The following libraries are specific to the ESP8266 and ESP32.

esp — functions related to the ESP8266 and ESP32

The esp module contains specific functions related to both the ESP8266 and ESP32 modules. Some functions are only available on one or the other of these ports.

Functions

esp.**sleep_type**([*sleep_type*])

Note: ESP8266 only

Get or set the sleep type.

If the *sleep_type* parameter is provided, sets the sleep type to its value. If the function is called without parameters, returns the current sleep type.

The possible sleep types are defined as constants:

- SLEEP_NONE – all functions enabled,
- SLEEP_MODEM – modem sleep, shuts down the WiFi Modem circuit.
- SLEEP_LIGHT – light sleep, shuts down the WiFi Modem circuit and suspends the processor periodically.

The system enters the set sleep mode automatically when possible.

esp.**deepsleep**(*time_us=0*, */*)

Note: ESP8266 only - use *machine.deepsleep()* on ESP32

Enter deep sleep.

The whole module powers down, except for the RTC clock circuit, which can be used to restart the module after the specified time if the pin 16 is connected to the reset pin. Otherwise the module will sleep until manually reset.

esp.**flash_id**()

Note: ESP8266 only

Read the device ID of the flash memory.

esp.**flash_size**()

Read the total size of the flash memory.

esp.**flash_user_start**()

Read the memory offset at which the user flash space begins.

esp.**flash_read**(*byte_offset*, *length_or_buffer*)

esp.**flash_write**(*byte_offset*, *bytes*)

esp.**flash_erase**(*sector_no*)

esp.**osdebug**(*uart_no*)

Note: This is the ESP8266 form of this function.

Change the level of OS serial debug log messages. On boot, OS serial debug log messages are disabled.

uart_no is the number of the UART peripheral which should receive OS-level output, or None to disable OS serial debug log messages.

esp.**osdebug**(*uart_no*[, *level*])

Note: This is the ESP32 form of this function.

Change the level of OS serial debug log messages. On boot, OS serial debug log messages are limited to Error output only.

The behaviour of this function depends on the arguments passed to it. The following combinations are supported:

`osdebug(None)` restores the default OS debug log message level (`LOG_ERROR`).

`osdebug(0)` enables all available OS debug log messages (in the default build configuration this is `LOG_INFO`).

`osdebug(0, level)` sets the OS debug log message level to the specified value. The log levels are defined as constants:

- `LOG_NONE` – No log output
- `LOG_ERROR` – Critical errors, software module can not recover on its own
- `LOG_WARN` – Error conditions from which recovery measures have been taken
- `LOG_INFO` – Information messages which describe normal flow of events
- `LOG_DEBUG` – Extra information which is not necessary for normal use (values, pointers, sizes, etc)
- `LOG_VERBOSE` – Bigger chunks of debugging information, or frequent messages which can potentially flood the output

Note: `LOG_DEBUG` and `LOG_VERBOSE` are not compiled into the MicroPython binary by default, to save size. A custom build with a modified "`sdkconfig`" source file is needed to see any output at these log levels.

Note: Log output on ESP32 is automatically suspended in "Raw REPL" mode, to prevent communications issues. This means OS level logging is never seen when using `mpremote run` and similar tools.

esp.**set_native_code_location**(*start*, *length*)

Note: ESP8266 only

Set the location that native code will be placed for execution after it is compiled. Native code is emitted when the `@micropython.native`, `@micropython.viper` and `@micropython.asm_xtensa` decorators are applied to a function. The ESP8266 must execute code from either iRAM or the lower 1MByte of flash (which is memory mapped), and this function controls the location.

If *start* and *length* are both `None` then the native code location is set to the unused portion of memory at the end of the iRAM1 region. The size of this unused portion depends on the firmware and is typically quite small (around 500 bytes), and is enough to store a few very small functions. The advantage of using this iRAM1 region is that it does not get worn out by writing to it.

If neither *start* nor *length* are `None` then they should be integers. *start* should specify the byte offset from the beginning of the flash at which native code should be stored. *length* specifies how many bytes of flash from *start* can be used to store native code. *start* and *length* should be multiples of the sector size (being 4096 bytes). The flash will be automatically erased before writing to it so be sure to use a region of flash that is not otherwise used, for example by the firmware or the filesystem.

When using the flash to store native code *start*+*length* must be less than or equal to 1MByte. Note that the flash can be worn out if repeated erasures (and writes) are made so use this feature sparingly. In particular, native code needs to be recompiled and rewritten to flash on each boot (including wake from deepsleep).

In both cases above, using iRAM1 or flash, if there is no more room left in the specified region then the use of a native decorator on a function will lead to *MemoryError* exception being raised during compilation of that function.

esp32 — functionality specific to the ESP32

The esp32 module contains functions and classes specifically aimed at controlling ESP32 modules.

Functions

esp32.**wake_on_touch**(*wake*)

Configure whether or not a touch will wake the device from sleep. *wake* should be a boolean value.

Note: This is only available for boards that have touch sensor support.

esp32.**wake_on_ulp**(*wake*)

Configure whether or not the Ultra-Low-Power co-processor can wake the device from sleep. *wake* should be a boolean value.

Note: This is only available for boards that have ULP coprocessor support.

esp32.**wake_on_ext0**(*pin*, *level*)

Configure how EXT0 wakes the device from sleep. *pin* can be None or a valid Pin object. *level* should be esp32.WAKEUP_ALL_LOW or esp32.WAKEUP_ANY_HIGH.

Note: This is only available for boards that have ext0 support.

esp32.**wake_on_ext1**(*pins*, *level*)

Configure how EXT1 wakes the device from sleep. *pins* can be None or a tuple/list of valid Pin objects. *level* should be esp32.WAKEUP_ALL_LOW or esp32.WAKEUP_ANY_HIGH.

Note: This is only available for boards that have ext1 support.

esp32.**wake_on_gpio**(*pins*, *level*)

Configure how GPIO wakes the device from sleep. *pins* can be None or a tuple/list of valid Pin objects. *level* should be esp32.WAKEUP_ALL_LOW or esp32.WAKEUP_ANY_HIGH.

Note: Some boards don't support waking on GPIO from deep sleep, on those boards, the pins set here can only be used to wake from light sleep.

esp32.**gpio_deep_sleep_hold**(*enable*)

Configure whether non-RTC GPIO pin configuration is retained during deep-sleep mode for held pads. *enable* should be a boolean value.

esp32.**raw_temperature**()

Read the raw value of the internal temperature sensor, returning an integer.

esp32.**idf_heap_info**(*capabilities*)

Returns information about the ESP-IDF heap memory regions. One of them contains the MicroPython heap and the others are used by ESP-IDF, e.g., for network buffers and other data. This data is useful to get a sense of how much memory is available to ESP-IDF and the networking stack in particular. It may shed some light on situations where ESP-IDF operations fail due to allocation failures.

The capabilities parameter corresponds to ESP-IDF's MALLOC_CAP_XXX values but the two most useful ones are predefined as *esp32.HEAP_DATA* for data heap regions and *esp32.HEAP_EXEC* for executable regions as used by the native code emitter.

The return value is a list of 4-tuples, where each 4-tuple corresponds to one heap and contains: the total bytes, the free bytes, the largest free block, and the minimum free seen over time.

Example after booting:

```
>>> import esp32; esp32.idf_heap_info(esp32.HEAP_DATA)
[(240, 0, 0, 0), (7288, 0, 0, 0), (16648, 4, 4, 4), (79912, 35712, 35512, 35108),
 (15072, 15036, 15036, 15036), (113840, 0, 0, 0)]
```

Note: Free IDF heap memory in the *esp32.HEAP_DATA* region is available to be automatically added to the MicroPython heap to prevent a MicroPython allocation from failing. However, the information returned here is otherwise *not* useful to troubleshoot Python allocation failures. *micropython.mem_info()* and *gc.mem_free()* should be used instead:

The "max new split" value in *micropython.mem_info()* output corresponds to the largest free block of ESP-IDF heap that could be automatically added on demand to the MicroPython heap.

The result of *gc.mem_free()* is the total of the current "free" and "max new split" values printed by *micropython.mem_info()*.

---

esp32.**idf_task_info**()

Returns information about running ESP-IDF/FreeRTOS tasks, which include MicroPython threads. This data is useful to gain insight into how much time tasks spend running or if they are blocked for significant parts of time, and to determine if allocated stacks are fully utilized or might be reduced.

CONFIG_FREERTOS_USE_TRACE_FACILITY=y must be set in the board configuration to make this method available. Additionally configuring CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS=y and CONFIG_FREERTOS_VTASKLIST_INCLUDE_COREID=y is recommended to be able to retrieve the total and per-task runtime and the core ID respectively.

The return value is a 2-tuple where the first value is the total runtime, and the second a list of tasks. Each task is a 7-tuple containing: the task ID, name, current state, priority, runtime, stack high water mark, and the ID of the core it is running on. Runtime and core ID will be None when the respective FreeRTOS configuration option is not enabled.

Note: For an easier to use output based on this function you can use the utop library, which implements a live overview similar to the Unix top command.

Flash partitions

This class gives access to the partitions in the device's flash memory and includes methods to enable over-the-air (OTA) updates.

class esp32.**Partition**(*id*, *block_size=4096*, */*)

Create an object representing a partition. *id* can be a string which is the label of the partition to retrieve, or one of the constants: `BOOT` or `RUNNING`. *block_size* specifies the byte size of an individual block.

classmethod Partition.**find**(*type=TYPE_APP*, *subtype=0xff*, *label=None*, *block_size=4096*)

Find a partition specified by *type*, *subtype* and *label*. Returns a (possibly empty) list of Partition objects. Note: `subtype=0xff` matches any subtype and `label=None` matches any label.

block_size specifies the byte size of an individual block used by the returned objects.

Partition.**info**()

Returns a 6-tuple `(type, subtype, addr, size, label, encrypted)`.

Partition.**readblocks**(*block_num*, *buf*)

Partition.**readblocks**(*block_num*, *buf*, *offset*)

Partition.**writeblocks**(*block_num*, *buf*)

Partition.**writeblocks**(*block_num*, *buf*, *offset*)

Partition.**ioctl**(*cmd*, *arg*)

These methods implement the simple and *extended* block protocol defined by *vfs.AbstractBlockDev*.

Partition.**set_boot**()

Sets the partition as the boot partition.

Note: Do not enter *deepsleep* after changing the OTA boot partition, without first performing a hard *reset* or power cycle. This ensures the bootloader will validate the new image before booting.

Partition.**get_next_update**()

Gets the next update partition after this one, and returns a new Partition object. Typical usage is `Partition(Partition.RUNNING).get_next_update()` which returns the next partition to update given the current running one.

classmethod Partition.**mark_app_valid_cancel_rollback**()

Signals that the current boot is considered successful. Calling `mark_app_valid_cancel_rollback` is required on the first boot of a new partition to avoid an automatic rollback at the next boot. This uses the ESP-IDF "app rollback" feature with "CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE" and an `OSError(-261)` is raised if called on firmware that doesn't have the feature enabled. It is OK to call `mark_app_valid_cancel_rollback` on every boot and it is not necessary when booting firmware that was loaded using esptool.

Constants

`Partition.`**`BOOT`**

`Partition.`**`RUNNING`**

Used in the *`Partition`* constructor to fetch various partitions: `BOOT` is the partition that will be booted at the next reset and `RUNNING` is the currently running partition.

`Partition.`**`TYPE_APP`**

`Partition.`**`TYPE_DATA`**

Used in *`Partition.find`* to specify the partition type: `APP` is for bootable firmware partitions (typically labelled `factory`, `ota_0`, `ota_1`), and `DATA` is for other partitions, e.g. `nvs`, `otadata`, `phy_init`, `vfs`.

`esp32.`**`HEAP_DATA`**

`esp32.`**`HEAP_EXEC`**

Used in *`idf_heap_info`*.

PCNT

This class provides access to the ESP32 hardware support for pulse counting. There are 8 pulse counter units, with id 0..7.

See the *machine.Counter* and *machine.Encoder* classes for simpler and portable abstractions of common pulse counting applications. These classes are implemented as thin Python shims around *`PCNT`*.

`class` `esp32.`**`PCNT`**(*id*, *, ...)

Returns the singleton PCNT instance for the given unit `id`.

Keyword arguments are passed to the `init()` method as described below.

`PCNT.`**`init`**(*, ...)

(Re-)initialise a pulse counter unit. Supported keyword arguments are:

- `channel`: see description below
- `pin`: the input Pin to monitor for pulses
- `rising`: an action to take on a rising edge - one of `PCNT.INCREMENT`, `PCNT.DECREMENT` or `PCNT.IGNORE` (the default)
- `falling`: an action to take on a falling edge (takes the save values as the `rising` argument).
- `mode_pin`: ESP32 pulse counters support monitoring a second pin and altering the behaviour of the counter based on its level - set this keyword to any input Pin
- `mode_low`: set to either `PCNT.HOLD` or `PCNT.REVERSE` to either suspend counting or reverse the direction of the counter (i.e., `PCNT.INCREMENT` behaves as `PCNT.DECREMENT` and vice versa) when `mode_pin` is low
- `mode_high`: as `mode_low` but for the behaviour when `mode_pin` is high
- `filter`: set to a value 1..1023, in ticks of the 80MHz clock, to enable the pulse width filter
- `min`: set to the minimum level of the counter value when decrementing (-32768..-1) or 0 to disable
- `max`: set to the maximum level of the counter value when incrementing (1..32767) or 0 to disable
- `threshold0`: sets the counter value for the `PCNT.IRQ_THRESHOLD0` event (see `irq` method)
- `threshold1`: sets the counter value for the `PCNT.IRQ_THRESHOLD1` event (see `irq` method)

- `value`: can be set to `0` to reset the counter value

The hardware initialisation is done in stages and so some of the keyword arguments can be used in groups or in isolation to partially reconfigure a unit:

- the `pin` keyword (optionally combined with `mode_pin`) can be used to change just the bound pin(s)
- `rising`, `falling`, `mode_low` and `mode_high` can be used (singly or together) to change the counting logic - omitted keywords use their default (`PCNT.IGNORE` or `PCNT.NORMAL`)
- `filter` can be used to change only the pulse width filter (with 0 disabling it)
- each of `min`, `max`, `threshold0` and `threshold1` can be used to change these limit/event values individually; however, setting any will reset the counter to zero (i.e., they imply `value=0`)

Each pulse counter unit supports two channels, 0 and 1, each able to monitor different pins with different counting logic but updating the same counter value. Use `channel=1` with the `pin`, `rising`, `falling`, `mode_pin`, `mode_low` and `mode_high` keywords to configure the second channel.

The second channel can be used to configure 4X quadrature decoding with a single counter unit:

```
pin_a = Pin(2, Pin.INPUT, pull=Pin.PULL_UP)
pin_b = Pin(3, Pin.INPUT, pull=Pin.PULL_UP)
rotary = PCNT(0, min=-32000, max=32000)
rotary.init(channel=0, pin=pin_a, falling=PCNT.INCREMENT, rising=PCNT.DECREMENT,␣
↪mode_pin=pin_b, mode_low=PCNT.REVERSE)
rotary.init(channel=1, pin=pin_b, falling=PCNT.DECREMENT, rising=PCNT.INCREMENT,␣
↪mode_pin=pin_a, mode_low=PCNT.REVERSE)
rotary.start()
```

`PCNT.`**`value`**([*value*])

Call this method with no arguments to return the current counter value.

If the optional *value* argument is set to `0` then the counter is reset (but the previous value is returned). Read and reset is not atomic and so it is possible for a pulse to be missed. Any value other than `0` will raise an error.

`PCNT.`**`irq`**(*handler=None*, *trigger=PCNT.IRQ_ZERO*)

ESP32 pulse counters support interrupts on these counter events:

- `PCNT.IRQ_ZERO`: the counter has reset to zero
- `PCNT.IRQ_MIN`: the counter has hit the `min` value
- `PCNT.IRQ_MAX`: the counter has hit the `max` value
- `PCNT.IRQ_THRESHOLD0`: the counter has hit the `threshold0` value
- `PCNT.IRQ_THRESHOLD1`: the counter has hit the `threshold1` value

`trigger` should be a bit-mask of the desired events OR'ed together. The `handler` function should take a single argument which is the *PCNT* instance that raised the event.

This method returns a callback object. The callback object can be used to access the bit-mask of events that are outstanding on the PCNT unit.:

```
def pcnt_irq(pcnt):
    flags = pcnt.irq().flags()
    if flags & PCNT.IRQ_ZERO:
        # reset
    if flags & PCNT.IRQ_MAX:
        # overflow...
    ... etc
```

(continues on next page)

(continued from previous page)

```
pcnt.irq(handler=pcnt_irq, trigger=PCNT.IRQ_ZERO | PCNT.IRQ_MAX | ...)
```

Note: Accessing `irq.flags()` will clear the flags, so only call it once per invocation of the handler.

The handler is called with the MicroPython scheduler and so will run at a point after the interrupt. If another interrupt occurs before the handler has been called then the events will be coalesced together into a single call and the bit mask will indicate all events that have occurred.

To avoid race conditions between a handler being called and retrieving the current counter value, the `value()` method will force execution of any pending events before returning the current counter value (and potentially resetting the value).

Only one handler can be in place per-unit. Set `handler` to `None` to disable the event interrupt.

Note: ESP32 pulse counters reset to *zero* when reaching the minimum or maximum value. Thus the `IRQ_ZERO` event will also trigger when either of these events occurs.

See the *machine.Counter* and *machine.Encoder* classes for simpler abstractions of common pulse counting applications.

RMT

The RMT (Remote Control) module, specific to the ESP32, was originally designed to send and receive infrared remote control signals. However, due to a flexible design and very accurate (as low as 12.5ns) pulse generation, it can also be used to transmit or receive many other types of digital signals:

```
import esp32
from machine import Pin

r = esp32.RMT(pin=Pin(18), resolution_hz=10000000)
r  # RMT(pin=18, source_freq=80000000, resolution_hz=10000000, idle_level=0)

# To apply a carrier frequency to the high output
r = esp32.RMT(pin=Pin(18), resolution_hz=10000000, tx_carrier=(38000, 50, 1))

# The channel resolution is 100ns (1/resolution_hz)
r.write_pulses((1, 20, 2, 40), 0)  # Send 0 for 100ns, 1 for 2000ns, 0 for 200ns, 1␣
↪for 4000ns
```

The input to the RMT module is an 80MHz clock (in the future it may be able to configure the input clock but, for now, it's fixed). `resolution_hz` determines the resolution of the RMT channel. The numbers specified in `write_pulses` are multiplied by the resolution to define the pulses.

So, in the example above, the resolution is resolution is (1/10Mhz) = 100ns. Since the `start` level is 0 and toggles with each number, the bitstream is `0101` with durations of [100ns, 2000ns, 100ns, 4000ns].

For more details see Espressif's ESP-IDF RMT documentation..

Warning: The current MicroPython RMT implementation lacks some features, most notably receiving pulses. RMT should be considered a *beta feature* and the interface may change in the future.

class esp32.**RMT**(*channel*, ***, *pin=None*, *resolution_hz=10000000*, *clock_div=None*, *idle_level=False*, *num_symbols=48 | 64*, *tx_carrier=None*)

This class provides access to one of the eight RMT channels. *channel* is optional and a dummy parameter for backward compatibility. *pin* is required and configures which Pin is bound to the RMT channel. *resolution_hz* defines the resolution/unit of the samples. For example, 1,000,000 means the unit is microsecond. The pulse widths can assume values up to *RMT.PULSE_MAX*, so the resolution should be selected accordingly to the signal to be transmitted. *clock_div* (deprecated) is equivalent to *resolution_hz*, but expressed as a clock divider that divides the source clock (80MHz) to the RMT channel allowing the resolution to be specified. Either *clock_div* and *resolution_hz* may be supplied, but not both. *num_symbols* specifies the RMT buffer allocated for this channel (minimum 48 or 64, depending on chip), from a small pool of symbols (192 to 512, depending on chip) that are shared by all channels. This buffer does not limit the size of the pulse train that you can send, but bigger buffers reduce the CPU load and the potential of glitches/imprecise pulse lengths. *idle_level* specifies what level the output will be when no transmission is in progress and can be any value that converts to a boolean, with `True` representing high voltage and `False` representing low.

To enable the transmission carrier feature, *tx_carrier* should be a tuple of three positive integers: carrier frequency, duty percent (`0` to `100`) and the output level to apply the carrier to (a boolean as per *idle_level*).

classmethod RMT.**source_freq**()

Returns the source clock frequency. Currently the source clock is not configurable so this will always return 80MHz.

RMT.**clock_div**()

Return the clock divider. Note that the channel resolution is `1 / (source_freq / clock_div)`. (Method deprecated. The value may not be faithful if resolution was supplied as *resolution_hz*.)

RMT.**wait_done**(*, *timeout=0*)

Returns `True` if the channel is idle or `False` if a sequence of pulses started with *RMT.write_pulses* is being transmitted. If the *timeout* keyword argument is given then block for up to this many milliseconds for transmission to complete. Timeout of -1 blocks until transmission is complete (and blocks forever if loop is enabled).

RMT.**loop**(*enable_loop*)

Configure looping on the channel. *enable_loop* is bool, set to `True` to enable looping on the *next* call to *RMT.write_pulses*. If called with `False` while a looping sequence is currently being transmitted then the transmission will stop. (Method deprecated by *RMT.loop_count*.)

RMT.**loop_count**(*n*)

Configure looping on the channel. *n* is int. Affects the *next* call to *RMT.write_pulses*. Set to `0` to disable looping, `-1` to enable infinite looping, or a positive number to loop for a given number of times. If *n* is changed, the current transmission is stopped.

Note: looping for a finite number of times is not supported by all flavors of ESP32.

RMT.**active**([*boolean*])

If called without parameters, returns *True* if there is an ongoing transmission.

If called with parameter *False*, stops the ongoing transmission. This is useful to stop an infinite transmission loop. The current loop is finished and transmission stops. The object is not invalidated, and the RMT channel is again enabled when a new transmission is started.

Calling with parameter *True* does not restart transmission. A new transmission should always be initiated by *write_pulses()*.

RMT.**deinit**()

Release all RMT resources and invalidate the object. All subsequent method calls will raise OSError. Useful to free RMT resources without having to wait for the object to be garbage-collected.

RMT.**write_pulses**(*duration*, *data=True*)

Begin transmitting a sequence. There are three ways to specify this:

Mode 1: *duration* is a list or tuple of durations. The optional *data* argument specifies the initial output level. The output level will toggle after each duration.

Mode 2: *duration* is a positive integer and *data* is a list or tuple of output levels. *duration* specifies a fixed duration for each.

Mode 3: *duration* and *data* are lists or tuples of equal length, specifying individual durations and the output level for each.

Durations are in integer units of the channel resolution (as described above), between 1 and `PULSE_MAX` units. Output levels are any value that can be converted to a boolean, with `True` representing high voltage and `False` representing low.

If transmission of an earlier sequence is in progress then this method will block until that transmission is complete before beginning the new sequence.

If looping has been enabled with *RMT.loop*, the sequence will be repeated indefinitely. Further calls to this method will block until the end of the current loop iteration before immediately beginning to loop the new sequence of pulses. Looping sequences longer than 126 pulses is not supported by the hardware.

static RMT.**bitstream_rmt**([*value*])

Configure RMT usage in the *machine.bitstream* implementation.

If *value* is `True`, bitstream tries to use RMT if possible. If *value* is `False`, bitstream sticks to the bit-banging implementation.

If no parameter is supplied, it returns the current state. The default state is `True`.

static RMT.**bitstream_channel**([*value*])

This function is deprecated and will be replaced by `RMT.bitstream_rmt()`.

Passing in no argument will return `1` if RMT was enabled for the *machine.bitstream* feature, and `None` otherwise.

Passing any non-negative integer argument is equivalent to calling `RMT.bitstream_rmt(True)`.

Note: In previous versions of MicroPython it was necessary to use this function to assign a specific RMT channel number for the bitstream, but the channel number is now assigned dynamically.

Constants

RMT.**PULSE_MAX**

Maximum integer that can be set for a pulse duration.

Ultra-Low-Power co-processor

This class gives access to the Ultra Low Power (ULP) co-processor on the ESP32, ESP32-S2 and ESP32-S3 chips.

Warning: This class does not provide access to the RISCV ULP co-processor available on the ESP32-S2 and ESP32-S3 chips.

class esp32.**ULP**

This class provides access to the Ultra-Low-Power co-processor.

ULP.**set_wakeup_period**(*period_index*, *period_us*)

Set the wake-up period.

ULP.**load_binary**(*load_addr*, *program_binary*)

Load a *program_binary* into the ULP at the given *load_addr*.

ULP.**run**(*entry_point*)

Start the ULP running at the given *entry_point*.

Constants

esp32.**WAKEUP_ALL_LOW**

esp32.**WAKEUP_ANY_HIGH**

Selects the wake level for pins.

Non-Volatile Storage

This class gives access to the Non-Volatile storage managed by ESP-IDF. The NVS is partitioned into namespaces and each namespace contains typed key-value pairs. The keys are strings and the values may be various integer types, strings, and binary blobs. The driver currently only supports 32-bit signed integers and blobs.

Warning: Changes to NVS need to be committed to flash by calling the commit method. Failure to call commit results in changes being lost at the next reset.

class esp32.**NVS**(*namespace*)

Create an object providing access to a namespace (which is automatically created if not present).

NVS.**set_i32**(*key*, *value*)

Sets a 32-bit signed integer value for the specified key. Remember to call *commit*!

NVS.**get_i32**(*key*)

Returns the signed integer value for the specified key. Raises an OSError if the key does not exist or has a different type.

NVS.**set_blob**(*key*, *value*)

Sets a binary blob value for the specified key. The value passed in must support the buffer protocol, e.g. bytes, bytearray, str. (Note that esp-idf distinguishes blobs and strings, this method always writes a blob even if a string is passed in as value.) Remember to call *commit*!

NVS.**get_blob**(*key*, *buffer*)

Reads the value of the blob for the specified key into the buffer, which must be a bytearray. Returns the actual length read. Raises an OSError if the key does not exist, has a different type, or if the buffer is too small.

NVS.**erase_key**(*key*)

Erases a key-value pair.

NVS.**commit**()

Commits changes made by *set_xxx* methods to flash.

Low Dropout Voltage Regulator (LDO)

This class gives access to the Low Dropout Voltage Regulator (LDO) on the ESP32P4 chip.

For more details see Espressif's ESP-IDF LDO documentation..

class esp32.**LDO**(*channel*, *voltage_mv*, *adjustable=False*)

Create an object providing access to the Low Dropout Voltage Regulator for a given LDO *channel*.

Supported keyword arguments:

- *voltage_mv*: The voltage value to be set to the LDO channel, in millivolts
- *adjustable*: (Optional): Whether the LDO channel is adjustable, and the voltage can be updated by *adjust_voltage*

LDO.**adjust_voltage**(*voltage_mv*)

Adjust the voltage of the LDO channel.

LDO.**release**()

Release the LDO channel, disabling its voltage output. Create a new LDO object to re-enable the output.

espnow — support for the ESP-NOW wireless protocol

This module provides an interface to the ESP-NOW protocol provided by Espressif on ESP32 and ESP8266 devices (API docs).

Table of Contents:

- *Introduction*
- *Configuration*
- *Sending and Receiving Data*
- *Peer Management*
- *Callback Methods*
- *Exceptions*
- *Constants*
- *Wifi Signal Strength (RSSI) - (ESP32 Only)*
- *Supporting asyncio*
- *Broadcast and Multicast*
- *ESPNow and Wifi Operation*
- *ESPNow and Sleep Modes*

Introduction

ESP-NOW is a connection-less wireless communication protocol supporting:

- Direct communication between up to 20 registered peers:
- Without the need for a wireless access point (AP),
- Encrypted and unencrypted communication (up to 6 encrypted peers),
- Message sizes up to 1470 bytes (For ESP-NOW v2),
- Can operate alongside Wi-Fi operation (*network.WLAN*) on ESP32 and ESP8266 devices.
- Track the Wi-Fi signal strength (RSSI) of ESP-NOW peer devices.

It is especially useful for small IoT networks, latency sensitive or power sensitive applications (such as battery operated devices) and for long-range communication between devices (hundreds of metres).

ESP-NOW Versions

Since ESP-IDF V5.4, two ESP-NOW versions are supported when running on ESP32: V1 and V2.

- The maximum packet length supported by V2 devices is 1470 bytes
- The maximum packet length supported by V1 devices is 250 bytes.

To check at runtime whether ESP-NOW V2 is available, check the value of *espnow.MAX_DATA_LEN*.

ESP-NOW V2 devices are capable of receiving packets from both V2 and V1 devices.

ESP-NOW V1 devices (including ESP8266) can receive packets from other V1 devices, or from V2 devices if the packet length doesn't exceed 250 bytes. For packets exceeding this length, a V1 device will either truncate the data to the first 250 bytes or discard the packet entirely.

Example

Sender:

```
import network
import espnow

# A WLAN interface must be active to send()/recv()
sta = network.WLAN(network.WLAN.IF_STA)  # Or network.WLAN.IF_AP
sta.active(True)
sta.disconnect()      # For ESP8266

e = espnow.ESPNow()
e.active(True)
peer = b'\xbb\xbb\xbb\xbb\xbb\xbb'    # MAC address of peer's wifi interface
e.add_peer(peer)      # Must add_peer() before send()

e.send(peer, "Starting...")
for i in range(100):
    e.send(peer, str(i)*20, True)
e.send(peer, b'end')
```

Receiver:

```
import network
import espnow

# A WLAN interface must be active to send()/recv()
sta = network.WLAN(network.WLAN.IF_STA)
sta.active(True)
sta.disconnect()   # Because ESP8266 auto-connects to last Access Point

e = espnow.ESPNow()
e.active(True)

while True:
    host, msg = e.recv()
    if msg:             # msg == None if timeout in recv()
        print(host, msg)
        if msg == b'end':
            break
```

class ESPNow

Constructor

class espnow.**ESPNow**

Returns the singleton ESPNow object. As this is a singleton, all calls to *espnow.ESPNow()* return a reference to the same object.

Note: Some methods are available only on the ESP32 due to code size restrictions on the ESP8266 and differences in the Espressif API.

Configuration

ESPNow.**active**([*flag*])

Initialise or de-initialise the ESP-NOW communication protocol depending on the value of the `flag` optional argument.

Arguments:

- *flag*: Any python value which can be converted to a boolean type.
  - `True`: Prepare the software and hardware for use of the ESP-NOW communication protocol, including:
    - initialise the ESPNow data structures,
    - allocate the recv data buffer,
    - invoke esp_now_init() and
    - register the send and recv callbacks.
  - `False`: De-initialise the Espressif ESP-NOW software stack (esp_now_deinit()), disable callbacks, deallocate the recv data buffer and deregister all peers.

If *flag* is not provided, return the current status of the ESPNow interface.

Returns:

`True` if interface is currently *active*, else `False`.

`ESPNow.`**`config`**(*param=value, ...*)

`ESPNow.`**`config`**(*'param') (ESP32 only*)

Set or get configuration values of the ESPNow interface. To set values, use the keyword syntax, and one or more parameters can be set at a time. To get a value the parameter name should be quoted as a string, and just one parameter is queried at a time.

Note: *Getting* parameters is not supported on the ESP8266.

Options:

rxbuf: (default=528 or 2972) Get/set the size in bytes of the internal buffer used to store incoming ESP-NOW packet data. The default size is selected to fit two max-sized ESP-NOW packets (250 or 1470 bytes) with associated mac_address (6 bytes), a message byte count (2 byte) and RSSI data plus buffer overhead. Increase this if you expect to receive a lot of large packets or expect bursty incoming traffic.

Note: If only using ESP-NOW V1 packets and low throughput, recommend setting `rxbuf=528` here to reduce memory overhead.

Note: The recv buffer is allocated by *`ESPNow.active()`*. Changing this value will have no effect until the next call of *`ESPNow.active(True)`*.

---

timeout_ms: (default=300_000) Default timeout (in milliseconds) for receiving ESP-NOW messages. If *timeout_ms* is less than zero, then wait forever. The timeout can also be provided as arg to *`recv()`*/*`irecv()`*/*`recvinto()`*.

rate: (ESP32 only) Set the transmission data rate for ESP-NOW packets. The default setting is *`espnow.RATE_1M`*. It's recommended to use one of the other `espnow.RATE_nnn` constants to set this, but it's also possible to pass an integer corresponding to the enum wifi_phy_rate_t. This parameter is actually *write-only* due to ESP-IDF not providing any means for querying the radio interface's rate parameter. See also *Long Range Mode*. This API currently doesn't work on ESP32-C6.

Returns:

`None` or the value of the parameter being queried.

Raises:

- `OSError(num, "ESP_ERR_ESPNOW_NOT_INIT")` if not initialised.
- `ValueError()` on invalid configuration options or values.

Sending and Receiving Data

A wifi interface (`network.WLAN.IF_STA` or `network.WLAN.IF_AP`) must be *`active()`* before messages can be sent or received, but it is not necessary to connect or configure the WLAN interface. For example:

```
import network

sta = network.WLAN(network.WLAN.IF_STA)
sta.active(True)
sta.disconnect()    # For ESP8266
```

Note: The ESP8266 has a *feature* that causes it to automatically reconnect to the last wifi Access Point when set `active(True)` (even after reboot/reset). This reduces the reliability of receiving ESP-NOW messages (see *ESPNow and Wifi Operation*). You can avoid this by calling `disconnect()` after `active(True)`.

ESPNow.**send**(*mac*, *msg*[, *sync*])

ESPNow.**send**(*msg*) *(ESP32 only)*

Send the data contained in `msg` to the peer with given network `mac` address. In the second form, `mac=None` and `sync=True`. The peer must be registered with `ESPNow.add_peer()` before the message can be sent.

Arguments:

- *mac*: byte string exactly `espnow.ADDR_LEN` (6 bytes) long or `None`. If *mac* is `None` (ESP32 only) the message will be sent to all registered peers, except any broadcast or multicast MAC addresses.
- *msg*: string or byte-string up to `espnow.MAX_DATA_LEN` (250 or 1470) bytes long.
- *sync*:
  - `True`: (default) send `msg` to the peer(s) and wait for a response (or not).
  - `False` send `msg` and return immediately. Responses from the peers will be discarded.

Returns:

`True` if `sync=False` or if `sync=True` and *all* peers respond, else `False`.

Raises:

- `OSError(num, "ESP_ERR_ESPNOW_NOT_INIT")` if not initialised.
- `OSError(num, "ESP_ERR_ESPNOW_NOT_FOUND")` if peer is not registered.
- `OSError(num, "ESP_ERR_ESPNOW_IF")` the wifi interface is not `active()`.
- `OSError(num, "ESP_ERR_ESPNOW_NO_MEM")` internal ESP-NOW buffers are full.
- `ValueError()` or `TypeError()` on invalid values or types for the parameters.

Note: A peer will respond with success if its wifi interface is `active()` and set to the same channel as the sender, regardless of whether it has initialised it's ESP-NOW system or is actively listening for ESP-NOW traffic (see the Espressif ESP-NOW docs).

ESPNow.**recv**([*timeout_ms*])

Wait for an incoming message and return the `mac` address of the peer and the message. **Note**: It is **not** necessary to register a peer (using `add_peer()`) to receive a message from that peer.

Arguments:

- *timeout_ms*: (Optional): May have the following values.
  - `0`: No timeout. Return immediately if no data is available;
  - `> 0`: Specify a timeout value in milliseconds;
  - `< 0`: Do not timeout, ie. wait forever for new messages; or
  - `None` (or not provided): Use the default timeout value set with `ESPNow.config()`.

Returns:

- `(None, None)` if timeout is reached before a message is received, or
- `[mac, msg]`: where:
  - `mac` is a bytestring containing the address of the device which sent the message, and
  - `msg` is a bytestring containing the message.

Raises:

- `OSError(num, "ESP_ERR_ESPNOW_NOT_INIT")` if not initialised.
- `OSError(num, "ESP_ERR_ESPNOW_IF")` if the wifi interface is not *active()*.
- `ValueError()` on invalid *timeout_ms* values.

ESPNow.recv() will allocate new storage for the returned list and the `peer` and `msg` bytestrings. This can lead to memory fragmentation if the data rate is high. See *ESPNow.irecv()* for a memory-friendly alternative.

ESPNow.**irecv**(*[timeout_ms]*)

Works like *ESPNow.recv()* but will reuse internal bytearrays to store the return values: `[mac, msg]`, so that no new memory is allocated on each call.

Arguments:

timeout_ms: (Optional) Timeout in milliseconds (see *ESPNow.recv()*).

Returns:

- As for *ESPNow.recv()*, except that `msg` is a bytearray, instead of a bytestring. On the ESP8266, `mac` will also be a bytearray.

Raises:

- See *ESPNow.recv()*.

Note: You may also read messages by iterating over the ESPNow object, which will use the *irecv()* method for alloc-free reads, eg:

```
import espnow
e = espnow.ESPNow(); e.active(True)
for mac, msg in e:
    print(mac, msg)
    if mac is None:   # mac, msg will equal (None, None) on timeout
        break
```

ESPNow.**recvinto**(*data*[, *timeout_ms*])

Wait for an incoming message and return the length of the message in bytes. This is the low-level method used by both *recv()* and *irecv()* to read messages.

Arguments:

data: A list of at least two elements, `[peer, msg]`. `msg` must be a bytearray large enough to hold the received message (recommended at least *espnow.MAX_DATA_LEN*). On the ESP8266, `peer` should be a bytearray of 6 bytes. The MAC address of the sender and the message will be stored in these bytearrays (see Note on ESP32 below).

timeout_ms: (Optional) Timeout in milliseconds (see *ESPNow.recv()*).

Returns:

- Length of message in bytes or 0 if *timeout_ms* is reached before a message is received.

Raises:

- See *ESPNow.recv()*.
- This function will also raise a `ValueError` if the received message is too large for the provided buffer. If this error is raised, received message(s) will be lost.

Note: On the ESP32:

- It is unnecessary to provide a bytearray in the first element of the `data` list because it will be replaced by a reference to a unique `peer` address in the **peer device table** (see *ESPNow.peers_table*).
- If the list is at least 4 elements long, the rssi and timestamp values will be saved as the 3rd and 4th elements.

ESPNow.**any**()

Check if data is available to be read with *ESPNow.recv()*.

For more sophisticated querying of available characters use *select.poll()*:

```
import select
import espnow

e = espnow.ESPNow()
poll = select.poll()
poll.register(e, select.POLLIN)
poll.poll(timeout)
```

Returns:

`True` if data is available to be read, else `False`.

ESPNow.**stats**() *(ESP32 only)*

Returns:

A 5-tuple containing the number of packets sent/received/lost:

```
(tx_pkts, tx_responses, tx_failures, rx_packets, rx_dropped_packets)
```

Incoming packets are *dropped* when the recv buffers are full. To reduce packet loss, increase the `rxbuf` config parameters and ensure you are reading messages as quickly as possible.

Note: Dropped packets will still be acknowledged to the sender as received.

Peer Management

On ESP32 devices, the Espressif ESP-NOW software requires that other devices (peers) must be *registered* using *add_peer()* before we can *send()* them messages (this is *not* enforced on ESP8266 devices). It is **not** necessary to register a peer to receive an un-encrypted message from that peer.

Encrypted messages: To receive an *encrypted* message, the receiving device must first register the sender and use the same encryption keys as the sender (PMK and LMK) (see *set_pmk()* and *add_peer()*.

ESPNow.**set_pmk**(*pmk*)

Set the Primary Master Key (PMK) which is used to encrypt the Local Master Keys (LMK) for encrypting messages. If this is not set, a default PMK is used by the underlying Espressif ESP-NOW software stack.

Note: messages will only be encrypted if *lmk* is also set in *ESPNow.add_peer()* (see Security in the Espressif API docs).

Arguments:

pmk: Must be a byte string, bytearray or string of length *espnow.KEY_LEN* (16 bytes).

Returns:

`None`

Raises:

`ValueError()` on invalid *pmk* values.

ESPNow.**add_peer**(*mac*[, *lmk*][, *channel*][, *ifidx*][, *encrypt*])

ESPNow.**add_peer**(*mac*, *param=value*, ...) *(ESP32 only)*

Add/register the provided *mac* address as a peer. Additional parameters may also be specified as positional or keyword arguments (any parameter set to `None` will be set to it's default value):

Arguments:

- *mac*: The MAC address of the peer (as a 6-byte byte-string).
- *lmk*: The Local Master Key (LMK) key used to encrypt data transfers with this peer (unless the *encrypt* parameter is set to `False`). Must be:
  - a byte-string or bytearray or string of length `espnow.KEY_LEN` (16 bytes), or
  - any non `True` python value (default= `b''`), signifying an *empty* key which will disable encryption.
- *channel*: The wifi channel (2.4GHz) to communicate with this peer. Must be an integer from 0 to 14. If channel is set to 0 the current channel of the wifi device will be used, if channel is set to another value then this must match the channel currently configured on the interface (see `WLAN.config()`). (default=0)
- *ifidx*: (ESP32 only) Index of the wifi interface which will be used to send data to this peer. Must be an integer set to `network.WLAN.IF_STA` (=0) or `network.WLAN.IF_AP` (=1). (default=0/`network.WLAN.IF_STA`). See *ESPNow and Wifi Operation* below for more information.
- *encrypt*: (ESP32 only) If set to `True` data exchanged with this peer will be encrypted with the PMK and LMK. (default = `True` if *lmk* is set to a valid key, else `False`)

ESP8266: Keyword args may not be used on the ESP8266.

Note: The maximum number of peers which may be registered is 20 (*`espnow.MAX_TOTAL_PEER_NUM`*), with a maximum of 6 (*`espnow.MAX_ENCRYPT_PEER_NUM`*) of those peers with encryption enabled (see ESP_NOW_MAX_ENCRYPT_PEER_NUM in the Espressif API docs).

Raises:

- `OSError(num, "ESP_ERR_ESPNOW_NOT_INIT")` if not initialised.
- `OSError(num, "ESP_ERR_ESPNOW_EXIST")` if *mac* is already registered.
- `OSError(num, "ESP_ERR_ESPNOW_FULL")` if too many peers are already registered.
- `OSError(num, "ESP_ERR_ESPNOW_CHAN")` if a channel value was set that doesn't match the channel currently configured for this interface.
- `ValueError()` or `TypeError()` on invalid keyword args or values.

ESPNow.**del_peer**(*mac*)

Deregister the peer associated with the provided *mac* address.

Returns:

`None`

Raises:

- `OSError(num, "ESP_ERR_ESPNOW_NOT_INIT")` if not initialised.
- `OSError(num, "ESP_ERR_ESPNOW_NOT_FOUND")` if *mac* is not registered.
- `ValueError()` or `TypeError()` on invalid *mac* values.

ESPNow.**get_peer**(*mac) (ESP32 only*)

Return information on a registered peer.

Returns:

`(mac, lmk, channel, ifidx, encrypt)`: a tuple of the "peer info" associated with the given *mac* address.

Raises:

- `OSError(num, "ESP_ERR_ESPNOW_NOT_INIT")` if not initialised.
- `OSError(num, "ESP_ERR_ESPNOW_NOT_FOUND")` if *mac* is not registered.
- `ValueError()` or `TypeError()` on invalid *mac* values.

`ESPNow.`**`peer_count`**`()` *(ESP32 only)*

Return the number of registered peers:

- `(peer_num, encrypt_num)`: where
  - `peer_num` is the number of peers which are registered, and
  - `encrypt_num` is the number of encrypted peers.

`ESPNow.`**`get_peers`**`()` *(ESP32 only)*

Return the "peer info" parameters for all the registered peers (as a tuple of tuples).

`ESPNow.`**`mod_peer`**`(`*mac, lmk, [channel], [ifidx], [encrypt])* *(ESP32 only)*

`ESPNow.`**`mod_peer`**`(`*mac, 'param'=value, ...)* *(ESP32 only)*

Modify the parameters of the peer associated with the provided *mac* address. Parameters may be provided as positional or keyword arguments (see *`ESPNow.add_peer()`*). Any parameter that is not set (or set to `None`) will retain the existing value for that parameter.

Callback Methods

`ESPNow.`**`irq`**`(`*callback) (ESP32 only*)

Set a callback function to be called *as soon as possible* after a message has been received from another ESPNow device. The callback function will be called with the *`ESPNow`* instance object as an argument. For more reliable operation, it is recommended to read out as many messages as are available when the callback is invoked and to set the read timeout to zero, eg:

```
def recv_cb(e):
    while True:  # Read out all messages waiting in the buffer
        mac, msg = e.irecv(0)  # Don't wait if no messages left
        if mac is None:
            return
        print(mac, msg)
e.irq(recv_cb)
```

The *`irq()`* callback method is an alternative method for processing incoming messages, especially if the data rate is moderate and the device is *not too busy* but there are some caveats:

- The scheduler stack *can* overflow and callbacks will be missed if packets are arriving at a sufficient rate or if other MicroPython components (eg, bluetooth, machine.Pin.irq(), machine.timer, i2s, …) are exercising the scheduler stack. This method may be less reliable for dealing with bursts of messages, or high throughput or on a device which is busy dealing with other hardware operations.
- For more information on *scheduled* function callbacks see: *`micropython.schedule()`*.

Constants

espnow.**MAX_DATA_LEN** (*=250 or 1470 for ESPNow V1 or V2*)

espnow.**KEY_LEN** (*=16*)

espnow.**ADDR_LEN** (*=6*)

espnow.**MAX_TOTAL_PEER_NUM** (*=20*)

espnow.**MAX_ENCRYPT_PEER_NUM** (*=6*)

The following constants correspond to different transmit data rates on ESP32 only. Lower data rates are generally more reliable over long distances:

espnow.**RATE_LORA_250K**

espnow.**RATE_LORA_500K**

See *Long Range Mode*.

espnow.**RATE_1M**

espnow.**RATE_2M**

espnow.**RATE_5M**

espnow.**RATE_6M**

espnow.**RATE_11M**

espnow.**RATE_12M**

espnow.**RATE_24M**

espnow.**RATE_54M**

Unless using the two proprietary long range data rates, only the sender must configure the data rate.

Long Range Mode

(ESP32 Only, except ESP32-C2)

To use the *espnow.RATE_LORA_250K* and *espnow.RATE_LORA_500K* data rates, first set the *WLAN* interface object to long-range mode, i.e.:

```
import network, espnow
sta = network.WLAN(network.WLAN.IF_STA)
sta.active(True)
sta.config(channel=6, protocol=WLAN.PROTOCOL_LR)  # Set on sender & receiver
e = espnow.ESPNow()
e.config(rate=espnow.RATE_LORA_250K)  # Needed on sender only
```

For more information about the limitations of long-range mode, see *WLAN.PROTOCOL_LR*.

Exceptions

If the underlying Espressif ESP-NOW software stack returns an error code, the MicroPython espnow module will raise an `OSError(errnum, errstring)` exception where `errstring` is set to the name of one of the error codes identified in the Espressif ESP-NOW docs. For example:

```
try:
    e.send(peer, 'Hello')
except OSError as err:
    if len(err.args) < 2:
        raise err
    if err.args[1] == 'ESP_ERR_ESPNOW_NOT_INIT':
        e.active(True)
    elif err.args[1] == 'ESP_ERR_ESPNOW_NOT_FOUND':
        e.add_peer(peer)
    elif err.args[1] == 'ESP_ERR_ESPNOW_IF':
        network.WLAN(network.WLAN.IF_STA).active(True)
    else:
        raise err
```

Wifi Signal Strength (RSSI) - (ESP32 only)

The ESPNow object maintains a **peer device table** which contains the signal strength and timestamp of the last received message from all hosts. The **peer device table** can be accessed using *ESPNow.peers_table* and can be used to track device proximity and identify *nearest neighbours* in a network of peer devices. This feature is **not** available on ESP8266 devices.

ESPNow.**peers_table**

A reference to the **peer device table**: a dict of known peer devices and rssi values:

```
{peer: [rssi, time_ms], ...}
```

where:

- `peer` is the peer MAC address (as *bytes*);
- `rssi` is the wifi signal strength in dBm (-127 to 0) of the last message received from the peer; and
- `time_ms` is the time the message was received (in milliseconds since system boot - wraps every 12 days).

Example:

```
>>> e.peers_table
{b'\xaa\xaa\xaa\xaa\xaa\xaa': [-31, 18372],
 b'\xbb\xbb\xbb\xbb\xbb\xbb': [-43, 12541]}
```

Note: the `mac` addresses returned by *recv()* are references to the `peer` key values in the **peer device table**.

Note: RSSI and timestamp values in the device table are updated only when the message is read by the application.

Supporting asyncio

A supplementary module (*aioespnow*) is available to provide *asyncio* support.

Note: Asyncio support is available on all ESP32 targets as well as those ESP8266 boards which include the asyncio module (ie. ESP8266 devices with at least 2MB flash memory).

A small async server example:

```
import network
import aioespnow
import asyncio
```

(continues on next page)

(continued from previous page)

```
# A WLAN interface must be active to send()/recv()
network.WLAN(network.WLAN.IF_STA).active(True)

e = aioespnow.AIOESPNow()  # Returns AIOESPNow enhanced with async support
e.active(True)
peer = b'\xbb\xbb\xbb\xbb\xbb\xbb'
e.add_peer(peer)

# Send a periodic ping to a peer
async def heartbeat(e, peer, period=30):
    while True:
        if not await e.asend(peer, b'ping'):
            print("Heartbeat: peer not responding:", peer)
        else:
            print("Heartbeat: ping", peer)
        await asyncio.sleep(period)

# Echo any received messages back to the sender
async def echo_server(e):
    async for mac, msg in e:
        print("Echo:", msg)
        try:
            await e.asend(mac, msg)
        except OSError as err:
            if len(err.args) > 1 and err.args[1] == 'ESP_ERR_ESPNOW_NOT_FOUND':
                e.add_peer(mac)
                await e.asend(mac, msg)

async def main(e, peer, timeout, period):
    asyncio.create_task(heartbeat(e, peer, period))
    asyncio.create_task(echo_server(e))
    await asyncio.sleep(timeout)

asyncio.run(main(e, peer, 120, 10))
```

class aioespnow.**AIOESPNow**

The *AIOESPNow* class inherits all the methods of *ESPNow* and extends the interface with the following async methods.

async AIOESPNow.arecv()

Asyncio support for *ESPNow.recv()*. Note that this method does not take a timeout value as argument.

async AIOESPNow.airecv()

Asyncio support for *ESPNow.irecv()*. Note that this method does not take a timeout value as argument.

async AIOESPNow.asend(mac, msg, sync=True)

async AIOESPNow.asend(msg)

Asyncio support for *ESPNow.send()*.

AIOESPNow.**_aiter__**() / *async AIOESPNow.__anext__*()

AIOESPNow also supports reading incoming messages by asynchronous iteration using `async for`; eg:

```
e = AIOESPNow()
e.active(True)
async def recv_till_halt(e):
```

(continues on next page)

(continued from previous page)

```
    async for mac, msg in e:
        print(mac, msg)
        if msg == b'halt':
          break
asyncio.run(recv_till_halt(e))
```

Broadcast and Multicast

All active ESPNow clients will receive messages sent to their MAC address and all devices (**except ESP8266 devices**) will also receive messages sent to the *broadcast* MAC address (`b'\xff\xff\xff\xff\xff\xff'`) or any multicast MAC address.

All ESPNow devices (including ESP8266 devices) can also send messages to the broadcast MAC address or any multicast MAC address.

To *`send()`* a broadcast message, the broadcast (or multicast) MAC address must first be registered using *`add_peer()`*. *`send()`* will always return `True` for broadcasts, regardless of whether any devices receive the message. It is not permitted to encrypt messages sent to the broadcast address or any multicast address.

Note: *`ESPNow.send(None, msg)`* will send to all registered peers *except* the broadcast address. To send a broadcast or multicast message, you must specify the broadcast (or multicast) MAC address as the peer. For example:

```
bcast = b'\xff' * 6
e.add_peer(bcast)
e.send(bcast, "Hello World!")
```

ESPNow and Wifi Operation

ESPNow messages may be sent and received on any *`active() WLAN`* interface (`network.WLAN.IF_STA` or `network.WLAN.IF_AP`), even if that interface is also connected to a wifi network or configured as an access point. When an ESP32 or ESP8266 device connects to a Wifi Access Point (see ESP32 Quickref) the following things happen which affect ESPNow communications:

1. Wifi Power-saving Mode (*`network.WLAN.PM_PERFORMANCE`*) is automatically activated and
2. The radio on the esp device changes wifi `channel` to match the channel used by the Access Point.

Wifi Power-saving Mode: (see Espressif Docs) The power saving mode causes the device to turn off the radio periodically (typically for hundreds of milliseconds), making it unreliable in receiving ESPNow messages. This can be resolved by either of:

1. Disabling the power-saving mode on the STA_IF interface;
   - Use `sta.config(pm=sta.PM_NONE)`
2. Turning on the AP_IF interface, which will disable the power saving mode. However, the device will then be advertising an active wifi access point.
   - You **may** also choose to send your messages via the AP_IF interface, but this is not necessary.
   - ESP8266 peers must send messages to this AP_IF interface (see below).
3. Configuring ESPNow clients to retry sending messages.

Receiving messages from an ESP8266 device: Strangely, an ESP32 device connected to a wifi network using method 1 or 2 above, will receive ESPNow messages sent to the STA_IF MAC address from another ESP32 device, but will **reject**

messages from an ESP8266 device!!!. To receive messages from an ESP8266 device, the AP_IF interface must be set to `active(True)` **and** messages must be sent to the AP_IF MAC address.

Managing wifi channels: Any other ESPNow devices wishing to communicate with a device which is also connected to a Wifi Access Point MUST use the same channel. A common scenario is where one ESPNow device is connected to a wifi router and acts as a proxy for messages from a group of sensors connected via ESPNow:

Proxy:

```
import network, time, espnow

sta, ap = wifi_reset()  # Reset wifi to AP off, STA on and disconnected
sta.connect('myssid', 'mypassword')
while not sta.isconnected():  # Wait until connected...
    time.sleep(0.1)
sta.config(pm=sta.PM_NONE)  # ..then disable power saving

# Print the wifi channel used AFTER finished connecting to access point
print("Proxy running on channel:", sta.config("channel"))
e = espnow.ESPNow(); e.active(True)
for peer, msg in e:
    # Receive espnow messages and forward them to MQTT broker over wifi
```

Sensor:

```
import network, espnow

sta, ap = wifi_reset()    # Reset wifi to AP off, STA on and disconnected
sta.config(channel=6)     # Change to the channel used by the proxy above.
peer = b'0\xaa\xaa\xaa\xaa\xaa'  # MAC address of proxy
e = espnow.ESPNow(); e.active(True);
e.add_peer(peer)
while True:
    msg = read_sensor()
    e.send(peer, msg)
    time.sleep(1)
```

Other issues to take care with when using ESPNow with wifi are:

- **Set WIFI to known state on startup:** MicroPython does not reset the wifi peripheral after a soft reset. This can lead to unexpected behaviour. To guarantee the wifi is reset to a known state after a soft reset make sure you deactivate the STA_IF and AP_IF before setting them to the desired state at startup, eg.:

  ```
  import network, time

  def wifi_reset():   # Reset wifi to AP_IF off, STA_IF on and disconnected
    sta = network.WLAN(network.WLAN.IF_STA); sta.active(False)
    ap = network.WLAN(network.WLAN.IF_AP); ap.active(False)
    sta.active(True)
    while not sta.active():
        time.sleep(0.1)
    sta.disconnect()   # For ESP8266
    while sta.isconnected():
        time.sleep(0.1)
    return sta, ap

  sta, ap = wifi_reset()
  ```

  Remember that a soft reset occurs every time you connect to the device REPL and when you type `ctrl-D`.

- **STA_IF and AP_IF always operate on the same channel:** the AP_IF will change channel when you connect to a wifi network; regardless of the channel you set for the AP_IF (see Attention Note 3). After all, there is really only one wifi radio on the device, which is shared by the STA_IF and AP_IF virtual devices.
- **Disable automatic channel assignment on your wifi router:** If the wifi router for your wifi network is configured to automatically assign the wifi channel, it may change the channel for the network if it detects interference from other wifi routers. When this occurs, the ESP devices connected to the wifi network will also change channels to match the router, but other ESPNow-only devices will remain on the previous channel and communication will be lost. To mitigate this, either set your wifi router to use a fixed wifi channel or configure your devices to re-scan the wifi channels if they are unable to find their expected peers on the current channel.
- **MicroPython re-scans wifi channels when trying to reconnect:** If the esp device is connected to a Wifi Access Point that goes down, MicroPython will automatically start scanning channels in an attempt to reconnect to the Access Point. This means ESPNow messages will be lost while scanning for the AP. This can be disabled by `sta.config(reconnects=0)`, which will also disable the automatic reconnection after losing connection.
- Some versions of the ESP IDF only permit sending ESPNow packets from the STA_IF interface to peers which have been registered on the same wifi channel as the STA_IF:

```
ESPNOW: Peer channel is not equal to the home channel, send fail!
```

ESPNow and Sleep Modes

The *machine.lightsleep([time_ms])* and *machine.deepsleep([time_ms])* functions can be used to put the ESP32 and peripherals (including the WiFi and Bluetooth radios) to sleep. This is useful in many applications to conserve battery power. However, applications must disable the WLAN peripheral (using *active(False)*) before entering light or deep sleep (see Sleep Modes). Otherwise the WiFi radio may not be initialised properly after wake from sleep. If the `STA_IF` and `AP_IF` interfaces have both been set *active(True)* then both interfaces should be set *active(False)* before entering any sleep mode.

Example: deep sleep:

```
import network, machine, espnow

sta, ap = wifi_reset()              # Reset wifi to AP off, STA on and disconnected
peer = b'0\xaa\xaa\xaa\xaa\xaa'     # MAC address of peer
e = espnow.ESPNow()
e.active(True)
e.add_peer(peer)                    # Register peer on STA_IF

print('Sending ping...')
if not e.send(peer, b'ping'):
  print('Ping failed!')
e.active(False)
sta.active(False)                   # Disable the wifi before sleep
print('Going to sleep...')
machine.deepsleep(10000)            # Sleep for 10 seconds then reboot
```

Example: light sleep:

```
import network, machine, espnow

sta, ap = wifi_reset()              # Reset wifi to AP off, STA on and disconnected
sta.config(channel=6)
peer = b'0\xaa\xaa\xaa\xaa\xaa'     # MAC address of peer
e = espnow.ESPNow()
```

(continues on next page)

(continued from previous page)

```
e.active(True)
e.add_peer(peer)                    # Register peer on STA_IF

while True:
  print('Sending ping...')
  if not e.send(peer, b'ping'):
    print('Ping failed!')
  sta.active(False)                 # Disable the wifi before sleep
  print('Going to sleep...')
  machine.lightsleep(10000)         # Sleep for 10 seconds
  sta.active(True)
  sta.config(channel=6)             # Wifi loses config after lightsleep()
```

1.3.4 Libraries specific to NXP i.MXRT

The following libraries are specific to the NXP i.MXRT family of microcontrollers.

`mimxrt` — functionality specific to NXP i.MXRT

The `mimxrt` module contains functions and classes specific to the NXP i.MXRT family of microcontrollers.

Classes

class Flash – access to built-in flash storage

This class gives access to the SPI flash memory.

In most cases, to store persistent data on the device, you'll want to use a higher-level abstraction, for example the filesystem via Python's standard file API, but this interface is useful to *customise the filesystem configuration* or implement a low-level storage system for your application.

Constructors

class mimxrt.**Flash**

Gets the singleton object for accessing the SPI flash memory.

Methods

Flash.**readblocks**(*block_num*, *buf*)

Flash.**readblocks**(*block_num*, *buf*, *offset*)

Flash.**writeblocks**(*block_num*, *buf*)

Flash.**writeblocks**(*block_num*, *buf*, *offset*)

Flash.**ioctl**(*cmd*, *arg*)

These methods implement the simple and extended *block protocol* defined by `vfs.AbstractBlockDev`.

The block size can be queried by calling `ioctl(5, 0)`. Block numbers are relative to the start of the user flash storage area, not the physical start of flash memory.

1.3.5 Libraries specific to the RP2040

The following libraries are specific to the RP2040, as used in the Raspberry Pi Pico.

rp2 — functionality specific to the RP2040

The `rp2` module contains functions and classes specific to the RP2040, as used in the Raspberry Pi Pico.

See the RP2040 Python datasheet for more information, and pico-micropython-examples for example code.

PIO related functions

The `rp2` module includes functions for assembling PIO programs.

For running PIO programs, see *`rp2.StateMachine`*.

`rp2.`**`asm_pio`**(*, *out_init=None*, *set_init=None*, *sideset_init=None*, *side_pindir=False*, *in_shiftdir=PIO.SHIFT_LEFT*, *out_shiftdir=PIO.SHIFT_LEFT*, *autopush=False*, *autopull=False*, *push_thresh=32*, *pull_thresh=32*, *fifo_join=PIO.JOIN_NONE*)

Assemble a PIO program.

The following parameters control the initial state of the GPIO pins, as one of *`PIO.IN_LOW`*, *`PIO.IN_HIGH`*, *`PIO.OUT_LOW`* or *`PIO.OUT_HIGH`*. If the program uses more than one pin, provide a tuple, e.g. `out_init=(PIO.OUT_LOW, PIO.OUT_LOW)`.

- *out_init* configures the pins used for `out()` instructions.
- *set_init* configures the pins used for `set()` instructions. There can be at most 5.
- *sideset_init* configures the pins used for `.side()` modifiers. There can be at most 5.
- *side_pindir* when set to `True` configures `.side()` modifiers to be used for pin directions, instead of pin values (the default, when `False`).

The following parameters are used by default, but can be overridden in *`StateMachine.init()`*:

- *in_shiftdir* is the default direction the ISR will shift, either *`PIO.SHIFT_LEFT`* or *`PIO.SHIFT_RIGHT`*.
- *out_shiftdir* is the default direction the OSR will shift, either *`PIO.SHIFT_LEFT`* or *`PIO.SHIFT_RIGHT`*.
- *push_thresh* is the threshold in bits before auto-push or conditional re-pushing is triggered.
- *pull_thresh* is the threshold in bits before auto-pull or conditional re-pulling is triggered.

The remaining parameters are:

- *autopush* configures whether auto-push is enabled.
- *autopull* configures whether auto-pull is enabled.
- *fifo_join* configures whether the 4-word TX and RX FIFOs should be combined into a single 8-word FIFO for one direction only. The options are *`PIO.JOIN_NONE`*, *`PIO.JOIN_RX`* and *`PIO.JOIN_TX`*.

`rp2.`**`asm_pio_encode`**(*instr*, *sideset_count*, *sideset_opt=False*)

Assemble a single PIO instruction. You usually want to use *`asm_pio()`* instead.

```
>>> rp2.asm_pio_encode("set(0, 1)", 0)
57345
```

rp2.**bootsel_button**()

Temporarily turns the QSPI_SS pin into an input and reads its value, returning 1 for low and 0 for high. On a typical RP2040 board with a BOOTSEL button, a return value of 1 indicates that the button is pressed.

Since this function temporarily disables access to the external flash memory, it also temporarily disables interrupts and the other core to prevent them from trying to execute code from flash.

class rp2.**PIOASMError**

This exception is raised from *asm_pio()* or *asm_pio_encode()* if there is an error assembling a PIO program.

PIO assembly language instructions

PIO state machines are programmed in a custom assembly language with nine core PIO-machine instructions. In MicroPython, PIO assembly routines are written as a Python function with the decorator `@rp2.asm_pio()`, and they use Python syntax. Such routines support standard Python variables and arithmetic, as well as the following custom functions that encode PIO instructions and direct the assembler. See sec 3.4 of the RP2040 datasheet for further details.

wrap_target()

Specify the location where execution continues after program wrapping. By default this is the start of the PIO routine.

wrap()

Specify the location where the program finishes and wraps around. If this directive is not used then it is added automatically at the end of the PIO routine. Wrapping does not cost any execution cycles.

label(label)

Define a label called *label* at the current location. *label* can be a string or integer.

word(instr, label=None)

Insert an arbitrary 16-bit word in the assembled output.

- *instr*: the 16-bit value
- *label*: if given, look up the label and logical-or the label's value with *instr*

jmp(…)

This instruction takes two forms:

jmp(label)

- *label*: label to jump to unconditionally

jmp(cond, label)

- *cond*: the condition to check, one of:
  - `not_x`, `not_y`: true if register is zero
  - `x_dec`, `y_dec`: true if register is non-zero, and do post decrement
  - `x_not_y`: true if X is not equal to Y
  - `pin`: true if the input pin is set
  - `not_osre`: true if OSR is not empty (hasn't reached its threshold)
- *label*: label to jump to if condition is true

wait(polarity, src, index)

Block, waiting for high/low on a pin or IRQ line.

- *polarity*: 0 or 1, whether to wait for a low or high value

- *src*: one of: `gpio` (absolute pin), `pin` (pin relative to StateMachine's `in_base` argument), `irq`
- *index*: 0-31, the index for *src*

in_(src, bit_count)
Shift data in from *src* to ISR.

- *src*: one of: `pins`, `x`, `y`, `null`, `isr`, `osr`
- *bit_count*: number of bits to shift in (1-32)

out(dest, bit_count)
Shift data out from OSR to *dest*.

- *dest*: one of: `pins`, `x`, `y`, `pindirs`, `pc`, `isr`, `exec`
- *bit_count*: number of bits to shift out (1-32)

push(…)
Push ISR to the RX FIFO, then clear ISR to zero. This instruction takes the following forms:

- push()
- push(block)
- push(noblock)
- push(iffull)
- push(iffull, block)
- push(iffull, noblock)

If `block` is used then the instruction stalls if the RX FIFO is full. The default is to block. If `iffull` is used then it only pushes if the input shift count has reached its threshold.

pull(…)
Pull from the TX FIFO into OSR. This instruction takes the following forms:

- pull()
- pull(block)
- pull(noblock)
- pull(ifempty)
- pull(ifempty, block)
- pull(ifempty, noblock)

If `block` is used then the instruction stalls if the TX FIFO is empty. The default is to block. If `ifempty` is used then it only pulls if the output shift count has reached its threshold.

mov(dest, src)
Move into *dest* the value from *src*.

- *dest*: one of: `pins`, `x`, `y`, `exec`, `pc`, `isr`, `osr`
- *src*: one of: `pins`, `x`, `y`, `null`, `status`, `isr`, `osr`; this argument can be optionally modified by wrapping it in `invert()` or `reverse()` (but not both together)

irq(…)
Set or clear an IRQ flag. This instruction takes two forms:

irq(index)

- *index*: 0-7, or `rel(0)` to `rel(7)`

irq(mode, index)

- *mode*: one of: `block`, `clear`
- *index*: 0-7, or `rel(0)` to `rel(7)`

If `block` is used then the instruction stalls until the flag is cleared by another entity. If `clear` is used then the flag is cleared instead of being set. Relative IRQ indices add the state machine ID to the IRQ index with modulo-4 addition. IRQs 0-3 are visible from to the processor, 4-7 are internal to the state machines.

set(dest, data)
Set *dest* with the value *data*.

- *dest*: `pins`, `x`, `y`, `pindirs`
- *data*: value (0-31)

nop()
This is a pseudoinstruction that assembles to `mov(y, y)` and has no side effect.

.side(value)
This is a modifier which can be applied to any instruction, and is used to control side-set pin values.

- *value*: the value (bits) to output on the side-set pins

.delay(value)
This is a modifier which can be applied to any instruction, and specifies how many cycles to delay for after the instruction executes.

- *value*: cycles to delay, 0-31 (maximum value reduced if side-set pins are used)

[value]
This is a modifier and is equivalent to `.delay(value)`.

Classes

class DMA – access to the RP2040's DMA controller

The *DMA* class offers access to the RP2040's Direct Memory Access (DMA) controller, providing the ability to move data between memory blocks and/or IO registers. The DMA controller has its own, separate read and write bus master connections onto the bus fabric and each DMA channel can independently read data from one address and write it back to another address, optionally incrementing one or both pointers, allowing it to perform transfers on behalf of the processor while the processor carries out other tasks or enters a low power state. The RP2040's DMA controller has 12 independent DMA channels that can run concurrently. For full details of the RP2040's DMA system see section 2.5 of the RP2040 Datasheet.

The companion class *DMATimer* provides access to the DMA controller's pacing timers. These timers can be used to control the speed of transfers into memory or peripherals that do not have their own transfer request signalling.

Examples

The simplest use of the DMA controller is to move data from one block of memory to another. This can be accomplished with the following code:

```
a = bytearray(32*1024)
b = bytearray(32*1024)
d = rp2.DMA()
c = d.pack_ctrl()  # Just use the default control value.
# The count is in 'transfers', which defaults to four-byte words, so divide length by␣
↪4
d.config(read=a, write=b, count=len(a)//4, ctrl=c, trigger=True)
# Wait for completion
while d.active():
    pass
```

Note that while this example sits in an idle loop while it waits for the transfer to complete, the program could just as well do some useful work in this time instead.

Another, perhaps more common use of the DMA controller is to transfer between memory and an IO peripheral. In this situation the address of the IO register does not change for each transfer but the memory address needs to be incremented. It is also necessary to control the pace of the transfer so as to not write data before it can be accepted by a peripheral or read it before the data is ready, and this can be controlled with the `treq_sel` field of the DMA channel's control register. The various fields of the control register for each DMA channel can be packed using the *DMA.pack_ctrl()* method and unpacked using the *DMA.unpack_ctrl()* static method. Code to transfer data from a byte array to the TX FIFO of a PIO state machine, one byte at a time, looks like this:

```
# pio_num is index of the PIO block being used, sm_num is the state machine in that␣
↪block.
# my_state_machine is an rp2.PIO() instance.
DATA_REQUEST_INDEX = (pio_num << 3) + sm_num

src_data = bytearray(1024)
d = rp2.DMA()

# Transfer bytes, rather than words, don't increment the write address and pace the␣
↪transfer.
c = d.pack_ctrl(size=0, inc_write=False, treq_sel=DATA_REQUEST_INDEX)

d.config(
    read=src_data,
    write=my_state_machine,
    count=len(src_data),
    ctrl=c,
    trigger=True
)
```

Note that in this example the value given for the write address is just the PIO state machine to which we are sending the data. This works because PIO state machines present the buffer protocol, allowing direct access to their data FIFO registers.

Constructor

class rp2.**DMA**

Claim one of the DMA controller channels for exclusive use.

Methods

DMA.**config**(*read=None*, *write=None*, *count=None*, *ctrl=None*, *trigger=False*)

Configure the DMA registers for the channel and optionally start the transfer. Parameters are:

- *read*: The address from which the DMA controller will start reading data or an object that will provide data to be read. It can be an integer or any object that supports the buffer protocol.
- *write*: The address to which the DMA controller will start writing or an object into which data will be written. It can be an integer or any object that supports the buffer protocol.
- *count*: The number of bus transfers that will execute before this channel stops. Note that this is the number of transfers, not the number of bytes. If the transfers are 2 or 4 bytes wide then the total amount of data moved (and thus the size of required buffer) needs to be multiplied accordingly.
- *ctrl*: The value for the DMA control register. This is an integer value that is typically packed using the *DMA.pack_ctrl()*.
- *trigger*: Optionally commence the transfer immediately.

DMA.**irq**(*handler=None*, *hard=False*)

Returns the IRQ object for this DMA channel and optionally configures it.

DMA.**close**()

Release the claim on the underlying DMA channel and free the interrupt handler. The *DMA* object can not be used after this operation.

DMA.**pack_ctrl**(*default=None*, ***kwargs*)

Pack the values provided in the keyword arguments into the named fields of a new control register value. Any field that is not provided will be set to a default value. The default will either be taken from the provided `default` value, or if that is not given, a default suitable for the current channel; setting this to the current value of the *DMA.ctrl* attribute provides an easy way to override a subset of the fields.

The keys for the keyword arguments can be any key returned by the *DMA.unpack_ctrl()* method. The writable values are:

- *enable*: `bool` Set to enable the channel (default: `True`).
- *high_pri*: `bool` Make this channel's bus traffic high priority (default: `False`).
- *size*: `int` Transfer size: 0=byte, 1=half word, 2=word (default: 2).
- *inc_read*: `bool` Increment the read address after each transfer (default: `True`).
- *inc_write*: `bool` Increment the write address after each transfer (default: `True`).
- *ring_size*: `int` If non-zero, only the bottom `ring_size` bits of one address register will change when an address is incremented, causing the address to wrap at the next `1 << ring_size` byte boundary. Which address is wrapped is controlled by the `ring_sel` flag. A zero value disables address wrapping (default: 0).
- *ring_sel*: `bool` Set to `False` to have the `ring_size` apply to the read address or `True` to apply to the write address (default: `False`).
- *chain_to*: `int` The channel number for a channel to trigger after this transfer completes. Setting this value to this DMA object's own channel number disables chaining (this is the default).

- *treq_sel*: `int` Select a Transfer Request signal. See section 2.5.3 in the RP2040 datasheet for details. You may also pass a *DMATimer* instance to use that timer for pacing the transfer. The default is `0x3f` (`PERMANENT`), which means the DMA channel runs at full bus speed without waiting for any external request.
- *irq_quiet*: `bool` Do not generate interrupt at the end of each transfer. Interrupts will instead be generated when a zero value is written to the trigger register, which will halt a sequence of chained transfers (default: `True`).
- *bswap*: `bool` If set to true, bytes in words or half-words will be reversed before writing (default: `False`).
- *sniff_en*: `bool` Set to `True` to allow data to be accessed by the chip's sniff hardware (default: `False`).
- *write_err*: `bool` Setting this to `True` will clear a previously reported write error (default: `False`).
- *read_err*: `bool` Setting this to `True` will clear a previously reported read error (default: `False`).

See the description of the `CH0_CTRL_TRIG` register in section 2.5.7 of the RP2040 datasheet for details of all of these fields.

`DMA.`**`unpack_ctrl`**`(`*value*`)`

Unpack a value for a DMA channel control register into a dictionary with key/value pairs for each of the fields in the control register. *value* is the `ctrl` register value to unpack.

This method will return values for all the keys that can be passed to `DMA.pack_ctrl`. In addition, it will also return the read-only flags in the control register: `busy`, which goes high when a transfer starts and low when it ends, and `ahb_err`, which is the logical OR of the `read_err` and `write_err` flags. These values will be ignored when packing, so that the dictionary created by unpacking a control register can be used directly as the keyword arguments for packing.

`DMA.`**`active`**`(`[*value*]`)`

Gets or sets whether the DMA channel is currently running.

```
>>> sm.active()
0
>>> sm.active(1)
>>> while sm.active():
...     pass
```

Attributes

`DMA.`**`read`**

This attribute reflects the address from which the next bus transfer will read. It may be written with either an integer or an object that supports the buffer protocol and doing so has immediate effect.

`DMA.`**`write`**

This attribute reflects the address to which the next bus transfer will write. It may be written with either an integer or an object that supports the buffer protocol and doing so has immediate effect.

`DMA.`**`count`**

Reading this attribute will return the number of remaining bus transfers in the *current* transfer sequence. Writing this attribute sets the total number of transfers to be the *next* transfer sequence.

`DMA.`**`ctrl`**

This attribute reflects DMA channel control register. It is typically written with an integer packed using the *DMA.pack_ctrl()* method. The returned register value can be unpacked using the *DMA.unpack_ctrl()* method.

DMA.**channel**

The channel number of the DMA channel. This can be passed in the `chain_to` argument of *DMA.pack_ctrl()* on another channel to allow DMA chaining.

DMA.**registers**

This attribute is an array-like object that allows direct access to the DMA channel's registers. The index is by word, rather than by byte, so the register indices are the register address offsets divided by 4. See the RP2040 data sheet for register details.

Chaining and trigger register access

The DMA controller in the RP2040 offers a couple advanced features to allow one DMA channel to initiate a transfer on another channel. One is the use of the `chain_to` value in the control register and the other is writing to one of the DMA channel's registers that has a trigger effect. When coupled with the ability to have one DMA channel write directly to the *DMA.registers* of another channel, this allows for complex transactions to be performed without any CPU intervention.

Below is an example of using both chaining and register triggering to implement gathering of multiple blocks of data into a single destination. Full details of these features can be found in section 2.5 of the RP2040 data sheet and the code below is a Pythonic version of the example in sub-section 2.5.6.2.

```
from rp2 import DMA
from uctypes import addressof
from array import array

def gather_strings(string_list, buf):
    # We use two DMA channels. The first sends lengths and source addresses from the␣
↪gather
    # list to the registers of the second. The second copies the data itself.
    gather_dma = DMA()
    buffer_dma = DMA()

    # Pack up length/address pairs to be sent to the registers.
    gather_list = array("I")

    for s in string_list:
        gather_list.append(len(s))
        gather_list.append(addressof(s))

    gather_list.append(0)
    gather_list.append(0)

    # When writing to the registers of the second DMA channel, we need to wrap the
    # write address on an 8-byte (1<<3 bytes) boundary. We write to the ``TRANS_
↪COUNT``
    # and ``READ_ADD_TRIG`` registers in the last register alias (registers 14 and␣
↪15).
    gather_ctrl = gather_dma.pack_ctrl(ring_size=3, ring_sel=True)
    gather_dma.config(
        read=gather_list, write=buffer_dma.registers[14:16],
        count=2, ctrl=gather_ctrl
    )

    # When copying the data, the transfer size is single bytes, and when completed we␣
↪need
    # to chain back to the start another gather DMA transaction.
```

(continues on next page)

(continued from previous page)

```
    buffer_ctrl = buffer_dma.pack_ctrl(size=0, chain_to=gather_dma.channel)
    # The read and count values will be set by the other DMA channel.
    buffer_dma.config(write=buf, ctrl=buffer_ctrl)

    # Set the transfer in motion.
    gather_dma.active(1)

    # Wait until all the register values have been sent
    end_address = addressof(gather_list) + 4 * len(gather_list)
    while gather_dma.read != end_address:
        pass

input = ["This is ", "a ", "test", " of the scatter", " gather", " process"]
output = bytearray(64)

print(output)
gather_strings(input, output)
print(output)
```

This example idles while waiting for the transfer to complete; alternatively it could set an interrupt handler and return immediately.

class DMATimer – pacing timers for DMA transfers

The RP2040 and RP2350 DMA controllers provide four "pacing" timers that can be used to control the rate at which DMA transfers take place. In the absence of specifying a transfer request signal using the `treq_sel` parameter in the control configuration the DMA controller will try to transfer data as fast as the bus will allow, which can be as fast as the system clock speed. Often I/O operations should happen at some lower rate, and it is also sometimes valuable to moderate the rate of transfers from memory to memory in order to avoid overloading the bus (particularly when using external PSRAM, which is much slower than the on-chip SRAM). By using a *DMATimer* the user can select a rate that is a rational fraction of the system clock speed. Each timer can independently trigger transfer requests at rate that is `X/Y` times the system clock, where `X < Y`.

DMATimer objects can be used directly as the value for treq_sel passed into the *DMA.pack_ctrl()* function, since the value if `int(dma_timer)` is the index of the transfer request selector for the timer. Thus if you want to pace a transfer to run at 10,000 operations per second you can use:

```
dma = rp2.DMA()
timer = rp2.DMATimer(freq=10000)
ctrl = d.pack_ctrl(treq_sel=timer)  # Default control value with paced by the timer
dma.config(read=src, write=dst, count=length, ctrl=ctrl, trigger=True)
```

Note: The underlying DMA pacing timer will get released when a *DMATimer* gets garbage collected. If you are setting in motion a DMA transfer that is not expected to complete before the timer object goes out of scope then it is a good idea to keep a reference to it so that the timer does not get reassigned to some other caller (which might change the frequency on you).

Constructor

class rp2.**DMATimer**(*timer_id=None*, ***, *freq=None*, *ratio=None*)

Claim one of the DMA pacing timers for exclusive use and optionally set the frequency or ratio.

- *timer_id*: Which timer to use. Leave empty to select any unclaimed timer.
- *freq*: The optional value to assign to the `freq` attribute.
- *ratio*: The optional value to assign to the `ratio` attribute.

If both `freq` and `ratio` are provided then `ratio` is used.

Methods

DMATimer.**close**()

Release the exclusive claim on the underlying timer.

Attributes

DMATimer.**ratio**

Set or read the `(X, Y)` tuple for the system clock division ratio. When setting the ratio both X and Y need to in the range 0 < X, Y < 65536.

DMATimer.**freq**

Set or read the DMATimer frequency in Hz. When setting, the frequency will be set to the closest frequency that can be achieved by the divider. Reading the frequency back will show the actual selected frequency, to the nearest 1Hz. The requested value needs be less than or equal to the system clock speed and greater than or equal to 1/65535 of the system clock.

class Flash – access to built-in flash storage

This class gives access to the SPI flash memory.

In most cases, to store persistent data on the device, you'll want to use a higher-level abstraction, for example the filesystem via Python's standard file API, but this interface is useful to *customise the filesystem configuration* or implement a low-level storage system for your application.

Constructors

class rp2.**Flash**

Gets the singleton object for accessing the SPI flash memory.

Methods

Flash.**readblocks**(*block_num*, *buf*)
Flash.**readblocks**(*block_num*, *buf*, *offset*)

Flash.**writeblocks**(*block_num*, *buf*)
Flash.**writeblocks**(*block_num*, *buf*, *offset*)

Flash.**ioctl**(*cmd*, *arg*)

These methods implement the simple and extended *block protocol* defined by `vfs.AbstractBlockDev`.

class PIO – advanced PIO usage

The `PIO` class gives access to an instance of the RP2040's PIO (programmable I/O) interface.

The preferred way to interact with PIO is using `rp2.StateMachine`, the PIO class is for advanced use.

For assembling PIO programs, see `rp2.asm_pio()`.

Constructors

class rp2.**PIO**(*id*)

Gets the PIO instance numbered *id*. The RP2040 has two PIO instances, numbered 0 and 1.

Raises a `ValueError` if any other argument is provided.

Methods

PIO.**gpio_base**([*base*])

Query and optionally set the current GPIO base for this PIO instance.

If an argument is given then it must be a pin (or integer corresponding to a pin number), restricted to either GPIO0 or GPIO16. The GPIO base will then be set to that pin. Setting the GPIO base must be done before any programs are added or state machines created.

Returns the current GPIO base pin.

PIO.**add_program**(*program*)

Add the *program* to the instruction memory of this PIO instance.

The amount of memory available for programs on each PIO instance is limited. If there isn't enough space left in the PIO's program memory this method will raise `OSError(ENOMEM)`.

PIO.**remove_program**([*program*])

Remove *program* from the instruction memory of this PIO instance.

If no program is provided, it removes all programs.

It is not an error to remove a program which has already been removed.

PIO.**state_machine**(*id*[, *program*, ...])

Gets the state machine numbered *id*. On the RP2040, each PIO instance has four state machines, numbered 0 to 3.

Optionally initialize it with a *program*: see `StateMachine.init`.

```
>>> rp2.PIO(1).state_machine(3)
StateMachine(7)
```

PIO.**irq**(*handler=None*, *trigger=IRQ_SM0 | IRQ_SM1 | IRQ_SM2 | IRQ_SM3*, *hard=False*)

Returns the IRQ object for this PIO instance.

MicroPython only uses IRQ 0 on each PIO instance. IRQ 1 is not available.

Optionally configure it.

Constants

PIO.**IN_LOW**

PIO.**IN_HIGH**

PIO.**OUT_LOW**

PIO.**OUT_HIGH**

These constants are used for the *out_init*, *set_init*, and *sideset_init* arguments to *asm_pio*.

PIO.**SHIFT_LEFT**

PIO.**SHIFT_RIGHT**

These constants are used for the *in_shiftdir* and *out_shiftdir* arguments to *asm_pio* or *StateMachine.init*.

PIO.**JOIN_NONE**

PIO.**JOIN_TX**

PIO.**JOIN_RX**

These constants are used for the *fifo_join* argument to *asm_pio*.

PIO.**IRQ_SM0**

PIO.**IRQ_SM1**

PIO.**IRQ_SM2**

PIO.**IRQ_SM3**

These constants are used for the *trigger* argument to *PIO.irq*.

class StateMachine – access to the RP2040's programmable I/O interface

The *StateMachine* class gives access to the RP2040's PIO (programmable I/O) interface.

For assembling PIO programs, see *rp2.asm_pio()*.

Constructors

class rp2.**StateMachine**(*id*[, *program*, ...])

Get the state machine numbered *id*. The RP2040 has two identical PIO instances, each with 4 state machines: so there are 8 state machines in total, numbered 0 to 7.

Optionally initialize it with the given program *program*: see *StateMachine.init*.

Methods

StateMachine.**init**(*program*, *freq=-1*, ***, *in_base=None*, *out_base=None*, *set_base=None*, *jmp_pin=None*, *sideset_base=None*, *in_shiftdir=None*, *out_shiftdir=None*, *push_thresh=None*, *pull_thresh=None*)

Configure the state machine instance to run the given *program*.

The program is added to the instruction memory of this PIO instance. If the instruction memory already contains this program, then its offset is reused so as to save on instruction memory.

- *freq* is the frequency in Hz to run the state machine at. Defaults to the system clock frequency.

  The clock divider is computed as `system clock frequency / freq`, so there can be slight rounding errors.

  The minimum possible clock divider is one 65536th of the system clock: so at the default system clock frequency of 125MHz, the minimum value of *freq* is `1908`. To run state machines at slower frequencies, you'll need to reduce the system clock speed with *machine.freq()*.

- *in_base* is the first pin to use for `in()` instructions.
- *out_base* is the first pin to use for `out()` instructions.
- *set_base* is the first pin to use for `set()` instructions.
- *jmp_pin* is the first pin to use for `jmp(pin, ...)` instructions.
- *sideset_base* is the first pin to use for side-setting.
- *in_shiftdir* is the direction the ISR will shift, either *PIO.SHIFT_LEFT* or *PIO.SHIFT_RIGHT*.
- *out_shiftdir* is the direction the OSR will shift, either *PIO.SHIFT_LEFT* or *PIO.SHIFT_RIGHT*.
- *push_thresh* is the threshold in bits before auto-push or conditional re-pushing is triggered.
- *pull_thresh* is the threshold in bits before auto-pull or conditional re-pulling is triggered.

Note: pins used for *in_base* need to be configured manually for input (or otherwise) so that the PIO can see the desired signal (they could be input pins, output pins, or connected to a different peripheral). The *jmp_pin* can also be configured manually, but by default will be an input pin.

StateMachine.**active**([*value*])

Gets or sets whether the state machine is currently running.

```
>>> sm.active()
True
>>> sm.active(0)
False
```

StateMachine.**restart**()

Restarts the state machine and jumps to the beginning of the program.

This method clears the state machine's internal state using the RP2040's `SM_RESTART` register. This includes:

- input and output shift counters
- the contents of the input shift register
- the delay counter
- the waiting-on-IRQ state
- a stalled instruction run using *StateMachine.exec()*

StateMachine.**exec**(*instr*)

Execute a single PIO instruction.

If *instr* is a string then uses *asm_pio_encode* to encode the instruction from the given string.

```
>>> sm.exec("set(0, 1)")
```

If *instr* is an integer then it is treated as an already encoded PIO machine code instruction to be executed.

```
>>> sm.exec(rp2.asm_pio_encode("out(y, 8)", 0))
```

StateMachine.**get**(*buf=None*, *shift=0*)

Pull a word from the state machine's RX FIFO.

If the FIFO is empty, it blocks until data arrives (i.e. the state machine pushes a word).

The value is shifted right by *shift* bits before returning, i.e. the return value is `word >> shift`.

StateMachine.**put**(*value*, *shift=0*)

Push words onto the state machine's TX FIFO.

value can be an integer, an array of type `B`, `H` or `I`, or a *bytearray*.

This method will block until all words have been written to the FIFO. If the FIFO is, or becomes, full, the method will block until the state machine pulls enough words to complete the write.

Each word is first shifted left by *shift* bits, i.e. the state machine receives `word << shift`.

StateMachine.**rx_fifo**()

Returns the number of words in the state machine's RX FIFO. A value of 0 indicates the FIFO is empty.

Useful for checking if data is waiting to be read, before calling *StateMachine.get()*.

StateMachine.**tx_fifo**()

Returns the number of words in the state machine's TX FIFO. A value of 0 indicates the FIFO is empty.

Useful for checking if there is space to push another word using *StateMachine.put()*.

StateMachine.**irq**(*handler=None*, *trigger=0 | 1*, *hard=False*)

Returns the IRQ object for the given StateMachine.

Optionally configure it.

Buffer protocol

The StateMachine class supports the *buffer protocol*, allowing direct access to the transmit and receive FIFOs for each state machine. This is primarily in order to allow StateMachine objects to be passed directly as the read or write parameters when configuring a *rp2.DMA()* channel.

1.3.6 Libraries specific to Zephyr

The following libraries are specific to the Zephyr port.

zephyr — functionality specific to the Zephyr port

The zephyr module contains functions and classes specific to the Zephyr port.

Functions

zephyr.**is_preempt_thread**()

Returns true if the current thread is a preemptible thread.

Zephyr preemptible threads are those with non-negative priority values (low priority levels), which therefore, can be supplanted as soon as a higher or equal priority thread becomes ready.

zephyr.**current_tid**()

Returns the thread id of the current thread, which is used to reference the thread.

zephyr.**thread_analyze**(*cpu*)

Runs the Zephyr debug thread analyzer on the current thread on the given cpu and prints stack size statistics in the format:

> "thread_name-20s: STACK: unused available_stack_space usage stack_space_used / stack_size (percent_stack_space_used %); CPU: cpu_utilization %"
>
> - *CPU utilization is only printed if runtime statistics are configured via the ``CONFIG_THREAD_RUNTIME_STATS`` kconfig*

This function can only be accessed if CONFIG_THREAD_ANALYZER is configured for the port in zephyr/prj.conf. For more information, see documentation for Zephyr thread analyzer.

Note that the cpu argument is only used in Zephyr v4.0.0 and newer and ignored otherwise.

zephyr.**shell_exec**(*cmd_in*)

Executes the given command on an UART backend. This function can only be accessed if CONFIG_SHELL_BACKEND_SERIAL is configured for the port in zephyr/prj.conf.

A list of possible commands can be found in the documentation for Zephyr shell commands.

Classes

class DiskAccess – access to disk storage

Uses Zephyr Disk Access API.

This class allows access to storage devices on the board, such as support for SD card controllers and interfacing with SD cards via SPI. Disk devices are automatically detected and initialized on boot using Zephyr devicetree data.

The Zephyr disk access class enables the transfer of data between a disk device and an accessible memory buffer given a disk name, buffer, starting disk block, and number of sectors to read. MicroPython reads as many blocks as necessary to fill the buffer, so the number of sectors to read is found by dividing the buffer length by block size of the disk.

Constructors

class zephyr.**DiskAccess**(*disk_name*)

Gets an object for accessing disk memory of the specific disk. For accessing an SD card on the mimxrt1050_evk, `disk_name` would be `SDHC`. See board documentation and devicetree for usable disk names for your board (ex. RT boards use style USDHC#).

Methods

DiskAccess.**readblocks**(*block_num*, *buf*)

DiskAccess.**readblocks**(*block_num*, *buf*, *offset*)

DiskAccess.**writeblocks**(*block_num*, *buf*)

DiskAccess.**writeblocks**(*block_num*, *buf*, *offset*)

DiskAccess.**ioctl**(*cmd*, *arg*)

These methods implement the simple and extended *block protocol* defined by `vfs.AbstractBlockDev`.

class Display – access to Zephyr Displays

Uses the Zephyr Display API.

This class allows access to Zephyr-provided Displays (zephyr,display chosen Node and zephyr,displays Node) via an API reproducing the Zephyr one.

Constructors

class zephyr.**Display**(*id*)

Gets an object for accessing a Display identified by `id`.

`id` can be an integer (`0`, `1`...) or a string (`"ssd1306@3c"`) identifying a display node by its position or by its node identifiers.

Methods

Display.**write**(*buf*[, *x*[, *y*[, *size_x*[, *size_y*]]]])

Write a buffer-protocol object in the Display's Pixel Format to the display.

Optionally x and y position, x size, and y size can be specified.

Display.**rgb**(*r*, *g*, *b*)

Convert a RGB color to the Display's Pixel Format.

Display.**capabilities**()

Retrieve a tuple describing the display in the format:

```
(X Size, Y Size, Supported PFs, Current PF, Current Orientation, Misc
Characteristics, Current PF as framebuf format)
```

Display.**format**([*format*])

Get and set the Pixel Format of the Display.

`Display.`**`blanking`**`(`*value*`)`

Enable or disable blanking.

`Display.`**`clear`**`()`

Clear the Display.

`Display.`**`set_brightness`**`(`*value*`)`

Set the Display's brightness from `0` to `255`.

`Display.`**`set_contrast`**`(`*value*`)`

Set the Display's contrast from `0` to `255`.

`Display.`**`orientation`**`(`[*orientation*]`)`

Get and set the Orientation of the Display.

`Display.`**`as_framebuf`**`()`

If *`framebuf`* is enabled, generate a *`framebuf.FrameBuffer`* instance augmented with a `show()` function that directly maps to the display with the currently configured settings.

class FlashArea – access to built-in flash storage

Uses Zephyr flash map API.

This class allows access to device flash partition data. Flash area structs consist of a globally unique ID number, the name of the flash device the partition is in, the start offset (expressed in relation to the flash memory beginning address per partition), and the size of the partition that the device represents. For fixed flash partitions, data from the device tree is used; however, fixed flash partitioning is not enforced in MicroPython because MCUBoot is not enabled.

Constructors

`class` `zephyr.`**`FlashArea`**`(`*id*, *block_size*`)`

Gets an object for accessing flash memory at partition specified by `id` and with block size of `block_size`.

`id` values are integers correlating to fixed flash partitions defined in the devicetree. A commonly used partition is the designated flash storage area defined as `FlashArea.STORAGE` if `FLASH_AREA_LABEL_EXISTS(storage)` returns true at boot. Zephyr devicetree fixed flash partitions are `boot_partition`, `slot0_partition`, `slot1_partition`, and `scratch_partition`. Because MCUBoot is not enabled by default for MicroPython, these fixed partitions can be accessed by ID integer values 1, 2, 3, and 4, respectively.

Methods

`FlashArea.`**`readblocks`**`(`*block_num*, *buf*`)`

`FlashArea.`**`readblocks`**`(`*block_num*, *buf*, *offset*`)`

`FlashArea.`**`writeblocks`**`(`*block_num*, *buf*`)`

`FlashArea.`**`writeblocks`**`(`*block_num*, *buf*, *offset*`)`

`FlashArea.`**`ioctl`**`(`*cmd*, *arg*`)`

These methods implement the simple and extended *block protocol* defined by *`vfs.AbstractBlockDev`*.

Additional Modules

zsensor — Zephyr sensor bindings

The zsensor module contains a class for using sensors with Zephyr.

class Sensor — sensor control for the Zephyr port

Use this class to access data from sensors on your board. See Zephyr documentation for sensor usage here: Sensors.

Sensors are defined in the Zephyr devicetree for each board. The quantities that a given sensor can measure are called a sensor channels. Sensors can have multiple channels to represent different axes of one property or different properties a sensor can measure. See *Channels* below for defined sensor channels. Each channel may have multiple attributes that can be changed and/or queried. See *Channel Attributes* below for defined sensor channel attributes.

Constructor

class zsensor.**Sensor**(*device_name*)

Device names are defined in the devicetree for your board. For example, the device name for the accelerometer in the FRDM-k64f board is "FXOS8700".

Methods

Sensor.**measure**()

Obtains a measurement sample from the sensor device using Zephyr sensor_sample_fetch and stores it in an internal driver buffer as a useful value, a pair of (integer part of value, fractional part of value in 1-millionths). Returns none if successful or OSError value if failure.

Sensor.**get_float**(*sensor_channel*)

Returns the value of the sensor measurement sample as a float.

Sensor.**get_micros**(*sensor_channel*)

Returns the value of the sensor measurement sample in millionths. (Ex. value of `(1, 500000)` returns as `1500000`)

Sensor.**get_millis**(*sensor_channel*)

Returns the value of sensor measurement sample in thousandths. (Ex. value of `(1, 500000)` returns as `1500`)

Sensor.**get_int**(*sensor_channel*)

Returns only the integer value of the measurement sample. (Ex. value of `(1, 500000)` returns as `1`)

Sensor.**attr_set**(*sensor_channel*, *channel_attribute*, *val1*[, *val2*])

Set the given channel's attribute to the given value. `val1` may be a float, in which case `val2` is not given, or `val1` can be used for the value's integer part and `val2` for the value's fractional part in millionths.

Returns `None` if successful, or raises `OSError`.

Sensor.**attr_get_float**(*sensor_channel*, *channel_attribute*)

Returns the value of the sensor channel's attribute as a float.

Many sensors do not support this or any other of the `attr_get` methods.

Sensor.**attr_get_micros**(*sensor_channel*, *channel_attribute*)

Returns the value of the sensor channel's attribute in millionths. (Ex. value of `(1, 500000)` returns as `1500000`)

Sensor.**attr_get_millis**(*sensor_channel*, *channel_attribute*)

Returns the value of the sensor channel's attribute in thousandths. (Ex. value of `(1, 500000)` returns as `1500`)

Sensor.**attr_get_int**(*sensor_channel*, *channel_attribute*)

Returns only the integer value of the channel's attribute. (Ex. value of `(1, 500000)` returns as `1`)

Channels

zsensor.**ACCEL_X**

Acceleration on the X axis, in m/s^2.

zsensor.**ACCEL_Y**

Acceleration on the Y axis, in m/s^2.

zsensor.**ACCEL_Z**

Acceleration on the Z axis, in m/s^2.

zsensor.**ACCEL_XYZ**

Pseudo-channel representing all three accelerometer axes. Used for *Sensor.attr_set()* and the Sensor.attr_get_xxx() methods.

zsensor.**GYRO_X**

Angular velocity around the X axis, in radians/s.

zsensor.**GYRO_Y**

Angular velocity around the Y axis, in radians/s.

zsensor.**GYRO_Z**

Angular velocity around the Z axis, in radians/s.

zsensor.**GYRO_XYZ**

Pseudo-channel representing all three gyroscope axes. Used for *Sensor.attr_set()* and the Sensor.attr_get_xxx() methods.

zsensor.**MAGN_X**

Magnetic field on the X axis, in Gauss.

zsensor.**MAGN_Y**

Magnetic field on the Y axis, in Gauss.

zsensor.**MAGN_Z**

Magnetic field on the Z axis, in Gauss.

zsensor.**DIE_TEMP**

Device die temperature in degrees Celsius.

zsensor.**PRESS**

Pressure in kilopascal.

zsensor.**PROX**

Proximity. Dimensionless. A value of 1 indicates that an object is close.

zsensor.**HUMIDITY**
: Humidity, in percent.

zsensor.**LIGHT**
: Illuminance in visible spectrum, in lux.

zsensor.**ALTITUDE**
: Altitude, in meters.

Channel Attributes

zsensor.**ATTR_SAMPLING_FREQUENCY**
: Sensor sampling frequency, i.e. how many times a second the sensor takes a measurement.

zsensor.**ATTR_LOWER_THRESH**
: Lower threshold for trigger.

zsensor.**ATTR_UPPER_THRESH**
: Upper threshold for trigger.

zsensor.**ATTR_SLOPE_TH**
: Threshold for any-motion (slope) trigger.

zsensor.**ATTR_SLOPE_DUR**
: Duration for which the slope values needs to be outside the threshold for the trigger to fire.

zsensor.**ATTR_HYSTERESIS**

zsensor.**ATTR_OVERSAMPLING**
: Oversampling factor.

zsensor.**ATTR_FULL_SCALE**
: Sensor range, in SI units.

zsensor.**ATTR_OFFSET**
: The sensor value returned will be altered by the amount indicated by offset: final_value = sensor_value + offset.

zsensor.**ATTR_CALIB_TARGET**
: Calibration target. This will be used by the internal chip's algorithms to calibrate itself on a certain axis, or all of them.

zsensor.**ATTR_CONFIGURATION**
: Configure the operating modes of a sensor.

zsensor.**ATTR_CALIBRATION**
: Set a calibration value needed by a sensor.

zsensor.**ATTR_FEATURE_MASK**
: Enable/disable sensor features.

zsensor.**ATTR_ALERT**
: Alert threshold or alert enable/disable.

zsensor.**ATTR_FF_DUR**
: Free-fall duration represented in milliseconds. If the sampling frequency is changed during runtime, this attribute should be set to adjust freefall duration to the new sampling frequency.

zsensor.**ATTR_BATCH_DURATION**

Hardware batch duration in ticks.

zsensor.**ATTR_GAIN**

zsensor.**ATTR_RESOLUTION**

1.4 Extending built-in libraries from Python

A subset of the built-in modules are able to be extended by Python code by providing a module of the same name in the filesystem. This extensibility applies to the following Python standard library modules which are built-in to the firmware: `array`, `binascii`, `collections`, `errno`, `gzip`, `hashlib`, `heapq`, `io`, `json`, `os`, `platform`, `random`, `re`, `select`, `socket`, `ssl`, `struct`, `time` `zlib`, as well as the MicroPython-specific `machine` module. All other built-in modules cannot be extended from the filesystem.

This allows the user to provide an extended implementation of a built-in library (perhaps to provide additional CPython compatibility or missing functionality). This is used extensively in *micropython-lib*, see *Package management* for more information. The filesystem module will typically do a wildcard import of the built-in module in order to inherit all the globals (classes, functions and variables) from the built-in.

In MicroPython v1.21.0 and higher, to prevent the filesystem module from importing itself, it can force an import of the built-in module it by temporarily clearing `sys.path` during the import. For example, to extend the `time` module from Python, a file named `time.py` on the filesystem would do the following:

```
_path = sys.path
sys.path = ()
try:
  from time import *
finally:
  sys.path = _path
  del _path

def extra_method():
  pass
```

The result is that `time.py` contains all the globals of the built-in `time` module, but adds `extra_method`.

In earlier versions of MicroPython, you can force an import of a built-in module by appending a `u` to the start of its name. For example, `import utime` instead of `import time`. For example, `time.py` on the filesystem could look like:

```
from utime import *

def extra_method():
  pass
```

This way is still supported, but the `sys.path` method described above is now preferred as the `u`-prefix will be removed from the names of built-in modules in a future version of MicroPython.

Other than when it specifically needs to force the use of the built-in module, code should always use `import module` *rather than* `import umodule`.

CHAPTER

TWO

MICROPYTHON LANGUAGE AND IMPLEMENTATION

MicroPython aims to implement the Python 3.4 standard (with selected features from later versions) with respect to language syntax, and most of the features of MicroPython are identical to those described by the "Language Reference" documentation at docs.python.org.

The MicroPython standard library is described in the *corresponding chapter*. The *MicroPython differences from CPython* chapter describes differences between MicroPython and CPython (which mostly concern standard library and types, but also some language-level features).

This chapter describes features and peculiarities of MicroPython implementation and the best practices to use them.

2.1 Glossary

baremetal
: A system without a (full-fledged) operating system, for example an *MCU*-based system. When running on a baremetal system, MicroPython effectively functions like a small operating system, running user programs and providing a command interpreter (*REPL*).

buffer protocol
: Any Python object that can be automatically converted into bytes, such as `bytes`, `bytearray`, `memoryview` and `str` objects, which all implement the "buffer protocol".

board
: Typically this refers to a printed circuit board (PCB) containing a *microcontroller* and supporting components. MicroPython firmware is typically provided per-board, as the firmware contains both MCU-specific functionality but also board-level functionality such as drivers or pin names.

bytecode
: A compact representation of a Python program that generated by compiling the Python source code. This is what the VM actually executes. Bytecode is typically generated automatically at runtime and is invisible to the user. Note that while *CPython* and MicroPython both use bytecode, the format is different. You can also pre-compile source code offline using the *cross-compiler*.

callee-owned tuple
: This is a MicroPython-specific construct where, for efficiency reasons, some built-in functions or methods may reuse the same underlying tuple object to return data. This avoids having to allocate a new tuple for every call, and reduces heap fragmentation. Programs should not hold references to callee-owned tuples and instead only extract data from them (or make a copy).

CircuitPython
: A variant of MicroPython developed by Adafruit Industries.

CPython
: CPython is the reference implementation of the Python programming language, and the most well-known one.

It is, however, one of many implementations (including Jython, IronPython, PyPy, and MicroPython). While MicroPython's implementation differs substantially from CPython, it aims to maintain as much compatibility as possible.

cross-compiler
Also known as `mpy-cross`. This tool runs on your PC and converts a *.py file* containing MicroPython code into a *.mpy file* containing MicroPython *bytecode*. This means it loads faster (the board doesn't have to compile the code), and uses less space on flash (the bytecode is more space efficient).

driver
A MicroPython library that implements support for a particular component, such as a sensor or display.

FFI
Acronym for Foreign Function Interface. A mechanism used by the *MicroPython Unix port* to access operating system functionality. This is not available on *baremetal* ports.

filesystem
Most MicroPython ports and boards provide a filesystem stored in flash that is available to user code via the standard Python file APIs such as `open()`. Some boards also make this internal filesystem accessible to the host via USB mass-storage.

frozen module
A Python module that has been cross compiled and bundled into the firmware image. This reduces RAM requirements as the code is executed directly from flash.

Garbage Collector
A background process that runs in Python (and MicroPython) to reclaim unused memory in the *heap*.

GPIO
General-purpose input/output. The simplest means to control electrical signals (commonly referred to as "pins") on a microcontroller. GPIO typically allows pins to be either input or output, and to set or get their digital value (logical "0" or "1"). MicroPython abstracts GPIO access using the `machine.Pin` and `machine.Signal` classes.

GPIO port
A group of *GPIO* pins, usually based on hardware properties of these pins (e.g. controllable by the same register).

heap
A region of RAM where MicroPython stores dynamic data. It is managed automatically by the *Garbage Collector*. Different MCUs and boards have vastly different amounts of RAM available for the heap, so this will affect how complex your program can be.

interned string
An optimisation used by MicroPython to improve the efficiency of working with strings. An interned string is referenced by its (unique) identity rather than its address and can therefore be quickly compared just by its identifier. It also means that identical strings can be de-duplicated in memory. String interning is almost always invisible to the user.

MCU
Microcontroller. Microcontrollers usually have much less resources than a desktop, laptop, or phone, but are smaller, cheaper and require much less power. MicroPython is designed to be small and optimized enough to run on an average modern microcontroller.

micropython-lib
MicroPython is (usually) distributed as a single executable/binary file with just few builtin modules. There is no extensive standard library comparable with *CPython*'s. Instead, there is a related, but separate project micropython-lib which provides implementations for many modules from CPython's standard library.

Some of the modules are implemented in pure Python, and are able to be used on all ports. However, the majority of these modules use *FFI* to access operating system functionality, and as such can only be used on the *MicroPython Unix port* (with limited support for Windows).

Unlike the *CPython* stdlib, micropython-lib modules are intended to be installed individually - either using manual copying or using *mip*.

MicroPython port
MicroPython supports different *boards*, RTOSes, and OSes, and can be relatively easily adapted to new systems. MicroPython with support for a particular system is called a "port" to that system. Different ports may have widely different functionality. This documentation is intended to be a reference of the generic APIs available across different ports ("MicroPython core"). Note that some ports may still omit some APIs described here (e.g. due to resource constraints). Any such differences, and port-specific extensions beyond the MicroPython core functionality, would be described in the separate port-specific documentation.

MicroPython Unix port
The unix port is one of the major *MicroPython ports*. It is intended to run on POSIX-compatible operating systems, like Linux, MacOS, FreeBSD, Solaris, etc. It also serves as the basis of Windows port. The Unix port is very useful for quick development and testing of the MicroPython language and machine-independent features. It can also function in a similar way to *CPython*'s `python` executable.

mip
A package installer for MicroPython (mip - "mip installs packages"). It installs MicroPython packages either from *micropython-lib*, GitHub, or arbitrary URLs. mip can be used on-device on network-capable boards, and internally by tools such as *mpremote*.

See *Package management* for more information on using `mip`.

mpremote
A tool for interacting with a MicroPython device. See *MicroPython remote control: mpremote*.

.mpy file
The output of the *cross-compiler*. A compiled form of a *.py file* that contains MicroPython *bytecode* instead of Python source code.

native
Usually refers to "native code", i.e. machine code for the target microcontroller (such as ARM Thumb, Xtensa, x86/x64). The `@native` decorator can be applied to a MicroPython function to generate native code instead of *bytecode* for that function, which will likely be faster but use more RAM.

port
Usually short for *MicroPython port*, but could also refer to *GPIO port*.

.py file
A file containing Python source code.

REPL
An acronym for "Read, Eval, Print, Loop". This is the interactive Python prompt, useful for debugging or testing short snippets of code. Most MicroPython boards make a REPL available over a UART, and this is typically accessible on a host PC via USB.

small integer
MicroPython optimises the internal representation of integers such that "small" values do not take up space on the heap, and calculations with them do not require heap allocation. On most 32-bit ports, this corresponds to values in the interval `-2**30 <= x < 2**30`, but this should be considered an implementation detail and not relied upon.

stream
Also known as a "file-like object". A Python object which provides sequential read-write access to the underlying data. A stream object implements a corresponding interface, which consists of methods like `read()`, `write()`, `readinto()`, `seek()`, `flush()`, `close()`, etc. A stream is an important concept in MicroPython; many I/O objects implement the stream interface, and thus can be used consistently and interchangeably in different contexts. For more information on streams in MicroPython, see the `io` module.

UART
: Acronym for "Universal Asynchronous Receiver/Transmitter". This is a peripheral that sends data over a pair of pins (TX & RX). Many boards include a way to make at least one of the UARTs available to a host PC as a serial port over USB.

upip
: A now-obsolete package manager for MicroPython, inspired by *CPython*'s pip, but much smaller and with reduced functionality. See its replacement, *mip*.

webrepl
: A way of connecting to the REPL (and transferring files) on a device over the internet from a browser. See https://micropython.org/webrepl

2.2 The MicroPython Interactive Interpreter Mode (aka REPL)

This section covers some characteristics of the MicroPython Interactive Interpreter Mode. A commonly used term for this is REPL (read-eval-print-loop) which will be used to refer to this interactive prompt.

2.2.1 Auto-indent

When typing python statements which end in a colon (for example if, for, while) then the prompt will change to three dots (...) and the cursor will be indented by 4 spaces. When you press return, the next line will continue at the same level of indentation for regular statements or an additional level of indentation where appropriate. If you press the backspace key then it will undo one level of indentation.

If your cursor is all the way back at the beginning, pressing RETURN will then execute the code that you've entered. The following shows what you'd see after entering a for statement (the underscore shows where the cursor winds up):

```
>>> for i in range(30):
...     _
```

If you then enter an if statement, an additional level of indentation will be provided:

```
>>> for i in range(30):
...     if i > 3:
...         _
```

Now enter `break` followed by RETURN and press BACKSPACE:

```
>>> for i in range(30):
...     if i > 3:
...         break
...     _
```

Finally type `print(i)`, press RETURN, press BACKSPACE and press RETURN again:

```
>>> for i in range(30):
...     if i > 3:
...         break
...     print(i)
...
0
1
2
```

(continues on next page)

(continued from previous page)

```
3
>>>
```

Auto-indent won't be applied if the previous two lines were all spaces. This means that you can finish entering a compound statement by pressing RETURN twice, and then a third press will finish and execute.

2.2.2 Auto-completion

While typing a command at the REPL, if the line typed so far corresponds to the beginning of the name of something, then pressing TAB will show possible things that could be entered. For example, first import the machine module by entering `import machine` and pressing RETURN. Then type `m` and press TAB and it should expand to `machine`. Enter a dot `.` and press TAB again. You should see something like:

```
>>> machine.
__name__        info            unique_id       reset
bootloader      freq            rng             idle
sleep           deepsleep       disable_irq     enable_irq
Pin
```

The word will be expanded as much as possible until multiple possibilities exist. For example, type `machine.Pin.AF3` and press TAB and it will expand to `machine.Pin.AF3_TIM`. Pressing TAB a second time will show the possible expansions:

```
>>> machine.Pin.AF3_TIM
AF3_TIM10       AF3_TIM11       AF3_TIM8        AF3_TIM9
>>> machine.Pin.AF3_TIM
```

2.2.3 Interrupting a running program

You can interrupt a running program by pressing Ctrl-C. This will raise a KeyboardInterrupt which will bring you back to the REPL, providing your program doesn't intercept the KeyboardInterrupt exception.

For example:

```
>>> for i in range(1000000):
...     print(i)
...
0
1
2
3
...
6466
6467
6468
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt:
>>>
```

2.2.4 Paste mode

If you want to paste some code into your terminal window, the auto-indent feature will mess things up. For example, if you had the following python code:

```
def foo():
    print('This is a test to show paste mode')
    print('Here is a second line')
foo()
```

and you try to paste this into the normal REPL, then you will see something like this:

```
>>> def foo():
...         print('This is a test to show paste mode')
...             print('Here is a second line')
...             foo()
...
Traceback (most recent call last):
  File "<stdin>", line 3
IndentationError: unexpected indent
```

If you press Ctrl-E, then you will enter paste mode, which essentially turns off the auto-indent feature, and changes the prompt from >>> to ===. For example:

```
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
=== def foo():
===     print('This is a test to show paste mode')
===     print('Here is a second line')
=== foo()
===
This is a test to show paste mode
Here is a second line
>>>
```

Paste Mode allows blank lines to be pasted. The pasted text is compiled as if it were a file. Pressing Ctrl-D exits paste mode and initiates the compilation.

2.2.5 Soft reset

A *Soft Reset* will reset the python interpreter, but tries not to reset the method by which you're connected to the MicroPython board (USB-serial, or Wifi).

You can perform a soft reset from the REPL by pressing Ctrl-D, or from your python code by executing:

```
machine.soft_reset()
```

For example, if you reset your MicroPython board, and you execute a dir() command, you'd see something like this:

```
>>> dir()
['__name__', 'pyb']
```

Now create some variables and repeat the dir() command:

```
>>> i = 1
>>> j = 23
```

(continues on next page)

(continued from previous page)

```
>>> x = 'abc'
>>> dir()
['j', 'x', '__name__', 'pyb', 'i']
>>>
```

Now if you enter Ctrl-D, and repeat the dir() command, you'll see that your variables no longer exist:

```
MPY: sync filesystems
MPY: soft reboot
MicroPython v1.5-51-g6f70283-dirty on 2015-10-30; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>> dir()
['__name__', 'pyb']
>>>
```

For more information about reset types and the startup process, see *Reset and Boot Sequence*.

2.2.6 The special variable _ (underscore)

When you use the REPL, you may perform computations and see the results. MicroPython stores the results of the previous statement in the variable _ (underscore). So you can use the underscore to save the result in a variable. For example:

```
>>> 1 + 2 + 3 + 4 + 5
15
>>> x = _
>>> x
15
>>>
```

2.2.7 Raw mode and raw-paste mode

Raw mode (also called raw REPL) is not something that a person would normally use. It is intended for programmatic use and essentially behaves like paste mode with echo turned off, and with optional flow control.

Raw mode is entered using Ctrl-A. You then send your python code, followed by a Ctrl-D. The Ctrl-D will be acknowledged by 'OK' and then the python code will be compiled and executed. Any output (or errors) will be sent back. Entering Ctrl-B will leave raw mode and return the regular (aka friendly) REPL.

Raw-paste mode is an additional mode within the raw REPL that includes flow control, and which compiles code as it receives it. This makes it more robust for high-speed transfer of code into the device, and it also uses less RAM when receiving because it does not need to store a verbatim copy of the code before compiling (unlike standard raw mode).

Raw-paste mode uses the following protocol:

1. Enter raw REPL as usual via ctrl-A.
2. Write 3 bytes: `b"\x05A\x01"` (ie ctrl-E then "A" then ctrl-A).
3. Read 2 bytes to determine if the device entered raw-paste mode:
    - If the result is `b"R\x00"` then the device understands the command but doesn't support raw paste.
    - If the result is `b"R\x01"` then the device does support raw paste and has entered this mode.
    - Otherwise the result should be `b"ra"` and the device doesn't support raw paste and the string `b"w REPL; CTRL-B to exit\r\n>"` should be read and discarded.

4. If the device is in raw-paste mode then continue, otherwise fallback to standard raw mode.
5. Read 2 bytes, this is the flow control window-size-increment (in bytes) stored as a 16-bit unsigned little endian integer. The initial value for the remaining-window-size variable should be set to this number.
6. Write out the code to the device:
    - While there are bytes to send, write up to the remaining-window-size worth of bytes, and decrease the remaining-window-size by the number of bytes written.
    - If the remaining-window-size is 0, or there is a byte waiting to read, read 1 byte. If this byte is `b"\x01"` then increase the remaining-window-size by the window-size-increment from step 5. If this byte is `b"\x04"` then the device wants to end the data reception, and `b"\x04"` should be written to the device and no more code sent after that. (Note: if there is a byte waiting to be read from the device then it does not need to be read and acted upon immediately, the device will continue to consume incoming bytes as long as reamining-window-size is greater than 0.)
7. When all code has been written to the device, write `b"\x04"` to indicate end-of-data.
8. Read from the device until `b"\x04"` is received. At this point the device has received and compiled all of the code that was sent and is executing it.
9. The device outputs any characters produced by the executing code. When (if) the code finishes `b"\x04"` will be output, followed by any exception that was uncaught, followed again by `b"\x04"`. It then goes back to the standard raw REPL and outputs `b">"`.

For example, starting at a new line at the normal (friendly) REPL, if you write:

```
b"\x01\x05A\x01print(123)\x04"
```

Then the device will respond with something like:

```
b"\r\nraw REPL; CTRL-B to exit\r\n>R\x01\x80\x00\x01\x04123\r\n\x04\x04>"
```

Broken down over time this looks like:

```
# Step 1: enter raw REPL
write: b"\x01"
read: b"\r\nraw REPL; CTRL-B to exit\r\n>"

# Step 2-5: enter raw-paste mode
write: b"\x05A\x01"
read: b"R\x01\x80\x00\x01"

# Step 6-8: write out code
write: b"print(123)\x04"
read: b"\x04"

# Step 9: code executes and result is read
read: b"123\r\n\x04\x04>"
```

In this case the flow control window-size-increment is 128 and there are two windows worth of data immediately available at the start, one from the initial window-size-increment value and one from the explicit `b"\x01"` value that is sent. So this means up to 256 bytes can be written to begin with before waiting or checking for more incoming flow-control characters.

The `tools/pyboard.py` program uses the raw REPL, including raw-paste mode, to execute Python code on a MicroPython-enabled board.

2.3 Reset and Boot Sequence

A device running MicroPython follows a particular boot sequence to start up and initialise itself after a reset.

2.3.1 Hard reset

Booting from hard reset is what happens when a board is first powered up, a cold boot. This is a complete reset of the MCU hardware.

The MicroPython port code initialises all essential hardware (including embedded clocks and power regulators, internal serial UART, etc), and then starts the MicroPython environment. Existing *RTC* configuration may be retained after a hard reset, but all other hardware state is cleared.

The same hard reset boot sequence can be triggered by a number of events such as:

- Python code executing `machine.reset()`.
- User presses a physical Reset button on the board (where applicable).
- Waking from deep sleep (on most ports).
- MCU hardware watchdog reset.
- MCU hardware brown out detector.

The details of hardware-specific reset triggers depend on the port and associated hardware. The `machine.reset_cause()` function can be used to further determine the cause of a reset.

2.3.2 Soft Reset

When MicroPython is already running, it's possible to trigger a soft reset by *typing Ctrl-D in the REPL* or executing `machine.soft_reset()`.

A soft reset clears the Python interpreter, frees all Python memory, and starts the MicroPython environment again.

State which is cleared by a soft reset includes:

- All Python variables, objects, imported modules, etc.
- Most peripherals configured using the *machine module*. There are very limited exceptions, for example *machine.Pin* modes (i.e. if a pin is input or output, high or low) are not reset on most ports. More advanced configuration such as `Pin.irq()` is always reset.
- Bluetooth.
- Network sockets. Open TCP sockets are closed cleanly with respect to the other party.
- Open files. The filesystem is left in a valid state.

Some system state remains the same after a soft reset, including:

- Any existing network connections (Ethernet, Wi-Fi, etc) remain active at the IP Network layer. Querying the *network interface from code* may indicate the network interface is still active with a configured IP address, etc.
- An active *REPL* appears continuous before and after soft reset, except in some unusual cases:
  - If the *machine.USBDevice* class has been used to create a custom USB interface then any built-in USB serial device will appear to disconnect and reconnect as the custom USB interface must be cleared during reset.
  - A serial UART REPL will restore its default hardware configuration (baud rate, etc).
- CPU clock speed is usually not changed by a soft reset.

- *RTC* configuration (i.e. setting of the current time) is not changed by soft reset.

2.3.3 Boot Sequence

When MicroPython boots following either a hard or soft reset, it follows this boot sequence in order:

_boot.py

This is an internal script *frozen into the MicroPython firmware*. It is provided by MicroPython on many ports to do essential initialisation.

For example, on most ports `_boot.py` will detect the first boot of a new device and format the *internal flash filesystem* ready for use.

Unless you're creating a custom MicroPython build or adding a new port then you probably don't need to worry about `_boot.py`. It's best not to change the contents unless you really know what you're doing.

boot.py

A file named `boot.py` can be copied to the board's internal *filesystem* using *mpremote*.

If `boot.py` is found then it is executed. You can add code in `boot.py` to perform custom one-off initialisation (for example, to configure the board's hardware).

A common practice is to configure a board's network connection in `boot.py` so that it's always available after reset for use with the *REPL*, *mpremote*, etc.

> **Warning:** boot.py should always exit and not run indefinitely.
>
> Depending on the port, some hardware initialisation is delayed until after `boot.py` exits. This includes initialising USB on the stm32 port and all ports which support *machine.USBDevice*. On these ports, output printed from `boot.py` may not be visible on the built-in USB serial port until after `boot.py` finishes running.
>
> The purpose of this late initialisation is so that it's possible to pre-configure particular hardware in `boot.py`, and then have it start with the correct configuration.

Note: It is sometimes simpler to not have a `boot.py` file and place any initialisation code at the top of `main.py` instead.

main.py

Similar to `boot.py`, a file named `main.py` can be copied to the board's internal *filesystem*. If found then it is executed next in the startup process.

`main.py` is for any Python code that you want to run each time your device starts.

Some tips for `main.py` usage:

- `main.py` doesn't have to exit, feel free to put an infinite `while True` loop in there.
- For complex Python applications then you don't need to put all your code in `main.py`. `main.py` can be a simple entry point that imports your application and starts execution:

```
import my_app
my_app.main()
```

This can help keep the structure of your application clear. It also makes it easy to install multiple applications on a board and switch among them.

- It's good practice when writing robust apps to wrap code in `main.py` with an exception handler to take appropriate action if the code crashes. For example:

```
import machine, sys
import my_app
try:
    my_app.main()
except Exception as e:
    print("Fatal error in main:")
    sys.print_exception(e)

# Following a normal Exception or main() exiting, reset the board.
# Following a non-Exception error such as KeyboardInterrupt (Ctrl-C),
# this code will drop to a REPL. Place machine.reset() in a finally
# block to always reset, instead.
machine.reset()
```

Otherwise MicroPython will drop to the REPL following any crash or if main exits (see below).

- Any global variables that were set in `boot.py` will still be set in the global context of `main.py`.
- To fully optimise flash usage and memory consumption, you can copy *pre-compiled* `main.mpy` and/or `boot.mpy` files to the filesystem, or even *freeze* them into the firmware build instead.
- `main.py` execution is skipped when a soft reset is initiated from *raw REPL mode* (for example, when *mpremote* or another program is interacting directly with MicroPython).

Interactive Interpreter (REPL)

If `main.py` is not found, or if `main.py` exits, then *The MicroPython Interactive Interpreter Mode (aka REPL)* will start immediately.

Note: Even if `main.py` contains an infinite loop, typing Ctrl-C on the REPL serial port will inject a *`KeyboardInterrupt`*. If no exception handler catches it then `main.py` will exit and the REPL will start.

Any global variables that were set in `boot.py` and `main.py` will still be set in the global context of the REPL.

The REPL continues executing until Python code triggers a hard or soft reset.

2.3.4 Soft Bricking (failure to boot)

It is rare but possible for MicroPython to become unresponsive during startup, a state sometimes called "soft bricked". For example:

- If `boot.py` execution gets stuck and the native USB serial port never initialises.
- If Python code reconfigures the REPL interface, making it inaccessible.

Rest assured, recovery is possible!

KeyboardInterrupt

In many cases, opening the REPL serial port and typing `Ctrl-C` will inject *`KeyboardInterrupt`* and may cause the running script to exit and a REPL to start. From the REPL, you can use *`os.remove()`* to remove the misbehaving Python file:

```
import os
os.remove('main.py')
```

To confirm which files are still present in the internal filesystem:

```
import os
os.listdir()
```

Safe Mode and Factory Reset

If you're unable to easily access the REPL then you may need to perform one of two processes:

1. "Safe mode" boot, which skips `boot.py` and `main.py` and immediately starts a REPL, allowing you to clean up. This is only supported on some ports.
2. Factory Reset to erase the entire contents of the flash filesystem. This may also be necessary if the internal flash filesystem has become corrupted somehow.

The specific process(es) are different on each port:

- *pyboard and stm32 port instructions*
- *esp32 port instructions*
- *renesas-ra port instructions*
- *rp2 port instructions*
- *wipy port instructions*

For ports without specific instructions linked above, the factory reset process involves erasing the board's entire flash and then flashing MicroPython again from scratch. Usually this will involve the same tool(s) that were originally used to install MicroPython. Consult the installation docs for your board, or ask on the GitHub Discussions if you're not sure.

Warning: Re-flashing the MicroPython firmware without erasing the entire flash first will usually not recover from soft bricking, as a firmware update usually preserves the contents of the filesystem.

2.4 MicroPython remote control: mpremote

The `mpremote` command line tool provides an integrated set of utilities to remotely interact with, manage the filesystem on, and automate a MicroPython device over a serial connection.

To use mpremote, first install it via `pip`:

```
$ pip install --user mpremote
```

Or via pipx:

```
$ pipx install mpremote
```

The simplest way to use this tool is just by invoking it without any arguments:

```
$ mpremote
```

This command automatically detects and connects to the first available USB serial device and provides an interactive terminal that you can use to access the REPL and your program's output. Serial ports are opened in exclusive mode, so running a second (or third, etc) instance of `mpremote` will connect to subsequent serial devices, if any are available.

Additionally `pipx` also allows you to directly run `mpremote` without installing first:

```
$ pipx run mpremote ...args
```

2.4.1 Commands

`mpremote` supports being given a series of commands given at the command line which will perform various actions in sequence on a remote MicroPython device. See the *examples section* below to get an idea of how this works and for some common combinations of commands.

Each command is of the form `<command name> [--options] [args...]`. For commands that support multiple arguments (e.g. a list of files), the argument list can be terminated with `+`.

If no command is specified, the default command is `repl`. Additionally, if any command needs to access the device, and no earlier `connect` has been specified, then an implicit `connect auto` is added.

Once connected, `mpremote` will stop any running program before running an action command, but it will not clear the interpreter state. Variables and imports therefore persist from one command to the next, and from one invocation of `mpremote` to the next. Use the `soft-reset` command where a clean state is required. See *auto-connection and soft-reset* for more details.

Multiple commands can be specified and they will be run sequentially.

The full list of supported commands are:

- *connect*
- *disconnect*
- *soft_reset*
- *repl*
- *eval*
- *exec*
- *run*

- *fs*
- *df*
- *edit*
- *mip*
- *mount*
- *unmount*
- *romfs*
- *rtc*
- *sleep*
- *reset*
- *bootloader*

- **connect** – connect to specified device via name:

```
$ mpremote connect <device>
```

  `<device>` may be one of:

  - `list`: list available devices
  - `auto`: connect to the first available USB serial port
  - `id:<serial>`: connect to the device with USB serial number `<serial>` (the second column from the `connect list` command output)
  - `port:<path>`: connect to the device with the given path (the first column from the `connect list` command output
  - `rfc2217://<host>:<port>`: connect to the device using serial over TCP (e.g. a networked serial port based on RFC2217)
  - any valid device name/path, to connect to that device

  Note: Instead of using the `connect` command, there are several *pre-defined shortcuts* for common device paths. For example the `a0` shortcut command is equivalent to `connect /dev/ttyACM0` (Linux), or `c1` for `COM1` (Windows).

  Note: The `auto` option will only detect USB serial ports, i.e. a serial port that has an associated USB VID/PID (i.e. CDC/ACM or FTDI-style devices). Other types of serial ports will not be auto-detected.

- **disconnect** – disconnect current device:

```
$ mpremote disconnect
```

  A subsequent command will reconnect, keeping the interpreter state that was on the device.

- **soft-reset** – perform a soft-reset of the device:

```
$ mpremote soft-reset
```

  This will clear out the Python heap and restart the interpreter.

- **repl** – enter the REPL on the connected device:

```
$ mpremote repl [--options]
```

Options are:

- `--escape-non-printable`, to print non-printable bytes/characters as their hex code
- `--capture <file>`, to capture output of the REPL session to the given file
- `--inject-code <string>`, to specify characters to inject at the REPL when `Ctrl-J` is pressed. This allows you to automate a common command.
- `--inject-file <file>`, to specify a file to inject at the REPL when `Ctrl-K` is pressed. This allows you to run a file (e.g. containing some useful setup code, or even the program you are currently working on).

While the `repl` command running, you can use `Ctrl-]` or `Ctrl-x` to exit.

Note: The name "REPL" here reflects that the common usage of this command to access the Read Eval Print Loop that is running on the MicroPython device. Strictly, the `repl` command is just functioning as a terminal (or "serial monitor") to access the device. Because this command does not stop a running program, this means that if a program is currently running, you will first need to interrupt it with `Ctrl-C` to get to the REPL, which will then allow you to access program state. You can also use `mpremote soft-reset repl` to get a "clean" REPL with all program state cleared.

- **eval –** evaluate and print the result of a Python expression:

```
$ mpremote eval <string>
```

- **exec –** execute the given Python code:

```
$ mpremote exec <string>
```

By default, `mpremote exec` will display any output from the expression until it terminates. The `--no-follow` flag can be specified to return immediately and leave the device running the expression in the background.

- **run –** run a script from the local filesystem:

```
$ mpremote run <file.py>
```

This will execute the file directly from RAM on the device without copying it to the filesystem. This is a very useful way to iterate on the development of a single piece of code without having to worry about deploying it to the filesystem.

By default, `mpremote run` will display any output from the script until it terminates. The `--no-follow` flag can be specified to return immediately and leave the device running the script in the background.

Note: Only the contents of the local file are sent to the device; the local filename has no special meaning, so passing a file called `main.py` is no different from any other name. The script is executed in raw REPL after a soft reset, so the device's own `main.py` is not run beforehand. Any `main.py` already stored on the device filesystem is left untouched.

- **fs –** execute filesystem commands on the device:

```
$ mpremote fs <sub-command>
```

`<sub-command>` may be:

- `cat <file..>` to show the contents of a file or files on the device
- `ls` to list the current directory
- `ls <dirs...>` to list the given directories
- `cp [-rf] <src...> <dest>` to copy files
- `rm [-r] <src...>` to remove files or folders on the device

- `mkdir <dirs...>` to create directories on the device
- `rmdir <dirs...>` to remove directories on the device
- `touch <file..>` to create the files (if they don't already exist)
- `sha256sum <file..>` to calculate the SHA256 sum of files
- `tree [-vsh] <dirs...>` to print a tree of the given directories

The `cp` command uses a convention where a leading `:` represents a remote path. Without a leading `:` means a local path. This is based on the convention used by the Secure Copy Protocol (scp) client.

So for example, `mpremote fs cp main.py :main.py` copies `main.py` from the current local directory to the remote filesystem, whereas `mpremote fs cp :main.py main.py` copies `main.py` from the device back to the current directory.

The `mpremote rm -r` command accepts both relative and absolute paths. Use `:` to refer to the current remote working directory (cwd) to allow a directory tree to be removed from the device's default path (eg `/flash`, `/`). Use `-v/--verbose` to see the files being removed.

For example:

- `mpremote rm -r :libs` will remove the `libs` directory and all its child items from the device.
- `mpremote rm -rv :/sd` will remove all files from a mounted SDCard and result in a non-blocking warning. The mount will be retained.
- `mpremote rm -rv :/` will remove all files on the device, including any located in mounted vfs such as `/sd` or `/flash`. After removing all folders and files, this will also return an error to mimic unix `rm -rf /` behaviour.

Warning: There is no supported way to undelete files removed by `mpremote rm -r :`. Please use with caution.

The `tree` command will print a tree of the given directories. Using the `--size/-s` option will print the size of each file, or use `--human/-h` to use a more human readable format. Note: Directory size is only printed when a non-zero size is reported by the device's filesystem. The `-v` option can be used to include the name of the serial device in the output.

All other commands implicitly assume the path is a remote path, but the `:` can be optionally used for clarity.

All of the filesystem sub-commands take multiple path arguments, so if there is another command in the sequence, you must use + to terminate the arguments, e.g.

```
$ mpremote fs cp main.py :main.py + repl
```

This will copy the file to the device then enter the REPL. The + prevents `"repl"` being interpreted as a path.

The `cp` command supports the `-r` option to make a recursive copy. By default `cp` will skip copying files to the remote device if the SHA256 hash of the source and destination file matches. To force a copy regardless of the hash use the `-f` option.

Note: For convenience, all of the filesystem sub-commands are also *aliased as regular commands*, i.e. you can write `mpremote cp ...` instead of `mpremote fs cp ...`.

- **df** – query device free/used space

```
$ mpremote df
```

The `df` command will print size/used/free statistics for the device filesystem, similar to the Unix `df` command.

- **edit** – edit a file on the device:

```
$ mpremote edit <files...>
```

  The `edit` command will copy each file from the device to a local temporary directory and then launch your editor for each file (defined by the environment variable `$EDITOR`). If the editor exits successfully, the updated file will be copied back to the device.

- **mip** – install packages from *micropython-lib* (or GitHub) using the `mip` tool:

```
$ mpremote mip install <packages...>
```

  See *Package management* for more information.

- **mount** – mount the local directory on the remote device:

```
$ mpremote mount [options] <local-dir>
```

  This allows the remote device to see the local host directory as if it were its own filesystem. This is useful for development, and avoids the need to copy files to the device while you are working on them.

  The device installs a filesystem driver, which is then mounted in the *device VFS* as `/remote`, which uses the serial connection to `mpremote` as a side-channel to access files. The device will have its current working directory (via `os.chdir`) set to `/remote` so that imports and file access will occur there instead of the default filesystem path while the mount is active.

  Note: If the `mount` command is not followed by another action in the sequence, a `repl` command will be implicitly added to the end of the sequence.

  During usage, Ctrl-D will trigger a soft-reset as normal, but the mount will automatically be re-connected. If the unit has a main.py running at startup however the remount cannot occur. In this case a raw mode soft reboot can be used: Ctrl-A Ctrl-D to reboot, then Ctrl-B to get back to normal repl at which point the mount will be ready.

  Options are:

  - `-l`, `--unsafe-links`: By default an error will be raised if the device accesses a file or directory which is outside (up one or more directory levels) the local directory that is mounted. This option disables this check for symbolic links, allowing the device to follow symbolic links outside of the local directory.

- **unmount** – unmount the local directory from the remote device:

```
$ mpremote umount
```

  This happens automatically when `mpremote` terminates, but it can be used in a sequence to unmount an earlier mount before subsequent command are run.

- **romfs** – manage ROMFS partitions on the device:

```
$ mpremote romfs <sub-command>
```

  See *mpremote ROMFS commands* for details.

- **rtc** – set/get the device clock (RTC):

```
$ mpremote rtc
```

  This will query the device RTC for the current time and print it as a datetime tuple.

```
$ mpremote rtc --set
```

  This will set the device RTC to the host PC's current time.

- **sleep –** sleep (delay) before executing the next command

```
$ mpremote sleep 0.5
```

  This will pause execution of the command sequence for the specified duration in seconds, e.g. to wait for the device to do something.

- **reset –** hard reset the device

```
$ mpremote reset
```

  Note: hard reset is equivalent to *machine.reset()*.

- **bootloader** enter the bootloader

```
$ mpremote bootloader
```

  This will make the device enter its bootloader. The bootloader is port- and board-specific (e.g. DFU on stm32, UF2 on rp2040/Pico).

2.4.2 ROMFS commands

The `romfs` command provides three sub-commands for managing ROMFS images on a connected device.

mpremote romfs query

```
$ mpremote romfs query
```

Lists all available ROMFS partitions on the device and their sizes. Also shows the first 12 bytes of each partition in hex and reports whether a valid ROMFS image is present.

Example output:

```
ROMFS0 partition has size 131072 bytes (32 blocks of 4096 bytes each)
  Raw contents: d2:cd:31:XX:XX:XX:XX:XX:XX:XX:XX:XX ...
  ROMFS image size: 1234
```

mpremote romfs build

```
$ mpremote romfs [-o <output>] build <source>
```

Build a ROMFS image from the directory *source* on the host PC. The image is written to *output* (default: `<source>.romfs`).

Options:

- `-o <output>`, `--output <output>`: Specify the output file path.
- `-m`, `--mpy` (default): Automatically compile `.py` files to `.mpy` using `mpy_cross` before adding them to the image. This requires the `mpy_cross` Python package (`pip install mpy_cross`); without it, `mpremote` prints a warning and leaves the `.py` files unchanged.
- `--no-mpy`: Disable automatic compilation of `.py` files.

Example:

```
$ mpremote romfs build myapp/
Building romfs filesystem, source directory: myapp/
/
|-- main.py -> .mpy
\-- lib/
    \-- helper.py -> .mpy
Writing 2048 bytes to output file myapp.romfs
```

mpremote romfs deploy

```
$ mpremote romfs [-p <partition>] deploy <source>
```

Deploy a ROMFS image to the device. *source* can be either:

- A directory on the host: the ROMFS image is built in memory and deployed directly.
- A `.romfs` or `.img` file: the image is read from disk and deployed.

Options:

- `-p <partition>`, `--partition <partition>`: Specify the target partition index (default: `0`).
- `-m`, `--mpy` (default): Compile `.py` to `.mpy` when *source* is a directory. If `mpy_cross` is not installed, `mpremote` prints a warning and leaves the `.py` files unchanged.
- `--no-mpy`: Disable automatic compilation of `.py` files.

After deployment, the device must be soft-reset for the new ROMFS to be mounted at `/rom`.

Example:

```
$ mpremote romfs deploy myapp/
Building romfs filesystem, source directory: myapp/
/
|-- main.py -> .mpy
\-- lib/
    \-- helper.py -> .mpy
Image size is 2048 bytes
ROMFS0 partition has size 131072 bytes (32 blocks of 4096 bytes each)
Preparing ROMFS0 partition for writing
Deploying ROMFS to ROMFS0 partition
ROMFS image deployed

$ mpremote soft-reset
```

2.4.3 Auto connection and soft-reset

Connection and disconnection will be done automatically at the start and end of the execution of the tool, if such commands are not explicitly given. Automatic connection will search for the first available USB serial device.

`mpremote` does not soft-reset the device on its own. An action command such as `mount`, `eval`, `exec`, `run` or `fs` will interrupt a running program, but the Python heap is left alone, so variables and imported modules set up by an earlier command are still there. This also holds across a `disconnect`, and across separate invocations of `mpremote`.

Use the `soft-reset` command to clear the Python heap and restart the interpreter, either at the start of a sequence of commands or partway through it. Note that a soft-reset drops the connection on some devices, such as those using a dynamic `USBDevice` or WebREPL, which is why it is left to the user to ask for.

The old behaviour of soft-resetting on connection can be turned back on by setting `auto_soft_reset` in the *user configuration file*:

```
auto_soft_reset = True
```

With that set, `mpremote` soft-resets the device the first time a command needs the device, and again after each `disconnect`. The `resume` command skips that soft-reset for a single invocation, and is otherwise accepted but does nothing.

2.4.4 Shortcuts

Shortcuts can be defined using the macro system. Built-in shortcuts are:

- `devs`: Alias for `connect list`
- `a0`, `a1`, `a2`, `a3`: Aliases for `connect /dev/ttyACMn`
- `u0`, `u1`, `u2`, `u3`: Aliases for `connect /dev/ttyUSBn`
- `c0`, `c1`, `c2`, `c3`: Aliases for `connect COMn`
- `cat`, `edit`, `ls`, `cp`, `rm`, `mkdir`, `rmdir`, `touch`: Aliases for `fs <sub-command>`

Additional shortcuts can be defined in the user configuration file `mpremote/config.py`, located in the User Configuration Directory. The correct location for each OS is determined using the `platformdirs` module.

This is typically: - `$XDG_CONFIG_HOME/mpremote/config.py` - `$HOME/.config/mpremote/config.py` - `$env:LOCALAPPDATA/mpremote/config.py`

The `config.py` file may set `auto_soft_reset` to control whether `mpremote` soft-resets the device on connection, see *auto connection and soft-reset*:

```
auto_soft_reset = True
```

The `config.py`` file should define a dictionary named `commands`. The keys of this dictionary are the shortcuts and the values are either a string or a list-of-strings:

```
"c33": "connect id:334D335C3138",
```

The command `c33` is replaced by `connect id:334D335C3138`.

```
"test": ["mount", ".", "exec", "import test"],
```

The command `test` is replaced by `mount . exec "import test"`.

Shortcuts can also accept arguments. For example:

```
"multiply x=4 y=7": "eval x*y",
```

Running `mpremote times 3 7` will set `x` and `y` as variables on the device, then evaluate the expression `x*y`.

An example `config.py` might look like:

```
commands = {
    "c33": "connect id:334D335C3138", # Connect to a specific device by ID.
    "bl": "bootloader", # Shorter alias for bootloader.
    "double x=4": "eval x*2",  # x is an argument, with default 4
    "wl_scan": ["exec", """
import network
```

(continues on next page)

(continued from previous page)

```
wl = network.WLAN()
wl.active(1)
for ap in wl.scan():
    print(ap)
""",], # Print out nearby WiFi networks.
    "wl_ipconfig": [
"exec",
"import network; sta_if = network.WLAN(network.WLAN.IF_STA); print(sta_if.ipconfig(
↪'addr4'))",
""",], # Print ip address of station interface.
    "test": ["mount", ".", "exec", "import test"], # Mount current directory and run␣
↪test.py.
    "demo": ["run", "path/to/demo.py"], # Execute demo.py on the device.
}
```

2.4.5 Examples

```
mpremote
```

Connect to the first available device and implicitly run the `repl` command.

```
mpremote a1
```

Connect to the device at `/dev/ttyACM1` (Linux) and implicitly run the `repl` command. See *shortcuts* above.

```
mpremote c1
```

Connect to the device at `COM1` (Windows) and implicitly run the `repl` command. See *shortcuts* above.

```
mpremote connect /dev/ttyUSB0
```

Explicitly specify which device to connect to, and as above, implicitly run the `repl` command.

```
mpremote a1 ls
```

Connect to the device at `/dev/ttyACM0` and then run the `ls` command.

It is equivalent to `mpremote connect /dev/ttyACM1 fs ls`.

```
mpremote exec "import micropython; micropython.mem_info()"
```

Run the specified Python command and display any output. This is equivalent to typing the command at the REPL prompt.

```
mpremote eval 1/2 eval 3/4
```

Evaluate each expression in turn and print the results.

```
mpremote a0 eval 1/2 a1 eval 3/4
```

Evaluate `1/2` on the device at `/dev/ttyACM0`, then `3/4` on the device at `/dev/ttyACM1`, printing each result.

```
mpremote exec "print_state_info()" soft-reset
```

Execute the `print_state_info()` function against the state already on the device (e.g. to find out information about the current program state), then trigger a *soft reset*.

```
mpremote reset sleep 0.5 bootloader
```

Hard-reset the device, wait 500ms for it to become available, then enter the bootloader.

```
mpremote cp utils/driver.py :utils/driver.py + run test.py
```

Update the copy of utils/driver.py on the device, then execute the local `test.py` script on the device. `test.py` is never copied to the device filesystem, rather it is run from RAM.

```
mpremote cp utils/driver.py :utils/driver.py + exec "import app"
```

Update the copy of utils/driver.py on the device, then execute app.py on the device.

This is a common development workflow to update a single file and then re-start your program. In this scenario, your `main.py` on the device would also do `import app`.

```
mpremote cp utils/driver.py :utils/driver.py + soft-reset repl
```

Update the copy of utils/driver.py on the device, then trigger a soft-reset to restart your program, and then monitor the output via the `repl` command.

```
mpremote cp -r utils/ :utils/ + soft-reset repl
```

Same as above, but update the entire utils directory first.

```
mpremote mount .
```

Mount the current local directory at `/remote` on the device and starts a `repl` session which will use `/remote` as the working directory.

```
mpremote mount . exec "import demo"
```

After mounting the current local directory, executes `demo.py` from the mounted directory.

```
mpremote mount app run test.py
```

After mounting the local directory `app` as `/remote` on the device, executes the local `test.py` from the host's current directory without copying it to the filesystem.

```
mpremote mount . repl --inject-code "import demo"
```

After mounting the current local directory, executes `demo.py` from the mounted directory each time `Ctrl-J` is pressed.

You will first need to press `Ctrl-D` to reset the interpreter state (which will preserve the mount) before pressing `Ctrl-J` to re-import `demo.py`.

```
mpremote mount app repl --inject-file demo.py
```

Same as above, but executes the contents of the local file demo.py at the REPL every time `Ctrl-K` is pressed. As above, use Ctrl-D to reset the interpreter state first.

```
mpremote cat boot.py
```

Displays the contents of `boot.py` on the device.

```
mpremote edit utils/driver.py
```

Edit `utils/driver.py` on the device using your local `$EDITOR`.

```
mpremote cp :main.py .
```

Copy `main.py` from the device to the local directory.

```
mpremote cp main.py :
```

Copy `main.py` from the local directory to the device.

```
mpremote cp :a.py :b.py
```

Copy `a.py` on the device to `b.py` on the device.

```
mpremote cp -r dir/ :
```

Recursively copy the local directory `dir` to the remote device.

```
mpremote cp a.py b.py : + repl
```

Copy `a.py` and `b.py` from the local directory to the device, then run the `repl` command.

```
mpremote mip install aioble
```

Install the `aioble` package from *micropython-lib* to the device. See *Package management*.

```
mpremote mip install github:org/repo@branch
```

Install the package from the specified branch at org/repo on GitHub to the device. See *Package management*.

```
mpremote mip install gitlab:org/repo@branch
```

Install the package from the specified branch at org/repo on GitLab to the device. See *Package management*.

```
mpremote mip install --target /flash/third-party functools
```

Install the `functools` package from *micropython-lib* to the `/flash/third-party` directory on the device. See *Package management*.

2.5 MicroPython .mpy files

MicroPython defines the concept of an .mpy file which is a binary container file format that holds precompiled code, and which can be imported like a normal .py module. The file `foo.mpy` can be imported via `import foo`, as long as `foo.mpy` can be found in the usual way by the import machinery. Usually, each directory listed in `sys.path` is searched in order. When searching a particular directory `foo.py` is looked for first and if that is not found then `foo.mpy` is looked for, then the search continues in the next directory if neither is found. As such, `foo.py` will take precedence over `foo.mpy`.

These .mpy files can contain bytecode which is usually generated from Python source files (.py files) via the `mpy-cross` program. For some architectures an .mpy file can also contain native machine code, which can be generated in a variety of ways, most notably from C source code.

2.5.1 Versioning and compatibility of .mpy files

A given .mpy file may or may not be compatible with a given MicroPython system. Compatibility is based on the following:

- Version of the .mpy file: the version of the file must match the version supported by the system loading it.
- Sub-version of the .mpy file: if the .mpy file contains native machine code then the sub-version of the file must match the version support by the system loading it. Otherwise, if there is no native machine code in the .mpy file, then the sub-version is ignored when loading.
- Small integer bits: the .mpy file will require a minimum number of bits in a small integer and the system loading it must support at least this many bits.
- Native architecture: if the .mpy file contains native machine code then it will specify the architecture of that machine code and the system loading it must support execution of that architecture's code.

If a MicroPython system supports importing .mpy files then the `sys.implementation._mpy` field will exist and return an integer which encodes the version (lower 8 bits), features and native architecture.

Trying to import an .mpy file that fails one of the first four tests will raise `ValueError('incompatible .mpy file')`. Trying to import an .mpy file that fails the native architecture test (if it contains native machine code) will raise `ValueError('incompatible .mpy arch')`.

If importing an .mpy file fails then try the following:

- Determine the .mpy version and flags supported by your MicroPython system by executing:

```
import sys
sys_mpy = sys.implementation._mpy
arch = [None, 'x86', 'x64',
    'armv6', 'armv6m', 'armv7m', 'armv7em', 'armv7emsp', 'armv7emdp',
    'xtensa', 'xtensawin', 'rv32imc', 'rv64imc'][(sys_mpy >> 10) & 0x0F]
print('mpy version:', sys_mpy & 0xff)
print('mpy sub-version:', sys_mpy >> 8 & 3)
print('mpy flags:', end='')
if arch:
    print(' -march=' + arch, end='')
if (sys_mpy >> 16) != 0:
    print(' -march-flags=' + (sys_mpy >> 16), end='')
print()
```

- Check the validity of the .mpy file by inspecting the first two bytes of the file. The first byte should be an uppercase 'M' and the second byte will be the version number, which should match the system version from above. If it doesn't match then rebuild the .mpy file.
- Check if the system .mpy version matches the version emitted by `mpy-cross` that was used to build the .mpy file, found by `mpy-cross --version`. If it doesn't match then recompile `mpy-cross` from the Git repository checked out at the tag (or hash) reported by `mpy-cross --version`.
- Make sure you are using the correct `mpy-cross` flags, found by the code above, or by inspecting the `MPY_CROSS_FLAGS` Makefile variable for the port that you are using.
- If the third byte of the .mpy file has bit #6 set, then check whether the encoded architecture-specific flag bits vuint is compatible with the target you're importing the file on.

The following table shows the correspondence between MicroPython release and .mpy version.

MicroPython release	.mpy version
v1.23.0 and up	6.3
v1.22.x	6.2
v1.20 - v1.21.0	6.1
v1.19.x	6
v1.12 - v1.18	5
v1.11	4
v1.9.3 - v1.10	3
v1.9 - v1.9.2	2
v1.5.1 - v1.8.7	0

For completeness, the next table shows the Git commit of the main MicroPython repository at which the .mpy version was changed.

.mpy version change	Git commit
6.2 to 6.3	bdbc869f9ea200c0d28b2bc7bfb60acd9d884e1b
6.1 to 6.2	6967ff3c581a66f73e9f3d78975f47528db39980
6 to 6.1	d94141e1473aebae0d3c63aeaa8397651ad6fa01
5 to 6	f2040bfc7ee033e48acef9f289790f3b4e6b74e5
4 to 5	5716c5cf65e9b2cb46c2906f40302401bdd27517
3 to 4	9a5f92ea72754c01cc03e5efcdfe94021120531e
2 to 3	ff93fd4f50321c6190e1659b19e64fef3045a484
1 to 2	dd11af209d226b7d18d5148b239662e30ed60bad
0 to 1	6a11048af1d01c78bdacddadd1b72dc7ba7c6478
initial version 0	d8c834c95d506db979ec871417de90b7951edc30

2.5.2 Binary encoding of .mpy files

MicroPython .mpy files are a binary container format with code objects (bytecode and native machine code) stored internally in a nested hierarchy. The code for the outer module is stored first, and then its children follow. Each child may have further children, for example in the case of a class having methods, or a function defining a lambda or comprehension. To keep files small while still providing a large range of possible values it uses the concept of a variably-encoded-unsigned-integer (vuint) in many places. Similar to utf-8 encoding, this encoding stores 7 bits per byte with the 8th bit (MSB) set if one or more bytes follow. The bits of the unsigned integer are stored in the vuint in LSB form.

The top-level of an .mpy file consists of three parts:

- The header.
- The global qstr and constant tables.
- The raw-code for the outer scope of the module. This outer scope is executed when the .mpy file is imported.

You can inspect the contents of a .mpy file by using `mpy-tool.py`, for example (run from the root of the main MicroPython repository):

```
$ ./tools/mpy-tool.py -xd myfile.mpy
```

The header

The .mpy header is:

size	field
byte	value 0x4d (ASCII 'M')
byte	.mpy major version number
byte	feature flags, native arch, minor version number (was feature flags in older versions)
byte	number of bits in a small int

The third byte is split as follows (MSB first):

bit	meaning
7	reserved, must be 0
6	an architecture-specific flags vuint follows the header
5..2	native arch number
1..0	minor version number

Architecture-specific flags

If bit #6 of the header's feature flags byte is set, then a vuint containing optional architecture-specific information will follow the header. The contents of this integer depends on which native architecture the file is meant for.

This is currently used to store which RISC-V processor extensions the MPY file needs to operate correctly besides I, M, C, and Zicsr. Different flavours of ArmV7 are identified by their native architecture number, but reusing that mechanism would complicate things for RV32 and RV64.

MPY files targeting RV32 or RV64 that do not need any particular processor extensions do not need to provide a flags integer (along with setting the appropriate bit in the header). The lack of a flags value for RV32 and RV64 MPY files is used to indicate that no specific extensions are needed, and saves one byte in the final output binary.

See also the `-march-flags` command-line option in both `mpy-tool.py` and `mpy-cross`, and the `--arch-flags` command-line option in `mpy_ld.py` to set this value when creating MPY files.

The global qstr and constant tables

An .mpy file contains a single qstr table, and a single constant object table. These are global to the .mpy file, they are referenced by all nested raw-code objects. The qstr table maps internal qstr number (internal to the .mpy file) to the resolved qstr number of the runtime that the .mpy file is imported into. This links the .mpy file with the rest of the system that it executes within. The constant object table is populated with references to all constant objects that the .mpy file needs.

size	field
vuint	number of qstrs
vuint	number of constant objects
…	qstr data
…	encoded constant objects

Raw code elements

A raw-code element contains code, either bytecode or native machine code. Its contents are:

size	field
vuint	type, size and whether there are sub-raw-code elements
…	code (bytecode or machine code)
vuint	number of sub-raw-code elements (only if non-zero)
…	sub-raw-code elements

The first vuint in a raw-code element encodes the type of code stored in this element (the two least-significant bits), whether this raw-code has any children (the third least-significant bit), and the length of the code that follows (the amount of RAM to allocate for it).

Following the vuint comes the code itself. Unless the code type is viper code with relocations, this code is constant data and does not need to be modified.

If this raw-code has any children (as indicated by a bit in the first vuint), following the code comes a vuint counting the number of sub-raw-code elements.

Finally any sub-raw-code elements are stored, recursively.

2.6 Writing interrupt handlers

On suitable hardware MicroPython offers the ability to write interrupt handlers in Python. Interrupt handlers - also known as interrupt service routines (ISR's) - are defined as callback functions. These are executed in response to an event such as a timer trigger or a voltage change on a pin. Such events can occur at any point in the execution of the program code. This carries significant consequences, some specific to the MicroPython language. Others are common to all systems capable of responding to real time events. This document covers the language specific issues first, followed by a brief introduction to real time programming for those new to it.

This introduction uses vague terms like "slow" or "as fast as possible". This is deliberate, as speeds are application dependent. Acceptable durations for an ISR are dependent on the rate at which interrupts occur, the nature of the main program, and the presence of other concurrent events.

2.6.1 Tips and recommended practices

This summarises the points detailed below and lists the principal recommendations for interrupt handler code.

- Keep the code as short and simple as possible.
- Avoid memory allocation: no appending to lists or insertion into dictionaries, no floating point.
- Consider using `micropython.schedule` to work around the above constraint.
- Where an ISR returns multiple bytes use a pre-allocated `bytearray`. If multiple integers are to be shared between an ISR and the main program consider an array (`array.array`).
- Where data is shared between the main program and an ISR, consider disabling interrupts prior to accessing the data in the main program and re-enabling them immediately afterwards (see Critical Sections).
- Allocate an emergency exception buffer (see below).

2.6.2 MicroPython issues

The emergency exception buffer

If an error occurs in an ISR, MicroPython is unable to produce an error report unless a special buffer is created for the purpose. Debugging is simplified if the following code is included in any program using interrupts.

```
import micropython
micropython.alloc_emergency_exception_buf(100)
```

The emergency exception buffer can only hold one exception stack trace. This means that if a second exception is thrown during the handling of an exception while the heap is locked, that second exception's stack trace will replace the original one - even if the second exception is cleanly handled. This can lead to confusing exception messages if the buffer is later printed.

Simplicity

For a variety of reasons it is important to keep ISR code as short and simple as possible. It should do only what has to be done immediately after the event which caused it: operations which can be deferred should be delegated to the main program loop. Typically an ISR will deal with the hardware device which caused the interrupt, making it ready for the next interrupt to occur. It will communicate with the main loop by updating shared data to indicate that the interrupt has occurred, and it will return. An ISR should return control to the main loop as quickly as possible. This is not a specific MicroPython issue so is covered in more detail *below*.

Communication between an ISR and the main program

Normally an ISR needs to communicate with the main program. The simplest means of doing this is via one or more shared data objects, either declared as global or shared via a class (see below). There are various restrictions and hazards around doing this, which are covered in more detail below. Integers, `bytes` and `bytearray` objects are commonly used for this purpose along with arrays (from the array module) which can store various data types.

The use of object methods as callbacks

MicroPython supports this powerful technique which enables an ISR to share instance variables with the underlying code. It also enables a class implementing a device driver to support multiple device instances. The following example causes two LED's to flash at different rates.

```
import pyb, micropython
micropython.alloc_emergency_exception_buf(100)
class Foo(object):
    def __init__(self, timer, led):
        self.led = led
        timer.callback(self.cb)
    def cb(self, tim):
        self.led.toggle()

red = Foo(pyb.Timer(4, freq=1), pyb.LED(1))
green = Foo(pyb.Timer(2, freq=0.8), pyb.LED(2))
```

In this example the `red` instance associates timer 4 with LED 1: when a timer 4 interrupt occurs `red.cb()` is called causing LED 1 to change state. The `green` instance operates similarly: a timer 2 interrupt results in the execution of `green.cb()` and toggles LED 2. The use of instance methods confers two benefits. Firstly a single class enables code to be shared between multiple hardware instances. Secondly, as a bound method the callback function's first argument is

`self`. This enables the callback to access instance data and to save state between successive calls. For example, if the class above had a variable `self.count` set to zero in the constructor, `cb()` could increment the counter. The `red` and `green` instances would then maintain independent counts of the number of times each LED had changed state.

Creation of Python objects

ISR's cannot create instances of Python objects. This is because MicroPython needs to allocate memory for the object from a store of free memory block called the heap. This is not permitted in an interrupt handler because heap allocation is not re-entrant. In other words the interrupt might occur when the main program is part way through performing an allocation - to maintain the integrity of the heap the interpreter disallows memory allocations in ISR code.

A consequence of this is that ISR's can't use floating point arithmetic; this is because floats are Python objects. Similarly an ISR can't append an item to a list. In practice it can be hard to determine exactly which code constructs will attempt to perform memory allocation and provoke an error message: another reason for keeping ISR code short and simple.

One way to avoid this issue is for the ISR to use pre-allocated buffers. For example a class constructor creates a `bytearray` instance and a boolean flag. The ISR method assigns data to locations in the buffer and sets the flag. The memory allocation occurs in the main program code when the object is instantiated rather than in the ISR.

The MicroPython library I/O methods usually provide an option to use a pre-allocated buffer. For example `pyb.i2c.recv()` can accept a mutable buffer as its first argument: this enables its use in an ISR.

A means of creating an object without employing a class or globals is as follows:

```
def set_volume(t, buf=bytearray(3)):
    buf[0] = 0xa5
    buf[1] = t >> 4
    buf[2] = 0x5a
    return buf
```

The compiler instantiates the default `buf` argument when the function is loaded for the first time (usually when the module it's in is imported).

An instance of object creation occurs when a reference to a bound method is created. This means that an ISR cannot pass a bound method to a function. One solution is to create a reference to the bound method in the class constructor and to pass that reference in the ISR. For example:

```
class Foo():
    def __init__(self):
        self.bar_ref = self.bar  # Allocation occurs here
        self.x = 0.1
        tim = pyb.Timer(4)
        tim.init(freq=2)
        tim.callback(self.cb)

    def bar(self, _):
        self.x *= 1.2
        print(self.x)

    def cb(self, t):
        # Passing self.bar would cause allocation.
        micropython.schedule(self.bar_ref, 0)
```

Other techniques are to define and instantiate the method in the constructor or to pass `Foo.bar()` with the argument *self*.

Use of Python objects

A further restriction on objects arises because of the way Python works. When an `import` statement is executed the Python code is compiled to bytecode, with one line of code typically mapping to multiple bytecodes. When the code runs the interpreter reads each bytecode and executes it as a series of machine code instructions. Given that an interrupt can occur at any time between machine code instructions, the original line of Python code may be only partially executed. Consequently a Python object such as a set, list or dictionary modified in the main loop may lack internal consistency at the moment the interrupt occurs.

A typical outcome is as follows. On rare occasions the ISR will run at the precise moment in time when the object is partially updated. When the ISR tries to read the object, a crash results. Because such problems typically occur on rare, random occasions they can be hard to diagnose. There are ways to circumvent this issue, described in *Critical Sections* below.

It is important to be clear about what constitutes the modification of an object. Altering the contents of an array or bytearray is safe. This is because bytes or words are written as a single machine code instruction which is not interruptible: in the parlance of real time programming the write is atomic. The same is true of updating a dictionary item because items are machine words, being integers or pointers to objects. A user defined object might instantiate an array or bytearray. It is valid for both the main loop and the ISR to alter the contents of these.

The hazard arises when the structure of an object is altered, notably in the case of dictionaries. Adding or deleting keys can trigger a rehash. If a hard ISR runs while a rehash is in progress and attempts to access an item, a crash may occur. Internally globals are implemented as a dictionary. Consequently the main program should create all necessary globals before starting a process that generates hard interrupts. Application code should also avoid deleting globals.

MicroPython supports integers of arbitrary precision. Values between 2**30 -1 and -2**30 will be stored in a single machine word. Larger values are stored as Python objects. Consequently changes to long integers cannot be considered atomic. The use of long integers in ISR's is unsafe because memory allocation may be attempted as the variable's value changes.

Overcoming the float limitation

In general it is best to avoid using floats in ISR code: hardware devices normally handle integers and conversion to floats is normally done in the main loop. However there are a few DSP algorithms which require floating point. On platforms with hardware floating point (such as the Pyboard) the inline ARM Thumb assembler can be used to work round this limitation. This is because the processor stores float values in a machine word; values can be shared between the ISR and main program code via an array of floats.

Using micropython.schedule

This function enables an ISR to schedule a callback for execution "very soon". The callback is queued for execution which will take place at a time when the heap is not locked. Hence it can create Python objects and use floats. The callback is also guaranteed to run at a time when the main program has completed any update of Python objects, so the callback will not encounter partially updated objects.

Typical usage is to handle sensor hardware. The ISR acquires data from the hardware and enables it to issue a further interrupt. It then schedules a callback to process the data.

Scheduled callbacks should comply with the principles of interrupt handler design outlined below. This is to avoid problems resulting from I/O activity and the modification of shared data which can arise in any code which preempts the main program loop.

Execution time needs to be considered in relation to the frequency with which interrupts can occur. If an interrupt occurs while the previous callback is executing, a further instance of the callback will be queued for execution; this will run after the current instance has completed. A sustained high interrupt repetition rate therefore carries a risk of unconstrained queue growth and eventual failure with a `RuntimeError`.

If the callback to be passed to *schedule()* is a bound method, consider the note in "Creation of Python objects".

2.6.3 Exceptions

If an ISR raises an exception it will not propagate to the main loop. The interrupt will be disabled unless the exception is handled by the ISR code.

2.6.4 Interfacing to asyncio

When an ISR runs it can preempt the *asyncio* scheduler. If the ISR performs a *asyncio* operation the scheduler's operation can be disrupted. This applies whether the interrupt is hard or soft and also applies if the ISR has passed execution to another function via *micropython.schedule*. In particular creating or cancelling tasks is invalid in an ISR context. The safe way to interact with *asyncio* is to implement a coroutine with synchronisation performed by *asyncio.ThreadSafeFlag*. The following fragment illustrates the creation of a task in response to an interrupt:

```
tsf = asyncio.ThreadSafeFlag()

def isr(_):  # Interrupt handler
    tsf.set()

async def foo():
    while True:
        await tsf.wait()
        asyncio.create_task(bar())
```

In this example there will be a variable amount of latency between the execution of the ISR and the execution of `foo()`. This is inherent to cooperative scheduling. The maximum latency is application and platform dependent but may typically be measured in tens of ms.

2.6.5 General issues

This is merely a brief introduction to the subject of real time programming. Beginners should note that design errors in real time programs can lead to faults which are particularly hard to diagnose. This is because they can occur rarely and at intervals which are essentially random. It is crucial to get the initial design right and to anticipate issues before they arise. Both interrupt handlers and the main program need to be designed with an appreciation of the following issues.

Interrupt handler design

As mentioned above, ISR's should be designed to be as simple as possible. They should always return in a short, predictable period of time. This is important because when the ISR is running, the main loop is not: inevitably the main loop experiences pauses in its execution at random points in the code. Such pauses can be a source of hard to diagnose bugs particularly if their duration is long or variable. In order to understand the implications of ISR run time, a basic grasp of interrupt priorities is required.

Interrupts are organised according to a priority scheme. ISR code may itself be interrupted by a higher priority interrupt. This has implications if the two interrupts share data (see Critical Sections below). If such an interrupt occurs it interposes a delay into the ISR code. If a lower priority interrupt occurs while the ISR is running, it will be delayed until the ISR is complete: if the delay is too long, the lower priority interrupt may fail. A further issue with slow ISR's is the case where a second interrupt of the same type occurs during its execution. The second interrupt will be handled on termination of the first. However if the rate of incoming interrupts consistently exceeds the capacity of the ISR to service them the outcome will not be a happy one.

Consequently looping constructs should be avoided or minimised. I/O to devices other than to the interrupting device should normally be avoided: I/O such as disk access, `print` statements and UART access is relatively slow, and its duration may vary. A further issue here is that filesystem functions are not reentrant: using filesystem I/O in an ISR and the main program would be hazardous. Crucially ISR code should not wait on an event. I/O is acceptable if the code can be guaranteed to return in a predictable period, for example toggling a pin or LED. Accessing the interrupting device via I2C or SPI may be necessary but the time taken for such accesses should be calculated or measured and its impact on the application assessed.

There is usually a need to share data between the ISR and the main loop. This may be done either through global variables or via class or instance variables. Variables are typically integer or boolean types, or integer or byte arrays (a pre-allocated integer array offers faster access than a list). Where multiple values are modified by the ISR it is necessary to consider the case where the interrupt occurs at a time when the main program has accessed some, but not all, of the values. This can lead to inconsistencies.

Consider the following design. An ISR stores incoming data in a bytearray, then adds the number of bytes received to an integer representing total bytes ready for processing. The main program reads the number of bytes, processes the bytes, then clears down the number of bytes ready. This will work until an interrupt occurs just after the main program has read the number of bytes. The ISR puts the added data into the buffer and updates the number received, but the main program has already read the number, so processes the data originally received. The newly arrived bytes are lost.

There are various ways of avoiding this hazard, the simplest being to use a circular buffer. If it is not possible to use a structure with inherent thread safety other ways are described below.

Reentrancy

A potential hazard may occur if a function or method is shared between the main program and one or more ISR's or between multiple ISR's. The issue here is that the function may itself be interrupted and a further instance of that function run. If this is to occur, the function must be designed to be reentrant. How this is done is an advanced topic beyond the scope of this tutorial.

Critical sections

An example of a critical section of code is one which accesses more than one variable which can be affected by an ISR. If the interrupt happens to occur between accesses to the individual variables, their values will be inconsistent. This is an instance of a hazard known as a race condition: the ISR and the main program loop race to alter the variables. To avoid inconsistency a means must be employed to ensure that the ISR does not alter the values for the duration of the critical section. One way to achieve this is to issue `pyb.disable_irq()` before the start of the section, and `pyb.enable_irq()` at the end. Here is an example of this approach:

```
import pyb, micropython, array
micropython.alloc_emergency_exception_buf(100)

class BoundsException(Exception):
    pass

ARRAYSIZE = const(20)
index = 0
data = array.array('i', 0 for x in range(ARRAYSIZE))

def callback1(t):
    global data, index
    for x in range(5):
        data[index] = pyb.rng() # simulate input
        index += 1
        if index >= ARRAYSIZE:
```

(continues on next page)

(continued from previous page)

```
            raise BoundsException('Array bounds exceeded')

tim4 = pyb.Timer(4, freq=100, callback=callback1)

for loop in range(1000):
    if index > 0:
        irq_state = pyb.disable_irq() # Start of critical section
        for x in range(index):
            print(data[x])
        index = 0
        pyb.enable_irq(irq_state) # End of critical section
        print('loop {}'.format(loop))
    pyb.delay(1)

tim4.callback(None)
```

A critical section can comprise a single line of code and a single variable. Consider the following code fragment.

```
count = 0
def cb(): # An interrupt callback
    count +=1
def main():
    # Code to set up the interrupt callback omitted
    while True:
        count += 1
```

This example illustrates a subtle source of bugs. The line `count += 1` in the main loop carries a specific race condition hazard known as a read-modify-write. This is a classic cause of bugs in real time systems. In the main loop MicroPython reads the value of `count`, adds 1 to it, and writes it back. On rare occasions the interrupt occurs after the read and before the write. The interrupt modifies `count` but its change is overwritten by the main loop when the ISR returns. In a real system this could lead to rare, unpredictable failures.

As mentioned above, care should be taken if an instance of a Python built in type is modified in the main code and that instance is accessed in an ISR. The code performing the modification should be regarded as a critical section to ensure that the instance is in a valid state when the ISR runs.

Particular care needs to be taken if a dataset is shared between different ISR's. The hazard here is that the higher priority interrupt may occur when the lower priority one has partially updated the shared data. Dealing with this situation is an advanced topic beyond the scope of this introduction other than to note that mutex objects described below can sometimes be used.

Disabling interrupts for the duration of a critical section is the usual and simplest way to proceed, but it disables all interrupts rather than merely the one with the potential to cause problems. It is generally undesirable to disable an interrupt for long. In the case of timer interrupts it introduces variability to the time when a callback occurs. In the case of device interrupts, it can lead to the device being serviced too late with possible loss of data or overrun errors in the device hardware. Like ISR's, a critical section in the main code should have a short, predictable duration.

An approach to dealing with critical sections which radically reduces the time for which interrupts are disabled is to use an object termed a mutex (name derived from the notion of mutual exclusion). The main program locks the mutex before running the critical section and unlocks it at the end. The ISR tests whether the mutex is locked. If it is, it avoids the critical section and returns. The design challenge is defining what the ISR should do in the event that access to the critical variables is denied. A simple example of a mutex may be found here. Note that the mutex code does disable interrupts, but only for the duration of eight machine instructions: the benefit of this approach is that other interrupts are virtually unaffected.

Interrupts and the REPL

Interrupt handlers, such as those associated with timers, can continue to run after a program terminates. This may produce unexpected results where you might have expected the object raising the callback to have gone out of scope. For example on the Pyboard:

```
def bar():
    foo = pyb.Timer(2, freq=4, callback=lambda t: print('.', end=''))

bar()
```

This continues to run until the timer is explicitly disabled or the board is reset with `ctrl D`.

2.7 Maximising MicroPython speed

Contents

- *Maximising MicroPython speed*
  - *Designing for speed*
    - *Algorithms*
    - *RAM allocation*
    - *Buffers*
    - *Floating point*
    - *Arrays*
    - *Memoryviews*
    - *Strings vs Bytes*
  - *Identifying the slowest section of code*
  - *MicroPython code improvements*
    - *The const() declaration*
    - *Caching object references*
    - *Controlling garbage collection*
  - *The Native code emitter*
  - *The Viper code emitter*
  - *Accessing hardware directly*

This tutorial describes ways of improving the performance of MicroPython code. Optimisations involving other languages are covered elsewhere, namely the use of modules written in C and the MicroPython inline assembler.

The process of developing high performance code comprises the following stages which should be performed in the order listed.

- Design for speed.
- Code and debug.

Optimisation steps:

- Identify the slowest section of code.
- Improve the efficiency of the Python code.
- Use the native code emitter.
- Use the viper code emitter.
- Use hardware-specific optimisations.

2.7.1 Designing for speed

Performance issues should be considered at the outset. This involves taking a view on the sections of code which are most performance critical and devoting particular attention to their design. The process of optimisation begins when the code has been tested: if the design is correct at the outset optimisation will be straightforward and may actually be unnecessary.

Algorithms

The most important aspect of designing any routine for performance is ensuring that the best algorithm is employed. This is a topic for textbooks rather than for a MicroPython guide but spectacular performance gains can sometimes be achieved by adopting algorithms known for their efficiency.

RAM allocation

To design efficient MicroPython code it is necessary to have an understanding of the way the interpreter allocates RAM. When an object is created or grows in size (for example where an item is appended to a list) the necessary RAM is allocated from a block known as the heap. This takes a significant amount of time; further it will on occasion trigger a process known as garbage collection which can take several milliseconds.

Consequently the performance of a function or method can be improved if an object is created once only and not permitted to grow in size. This implies that the object persists for the duration of its use: typically it will be instantiated in a class constructor and used in various methods.

This is covered in further detail *Controlling garbage collection* below.

Buffers

An example of the above is the common case where a buffer is required, such as one used for communication with a device. A typical driver will create the buffer in the constructor and use it in its I/O methods which will be called repeatedly.

The MicroPython libraries typically provide support for pre-allocated buffers. For example, objects which support stream interface (e.g., file or UART) provide `read()` method which allocates new buffer for read data, but also a `readinto()` method to read data into an existing buffer.

Some useful classes for creating reusable buffer objects:

- *`bytearray`*
- *`array`* (*discussed below*)
- *`io.StringIO`* and *`io.BytesIO`*
- *`micropython.RingIO`*

Floating point

Some MicroPython ports allocate floating point numbers on heap. Some other ports may lack dedicated floating-point coprocessor, and perform arithmetic operations on them in "software" at considerably lower speed than on integers. Where performance is important, use integer operations and restrict the use of floating point to sections of the code where performance is not paramount. For example, capture ADC readings as integers values to an array in one quick go, and only then convert them to floating-point numbers for signal processing.

Arrays

Consider the use of the various types of array classes as an alternative to lists. The *array* module supports various element types with 8-bit elements supported by Python's built in *bytes* and *bytearray* classes. These data structures all store elements in contiguous memory locations. Once again to avoid memory allocation in critical code these should be pre-allocated and passed as arguments or as bound objects.

Memoryviews

When passing slices of objects such as *bytearray* instances, Python creates a copy which involves allocation of the size proportional to the size of slice. This can be alleviated using a *memoryview* object. The *memoryview* itself is allocated on the heap, but is a small, fixed-size object, regardless of the size of slice it points too. Slicing a *memoryview* creates a new *memoryview*, so this cannot be done in an interrupt service routine. Further, the slice syntax `a:b` causes further allocation by instantiating a `slice(a, b)` object.

```
ba = bytearray(10000)   # big array
func(ba[30:2000])       # a copy is passed, ~2K new allocation
mv = memoryview(ba)     # small object is allocated
func(mv[30:2000])       # a pointer to memory is passed
```

A *memoryview* can only be applied to objects supporting the buffer protocol - this includes arrays but not lists. Small caveat is that while memoryview object is live, it also keeps alive the original buffer object. So, a memoryview isn't a universal panacea. For instance, in the example above, if you are done with 10K buffer and just need those bytes 30:2000 from it, it may be better to make a slice, and let the 10K buffer go (be ready for garbage collection), instead of making a long-living memoryview and keeping 10K blocked for GC.

Nonetheless, *memoryview* is indispensable for advanced preallocated buffer management. `readinto()` method discussed above puts data at the beginning of buffer and fills in entire buffer. What if you need to put data in the middle of existing buffer? Just create a memoryview into the needed section of buffer and pass it to `readinto()`.

Strings vs Bytes

MicroPython uses *string interning* to save space when there are multiple identical strings. Each time a new string is allocated at runtime (for example, when two other strings are concatenated), MicroPython checks whether the new string can be interned to save RAM.

If you have code which performs performance-critical string operations then consider using *bytes* objects and literals (i.e. `b"abc"`). This skips the interning check, and can be several times faster than performing the same operations with string objects.

Note: The fastest performance will always be achieved by avoiding new object creation entirely, for example with a reusable *buffer as described above*.

2.7.2 Identifying the slowest section of code

This is a process known as profiling and is covered in textbooks and (for standard Python) supported by various software tools. For the type of smaller embedded application likely to be running on MicroPython platforms the slowest function or method can usually be established by judicious use of the timing `ticks` group of functions documented in *time*. Code execution time can be measured in ms, us, or CPU cycles.

The following enables any function or method to be timed by adding an `@timed_function` decorator:

```
def timed_function(f, *args, **kwargs):
    myname = str(f).split(' ')[1]
    def new_func(*args, **kwargs):
        t = time.ticks_us()
        result = f(*args, **kwargs)
        delta = time.ticks_diff(time.ticks_us(), t)
        print('Function {} Time = {:6.3f}ms'.format(myname, delta/1000))
        return result
    return new_func
```

2.7.3 MicroPython code improvements

The const() declaration

MicroPython provides a `const()` declaration. This works in a similar way to `#define` in C in that when the code is compiled to bytecode the compiler substitutes the constant value for the identifier. This avoids a dictionary lookup at runtime. The argument to `const()` may be anything which, at compile time, evaluates to a constant e.g. `0x100`, `1 << 8`, `"string"`, `0.001`, `b"\x00\xff"` or `("read", "write")`.

See *micropython.const()* for complete documentation including usage requirements, limitations, and examples.

Caching object references

Where a function or method repeatedly accesses objects performance is improved by caching the object in a local variable:

```
class foo(object):
    def __init__(self):
        self.ba = bytearray(100)
    def bar(self, obj_display):
        ba_ref = self.ba
        fb = obj_display.framebuffer
        # iterative code using these two objects
```

This avoids the need repeatedly to look up `self.ba` and `obj_display.framebuffer` in the body of the method `bar()`.

Controlling garbage collection

When memory allocation is required, MicroPython attempts to locate an adequately sized block on the heap. This may fail, usually because the heap is cluttered with objects which are no longer referenced by code. If a failure occurs, the process known as garbage collection reclaims the memory used by these redundant objects and the allocation is then tried again - a process which can take several milliseconds.

There may be benefits in preempting this by periodically issuing *gc.collect()*. Firstly doing a collection before it is actually required is quicker - typically on the order of 1ms if done frequently. Secondly you can determine the point in code where this time is used rather than have a longer delay occur at random points, possibly in a speed critical section. Finally performing collections regularly can reduce fragmentation in the heap. Severe fragmentation can lead to non-recoverable allocation failures.

2.7.4 The Native code emitter

This causes the MicroPython compiler to emit native CPU opcodes rather than bytecode. It covers the bulk of the MicroPython functionality, so most functions will require no adaptation (but see below). It is invoked by means of a function decorator:

```
@micropython.native
def foo(self, arg):
    buf = self.linebuf # Cached object
    # code
```

There are certain limitations in the current implementation of the native code emitter.

- If `raise` is used an argument must be supplied.
- The background scheduler (see *micropython.schedule*) is not run during execution of native code.
- On targets with threading and the GIL, the GIL is not released during execution of native code.

To mitigate the last two points, long running native functions should call `time.sleep(0)` periodically, which will run the scheduler and bounce the GIL.

The trade-off for the improved performance (roughly twice as fast as bytecode) is an increase in compiled code size.

2.7.5 The Viper code emitter

The optimisations discussed above involve standards-compliant Python code. The Viper code emitter is not fully compliant. It supports special Viper native data types in pursuit of performance. Integer processing is non-compliant because it uses machine words: arithmetic on 32 bit hardware is performed modulo 2**32.

Like the Native emitter Viper produces machine instructions but further optimisations are performed, substantially increasing performance especially for integer arithmetic and bit manipulations. It is invoked using a decorator:

```
@micropython.viper
def foo(self, arg: int) -> int:
    # code
```

As the above fragment illustrates it is beneficial to use Python type hints to assist the Viper optimiser. Type hints provide information on the data types of arguments and of the return value; these are a standard Python language feature formally defined here PEP0484. Viper supports its own set of types namely `int`, `uint` (unsigned integer), `ptr`, `ptr8`, `ptr16` and `ptr32`. The `ptrX` types are discussed below. Currently the `uint` type serves a single purpose: as a type hint for a function return value. If such a function returns `0xffffffff` Python will interpret the result as 2**32 -1 rather than as -1.

In addition to the restrictions imposed by the native emitter the following constraints apply:

- Default argument values are not permitted.
- Floating point may be used but is not optimised.

Viper provides pointer types to assist the optimiser. These comprise

- `ptr` Pointer to an object.
- `ptr8` Points to a byte.
- `ptr16` Points to a 16 bit half-word.
- `ptr32` Points to a 32 bit machine word.

The concept of a pointer may be unfamiliar to Python programmers. It has similarities to a Python *memoryview* object in that it provides direct access to data stored in memory. Items are accessed using subscript notation, but slices are not supported: a pointer can return a single item only. Its purpose is to provide fast random access to data stored in contiguous memory locations - such as data stored in objects which support the buffer protocol, and memory-mapped peripheral registers in a microcontroller. It should be noted that programming using pointers is hazardous: bounds checking is not performed and the compiler does nothing to prevent buffer overrun errors.

Typical usage is to cache variables:

```
@micropython.viper
def foo(self, arg: int) -> int:
    buf = ptr8(self.linebuf) # self.linebuf is a bytearray or bytes object
    for x in range(20, 30):
        bar = buf[x] # Access a data item through the pointer
        # code omitted
```

In this instance the compiler "knows" that `buf` is the address of an array of bytes; it can emit code to rapidly compute the address of `buf[x]` at runtime. Where casts are used to convert objects to Viper native types these should be performed at the start of the function rather than in critical timing loops as the cast operation can take several microseconds. The rules for casting are as follows:

- Casting operators are currently: `int`, `bool`, `uint`, `ptr`, `ptr8`, `ptr16` and `ptr32`.
- The result of a cast will be a native Viper variable.
- Arguments to a cast can be a Python object or a native Viper variable.
- If argument is a native Viper variable, then cast is a no-op (i.e. costs nothing at runtime) that just changes the type (e.g. from `uint` to `ptr8`) so that you can then store/load using this pointer.
- If the argument is a Python object and the cast is `int` or `uint`, then the Python object must be of integral type and the value of that integral object is returned.
- The argument to a bool cast must be integral type (boolean or integer); when used as a return type the viper function will return True or False objects.
- If the argument is a Python object and the cast is `ptr`, `ptr`, `ptr16` or `ptr32`, then the Python object must either have the buffer protocol (in which case a pointer to the start of the buffer is returned) or it must be of integral type (in which case the value of that integral object is returned).

Writing to a pointer which points to a read-only object will lead to undefined behaviour.

The following example illustrates the use of a `ptr16` cast to toggle pin X1 `n` times:

```
BIT0 = const(1)
@micropython.viper
def toggle_n(n: int):
```

(continues on next page)

(continued from previous page)

```
    odr = ptr16(stm.GPIOA + stm.GPIO_ODR)
    for _ in range(n):
        odr[0] ^= BIT0
```

A detailed technical description of the three code emitters may be found on Kickstarter here Note 1 and here Note 2

2.7.6 Accessing hardware directly

Note: Code examples in this section are given for the Pyboard. The techniques described however may be applied to other MicroPython ports too.

This comes into the category of more advanced programming and involves some knowledge of the target MCU. Consider the example of toggling an output pin on the Pyboard. The standard approach would be to write

```
mypin.value(mypin.value() ^ 1) # mypin was instantiated as an output pin
```

This involves the overhead of two calls to the *Pin* instance's *value()* method. This overhead can be eliminated by performing a read/write to the relevant bit of the chip's GPIO port output data register (odr). To facilitate this the `stm` module provides a set of constants providing the addresses of the relevant registers. A fast toggle of pin `P4` (CPU pin `A14`) - corresponding to the green LED - can be performed as follows:

```
import machine
import stm

BIT14 = const(1 << 14)
machine.mem16[stm.GPIOA + stm.GPIO_ODR] ^= BIT14
```

2.8 MicroPython on microcontrollers

MicroPython is designed to be capable of running on microcontrollers. These have hardware limitations which may be unfamiliar to programmers more familiar with conventional computers. In particular the amount of RAM and nonvolatile "disk" (flash memory) storage is limited. This tutorial offers ways to make the most of the limited resources. Because MicroPython runs on controllers based on a variety of architectures, the methods presented are generic: in some cases it will be necessary to obtain detailed information from platform specific documentation.

2.8.1 Flash memory

On the Pyboard the simple way to address the limited capacity is to fit a micro SD card. In some cases this is impractical, either because the device does not have an SD card slot or for reasons of cost or power consumption; hence the on-chip flash must be used. The firmware including the MicroPython subsystem is stored in the onboard flash. The remaining capacity is available for use. For reasons connected with the physical architecture of the flash memory part of this capacity may be inaccessible as a filesystem. In such cases this space may be employed by incorporating user modules into a firmware build which is then flashed to the device.

There are two ways to achieve this: frozen modules and frozen bytecode. Frozen modules store the Python source with the firmware. Frozen bytecode uses the cross compiler to convert the source to bytecode which is then stored with the firmware. In either case the module may be accessed with an import statement:

```
import mymodule
```

The procedure for producing frozen modules and bytecode is platform dependent; instructions for building the firmware can be found in the README files in the relevant part of the source tree.

In general terms the steps are as follows:

- Clone the MicroPython repository.
- Acquire the (platform specific) toolchain to build the firmware.
- Build the cross compiler.
- Place the modules to be frozen in a specified directory (dependent on whether the module is to be frozen as source or as bytecode).
- Build the firmware. A specific command may be required to build frozen code of either type - see the platform documentation.
- Flash the firmware to the device.

2.8.2 RAM

When reducing RAM usage there are two phases to consider: compilation and execution. In addition to memory consumption, there is also an issue known as heap fragmentation. In general terms it is best to minimise the repeated creation and destruction of objects. The reason for this is covered in the section covering the *heap*.

Compilation phase

When a module is imported, MicroPython compiles the code to bytecode which is then executed by the MicroPython virtual machine (VM). The bytecode is stored in RAM. The compiler itself requires RAM, but this becomes available for use when the compilation has completed.

If a number of modules have already been imported the situation can arise where there is insufficient RAM to run the compiler. In this case the import statement will produce a memory exception.

If a module instantiates global objects on import it will consume RAM at the time of import, which is then unavailable for the compiler to use on subsequent imports. In general it is best to avoid code which runs on import; a better approach is to have initialisation code which is run by the application after all modules have been imported. This maximises the RAM available to the compiler.

If RAM is still insufficient to compile all modules one solution is to precompile modules. MicroPython has a cross compiler capable of compiling Python modules to bytecode (see the README in the mpy-cross directory). The resulting bytecode file has a .mpy extension; it may be copied to the filesystem and imported in the usual way. Alternatively some or all modules may be implemented as frozen bytecode: on most platforms this saves even more RAM as the bytecode is run directly from flash rather than being stored in RAM.

Execution phase

There are a number of coding techniques for reducing RAM usage.

Constants

MicroPython provides a `const` keyword which may be used as follows:

```
from micropython import const
ROWS = const(33)
_COLS = const(0x10)
a = ROWS
b = _COLS
```

In both instances where the constant is assigned to a variable the compiler will avoid coding a lookup to the name of the constant by substituting its literal value. This saves bytecode and hence RAM. However the `ROWS` value will occupy at least two machine words, one each for the key and value in the globals dictionary. The presence in the dictionary is necessary because another module might import or use it. This RAM can be saved by prepending the name with an underscore as in `_COLS`: this symbol is not visible outside the module so will not occupy RAM.

The argument to `const()` may be anything which, at compile time, evaluates to a constant e.g. `0x100`, `1 << 8` or `(True, "string", b"bytes")` (see section below for details). It can even include other const symbols that have already been defined, e.g. `1 << BIT`.

See *micropython.const()* for complete documentation including scope requirements, import syntax, and other important limitations.

Constant data structures

Where there is a substantial volume of constant data and the platform supports execution from Flash, RAM may be saved as follows. The data should be located in Python modules and frozen as bytecode. The data must be defined as *bytes* objects. The compiler 'knows' that *bytes* objects are immutable and ensures that the objects remain in flash memory rather than being copied to RAM. The *struct* module can assist in converting between *bytes* types and other Python built-in types.

When considering the implications of frozen bytecode, note that in Python strings, floats, bytes, integers, complex numbers and tuples are immutable. Accordingly these will be frozen into flash (for tuples, only if all their elements are immutable). Thus, in the line

```
mystring = "The quick brown fox"
```

the actual string "The quick brown fox" will reside in flash. At runtime a reference to the string is assigned to the *variable* `mystring`. The reference occupies a single machine word. In principle a long integer could be used to store constant data:

```
bar = 0xDEADBEEF0000DEADBEEF
```

As in the string example, at runtime a reference to the arbitrarily large integer is assigned to the variable `bar`. That reference occupies a single machine word.

Tuples of constant objects are themselves constant. Such constant tuples are optimised by the compiler so they do not need to be created at runtime each time they are used. For example:

```
foo = (1, 2, 3, 4, 5, 6, 100000, ("string", b"bytes", False, True))
```

This entire tuple will exist as a single object (potentially in flash if the code is frozen) and referenced each time it is needed.

Needless object creation

There are a number of situations where objects may unwittingly be created and destroyed. This can reduce the usability of RAM through fragmentation. The following sections discuss instances of this.

String concatenation

Consider the following code fragments which aim to produce constant strings:

```
var = "foo" + "bar"
var1 = "foo" "bar"
var2 = """\
foo\
bar"""
```

Each produces the same outcome, however the first needlessly creates two string objects at runtime, allocates more RAM for concatenation before producing the third. The others perform the concatenation at compile time which is more efficient, reducing fragmentation.

Where strings must be dynamically created before being fed to a stream such as a file it will save RAM if this is done in a piecemeal fashion. Rather than creating a large string object, create a substring and feed it to the stream before dealing with the next.

The best way to create dynamic strings is by means of the string `format()` method:

```
var = "Temperature {:5.2f} Pressure {:06d}\n".format(temp, press)
```

Buffers

When accessing devices such as instances of UART, I2C and SPI interfaces, using pre-allocated buffers avoids the creation of needless objects. Consider these two loops:

```
while True:
    var = spi.read(100)
    # process data

buf = bytearray(100)
while True:
    spi.readinto(buf)
    # process data in buf
```

The first creates a buffer on each pass whereas the second reuses a pre-allocated buffer; this is both faster and more efficient in terms of memory fragmentation.

Bytes are smaller than ints

On most platforms an integer consumes four bytes. Consider the three calls to the function `foo()`:

```
def foo(bar):
    for x in bar:
        print(x)
foo([1, 2, 0xff])
foo((1, 2, 0xff))
foo(b'\1\2\xff')
```

In the first call a *list* of integers is created in RAM each time the code is executed. The second call creates a constant *tuple* object (a *tuple* containing only constant objects) as part of the compilation phase, so it is only created once and is more efficient than the *list*. The third call efficiently creates a *bytes* object consuming the minimum amount of RAM. If the module were frozen as bytecode, both the *tuple* and *bytes* object would reside in flash.

Strings Versus Bytes

Python3 introduced Unicode support. This introduced a distinction between a string and an array of bytes. MicroPython ensures that Unicode strings take no additional space so long as all characters in the string are ASCII (i.e. have a value < 126). If values in the full 8-bit range are required *bytes* and *bytearray* objects can be used to ensure that no additional space will be required. Note that most string methods (e.g. `str.strip()`) apply also to *bytes* instances so the process of eliminating Unicode can be painless.

```
s = 'the quick brown fox'    # A string instance
b = b'the quick brown fox'   # A bytes instance
```

Where it is necessary to convert between strings and bytes the *str.encode()* and the *bytes.decode()* methods can be used. MicroPython validates the encoding parameter and only supports UTF-8 and ASCII. The *bytes.decode()* method also supports error handlers (`'ignore'` and `'replace'`) for handling invalid UTF-8, when enabled in the build configuration.

For memory-conscious applications processing untrusted data, using the `'ignore'` error handler can be more efficient than `'strict'` mode (the default), as it avoids raising exceptions while still recovering valid text:

```
# Strict mode (default) raises an error on invalid UTF-8
try:
    s = data.decode('utf-8')
except UnicodeError:
    # Handle error
    pass

# Ignore mode skips invalid bytes (more memory-efficient)
s = data.decode('utf-8', 'ignore')
```

Note that both strings and bytes are immutable. Any operation which takes as input such an object and produces another implies at least one RAM allocation to produce the result. In the second line below a new bytes object is allocated. This would also occur if `foo` were a string.

```
foo = b'   empty whitespace'
foo = foo.lstrip()
```

Runtime compiler execution

The Python functions *eval* and *exec* invoke the compiler at runtime, which requires significant amounts of RAM. Note that the `pickle` library from *micropython-lib* employs *exec*. It may be more RAM efficient to use the *json* library for object serialisation.

Storing strings in flash

Python strings are immutable hence have the potential to be stored in read only memory. The compiler can place in flash strings defined in Python code. As with frozen modules it is necessary to have a copy of the source tree on the PC and the toolchain to build the firmware. The procedure will work even if the modules have not been fully debugged, so long as they can be imported and run.

After importing the modules, execute:

```
micropython.qstr_info(1)
```

Then copy and paste all the Q(xxx) lines into a text editor. Check for and remove lines which are obviously invalid. Open the file qstrdefsport.h which will be found in ports/stm32 (or the equivalent directory for the architecture in use). Copy and paste the corrected lines at the end of the file. Save the file, rebuild and flash the firmware. The outcome can be checked by importing the modules and again issuing:

```
micropython.qstr_info(1)
```

The Q(xxx) lines should be gone.

2.8.3 The heap

When a running program instantiates an object the necessary RAM is allocated from a fixed size pool known as the heap. When the object goes out of scope (in other words becomes inaccessible to code) the redundant object is known as "garbage". A process known as "garbage collection" (GC) reclaims that memory, returning it to the free heap. This process runs automatically, however it can be invoked directly by issuing *gc.collect()*.

The discourse on this is somewhat involved. For a 'quick fix' issue the following periodically:

```
gc.collect()
gc.threshold(gc.mem_free() // 4 + gc.mem_alloc())
```

For more information, see below and the documentation for built-in module *gc*.

For details from MicroPython internals/developer perspective, see also *Memory Management*.

Fragmentation

Say a program creates an object `foo`, then an object `bar`. Subsequently `foo` goes out of scope but `bar` remains. The RAM used by `foo` will be reclaimed by GC. However if `bar` was allocated to a higher address, the RAM reclaimed from `foo` will only be of use for objects no bigger than `foo`. In a complex or long running program the heap can become fragmented: despite there being a substantial amount of RAM available, there is insufficient contiguous space to allocate a particular object, and the program fails with a memory error.

The techniques outlined above aim to minimise this. Where large permanent buffers or other objects are required it is best to instantiate these early in the process of program execution before fragmentation can occur. Further improvements may be made by monitoring the state of the heap and by controlling GC; these are outlined below.

Reporting

A number of library functions are available to report on memory allocation and to control GC. These are to be found in the *gc* and *micropython* modules. The following example may be pasted at the REPL (`ctrl e` to enter paste mode, `ctrl d` to run it).

```
import gc
import micropython
gc.collect()
micropython.mem_info()
print('------------------------------')
print('Initial free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
def func():
    a = bytearray(10000)
gc.collect()
print('Func definition: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
func()
print('Func run free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
gc.collect()
print('Garbage collect free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
print('------------------------------')
micropython.mem_info(1)
```

Methods employed above:

- *gc.collect()* Force a garbage collection. See footnote.
- *micropython.mem_info()* Print a summary of RAM utilisation.

- *gc.mem_free()* Return the free heap size in bytes.
- *gc.mem_alloc()* Return the number of bytes currently allocated.
- micropython.mem_info(1) Print a table of heap utilisation (detailed below).

The numbers produced are dependent on the platform, but it can be seen that declaring the function uses a small amount of RAM in the form of bytecode emitted by the compiler (the RAM used by the compiler has been reclaimed). Running the function uses over 10KiB, but on return a is garbage because it is out of scope and cannot be referenced. The final *gc.collect()* recovers that memory.

The verbose output from micropython.mem_info(1) is documented at *micropython.mem_info()*.

Control of garbage collection

A GC can be demanded at any time by issuing *gc.collect()*. It is advantageous to do this at intervals, firstly to preempt fragmentation and secondly for performance. A GC can take several milliseconds but is quicker when there is little work to do (about 1ms on the Pyboard). An explicit call can minimise that delay while ensuring it occurs at points in the program when it is acceptable.

Automatic GC is provoked under the following circumstances. When an attempt at allocation fails, a GC is performed and the allocation re-tried. Only if this fails is an exception raised. Secondly an automatic GC will be triggered if the amount of free RAM falls below a threshold. This threshold can be adapted as execution progresses:

```
gc.collect()
gc.threshold(gc.mem_free() // 4 + gc.mem_alloc())
```

This will provoke a GC when more than 25% of the currently free heap becomes occupied.

In general modules should instantiate data objects at runtime using constructors or other initialisation functions. The reason is that if this occurs on initialisation the compiler may be starved of RAM when subsequent modules are imported. If modules do instantiate data on import then *gc.collect()* issued after the import will ameliorate the problem.

2.8.4 String operations

MicroPython handles strings in an efficient manner and understanding this can help in designing applications to run on microcontrollers. When a module is compiled, strings which occur multiple times are stored once only, a process known as string interning. In MicroPython an interned string is known as a qstr. In a module imported normally that single instance will be located in RAM, but as described above, in modules frozen as bytecode it will be located in flash.

String comparisons are also performed efficiently using hashing rather than character by character. The penalty for using strings rather than integers may hence be small both in terms of performance and RAM usage - a fact which may come as a surprise to C programmers.

2.8.5 Postscript

MicroPython passes, returns and (by default) copies objects by reference. A reference occupies a single machine word so these processes are efficient in RAM usage and speed.

Where variables are required whose size is neither a byte nor a machine word there are standard libraries which can assist in storing these efficiently and in performing conversions. See the *array*, *struct* and *uctypes* modules.

Footnote: gc.collect() return value

On Unix and Windows platforms the *gc.collect()* method returns an integer which signifies the number of distinct memory regions that were reclaimed in the collection (more precisely, the number of heads that were turned into frees). For efficiency reasons bare metal ports do not return this value.

2.9 MicroPython manifest files

2.9.1 Summary

MicroPython has a feature that allows Python code to be "frozen" into the firmware, as an alternative to loading code from the filesystem.

This has the following benefits:

- the code is pre-compiled to bytecode, avoiding the need for the Python source to be compiled at load-time.
- the bytecode can be executed directly from ROM (i.e. flash memory) rather than being copied into RAM. Similarly any constant objects (strings, tuples, etc) are loaded from ROM also. This can lead to significantly more memory being available for your application.
- on devices that do not have a filesystem, this is the only way to load Python code.

During development, freezing is generally not recommended as it will significantly slow down your development cycle, as each update will require re-flashing the entire firmware. However, it can still be useful to selectively freeze some rarely-changing dependencies (such as third-party libraries).

The way to list the Python files to be frozen into the firmware is via a "manifest", which is a Python file that will be interpreted by the build process. Typically you would write a manifest file as part of a board definition, but you can also write a stand-alone manifest file and use it with an existing board definition.

Manifest files can define dependencies on libraries from *micropython-lib* as well as Python files on the filesystem, and also on other manifest files.

2.9.2 Writing manifest files

A manifest file is a Python file containing a series of function calls. See the available functions defined below.

Any paths used in manifest files can include the following variables. These all resolve to absolute paths.

- `$(MPY_DIR)` – path to the micropython repo.
- `$(MPY_LIB_DIR)` – path to the micropython-lib submodule. Prefer to use `require()`.
- `$(PORT_DIR)` – path to the current port (e.g. `ports/stm32`)
- `$(BOARD_DIR)` – path to the current board (e.g. `ports/stm32/boards/PYBV11`)

Custom manifest files should not live in the main MicroPython repository. You should keep them in version control with the rest of your project.

Typically a manifest used for compiling firmware will need to include the port manifest, which might include frozen modules that are required for the board to function. If you just want to add additional modules to an existing board, then include the board manifest (which will in turn include the port manifest).

Building with a custom manifest

Your manifest can be specified on the `make` command line with:

```
$ make BOARD=MYBOARD FROZEN_MANIFEST=/path/to/my/project/manifest.py
```

This applies to all ports, including CMake-based ones (e.g. esp32, rp2), as the Makefile wrapper that will pass this into the CMake build.

Adding a manifest to a board definition

If you have a custom board definition, you can make it include your custom manifest automatically. On make-based ports (most ports), in your `mpconfigboard.mk` set the `FROZEN_MANIFEST` variable.

```
FROZEN_MANIFEST ?= $(BOARD_DIR)/manifest.py
```

On CMake-based ports (e.g. esp32, rp2), instead use `mpconfigboard.cmake`

```
set(MICROPY_FROZEN_MANIFEST ${MICROPY_BOARD_DIR}/manifest.py)
```

High-level functions

Note: The `opt` keyword argument can be set on the various functions, this controls the optimisation level used by the cross-compiler. See *`micropython.opt_level()`*.

`add_library`(*library*, *library_path*, *prepend=False*)

Register the path to an external named *library*.

The path *library_path* will be automatically searched when using *`require`*. By default the added library is added to the end of the list of libraries to search. Pass `True` to *prepend* to add it to the start of the list.

Additionally, the added library can be explicitly requested by using `require("name", library="library")`.

`package`(*package_path*, *files=None*, *base_path='.'*, *opt=None*)

This is equivalent to copying the "package_path" directory to the device (except as frozen code).

In the simplest case, to freeze a package "foo" in the current directory:

```
package("foo")
```

will recursively include all .py files in foo, and will be frozen as `foo/**/*.py`.

If the package isn't in the same directory as the manifest file, use `base_path`:

```
package("foo", base_path="path/to/libraries")
```

You can use the variables above, such as `$(PORT_DIR)` in `base_path`.

To restrict to certain files in the package use `files` (note: paths should be relative to the package): `package("foo", files=["bar/baz.py"])`.

`module`(*module_path*, *base_path='.'*, *opt=None*)

Include a single Python file as a module.

If the file is in the current directory:

```
module("foo.py")
```

Otherwise use base_path to locate the file:

```
module("foo.py", base_path="src/drivers")
```

You can use the variables above, such as `$(PORT_DIR)` in `base_path`.

c_module (*module_path*)

Include a C module directory in the build.

The *module_path* should be a directory containing a `micropython.mk` and/or `micropython.cmake` file that defines the C module.

This function can be called multiple times to include multiple C modules:

```
c_module("$(MPY_DIR)/examples/usercmodule/cexample")
c_module("$(BOARD_DIR)/../../drivers/sensor")
```

Supports `$(VAR)` path substitution just like other manifest functions.

`c_module()` only takes effect when the manifest is loaded as a `FROZEN_MANIFEST` (it has no effect when the manifest is processed in package-build or compile-only modes). Modules added with `c_module()` are merged with any paths passed on the command line via `USER_C_MODULES`; both sources combine additively and duplicate paths are de-duplicated.

The referenced C module directories must already be present on disk at build time. If a path points into a submodule, ensure that `make submodules` has been run first.

Note: on Makefile-based ports, paths containing whitespace are not supported (a GNU make limitation). CMake-based ports handle whitespace paths correctly.

require (*name*, *library=None*)

Require a package by name (and its dependencies) from *micropython-lib*.

Optionally specify *library* (a string) to reference a package from a library that has been previously registered with *add_library*. Otherwise the list of library paths will be used.

include (*manifest_path*)

Include another manifest.

Typically a manifest used for compiling firmware will need to include the port manifest, which might include frozen modules that are required for the board to function.

The *manifest* argument can be a string (filename) or an iterable of strings.

Relative paths are resolved with respect to the current manifest file.

If the path is to a directory, then it implicitly includes the manifest.py file inside that directory.

You can use the variables above, such as `$(PORT_DIR)` in `manifest_path`.

metadata (*description=None*, *version=None*, *license=None*, *author=None*)

Define metadata for this manifest file. This is useful for manifests for micropython-lib packages.

Low-level functions

These functions are documented for completeness, but with the exception of `freeze_as_str` all functionality can be accessed via the high-level functions.

freeze(*path*, *script=None*, *opt=0*)

Freeze the input specified by *path*, automatically determining its type. A `.py` script will be compiled to a `.mpy` first then frozen, and a `.mpy` file will be frozen directly.

path must be a directory, which is the base directory to begin searching for files. When importing the resulting frozen modules, the name of the module will start after *path*, i.e. *path* is excluded from the module name.

If *path* is relative, it is resolved to the current `manifest.py`.

If *script* is None, all files in *path* will be frozen.

If *script* is an iterable then `freeze()` is called on all items of the iterable (with the same *path* and *opt* passed through).

If *script* is a string then it specifies the file or directory to freeze, and can include extra directories before the file or last directory. The file or directory will be searched for in *path*. If *script* is a directory then all files in that directory will be frozen.

opt is the optimisation level to pass to mpy-cross when compiling `.py` to `.mpy`. These levels are described in *micropython.opt_level()*.

freeze_as_str(*path*)

Freeze the given *path* and all `.py` scripts within it as a string, which will be compiled upon import.

freeze_as_mpy(*path*, *script=None*, *opt=0*)

Freeze the input by first compiling the `.py` scripts to `.mpy` files, then freezing the resulting `.mpy` files. See `freeze()` for further details on the arguments.

freeze_mpy(*path*, *script=None*, *opt=0*)

Freeze the input, which must be `.mpy` files that are frozen directly. See `freeze()` for further details on the arguments.

2.9.3 Examples

To freeze a single file from the current directory which will be available as `import mydriver`, use:

```
module("mydriver.py")
```

To freeze a directory of files in a subdirectory "mydriver" of the current directory which will be available as `import mydriver`, use:

```
package("mydriver")
```

To freeze the "hmac" library from *micropython-lib*, use:

```
require("hmac")
```

A more complete example of a custom `manifest.py` file for the `PYBD_SF2` board is:

```
# Include the board's default manifest.
include("$(BOARD_DIR)/manifest.py")
# Add a custom driver
module("mydriver.py")
```

(continues on next page)

(continued from previous page)

```
# Add aiorepl from micropython-lib
require("aiorepl")
```

Then the board can be compiled with

```
$ cd ports/stm32
$ make BOARD=PYBD_SF2 FROZEN_MANIFEST=~/src/myproject/manifest.py
```

Note that most boards do not have their own `manifest.py`, rather they use the port one directly, in which case your manifest should just `include("$(PORT_DIR)/boards/manifest.py")` instead.

2.10 Package management

2.10.1 Installing packages with `mip`

Network-capable boards include the `mip` module, which can install packages from *micropython-lib* and from third-party sites (including GitHub, GitLab).

`mip` ("mip installs packages") is similar in concept to Python's `pip` tool, however it does not use the PyPI index, rather it uses *micropython-lib* as its index by default. `mip` will automatically fetch compiled *.mpy file* when downloading from micropython-lib.

The most common way to use `mip` is from the 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)
```

`mip` will detect an appropriate location on the filesystem by searching `sys.path` for the first entry ending in `/lib`. You can override the destination using `target`, but note that this path must be in `sys.path` to be able to subsequently import it.:

```
>>> mip.install("pkgname", target="third-party")
>>> sys.path.append("third-party")
```

As well as downloading packages from the micropython-lib index, `mip` can also install third-party libraries. The simplest way is to download a file directly:

```
>>> mip.install("http://example.com/x/y/foo.py")
>>> mip.install("http://example.com/x/y/foo.mpy")
```

When installing a file directly, the `target` argument is still supported to set the destination path, but `mpy` and `version` are ignored.

The URL can also start with `github:`, `gitlab:`, or `codeberg:` as a simple way of pointing to content hosted on GitHub, GitLab, or Codeberg:

```
>>> mip.install("github:org/repo/path/foo.py")  # Uses default branch
>>> mip.install("github:org/repo/path/foo.py", version="branch-or-tag")  # Optionally␣
↪specify the branch or tag
>>> mip.install("gitlab:org/repo/path/foo.py")  # Uses default branch
```

(continues on next page)

(continued from previous page)

```
>>> mip.install("gitlab:org/repo/path/foo.py", version="branch-or-tag")  # Optionally␣
↪specify the branch or tag
>>> mip.install("codeberg:org/repo/path/foo.py")  # Uses default branch
>>> mip.install("codeberg:org/repo/path/foo.py", version="branch-or-tag")  #␣
↪Optionally specify the branch or tag
```

More sophisticated packages (i.e. with more than one file, or with dependencies) can be downloaded by specifying the path to their `package.json`.

```
>>> mip.install("http://example.com/x/package.json")
>>> mip.install("github:org/user/path/package.json")
>>> mip.install("gitlab:org/user/path/package.json")
>>> mip.install("codeberg:org/user/path/package.json")
```

If no json file is specified, then "package.json" is implicitly added:

```
>>> mip.install("http://example.com/x/")
>>> mip.install("github:org/repo")  # Uses default branch of that repo
>>> mip.install("github:org/repo", version="branch-or-tag")
>>> mip.install("gitlab:org/repo")  # Uses default branch of that repo
>>> mip.install("gitlab:org/repo", version="branch-or-tag")
>>> mip.install("codeberg:org/repo")  # Uses default branch of that repo
>>> mip.install("codeberg:org/repo", version="branch-or-tag")
```

Using `mip` on the Unix port

On the Unix port, `mip` can be used at the REPL as above, and also by using `-m`:

```
$ ./micropython -m mip install pkgname-or-url
$ ./micropython -m mip install pkgname-or-url@version
```

The `--target path`, `--no-mpy`, and `--index` arguments can be set:

```
$ ./micropython -m mip install --target third-party pkgname
$ ./micropython -m mip install --no-mpy pkgname
$ ./micropython -m mip install --index https://host/pi pkgname
```

2.10.2 Installing packages with `mpremote`

The *mpremote* tool also includes the same functionality as `mip` and can be used from a host PC to install packages to a locally connected device (e.g. via USB or UART):

```
$ mpremote mip install pkgname
$ mpremote mip install pkgname@x.y
$ mpremote mip install http://example.com/x/y/foo.py
$ mpremote mip install github:org/repo
$ mpremote mip install github:org/repo@branch-or-tag
$ mpremote mip install gitlab:org/repo
$ mpremote mip install gitlab:org/repo@branch-or-tag
$ mpremote mip install codeberg:org/repo
$ mpremote mip install codeberg:org/repo@branch-or-tag
```

The `--target=path`, `--no-mpy`, and `--index` arguments can be set:

```
$ mpremote mip install --target=/flash/third-party pkgname
$ mpremote mip install --no-mpy pkgname
$ mpremote mip install --index https://host/pi pkgname
```

mpremote can also install packages from files stored on the host's local filesystem:

```
$ mpremote mip install path/to/pkg.py
$ mpremote mip install path/to/app/package.json
$ mpremote mip install \\path\\to\\pkg.py
```

This is especially useful for testing packages during development and for installing packages from local clones of GitHub repositories. Note that URLs in `package.json` files must use forward slashes ("/") as directory separators, even on Windows, so that they are compatible with installing from the web.

2.10.3 Installing packages manually

Packages can also be installed (in either .py or .mpy form) by manually copying the files to the device. Depending on the board this might be via USB Mass Storage, the *mpremote* tool (e.g. `mpremote fs cp path/to/package.py :package.py`), *webrepl*, etc.

2.10.4 Writing & publishing packages

Publishing to *micropython-lib* is the easiest way to make your package broadly accessible to MicroPython users, and automatically available via `mip` and `mpremote` and compiled to bytecode. See https://github.com/micropython/micropython-lib for more information.

To write a "self-hosted" package that can be downloaded by `mip` or `mpremote`, you need a static webserver (or GitHub) to host either a single .py file, or a `package.json` file alongside your .py files.

An example `mlx90640` library hosted on GitHub could be installed with:

```
$ mpremote mip install github:org/micropython-mlx90640
```

The layout for the package on GitHub might look like:

```
https://github.com/org/micropython-mlx90640/
    package.json
    mlx90640/
        __init__.py
        utils.py
```

The `package.json` specifies the location of files to be installed and other dependencies:

```
{
  "urls": [
    ["mlx90640/__init__.py", "mlx90640/__init__.py"],
    ["mlx90640/utils.py", "mlx90640/utils.py"]
  ],
  "deps": [
    ["collections-defaultdict", "latest"],
    ["os-path", "latest"],
    ["github:org/micropython-additions", "main"],
    ["gitlab:org/micropython-otheradditions", "main"]
  ],
```

(continues on next page)

(continued from previous page)

```
  "version": "0.2"
}
```

The `urls` list specifies the files to be installed according to:

```
"urls": [
    [destination_path, source_url]
    ...
```

where `destination_path` is the location and name of the file to be installed on the device and `source_url` is the URL of the file to be installed. The source URL would usually be specified relative to the directory containing the `package.json` file, but can also be an absolute URL, eg:

```
["mlx90640/utils.py", "github:org/micropython-mlx90640/mlx90640/utils.py"]
```

The package depends on `collections-defaultdict` and `os-path` which will be installed automatically from the *micropython-lib*. The third dependency installs the content as defined by the `package.json` file of the `main` branch of the GitHub repo `org/micropython-additions`.

2.10.5 Freezing packages

When a Python module or package is imported from the device filesystem, it is compiled into *bytecode* in RAM, ready to be executed by the VM. For a *.mpy file*, this conversion has been done already, but the bytecode still ends up in RAM.

For low-memory devices, or for large applications, it can be advantageous to instead run the bytecode from ROM (i.e. flash memory). This can be done by "freezing" the bytecode into the MicroPython firmware, which is then flashed to the device. The runtime performance is the same (although importing is faster), but it can free up significant amounts of RAM for your program to use.

The downside of this approach is that it's much slower to develop, because you have to flash the firmware each time, but it can be still useful to freeze dependencies that don't change often.

Freezing is done by writing a manifest file and using it in the build, often as part of a custom board definition. See the *MicroPython manifest files* guide for more information.

2.11 Inline assembler for Thumb2 architectures

This document assumes some familiarity with assembly language programming and should be read after studying the *tutorial*. For a detailed description of the instruction set consult the Architecture Reference Manual detailed below. The inline assembler supports a subset of the ARM Thumb-2 instruction set described here. The syntax tries to be as close as possible to that defined in the above ARM manual, converted to Python function calls.

Instructions operate on 32 bit signed integer data except where stated otherwise. Most supported instructions operate on registers `R0-R7` only: where `R8-R15` are supported this is stated. Registers `R8-R12` must be restored to their initial value before return from a function. Registers `R13-R15` constitute the Link Register, Stack Pointer and Program Counter respectively.

2.11.1 Document conventions

Where possible the behaviour of each instruction is described in Python, for example

- add(Rd, Rn, Rm) `Rd = Rn + Rm`

This enables the effect of instructions to be demonstrated in Python. In certain case this is impossible because Python doesn't support concepts such as indirection. The pseudocode employed in such cases is described on the relevant page.

2.11.2 Instruction categories

The following sections details the subset of the ARM Thumb-2 instruction set supported by MicroPython.

Register move instructions

Document conventions

Notation: `Rd, Rn` denote ARM registers R0-R15. `immN` denotes an immediate value having a width of N bits. These instructions affect the condition flags.

Register moves

Where immediate values are used, these are zero-extended to 32 bits. Thus `mov(R0, 0xff)` will set R0 to 255.

- mov(Rd, imm8) `Rd = imm8`
- mov(Rd, Rn) `Rd = Rn`
- movw(Rd, imm16) `Rd = imm16`
- movt(Rd, imm16) `Rd = (Rd & 0xffff) | (imm16 << 16)`

movt writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

- movwt(Rd, imm32) `Rd = imm32`

movwt is a pseudo-instruction: the MicroPython assembler emits a `movw` followed by a `movt` to move a 32-bit value into Rd.

Load register from memory

Document conventions

Notation: `Rt, Rn` denote ARM registers R0-R7 except where stated. `immN` represents an immediate value having a width of N bits hence `imm5` is constrained to the range 0-31. `[Rn + immN]` is the contents of the memory address obtained by adding Rn and the offset `immN`. Offsets are measured in bytes. These instructions affect the condition flags.

Register Load

- ldr(Rt, [Rn, imm7]) `Rt = [Rn + imm7]` Load a 32 bit word
- ldrb(Rt, [Rn, imm5]) `Rt = [Rn + imm5]` Load a byte
- ldrh(Rt, [Rn, imm6]) `Rt = [Rn + imm6]` Load a 16 bit half word

Where a byte or half word is loaded, it is zero-extended to 32 bits.

The specified immediate offsets are measured in bytes. Hence in the case of `ldr` the 7 bit value enables 32 bit word aligned values to be accessed with a maximum offset of 31 words. In the case of `ldrh` the 6 bit value enables 16 bit half-word aligned values to be accessed with a maximum offset of 31 half-words.

Store register to memory

Document conventions

Notation: `Rt, Rn` denote ARM registers R0-R7 except where stated. `immN` represents an immediate value having a width of N bits hence `imm5` is constrained to the range 0-31. `[Rn + imm5]` is the contents of the memory address obtained by adding Rn and the offset `imm5`. Offsets are measured in bytes. These instructions do not affect the condition flags.

Register Store

- str(Rt, [Rn, imm7]) `[Rn + imm7] = Rt` Store a 32 bit word
- strb(Rt, [Rn, imm5]) `[Rn + imm5] = Rt` Store a byte (b0-b7)
- strh(Rt, [Rn, imm6]) `[Rn + imm6] = Rt` Store a 16 bit half word (b0-b15)

The specified immediate offsets are measured in bytes. Hence in the case of `str` the 7 bit value enables 32 bit word aligned values to be accessed with a maximum offset of 31 words. In the case of `strh` the 6 bit value enables 16 bit half-word aligned values to be accessed with a maximum offset of 31 half-words.

Logical & bitwise instructions

Document conventions

Notation: `Rd, Rn` denote ARM registers R0-R7 except in the case of the special instructions where R0-R15 may be used. `Rn<a-b>` denotes an ARM register whose contents must lie in range `a <= contents <= b`. In the case of instructions with two register arguments, it is permissible for them to be identical. For example the following will zero R0 (Python `R0 ^= R0`) regardless of its initial contents.

- eor(r0, r0)

These instructions affect the condition flags except where stated.

Logical instructions

- and_(Rd, Rn) `Rd &= Rn`
- orr(Rd, Rn) `Rd |= Rn`
- eor(Rd, Rn) `Rd ^= Rn`
- mvn(Rd, Rn) `Rd = Rn ^ 0xffffffff` i.e. Rd = 1's complement of Rn
- bic(Rd, Rn) `Rd &= ~Rn` bit clear Rd using mask in Rn

Note the use of "and_" instead of "and", because "and" is a reserved keyword in Python.

Shift and rotation instructions

- lsl(Rd, Rn<0-31>) `Rd <<= Rn`
- lsr(Rd, Rn<1-32>) `Rd = (Rd & 0xffffffff) >> Rn` Logical shift right
- asr(Rd, Rn<1-32>) `Rd >>= Rn` arithmetic shift right
- ror(Rd, Rn<1-31>) `Rd = rotate_right(Rd, Rn)` Rd is rotated right Rn bits.

A rotation by (for example) three bits works as follows. If Rd initially contains bits `b31 b30..b0` after rotation it will contain `b2 b1 b0 b31 b30..b3`

Special instructions

Condition codes are unaffected by these instructions.

- clz(Rd, Rn) `Rd = count_leading_zeros(Rn)`

count_leading_zeros(Rn) returns the number of binary zero bits before the first binary one bit in Rn.

- rbit(Rd, Rn) `Rd = bit_reverse(Rn)`

bit_reverse(Rn) returns the bit-reversed contents of Rn. If Rn contains bits `b31 b30..b0` Rd will be set to `b0 b1 b2..b31`

Trailing zeros may be counted by performing a bit reverse prior to executing clz.

Arithmetic instructions

Document conventions

Notation: `Rd, Rm, Rn` denote ARM registers R0-R7. `immN` denotes an immediate value having a width of N bits e.g. `imm8`, `imm3`. `carry` denotes the carry condition flag, `not(carry)` denotes its complement. In the case of instructions with more than one register argument, it is permissible for some to be identical. For example the following will add the contents of R0 to itself, placing the result in R0:

- add(r0, r0, r0)

Arithmetic instructions affect the condition flags except where stated.

Addition

- add(Rdn, imm8) `Rdn = Rdn + imm8`
- add(Rd, Rn, imm3) `Rd = Rn + imm3`
- add(Rd, Rn, Rm) `Rd = Rn +Rm`
- adc(Rd, Rn) `Rd = Rd + Rn + carry`

Subtraction

- sub(Rdn, imm8) `Rdn = Rdn - imm8`
- sub(Rd, Rn, imm3) `Rd = Rn - imm3`
- sub(Rd, Rn, Rm) `Rd = Rn - Rm`
- sbc(Rd, Rn) `Rd = Rd - Rn - not(carry)`

Negation

- neg(Rd, Rn) `Rd = -Rn`

Multiplication and division

- mul(Rd, Rn) `Rd = Rd * Rn`

This produces a 32 bit result with overflow lost. The result may be treated as signed or unsigned according to the definition of the operands.

- sdiv(Rd, Rn, Rm) `Rd = Rn / Rm`
- udiv(Rd, Rn, Rm) `Rd = Rn / Rm`

These functions perform signed and unsigned division respectively. Condition flags are not affected.

Comparison instructions

These perform an arithmetic or logical instruction on two arguments, discarding the result but setting the condition flags. Typically these are used to test data values without changing them prior to executing a conditional branch.

Document conventions

Notation: `Rd, Rm, Rn` denote ARM registers R0-R7. `imm8` denotes an immediate value having a width of 8 bits.

The Application Program Status Register (APSR)

This contains four bits which are tested by the conditional branch instructions. Typically a conditional branch will test multiple bits, for example `bge(LABEL)`. The meaning of condition codes can depend on whether the operands of an arithmetic instruction are viewed as signed or unsigned integers. Thus `bhi(LABEL)` assumes unsigned numbers were processed while `bgt(LABEL)` assumes signed operands.

APSR Bits

- Z (zero)

This is set if the result of an operation is zero or the operands of a comparison are equal.

- N (negative)

Set if the result is negative.

- C (carry)

An addition sets the carry flag when the result overflows out of the MSB, for example adding 0x80000000 and 0x80000000. By the nature of two's complement arithmetic this behaviour is reversed on subtraction, with a borrow indicated by the carry bit being clear. Thus 0x10 - 0x01 is executed as 0x10 + 0xffffffff which will set the carry bit.

- V (overflow)

The overflow flag is set if the result, viewed as a two's compliment number, has the "wrong" sign in relation to the operands. For example adding 1 to 0x7fffffff will set the overflow bit because the result (0x8000000), viewed as a two's complement integer, is negative. Note that in this instance the carry bit is not set.

Comparison instructions

These set the APSR (Application Program Status Register) N (negative), Z (zero), C (carry) and V (overflow) flags.

- cmp(Rn, imm8) `Rn - imm8`
- cmp(Rn, Rm) `Rn - Rm`
- cmn(Rn, Rm) `Rn + Rm`
- tst(Rn, Rm) `Rn & Rm`

Conditional execution

The `it` and `ite` instructions provide a means of conditionally executing from one to four subsequent instructions without the need for a label.

- it(<condition>) If then

Execute the next instruction if <condition> is true:

```
cmp(r0, r1)
it(eq)
mov(r0, 100) # runs if r0 == r1
# execution continues here
```

- ite(<condition>) If then else

If <condition> is true, execute the next instruction, otherwise execute the subsequent one. Thus:

```
cmp(r0, r1)
ite(eq)
mov(r0, 100) # runs if r0 == r1
mov(r0, 200) # runs if r0 != r1
# execution continues here
```

This may be extended to control the execution of up to four subsequent instructions: it[x[y[z]]] where x,y,z=t/e; e.g. itt, itee, itete, ittte, itttt, iteee, etc.

Branch instructions

These cause execution to jump to a target location usually specified by a label (see the `label` assembler directive). Conditional branches and the `it` and `ite` instructions test the Application Program Status Register (APSR) N (negative), Z (zero), C (carry) and V (overflow) flags to determine whether the branch should be executed.

Most of the exposed assembler instructions (including move operations) set the flags but there are explicit comparison instructions to enable values to be tested.

Further detail on the meaning of the condition flags is provided in the section describing comparison functions.

Document conventions

Notation: `Rm` denotes ARM registers R0-R15. `LABEL` denotes a label defined with the `label()` assembler directive. `<condition>` indicates one of the following condition specifiers:

- eq Equal to (result was zero)
- ne Not equal
- cs Carry set
- cc Carry clear
- mi Minus (negative)
- pl Plus (positive)
- vs Overflow set
- vc Overflow clear
- hi > (unsigned comparison)
- ls <= (unsigned comparison)
- ge >= (signed comparison)
- lt < (signed comparison)
- gt > (signed comparison)
- le <= (signed comparison)

Branch to label

- b(LABEL) Unconditional branch
- beq(LABEL) branch if equal
- bne(LABEL) branch if not equal
- bge(LABEL) branch if greater than or equal
- bgt(LABEL) branch if greater than
- blt(LABEL) branch if less than (<) (signed)
- ble(LABEL) branch if less than or equal to (<=) (signed)
- bcs(LABEL) branch if carry flag is set
- bcc(LABEL) branch if carry flag is clear
- bmi(LABEL) branch if negative
- bpl(LABEL) branch if positive
- bvs(LABEL) branch if overflow flag set
- bvc(LABEL) branch if overflow flag is clear
- bhi(LABEL) branch if higher (unsigned)
- bls(LABEL) branch if lower or equal (unsigned)

Long branches

The code produced by the branch instructions listed above uses a fixed bit width to specify the branch destination, which is PC relative. Consequently in long programs where the branch instruction is remote from its destination the assembler will produce a "branch not in range" error. This can be overcome with the "wide" variants such as

- beq_w(LABEL) long branch if equal

Wide branches use 4 bytes to encode the instruction (compared with 2 bytes for standard branch instructions).

Subroutines (functions)

When entering a subroutine the processor stores the return address in register r14, also known as the link register (lr). Return to the instruction after the subroutine call is performed by updating the program counter (r15 or pc) from the link register, This process is handled by the following instructions.

- bl(LABEL)

Transfer execution to the instruction after `LABEL` storing the return address in the link register (r14).

- bx(Rm) Branch to address specified by Rm.

Typically `bx(lr)` is issued to return from a subroutine. For nested subroutines the link register of outer scopes must be saved (usually on the stack) before performing inner subroutine calls.

Stack push and pop

Document conventions

The `push()` and `pop()` instructions accept as their argument a register set containing a subset, or possibly all, of the general-purpose registers R0-R12 and the link register (lr or R14). As with any Python set the order in which the registers are specified is immaterial. Thus the in the following example the pop() instruction would restore R1, R7 and R8 to their contents prior to the push():

- push({r1, r8, r7}) Save three registers on the stack.
- pop({r7, r1, r8}) Restore them

Stack operations

- push({regset}) Push a set of registers onto the stack
- pop({regset}) Restore a set of registers from the stack

Miscellaneous instructions

- nop() `pass` no operation.
- wfi() Suspend execution in a low power state until an interrupt occurs.
- cpsid(flags) set the Priority Mask Register - disable interrupts.
- cpsie(flags) clear the Priority Mask Register - enable interrupts.
- mrs(Rd, special_reg) `Rd = special_reg` copy a special register to a general register. The special register may be IPSR (Interrupt Status Register) or BASEPRI (Base Priority Register). The IPSR provides a means of determining the exception number of an interrupt being processed. It contains zero if no interrupt is being processed.

Currently the `cpsie()` and `cpsid()` functions are partially implemented. They require but ignore the flags argument and serve as a means of enabling and disabling interrupts.

Floating point instructions

These instructions support the use of the ARM floating point coprocessor (on platforms such as the Pyboard which are equipped with one). The FPU has 32 registers known as `s0-s31` each of which can hold a single precision float. Data can be passed between the FPU registers and the ARM core registers with the `vmov` instruction.

Note that MicroPython doesn't support passing floats to assembler functions, nor can you put a float into `r0` and expect a reasonable result. There are two ways to overcome this. The first is to use arrays, and the second is to pass and/or return integers and convert to and from floats in code.

Document conventions

Notation: `Sd, Sm, Sn` denote FPU registers, `Rd, Rm, Rn` denote ARM core registers. The latter can be any ARM core register although registers `R13-R15` are unlikely to be appropriate in this context.

Arithmetic

- vadd(Sd, Sn, Sm) `Sd = Sn + Sm`
- vsub(Sd, Sn, Sm) `Sd = Sn - Sm`
- vneg(Sd, Sm) `Sd = -Sm`
- vmul(Sd, Sn, Sm) `Sd = Sn * Sm`
- vdiv(Sd, Sn, Sm) `Sd = Sn / Sm`
- vsqrt(Sd, Sm) `Sd = sqrt(Sm)`

Registers may be identical: `vmul(S0, S0, S0)` will execute `S0 = S0*S0`

Move between ARM core and FPU registers

- vmov(Sd, Rm) `Sd = Rm`
- vmov(Rd, Sm) `Rd = Sm`

The FPU has a register known as FPSCR, similar to the ARM core's APSR, which stores condition codes plus other data. The following instructions provide access to this.

- vmrs(APSR_nzcv, FPSCR)

Move the floating-point N, Z, C, and V flags to the APSR N, Z, C, and V flags.

This is done after an instruction such as an FPU comparison to enable the condition codes to be tested by the assembler code. The following is a more general form of the instruction.

- vmrs(Rd, FPSCR) `Rd = FPSCR`

Move between FPU register and memory

- vldr(Sd, [Rn, offset]) `Sd = [Rn + offset]`
- vstr(Sd, [Rn, offset]) `[Rn + offset] = Sd`

Where `[Rn + offset]` denotes the memory address obtained by adding Rn to the offset. This is specified in bytes. Since each float value occupies a 32 bit word, when accessing arrays of floats the offset must always be a multiple of four bytes.

Data comparison

- vcmp(Sd, Sm)

Compare the values in Sd and Sm and set the FPU N, Z, C, and V flags. This would normally be followed by `vmrs(APSR_nzcv, FPSCR)` to enable the results to be tested.

Convert between integer and float

- vcvt_f32_s32(Sd, Sm) `Sd = float(Sm)`
- vcvt_s32_f32(Sd, Sm) `Sd = int(Sm)`

Assembler directives

Labels

- label(INNER1)

This defines a label for use in a branch instruction. Thus elsewhere in the code a `b(INNER1)` will cause execution to continue with the instruction after the label directive.

Defining inline data

The following assembler directives facilitate embedding data in an assembler code block.

- data(size, d0, d1 .. dn)

The data directive creates n array of data values in memory. The first argument specifies the size in bytes of the subsequent arguments. Hence the first statement below will cause the assembler to put three bytes (with values 2, 3 and 4) into consecutive memory locations while the second will cause it to emit two four byte words.

```
data(1, 2, 3, 4)
data(4, 2, 100000)
```

Data values longer than a single byte are stored in memory in little-endian format.

- align(nBytes)

Align the following instruction to an nBytes value. ARM Thumb-2 instructions must be two byte aligned, hence it's advisable to issue `align(2)` after `data` directives and prior to any subsequent code. This ensures that the code will run irrespective of the size of the data array.

2.11.3 Usage examples

These sections provide further code examples and hints on the use of the assembler.

Hints and tips

The following are some examples of the use of the inline assembler and some information on how to work around its limitations. In this document the term "assembler function" refers to a function declared in Python with the `@micropython.asm_thumb` decorator, whereas "subroutine" refers to assembler code called from within an assembler function.

Code branches and subroutines

It is important to appreciate that labels are local to an assembler function. There is currently no way for a subroutine defined in one function to be called from another.

To call a subroutine the instruction `bl(LABEL)` is issued. This transfers control to the instruction following the `label(LABEL)` directive and stores the return address in the link register (`lr` or `r14`). To return the instruction `bx(lr)` is issued which causes execution to continue with the instruction following the subroutine call. This mechanism implies that, if a subroutine is to call another, it must save the link register prior to the call and restore it before terminating.

The following rather contrived example illustrates a function call. Note that it's necessary at the start to branch around all subroutine calls: subroutines end execution with `bx(lr)` while the outer function simply "drops off the end" in the style of Python functions.

```
@micropython.asm_thumb
def quad(r0):
    b(START)
    label(DOUBLE)
    add(r0, r0, r0)
    bx(lr)
    label(START)
    bl(DOUBLE)
    bl(DOUBLE)

print(quad(10))
```

The following code example demonstrates a nested (recursive) call: the classic Fibonacci sequence. Here, prior to a recursive call, the link register is saved along with other registers which the program logic requires to be preserved.

```
@micropython.asm_thumb
def fib(r0):
    b(START)
    label(DOFIB)
    push({r1, r2, lr})
    cmp(r0, 1)
    ble(FIBDONE)
    sub(r0, 1)
    mov(r2, r0) # r2 = n -1
    bl(DOFIB)
    mov(r1, r0) # r1 = fib(n -1)
    sub(r0, r2, 1)
    bl(DOFIB)   # r0 = fib(n -2)
    add(r0, r0, r1)
    label(FIBDONE)
    pop({r1, r2, lr})
    bx(lr)
    label(START)
    bl(DOFIB)

for n in range(10):
    print(fib(n))
```

Argument passing and return

The tutorial details the fact that assembler functions can support from zero to three arguments, which must (if used) be named `r0`, `r1` and `r2`. When the code executes the registers will be initialised to those values.

The data types which can be passed in this way are integers and memory addresses. With current firmware all possible 32 bit values may be passed and returned. If the return value may have the most significant bit set a Python type hint should be employed to enable MicroPython to determine whether the value should be interpreted as a signed or unsigned integer: types are `int` or `uint`.

```
@micropython.asm_thumb
def uadd(r0, r1) -> uint:
    add(r0, r0, r1)
```

`hex(uadd(0x40000000,0x40000000))` will return 0x80000000, demonstrating the passing and return of integers where bits 30 and 31 differ.

The limitations on the number of arguments and return values can be overcome by means of the `array` module which enables any number of values of any type to be accessed.

Multiple arguments

If a Python array of integers is passed as an argument to an assembler function, the function will receive the address of a contiguous set of integers. Thus multiple arguments can be passed as elements of a single array. Similarly a function can return multiple values by assigning them to array elements. Assembler functions have no means of determining the length of an array: this will need to be passed to the function.

This use of arrays can be extended to enable more than three arrays to be used. This is done using indirection: the `uctypes` module supports `addressof()` which will return the address of an array passed as its argument. Thus you can populate an integer array with the addresses of other arrays:

```
from uctypes import addressof
@micropython.asm_thumb
def getindirect(r0):
    ldr(r0, [r0, 0]) # Address of array loaded from passed array
    ldr(r0, [r0, 4]) # Return element 1 of indirect array (24)

def testindirect():
    a = array.array('i',[23, 24])
    b = array.array('i',[0,0])
    b[0] = addressof(a)
    print(getindirect(b))
```

Non-integer data types

These may be handled by means of arrays of the appropriate data type. For example, single precision floating point data may be processed as follows. This code example takes an array of floats and replaces its contents with their squares.

```
from array import array

@micropython.asm_thumb
def square(r0, r1):
    label(LOOP)
    vldr(s0, [r0, 0])
```

(continues on next page)

(continued from previous page)

```
    vmul(s0, s0, s0)
    vstr(s0, [r0, 0])
    add(r0, 4)
    sub(r1, 1)
    bgt(LOOP)

a = array('f', (x for x in range(10)))
square(a, len(a))
print(a)
```

The uctypes module supports the use of data structures beyond simple arrays. It enables a Python data structure to be mapped onto a bytearray instance which may then be passed to the assembler function.

Named constants

Assembler code may be made more readable and maintainable by using named constants rather than littering code with numbers. This may be achieved thus:

```
MYDATA = const(33)

@micropython.asm_thumb
def foo():
    mov(r0, MYDATA)
```

The const() construct causes MicroPython to replace the variable name with its value at compile time. If constants are declared in an outer Python scope they can be shared between multiple assembler functions and with Python code.

Assembler code as class methods

MicroPython passes the address of the object instance as the first argument to class methods. This is normally of little use to an assembler function. It can be avoided by declaring the function as a static method thus:

```
class foo:
  @staticmethod
  @micropython.asm_thumb
  def bar(r0):
    add(r0, r0, r0)
```

Use of unsupported instructions

These can be coded using the data statement as shown below. While `push()` and `pop()` are supported the example below illustrates the principle. The necessary machine code may be found in the ARM v7-M Architecture Reference Manual. Note that the first argument of data calls such as

```
data(2, 0xe92d, 0x0f00) # push r8,r9,r10,r11
```

indicates that each subsequent argument is a two byte quantity.

Overcoming MicroPython's integer restriction

The Pyboard chip includes a CRC generator. Its use presents a problem in MicroPython because the returned values cover the full gamut of 32 bit quantities whereas small integers in MicroPython cannot have differing values in bits 30 and 31. This limitation is overcome with the following code, which uses assembler to put the result into an array and Python code to coerce the result into an arbitrary precision unsigned integer.

```
from array import array
import stm

def enable_crc():
    stm.mem32[stm.RCC + stm.RCC_AHB1ENR] |= 0x1000

def reset_crc():
    stm.mem32[stm.CRC+stm.CRC_CR] = 1

@micropython.asm_thumb
def getval(r0, r1):
    movwt(r3, stm.CRC + stm.CRC_DR)
    str(r1, [r3, 0])
    ldr(r2, [r3, 0])
    str(r2, [r0, 0])

def getcrc(value):
    a = array('i', [0])
    getval(a, value)
    return a[0] & 0xffffffff # coerce to arbitrary precision

enable_crc()
reset_crc()
for x in range(20):
    print(hex(getcrc(0)))
```

2.11.4 References

- *Assembler Tutorial*
- Wiki hints and tips
- MicroPython Inline Assembler source-code, emitinlinethumb.c
- ARM Thumb2 Instruction Set Quick Reference Card
- RM0090 Reference Manual
- ARM v7-M Architecture Reference Manual (Available on the ARM site after a simple registration procedure. Also available on academic sites but beware of out of date versions.)

2.12 Working with filesystems

Contents

- *Working with filesystems*
  - *VFS*
  - *Block devices*
    - *Built-in block devices*
      - *STM32 / Pyboard*
      - *ESP8266*
      - *ESP32*
    - *Custom block devices*
  - *Filesystems*
    - *FAT*
    - *Littlefs*
    - *Hybrid (STM32)*
    - *Hybrid (ESP32)*

This tutorial describes how MicroPython provides an on-device filesystem, allowing standard Python file I/O methods to be used with persistent storage.

MicroPython automatically creates a default configuration and auto-detects the primary filesystem, so this tutorial will be mostly useful if you want to modify the partitioning, filesystem type, or use custom block devices.

The filesystem is typically backed by internal flash memory on the device, but can also use external flash, RAM, or a custom block device.

On some ports (e.g. STM32), the filesystem may also be available over USB MSC to a host PC. *The pyboard.py tool* also provides a way for the host PC to access to the filesystem on all ports.

Note: This is mainly for use on bare-metal ports like STM32 and ESP32. On ports with an operating system (e.g. the Unix port) the filesystem is provided by the host OS.

2.12.1 VFS

MicroPython implements a Unix-like Virtual File System (VFS) layer. All mounted filesystems are combined into a single virtual filesystem, starting at the root `/`. Filesystems are mounted into directories in this structure, and at startup the working directory is changed to where the primary filesystem is mounted.

On STM32 / Pyboard, the internal flash is mounted at `/flash`, and optionally the SDCard at `/sd`. On ESP8266/ESP32, the primary filesystem is mounted at `/`.

2.12.2 Block devices

A block device is an instance of a class that implements the *vfs.AbstractBlockDev* protocol.

Built-in block devices

Ports provide built-in block devices to access their primary flash.

On power-on, MicroPython will attempt to detect the filesystem on the default flash and configure and mount it automatically. If no filesystem is found, MicroPython will attempt to create a FAT filesystem spanning the entire flash. Ports can also provide a mechanism to "factory reset" the primary flash, usually by some combination of button presses at power on.

STM32 / Pyboard

The *pyb.Flash* class provides access to the internal flash. On some boards which have larger external flash (e.g. Pyboard D), it will use that instead. The `start` kwarg should always be specified, i.e. `pyb.Flash(start=0)`.

Note: For backwards compatibility, when constructed with no arguments (i.e. `pyb.Flash()`), it only implements the simple block interface and reflects the virtual device presented to USB MSC (i.e. it includes a virtual partition table at the start).

ESP8266

The internal flash is exposed as a block device object which is created in the `flashbdev` module on start up. This object is by default added as a global variable so it can usually be accessed simply as `bdev`. This implements the extended interface.

ESP32

The *esp32.Partition* class implements a block device for partitions defined for the board. Like ESP8266, there is a global variable `bdev` which points to the default partition. This implements the extended interface.

Custom block devices

The following class implements a simple block device that stores its data in RAM using a `bytearray`:

```
class RAMBlockDev:
    def __init__(self, block_size, num_blocks):
        self.block_size = block_size
        self.data = bytearray(block_size * num_blocks)

    def readblocks(self, block_num, buf):
        for i in range(len(buf)):
            buf[i] = self.data[block_num * self.block_size + i]

    def writeblocks(self, block_num, buf):
        for i in range(len(buf)):
            self.data[block_num * self.block_size + i] = buf[i]

    def ioctl(self, op, arg):
```

(continues on next page)

(continued from previous page)

```
        if op == 4: # get number of blocks
            return len(self.data) // self.block_size
        if op == 5: # get block size
            return self.block_size
```

It can be used as follows:

```
import vfs

bdev = RAMBlockDev(512, 50)
vfs.VfsFat.mkfs(bdev)
vfs.mount(bdev, '/ramdisk')
```

An example of a block device that supports both the simple and extended interface (i.e. both signatures and behaviours of the *vfs.AbstractBlockDev.readblocks()* and *vfs.AbstractBlockDev.writeblocks()* methods) is:

```
class RAMBlockDev:
    def __init__(self, block_size, num_blocks):
        self.block_size = block_size
        self.data = bytearray(block_size * num_blocks)

    def readblocks(self, block_num, buf, offset=0):
        addr = block_num * self.block_size + offset
        for i in range(len(buf)):
            buf[i] = self.data[addr + i]

    def writeblocks(self, block_num, buf, offset=None):
        if offset is None:
            # do erase, then write
            for i in range(len(buf) // self.block_size):
                self.ioctl(6, block_num + i)
            offset = 0
        addr = block_num * self.block_size + offset
        for i in range(len(buf)):
            self.data[addr + i] = buf[i]

    def ioctl(self, op, arg):
        if op == 4: # block count
            return len(self.data) // self.block_size
        if op == 5: # block size
            return self.block_size
        if op == 6: # block erase
            return 0
```

As it supports the extended interface, it can be used with *littlefs*:

```
import vfs

bdev = RAMBlockDev(512, 50)
vfs.VfsLfs2.mkfs(bdev)
vfs.mount(bdev, '/ramdisk')
```

Once mounted, the filesystem (regardless of its type) can be used as it normally would be used from Python code, for example:

```
with open('/ramdisk/hello.txt', 'w') as f:
    f.write('Hello world')
print(open('/ramdisk/hello.txt').read())
```

For further details of the block device methods and their return values see *vfs.AbstractBlockDev*.

2.12.3 Filesystems

MicroPython ports can provide implementations of *FAT*, *littlefs v1* and *littlefs v2*.

The following table shows which filesystems are included in the firmware by default for given port/board combinations, however they can be optionally enabled in a custom firmware build.

Board	FAT	littlefs v1	littlefs v2
pyboard 1.0, 1.1, D	Yes	No	Yes
Other STM32	Yes	No	No
ESP8266 (1M)	No	No	Yes
ESP8266 (2M+)	Yes	No	Yes
ESP32	Yes	No	Yes

FAT

The main advantage of the FAT filesystem is that it can be accessed over USB MSC on supported boards (e.g. STM32) without any additional drivers required on the host PC.

However, FAT is not tolerant to power failure during writes and this can lead to filesystem corruption. For applications that do not require USB MSC, it is recommended to use littlefs instead.

To format the entire flash using FAT:

```
# ESP8266 and ESP32
import vfs
vfs.umount('/')
vfs.VfsFat.mkfs(bdev)
vfs.mount(bdev, '/')

# STM32
import os, vfs, pyb
vfs.umount('/flash')
vfs.VfsFat.mkfs(pyb.Flash(start=0))
vfs.mount(pyb.Flash(start=0), '/flash')
os.chdir('/flash')
```

Littlefs

Littlefs is a filesystem designed for flash-based devices, and is much more resistant to filesystem corruption.

Note: There are reports of littlefs v1 and v2 failing in certain situations, for details see littlefs issue 347 and littlefs issue 295.

To format the entire flash using littlefs v2:

```
# ESP8266 and ESP32
import vfs
vfs.umount('/')
vfs.VfsLfs2.mkfs(bdev)
vfs.mount(bdev, '/')

# STM32
import os, vfs, pyb
vfs.umount('/flash')
vfs.VfsLfs2.mkfs(pyb.Flash(start=0))
vfs.mount(pyb.Flash(start=0), '/flash')
os.chdir('/flash')
```

A littlefs filesystem can be still be accessed on a PC over USB MSC using the littlefs FUSE driver. Note that you must specify both the `--block_size` and `--block_count` options to override the defaults. For example (after building the littlefs-fuse executable):

```
$ ./lfs --block_size=4096 --block_count=512 -o allow_other /dev/sdb1 mnt
```

This will allow the board's littlefs filesystem to be accessed at the `mnt` directory. To get the correct values of `block_size` and `block_count` use:

```
import pyb
f = pyb.Flash(start=0)
f.ioctl(1, 1)  # initialise flash in littlefs raw-block mode
block_count = f.ioctl(4, 0)
block_size = f.ioctl(5, 0)
```

Hybrid (STM32)

By using the `start` and `len` kwargs to *pyb.Flash*, you can create block devices spanning a subset of the flash device.

For example, to configure the first 256kiB as FAT (and available over USB MSC), and the remainder as littlefs:

```
import os, vfs, pyb
vfs.umount('/flash')
p1 = pyb.Flash(start=0, len=256*1024)
p2 = pyb.Flash(start=256*1024)
vfs.VfsFat.mkfs(p1)
vfs.VfsLfs2.mkfs(p2)
vfs.mount(p1, '/flash')
vfs.mount(p2, '/data')
os.chdir('/flash')
```

This might be useful to make your Python files, configuration and other rarely-modified content available over USB MSC, but allowing for frequently changing application data to reside on littlefs with better resilience to power failure, etc.

The partition at offset `0` will be mounted automatically (and the filesystem type automatically detected), but you can add:

```
import vfs, pyb
p2 = pyb.Flash(start=256*1024)
vfs.mount(p2, '/data')
```

to `boot.py` to mount the data partition.

Hybrid (ESP32)

On ESP32, if you build custom firmware, you can modify `partitions.csv` to define an arbitrary partition layout.

At boot, the partition named "vfs" will be mounted at `/` by default, but any additional partitions can be mounted in your `boot.py` using:

```
import esp32, vfs
p = esp32.Partition.find(esp32.Partition.TYPE_DATA, label='foo')
vfs.mount(p, '/foo')
```

2.13 Working with ROMFS

Contents

- *Working with ROMFS*
  - *Overview*
  - *Port support*
  - *Enabling ROMFS for a port or board*
  - *Workflow*
  - *Automatic mounting at boot*
  - *Using mpremote to manage ROMFS*
  - *ROMFS usage examples*
    - *Deploying a simple application*
    - *Listing ROMFS contents from Python*
    - *Manually mount a second ROMFS image*
    - *Mount a ROMFS image stored in a file*
- *ROMFS filesystem format*

2.13.1 Overview

ROMFS (Read-Only Memory Filesystem) is a lightweight, read-only filesystem optimised for microcontrollers and embedded systems where code and data need to be stored in flash memory and accessed efficiently without being copied into RAM.

The key benefits of ROMFS are:

- **Zero-copy imports**: `.mpy` bytecode files stored in a ROMFS can be executed directly from flash memory (memory-mapped) rather than being copied into RAM first. This is similar to how *frozen modules* work, but does not require reflashing the entire firmware.
- **Low RAM overhead**: String and byte constant objects in `.mpy` files loaded from ROMFS are referenced directly from flash, not duplicated in RAM.
- **Flexible deployment**: A ROMFS image can be built on a host PC and deployed to the device using `mpremote`, without rebuilding the firmware.
- **Standard filesystem interface**: A ROMFS is mounted in the *VFS* and accessed via normal Python file operations (`open`, `os.listdir`, `import`, etc.).

ROMFS is complementary to both the read-write LittleFS/FAT filesystems (which live in other flash partitions) and to *frozen modules* (which are compiled into the firmware itself).

Note: ROMFS requires firmware that has been built with ROMFS support enabled (`MICROPY_VFS_ROM`). Not all ports or boards include this by default; check your board's documentation or build configuration.

2.13.2 Port support

The following ports support ROMFS. On these ports, if a ROMFS partition is configured for the board, it will be automatically detected at boot time and mounted at `/rom` in the VFS. Both `/rom` and `/rom/lib` are automatically added to `sys.path` so that modules stored there can be imported directly.

Port	Notes
alif	Supported on boards with ROMFS partition configured.
esp32	Supported with custom partition table.
esp8266	Supported on 2MiB+ boards (ESP8266_GENERIC FLASH_2M_ROMFS variant).
mimxrt	Supported on boards with ROMFS partition configured.
nrf	Supported on boards with ROMFS partition configured.
qemu	Supported (used for CI testing).
renesas-ra	Supported on boards with ROMFS partition configured.
rp2	Supported on boards with ROMFS partition configured.
samd	Supported on boards with ROMFS partition configured.
stm32	Supported on boards with ROMFS partition configured.
unix	Supported (primarily for testing).

2.13.3 Enabling ROMFS for a port or board

The ROMFS implementation is port specific at this time and requires too much detail to explain here.

2.13.4 Workflow

The typical workflow for using ROMFS is:

1. Create a directory on your PC with the Python files (or `.mpy` files) you want to deploy.
2. Use `mpremote romfs deploy <directory>` to build and deploy the ROMFS image to the device.
3. The ROMFS will be mounted at `/rom` on next boot (or can be mounted immediately if the device is soft-reset).
4. Python code on the device can then `import` modules from the ROMFS just like from any other filesystem.

For example, on the host PC, with a directory "myapp/" containing app.py:

```
$ mpremote romfs deploy myapp/
```

After a soft-reset, the device will have `/rom/app.mpy` available for import (or `/rom/app.py` if `mpy_cross` is not installed).

Alternatively, you can build the ROMFS image on the host PC first, then deploy it to the device:

```
$ mpremote romfs build myapp
$ mpremote romfs deploy myapp.romfs
```

See the *mpremote romfs commands* section for full details of the `mpremote` commands.

2.13.5 Automatic mounting at boot

When ROMFS support is enabled in the firmware, MicroPython will automatically attempt to mount the first ROM partition at `/rom` during initialisation (after `mp_init()`). If the partition contains a valid ROMFS image, it is mounted and both `/rom` and `/rom/lib` are added to `sys.path` automatically.

This means that after deploying a ROMFS image with `mpremote`, a soft-reset is sufficient to make the new modules importable.

If no valid ROMFS image is found in the partition (e.g. on a freshly-programmed board), the mount is silently skipped.

2.13.6 Using mpremote to manage ROMFS

The *mpremote* `romfs` command can query ROMFS partitions, build ROMFS images, and deploy images to a connected device. See *mpremote ROMFS commands* for command syntax, options, and examples.

2.13.7 ROMFS usage examples

Deploying a simple application

Suppose you have a project directory `myapp/` with the following structure:

```
── myapp
   ├── myapp.py
   ├── utils.py
   └── lib
       └── helper.py
```

To deploy it to the device's ROMFS:

```
$ mpremote romfs deploy myapp/
$ mpremote tree
  tree :
  :/
  └── rom
      ├── lib
      │   └── helper.mpy
      ├── myapp.mpy
      └── utils.mpy
```

After a soft-reset, the modules are importable from the ROMFS as its mount point and lib folder have been added to `sys.path`:

```
import myapp
import utils
import helper
```

Listing ROMFS contents from Python

After mounting, the ROMFS contents can be explored like any other filesystem:

```
import os

for entry in os.ilistdir('/rom'):
    print(entry)

# Or simply:
print(os.listdir('/rom'))
```

Manually mount a second ROMFS image

If there are multiple ROMFS partitions, it is possible to mount a second ROMFS image from another partition on the device. For example, if the second ROMFS partition is at index 2, you can mount it using:

```
import vfs
dev = vfs.rom_ioctl(2, 1)  # get second partition
vfs.mount(vfs.VfsRom(dev), "/rom2")
```

Mount a ROMFS image stored in a file

A ROMFS image stored as a file within a ROMFS can be mounted as a nested filesystem without copying it into RAM. For example, if `/rom/data.romfs` exists, you can mount it using:

```
# boot.py
import vfs
with open('/rom/data.romfs', 'rb') as f:
    fs = vfs.VfsRom(f)
vfs.mount(fs, '/data')
```

Then after a soft-reset, the nested ROMFS is available at `/data`:

```
$ mpremote tree
  tree :
  :/
  ├── boot.py
  ├── data
  │   └── facts.db
  ├── main.py
  └── rom
      ├── data.romfs
      ├── lib
      │   └── helper.mpy
      ├── myapp.mpy
      └── utils.mpy
```

2.14 ROMFS filesystem format

ROMFS is a flexible and extensible filesystem format designed to represent a directory hierarchy with files, where those files are read-only and their data can be memory mapped.

Concepts:

- `varuint` : An unsigned integer that is encoded in a variable number of bytes. It is stored big-endian with the high bit of the byte set if there are following bytes.
- `record` : A variable sized element with a type. It is stored as two `varuint`'s and then a payload. The first `varuint` is the record kind and the second `varuint` is the payload length (which may be zero bytes long).

A ROMFS filesystem is a record with record kind 0x14a6b1, chosen so the encoded value is `0xd2-0xcd-0x31` which is `"RM1"` with the first two bytes having their high bit set. If the ROMFS record's payload is non-empty then it contains records.

Record types:

- `0` – **unused**: Can be used to detect corruption of the filesystem.
- `1` – **padding/comments**: Can contain any data in the payload.
- `2` – **verbatim data**: Used to store file data.
- `3` – **indirect data**: Points to an offset within the ROMFS payload.
- `4` – **directory**: The payload contains a `varuint` giving the length of the directory name in bytes, followed by the name and optional nested records for the directory contents (including optional metadata).
- `5` – **file**: The payload contains a `varuint` giving the length of the filename in bytes, followed by the name and optional nested records.

Unknown record types are silently skipped, providing forwards compatibility.

This format is defined in `extmod/vfs_rom.c` in the MicroPython source. The Python implementation used by `mpremote` to build images is in `tools/mpremote/mpremote/romfs.py`.

See also:

- *Working with filesystems* – Overview of the MicroPython VFS and available filesystem types.
- *MicroPython manifest files* – How to freeze Python modules into firmware.
- *MicroPython .mpy files* – MicroPython `.mpy` binary file format.
- *MicroPython remote control: mpremote* – The full `mpremote` command reference.

2.15 Unicode Support

MicroPython provides Unicode support for strings. All Tier 1, 2 and 3 ports have Unicode support enabled by default, but it is possible to change that with a different build configuration.

2.15.1 Terminology

This document uses the following Unicode terms:

- **Code point**: a single Unicode value in the range U+0000 to U+10FFFF, for example U+0041 `A` or U+1F600 □. MicroPython strings are sequences of code points.
- **Character**: informally used to mean a code point. Be aware that a user-perceived character (a *grapheme*) may consist of several code points, such as a base letter followed by combining marks.
- **Byte**: a single 8-bit value. In UTF-8 each code point is stored as one to four bytes (see below).

Operations such as `len()`, indexing and slicing act on code points, not on graphemes or display width, so a base letter followed by a combining mark counts as two code points.

2.15.2 Character Encoding

MicroPython uses UTF-8 encoding for all strings. When Unicode support is enabled (`MICROPY_PY_BUILTINS_STR_UNICODE`), strings can contain any valid Unicode code point from U+0000 to U+10FFFF.

ASCII characters (0-127) are stored in a single byte, making them as memory-efficient as on systems without Unicode support. Multi-byte UTF-8 code points use 2-4 bytes depending on the code point:

- U+0000 to U+007F: 1 byte (ASCII)
- U+0080 to U+07FF: 2 bytes
- U+0800 to U+FFFF: 3 bytes
- U+10000 to U+10FFFF: 4 bytes

2.15.3 Encoding and Decoding

The *bytes.decode()* and *str.encode()* methods support the following encodings:

- UTF-8 (`'utf-8'` or `'utf8'`)
- ASCII (`'ascii'`)

Other encodings (such as `'latin-1'`, `'utf-16'`, etc.) are not supported and will raise `LookupError`. The encoding argument must also match one of the supported strings exactly (for example, `'utf8'` is valid but `'UTF8'` is not). *More details*.

Example:

```
>>> '□□□'.encode('utf-8')
b'\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e'
>>> b'\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e'.decode('utf-8')
'□□□'
```

Error Handling

When decoding bytes that contain invalid UTF-8 sequences, the `errors` parameter of *bytes.decode()* controls the behavior:

- `'strict'` (default): Raise `UnicodeError`
- `'ignore'`: Skip invalid bytes (requires `MICROPY_PY_BUILTINS_BYTES_DECODE_ERRORS`)
- `'replace'`: Replace invalid bytes with U+FFFD `�` (requires `MICROPY_PY_BUILTINS_BYTES_DECODE_ERRORS`)

Example:

```
>>> # Strict mode (default) raises an error
>>> b'hello\xffworld'.decode('utf-8')
UnicodeError: invalid UTF-8

>>> # Ignore mode skips invalid bytes
>>> b'hello\xffworld'.decode('utf-8', 'ignore')
'helloworld'

>>> # Replace mode substitutes replacement character
>>> b'hello\xffworld'.decode('utf-8', 'replace')
'hello□world'
```

For memory-conscious applications, consider using `'ignore'` mode when processing untrusted or partially corrupted data, as it avoids raising exceptions while still recovering valid text.

The same `errors` handling applies when decoding any bytes-like object, including via the `str()` constructor (for example `str(buf, 'utf-8', 'replace')` where `buf` is a `bytes`, `bytearray`, `memoryview` or `array` object).

Encoding to Bytes

Function `str.encode()` and the `bytes()` constructor accept an `encoding` argument which can be `'utf8'`, `'utf-8'` or `'ascii'`:

```
>>> "abc".encode("ascii")
b'abc'
```

If encoding `'ascii'` is specified then an exception is raised if the string contains non-ASCII characters.

The `'ignore'` and `'replace'` `errors` values are not supported in these conversions from string to bytes and *any errors argument is ignored*.

2.15.4 String Methods

When Unicode support is enabled, string methods operate on code points rather than bytes:

- *`str.center()`* - Counts code points for width calculation
- `len(s)` - Returns number of code points (not bytes)
- String indexing and slicing work on code-point boundaries
- No support for display width calculations (East Asian width, combining characters, etc.)

Example:

```
>>> s = 'Hello 🐍🐍'
>>> len(s)            # 8 code points
8
>>> len(s.encode())   # 12 bytes
12
>>> s.center(12)      # Centered by code-point count
'  Hello 🐍🐍  '
```

2.15.5 String Formatting

The `%c` format specifier and `{:c}` format code support full Unicode:

- Accepts code points from 0 to 0x10FFFF
- Properly encodes multi-byte UTF-8 code points
- Raises `ValueError` for invalid code points

Example:

```
>>> '%c' % 65           # ASCII
'A'
>>> '%c' % 0x03B1       # Greek α
'α'
>>> '%c' % 0x1F600      # Emoji 😀
'😀'
>>> '{:c}'.format(0x4E2D)   # Chinese 中
'中'

>>> # Invalid code point
>>> '%c' % 0x110000
ValueError: %c arg not in range(0x110000)
```

F-strings also support the `:c` format code:

```
>>> code_point = 0x2665  # Heart suit ♥
>>> f'I {code_point:c} Python'
'I ♥ Python'
```

2.15.6 Build Configuration

Unicode features are controlled by several build-time flags in `mpconfigport.h`:

MICROPY_PY_BUILTINS_STR_UNICODE
Enable Unicode string support. When enabled, strings can contain any valid Unicode character and string operations work on character boundaries rather than byte boundaries.

Default: Enabled at `MICROPY_CONFIG_ROM_LEVEL_BASIC_FEATURES` and above.

Enabled on all Tier 1, 2 and 3 ports.

MICROPY_PY_BUILTINS_STR_UNICODE_CHECK
Enable UTF-8 validation during string operations. When disabled, string operations may produce incorrect results with invalid UTF-8 sequences.

Default: Follows `MICROPY_PY_BUILTINS_STR_UNICODE` setting.

Enabled on all Tier 1, 2 and 3 ports.

MICROPY_PY_BUILTINS_BYTES_DECODE_ERRORS
Enable the `'ignore'` and `'replace'` error handlers for *bytes.decode()*. When enabled, invalid UTF-8 bytes can be either skipped (`'ignore'`) or replaced with U+FFFD (`'replace'`).

Default: Enabled at `MICROPY_CONFIG_ROM_LEVEL_EXTRA_FEATURES` and above.

Enabled on alif, esp32, esp8266, mimxrt, renesas-ra, rp2, samd (SAMD51 only), stm32, unix and webassembly ports.

Example Configuration

For a constrained port with limited flash, disable error handlers:

```
#define MICROPY_PY_BUILTINS_BYTES_DECODE_ERRORS (0)
```

For a port with more resources, enable all Unicode features:

```
#define MICROPY_CONFIG_ROM_LEVEL (MICROPY_CONFIG_ROM_LEVEL_EXTRA_FEATURES)
// This automatically enables:
// - MICROPY_PY_BUILTINS_STR_UNICODE
// - MICROPY_PY_BUILTINS_BYTES_DECODE_ERRORS
```

2.15.7 Limitations

MicroPython's Unicode support has some limitations compared to CPython:

- Only UTF-8 and ASCII encodings are supported
- No support for Unicode normalization
- No locale-aware string operations
- The `errors` parameter accepts only positional arguments (not keyword arguments)
- String methods like `upper()`, `lower()`, etc. work correctly only for ASCII
- The MicroPython interactive REPL and `input()` function currently have limited Unicode support. The line editor is unaware of the displayed width of characters: wide characters (for example many CJK characters) take two terminal columns, while a grapheme cluster (a base code point plus combining marks, or an emoji sequence) can span several code points yet occupy a single column. Because editing tracks code points rather than displayed columns, line-editing keys such as backspace and the left/right arrows may leave the cursor misaligned with the text shown on screen. A workaround is to place the Unicode text in a UTF-8 encoded MicroPython script and run it using `mpremote run <script.py>`.

2.16 The pyboard.py tool

This is a standalone Python tool that runs on your PC that provides a way to:

- Quickly run a Python script or command on a MicroPython device. This is useful while developing MicroPython programs to quickly test code without needing to copy files to/from the device.
- Access the filesystem on a device. This allows you to deploy your code to the device (even if the board doesn't support USB MSC).

Despite the name, `pyboard.py` works on all MicroPython ports that support the raw REPL (including STM32, ESP32, ESP8266, NRF).

You can download the latest version from GitHub. The only dependency is the `pyserial` library which can be installed from PiPy or your system package manager.

Running `pyboard.py --help` gives the following output:

```
usage: pyboard [-h] [-d DEVICE] [-b BAUDRATE] [-u USER] [-p PASSWORD]
               [-c COMMAND] [-w WAIT] [--follow | --no-follow] [-f]
               [files [files ...]]

Run scripts on the pyboard.

positional arguments:
  files                 input files

optional arguments:
  -h, --help            show this help message and exit
  -d DEVICE, --device DEVICE
                        the serial device or the IP address of the pyboard
  -b BAUDRATE, --baudrate BAUDRATE
                        the baud rate of the serial device
  -u USER, --user USER  the telnet login username
  -p PASSWORD, --password PASSWORD
                        the telnet login password
  -c COMMAND, --command COMMAND
```

(continues on next page)

(continued from previous page)

```
                        program passed in as string
  -w WAIT, --wait WAIT  seconds to wait for USB connected board to become
                        available
  --follow              follow the output after running the scripts
                        [default if no scripts given]
  -f, --filesystem      perform a filesystem action: cp local :device | cp
                        :device local | cat path | ls [path] | rm path | mkdir
                        path | rmdir path
```

2.16.1 Running a command on the device

This is useful for testing short snippets of code, or to script an interaction with the device.:

```
$ pyboard.py --device /dev/ttyACM0 -c 'print(1+1)'
2
```

If you are often interacting with the same device, you can set the environment variable `PYBOARD_DEVICE` as an alternative to using the `--device` command line option. For example, the following is equivalent to the previous example:

```
$ export PYBOARD_DEVICE=/dev/ttyACM0
$ pyboard.py -c 'print(1+1)'
```

Similarly, the `PYBOARD_BAUDRATE` environment variable can be used to set the default for the `--baudrate` option.

2.16.2 Running a script on the device

If you have a script, `app.py` that you want to run on a device, then use:

```
$ pyboard.py --device /dev/ttyACM0 app.py
```

Note that this doesn't actually copy app.py to the device's filesystem, it just loads the code into RAM and executes it. Any output generated by the program will be displayed.

If the program app.py does not finish then you'll need to stop `pyboard.py`, eg with Ctrl-C. The program `app.py` will still continue to run on the MicroPython device.

2.16.3 Filesystem access

Using the `-f` flag, the following filesystem operations are supported:

- `cat path` Print the contents of a file on the device.
- `cp src [src...] dest` Copy files to/from the device.
- `ls [path]` List contents of a directory (defaults to current working directory).
- `mkdir path` Create a directory.
- `rm path` Remove a file.
- `rmdir path` Remove a directory.
- `touch path` Create a file if it doesn't already exist.

The `cp` command uses a `ssh`-like convention for referring to local and remote files. Any path starting with a `:` will be interpreted as on the device, otherwise it will be local. So:

```
$ pyboard.py --device /dev/ttyACM0 -f cp main.py :main.py
```

will copy main.py from the current directory on the PC to a file named main.py on the device. The filename can be omitted, e.g.:

```
$ pyboard.py --device /dev/ttyACM0 -f cp main.py :
```

is equivalent to the above.

Some more examples:

```
# Copy main.py from the device to the local PC.
$ pyboard.py --device /dev/ttyACM0 -f cp :main.py main.py
# Same, but using . instead.
$ pyboard.py --device /dev/ttyACM0 -f cp :main.py .

# Copy three files to the device, keeping their names.
$ pyboard.py --device /dev/ttyACM0 -f cp main.py app.py foo.py :

# Remove a file from the device.
$ pyboard.py --device /dev/ttyACM0 -f rm util.py

# Print the contents of a file on the device.
$ pyboard.py --device /dev/ttyACM0 -f cat boot.py
...contents of boot.py...
```

2.16.4 Using the pyboard library

You can also use `pyboard.py` as a library for scripting interactions with a MicroPython board.

```
import pyboard
pyb = pyboard.Pyboard('/dev/ttyACM0', 115200)
pyb.enter_raw_repl()
ret = pyb.exec('print(1+1)')
print(ret)
pyb.exit_raw_repl()
```

2.17 MicroPython 2.0 Migration Guide

MicroPython 2.0 is the (currently in development, not yet available) next major release of MicroPython.

After maintaining API compatibility for almost a decade with the `1.x` series, in order to unblock some project-wide improvements MicroPython 2.0 will introduce a small number of breaking API changes that will require some programs to be updated. This guide explains how to update your Python code to accommodate these changes.

This document is a work-in-progress. As more work is done on MicroPython 2.0, more items will be added to the lists below.

Note: There are currently no MicroPython 2.0 firmware builds available for download. You can build it yourself by enabling the `MICROPY_PREVIEW_VERSION_2` config option. As it gets closer to being ready for release, builds will be provided for both `1.x.y` and `2.0.0-preview`.

2.17.1 Hardware and peripherals

Overview

The goal is to improve consistency in the `machine` APIs across different ports, making it easier to write code, documentation, and tutorials that work on any supported microcontroller.

This means that some ports' APIs need to change to match other ports.

Changes

None yet

2.17.2 OS & filesystem

Overview

The primary goal is to support the ability to execute *.mpy files* directly from the filesystem without first copying them into RAM. This improves code deployment time and reduces memory overhead and fragmentation.

Additionally, a further goal is to support a more flexible way of configuring partitions, filesystem types, and options like USB mass storage.

Changes

None yet

2.17.3 CPython compatibility

Overview

The goal is to improve compatibility with CPython by removing MicroPython extensions from CPython APIs. In most cases this means moving existing MicroPython-specific functions or classes to new modules.

This makes it easier to write code that works on both CPython and MicroPython, which is useful for development and testing.

Changes

Introduction of a new module `vfs`. The following functions and classes have moved out of `os` to `vfs`: - `os.mount` - `os.umount` - `os.VfsFat` - `os.VfsLfs1` - `os.VfsLfs2` - `os.VfsPosix`

CHAPTER

THREE

MICROPYTHON DIFFERENCES FROM CPYTHON

MicroPython implements Python 3.4 and some select features of Python 3.5 and above. The sections below describe the current status of these features.

3.1 Python 3.5

Below is a list of finalised/accepted PEPs for Python 3.5 grouped into their impact to MicroPython.

Extensions to the syntax		**Status**
PEP 448	Additional unpacking generalizations	Partial
PEP 465	A new matrix multiplication operator	Complete
PEP 492	Coroutines with `async` and `await` syntax	Complete
Extensions and changes to runtime		
PEP 461	% formatting for binary strings	Complete
PEP 475	Retrying system calls that fail with `EINTR`	Complete
PEP 479	Change `StopIteration` handling inside generators	Complete
Standard library changes		
PEP 471	`os.scandir()`	
PEP 485	`math.isclose()`, a function for testing approximate equality	Complete
Miscellaneous changes		
PEP 441	Improved Python zip application support	
PEP 486	Make the Python Launcher aware of virtual environments	Not relevant
PEP 484	Type hints (advisory only)	Complete[1]
PEP 488	Elimination of PYO files	Not relevant
PEP 489	Redesigning extension module loading	

Other Language Changes:

Added the *namereplace* error handlers. The *backslashreplace* error handlers now work with decoding and translating.
Property docstrings are now writable. This is especially useful for collections.namedtuple() docstrings
Circular imports involving relative imports are now supported.

New Modules:

- typing

[1] The MicroPython parser correct ignores all type hints. However, the `typing` module is not built-in.

- zipzap

Changes to built-in modules:

collections	
The *OrderedDict* class is now implemented in C, which makes it 4 to 100 times faster.	
OrderedDict.items() , *OrderedDict.keys()* , *OrderedDict.values()* views now support reversed() iteration.	
The deque class now defines *index()*, *insert()*, and *copy()*, and supports the + and * operators.	
Docstrings produced by namedtuple() can now be updated.	
The UserString class now implements the *__getnewargs__()*, *__rmod__()*, *casefold()*, *format_map()*, *isprintable()*, and *maketrans()* methods to match the corresponding methods of str.	
heapq	
Element comparison in *merge()* can now be customized by passing a key function in a new optional key keyword argument, and a new optional *reverse* keyword argument can be used to reverse element comparison	
io	
A new *BufferedIOBase.readinto1()* method, that uses at most one call to the underlying raw stream's *RawIOBase.read()* or *RawIOBase.readinto()* methods	
json	
JSON decoder now raises JSONDecodeError instead of ValueError to provide better context information about the error.	
math	
Two new constants have been added to the math module: *inf* and *nan*.	Complete
A new function *isclose()* provides a way to test for approximate equality.	
A new *gcd()* function has been added. The *fractions.gcd()* function is now deprecated.	
os	
The new *scandir()* function returning an iterator of DirEntry objects has been added.	
The *urandom()* function now uses the *getrandom()* syscall on Linux 3.17 or newer, and *getentropy()* on OpenBSD 5.6 and newer, removing the need to use /dev/urandom and avoiding failures due to potential file descriptor exhaustion.	
New *get_blocking()* and *set_blocking()* functions allow getting and setting a file descriptor's blocking mode (O_NONBLOCK.)	
There is a new *os.path.commonpath()* function returning the longest common sub-path of each passed pathname	
re	
References and conditional references to groups with fixed length are now allowed in lookbehind assertions	
The number of capturing groups in regular expressions is no longer limited to 100.	
The *sub()* and *subn()* functions now replace unmatched groups with empty strings instead of raising an exception.	
The *re.error* exceptions have new attributes, msg, pattern, pos, lineno, and colno, that provide better context information about the error	
socket	
Functions with timeouts now use a monotonic clock, instead of a system clock.	
A new *socket.sendfile()* method allows sending a file over a socket by using the high-performance *os.sendfile()* function on UNIX, resulting in uploads being from 2 to 3 times faster than when using plain *socket.send()*	
The *socket.sendall()* method no longer resets the socket timeout every time bytes are received or sent. The socket timeout is now the maximum total duration to send all data.	
The backlog argument of the *socket.listen()* method is now optional. By default it is set to SOMAXCONN or to 128, whichever is less.	Complete
ssl	
Memory BIO Support	

continues on next page

Table 1 – continued from previous page

Application-Layer Protocol Negotiation Support
There is a new *SSLSocket.version()* method to query the actual protocol version in use.
The SSLSocket class now implements a *SSLSocket.sendfile()* method.
The *SSLSocket.send()* method now raises either the *ssl.SSLWantReadError* or *ssl.SSLWantWriteError* exception on a non-blocking socket if the operation would block. Previously, it would return 0.
The *cert_time_to_seconds()* function now interprets the input time as UTC and not as local time, per RFC 5280. Additionally, the return value is always an int.
New *SSLObject.shared_ciphers()* and *SSLSocket.shared_ciphers()* methods return the list of ciphers sent by the client during the handshake.
The *SSLSocket.do_handshake()*, *SSLSocket.read()*, *SSLSocket.shutdown()*, and *SSLSocket.write()* methods of the SSLSocket class no longer reset the socket timeout every time bytes are received or sent.
The *match_hostname()* function now supports matching of IP addresses.
sys
A new *set_coroutine_wrapper()* function allows setting a global hook that will be called whenever a coroutine object is created by an async def function. A corresponding *get_coroutine_wrapper()* can be used to obtain a currently set wrapper.
A new *is_finalizing()* function can be used to check if the Python interpreter is shutting down.
time
The *monotonic()* function is now always available

Notes

3.2 Python 3.6

Python 3.6 beta 1 was released on 12 Sep 2016, and a summary of the new features can be found here:

New Syntax Features		**Status**
PEP 498	Literal String Formatting	Complete
PEP 515	Underscores in Numeric Literals	Complete
PEP 525	Asynchronous Generators	
PEP 526	Syntax for Variable Annotations (provisional)	Complete
PEP 530	Asynchronous Comprehensions	
New Built-in Features		
PEP 468	Preserving the order of *kwargs* in a function	
PEP 487	Simpler customization of class creation	Partial[1]
PEP 520	Preserving Class Attribute Definition Order	
Standard Library Changes		
PEP 495	Local Time Disambiguation	
PEP 506	Adding A Secrets Module To The Standard Library	
PEP 519	Adding a file system path protocol	
CPython Internals		
PEP 509	Add a private version to dict	Won't do
PEP 523	Adding a frame evaluation API to CPython	
Linux/Window Changes		
PEP 524	Make `os.urandom()` blocking on Linux (during system startup)	
PEP 528	Change Windows console encoding to UTF-8	
PEP 529	Change Windows filesystem encoding to UTF-8	

[1] Currently, only `__set_name__()` is implemented.

Other Language Changes:

A *global* or *nonlocal* statement must now textually appear before the first use of the affected name in the same scope. Previously this was a SyntaxWarning.	
It is now possible to set a special method to None to indicate that the corresponding operation is not available. For example, if a class sets *__iter__()* to *None* , the class is not iterable.	
Long sequences of repeated traceback lines are now abbreviated as *[Previous line repeated {count} more times]*	
Import now raises the new exception *ModuleNotFoundError* when it cannot find a module. Code that currently checks for ImportError (in try-except) will still work.	
Class methods relying on zero-argument *super()* will now work correctly when called from metaclass methods during class creation.	

Changes to built-in modules:

array	
Exhausted iterators of *array.array* will now stay exhausted even if the iterated array is extended.	
binascii	
The b2a_base64() function now accepts an optional newline keyword argument to control whether the newline character is appended to the return value	Complete
cmath	
The new cmath.tau (τ) constant has been added	
New constants: *cmath.inf* and *cmath.nan* to match *math.inf* and *math.nan* , and also *cmath.infj* and *cmath.nanj* to match the format used by complex repr	
collections	
The new Collection abstract base class has been added to represent sized iterable container classes	
The new *Reversible* abstract base class represents iterable classes that also provide the *__reversed__()* method.	
The new *AsyncGenerator* abstract base class represents asynchronous generators.	
The *namedtuple()* function now accepts an optional keyword argument module, which, when specified, is used for the *__module__* attribute of the returned named tuple class.	
The verbose and rename arguments for *namedtuple()* are now keyword-only.	
Recursive *collections.deque* instances can now be pickled.	
hashlib	
BLAKE2 hash functions were added to the module. *blake2b()* and *blake2s()* are always available and support the full feature set of BLAKE2.	
The SHA-3 hash functions *sha3_224()*, *sha3_256()*, *sha3_384()*, *sha3_512()*, and *SHAKE* hash functions *shake_128()* and *shake_256()* were added.	
The password-based key derivation function *scrypt()* is now available with OpenSSL 1.1.0 and newer.	
json	
json.load() and *json.loads()* now support binary input. Encoded JSON should be represented using either UTF-8, UTF-16, or UTF-32.	
math	
The new math.tau (τ) constant has been added	Complete
os	
A new *close()* method allows explicitly closing a *scandir()* iterator. The *scandir()* iterator now supports the context manager protocol.	
On Linux, *os.urandom()* now blocks until the system urandom entropy pool is initialized to increase the security.	
The Linux *getrandom()* syscall (get random bytes) is now exposed as the new *os.getrandom()* function.	

continues on next page

Table 2 – continued from previous page

re
Added support of modifier spans in regular expressions. Examples: *'(?i:p)ython'* matches 'python' and 'Python', but not 'PYTHON'; *'(?i)g(?-i:v)r'* matches *'GvR'* and *'gvr'*, but not *'GVR'*.
Match object groups can be accessed by *__getitem__*, which is equivalent to *group()*. So *mo['name']* is now equivalent to *mo.group('name')*.
Match objects now support index-like objects as group indices.
socket
The *ioctl()* function now supports the *SIO_LOOPBACK_FAST_PATH* control code.
The *getsockopt()* constants *SO_DOMAIN* , *SO_PROTOCOL*, *SO_PEERSEC* , and *SO_PASSSEC* are now supported.
The *setsockopt()* now supports the *setsockopt(level, optname, None, optlen: int)* form.
The socket module now supports the address family *AF_ALG* to interface with Linux Kernel crypto API. *ALG_*, *SOL_ALG* and *sendmsg_afalg()* were added.
New Linux constants *TCP_USER_TIMEOUT* and *TCP_CONGESTION* were added.
ssl
ssl supports OpenSSL 1.1.0. The minimum recommend version is 1.0.2.
3DES has been removed from the default cipher suites and ChaCha20 Poly1305 cipher suites have been added.
SSLContext has better default configuration for options and ciphers.
SSL session can be copied from one client-side connection to another with the new *SSLSession* class. TLS session resumption can speed up the initial handshake, reduce latency and improve performance.
The new *get_ciphers()* method can be used to get a list of enabled ciphers in order of cipher priority.
All constants and flags have been converted to *IntEnum* and *IntFlags*.
Server and client-side specific TLS protocols for *SSLContext* were added.
Added *SSLContext.post_handshake_auth* to enable and *ssl.SSLSocket.verify_client_post_handshake()* to initiate TLS 1.3 post-handshake authentication.
struct
now supports IEEE 754 half-precision floats via the 'e' format specifier.
sys
The new *getfilesystemencodeerrors()* function returns the name of the error mode used to convert between Unicode filenames and bytes filenames.
zlib
The *compress()* and *decompress()* functions now accept keyword arguments

Notes

3.3 Python 3.7

New Features:

Feature		Status
PEP 538	Coercing the legacy C locale to a UTF-8 based locale	
PEP 539	A New C-API for Thread-Local Storage in CPython	
PEP 540	UTF-8 mode	
PEP 552	Deterministic pyc	
PEP 553	Built-in `breakpoint()`	
PEP 557	Data Classes	
PEP 560	Core support for typing module and generic types	
PEP 562	Module `__getattr__` and `__dir__`	Partial
PEP 563	Postponed Evaluation of Annotations	
PEP 564	Time functions with nanosecond resolution	Partial[1]
PEP 565	Show DeprecationWarning in `__main__`	
PEP 567	Context Variables	

Other Language Changes:

`async` and `await` are now reserved keywords	Complete
`dict` objects must preserve insertion-order	
More than 255 arguments can now be passed to a function; a function can now have more than 255 parameters	
`bytes.fromhex()` and `bytearray.fromhex()` now ignore all ASCII whitespace, not only spaces	
`str`, `bytes`, and `bytearray` gained support for the new `isascii()` method, which can be used to test if a string or bytes contain only the ASCII characters	
`ImportError` now displays module name and module `__file__` path when `from ... import ...` fails	
Circular imports involving absolute imports with binding a submodule to a name are now supported	
`object.__format__(x, '')` is now equivalent to `str(x)` rather than `format(str(self), '')`	
In order to better support dynamic creation of stack traces, `types.TracebackType` can now be instantiated from Python code, and the `tb_next` attribute on tracebacks is now writable	
When using the `-m` switch, `sys.path[0]` is now eagerly expanded to the full starting directory path, rather than being left as the empty directory (which allows imports from the current working directory at the time when an import occurs)	
The new `-X importtime` option or the `PYTHONPROFILEIMPORTTIME` environment variable can be used to show the timing of each module import	

Changes to built-in modules:

[1] Only *`time.time_ns()`* is implemented.

asyncio
Too many to list
gc
New features include *gc.freeze()*, *gc.unfreeze()*, *gc-get_freeze_count*
math
math.remainder() added to implement IEEE 754-style remainder
re
A number of tidy up features including better support for splitting on empty strings and copy support for compiled expressions and match objects
sys
sys.breakpointhook() added. sys.get(/set)_coroutine_origin_tracking_depth() added
time
Mostly updates to support nanosecond resolution in PEP564, see above

Notes

3.4 Python 3.8

Python 3.8.0 (final) was released on the 14 October 2019. The Features for 3.8 are defined in PEP 569 and a detailed description of the changes can be found in What's New in Python 3.8.

Features		**Status**
PEP 570	Positional-only arguments	
PEP 572	Assignment Expressions	Complete
PEP 574	Pickle protocol 5 with out-of-band data	
PEP 578	Runtime audit hooks	
PEP 587	Python Initialization Configuration	
PEP 590	Vectorcall: a fast calling protocol for CPython	
Miscellaneous		
f-strings support = for self-documenting expressions and debugging		Complete

Other Language Changes:

A *continue* statement was illegal in the *finally* clause due to a problem with the implementation. In Python 3.8 this restriction was lifted	Complete
The *bool*, *int* , and *fractions.Fraction* types now have an *as_integer_ratio()* method like that found in *float* and *decimal.Decimal*	
Constructors of *int*, *float* and *complex* will now use the *__index__()* special method, if available and the corresponding method *__int__()*, *__float__()* or *__complex__()* is not available	
Added support of *N{name}* escapes in regular expressions	
Dict and dictviews are now iterable in reversed insertion order using *reversed()*	
The syntax allowed for keyword names in function calls was further restricted. In particular, f((keyword)=arg) is no longer allowed	
Generalized iterable unpacking in yield and return statements no longer requires enclosing parentheses	
When a comma is missed in code such as [(10, 20) (30, 40)], the compiler displays a SyntaxWarning with a helpful suggestion	
Arithmetic operations between subclasses of *datetime.date* or *datetime.datetime* and *datetime.timedelta* objects now return an instance of the subclass, rather than the base class	
When the Python interpreter is interrupted by *Ctrl-C (SIGINT)* and the resulting *KeyboardInterrupt* exception is not caught, the Python process now exits via a SIGINT signal or with the correct exit code such that the calling process can detect that it died due to a *Ctrl-C*	
Some advanced styles of programming require updating the *types.CodeType* object for an existing function	
For integers, the three-argument form of the pow() function now permits the exponent to be negative in the case where the base is relatively prime to the modulus	
Dict comprehensions have been synced-up with dict literals so that the key is computed first and the value second	
The *object.__reduce__()* method can now return a tuple from two to six elements long	

Changes to built-in modules:

asyncio	
asyncio.run() has graduated from the provisional to stable API	Complete
Running *python -m asyncio* launches a natively async REPL	
The exception *asyncio.CancelledError* now inherits from *BaseException* rather than *Exception* and no longer inherits from *concurrent.futures.CancelledError*	Complete
Added *asyncio.Task.get_coro()* for getting the wrapped coroutine within an *asyncio.Task*	
Asyncio tasks can now be named, either by passing the name keyword argument to *asyncio.create_task()* or the *create_task()* event loop method, or by calling the *set_name()* method on the task object	
Added support for Happy Eyeballs to *asyncio.loop.create_connection()*. To specify the behavior, two new parameters have been added: *happy_eyeballs_delay* and interleave.	
gc	
get_objects() can now receive an optional generation parameter indicating a generation to get objects from. (Note, though, that while *gc* is a built-in, *get_objects()* is not implemented for MicroPython)	
math	
Added new function *math.dist()* for computing Euclidean distance between two points	
Expanded the *math.hypot()* function to handle multiple dimensions	
Added new function, *math.prod()*, as analogous function to *sum()* that returns the product of a "start" value (default: 1) times an iterable of numbers	
Added two new combinatoric functions *math.perm()* and *math.comb()*	
Added a new function *math.isqrt()* for computing accurate integer square roots without conversion to floating point	
The function *math.factorial()* no longer accepts arguments that are not int-like	Complete
sys	
Add new *sys.unraisablehook()* function which can be overridden to control how "unraisable exceptions" are handled	

3.5 Python 3.9

Python 3.9.0 (final) was released on the 5th October 2020. The Features for 3.9 are defined in PEP 596 and a detailed description of the changes can be found in What's New in Python 3.9

Features		Status
PEP 573	Fast access to module state from methods of C extension types	Not relevant
PEP 584	Union operators added to dict	Complete[1]
PEP 585	Type hinting generics in standard collections	
PEP 593	Flexible function and variable annotations	
PEP 602	CPython adopts an annual release cycle. Instead of annual, aiming for two month release cycle	Not relevant
PEP 614	Relaxed grammar restrictions on decorators	
PEP 615	The IANA Time Zone Database is now present in the standard library in the zoneinfo module	
PEP 616	String methods to remove prefixes and suffixes	
PEP 617	CPython now uses a new parser based on PEG	Not relevant

Other Language Changes:

[1] PEP 584 `dict` union operator is only available on MicroPython builds with `MICROPY_CPYTHON_COMPAT` enabled.

__import__() now raises *ImportError* instead of *ValueError*	Complete
Python now gets the absolute path of the script filename specified on the command line (ex: *python3 script.py*): the *__file__* attribute of the *__main__* module became an absolute path, rather than a relative path	
By default, for best performance, the errors argument is only checked at the first encoding/decoding error and the encoding argument is sometimes ignored for empty strings	
“”.replace(“”, s, n) now returns *s* instead of an empty string for all non-zero n. It is now consistent with *“”.replace(“”, s)*	
Any valid expression can now be used as a decorator. Previously, the grammar was much more restrictive	
Parallel running of *aclose()* / *asend()* / *athrow()* is now prohibited, and *ag_running* now reflects the actual running status of the async generator	
Unexpected errors in calling the *__iter__* method are no longer masked by TypeError in the in operator and functions contains(), indexOf() and countOf() of the operator module	
Unparenthesized lambda expressions can no longer be the expression part in an if clause in comprehensions and generator expressions	

Changes to built-in modules:

asyncio	
Due to significant security concerns, the reuse_address parameter of *asyncio.loop.create_datagram_endpoint()* is no longer supported	
Added a new coroutine *shutdown_default_executor()* that schedules a shutdown for the default executor that waits on the *ThreadPoolExecutor* to finish closing. Also, *asyncio.run()* has been updated to use the new coroutine.	
Added *asyncio.PidfdChildWatcher*, a Linux-specific child watcher implementation that polls process file descriptors	
added a new *coroutine asyncio.to_thread()*	
When cancelling the task due to a timeout, *asyncio.wait_for()* will now wait until the cancellation is complete also in the case when timeout is <= 0, like it does with positive timeouts	
asyncio now raises *TyperError* when calling incompatible methods with an *ssl.SSLSocket* socket	
gc	
Garbage collection does not block on resurrected objects	
Added a new function *gc.is_finalized()* to check if an object has been finalized by the garbage collector	
math	
Expanded the *math.gcd()* function to handle multiple arguments. Formerly, it only supported two arguments	
Added *math.lcm()*: return the least common multiple of specified arguments	
Added *math.nextafter()*: return the next floating-point value after x towards y	
Added *math.ulp()*: return the value of the least significant bit of a float	
os	
Exposed the Linux-specific *os.pidfd_open()* and *os.P_PIDFD*	
The *os.unsetenv()* function is now also available on Windows	Complete
The *os.putenv()* and *os.unsetenv()* functions are now always available	Complete
Added *os.waitstatus_to_exitcode()* function: convert a wait status to an exit code	
random	
Added a new *random.Random.randbytes* method: generate random bytes	
sys	
Added a new *sys.platlibdir* attribute: name of the platform-specific library directory	
Previously, *sys.stderr* was block-buffered when non-interactive. Now stderr defaults to always being line-buffered	

Notes

3.6 Python 3.10

Python 3.10.0 (final) was released on the 4 October 2021. The Features for 3.10 are defined in PEP 619 and a detailed description of the changes can be found in What's New in Python 3.10.

New syntax features		Status
PEP 634	Structural Pattern Matching: Specification	[1]
PEP 635	Structural Pattern Matching: Motivation and Rationale	Page 382, 1
PEP 636	Structural Pattern Matching: Tutorial	[1]
bpo-12782	Parenthesized context managers are now officially allowed	
New features in the standard library		
PEP 618	Add Optional Length-Checking To zip	
Interpreter improvements		
PEP 626	Precise line numbers for debugging and other tools	
New typing features		
PEP 604	Allow writing union types as X | Y	
PEP 613	Explicit Type Aliases	
PEP 612	Parameter Specification Variables	
Important deprecations, removals or restrictions		
PEP 644	Require OpenSSL 1.1.1 or newer	
PEP 632	Deprecate distutils module.	Not relevant
PEP 623	Deprecate and prepare for the removal of the wstr member in PyUnicodeObject.	Not relevant
PEP 624	Remove Py_UNICODE encoder APIs	Not relevant
PEP 597	Add optional EncodingWarning	

Other Language Changes:

[1] The structural pattern matching feature is discussed in issue #7847.

The *int* type has a new method `int.bit_count()`, returning the number of ones in the binary expansion of a given integer, also known as the population count.
The views returned by `dict.keys()`, `dict.values()` and `dict.items()` now all have a `mapping` attribute that gives a `types.MappingProxyType` object wrapping the original dictionary.
PEP 618: The *zip()* function now has an optional `strict` flag, used to require that all the iterables have an equal length.
Builtin and extension functions that take integer arguments no longer accept `Decimal`s, `Fraction`s and other objects that can be converted to integers only with a loss (e.g. that have the `__int__()` method but do not have the `__index__()` method).
If `object.__ipow__()` returns `NotImplemented`, the operator will correctly fall back to `object.__pow__()` and `object.__rpow__()` as expected.
Assignment expressions can now be used unparenthesized within set literals and set comprehensions, as well as in sequence indexes (but not slices).
Functions have a new `__builtins__` attribute which is used to look for builtin symbols when a function is executed, instead of looking into `__globals__['__builtins__']`. The attribute is initialized from `__globals__["__builtins__"]` if it exists, else from the current builtins.
Two new builtin functions – `aiter()` and `anext()` have been added to provide asynchronous counterparts to *iter()* and *next()*, respectively.
Static methods (*@staticmethod*) and class methods (*@classmethod*) now inherit the method attributes (`__module__`, `__name__`, `__qualname__`, `__doc__`, `__annotations__`) and have a new `__wrapped__` attribute. Moreover, static methods are now callable as regular functions.
Annotations for complex targets (everything beside `simple name` targets defined by **PEP 526**) no longer cause any runtime effects with `from __future__ import annotations`.
Class and module objects now lazy-create empty annotations dicts on demand. The annotations dicts are stored in the object's `__dict__` for backwards compatibility. This improves the best practices for working with `__annotations__`.
Annotations consist of `yield`, `yield from`, `await` or named expressions are now forbidden under `from __future__ import annotations` due to their side effects.
Usage of unbound variables, `super()` and other expressions that might alter the processing of symbol table as annotations are now rendered effectless under `from __future__ import annotations`.
Hashes of NaN values of both *float* type and `decimal.Decimal` type now depend on object identity. Formerly, they always hashed to `0` even though NaN values are not equal to one another. This caused potentially quadratic runtime behavior due to excessive hash collisions when creating dictionaries and sets containing multiple NaNs.
A *SyntaxError* (instead of a *NameError*) will be raised when deleting the `__debug__` constant.
SyntaxError exceptions now have `end_lineno` and `end_offset` attributes. They will be `None` if not determined.

Changes to built-in modules:

asyncio
Add missing `connect_accepted_socket()` method.
array
The `index()` method of *array.array* now has optional *start* and *stop* parameters.
gc
Add audit hooks for `gc.get_objects()`, `gc.get_referrers()` and `gc.get_referents()`.
hashlib
The hashlib module requires OpenSSL 1.1.1 or newer.
The hashlib module has preliminary support for OpenSSL 3.0.0.

continues on next page

Table 3 – continued from previous page

The pure-Python fallback of `pbkdf2_hmac()` is deprecated. In the future PBKDF2-HMAC will only be available when Python has been built with OpenSSL support.
os
Add `os.cpu_count()` support for VxWorks RTOS.
Add a new function `os.eventfd()` and related helpers to wrap the `eventfd2` syscall on Linux.
Add `os.splice()` that allows to move data between two file descriptors without copying between kernel address space and user address space, where one of the file descriptors must refer to a pipe.
Add `O_EVTONLY`, `O_FSYNC`, `O_SYMLINK` and `O_NOFOLLOW_ANY` for macOS.
platform
Add `platform.freedesktop_os_release()` to retrieve operation system identification from freedesktop.org os-release standard file.
socket
The exception `socket.timeout` is now an alias of `TimeoutError`.
Add option to create MPTCP sockets with `IPPROTO_MPTCP`.
Add `IP_RECVTOS` option to receive the type of service (ToS) or DSCP/ECN fields.
ssl
The ssl module requires OpenSSL 1.1.1 or newer.
The ssl module has preliminary support for OpenSSL 3.0.0 and new option `OP_IGNORE_UNEXPECTED_EOF`.
Deprecated function and use of deprecated constants now result in a `DeprecationWarning`. `ssl.SSLContext.options` has `OP_NO_SSLv2` and `OP_NO_SSLv3` set by default and therefore cannot warn about setting the flag again.
The ssl module now has more secure default settings. Ciphers without forward secrecy or SHA-1 MAC are disabled by default. Security level 2 prohibits weak RSA, DH, and ECC keys with less than 112 bits of security. *`SSLContext`* defaults to minimum protocol version TLS 1.2. Settings are based on Hynek Schlawack's research.
The deprecated protocols SSL 3.0, TLS 1.0, and TLS 1.1 are no longer officially supported. Python does not block them actively. However OpenSSL build options, distro configurations, vendor patches, and cipher suites may prevent a successful handshake.
Add a *timeout* parameter to the `ssl.get_server_certificate()` function.
The ssl module uses heap-types and multi-phase initialization.
A new verify flag `VERIFY_X509_PARTIAL_CHAIN` has been added.
sys
Add `sys.orig_argv` attribute: the list of the original command line arguments passed to the Python executable.
Add `sys.stdlib_module_names`, containing the list of the standard library module names.
_thread
`_thread.interrupt_main()` now takes an optional signal number to simulate (the default is still `signal.SIGINT`).

Notes

For the features of Python that are implemented by MicroPython, there are sometimes differences in their behaviour compared to standard Python. The operations listed in the sections below produce conflicting results in MicroPython when compared to standard Python.

3.7 Syntax

Generated Tue 29 Sep 2026 23:16:29 UTC

3.7.1 Annotations

MicroPython accepts type annotations on expressions where CPython forbids them.

Cause: To reduce code size, MicroPython does not check the form of expressions with type annotations applied.

Workaround: Always check for valid Python code using a linting tool.

The expressions themselves are not evaluated.

Sample code:

```
def test(expr):
    code = f"def f():\n    {expr}: int"
    print(code)
    try:
        exec(code)
        print("OK")
    except SyntaxError as e:
        print("SyntaxError")
    print()


test("print('test')")
test("[x,y]")
test("x,y")
```

CPython output:	MicroPython output:

`def f():`
`  print('test'): int`
`SyntaxError`
 	
`def f():`
`  [x,y]: int`
`SyntaxError`
 	
`def f():`
`  x,y: int`
`SyntaxError`
	`def f():`
	`  print('test'): int`
	`OK`
	
	`def f():`
	`  [x,y]: int`
	`OK`
	
	`def f():`
	`  x,y: int`
	`OK`

3.7.2 Literals

MicroPython accepts underscores in numeric literals where CPython doesn't

Cause: Different parser implementation

MicroPython's tokenizer ignores underscores in numeric literals, while CPython rejects multiple consecutive underscores and underscores after the last digit.

Workaround: Remove the underscores not accepted by CPython.

Sample code:

```
try:
    print(eval("1__1"))
except SyntaxError:
    print("Should not work")
try:
    print(eval("1_"))
except SyntaxError:
    print("Should not work")
```

CPython output:	MicroPython output:
`Should not work` `Should not work`	`11` `1`

MicroPython requires spaces between literal numbers and keywords or ".", CPython doesn't

Cause: Different parser implementation

MicroPython's tokenizer treats a sequence like `1and` as a single token, while CPython treats it as two tokens.

Since CPython 3.11, when the literal number is followed by a token, this syntax causes a `SyntaxWarning` for an "invalid literal". When a literal number is followed by a "." denoting attribute access, CPython does not warn.

Workaround: Add a space between the integer literal and the intended next token.

This also fixes the `SyntaxWarning` in CPython.

Sample code:

```
try:
    print(eval("1and 0"))
except SyntaxError:
    print("Should have worked")
try:
    print(eval("1or 0"))
except SyntaxError:
    print("Should have worked")
try:
    print(eval("1if 1else 0"))
except SyntaxError:
    print("Should have worked")
try:
    print(eval("0x1.to_bytes(1)"))
```

(continues on next page)

(continued from previous page)

```
except SyntaxError:
    print("Should have worked")
```

CPython output:

```
<string>:1: SyntaxWarning: invalid␣
↪decimal literal
0
<string>:1: SyntaxWarning: invalid␣
↪decimal literal
1
<string>:1: SyntaxWarning: invalid␣
↪decimal literal
<string>:1: SyntaxWarning: invalid␣
↪decimal literal
1
b'\x01'
```

MicroPython output:

```
Should have worked
Should have worked
Should have worked
Should have worked
```

3.7.3 Operators

MicroPython allows := to assign to the iteration variable in nested comprehensions, CPython does not.

Cause: MicroPython is optimised for code size. Although it is a syntax error to assign to the iteration variable in a standard comprehension (same as CPython), it doesn't check if an inner nested comprehension assigns to the iteration variable of the outer comprehension.

Workaround: Do not use := to assign to the iteration variable of a comprehension.

Sample code:

```
print([[(j := i) for i in range(2)] for j in range(2)])
```

CPython output:

```
  File "<stdin>", line 8
SyntaxError: assignment expression␣
↪cannot rebind comprehension iteration␣
↪variable 'j'
```

MicroPython output:

```
[[0, 1], [0, 1]]
```

3.7.4 Unicode

Unicode name escapes are not implemented

Sample code:

```
print("\N{LATIN SMALL LETTER A}")
```

CPython output:	MicroPython output:
`a`	`NotImplementedError: unicode name escapes`

3.7.5 Unpacking

Argument unpacking does not work if the argument being unpacked is the nth or greater argument where n is the number of bits in an MP_SMALL_INT.

Cause: The implementation uses an MP_SMALL_INT to flag args that need to be unpacked.

Workaround: Use fewer arguments.

Sample code:

```
def example(*args):
    print(len(args))


MORE = ["a", "b", "c"]

example(
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
    32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
    48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
    *MORE,
)
```

CPython output:	MicroPython output:

`67`	`Traceback (most recent call last):`
	`  File "<stdin>", line 21, in <module>`
	`SyntaxError: too many args`

3.8 Core language

Generated Tue 29 Sep 2026 23:16:29 UTC

3.8.1 Classes

Special method __del__ not implemented for user-defined classes

Sample code:

```
import gc


class Foo:
    def __del__(self):
        print("__del__")


f = Foo()
del f

gc.collect()
```

CPython output:	MicroPython output:
`__del__`	

dir() does not convert __dir__ return value to a sorted list

Cause: MicroPython's dir() returns the value from __dir__ as-is, without iterating it into a list or sorting it.

Workaround: Have __dir__ return a sorted list directly.

Sample code:

```
class C:
    def __dir__(self):
        return "cba"


print(dir(C()))
```

CPython output:	MicroPython output:
`['a', 'b', 'c']`	`cba`

`__init_subclass__` isn't automatically called.

Cause: MicroPython does not currently implement PEP 487.

Workaround: Manually call `__init_subclass__` after class creation if needed. e.g.:

```
class A(Base):
    pass
A.__init_subclass__()
```

Sample code:

```
class Base:
    @classmethod
    def __init_subclass__(cls):
        print(f"Base.__init_subclass__({cls.__name__})")


class A(Base):
    pass
```

CPython output:	MicroPython output:
`Base.__init_subclass__(A)`	

`__init_subclass__` isn't an implicit classmethod.

Cause: MicroPython does not currently implement PEP 487. `__init_subclass__` is not currently in the list of special-cased class/static methods.

Workaround: Decorate declarations of `__init_subclass__` with `@classmethod`.

Sample code:

```
def regularize_spelling(text, prefix="bound_"):
    # for regularizing across the CPython "method" vs MicroPython "bound_method"␣
↪spelling for the type of a bound classmethod
    if text.startswith(prefix):
        return text[len(prefix) :]
    return text


class A:
    def __init_subclass__(cls):
        pass

    @classmethod
    def manual_decorated(cls):
        pass


a = type(A.__init_subclass__).__name__
b = type(A.manual_decorated).__name__
```

(continues on next page)

(continued from previous page)

```
print(regularize_spelling(a))
print(regularize_spelling(b))
if a != b:
    print("FAIL")
```

CPython output:	MicroPython output:

`method`
`method`
	`function`
	`method`
	`FAIL`

MicroPython doesn't support parameterized `__init_subclass__` class customization.

Cause: MicroPython does not currently implement PEP 487. The MicroPython syntax tree does not include a kwargs node after the class inheritance list.

Workaround: Use class variables or another mechanism to specify base-class customizations.

Sample code:

```
class Base:
    @classmethod
    def __init_subclass__(cls, arg=None, **kwargs):
        cls.init_subclass_was_called = True
        print(f"Base.__init_subclass__({cls.__name__}, {arg=!r}, {kwargs=!r})")


class A(Base, arg="arg"):
    pass


# Regularize across MicroPython not automatically calling __init_subclass__ either.
if not getattr(A, "init_subclass_was_called", False):
    A.__init_subclass__()
```

CPython output:	MicroPython output:

`Base.__init_subclass__(A, arg='arg',`
`↪kwargs={})`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 16, in <module>`
	`TypeError: function doesn't take keyword`
	`↪arguments`

`__init_subclass__` can't be defined a cooperatively-recursive way.

Cause: MicroPython does not currently implement PEP 487. The base object type does not have an `__init_subclass__` implementation.

Workaround: Omit the recursive `__init_subclass__` call unless it's known that the grandparent also defines it.

Sample code:

```
class Base:
    @classmethod
    def __init_subclass__(cls, **kwargs):
        cls.init_subclass_was_called = True
        super().__init_subclass__(**kwargs)


class A(Base):
    pass


# Regularize across MicroPython not automatically calling __init_subclass__ either.
if not getattr(A, "init_subclass_was_called", False):
    A.__init_subclass__()
```

CPython output:	MicroPython output:

	`Traceback (most recent call last):`
	`  File "<stdin>", line 22, in <module>`
	`  File "<stdin>", line 13, in __init_`
	`↪subclass__`
	`AttributeError: 'super' object has no␣`
	`↪attribute '__init_subclass__'`

Method Resolution Order (MRO) is not compliant with CPython

Cause: Depth first non-exhaustive method resolution order

Workaround: Avoid complex class hierarchies with multiple inheritance and complex method overrides. Keep in mind that many languages don't support multiple inheritance at all.

Sample code:

```
class Foo:
    def __str__(self):
        return "Foo"


class C(tuple, Foo):
    pass


t = C((1, 2, 3))
print(t)
```

CPython output:	MicroPython output:
`Foo`	`(1, 2, 3)`

Private Class Members name mangling is not implemented

Cause: The MicroPython compiler does not implement name mangling for private class members.

Workaround: Avoid using or having a collision with global names, by adding a unique prefix to the private class member name manually.

Sample code:

```
def __print_string(string):
    print(string)


class Foo:
    def __init__(self, string):
        self.string = string

    def do_print(self):
        __print_string(self.string)


example_string = "Example String to print."

class_item = Foo(example_string)
print(class_item.string)

class_item.do_print()
```

CPython output:	MicroPython output:

`Example String to print.`
`Traceback (most recent call last):`
`  File "<stdin>", line 26, in <module>`
`  File "<stdin>", line 18, in do_print`
`NameError: name '_Foo__print_string' is not defined. Did you mean: '__print_string'?`
	`Example String to print.`
	`Example String to print.`

`__str__` returning non-string type does not raise TypeError

Cause: MicroPython's instance_print does not validate that `__str__` or `__repr__` return a str or its subclass

Workaround: Ensure `__str__` and `__repr__` always return a str instance or its subclass

Sample code:

```
class Foo:
    def __str__(self):
        return True


print(str(Foo()))
```

CPython output:	MicroPython output:

`Traceback (most recent call last):`
`  File "<stdin>", line 14, in <module>`
`TypeError: __str__ returned non-string (type bool)`
	`True`

str() does not preserve str subclass type from `__str__` or `__repr__` return value

Cause: Implementation discards str subclass type returned by `__str__` or `__repr__` and always returns a plain str instance from str().

Workaround: Do not rely on str() preserving str subclass types

Sample code:

```
class MyStr(str):
    pass


class Foo:
    def __str__(self):
        return MyStr("abc")


result = str(Foo())
print(type(result))
```

CPython output:	MicroPython output:
`<class '__main__.MyStr'>`	`<class 'str'>`

When inheriting native types, calling a method in `__init__(self, ...)` before `super().__init__()` raises an `AttributeError` (or segfaults if `MICROPY_BUILTIN_METHOD_CHECK_SELF_ARG` is not enabled).

Cause: MicroPython does not have separate `__new__` and `__init__` methods in native types.

Workaround: Call `super().__init__()` first.

Sample code:

```
class L1(list):
    def __init__(self, a):
        self.append(a)


try:
    L1(1)
    print("OK")
except AttributeError:
    print("AttributeError")


class L2(list):
    def __init__(self, a):
        super().__init__()
        self.append(a)


try:
    L2(1)
    print("OK")
except AttributeError:
    print("AttributeError")
```

CPython output:	MicroPython output:
`OK` `OK`	`AttributeError` `OK`

When inheriting from multiple classes super() only calls one class

Cause: See *Method Resolution Order (MRO) is not compliant with CPython*

Workaround: See *Method Resolution Order (MRO) is not compliant with CPython*

Sample code:

```
class A:
    def __init__(self):
        print("A.__init__")


class B(A):
    def __init__(self):
        print("B.__init__")
```

(continues on next page)

(continued from previous page)

```
        super().__init__()


class C(A):
    def __init__(self):
        print("C.__init__")
        super().__init__()


class D(B, C):
    def __init__(self):
        print("D.__init__")
        super().__init__()


D()
```

CPython output:	MicroPython output:

`D.__init__`
`B.__init__`
`C.__init__`
`A.__init__`
	`D.__init__`
	`B.__init__`
	`A.__init__`

Calling super() getter property in subclass will return a property object, not the value

Sample code:

```
class A:
    @property
    def p(self):
        return {"a": 10}


class AA(A):
    @property
    def p(self):
        return super().p


a = AA()
print(a.p)
```

CPython output:	MicroPython output:
`{'a': 10}`	`<property>`

3.8.2 Exceptions

Throwing a derived exception class instance in its `__init__` without first calling `super().__init__` is a TypeError

Cause: In MicroPython, an object is incompletely constructed if it does not call its superclass init function or return normally from its `__init__`. This prevents its usage in some circumstances.

Workaround: Call the superclass *`__init__`* method before raising the exception.

Sample code:

```
class C(Exception):
    def __init__(self):
        raise self


class C1(Exception):
    def __init__(self):
        super().__init__()
        raise self


try:
    C()
except Exception as e:
    print(type(e).__name__)

try:
    C1()
except Exception as e:
    print(type(e).__name__)
```

CPython output:	MicroPython output:
`C` `C1`	`TypeError` `C1`

3.8.3 Functions

Error messages for methods may display unexpected argument counts

Cause: MicroPython counts "self" as an argument.

Workaround: Interpret error messages with the information above in mind.

Sample code:

```
try:
    [].append()
except Exception as e:
    print(e)
```

CPython output:	MicroPython output:
`list.append() takes exactly one argument (0 given)`	`function takes 2 positional arguments but 1 were given`

Function objects do not have the `__module__` attribute

Cause: MicroPython is optimized for reduced code size and RAM usage.

Workaround: Use `sys.modules[function.__globals__['__name__']]` for non-builtin modules.

Sample code:

```
def f():
    pass


print(f.__module__)
```

CPython output:	MicroPython output:

`__main__`	`Traceback (most recent call last):`
	`  File "<stdin>", line 13, in <module>`
	`AttributeError: 'function' object has no attribute '__module__'`

`*args` cannot follow a keyword argument

Cause: MicroPython is optimised for code space. For more information see this issue.

Workaround: Re-order the arguments

Sample code:

```
def f(x, y):
    return x + y


try:
    print(f(y=1, *(3,)))
except Exception as e:
    print(e)
```

CPython output:	MicroPython output:

`4`	`Traceback (most recent call last):`
	`  File "<stdin>", line 14, in <module>`
	`SyntaxError: * arg after kwarg`

User-defined attributes for functions are not supported

Cause: MicroPython is highly optimized for memory usage.

Workaround: Use external dictionary, e.g. `FUNC_X[f] = 0`.

Sample code:

```
def f():
    pass


f.x = 0
print(f.x)
```

CPython output:	MicroPython output:

`0`	`Traceback (most recent call last):`
	`  File "<stdin>", line 13, in <module>`
	`AttributeError: 'function' object has no attribute 'x'`

3.8.4 Generator

Context manager __exit__() not called in a generator which does not run to completion

Sample code:

```
class foo(object):
    def __enter__(self):
        print("Enter")

    def __exit__(self, *args):
        print("Exit")


def bar(x):
    with foo():
        while True:
            x += 1
            yield x


def func():
    g = bar(0)
    for _ in range(3):
        print(next(g))


func()
```

CPython output:	MicroPython output:

`Enter`
`1`
`2`
`3`
`Exit`
	`Enter`
	`1`
	`2`
	`3`

3.8.5 Runtime

Local variables aren't included in locals() result

Cause: MicroPython doesn't maintain symbolic local environment, it is optimized to an array of slots. Thus, local variables can't be accessed by a name.

Sample code:

```
def test():
    val = 2
    print(locals())


test()
```

CPython output:	MicroPython output:
`{'val': 2}`	`{'test': <function test at 0x7fc39a605260>, '__name__': '__main__', '__file__': '<stdin>'}`

Code running in eval() function doesn't have access to local variables

Cause: MicroPython doesn't maintain symbolic local environment, it is optimized to an array of slots. Thus, local variables can't be accessed by a name. Effectively, `eval(expr)` in MicroPython is equivalent to `eval(expr, globals(), globals())`.

Sample code:

```
val = 1


def test():
    val = 2
    print(val)
    eval("print(val)")


test()
```

CPython output:	MicroPython output:
`2` `2`	`2` `1`

3.8.6 f-strings

f-strings don't support concatenation with adjacent literals if the adjacent literals contain braces

Cause: MicroPython is optimised for code space.

Workaround: Use the + operator between literal strings when they are not both f-strings

Sample code:

```
x, y = 1, 2
print("aa" f"{x}")  # works
print(f"{x}" "ab")  # works
print("a{}a" f"{x}")  # fails
print(f"{x}" "a{}b")  # fails
```

CPython output:	MicroPython output:

`aa1`
`1ab`
`a{}a1`
`1a{}b`
	`aa1`
	`1ab`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 12, in <module>`
	`IndexError: tuple index out of range`

f-strings cannot support expressions that require parsing to resolve unbalanced nested braces and brackets

Cause: MicroPython is optimised for code space.

Workaround: Always use balanced braces and brackets in expressions inside f-strings

Sample code:

```
print(f"{'hello { world'}")
print(f"{'hello ] world'}")
```

CPython output:	MicroPython output:

`hello { world`
`hello ] world`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 9`
	`SyntaxError: invalid syntax`

f-strings don't support !a conversions

Cause: MicropPython does not implement ascii()

Workaround: None

Sample code:

```
f"{'unicode text'!a}"
```

CPython output:	MicroPython output:

	`Traceback (most recent call last):`
	`  File "<stdin>", line 8`
	`SyntaxError: invalid syntax`

3.8.7 import

__path__ attribute of a package has a different type (single string instead of list of strings) in MicroPython

Cause: MicroPython doesn't support namespace packages split across filesystem. Beyond that, MicroPython's import system is highly optimized for minimal memory usage.

Workaround: Details of import handling is inherently implementation dependent. Don't rely on such details in portable applications.

Sample code:

```
import modules

print(modules.__path__)
```

CPython output:	MicroPython output:
`['/home/micropython/micropython-autodocs/` `↪tests/cpydiff/modules']`	`../tests/cpydiff/modules`

MicroPython doesn't support namespace packages split across filesystem.

Cause: MicroPython's import system is highly optimized for simplicity, minimal memory usage, and minimal filesystem search overhead.

Workaround: Don't install modules belonging to the same namespace package in different directories. For MicroPython, it's recommended to have at most 3-component module search paths: for your current application, per-user (writable), system-wide (non-writable).

Sample code:

```
import sys

sys.path.append(sys.path[1] + "/modules")
sys.path.append(sys.path[1] + "/modules2")

import subpkg.foo
import subpkg.bar

print("Two modules of a split namespace package imported")
```

CPython output:	MicroPython output:

`Two modules of a split namespace package imported`	`Traceback (most recent call last):`
	`  File "<stdin>", line 14, in <module>`
	`ImportError: no module named 'subpkg.bar'`

3.9 Builtin types

Generated Tue 29 Sep 2026 23:16:29 UTC

3.9.1 Exception

All exceptions have readable `value` and `errno` attributes, not just `StopIteration` and `OSError`.

Cause: MicroPython is optimised to reduce code size.

Workaround: Only use `value` on `StopIteration` exceptions, and `errno` on `OSError` exceptions. Do not use or rely on these attributes on other exceptions.

Sample code:

```
e = Exception(1)
print(e.value)
print(e.errno)
```

CPython output:	MicroPython output:

`Traceback (most recent call last):`
`  File "<stdin>", line 9, in <module>`
`AttributeError: 'Exception' object has no attribute 'value'`
	`1`
	`1`

Exception chaining not implemented

Sample code:

```
try:
    raise TypeError
except TypeError:
    raise ValueError
```

CPython output:	MicroPython output:

`Traceback (most recent call last):`
`  File "<stdin>", line 9, in <module>`
`TypeError`
 	
`During handling of the above exception, another exception occurred:`
 	
`Traceback (most recent call last):`
`  File "<stdin>", line 11, in <module>`
`ValueError`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 11, in <module>`
	`ValueError:`

User-defined attributes for builtin exceptions are not supported

Cause: MicroPython is highly optimized for memory usage.

Workaround: Use user-defined exception subclasses.

Sample code:

```
e = Exception()
e.x = 0
print(e.x)
```

CPython output:	MicroPython output:

`0`	`Traceback (most recent call last):`
	`  File "<stdin>", line 9, in <module>`
	`AttributeError: 'Exception' object has no attribute 'x'`

Exception in while loop condition may have unexpected line number

Cause: Condition checks are optimized to happen at the end of loop body, and that line number is reported.

Sample code:

```
l = ["-foo", "-bar"]

i = 0
while l[i][0] == "-":
```

(continues on next page)

(continued from previous page)

```
    print("iter")
    i += 1
```

CPython output:	MicroPython output:

`iter`
`iter`
`Traceback (most recent call last):`
`  File "<stdin>", line 11, in <module>`
`IndexError: list index out of range`
	`iter`
	`iter`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 13, in <module>`
	`IndexError: list index out of range`

Exception.__init__ method does not exist.

Cause: Subclassing native classes is not fully supported in MicroPython.

Workaround: Call using `super()` instead:

```
class A(Exception):
    def __init__(self):
        super().__init__()
```

Sample code:

```
class A(Exception):
    def __init__(self):
        Exception.__init__(self)


a = A()
```

CPython output:	MicroPython output:

	`Traceback (most recent call last):`
	`  File "<stdin>", line 18, in <module>`
	`  File "<stdin>", line 15, in __init__`
	`AttributeError: type object 'Exception' has no attribute '__init__'`

3.9.2 OSError

OSError constructor returns a plain OSError for all errno values, rather than a relevant subtype.

Cause: MicroPython does not include the CPython-standard OSError subclasses.

Workaround: Catch OSError and use its errno attribute to discriminate the cause.

Sample code:

```
import errno

errno_list = [  # i.e. the set implemented by micropython
    errno.EPERM,
    errno.ENOENT,
    errno.EIO,
    errno.EBADF,
    errno.EAGAIN,
    errno.ENOMEM,
    errno.EACCES,
    errno.EEXIST,
    errno.ENODEV,
    errno.EISDIR,
    errno.EINVAL,
    errno.EOPNOTSUPP,
    errno.EADDRINUSE,
    errno.ECONNABORTED,
    errno.ECONNRESET,
    errno.ENOBUFS,
    errno.ENOTCONN,
    errno.ETIMEDOUT,
    errno.ECONNREFUSED,
    errno.EHOSTUNREACH,
    errno.EALREADY,
    errno.EINPROGRESS,
]


def errno_output_type(n):
    try:
        raise OSError(n, "")
    except OSError as e:
        return f"{type(e).__name__}"
    except Exception as e:
        return f"non-OSError {type(e).__name__}"
    else:
        return "no error"


for n in errno_list:
    print(errno.errorcode[n], "=", errno_output_type(n))
```

CPython output:

```
EPERM = PermissionError
ENOENT = FileNotFoundError
EIO = OSError
EBADF = OSError
EAGAIN = BlockingIOError
ENOMEM = OSError
EACCES = PermissionError
EEXIST = FileExistsError
ENODEV = OSError
EISDIR = IsADirectoryError
EINVAL = OSError
ENOTSUP = OSError
EADDRINUSE = OSError
ECONNABORTED = ConnectionAbortedError
ECONNRESET = ConnectionResetError
ENOBUFS = OSError
ENOTCONN = OSError
ETIMEDOUT = TimeoutError
ECONNREFUSED = ConnectionRefusedError
EHOSTUNREACH = OSError
EALREADY = BlockingIOError
EINPROGRESS = BlockingIOError
```

MicroPython output:

```
EPERM = OSError
ENOENT = OSError
EIO = OSError
EBADF = OSError
EAGAIN = OSError
ENOMEM = OSError
EACCES = OSError
EEXIST = OSError
ENODEV = OSError
EISDIR = OSError
EINVAL = OSError
EOPNOTSUPP = OSError
EADDRINUSE = OSError
ECONNABORTED = OSError
ECONNRESET = OSError
ENOBUFS = OSError
ENOTCONN = OSError
ETIMEDOUT = OSError
ECONNREFUSED = OSError
EHOSTUNREACH = OSError
EALREADY = OSError
EINPROGRESS = OSError
```

3.9.3 bytearray

Array slice assignment with unsupported RHS

Sample code:

```
b = bytearray(4)
b[0:1] = [1, 2]
print(b)
```

CPython output:

```
bytearray(b'\x01\x02\x00\x00\x00')
```

MicroPython output:

```
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
NotImplementedError: array/bytes␣
↪required on right side
```

3.9.4 bytes

bytes.decode() only supports encoding arguments 'utf8', 'utf-8' and 'ascii'. Other encodings like 'latin-1' are not supported. Other string forms such as 'UTF8' are not supported.

Cause: MicroPython is optimised for embedded systems and only includes UTF-8 and ASCII codec support with simple matching to save memory and code size. *The same restriction applies to str constructor*. See also *Unicode Support*.

Workaround: Convert data to UTF-8 before processing, or implement custom encoding/decoding if needed. Ensure encoding argument is one of the accepted forms.

Sample code:

```
# Both CPython and MicroPython support utf8 and ascii encodings
print(b"caf\xc3\xa9".decode("utf8"))  # codespell:ignore caf
print(b"cafe".decode("ascii"))

# MicroPython does not support additional encodings
try:
    b"\xe9".decode("latin-1")  # 'é' in latin-1
    print("latin-1 supported")
except (ValueError, NotImplementedError, LookupError) as e:
    print("latin-1 not supported:", type(e).__name__)

try:
    b"\x80".decode("cp1252")  # Euro sign in cp1252
    print("cp1252 supported")
except (ValueError, NotImplementedError, LookupError) as e:
    print("cp1252 not supported:", type(e).__name__)

# Encoding arguments must match exactly in MicroPython
try:
    b"hello".decode("ASCII")
    print("Capital letters ASCII supported")
except (ValueError, NotImplementedError, LookupError) as e:
    print("Capital letters ASCII not supported:", type(e).__name__)
```

CPython output:

```
café
cafe
latin-1 supported
cp1252 supported
Capital letters ASCII supported
```

MicroPython output:

```
café
cafe
latin-1 not supported: LookupError
cp1252 not supported: LookupError
Capital letters ASCII not supported:␣
↪LookupError
```

bytes.decode() does not accept keyword arguments, only positional arguments

Cause: MicroPython optimizes for code size and does not implement keyword argument handling for bytes.decode()

Workaround: Use positional arguments instead of keyword arguments

Sample code:

```
# CPython accepts keyword arguments, MicroPython only accepts positional
b = b"hello\xffworld"

try:
    # Using keyword arguments
    result = b.decode(encoding="utf-8", errors="ignore")
    print("kwargs supported:", repr(result))
except TypeError as e:
    print("kwargs not supported: TypeError")
    # Workaround: use positional arguments
    result = b.decode("utf-8", "ignore")
    print("positional args work:", repr(result))
```

CPython output:	MicroPython output:
`kwargs supported: 'helloworld'`	`kwargs not supported: TypeError` `positional args work: 'helloworld'`

bytes() constructor only supports encoding arguments 'utf8', 'utf-8' and 'ascii'. Other encodings like 'latin-1' are not supported. Other string forms such as 'UTF8' are not supported.

Cause: MicroPython is optimised for embedded systems and has limited codec support to save code size. *A similar restriction applies to str.encode()*. See also *Unicode Support*.

Workaround: Implement other encoding conversions manually by parsing the result of str.encode() or bytes() constructor.

Sample code:

```
# Both CPython and MicroPython can encode this emoji as UTF-8 bytes
print(bytes("", "utf8"))

# Both CPython and MicroPython will fail to encode this emoji as ASCII bytes
try:
    print(bytes("", "ascii"))
except UnicodeError:
    print("UnicodeError")

# Other encodings or string formats aren't accepted by MicroPython
try:
    print(bytes("", "UTF-8"))
except LookupError:
    print("LookupError")
```

CPython output:	MicroPython output:

`b'\xf0\x9f\x98\x80'`
`UnicodeError`
`b'\xf0\x9f\x98\x80'`
	`b'\xf0\x9f\x98\x80'`
	`UnicodeError`
	`LookupError`

bytes objects support .format() method

Cause: MicroPython strives to be a more regular implementation, so if both *str* and *bytes* support `__mod__()` (the % operator), it makes sense to support `format()` for both too. Support for `__mod__` can also be compiled out, which leaves only `format()` for bytes formatting.

Workaround: If you are interested in CPython compatibility, don't use `.format()` on bytes objects.

Sample code:

```
print(b"{}".format(1))
```

CPython output:	MicroPython output:

`Traceback (most recent call last):`
`  File "<stdin>", line 8, in <module>`
`AttributeError: 'bytes' object has no attribute 'format'`
	`b'1'`

bytes() with keywords not implemented

Workaround: Pass the encoding as a positional parameter, e.g. `print(bytes('abc', 'utf-8'))`

Sample code:

```
print(bytes("abc", encoding="utf8"))
```

CPython output:	MicroPython output:

`b'abc'`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError: keyword argument(s) not implemented - use normal args instead`

Bytes subscription with step != 1 not implemented

Cause: MicroPython is highly optimized for memory usage.

Workaround: Use explicit loop for this very rare operation.

Sample code:

```
print(b"123"[0:3:2])
```

CPython output:	MicroPython output:

`b'13'`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError: only slices with step=1 (aka None) are supported`

3.9.5 complex

MicroPython's complex() accepts certain incorrect values that CPython rejects

Cause: MicroPython is highly optimized for memory usage.

Workaround: Do not use non-standard complex literals as argument to complex()

MicroPython's `complex()` function accepts literals that contain a space and no sign between the real and imaginary parts, and interprets it as a plus.

Sample code:

```
try:
    print(complex("1 1j"))
except ValueError:
    print("ValueError")
```

CPython output:	MicroPython output:
`ValueError`	`(1+1j)`

3.9.6 dict

Dictionary keys view does not behave as a set.

Cause: Not implemented.

Workaround: Explicitly convert keys to a set before using set operations.

Sample code:

```
print({1: 2, 3: 4}.keys() & {1})
```

CPython output:	MicroPython output:

`{1}`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`TypeError: unsupported types for __and__`
	`↪: 'dict_view', 'set'`

3.9.7 float

MicroPython allows implicit conversion of objects in maths operations while CPython does not.

Workaround: Objects should be wrapped in `float(obj)` for compatibility with CPython.

Sample code:

```
class Test:
    def __float__(self):
        return 0.5


print(2.0 * Test())
```

CPython output:	MicroPython output:

`Traceback (most recent call last):`
`  File "<stdin>", line 14, in <module>`
`TypeError: unsupported operand type(s)␣`
`↪for *: 'float' and 'Test'`
	`1.0`

3.9.8 int

`bit_length` method doesn't exist.

Cause: bit_length method is not implemented.

Workaround: Avoid using this method on MicroPython.

Sample code:

```
x = 255
print("{} is {} bits long.".format(x, x.bit_length()))
```

CPython output:	MicroPython output:

`255 is 8 bits long.`	`Traceback (most recent call last):`
	`  File "<stdin>", line 9, in <module>`
	`AttributeError: 'int' object has no␣`
	`↪attribute 'bit_length'`

No int conversion for int-derived types available

Workaround: Avoid subclassing builtin types unless really needed. Prefer https://en.wikipedia.org/wiki/Composition_over_inheritance .

Sample code:

```
class A(int):
    __add__ = lambda self, other: A(int(self) + other)


a = A(42)
print(a + a)
```

CPython output:	MicroPython output:

`84`	`Traceback (most recent call last):`
	`  File "<stdin>", line 14, in <module>`
	`  File "<stdin>", line 10, in <lambda>`
	`TypeError: unsupported types for __radd__`
	`↪: 'int', 'int'`

3.9.9 list

List delete with step != 1 not implemented

Workaround: Use explicit loop for this rare operation.

Sample code:

```
l = [1, 2, 3, 4]
del l[0:4:2]
print(l)
```

CPython output:	MicroPython output:

`[2, 4]`	`Traceback (most recent call last):`
	`  File "<stdin>", line 9, in <module>`
	`NotImplementedError:`

List slice-store with non-iterable on RHS is not implemented

Cause: RHS is restricted to be a tuple or list

Workaround: Use `list(<iter>)` on RHS to convert the iterable to a list

Sample code:

```
l = [10, 20]
l[0:1] = range(4)
print(l)
```

CPython output:	MicroPython output:

`[0, 1, 2, 3, 20]`	`Traceback (most recent call last):`
	`  File "<stdin>", line 9, in <module>`
	`TypeError: object 'range' isn't a tuple or list`

List store with step != 1 not implemented

Workaround: Use explicit loop for this rare operation.

Sample code:

```
l = [1, 2, 3, 4]
l[0:4:2] = [5, 6]
print(l)
```

CPython output:	MicroPython output:

`[5, 2, 6, 4]`	`Traceback (most recent call last):`
	`  File "<stdin>", line 9, in <module>`
	`NotImplementedError:`

3.9.10 memoryview

memoryview can become invalid if its target is resized

Cause: CPython prevents a `bytearray` or `io.bytesIO` object from changing size while there is a `memoryview` object that references it. MicroPython requires the programmer to manually ensure that an object is not resized while any `memoryview` references it.

In the worst case scenario, resizing an object which is the target of a memoryview can cause the memoryview(s) to reference invalid freed memory (a use-after-free bug) and corrupt the MicroPython runtime.

Workaround: Do not change the size of any `bytearray` or `io.bytesIO` object that has a `memoryview` assigned to it.

Sample code:

```
b = bytearray(b"abcdefg")
m = memoryview(b)
b.extend(b"hijklmnop")
print(b, bytes(m))
```

CPython output:	MicroPython output:

`Traceback (most recent call last):`
`  File "<stdin>", line 12, in <module>`
`BufferError: Existing exports of data: object cannot be re-sized`
	`bytearray(b'abcdefghijklmnop') b'abcdefg'`

3.9.11 range

Range arguments must fit in a machine word; large start or stop values misbehave.

Cause: Range stores its arguments as the C mp_int_t type, and intermediate calculations also use it.

Workaround: Avoid using such ranges

Sample code:

```
from sys import maxsize

# A range including `maxsize-1` cannot be created
try:
    print(range(-maxsize - 1, 0))
except OverflowError:
    print("OverflowError")

# A range with start or stop outside [-maxsize, maxsize] cannot be created, even if the range itself would be small.
try:
    print(range(maxsize + 1, maxsize + 2))
except OverflowError:
    print("OverflowError")

# A range with `stop-start` exceeding sys.maxsize has incorrect len(), while CPython cannot calculate len().
try:
    print(len(range(-maxsize, maxsize)))
except OverflowError:
    print("OverflowError")

# A range with `stop-start` exceeding sys.maxsize has incorrect len()
try:
    print(len(range(-maxsize, maxsize, maxsize)))
except OverflowError:
    print("OverflowError")
```

CPython output:	MicroPython output:

`range(-9223372036854775808, 0)`
`range(9223372036854775808, 9223372036854775809)`
`OverflowError`
`2`
	`OverflowError`
	`OverflowError`
	`0`
	`0`

3.9.12 str

str.encode() constructor only supports encoding arguments 'utf8', 'utf-8' and 'ascii'. Other encodings like 'latin-1' are not supported. Other string forms such as 'UTF8' are not supported.

Cause: MicroPython is optimised for embedded systems and has limited codec support to save code size. *A similar restriction applies to bytes constructor*. See also *Unicode Support*.

Workaround: Implement encoding conversions manually by parsing the result of str.encode() or bytes() constructor.

Sample code:

```
# Both CPython and MicroPython can encode this emoji as UTF-8 bytes
print("🌀".encode("utf8"))

# Both CPython and MicroPython will fail to encode this emoji as ASCII bytes
try:
    print("🌀".encode("ascii"))
except UnicodeError:
    print("UnicodeError")

# Other encodings or string formats aren't accepted by MicroPython:
try:
    print("🌀".encode("UTF-8"))
except LookupError:
    print("LookupError")
```

CPython output:	MicroPython output:

`b'\xf0\x9f\x98\x80'`
`UnicodeError`
`b'\xf0\x9f\x98\x80'`
	`b'\xf0\x9f\x98\x80'`
	`UnicodeError`
	`LookupError`

str.encode() and bytes() constructor ignore any `errors` argument specified. If the encoding is specified as `'ascii'` and a non-ASCII byte is found in the string then an exception is always raised.

Cause: MicroPython is optimised for embedded systems and has limited codec support to save code size. See also *Unicode Support*.

Workaround: Handle encoding errors manually by parsing the result of str.encode() or bytes() constructor.

Sample code:

```
# CPython will replace the emoji with an ASCII '?' but MicroPython will
# raise an exception
try:
    print("🌀".encode("ascii", "replace"))
except UnicodeError:
    print("UnicodeError")

# CPython will ignore the emoji in the result but MicroPython will raise an exception
try:
    print("🌀".encode("ascii", "ignore"))
except UnicodeError:
    print("UnicodeError")
```

CPython output:	MicroPython output:
`b'?'` `b''`	`UnicodeError` `UnicodeError`

str() constructor only supports encoding arguments 'utf8', 'utf-8' and 'ascii'. Other encodings like 'latin-1' are not supported. Other string forms such as 'UTF8' are not supported.

Cause: MicroPython is optimised for embedded systems and only includes UTF-8 and ASCII codec support with simple matching to save memory and code size. *The same restriction applies to bytes.decode()*. See also *Unicode Support*.

Workaround: Convert data to UTF-8 before processing, or implement custom encoding/decoding if needed. Ensure encoding argument is one of the accepted forms.

Sample code:

```
# Both CPython and MicroPython support utf8 and ascii encodings
print(str(b"caf\xc3\xa9", "utf8"))  # codespell:ignore caf
print(str(b"cafe", "ascii"))

# MicroPython does not support additional encodings
try:
    str(b"\xe9", "latin-1")  # 'é' in latin-1
    print("latin-1 supported")
except (ValueError, NotImplementedError, LookupError) as e:
    print("latin-1 not supported:", type(e).__name__)

try:
    str(b"\x80", "cp1252")  # Euro sign in cp1252
    print("cp1252 supported")
except (ValueError, NotImplementedError, LookupError) as e:
    print("cp1252 not supported:", type(e).__name__)

# Encoding arguments must match exactly in MicroPython
try:
    str(b"hello", "ASCII")
    print("Capital letters ASCII supported")
except (ValueError, NotImplementedError, LookupError) as e:
    print("Capital letters ASCII not supported:", type(e).__name__)
```

CPython output:	MicroPython output:

`café`
`cafe`
`latin-1 supported`
`cp1252 supported`
`Capital letters ASCII supported`
	`café`
	`cafe`
	`latin-1 not supported: LookupError`
	`cp1252 not supported: LookupError`
	`Capital letters ASCII not supported:`
	`↪LookupError`

MicroPython accepts the "," grouping option with any radix, unlike CPython

Cause: To reduce code size, MicroPython does not issue an error for this combination

Workaround: Do not use a format string like `{:,b}` if CPython compatibility is required.

Sample code:

```
try:
    print("{:,b}".format(99))
except ValueError:
    print("ValueError")
try:
    print("{:,x}".format(99))
except ValueError:
    print("ValueError")
try:
    print("{:,o}".format(99))
except ValueError:
    print("ValueError")
```

CPython output:	MicroPython output:

`ValueError`
`ValueError`
`ValueError`
	`110,0011`
	`63`
	`143`

MicroPython accepts but does not properly implement the "," or "_" grouping character for float values

Cause: To reduce code size, MicroPython does not implement this combination. Grouping characters will not appear in the number's significant digits and will appear at incorrect locations in leading zeros.

Workaround: Do not use a format string like `{:,f}` if exact CPython compatibility is required.

Sample code:

```
print("{:,f}".format(3141.159))
print("{:_f}".format(3141.159))
print("{:011,.2f}".format(3141.159))
print("{:011_.2f}".format(3141.159))
```

CPython output:	MicroPython output:

`3,141.159000`
`3_141.159000`
`0,003,141.16`
`0_003_141.16`
	`3141.159000`
	`3141.159000`
	`000,3141.16`
	`0_003141.16`

Attributes/subscr not implemented

Sample code:

```
print("{a[0]}".format(a=[1, 2]))
```

CPython output:	MicroPython output:

`1`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError: attributes not supported`

str(...) with keywords not implemented

Workaround: Input the encoding format directly. eg `print(bytes('abc', 'utf-8'))`

Sample code:

```
print(str(b"abc", encoding="utf8"))
```

CPython output:	MicroPython output:

`abc`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError: keyword argument(s) not implemented - use normal args instead`

str.ljust() and str.rjust() not implemented

Cause: MicroPython is highly optimized for memory usage. Easy workarounds available.

Workaround: Instead of `s.ljust(10)` use `"%-10s" % s`, instead of `s.rjust(10)` use `"% 10s" % s`. Alternatively, `"{:<10}".format(s)` or `"{:>10}".format(s)`.

Sample code:

```
print("abc".ljust(10))
```

CPython output:	MicroPython output:

`abc`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`AttributeError: 'str' object has no attribute 'ljust'`

repr() may print some non-printable Unicode characters literally instead of as escape sequences

Cause: MicroPython uses a simplified heuristic to determine printable characters, avoiding the need for a full Unicode character database (saves memory). It prints characters >= U+0080 (excluding surrogates) as UTF-8. CPython uses the Unicode database to identify non-printable characters like noncharacters (U+FFFx in each plane).

Workaround: Accept the difference for embedded use cases, or use ascii() or manual escaping if exact control is needed.

Sample code:

```
# These are noncharacters that CPython escapes but MicroPython prints
# showing as hex to avoid display issues in documentation tables
print("U+FFFF:", repr("\uffff").encode("utf-8").hex())
print("U+1FFFF:", repr("\U0001ffff").encode("utf-8").hex())
```

CPython output:	MicroPython output:
`U+FFFF: 275c756666666627` `U+1FFFF: 275c55303030316666666627`	`U+FFFF: 27efbfbf27` `U+1FFFF: 27f09fbfbf27`

None as first argument for rsplit such as str.rsplit(None, n) not implemented

Sample code:

```
print("a a a".rsplit(None, 1))
```

CPython output:	MicroPython output:

`['a a', 'a']`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError: rsplit(None,n)`

Subscript with step != 1 is not yet implemented

Sample code:

```
print("abcdefghi"[0:9:2])
```

CPython output:	MicroPython output:

`acegi`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError: only slices with step=1 (aka None) are supported`

3.9.13 tuple

Tuple load with step != 1 not implemented

Sample code:

```
print((1, 2, 3, 4)[0:4:2])
```

CPython output:	MicroPython output:

`(1, 3)`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError: only slices with step=1 (aka None) are supported`

3.10 Modules

Generated Tue 29 Sep 2026 23:16:29 UTC

3.10.1 Positional-only Parameters

To save code size, many functions that accept keyword arguments in CPython only accept positional arguments in MicroPython.

MicroPython marks positional-only parameters in the same way as CPython, by inserting a `/` to mark the end of the positional parameters. Any function whose signature ends in `/` takes *only* positional arguments. For more details, see PEP 570.

Example

For example, in CPython 3.4 this is the signature of the constructor `socket.socket`:

```
socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)
```

However, the signature documented in *MicroPython* is:

```
socket(af=AF_INET, type=SOCK_STREAM, proto=IPPROTO_TCP, /)
```

The `/` at the end of the parameters indicates that they are all positional-only in MicroPython. The following code works in CPython but not in most MicroPython ports:

```
import socket
s = socket.socket(type=socket.SOCK_DGRAM)
```

MicroPython will raise an exception:

```
TypeError: function doesn't take keyword arguments
```

The following code will work in both CPython and MicroPython:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

3.10.2 array

Comparison between different typecodes not supported

Cause: Code size

Workaround: Compare individual elements

Sample code:

```
import array

array.array("b", [1, 2]) == array.array("i", [1, 2])
```

CPython output:	MicroPython output:

	`Traceback (most recent call last):`
	`  File "<stdin>", line 10, in <module>`
	`NotImplementedError:`

Overflow checking is not implemented

Cause: MicroPython implements implicit truncation in order to reduce code size and execution time

Workaround: If CPython compatibility is needed then mask the value explicitly

Sample code:

```
import array

a = array.array("b", [257])
print(a)
```

CPython output:	MicroPython output:

`Traceback (most recent call last):`
`  File "<stdin>", line 10, in <module>`
`OverflowError: signed char is greater`
`↪than maximum`
	`array('b', [1])`

Looking for integer not implemented

Sample code:

```
import array

print(1 in array.array("B", b"12"))
```

CPython output:	MicroPython output:

`False`	`Traceback (most recent call last):`
	`  File "<stdin>", line 10, in <module>`
	`NotImplementedError:`

Array deletion not implemented

Sample code:

```
import array

a = array.array("b", (1, 2, 3))
del a[1]
print(a)
```

CPython output:	MicroPython output:

`array('b', [1, 3])`	`Traceback (most recent call last):`
	`  File "<stdin>", line 11, in <module>`
	`TypeError: 'array' object doesn't␣`
	`↪support item deletion`

Subscript with step != 1 is not yet implemented

Sample code:

```
import array

a = array.array("b", (1, 2, 3))
print(a[3:2:2])
```

CPython output:	MicroPython output:

`array('b')`	`Traceback (most recent call last):`
	`  File "<stdin>", line 11, in <module>`
	`NotImplementedError: only slices with␣`
	`↪step=1 (aka None) are supported`

3.10.3 errno

MicroPython does not include `ENOTSUP` as a name for errno 95.

Cause: MicroPython does not implement the `ENOTSUP` canonical alias for `EOPNOTSUPP` added in CPython 3.2.

Workaround: Use `errno.EOPNOTSUPP` in place of `errno.ENOTSUP`.

Sample code:

```
import errno

print(f"{errno.errorcode[errno.EOPNOTSUPP]=!s}")
```

CPython output:	MicroPython output:
`errno.errorcode[errno.EOPNOTSUPP]=ENOTSUP`	`errno.errorcode[errno.` `↪EOPNOTSUPP]=EOPNOTSUPP`

3.10.4 json

JSON module does not throw exception when object is not serialisable

Sample code:

```
import json

try:
    print(json.dumps(b"shouldn't be able to serialise bytes"))
except TypeError:
    print("TypeError")
```

CPython output:	MicroPython output:
`TypeError`	`"shouldn't be able to serialise bytes"`

3.10.5 os

`environ` attribute is not implemented

Workaround: Use `getenv`, `putenv` and `unsetenv`

Sample code:

```
import os

try:
    print(os.environ.get("NEW_VARIABLE"))
    os.environ["NEW_VARIABLE"] = "VALUE"
```

(continues on next page)

(continued from previous page)

```
    print(os.environ["NEW_VARIABLE"])
except AttributeError:
    print("should not get here")
    print(os.getenv("NEW_VARIABLE"))
    os.putenv("NEW_VARIABLE", "VALUE")
    print(os.getenv("NEW_VARIABLE"))
```

CPython output:	MicroPython output:

`None`
`VALUE`
	`should not get here`
	`None`
	`VALUE`

`getenv` returns actual value instead of cached value

Cause: The `environ` attribute is not implemented

Sample code:

```
import os

print(os.getenv("NEW_VARIABLE"))
os.putenv("NEW_VARIABLE", "VALUE")
print(os.getenv("NEW_VARIABLE"))
```

CPython output:	MicroPython output:
`None` `None`	`None` `VALUE`

3.10.6 random

`getrandbits` method can only return a maximum of 32 bits at a time.

Cause: PRNG's internal state is only 32bits so it can only return a maximum of 32 bits of data at a time.

Workaround: If you need a number that has more than 32 bits then utilize the random module from micropython-lib.

Sample code:

```
import random


x = random.getrandbits(64)
print("{}".format(x))
```

CPython output:	MicroPython output:

`16069454902792869424`	`Traceback (most recent call last):`
	`  File "<stdin>", line 11, in <module>`
	`ValueError: bits must be 32 or less`

`randint` method can only return an integer that is at most the native word size.

Cause: PRNG is only able to generate 32 bits of state at a time. The result is then cast into a native sized int instead of a full int object.

Workaround: If you need integers larger than native wordsize use the random module from micropython-lib.

Sample code:

```
import random


x = random.randint(2**128 - 1, 2**128)
print("x={}".format(x))
```

CPython output:	MicroPython output:

`x=340282366920938463463374607431768211455`	`Traceback (most recent call last):`
	`  File "<stdin>", line 11, in <module>`
	`OverflowError: overflow converting long int to machine word`

3.10.7 struct

Struct pack with too few args, not checked by MicroPython

Sample code:

```
import struct

try:
    print(struct.pack("bb", 1))
    print("Should not get here")
except:
    print("struct.error")
```

CPython output:	MicroPython output:
`struct.error`	`b'\x01\x00'` `Should not get here`

Struct pack with too many args, not checked by MicroPython

Sample code:

```
import struct

try:
    print(struct.pack("bb", 1, 2, 3))
    print("Should not get here")
except:
    print("struct.error")
```

CPython output:	MicroPython output:
`struct.error`	`b'\x01\x02'` `Should not get here`

Struct pack with whitespace in format, whitespace ignored by CPython, error on MicroPython

Cause: MicroPython is optimised for code size.

Workaround: Don't use spaces in format strings.

Sample code:

```
import struct

try:
    print(struct.pack("b b", 1, 2))
    print("Should have worked")
except:
    print("struct.error")
```

CPython output:	MicroPython output:
`b'\x01\x02'` `Should have worked`	`struct.error`

3.10.8 sys

Overriding sys.stdin, sys.stdout and sys.stderr not possible

Cause: They are stored in read-only memory.

Sample code:

```
import sys

sys.stdin = None
print(sys.stdin)
```

CPython output:	MicroPython output:

`None`	`Traceback (most recent call last):`
	`  File "<stdin>", line 10, in <module>`
	`AttributeError: 'module' object has no attribute 'stdin'`

CHAPTER

FOUR

MICROPYTHON INTERNALS

This chapter covers a tour of MicroPython from the perspective of a developer, contributing to MicroPython. It acts as a comprehensive resource on the implementation details of MicroPython for both novice and expert contributors.

Development around MicroPython usually involves modifying the core runtime, porting or maintaining a new library. This guide describes at great depth, the implementation details of MicroPython including a getting started guide, compiler internals, porting MicroPython to a new platform and implementing a core MicroPython library.

4.1 Getting Started

This guide covers a step-by-step process on setting up version control, obtaining and building a copy of the source code for a port, building the documentation, running tests, and a description of the directory structure of the MicroPython code base.

4.1.1 Source control with git

MicroPython is hosted on GitHub and uses Git for source control. The workflow is such that code is pulled and pushed to and from the main repository. Install the respective version of Git for your operating system to follow through the rest of the steps.

Note: For a reference on the installation instructions, please refer to the Git installation instructions. Learn about the basic git commands in this Git Handbook or any other sources on the internet.

Note: A .git-blame-ignore-revs file is included which avoids the output of git blame getting cluttered by commits which are only for formatting code but have no functional changes. See git blame documentation on how to use this.

4.1.2 Get the code

It is recommended that you maintain a fork of the MicroPython repository for your development purposes. The process of obtaining the source code includes the following:

1. Fork the repository https://github.com/micropython/micropython
2. You will now have a fork at <https://github.com/<your-user-name>/micropython>.
3. Clone the forked repository using the following command:

```
$ git clone https://github.com/<your-user-name>/micropython
```

Then, configure the remote repositories to be able to collaborate on the MicroPython project.

Configure remote upstream:

```
$ cd micropython
$ git remote add upstream https://github.com/micropython/micropython
```

It is common to configure `upstream` and `origin` on a forked repository to assist with sharing code changes. You can maintain your own mapping but it is recommended that `origin` maps to your fork and `upstream` to the main MicroPython repository.

After the above configuration, your setup should be similar to this:

```
$ git remote -v
origin       https://github.com/<your-user-name>/micropython (fetch)
origin       https://github.com/<your-user-name>/micropython (push)
upstream     https://github.com/micropython/micropython (fetch)
upstream     https://github.com/micropython/micropython (push)
```

You should now have a copy of the source code. By default, you are pointing to the master branch. To prepare for further development, it is recommended to work on a development branch.

```
$ git checkout -b dev-branch
```

You can give it any name. You will have to compile MicroPython whenever you change to a different branch.

4.1.3 Compile and build the code

When compiling MicroPython, you compile a specific *port*, usually targeting a specific board. Start by installing the required dependencies. Then build the MicroPython cross-compiler before you can successfully compile and build. This applies specifically when using Linux to compile. The Windows instructions are provided in a later section.

Required dependencies

Install the required dependencies for Linux:

```
$ sudo apt-get install build-essential libffi-dev git pkg-config
```

For the stm32 port, the ARM cross-compiler is required:

```
$ sudo apt-get install gcc-arm-none-eabi libnewlib-arm-none-eabi
```

See the ARM GCC toolchain for the latest details.

Python 3 is also required. Check that you have Python available on your system:

```
$ python3
Python 3.5.0 (default, Jul 17 2020, 14:04:10)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

All supported ports have different dependency requirements, see their respective readme files.

Building the MicroPython cross-compiler

Almost all ports require building `mpy-cross` first to perform pre-compilation of Python code that will be included in the port firmware:

```
$ cd mpy-cross
$ make
```

Note: Note that, `mpy-cross` must be built for the host architecture and not the target architecture.

If it built successfully, you should see a message similar to this:

```
LINK mpy-cross
   text          data     bss     dec     hex filename
 279328           776     880  280984   44998 mpy-cross
```

Note: Use `make -C mpy-cross` to build the cross-compiler in one statement without moving to the `mpy-cross` directory otherwise, you will need to do `cd ..` for the next steps.

Building the Unix port of MicroPython

The Unix port is a version of MicroPython that runs on Linux, macOS, and other Unix-like operating systems. It's extremely useful for developing MicroPython as it avoids having to deploy your code to a device to test it. In many ways, it works a lot like CPython's python binary.

To build for the Unix port, make sure all Linux related dependencies are installed as detailed in the required dependencies section. See the *Required dependencies* to make sure that all dependencies are installed for this port. Also, make sure you have a working environment for `gcc` and `GNU make`. Ubuntu 20.04 has been used for the example below but other unixes ought to work with little modification:

```
$ gcc --version
gcc (Ubuntu 9.3.0-10ubuntu2) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.then build:
```

```
$ cd ports/unix
$ make submodules
$ make
```

If MicroPython built correctly, you should see the following:

```
LINK micropython
   text         data     bss     dec     hex filename
 412033         5680    2496  420209   66971 micropython
```

Now run it:

```
$ ./micropython
MicroPython v1.13-38-gc67012d-dirty on 2020-09-13; linux version
Use Ctrl-D to exit, Ctrl-E for paste mode
>>> print("hello world")
```

(continues on next page)

(continued from previous page)

```
hello world
>>>
```

Building the Windows port

The Windows port includes a Visual Studio project file micropython.vcxproj that you can use to build micropython.exe. It can be opened in Visual Studio or built from the command line using msbuild. Alternatively, it can be built using mingw, either in Windows with Cygwin, or on Linux. See windows port documentation for more information.

Building the STM32 port

Like the Unix port, you need to install some required dependencies as detailed in the *Required dependencies* section, then build:

```
$ cd ports/stm32
$ make submodules
$ make
```

Please refer to the stm32 documentation for more details on flashing the firmware.

Note: See the *Required dependencies* to make sure that all dependencies are installed for this port. The cross-compiler is needed. `arm-none-eabi-gcc` should also be in the $PATH or specified manually via CROSS_COMPILE, either by setting the environment variable or in the `make` command line arguments.

You can also specify which board to use:

```
$ cd ports/stm32
$ make BOARD=<board> submodules
$ make BOARD=<board>
```

See ports/stm32/boards for the available boards. e.g. “PYBV11” or “NUCLEO_WB55”.

4.1.4 Building the documentation

MicroPython documentation is created using `Sphinx`. If you have already installed Python, then install `Sphinx` using `pip`. It is recommended that you use a virtual environment:

```
$ python3 -m venv env
$ source env/bin/activate
$ pip install -r docs/requirements.txt
```

Navigate to the `docs` directory:

```
$ cd docs
```

Build the docs:

```
$ make html
```

Open `docs/build/html/index.html` in your browser to view the docs locally. Refer to the documentation on importing your documentation to use Read the Docs.

4.1.5 Running the tests

To run all tests in the test suite on the Unix port use:

```
$ cd ports/unix
$ make test
```

To run a selection of tests on a board/device connected over USB use:

```
$ cd tests
$ ./run-tests.py -t /dev/ttyACM0
```

See also *Writing tests*.

4.1.6 Additional make targets for developers

In all `make`-based ports, there is a target to print the size of a specific object file. When a change is confined to a single file, this is useful when testing variations to find smaller alternatives.

For instance, to print the size of `objstr.o` in the `py/` directory when making a unix standard build:

```
$ make build-standard/py/objstr.sz
```

Similarly, there is a target to save the preprocessed version of a file:

```
$ make build-standard/py/objstr.pp
```

In `ports/unix` there are additional targets related to running tests:

```
$ make test//int       # Run all tests matching the pattern "int"
$ make test/ports/unix # Run all tests in ports/unix
$ make test-failures   # Re-run only the failed tests
$ make print-failures  # print the differences for failed tests
$ make clean-failures  # delete the .exp and .out files from failed tests
```

4.1.7 Using ci.sh locally

MicroPython uses GitHub Actions for continuous integration. To reduce dependence on any specific CI system, the actual build steps for Unix-based builds are in the file `tools/ci.sh`. This can also be used as a script on developer desktops, with caveats:

- For most steps, An Ubuntu/Debian system similar to the one used during CI is assumed.
- Some specific steps assume specific Ubuntu versions.
- The setup steps may invoke the system package manager to install packages, download and install software from the internet, etc.

To get a usage message including the list of commands, run:

```
$ tools/ci.sh --help
```

As an example, you can build and test the unix minimal port with:

```
$ tools/ci.sh unix_minimal_build unix_minimal_run_tests
```

If you use the bash shell, you can add a `ci` command with tab completion:

```
$ eval $(tools/ci.sh --bash-completion)
```

For the zsh shell, replace `--bash-completion` with `--zsh-completion`. For the fish shell, replace `--bash-completion` with `--fish-completion`.

Then, typing:

```
$ ci unix_cov<tab>
```

This will complete the ci step name to `unix_coverage_`. Pressing tab a second time will show the list of matching steps:

```
$ ci unix_coverage_<tab>
unix_coverage_32bit_build
unix_coverage_32bit_run_native_mpy_tests…
```

4.1.8 Folder structure

There are a couple of directories to take note of in terms of where certain implementation details are. The following is a break down of the top-level folders in the source code.

py

: Contains the compiler, runtime, and core library implementation.

mpy-cross

: Has the MicroPython cross-compiler which pre-compiles the Python scripts to bytecode.

ports

: Code for all the versions of MicroPython for the supported ports.

lib

: Low-level C libraries used by any port which are mostly 3rd-party libraries.

drivers

: Has drivers for specific hardware and intended to work across multiple ports.

extmod

: Contains a C implementation of more non-core modules.

docs

: Has the standard documentation found at https://docs.micropython.org/.

tests

: An implementation of the test suite.

tools

: Contains scripts used by the build and CI process, as well as user tools such as `pyboard.py` and `mpremote`.

examples

: Example code for building MicroPython as a library as well as native modules.

4.2 Writing tests

Tests in MicroPython are located at the path `tests/`. The following is a listing of key directories and the run-tests.py runner script:

```
.
 ├── basics
 ├── extmod
 ├── float
 ├── micropython
 ├── run-tests.py
 ...
```

There are subfolders maintained to categorize the tests. Add a test by creating a new file in one of the existing folders or in a new folder. It's also possible to make custom tests outside this tests folder, which would be recommended for a custom port.

For example, add the following code in a file `print.py` in the `tests/unix/` subdirectory:

```
def print_one():
    print(1)

print_one()
```

If you run your tests, this test should appear in the test output:

```
$ cd ports/unix
$ make tests
skip  unix/extra_coverage.py
pass  unix/ffi_callback.py
pass  unix/ffi_float.py
pass  unix/ffi_float2.py
pass  unix/print.py
pass  unix/time.py
pass  unix/time2.py
```

Tests are run by comparing the output from the test target against the output from CPython. So any test should use print statements to indicate test results.

When writing tests for name or string-related functionality, please add both English/ASCII as well as non-English/non-ASCII text and include Unicode examples. Please do add comments in English explaining the meaning and intent of the Unicode text. This help ensure Unicode support is tested and verified across different platforms.

For tests that can't be compared to CPython (i.e. micropython-specific functionality), you can provide a `.py.exp` file which will be used as the truth for comparison.

The other way to run tests, which is useful when running on targets other than the Unix port, is:

```
$ cd tests
$ ./run-tests.py
```

Then to run on a board:

```
$ ./run-tests.py -t /dev/ttyACM0
```

And to run only a certain set of tests (eg a directory):

```
$ ./run-tests.py -d basics
$ ./run-tests.py float/builtin*.py
```

4.2.1 Using run-tests.py

The `run-tests.py` script supports several parameters to customize test execution:

Target and Device Selection:

- `-t, --test-instance`

The -t option accepts the following for the test instance:

- **unix** - use the unix port of MicroPython, specified by the MICROPY_MICROPYTHON environment variable (which defaults to the standard variant of either the unix or windows ports, depending on the host platform)
- **webassembly** - use the webassembly port of MicroPython, specified by the MICROPY_MICROPYTHON_MJS environment variable (which defaults to the standard variant of the webassembly port)
- **port:<device>** - connect to and use the given serial port device
- **a<n>** - connect to and use /dev/ttyACM<n>
- **u<n>** - connect to and use /dev/ttyUSB<n>
- **c<n>** - connect to and use COM<n>
- **exec:<command>** - execute a command and attach to it's stdin/stdout
- **execpty:<command>** - execute a command and attach to the printed /dev/pts/<n> device
- **<a>.<b>.<c>.<d>** - connect to the given IPv4 address
- anything else specifies a serial port

Test Selection:

- `-d, --test-dirs` - Specify one or more test directories to run
- `-i, --include REGEX` - Include tests matching regex pattern
- `-e, --exclude REGEX` - Exclude tests matching regex pattern
- `files` - Specific test files to run

Execution Options:

- `--emit <EMITTER>` - MicroPython emitter, EMITTER can be bytecode or native. Default: bytecode
- `--via-mpy` - Compile .py files to .mpy first
- `--heapsize` - Set heap size for tests
- `-j, --jobs N` - Number of tests to run simultaneously

Set the MICROPY_MPYCROSS environment variable to use a specific version of `mpy-cross` when using `--via-mpy`.

Result Management:

- `-r, --result-dir` - Directory for test results. Default: results/
- `--print-failures` - Show diff of failed tests and exit
- `--clean-failures` - Delete .exp and .out files from prior failed tests
- `--run-failures` - Re-run only previously failed tests

Examples:

```
# Run only basic tests with native emitter
$ ./run-tests.py --emit native -d basics extmod

# Run tests excluding async functionality
$ ./run-tests.py -e async

# Run tests matching *_pep_*
$ ./run-tests.py -i *_pep_*

# Run specific test files in parallel
$ ./run-tests.py -j 4 basics/list*.py

# Test on connected ESP32 board
$ ./run-tests.py -t /dev/ttyUSB0
# or
$ ./run-tests.py -t u0

# Re-run only failed tests from previous run
$ ./run-tests.py --run-failures
```

4.3 The Compiler

The compilation process in MicroPython involves the following steps:

- The lexer converts the stream of text that makes up a MicroPython program into tokens.
- The parser then converts the tokens into an abstract syntax (parse tree).
- Then bytecode or native code is emitted based on the parse tree.

For purposes of this discussion we are going to add a simple language feature `add1` that can be use in Python as:

```
>>> add1 3
4
>>>
```

The `add1` statement takes an integer as argument and adds `1` to it.

4.3.1 Adding a grammar rule

MicroPython's grammar is based on the CPython grammar and is defined in py/grammar.h. This grammar is what is used to parse MicroPython source files.

There are two macros you need to know to define a grammar rule: `DEF_RULE` and `DEF_RULE_NC`. `DEF_RULE` allows you to define a rule with an associated compile function, while `DEF_RULE_NC` has no compile (NC) function for it.

A simple grammar definition with a compile function for our new `add1` statement looks like the following:

```
DEF_RULE(add1_stmt, c(add1_stmt), and(2), tok(KW_ADD1), rule(testlist))
```

The second argument `c(add1_stmt)` is the corresponding compile function that should be implemented in `py/compile.c` to turn this rule into executable code.

The third required argument can be `or` or `and`. This specifies the number of nodes associated with a statement. For example, in this case, our `add1` statement is similar to ADD1 in assembly language. It takes one numeric argument.

Therefore, the `add1_stmt` has two nodes associated with it. One node is for the statement itself, i.e the literal `add1` corresponding to `KW_ADD1`, and the other for its argument, a `testlist` rule which is the top-level expression rule.

Note: The `add1` rule here is just an example and not part of the standard MicroPython grammar.

The fourth argument in this example is the token associated with the rule, `KW_ADD1`. This token should be defined in the lexer by editing `py/lexer.h`.

Defining the same rule without a compile function is achieved by using the `DEF_RULE_NC` macro and omitting the compile function argument:

```
DEF_RULE_NC(add1_stmt, and(2), tok(KW_ADD1), rule(testlist))
```

The remaining arguments take on the same meaning. A rule without a compile function must be handled explicitly by all rules that may have this rule as a node. Such NC-rules are usually used to express sub-parts of a complicated grammar structure that cannot be expressed in a single rule.

Note: The macros `DEF_RULE` and `DEF_RULE_NC` take other arguments. For an in-depth understanding of supported parameters, see py/grammar.h.

4.3.2 Adding a lexical token

Every rule defined in the grammar should have a token associated with it that is defined in `py/lexer.h`. Add this token by editing the `_mp_token_kind_t` enum:

```
typedef enum _mp_token_kind_t {
    ...
    MP_TOKEN_KW_OR,
    MP_TOKEN_KW_PASS,
    MP_TOKEN_KW_RAISE,
    MP_TOKEN_KW_RETURN,
    MP_TOKEN_KW_TRY,
    MP_TOKEN_KW_WHILE,
    MP_TOKEN_KW_WITH,
    MP_TOKEN_KW_YIELD,
    MP_TOKEN_KW_ADD1,
    ...
} mp_token_kind_t;
```

Then also edit `py/lexer.c` to add the new keyword literal text:

```
static const char *const tok_kw[] = {
    ...
    "or",
    "pass",
    "raise",
    "return",
    "try",
    "while",
    "with",
    "yield",
    "add1",
```

(continues on next page)

(continued from previous page)

```
    ...
};
```

Notice the keyword is named depending on what you want it to be. For consistency, maintain the naming standard accordingly.

Note: The order of these keywords in `py/lexer.c` must match the order of tokens in the enum defined in `py/lexer.h`.

4.3.3 Parsing

In the parsing stage the parser takes the tokens produced by the lexer and converts them to an abstract syntax tree (AST) or *parse tree*. The implementation for the parser is defined in py/parse.c.

The parser also maintains a table of constants for use in different aspects of parsing, similar to what a symbol table does.

Several optimizations like constant folding on integers for most operations e.g. logical, binary, unary, etc, and optimizing enhancements on parenthesis around expressions are performed during this phase, along with some optimizations on strings.

It's worth noting that *docstrings* are discarded and not accessible to the compiler. Even optimizations like string interning are not applied to *docstrings*.

4.3.4 Compiler passes

Like many compilers, MicroPython compiles all code to MicroPython bytecode or native code. The functionality that achieves this is implemented in py/compile.c. The most relevant method you should know about is this:

```
mp_obj_t mp_compile(mp_parse_tree_t *parse_tree, qstr source_file, bool is_repl) {
    // Create a context for this module, and set its globals dict.
    mp_module_context_t *context = m_new_obj(mp_module_context_t);
    context->module.globals = mp_globals_get();

    // Compile the input parse_tree to a raw-code structure.
    mp_compiled_module_t cm;
    cm.context = context;
    mp_compile_to_raw_code(parse_tree, source_file, is_repl, &cm);

    // Create and return a function object that executes the outer module.
    return mp_make_function_from_proto_fun(cm.rc, cm.context, NULL);
}
```

The compiler compiles the code in four passes: scope, stack size, code size and emit. Each pass runs the same C code over the same AST data structure, with different things being computed each time based on the results of the previous pass.

First pass

In the first pass, the compiler learns about the known identifiers (variables) and their scope, being global, local, closed over, etc. In the same pass the emitter (bytecode or native code) also computes the number of labels needed for the emitted code.

```
// Compile pass 1.
comp->emit = emit_bc;
comp->emit_method_table = &emit_bc_method_table;

uint max_num_labels = 0;
for (scope_t *s = comp->scope_head; s != NULL && comp->compile_error == MP_OBJ_NULL;␣
↪s = s->next) {
    if (s->emit_options == MP_EMIT_OPT_ASM) {
        compile_scope_inline_asm(comp, s, MP_PASS_SCOPE);
    } else {
        compile_scope(comp, s, MP_PASS_SCOPE);

        // Check if any implicitly declared variables should be closed over.
        for (size_t i = 0; i < s->id_info_len; ++i) {
            id_info_t *id = &s->id_info[i];
            if (id->kind == ID_INFO_KIND_GLOBAL_IMPLICIT) {
                scope_check_to_close_over(s, id);
            }
        }
    }
    ...
}
```

Second and third passes

The second and third passes involve computing the Python stack size and code size for the bytecode or native code. After the third pass the code size cannot change, otherwise jump labels will be incorrect.

```
for (scope_t *s = comp->scope_head; s != NULL && comp->compile_error == MP_OBJ_NULL;␣
↪s = s->next) {
    ...

    // Pass 2: Compute the Python stack size.
    compile_scope(comp, s, MP_PASS_STACK_SIZE);

    // Pass 3: Compute the code size.
    if (comp->compile_error == MP_OBJ_NULL) {
        compile_scope(comp, s, MP_PASS_CODE_SIZE);
    }

    ...
}
```

Just before pass two there is a selection for the type of code to be emitted, which can either be native or bytecode.

```
// Choose the emitter type.
switch (s->emit_options) {
    case MP_EMIT_OPT_NATIVE_PYTHON:
    case MP_EMIT_OPT_VIPER:
        if (emit_native == NULL) {
```

(continues on next page)

(continued from previous page)

```
            emit_native = NATIVE_EMITTER(new)(&comp->compile_error, &comp->next_label,
↪ max_num_labels);
        }
        comp->emit_method_table = NATIVE_EMITTER_TABLE;
        comp->emit = emit_native;
        break;

    default:
        comp->emit = emit_bc;
        comp->emit_method_table = &emit_bc_method_table;
        break;
}
```

The bytecode option is the default but something unique to note for the native code option is that there is another option via `VIPER`. See the *Emitting native code* section for more details on viper annotations.

There is also support for *inline assembly code*, where assembly instructions are written as Python function calls but are emitted directly as the corresponding machine code. This assembler has only three passes (scope, code size, emit) and uses a different implementation, not the `compile_scope` function. See the inline assembler tutorial for more details.

Fourth pass

The fourth pass emits the final code that can be executed, either bytecode in the virtual machine, or native code directly by the CPU.

```
for (scope_t *s = comp->scope_head; s != NULL && comp->compile_error == MP_OBJ_NULL;␣
↪s = s->next) {
    ...

    // Pass 4: Emit the compiled bytecode or native code.
    if (comp->compile_error == MP_OBJ_NULL) {
        compile_scope(comp, s, MP_PASS_EMIT);
    }
}
```

4.3.5 Emitting bytecode

Statements in Python code usually correspond to emitted bytecode, for example `a + b` generates "push a" then "push b" then "binary op add". Some statements do not emit anything but instead affect other things like the scope of variables, for example `global a`.

The implementation of a function that emits bytecode looks similar to this:

```
void mp_emit_bc_unary_op(emit_t *emit, mp_unary_op_t op) {
    emit_write_bytecode_byte(emit, 0, MP_BC_UNARY_OP_MULTI + op);
}
```

We use the unary operator expressions for an example here but the implementation details are similar for other statements/expressions. The method `emit_write_bytecode_byte()` is a wrapper around the main function `emit_get_cur_to_write_bytecode()` that all functions must call to emit bytecode.

4.3.6 Emitting native code

Similar to how bytecode is generated, there should be a corresponding function in `py/emitnative.c` for each code statement:

```
static void emit_native_unary_op(emit_t *emit, mp_unary_op_t op) {
    vtype_kind_t vtype;
    emit_pre_pop_reg(emit, &vtype, REG_ARG_2);
    if (vtype == VTYPE_PYOBJ) {
        emit_call_with_imm_arg(emit, MP_F_UNARY_OP, op, REG_ARG_1);
        emit_post_push_reg(emit, VTYPE_PYOBJ, REG_RET);
    } else {
        adjust_stack(emit, 1);
        EMIT_NATIVE_VIPER_TYPE_ERROR(emit,
            MP_ERROR_TEXT("unary op %q not implemented"), mp_unary_op_method_
↪name[op]);
    }
}
```

The difference here is that we have to handle *viper typing*. Viper annotations allow us to handle more than one type of variable. By default all variables are Python objects, but with viper a variable can also be declared as a machine-typed variable like a native integer or pointer. Viper can be thought of as a superset of Python, where normal Python objects are handled as usual, while native machine variables are handled in an optimised way by using direct machine instructions for the operations. Viper typing may break Python equivalence because, for example, integers become native integers and can overflow (unlike Python integers which extend automatically to arbitrary precision).

4.4 Memory Management

Unlike programming languages such as C/C++, MicroPython hides memory management details from the developer by supporting automatic memory management of a "*Python heap*" that holds all Python objects. MicroPython uses garbage collection (GC) for automatic memory management. The garbage collector is responsible for freeing memory which is no longer in use.

Specifically, MicroPython uses a Mark and Sweep garbage collection algorithm. This algorithm has a mark phase that scans the heap marking all live objects, and then a sweep phase goes through the heap reclaiming all unmarked objects.

The MicroPython garbage collector is by default automatic, but manual control is available through the *gc* built-in module.

4.4.1 MicroPython Memory from C code

Awareness of the garbage collector is needed when writing C code that allocates memory from the "Python heap" (i.e. functions `m_malloc()`, `m_malloc0()`, `m_free()`, etc).

The mark phase of the garbage collector scans for live pointers to heap memory starting from the following roots:

- The stack of the main Python runtime (or REPL).
- The stacks of each "Python thread", for ports which implement Python threads on top of native operating system threads or tasks.
- The "root pointers" defined in C code using the macro `MP_REGISTER_ROOT_POINTER`. These are the recommended way to have statically scoped pointers to the Python heap.
- Tracked allocations made with the `m_tracked_calloc()`, `m_tracked_realloc` and `m_tracked_free()` functions. These special functions allow allocating a block of memory which is always considered live by the garbage collector. Similar to memory allocation in C, this memory is only freed by

calling `m_tracked_free()` or by soft reset. There is a small memory usage and runtime overhead to each tracked allocation. This feature is not enabled by default on all ports.

The garbage collector then recursively scans and marks all the memory pointed to by the root pointers, until all addresses are exhausted. This is sufficient to find all Python objects that are still in use by the MicroPython runtime.

However, the following memory will **not** be scanned by the garbage collector and could be freed prematurely:

- Static or global C variables which contain pointers to heap memory.
- Pointers which don't point to the "head" of an allocated buffer (i.e. to the exact address returned by `m_malloc()`), but instead to an address inside the allocated buffer (for example, a pointer to a nested struct). For performance reasons, the garbage collector doesn't mark the enclosing buffer in these cases.
- The stack of any thread or RTOS task which isn't running Python code or manually registered as a "Python thread" (for ports which support native threads or tasks).

Ways to avoid use-after-free in these scenarios:

- Use the tracked allocation API `m_tracked_calloc()`, `m_tracked_realloc()` and `m_tracked_free()`.
- Register a root pointer (see above), instead of storing a pointer in a static variable.
- Restructure the code, for example by having an API where Python code initialises a singleton Python object (implemented in C) which holds all of the relevant pointers instead of having them in static variables.

Note: *Soft Reset* always clears the Python heap and frees all memory. It's important not to hold any pointers to the heap after a soft reset, as they will become dangling pointers to freed memory.

Some ports support a "C heap" as well (see *C Dynamic Memory Allocation*), in which case you can allocate memory that will stay valid over soft reset by calling standard C functions `malloc`, etc.

4.4.2 The object model

All MicroPython objects are referred to by the `mp_obj_t` data type. This is usually word-sized (i.e. the same size as a pointer on the target architecture), and can be typically 32-bit (STM32, nRF, ESP32, Unix x86) or 64-bit (Unix x64). It can also be greater than a word-size for certain object representations, for example `OBJ_REPR_D` has a 64-bit sized `mp_obj_t` on a 32-bit architecture.

An `mp_obj_t` represents a MicroPython object, for example an integer, float, type, dict or class instance. Some objects, like booleans and small integers, have their value stored directly in the `mp_obj_t` value and do not require additional memory. Other objects have their value store elsewhere in memory (for example on the garbage-collected heap) and their `mp_obj_t` contains a pointer to that memory. A portion of `mp_obj_t` is the tag which tells what type of object it is.

See `py/mpconfig.h` for the specific details of the available representations.

Pointer tagging

Because pointers are word-aligned, when they are stored in an `mp_obj_t` the lower bits of this object handle will be zero. For example on a 32-bit architecture the lower 2 bits will be zero:

`********|********|********|******00`

These bits are reserved for purposes of storing a tag. The tag stores extra information as opposed to introducing a new field to store that information in the object, which may be inefficient. In MicroPython the tag tells if we are dealing with a small integer, interned (small) string or a concrete object, and different semantics apply to each of these.

For small integers the mapping is this:

```
********|********|********|*******1
```

Where the asterisks hold the actual integer value. For an interned string or an immediate object (e.g. `True`) the layout of the `mp_obj_t` value is, respectively:

```
********|********|********|*****010
```

```
********|********|********|*****110
```

While a concrete object that is none of the above takes the form:

```
********|********|********|******00
```

The stars here correspond to the address of the concrete object in memory.

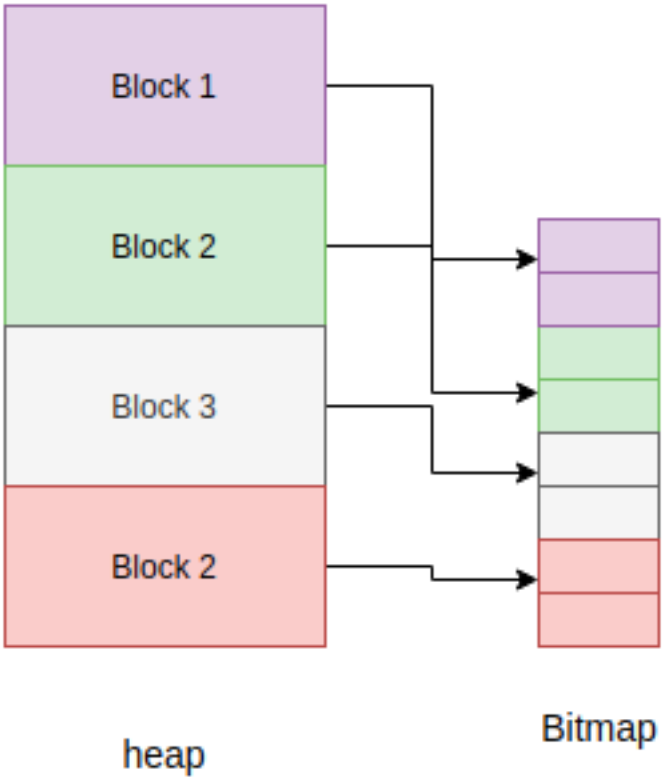
4.4.3 Allocation of objects

The value of a small integer is stored directly in the `mp_obj_t` and will be allocated in-place, not on the heap or elsewhere. As such, creation of small integers does not affect the heap. Similarly for interned strings that already have their textual data stored elsewhere, and immediate values like `None`, `False` and `True`.

Everything else which is a concrete object is allocated on the heap and its object structure is such that a field is reserved in the object header to store the type of the object.

```
++++++++++
+         +
+ type    + object header
+         +
++++++++++
+         + object items
+         +
+         +
++++++++++
```

The heap's smallest unit of allocation is a block, which is four machine words in size (16 bytes on a 32-bit machine, 32 bytes on a 64-bit machine). Another structure also allocated on the heap tracks the allocation of objects in each block. This structure is called a *bitmap*.



The bitmap tracks whether a block is “free” or “in use” and use two bits to track this state for each block.

The mark-sweep garbage collector manages the objects allocated on the heap, and also utilises the bitmap to mark objects that are still in use. See py/gc.c for the full implementation of these details.

Allocation: heap layout

The heap is arranged such that it consists of blocks in pools. A block can have different properties:

- *ATB(allocation table byte):* If set, then the block is a normal block
- *FREE:* Free block
- *HEAD:* Head of a chain of blocks
- *TAIL:* In the tail of a chain of blocks
- *MARK :* Marked head block
- *FTB(finaliser table byte):* If set, then the block has a finaliser

4.5 Implementing a Module

This chapter details how to implement a core module in MicroPython. MicroPython modules can be one of the following:

- Built-in module: A general module that is be part of the MicroPython repository.
- User module: A module that is useful for your specific project that you maintain in your own repository or private codebase.
- Dynamic module: A module that can be deployed and imported at runtime to your device.

A module in MicroPython can be implemented in one of the following locations:

- py/: A core library that mirrors core CPython functionality.
- extmod/: A CPython or MicroPython-specific module that is shared across multiple ports.
- ports/<port>/: A port-specific module.

Note: This chapter describes modules implemented in `py/` or core modules. See *Extending MicroPython in C* for details on implementing an external module. For details on port-specific modules, see *Porting MicroPython*.

4.5.1 Implementing a core module

Like CPython, MicroPython has core builtin modules that can be accessed through import statements. An example is the `gc` module discussed in *Memory Management*.

MicroPython has several other builtin standard/core modules like `io`, `array`, etc. Adding a new core module involves several modifications.

First, create the `C` file in the `py/` directory. In this example we are adding a hypothetical new module `subsystem` in the file `modsubsystem.c`:

```
#include "py/builtin.h"
#include "py/runtime.h"

#if MICROPY_PY_SUBSYSTEM

// info()
static mp_obj_t py_subsystem_info(void) {
    return MP_OBJ_NEW_SMALL_INT(42);
}
MP_DEFINE_CONST_FUN_OBJ_0(subsystem_info_obj, py_subsystem_info);

static const mp_rom_map_elem_t mp_module_subsystem_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_subsystem) },
    { MP_ROM_QSTR(MP_QSTR_info), MP_ROM_PTR(&subsystem_info_obj) },
};
static MP_DEFINE_CONST_DICT(mp_module_subsystem_globals, mp_module_subsystem_globals_
↪table);

const mp_obj_module_t mp_module_subsystem = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&mp_module_subsystem_globals,
};

MP_REGISTER_MODULE(MP_QSTR_subsystem, mp_module_subsystem);

#endif
```

The implementation includes a definition of all functions related to the module and adds the functions to the module's global table in `mp_module_subsystem_globals_table`. It also creates the module object with `mp_module_subsystem`. The module is then registered with the wider system via the `MP_REGISTER_MODULE` macro.

After building and running the modified MicroPython, the module should now be importable:

```
>>> import subsystem
>>> subsystem.info()
```

(continues on next page)

(continued from previous page)

```
42
>>>
```

Our `info()` function currently returns just a single number but can be extended to do anything. Similarly, more functions can be added to this new module.

4.6 Optimizations

MicroPython uses several optimizations to save RAM but also ensure the efficient execution of programs. This chapter discusses some of these optimizations.

Note: *MicroPython string interning* and *Maps and Dictionaries* details other optimizations on strings and dictionaries.

4.6.1 Frozen bytecode

When MicroPython loads Python code from the filesystem, it first has to parse the file into a temporary in-memory representation, and then generate bytecode for execution, both of which are stored in the heap (in RAM). This can lead to significant amounts of memory being used. The MicroPython cross compiler can be used to generate a `.mpy` file, containing the pre-compiled bytecode for a Python module. This will still be loaded into RAM, but it avoids the additional overhead of the parsing stage.

As a further optimisation, the pre-compiled bytecode from a `.mpy` file can be "frozen" into the firmware image as part of the main firmware compilation process, which means that the bytecode will be executed from ROM. This can lead to a significant memory saving, and reduce heap fragmentation.

See *MicroPython manifest files* for more information.

4.6.2 Variables

MicroPython processes local and global variables differently. Global variables are stored and looked up from a global dictionary that is allocated on the heap (note that each module has its own separate dict, so separate namespace). Local variables on the other hand are stored on the Python value stack, which may live on the C stack or on the heap. They are accessed directly by their offset within the Python stack, which is more efficient than a global lookup in a dict.

The length of global variable names also affects how much RAM is used as identifiers are stored in RAM. The shorter the identifier, the less memory is used.

The other aspect is that `const` variables that start with an underscore are treated as proper constants and are not allocated or added in a dictionary, hence saving some memory. These variables use `const()` from the MicroPython library. Therefore:

```
from micropython import const

X = const(1)
_Y = const(2)
foo(X, _Y)
```

Compiles to:

```
X = 1
foo(1, 2)
```

See *micropython.const()* for complete details on usage requirements and limitations.

4.6.3 Allocation of memory

Most of the common MicroPython constructs are not allocated on the heap. However the following are:

- Dynamic data structures like lists, mappings, etc;
- Functions, classes and object instances;
- imports; and
- First-time assignment of global variables (to create the slot in the global dict).

For a detailed discussion on a more user-centric perspective on optimization, see Maximising MicroPython speed

4.7 MicroPython string interning

MicroPython uses string interning to save both RAM and ROM. This avoids having to store duplicate copies of the same string. Primarily, this applies to identifiers in your code, as something like a function or variable name is very likely to appear in multiple places in the code. In MicroPython an interned string is called a QSTR (uniQue STRing).

A QSTR value (with type `qstr`) is a index into a linked list of QSTR pools. QSTRs store their length and a hash of their contents for fast comparison during the de-duplication process. All bytecode operations that work with strings use a QSTR argument.

4.7.1 Compile-time QSTR generation

In the MicroPython C code, any strings that should be interned in the final firmware are written as `MP_QSTR_Foo`. At compile time this will evaluate to a `qstr` value that points to the index of `"Foo"` in the QSTR pool.

A multi-step process in the `Makefile` makes this work. In summary this process has three parts:

1. Find all `MP_QSTR_Foo` tokens in the code.
2. Generate a static QSTR pool containing all the string data (including lengths and hashes).
3. Replace all `MP_QSTR_Foo` (via the preprocessor) with their corresponding index.

`MP_QSTR_Foo` tokens are searched for in two sources:

1. All files referenced in `$(SRC_QSTR)`. This is all C code (i.e. `py`, `extmod`, `ports/stm32`) but not including third-party code such as `lib`.
2. Additional `$(QSTR_GLOBAL_DEPENDENCIES)` (which includes `mpconfig*.h`).

Note: `frozen_mpy.c` (generated by mpy-tool.py) has its own QSTR generation and pool.

Some additional strings that can't be expressed using the `MP_QSTR_Foo` syntax (e.g. they contain non-alphanumeric characters) are explicitly provided in `qstrdefs.h` and `qstrdefsport.h` via the `$(QSTR_DEFS)` variable.

Processing happens in the following stages:

1. `qstr.i.last` is the concatenation of putting every single input file through the C pre-processor. This means that any conditionally disabled code will be removed, and macros expanded. This means we don't add strings to the pool that won't be used in the final firmware. Because at this stage (thanks to the `NO_QSTR` macro added by `QSTR_GEN_CFLAGS`) there is no definition for `MP_QSTR_Foo` it passes through this stage unaffected. This file also includes comments from the preprocessor that include line number information. Note that this step only uses files that have changed, which means that `qstr.i.last` will only contain data from files that have changed since the last compile.
2. `qstr.split` is an empty file created after running `makeqstrdefs.py split` on qstr.i.last. It's just used as a dependency to indicate that the step ran. This script outputs one file per input C file, `genhdr/qstr/..file.c.qstr`, which contains only the matched QSTRs. Each QSTR is printed as `Q(Foo)`. This step is necessary to combine the existing files with the new data generated from the incremental update in `qstr.i.last`.
3. `qstrdefs.collected.h` is the output of concatenating `genhdr/qstr/*` using `makeqstrdefs.py cat`. This is now the full set of `MP_QSTR_Foo`'s found in the code, now formatted as `Q(Foo)`, one-per-line, with duplicates. This file is only updated if the set of qstrs has changed. A hash of the QSTR data is written to another file (`qstrdefs.collected.h.hash`) which allows it to track changes across builds.
4. Generate an enumeration, each entry of which maps a `MP_QSTR_Foo` to it's corresponding index. It concatenates `qstrdefs.collected.h` with `qstrdefs*.h`, then it transforms each line from `Q(Foo)` to `"Q(Foo)"` so they pass through the preprocessor unchanged. Then the preprocessor is used to deal with any conditional compilation in `qstrdefs*.h`. Then the transformation is undone back to `Q(Foo)`, and saved as `qstrdefs.preprocessed.h`.
5. `qstrdefs.generated.h` is the output of `makeqstrdata.py`. For each `Q(Foo)` in qstrdefs.preprocessed.h (plus some extra hard-coded ones), it outputs `QDEF(MP_QSTR_Foo, (const byte*)"hash" "Foo")`.

Then in the main compile, two things happen with `qstrdefs.generated.h`:

1. In qstr.h, each QDEF becomes an entry in an enum, which makes `MP_QSTR_Foo` available to code and equal to the index of that string in the QSTR table.
2. In qstr.c, the actual QSTR data table is generated as elements of the `mp_qstr_const_pool->qstrs`.

4.7.2 Run-time QSTR generation

Additional QSTR pools can be created at runtime so that strings can be added to them. For example, the code:

```
foo[x] = 3
```

Will need to create a QSTR for the value of `x` so it can be used by the "load attr" bytecode.

Also, when compiling Python code, identifiers and literals need to have QSTRs created. Note: only literals shorter than 10 characters become QSTRs. This is because a regular string on the heap always takes up a minimum of 16 bytes (one GC block), whereas QSTRs allow them to be packed more efficiently into the pool.

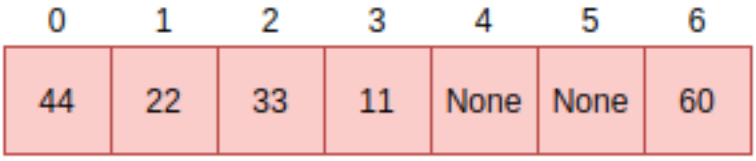
QSTR pools (and the underlying "chunks" that store the string data) are allocated on-demand on the heap with a minimum size.

4.8 Maps and Dictionaries

MicroPython dictionaries and maps use techniques called open addressing and linear probing. This chapter details both of these methods.

4.8.1 Open addressing

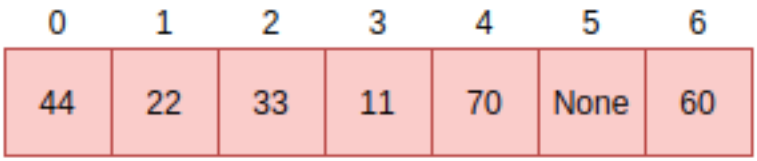
Open addressing is used to resolve collisions. Collisions are very common occurrences and happen when two items happen to hash to the same slot or location. For example, given a hash setup as this:



If there is a request to fill slot `0` with `70`, since the slot `0` is not empty, open addressing finds the next available slot in the dictionary to service this request. This sequential search for an alternate location is called *probing*. There are several sequence probing algorithms but MicroPython uses linear probing that is described in the next section.

4.8.2 Linear probing

Linear probing is one of the methods for finding an available address or slot in a dictionary. In MicroPython, it is used with open addressing. To service the request described above, unlike other probing algorithms, linear probing assumes a fixed interval of `1` between probes. The request will therefore be serviced by placing the item in the next free slot which is slot `4` in our example:



The same methods i.e open addressing and linear probing are used to search for an item in a dictionary. Assume we want to search for the data item `33`. The computed hash value will be 2. Looking at slot 2 reveals `33`, at this point, we return `True`. Searching for `70` is quite different as there was a collision at the time of insertion. Therefore computing the hash value is `0` which is currently holding `44`. Instead of simply returning `False`, we perform a sequential search starting at point `1` until the item `70` is found or we encounter a free slot. This is the general way of performing look-ups in hashes:

```
// not yet found, keep searching in this table
pos = (pos + 1) % set->alloc;

if (pos == start_pos) {
    // search got back to starting position, so index is not in table
    if (lookup_kind & MP_MAP_LOOKUP_ADD_IF_NOT_FOUND) {
        if (avail_slot != NULL) {
```

(continues on next page)

(continued from previous page)

```
            // there was an available slot, so use that
            set->used++;
            *avail_slot = index;
            return index;
        } else {
            // not enough room in table, rehash it
            mp_set_rehash(set);
            // restart the search for the new element
            start_pos = pos = hash % set->alloc;
        }
    }
} else {
    return MP_OBJ_NULL;
}
```

4.9 The public C API

The public C-API comprises functions defined in all C header files in the `py/` directory. Most of the important core runtime C APIs are exposed in `runtime.h` and `obj.h`.

The following is an example of public API functions from `obj.h`:

```
mp_obj_t mp_obj_new_list(size_t n, mp_obj_t *items);
mp_obj_t mp_obj_list_append(mp_obj_t self_in, mp_obj_t arg);
mp_obj_t mp_obj_list_remove(mp_obj_t self_in, mp_obj_t value);
void mp_obj_list_get(mp_obj_t self_in, size_t *len, mp_obj_t **items);
```

At its core, any functions and macros in header files make up the public API and can be used to access very low-level details of MicroPython. Static inline functions in header files are fine too, such functions will be inlined in the code when used.

Header files in the `ports` directory are only exposed to the functionality specific to a given port.

4.10 Extending MicroPython in C

This chapter describes options for implementing additional functionality in C, but from code written outside of the main MicroPython repository. The first approach is useful for building your own custom firmware with some project-specific additional modules or functions that can be accessed from Python. The second approach is for building modules that can be loaded at runtime.

Please see the *library section* for more information on building core modules that live in the main MicroPython repository.

4.10.1 MicroPython external C modules

When developing modules for use with MicroPython you may find you run into limitations with the Python environment, often due to an inability to access certain hardware resources or Python speed limitations.

If your limitations can't be resolved with suggestions in *Maximising MicroPython speed*, writing some or all of your module in C (and/or *C++ if implemented for your port*) is a viable option.

If your module is designed to access or work with commonly available hardware or libraries please consider implementing it inside the MicroPython source tree alongside similar modules and submitting it as a pull request. If however you're targeting obscure or proprietary systems it may make more sense to keep this external to the main MicroPython repository.

This chapter describes how to compile such external modules into the MicroPython executable or firmware image. Both Make and CMake build tools are supported, and when writing an external module it's a good idea to add the build files for both of these tools so the module can be used on all ports. But when compiling a particular port you will only need to use one method of building, either Make or CMake.

An alternative approach is to use *Native machine code in .mpy files* which allows writing custom C code that is placed in a .mpy file, which can be imported dynamically in to a running MicroPython system without the need to recompile the main firmware.

Structure of an external C module

A MicroPython user C module is a directory with the following files:

- `*.c` / `*.cpp` / `*.h` source code files for your module.

  These will typically include the low level functionality being implemented and the MicroPython binding functions to expose the functions and module(s).

  Currently the best reference for writing these functions/modules is to find similar modules within the MicroPython tree and use them as examples.

- `micropython.mk` contains the Makefile fragment for this module.

  `$(USERMOD_DIR)` is available in `micropython.mk` as the path to your module directory. As it's redefined for each c module, is should be expanded in your `micropython.mk` to a local make variable, eg `EXAMPLE_MOD_DIR := $(USERMOD_DIR)`

  Your `micropython.mk` must add your modules source files to the `SRC_USERMOD_C` or `SRC_USERMOD_LIB_C` variables. The former will be processed for `MP_QSTR_` and `MP_REGISTER_MODULE` definitions, the latter will not (e.g. helpers and library code that isn't MicroPython-specific). These paths should include your expanded copy of `$(USERMOD_DIR)`, e.g.:

  ```
  SRC_USERMOD_C += $(EXAMPLE_MOD_DIR)/modexample.c
  SRC_USERMOD_LIB_C += $(EXAMPLE_MOD_DIR)/utils/algorithm.c
  ```

  Similarly, use `SRC_USERMOD_CXX` and `SRC_USERMOD_LIB_CXX` for C++ source files. If you want to include assembly files use `SRC_USERMOD_LIB_ASM`.

  If you have custom compiler options (like `-I` to add directories to search for header files), these should be added to `CFLAGS_USERMOD` for C code and to `CXXFLAGS_USERMOD` for C++ code.

- `micropython.cmake` contains the CMake configuration for this module.

  In `micropython.cmake`, you may use `${CMAKE_CURRENT_LIST_DIR}` as the path to the current module.

  Your `micropython.cmake` should define an `INTERFACE` library and associate your source files, compile definitions and include directories with it. The library should then be linked to the `usermod` target.

```
add_library(usermod_cexample INTERFACE)

target_sources(usermod_cexample INTERFACE
    ${CMAKE_CURRENT_LIST_DIR}/examplemodule.c
)

target_include_directories(usermod_cexample INTERFACE
    ${CMAKE_CURRENT_LIST_DIR}
)

target_link_libraries(usermod INTERFACE usermod_cexample)
```

See below for full usage example.

Basic example

The `cexample` module provides examples for a function and a class. The `cexample.add_ints(a, b)` function adds two integer args together and returns the result. The `cexample.Timer()` type creates timers that can be used to measure the elapsed time since the object is instantiated.

The module can be found in the MicroPython source tree in the examples directory and has a source file and a Makefile fragment with content as described above:

```
micropython/
└──examples/
   └──usercmodule/
      └──cexample/
         ├── examplemodule.c
         ├── micropython.mk
         └── micropython.cmake
```

Refer to the comments in these files for additional explanation. Next to the `cexample` module there's also `cppexample` which works in the same way but shows one way of mixing C and C++ code in MicroPython.

Compiling the cmodule into MicroPython

To build such a module, compile MicroPython (see getting started), applying 2 modifications:

1. Set the build-time flag `USER_C_MODULES` to point to the modules you want to include. For ports that use Make this variable should be a directory which is searched automatically for modules. For ports that use CMake this variable should be a file which includes the modules to build. See below for details.
2. Enable the modules by setting the corresponding C preprocessor macro to 1. This is only needed if the modules you are building are not automatically enabled.

For building the example modules which come with MicroPython, set `USER_C_MODULES` to the `examples/usercmodule` directory for Make, or to `examples/usercmodule/micropython.cmake` for CMake.

For example, here's how the to build the unix port with the example modules:

```
cd micropython/ports/unix
make USER_C_MODULES=../../examples/usercmodule
```

You may need to run `make clean` once at the start when including new user modules in the build. The build output will show the modules found:

```
...
Including User C Module from ../../examples/usercmodule/cexample
Including User C Module from ../../examples/usercmodule/cppexample
...
```

For a CMake-based port such as rp2, this will look a little different (note that CMake is actually invoked by `make`):

```
cd micropython/ports/rp2
make USER_C_MODULES=../../examples/usercmodule/micropython.cmake
```

Again, you may need to run `make clean` first for CMake to pick up the user modules. The CMake build output lists the modules by name:

```
...
Including User C Module(s) from ../../examples/usercmodule/micropython.cmake
Found User C Module(s): usermod_cexample, usermod_cppexample
...
```

The contents of the top-level `micropython.cmake` can be used to control which modules are enabled.

For your own projects it's more convenient to keep custom code out of the main MicroPython source tree, so a typical project directory structure will look like this:

```
my_project/
├── modules/
│   ├── example1/
│   │   ├── example1.c
│   │   ├── micropython.mk
│   │   └── micropython.cmake
│   ├── example2/
│   │   ├── example2.c
│   │   ├── micropython.mk
│   │   └── micropython.cmake
│   └── micropython.cmake
└── micropython/
    ├──ports/
   ... ├──stm32/
       ...
```

When building with Make set `USER_C_MODULES` to the `my_project/modules` directory. For example, building the stm32 port:

```
cd my_project/micropython/ports/stm32
make USER_C_MODULES=../../../modules
```

When building with CMake the top level `micropython.cmake` – found directly in the `my_project/modules` directory – should `include` all of the modules you want to have available:

```
include(${CMAKE_CURRENT_LIST_DIR}/example1/micropython.cmake)
include(${CMAKE_CURRENT_LIST_DIR}/example2/micropython.cmake)
```

Then build with:

```
cd my_project/micropython/ports/esp32
make USER_C_MODULES=../../../../modules/micropython.cmake
```

Note that the esp32 port needs the extra `..` for relative paths due to the location of its main `CMakeLists.txt` file. You can also specify absolute paths to `USER_C_MODULES`.

All modules specified by the `USER_C_MODULES` variable (either found in this directory when using Make, or added via `include` when using CMake) will be compiled, but only those which are enabled will be available for importing. User modules are usually enabled by default (this is decided by the developer of the module), in which case there is nothing more to do than set `USER_C_MODULES` as described above.

If a module is not enabled by default then the corresponding C preprocessor macro must be enabled. This macro name can be found by searching for the `MP_REGISTER_MODULE` line in the module's source code (it usually appears at the end of the main source file). This macro should be surrounded by a `#if X` / `#endif` pair, and the configuration option `X` must be set to 1 using `CFLAGS_EXTRA` to make the module available. If there is no `#if X` / `#endif` pair then the module is enabled by default.

For example, the `examples/usercmodule/cexample` module is enabled by default so has the following line in its source code:

```
MP_REGISTER_MODULE(MP_QSTR_cexample, example_user_cmodule);
```

Alternatively, to make this module disabled by default but selectable through a preprocessor configuration option, it would be:

```
#if MODULE_CEXAMPLE_ENABLED
MP_REGISTER_MODULE(MP_QSTR_cexample, example_user_cmodule);
#endif
```

In this case the module is enabled by adding `CFLAGS_EXTRA=-DMODULE_CEXAMPLE_ENABLED=1` to the `make` command, or editing `mpconfigport.h` or `mpconfigboard.h` to add

```
#define MODULE_CEXAMPLE_ENABLED (1)
```

Note that the exact method depends on the port as they have different structures. If not done correctly it will compile but importing will fail to find the module.

Specifying C modules via a manifest

As an alternative to passing `USER_C_MODULES` on the command line, C modules can be listed inside a frozen manifest using `c_module()`. This is convenient when a board or project always pulls in the same set of C modules: the manifest becomes the single place that declares both frozen Python code and the C modules required to support it.

```
# In ports/myboard/boards/MYBOARD/manifest.py
include("$(PORT_DIR)/boards/manifest.py")
c_module("$(MPY_DIR)/examples/usercmodule/cexample")
c_module("$(BOARD_DIR)/../../drivers/sensor")
```

The manifest will need to be loaded via `FROZEN_MANIFEST` (either set in `mpconfigboard.{mk,cmake}` or passed on the make command line), and modules listed with `c_module()` combine additively with any paths supplied via `USER_C_MODULES` on the command line. Duplicate paths are de-duplicated, so mixing the two is safe.

See *MicroPython manifest files* for the full `c_module()` API and supported `$(VAR)` path substitutions.

Module usage in MicroPython

Once built into your copy of MicroPython, the module can now be accessed in Python just like any other builtin module, e.g.

```
import cexample
print(cexample.add_ints(1, 3))
# should display 4
```

```
from cexample import Timer
from time import sleep_ms

watch = Timer()
sleep_ms(1000)
print(watch.time())
# should display approximately 1000
```

C Dynamic Memory Allocation

MicroPython uses its own "Python heap" for *Memory Management*, which is not the same as the "C heap" used by C library functions `malloc()`, `free()`, etc. Not every MicroPython port comes with a "C heap" at all.

Tier 1 & 2 ports have varying support for C dynamic memory allocation via a "C heap":

- unix, windows, esp32 and webassembly ports support C dynamic memory allocation.
- rp2 port will fail to allocate any memory at runtime unless the firmware is built with `MICROPY_C_HEAP_SIZE=n` to reserve `n` bytes of memory for a C heap. This memory will not be available for Python code to use.
- alif, mimxrt, nrf, renesas-ra, samd, and stm32 port builds that include dynamic C allocation will fail at link-time with errors such as `undefined reference to `malloc'`. MicroPython has no built-in support for dynamic C allocation on these ports. Any solution requires manually adding a C heap implementation to the custom build.
- zephyr port currently does not support building with user modules.

Python heap as C heap

It may be practical for C code to call "Python heap" dynamic allocation functions such `m_malloc()`, `m_malloc0()` and `m_free()` instead.

See *MicroPython Memory from C code* for more information about this approach.

C++ Modules

Most Tier 1 & 2 MicroPython ports (and some Tier 3) support building C++ user modules, using the C++-specific environment variables described above.

Integrating C++ and MicroPython successfully involves some additional considerations:

C++ Dynamic Memory Allocation

C++ programs (as well as C++ Standard Library features) typically use dynamic memory allocation. The C++ default memory allocator (i.e. operators `new` and `delete`) is typically implemented as a layer on top of *C Dynamic Memory Allocation*.

For MicroPython ports which don't include C dynamic memory allocation support, C++ dynamic memory allocation can be supported in one of two ways:

- Implement C dynamic memory allocation in your custom build.
- Implement a custom C++ allocator in your custom build.

Linkage Considerations

Because MicroPython is a C-based project, any symbols which link to or from MicroPython need to be qualified `extern "C"` in C++ code.

It's strongly recommended to follow the pattern demonstrated in examples/usercmodule/cppexample, where the Python module is implemented in a minimal C file wrapper around the C++ code.

4.10.2 Native machine code in .mpy files

This section describes how to build and work with .mpy files that contain native machine code from a language other than Python. This allows you to write code in a language like C, compile and link it into a .mpy file, and then import this file like a normal Python module. This can be used for implementing functionality which is performance critical, or for including an existing library written in another language.

One of the main advantages of using native .mpy files is that native machine code can be imported by a script dynamically, without the need to rebuild the main MicroPython firmware. This is in contrast to *MicroPython external C modules* which also allows defining custom modules in C but they must be compiled into the main firmware image.

The focus here is on using C to build native modules, but in principle any language which can be compiled to stand-alone machine code can be put into a .mpy file.

A native .mpy module is built using the `mpy_ld.py` tool, which is found in the `tools/` directory of the project. This tool takes a set of object files (.o files) and links them together to create a native .mpy files. It requires CPython 3 and the library pyelftools v0.25 or greater.

Supported features and limitations

A .mpy file can contain MicroPython bytecode and/or native machine code. If it contains native machine code then the .mpy file has a specific architecture associated with it. Current supported architectures are (these are the valid options for the `ARCH` variable, see below):

- `x86` (32 bit)
- `x64` (64 bit x86)
- `armv6m` (ARM Thumb, eg Cortex-M0)
- `armv7m` (ARM Thumb 2, eg Cortex-M3)
- `armv7emsp` (ARM Thumb 2, single precision float, eg Cortex-M4F, Cortex-M7)
- `armv7emdp` (ARM Thumb 2, double precision float, eg Cortex-M7)
- `xtensa` (non-windowed, eg ESP8266)

- `xtensawin` (windowed with window size 8, eg ESP32, ESP32S3)
- `rv32imc` (RISC-V 32 bits with compressed instructions, eg ESP32C3, ESP32C6)
- `rv64imc` (RISC-V 64 bits with compressed instructions)

If the chosen platform supports explicit architecture flags and you want to let the output .mpy file carry those flags' value, you must pass them to the `ARCH_FLAGS` flags variable when building the .mpy file.

When compiling and linking the native .mpy file the architecture must be chosen and the corresponding file can only be imported on that architecture (and if architecture flags are present, only if they match the target's capabilities). For more details about .mpy files see *MicroPython .mpy files*.

Native code must be compiled as position independent code (PIC) and use a global offset table (GOT), although the details of this varies from architecture to architecture. When importing .mpy files with native code the import machinery is able to do some basic relocation of the native code. This includes relocating text, rodata and BSS sections.

Supported features of the linker and dynamic loader are:

- executable code (text)
- read-only data (rodata), including strings and constant data (arrays, structs, etc)
- zeroed data (BSS)
- pointers in text to text, rodata and BSS
- pointers in rodata to text, rodata and BSS

The known limitations are:

- data sections are not supported; workaround: use BSS data and initialise the data values explicitly
- static BSS variables are not supported; workaround: use global BSS variables
- thread-local storage variables are not supported on rv32imc; workaround: use global BSS variables or allocate some space on the heap to store them

So, if your C code has writable data, make sure the data is defined globally, without an initialiser, and only written to within functions.

The native module is not automatically linked against the standard static libraries like `libm.a` and `libgcc.a`, which can lead to `undefined symbol` errors. You can link the runtime libraries by setting `LINK_RUNTIME = 1` in your Makefile. Custom static libraries can also be linked by adding `MPY_LD_FLAGS += -l path/to/library.a`. Note that these are linked into the native module and will not be shared with other modules or the system.

Linker limitation: the native module is not linked against the symbol table of the full MicroPython firmware. Rather, it is linked against an explicit table of exported symbols found in `mp_fun_table` (in `py/nativeglue.h`), that is fixed at firmware build time. It is thus not possible to simply call some arbitrary HAL/OS/RTOS/system function, for example, unless that resides at a fixed address. In that case, the path of a linkerscript containing a series of symbol names and their fixed address can be passed to `mpy_ld.py` via the `--externs` command line argument. That way symbols appearing in the linkerscript will take precedence over what is provided from object files, but at the moment the object files' implementation will still reside in the final MPY file. The linkerscript parser is limited in its capabilities, and is currently used only for parsing the ESP8266 port ROM symbols list (see `ports/esp8266/boards/eagle.rom.addr.v6.ld`).

New symbols can be added to the end of the table and the firmware rebuilt. The symbols also need to be added to `tools/mpy_ld.py`'s `fun_table` dict in the same location. This allows `mpy_ld.py` to be able to pick the new symbols up and provide relocations for them when the mpy is imported. Finally, if the symbol is a function, a macro or stub should be added to `py/dynruntime.h` to make it easy to call the function.

Defining a native module

A native .mpy module is defined by a set of files that are used to build the .mpy. The filesystem layout consists of two main parts, the source files and the Makefile:

- In the simplest case only a single C source file is required, which contains all the code that will be compiled into the .mpy module. This C source code must include the `py/dynruntime.h` file to access the MicroPython dynamic API, and must at least define a function called `mpy_init`. This function will be the entry point of the module, called when the module is imported.

  The module can be split into multiple C source files if desired. Parts of the module can also be implemented in Python. All source files should be listed in the Makefile, by adding them to the `SRC` variable (see below). This includes both C source files as well as any Python files which will be included in the resulting .mpy file.

- The `Makefile` contains the build configuration for the module and list the source files used to build the .mpy module. It should define `MPY_DIR` as the location of the MicroPython repository (to find header files, the relevant Makefile fragment, and the `mpy_ld.py` tool), `MOD` as the name of the module, `SRC` as the list of source files, optionally specify the machine architecture via `ARCH`, along with optional machine architecture flags specified via `ARCH_FLAGS`, and then include `py/dynruntime.mk`.

Minimal example

This section provides a fully working example of a simple module named `factorial`. This module provides a single function `factorial.factorial(x)` which computes the factorial of the input and returns the result.

Directory layout:

```
factorial/
├── factorial.c
└── Makefile
```

The file `factorial.c` contains:

```
// Include the header file to get access to the MicroPython API
#include "py/dynruntime.h"

// Helper function to compute factorial
static mp_int_t factorial_helper(mp_int_t x) {
    if (x == 0) {
        return 1;
    }
    return x * factorial_helper(x - 1);
}

// This is the function which will be called from Python, as factorial(x)
static mp_obj_t factorial(mp_obj_t x_obj) {
    // Extract the integer from the MicroPython input object
    mp_int_t x = mp_obj_get_int(x_obj);
    // Calculate the factorial
    mp_int_t result = factorial_helper(x);
    // Convert the result to a MicroPython integer object and return it
    return mp_obj_new_int(result);
}
// Define a Python reference to the function above
static MP_DEFINE_CONST_FUN_OBJ_1(factorial_obj, factorial);

// This is the entry point and is called when the module is imported
```

(continues on next page)

(continued from previous page)

```
mp_obj_t mpy_init(mp_obj_fun_bc_t *self, size_t n_args, size_t n_kw, mp_obj_t *args) {
    // This must be first, it sets up the globals dict and other things
    MP_DYNRUNTIME_INIT_ENTRY

    // Make the function available in the module's namespace
    mp_store_global(MP_QSTR_factorial, MP_OBJ_FROM_PTR(&factorial_obj));

    // This must be last, it restores the globals dict
    MP_DYNRUNTIME_INIT_EXIT
}
```

The file `Makefile` contains:

```
# Location of top-level MicroPython directory
MPY_DIR = ../../..

# Name of module
MOD = factorial

# Source files (.c or .py)
SRC = factorial.c

# Architecture to build for (x86, x64, armv6m, armv7m, xtensa, xtensawin, rv32imc,␣
↪rv64imc)
ARCH = x64

# Include to get the rules for compiling and linking the module
include $(MPY_DIR)/py/dynruntime.mk
```

Compiling the module

The prerequisite tools needed to build a native .mpy file are:

- The MicroPython repository (at least the `py/` and `tools/` directories).
- CPython 3, and the library pyelftools (eg `pip install 'pyelftools>=0.25'`).
- GNU make.
- A C compiler for the target architecture (if C source is used).
- Optionally `mpy-cross`, built from the MicroPython repository (if .py source is used).

Be sure to select the correct `ARCH` for the target you are going to run on. Then build with:

```
$ make
```

Without modifying the Makefile you can specify the target architecture via:

```
$ make ARCH=armv7m
```

Same applies for optional architecture flags via:

```
$ make ARCH=rv32imc ARCH_FLAGS=zba
```

Module usage in MicroPython

Once the module is built there should be a file called `factorial.mpy`. Copy this so it is accessible on the filesystem of your MicroPython system and can be found in the import path. The module can now be accessed in Python just like any other module, for example:

```
import factorial
print(factorial.factorial(10))
# should display 3628800
```

Using Picolibc when building modules

Using Picolibc as your C standard library is not only supported, but in fact it is the default for the rv32imc and rv64imc platforms. However, there are a couple of things worth mentioning to make sure you don't run into problems later when building code.

Some pre-built Picolibc versions (for example, those provided by Ubuntu Linux as the `picolibc-arm-none-eabi`, `picolibc-riscv64-unknown-elf`, and `picolibc-xtensa-lx106-elf` packages) assume thread-local storage (TLS) is available at runtime, but unfortunately MicroPython modules do not support that on some architectures (namely `rv32imc` and `rv64imc`). This means that some functionalities provided by Picolibc will default to use TLS, returning an error either during compilation or during linking.

For an example on how this may affect you, the `examples/natmod/btree` example module contains a workaround to make sure `errno` works (look for `__PICOLIBC_ERRNO_FUNCTION` in the Makefile and follow the trail from there).

Further examples

See `examples/natmod/` for further examples which show many of the available features of native .mpy modules. Such features include:

- using multiple C source files
- including Python code alongside C code
- rodata and BSS data
- memory allocation
- use of floating point
- exception handling
- including external C libraries

4.11 Porting MicroPython

The MicroPython project contains several ports to different microcontroller families and architectures. The project repository has a ports directory containing a subdirectory for each supported port.

A port will typically contain definitions for multiple "boards", each of which is a specific piece of hardware that that port can run on, e.g. a development kit or device.

The minimal port is available as a simplified reference implementation of a MicroPython port. It can run on both the host system and an STM32F4xx MCU.

In general, starting a port requires:

- Setting up the toolchain (configuring Makefiles, etc).
- Implementing boot configuration and CPU initialization.
- Initialising basic drivers required for development and debugging (e.g. GPIO, UART).
- Performing the board-specific configurations.
- Implementing the port-specific modules.

4.11.1 Minimal MicroPython firmware

The best way to start porting MicroPython to a new board is by integrating a minimal MicroPython interpreter. For this walkthrough, create a subdirectory for the new port in the `ports` directory:

```
$ cd ports
$ mkdir example_port
```

The basic MicroPython firmware is implemented in the main port file, e.g `main.c`:

```
#include "py/builtin.h"
#include "py/compile.h"
#include "py/gc.h"
#include "py/mperrno.h"
#include "shared/runtime/gchelper.h"
#include "shared/runtime/pyexec.h"

// Allocate memory for the MicroPython GC heap.
static char heap[4096];

int main(int argc, char **argv) {
    // Initialise the MicroPython runtime.
    mp_cstack_init_with_sp_here(2048);
    gc_init(heap, heap + sizeof(heap));
    mp_init();

    // Start a normal REPL; will exit when ctrl-D is entered on a blank line.
    pyexec_friendly_repl();

    // Deinitialise the runtime.
    gc_sweep_all();
    mp_deinit();
    return 0;
}

// Handle uncaught exceptions (should never be reached in a correct C implementation).
void nlr_jump_fail(void *val) {
    for (;;) {
    }
}

// Do a garbage collection cycle.
void gc_collect(void) {
    gc_collect_start();
    gc_helper_collect_regs_and_stack();
    gc_collect_end();
}
```

(continues on next page)

(continued from previous page)

```
// There is no filesystem so stat'ing returns nothing.
mp_import_stat_t mp_import_stat(const char *path) {
    return MP_IMPORT_STAT_NO_EXIST;
}

// There is no filesystem so opening a file raises an exception.
mp_lexer_t *mp_lexer_new_from_file(qstr filename) {
    mp_raise_OSError(MP_ENOENT);
}
```

We also need a Makefile at this point for the port:

```
# Include the core environment definitions; this will set $(TOP).
include ../../py/mkenv.mk

# Include py core make definitions.
include $(TOP)/py/py.mk
include $(TOP)/extmod/extmod.mk

# Set CFLAGS and libraries.
CFLAGS += -I. -I$(BUILD) -I$(TOP)
LIBS += -lm

# Define the required source files.
SRC_C = \
    main.c \
    mphalport.c \
    shared/readline/readline.c \
    shared/runtime/gchelper_generic.c \
    shared/runtime/pyexec.c \
    shared/runtime/stdout_helpers.c \

# Define source files containung qstrs.
SRC_QSTR += shared/readline/readline.c shared/runtime/pyexec.c

# Define the required object files.
OBJ = $(PY_CORE_O) $(addprefix $(BUILD)/, $(SRC_C:.c=.o))

# Define the top-level target, the main firmware.
all: $(BUILD)/firmware.elf

# Define how to build the firmware.
$(BUILD)/firmware.elf: $(OBJ)
    $(ECHO) "LINK $@"
    $(Q)$(CC) $(LDFLAGS) -o $@ $^ $(LIBS)
    $(Q)$(SIZE) $@

# Include remaining core make rules.
include $(TOP)/py/mkrules.mk
```

Remember to use proper tabs to indent the Makefile.

4.11.2 MicroPython Configurations

After integrating the minimal code above, the next step is to create the MicroPython configuration files for the port. The compile-time configurations are specified in `mpconfigport.h` and additional hardware-abstraction functions, such as time keeping, in `mphalport.h`.

The following is an example of an `mpconfigport.h` file:

```
#include <stdint.h>

// Python internal features.
#define MICROPY_ENABLE_GC                       (1)
#define MICROPY_HELPER_REPL                     (1)
#define MICROPY_ERROR_REPORTING                 (MICROPY_ERROR_REPORTING_TERSE)
#define MICROPY_FLOAT_IMPL                      (MICROPY_FLOAT_IMPL_FLOAT)

// Fine control over Python builtins, classes, modules, etc.
#define MICROPY_PY_ASYNC_AWAIT                  (0)
#define MICROPY_PY_BUILTINS_SET                 (0)
#define MICROPY_PY_ATTRTUPLE                    (0)
#define MICROPY_PY_COLLECTIONS                  (0)
#define MICROPY_PY_MATH                         (0)
#define MICROPY_PY_IO                           (0)
#define MICROPY_PY_STRUCT                       (0)

// Type definitions for the specific machine.

typedef long mp_off_t;

// We need to provide a declaration/definition of alloca().
#include <alloca.h>

// Define the port's name and hardware.
#define MICROPY_HW_BOARD_NAME "example-board"
#define MICROPY_HW_MCU_NAME   "unknown-cpu"

#define MP_STATE_PORT MP_STATE_VM
```

This configuration file contains machine-specific configurations including aspects like if different MicroPython features should be enabled e.g. `#define MICROPY_ENABLE_GC (1)`. Making this Setting `(0)` disables the feature.

Other configurations include type definitions, root pointers, board name, microcontroller name etc.

Similarly, an minimal example `mphalport.h` file looks like this:

```
static inline void mp_hal_set_interrupt_char(char c) {}
```

4.11.3 Support for standard input/output

MicroPython requires at least a way to output characters, and to have a REPL it also requires a way to input characters. Functions for this can be implemented in the file `mphalport.c`, for example:

```
#include <unistd.h>
#include "py/mpconfig.h"

// Receive single character, blocking until one is available.
int mp_hal_stdin_rx_chr(void) {
```

(continues on next page)

(continued from previous page)

```
    unsigned char c = 0;
    int r = read(STDIN_FILENO, &c, 1);
    (void)r;
    return c;
}

// Send the string of given length.
void mp_hal_stdout_tx_strn(const char *str, mp_uint_t len) {
    int r = write(STDOUT_FILENO, str, len);
    (void)r;
}
```

These input and output functions have to be modified depending on the specific board API. This example uses the standard input/output stream.

4.11.4 Building and running

At this stage the directory of the new port should contain:

```
ports/example_port/
├── main.c
├── Makefile
├── mpconfigport.h
├── mphalport.c
└── mphalport.h
```

The port can now be built by running `make` (or otherwise, depending on your system).

If you are using the default compiler settings in the Makefile given above then this will create an executable called `build/firmware.elf` which can be executed directly. To get a functional REPL you may need to first configure the terminal to raw mode:

```
$ stty raw opost -echo
$ ./build/firmware.elf
```

That should give a MicroPython REPL. You can then run commands like:

```
MicroPython v1.26.0-preview on 2025-08-01; minimal with unknown-cpu
>>> def sum(n, m):
...     return n + m
...
>>> 3, 4, sum(3, 4)
(3, 4, 7)
>>>
```

Use Ctrl-D to exit, and then run `reset` to reset the terminal.

4.11.5 Adding a module to the port

To add a custom module like `myport`, first add the module definition in a file `modmyport.c`:

```
#include "py/runtime.h"

static mp_obj_t myport_info(void) {
    mp_printf(&mp_plat_print, "info about my port\n");
    return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_0(myport_info_obj, myport_info);

static const mp_rom_map_elem_t myport_module_globals_table[] = {
    { MP_OBJ_NEW_QSTR(MP_QSTR___name__), MP_OBJ_NEW_QSTR(MP_QSTR_myport) },
    { MP_ROM_QSTR(MP_QSTR_info), MP_ROM_PTR(&myport_info_obj) },
};
static MP_DEFINE_CONST_DICT(myport_module_globals, myport_module_globals_table);

const mp_obj_module_t myport_module = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&myport_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_myport, myport_module);
```

You will also need to edit the Makefile to add `modmyport.c` to the `SRC_C` list, and a new line adding the same file to `SRC_QSTR` (so qstrs are searched for in this new file), like this:

```
SRC_C = \
    main.c \
    modmyport.c \
    mphalport.c \
    ...

SRC_QSTR += modmyport.c
```

If all went correctly then, after rebuilding, you should be able to import the new module:

```
>>> import myport
>>> myport.info()
info about my port
>>>
```

4.12 MicroPython Support Tiers

MicroPython operates with a set of Support Tier levels for the various ports. Tiers 1, 2 and 3 are the main Tier levels with Tier 1 being the most mature and actively maintained. There is also Tier M for additional ports used primarily for maintenance, development and testing. These Tier levels are defined in the table below.

	Tier 1	Tier 2	Tier 3	Tier M
builds pass under CI	✓	✓	✓	✓
tests run under CI (where possible)	✓	✓	✓	✓
actively maintained	✓	✓		✓
stable Python API	✓	✓		
new features actively developed	✓	✓		
tested on hardware for releases	✓	✓		
prioritized bug reports	✓			✓
regressions warrant a patch release	✓			✓
has port-specific documentation	✓			

Lower Tiers may tick more boxes, but the above table defines the minimum requirements for a port to belong to a Tier.

Tier 1 ports:

- esp32
- mimxrt
- rp2
- samd
- stm32
- unix
- windows

Tier 2 ports:

- alif
- embed
- nrf
- psoc-edge
- renesas-ra
- webassembly
- zephyr

Tier 3 ports:

- cc3200
- esp8266
- pic16bit

Tier M ports:

- bare-arm
- minimal

- qemu

CHAPTER

FIVE

MICROPYTHON LICENSE INFORMATION

The MIT License (MIT)

Copyright (c) 2013-2017 Damien P. George, and others

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

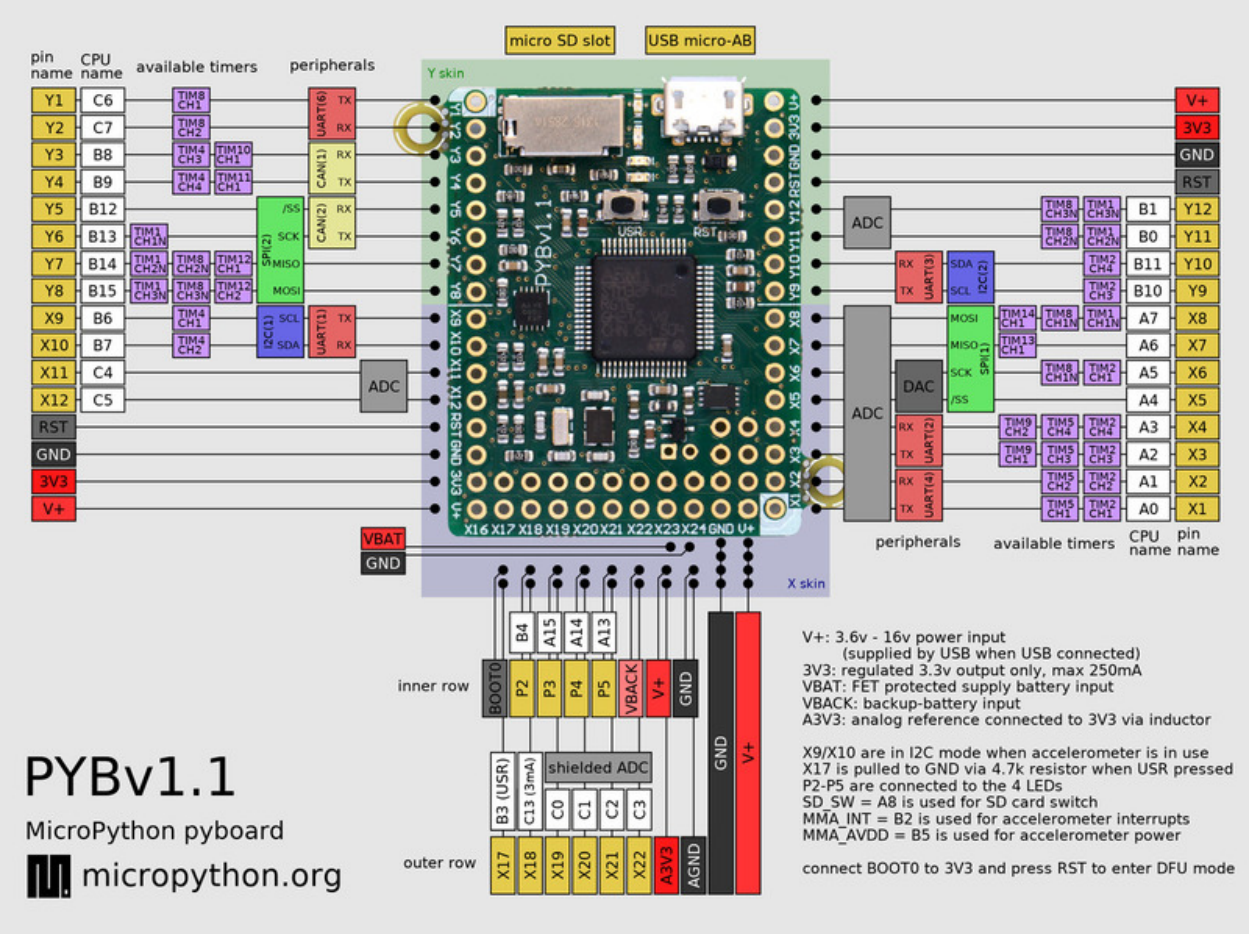
THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER

SIX

QUICK REFERENCE FOR THE PYBOARD

The below pinout is for PYBv1.1. You can also view pinouts for other versions of the pyboard: PYBv1.0 or PYBLITEv1.0-AC or PYBLITEv1.0.



Below is a quick reference for the pyboard. If it is your first time working with this board please consider reading the following sections first:

6.1 General information about the pyboard

Contents

- *General information about the pyboard*
  - *Local filesystem and SD card*
  - *Boot modes*
  - *Errors: flashing LEDs*
  - *Guide for using the pyboard with Windows*
  - *The pyboard hardware*
  - *Datasheets for the components on the pyboard*
  - *Datasheets for other components*

6.1.1 Local filesystem and SD card

There is a small internal filesystem (a drive) on the pyboard, called `/flash`, which is stored within the microcontroller's flash memory. If a micro SD card is inserted into the slot, it is available as `/sd`.

When the pyboard boots up, it needs to choose a filesystem to boot from. If there is no SD card, then it uses the internal filesystem `/flash` as the boot filesystem, otherwise, it uses the SD card `/sd`. After the boot, the current directory is set to one of the directories above.

If needed, you can prevent the use of the SD card by creating an empty file called `/flash/SKIPSD`. If this file exists when the pyboard boots up then the SD card will be skipped and the pyboard will always boot from the internal filesystem (in this case the SD card won't be mounted but you can still mount and use it later in your program using `vfs.mount`).

(Note that on older versions of the board, `/flash` is called `0:/` and `/sd` is called `1:/`).

The boot filesystem is used for 2 things: it is the filesystem from which the `boot.py` and `main.py` files are searched for, and it is the filesystem which is made available on your PC over the USB cable.

The filesystem will be available as a USB flash drive on your PC. You can save files to the drive, and edit `boot.py` and `main.py`.

Remember to eject (on Linux, unmount) the USB drive before you reset your pyboard.

6.1.2 Boot modes

If you power up normally, or press the reset button, the pyboard will boot into standard mode: the `boot.py` file will be executed first, then the USB will be configured, then `main.py` will run.

You can override this boot sequence by holding down the user switch as the board is booting up. Hold down user switch and press reset, and then as you continue to hold the user switch, the LEDs will count in binary. When the LEDs have reached the mode you want, let go of the user switch, the LEDs for the selected mode will flash quickly, and the board will boot.

The modes are:

1. Green LED only, *standard boot*: run `boot.py` then `main.py`.
2. Orange LED only, *safe boot*: don't run any scripts on boot-up.

3. Green and orange LED together, *filesystem reset*: resets the flash filesystem to its factory state, then boots in safe mode.

If your filesystem becomes corrupt, boot into mode 3 to fix it. If resetting the filesystem while plugged into your compute doesn't work, you can try doing the same procedure while the board is plugged into a USB charger, or other USB power supply without data connection.

6.1.3 Errors: flashing LEDs

There are currently 2 kinds of errors that you might see:

1. **If the red and green LEDs flash alternatively, then a Python script** (eg `main.py`) has an error. Use the REPL to debug it.
2. If all 4 LEDs cycle on and off slowly, then there was a hard fault. This cannot be recovered from and you need to do a hard reset.

6.1.4 Guide for using the pyboard with Windows

The following PDF guide gives information about using the pyboard with Windows, including setting up the serial prompt and downloading new firmware using DFU programming: PDF guide.

6.1.5 The pyboard hardware

For the pyboard:

- **v1.1**
  - PYBv1.1 schematics and layout (2.9MiB PDF)
- **v1.0**
  - PYBv1.0 schematics and layout (2.4MiB PDF)
  - PYBv1.0 metric dimensions (360KiB PDF)
  - PYBv1.0 imperial dimensions (360KiB PDF)

For the official skin modules:

- LCD32MKv1.0 schematics (194KiB PDF)
- AMPv1.0 schematics (209KiB PDF)
- LCD160CRv1.0: see *lcd160cr*

6.1.6 Datasheets for the components on the pyboard

- The microcontroller: STM32F405RGT6 (link to manufacturer's site)
- The accelerometer: Freescale MMA7660 (800kiB PDF)
- The LDO voltage regulator: Microchip MCP1802 (400kiB PDF)

6.1.7 Datasheets for other components

- The LCD display on the LCD touch-sensor skin: Newhaven Display NHD-C12832A1Z-FSW-FBW-3V3 (460KiB PDF)
- The touch sensor chip on the LCD touch-sensor skin: Freescale MPR121 (280KiB PDF)
- The digital potentiometer on the audio skin: Microchip MCP4541 (2.7MiB PDF)

6.2 MicroPython tutorial for the pyboard

This tutorial is intended to get you started with your pyboard. All you need is a pyboard and a micro-USB cable to connect it to your PC. If it is your first time, it is recommended to follow the tutorial through in the order below.

6.2.1 Introduction to the pyboard

To get the most out of your pyboard, there are a few basic things to understand about how it works.

Caring for your pyboard

Because the pyboard does not have a housing it needs a bit of care:

- Be gentle when plugging/unplugging the USB cable. Whilst the USB connector is soldered through the board and is relatively strong, if it breaks off it can be very difficult to fix.
- Static electricity can shock the components on the pyboard and destroy them. If you experience a lot of static electricity in your area (eg dry and cold climates), take extra care not to shock the pyboard. If your pyboard came in a black plastic box, then this box is the best way to store and carry the pyboard as it is an anti-static box (it is made of a conductive plastic, with conductive foam inside).

As long as you take care of the hardware, you should be okay. It's almost impossible to break the software on the pyboard, so feel free to play around with writing code as much as you like. If the filesystem gets corrupt, see below on how to reset it. In the worst case you might need to reflash the MicroPython software, but that can be done over USB.

Layout of the pyboard

The micro USB connector is on the top right, the micro SD card slot on the top left of the board. There are 4 LEDs between the SD slot and USB connector. The colours are: red on the bottom, then green, orange, and blue on the top. There are 2 switches: the right one is the reset switch, the left is the user switch.

Plugging in and powering on

The pyboard can be powered via USB. Connect it to your PC via a micro USB cable. There is only one way that the cable will fit. Once connected, the green LED on the board should flash quickly.

Powering by an external power source

The pyboard can be powered by a battery or other external power source.

Be sure to connect the positive lead of the power supply to VIN, and ground to GND. There is no polarity protection on the pyboard so you must be careful when connecting anything to VIN.

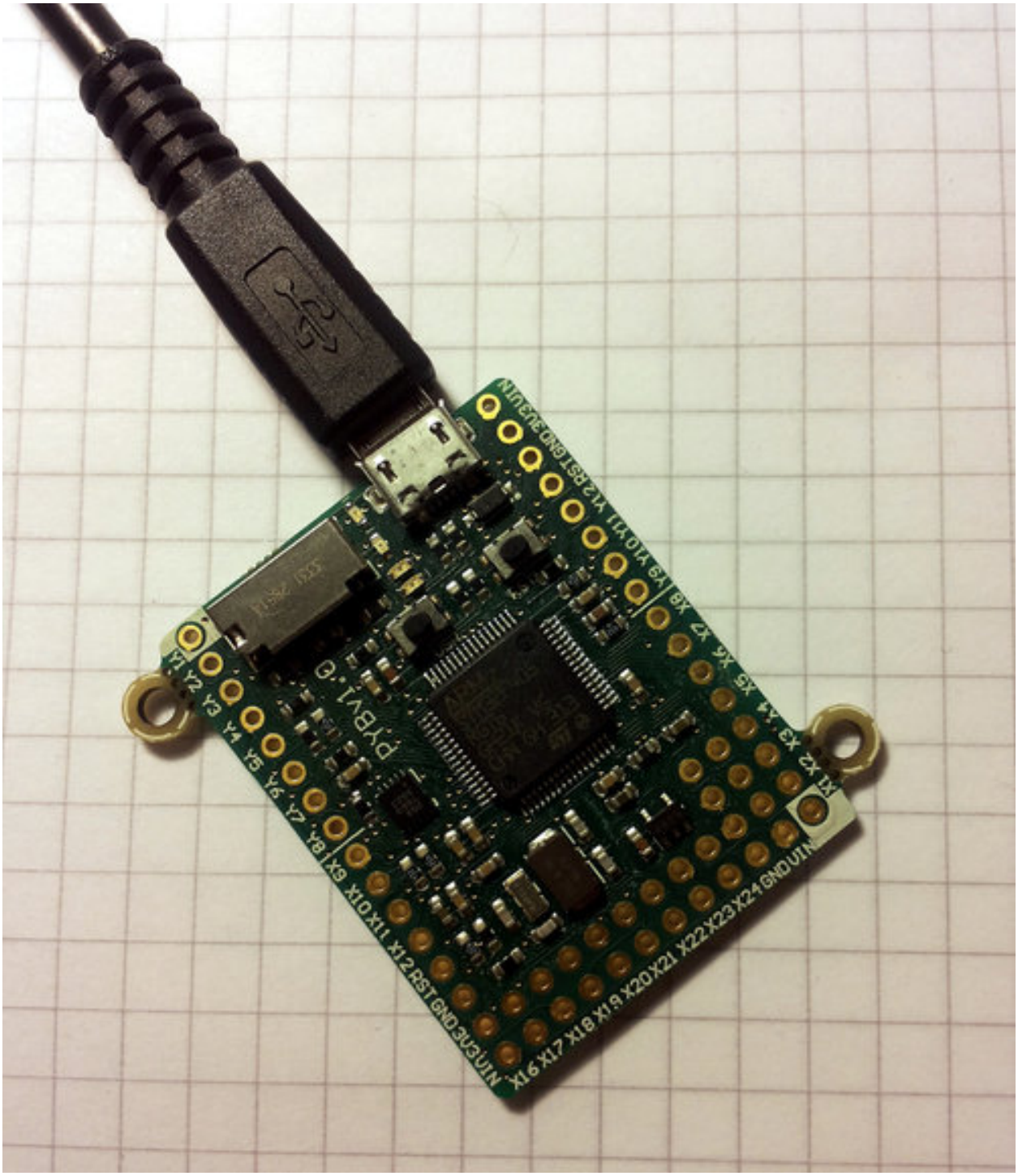
The input voltage must be between 3.6V and 10V.

6.2.2 Running your first script

Let's jump right in and get a Python script running on the pyboard. After all, that's what it's all about!

Connecting your pyboard

Connect your pyboard to your PC (Windows, Mac or Linux) with a micro USB cable. There is only one way that the cable will connect, so you can't get it wrong.



When the pyboard is connected to your PC it will power on and enter the start up process (the boot process). The green LED should light up for half a second or less, and when it turns off it means the boot process has completed.

Opening the pyboard USB drive

Your PC should now recognise the pyboard. It depends on the type of PC you have as to what happens next:

- **Windows**: Your pyboard will appear as a removable USB flash drive. Windows may automatically pop-up a window, or you may need to go there using Explorer.

  Windows will also see that the pyboard has a serial device, and it will try to automatically configure this device. If it does, cancel the process. We will get the serial device working in the next tutorial.

- **Mac**: Your pyboard will appear on the desktop as a removable disc. It will probably be called `PYBFLASH`. Click on it to open the pyboard folder.

- **Linux**: Your pyboard will appear as a removable medium. On Ubuntu it will mount automatically and pop-up a window with the pyboard folder. On other Linux distributions, the pyboard may be mounted automatically, or you may need to do it manually. At a terminal command line, type `lsblk` to see a list of connected drives, and then `mount /dev/sdb1` (replace `sdb1` with the appropriate device). You may need to be root to do this.

Okay, so you should now have the pyboard connected as a USB flash drive, and a window (or command line) should be showing the files on the pyboard drive.

The drive you are looking at is known as `/flash` by the pyboard, and should contain the following 4 files:

- **boot.py – the various configuration options for the pyboard.**
  It is executed when the pyboard boots up.
- **main.py – the Python program to be run.**
  It is executed after `boot.py`.
- **README.txt – basic information about getting started with the pyboard.**
  This provides pointers for new users and can be safely deleted.
- **pybcdc.inf – the Windows driver file to configure the serial USB device.**
  More about this in the next tutorial.

Editing `main.py`

Now we are going to write our Python program, so open the `main.py` file in a text editor. On Windows you can use notepad, or any other editor. On Mac and Linux, use your favourite text editor. With the file open you will see it contains 1 line:

```
# main.py -- put your code here!
```

This line starts with a # character, which means that it is a *comment*. Such lines will not do anything, and are there for you to write notes about your program.

Let's add 2 lines to this `main.py` file, to make it look like this:

```
# main.py -- put your code here!
import pyb
pyb.LED(4).on()
```

The first line we wrote says that we want to use the `pyb` module. This module contains all the functions and classes to control the features of the pyboard.

The second line that we wrote turns the blue LED on: it first gets the `LED` class from the `pyb` module, creates LED number 4 (the blue LED), and then turns it on.

Resetting the pyboard

To run this little script, you need to first save and close the `main.py` file, and then eject (or unmount) the pyboard USB drive. Do this like you would a normal USB flash drive.

When the drive is safely ejected/unmounted you can get to the fun part: press the RST switch on the pyboard to reset and run your script. The RST switch is the small black button just below the USB connector on the board, on the right edge.

When you press RST the green LED will flash quickly, and then the blue LED should turn on and stay on.

Congratulations! You have written and run your very first MicroPython program!

6.2.3 Getting a MicroPython REPL prompt

REPL stands for Read Evaluate Print Loop, and is the name given to the interactive MicroPython prompt that you can access on the pyboard. Using the REPL is by far the easiest way to test out your code and run commands. You can use the REPL in addition to writing scripts in `main.py`.

To use the REPL, you must connect to the serial USB device on the pyboard. How you do this depends on your operating system.

Windows

You need to install the pyboard driver to use the serial USB device. The driver is on the pyboard's USB flash drive, and is called `pybcdc.inf`.

To install this driver you need to go to Device Manager for your computer, find the pyboard in the list of devices (it should have a warning sign next to it because it's not working yet), right click on the pyboard device, select Properties, then Install Driver. You need to then select the option to find the driver manually (don't use Windows auto update), navigate to the pyboard's USB drive, and select that. It should then install. After installing, go back to the Device Manager to find the installed pyboard, and see which COM port it is (eg COM4). More comprehensive instructions can be found in the Guide for pyboard on Windows (PDF). Please consult this guide if you are having problems installing the driver.

You now need to run your terminal program. You can use HyperTerminal if you have it installed, or download the free program PuTTY: putty.exe. Using your serial program you must connect to the COM port that you found in the previous step. With PuTTY, click on "Session" in the left-hand panel, then click the "Serial" radio button on the right, then enter you COM port (eg COM4) in the "Serial Line" box. Finally, click the "Open" button.

Mac OS X

Open a terminal and run:

```
screen /dev/tty.usbmodem*
```

When you are finished and want to exit screen, type CTRL-A CTRL-\.

Linux

Open a terminal and run:

```
screen /dev/ttyACM0
```

You can also try `picocom` or `minicom` instead of screen. You may have to use `/dev/ttyACM1` or a higher number for `ttyACM`. And, you may need to give yourself the correct permissions to access this devices (eg group `uucp` or `dialout`, or use sudo).

Using the REPL prompt

Now let's try running some MicroPython code directly on the pyboard.

With your serial program open (PuTTY, screen, picocom, etc) you may see a blank screen with a flashing cursor. Press Enter and you should be presented with a MicroPython prompt, i.e. >>>. Let's make sure it is working with the obligatory test:

```
>>> print("hello pyboard!")
hello pyboard!
```

In the above, you should not type in the >>> characters. They are there to indicate that you should type the text after it at the prompt. In the end, once you have entered the text `print("hello pyboard!")` and pressed Enter, the output on your screen should look like it does above.

If you already know some python you can now try some basic commands here.

If any of this is not working you can try either a hard reset or a soft reset; see below.

Go ahead and try typing in some other commands. For example:

```
>>> pyb.LED(1).on()
>>> pyb.LED(2).on()
>>> 1 + 2
3
>>> 1 / 2
0.5
>>> 20 * 'py'
'pypypypypypypypypypypypypypypypypypypypypy'
```

Resetting the board

If something goes wrong, you can reset the board in two ways. The first is to press CTRL-D at the MicroPython prompt, which performs a soft reset. You will see a message something like

```
>>>
MPY: sync filesystems
MPY: soft reboot
Micro Python v1.0 on 2014-05-03; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>>
```

If that isn't working you can perform a hard reset (turn-it-off-and-on-again) by pressing the RST switch (the small black button closest to the micro-USB socket on the board). This will end your session, disconnecting whatever program (PuTTY, screen, etc) that you used to connect to the pyboard.

If you are going to do a hard-reset, it's recommended to first close your serial program and eject/unmount the pyboard drive.

6.2.4 Turning on LEDs and basic Python concepts

The easiest thing to do on the pyboard is to turn on the LEDs attached to the board. Connect the board, and log in as described in tutorial 1. We will start by turning and LED on in the interpreter, type the following

```
>>> myled = pyb.LED(1)
>>> myled.on()
>>> myled.off()
```

These commands turn the LED on and off.

This is all very well but we would like this process to be automated. Open the file MAIN.PY on the pyboard in your favourite text editor. Write or paste the following lines into the file. If you are new to python, then make sure you get the indentation correct since this matters!

```
led = pyb.LED(2)
while True:
    led.toggle()
    pyb.delay(1000)
```

When you save, the red light on the pyboard should turn on for about a second. To run the script, do a soft reset (CTRL-D). The pyboard will then restart and you should see a green light continuously flashing on and off. Success, the first step on your path to building an army of evil robots! When you are bored of the annoying flashing light then press CTRL-C at your terminal to stop it running.

So what does this code do? First we need some terminology. Python is an object-oriented language, almost everything in python is a *class* and when you create an instance of a class you get an *object*. Classes have *methods* associated to them. A method (also called a member function) is used to interact with or control the object.

The first line of code creates an LED object which we have then called led. When we create the object, it takes a single parameter which must be between 1 and 4, corresponding to the 4 LEDs on the board. The pyb.LED class has three important member functions that we will use: on(), off() and toggle(). The other function that we use is pyb.delay() this simply waits for a given time in milliseconds. Once we have created the LED object, the statement while True: creates an infinite loop which toggles the led between on and off and waits for 1 second.

Exercise: Try changing the time between toggling the led and turning on a different LED.

Exercise: Connect to the pyboard directly, create a pyb.LED object and turn it on using the on() method.

A Disco on your pyboard

So far we have only used a single LED but the pyboard has 4 available. Let's start by creating an object for each LED so we can control each of them. We do that by creating a list of LEDS with a list comprehension.

```
leds = [pyb.LED(i) for i in range(1,5)]
```

If you call pyb.LED() with a number that isn't 1,2,3,4 you will get an error message. Next we will set up an infinite loop that cycles through each of the LEDs turning them on and off.

```
n = 0
while True:
  n = (n + 1) % 4
```

(continues on next page)

(continued from previous page)

```
  leds[n].toggle()
  pyb.delay(50)
```

Here, n keeps track of the current LED and every time the loop is executed we cycle to the next n (the % sign is a modulus operator that keeps n between 0 and 3.) Then we access the nth LED and toggle it. If you run this you should see each of the LEDs turning on then all turning off again in sequence.

One problem you might find is that if you stop the script and then start it again that the LEDs are stuck on from the previous run, ruining our carefully choreographed disco. We can fix this by turning all the LEDs off when we initialise the script and then using a try/finally block. When you press CTRL-C, MicroPython generates a VCPInterrupt exception. Exceptions normally mean something has gone wrong and you can use a try: command to "catch" an exception. In this case it is just the user interrupting the script, so we don't need to catch the error but just tell MicroPython what to do when we exit. The finally block does this, and we use it to make sure all the LEDs are off. The full code is:

```
leds = [pyb.LED(i) for i in range(1,5)]
for l in leds:
    l.off()

n = 0
try:
   while True:
      n = (n + 1) % 4
      leds[n].toggle()
      pyb.delay(50)
finally:
    for l in leds:
        l.off()
```

The Special LEDs

The yellow and blue LEDs are special. As well as turning them on and off, you can control their intensity using the intensity() method. This takes a number between 0 and 255 that determines how bright it is. The following script makes the blue LED gradually brighter then turns it off again.

```
led = pyb.LED(4)
intensity = 0
while True:
    intensity = (intensity + 1) % 255
    led.intensity(intensity)
    pyb.delay(20)
```

You can call intensity() on LEDs 1 and 2 but they can only be off or on. 0 sets them off and any other number up to 255 turns them on.

6.2.5 Switches, callbacks and interrupts

The pyboard has 2 small switches, labelled USR and RST. The RST switch is a hard-reset switch, and if you press it then it restarts the pyboard from scratch, equivalent to turning the power off then back on.

The USR switch is for general use, and is controlled via a Switch object. To make a switch object do:

```
>>> sw = pyb.Switch()
```

Remember that you may need to type `import pyb` if you get an error that the name `pyb` does not exist.

With the switch object you can get its status:

```
>>> sw.value()
False
```

This will print `False` if the switch is not held, or `True` if it is held. Try holding the USR switch down while running the above command.

There is also a shorthand notation to get the switch status, by "calling" the switch object:

```
>>> sw()
False
```

Switch callbacks

The switch is a very simple object, but it does have one advanced feature: the `sw.callback()` function. The callback function sets up something to run when the switch is pressed, and uses an interrupt. It's probably best to start with an example before understanding how interrupts work. Try running the following at the prompt:

```
>>> sw.callback(lambda:print('press!'))
```

This tells the switch to print `press!` each time the switch is pressed down. Go ahead and try it: press the USR switch and watch the output on your PC. Note that this print will interrupt anything you are typing, and is an example of an interrupt routine running asynchronously.

As another example try:

```
>>> sw.callback(lambda:pyb.LED(1).toggle())
```

This will toggle the red LED each time the switch is pressed. And it will even work while other code is running.

To disable the switch callback, pass `None` to the callback function:

```
>>> sw.callback(None)
```

You can pass any function (that takes zero arguments) to the switch callback. Above we used the `lambda` feature of Python to create an anonymous function on the fly. But we could equally do:

```
>>> def f():
...   pyb.LED(1).toggle()
...
>>> sw.callback(f)
```

This creates a function called `f` and assigns it to the switch callback. You can do things this way when your function is more complicated than a `lambda` will allow.

Note that your callback functions must not allocate any memory (for example they cannot create a tuple or list). Callback functions should be relatively simple. If you need to make a list, make it beforehand and store it in a global variable (or make it local and close over it). If you need to do a long, complicated calculation, then use the callback to set a flag which some other code then responds to.

Technical details of interrupts

Let's step through the details of what is happening with the switch callback. When you register a function with `sw.callback()`, the switch sets up an external interrupt trigger (falling edge) on the pin that the switch is connected to. This means that the microcontroller will listen on the pin for any changes, and the following will occur:

1. When the switch is pressed a change occurs on the pin (the pin goes from low to high), and the microcontroller registers this change.
2. The microcontroller finishes executing the current machine instruction, stops execution, and saves its current state (pushes the registers on the stack). This has the effect of pausing any code, for example your running Python script.
3. The microcontroller starts executing the special interrupt handler associated with the switch's external trigger. This interrupt handler gets the function that you registered with `sw.callback()` and executes it.
4. Your callback function is executed until it finishes, returning control to the switch interrupt handler.
5. The switch interrupt handler returns, and the microcontroller is notified that the interrupt has been dealt with.
6. The microcontroller restores the state that it saved in step 2.
7. Execution continues of the code that was running at the beginning. Apart from the pause, this code does not notice that it was interrupted.

The above sequence of events gets a bit more complicated when multiple interrupts occur at the same time. In that case, the interrupt with the highest priority goes first, then the others in order of their priority. The switch interrupt is set at the lowest priority.

Further reading

For further information about using hardware interrupts see *writing interrupt handlers*.

6.2.6 The accelerometer

Here you will learn how to read the accelerometer and signal using LEDs states like tilt left and tilt right.

Using the accelerometer

The pyboard has an accelerometer (a tiny mass on a tiny spring) that can be used to detect the angle of the board and motion. There is a different sensor for each of the x, y, z directions. To get the value of the accelerometer, create a pyb.Accel() object and then call the x() method.

```
>>> accel = pyb.Accel()
>>> accel.x()
7
```

This returns a signed integer with a value between around -30 and 30. Note that the measurement is very noisy, this means that even if you keep the board perfectly still there will be some variation in the number that you measure. Because of this, you shouldn't use the exact value of the x() method but see if it is in a certain range.

We will start by using the accelerometer to turn on a light if it is not flat.

```
accel = pyb.Accel()
light = pyb.LED(3)
SENSITIVITY = 3

while True:
    x = accel.x()
    if abs(x) > SENSITIVITY:
        light.on()
    else:
        light.off()

    pyb.delay(100)
```

We create Accel and LED objects, then get the value of the x direction of the accelerometer. If the magnitude of x is bigger than a certain value `SENSITIVITY`, then the LED turns on, otherwise it turns off. The loop has a small `pyb.delay()` otherwise the LED flashes annoyingly when the value of x is close to `SENSITIVITY`. Try running this on the pyboard and tilt the board left and right to make the LED turn on and off.

Exercise: Change the above script so that the blue LED gets brighter the more you tilt the pyboard. HINT: You will need to rescale the values, intensity goes from 0-255.

Making a spirit level

The example above is only sensitive to the angle in the x direction but if we use the `y()` value and more LEDs we can turn the pyboard into a spirit level.

```
xlights = (pyb.LED(2), pyb.LED(3))
ylights = (pyb.LED(1), pyb.LED(4))

accel = pyb.Accel()
SENSITIVITY = 3

while True:
    x = accel.x()
    if x > SENSITIVITY:
        xlights[0].on()
        xlights[1].off()
    elif x < -SENSITIVITY:
        xlights[1].on()
        xlights[0].off()
    else:
        xlights[0].off()
        xlights[1].off()

    y = accel.y()
    if y > SENSITIVITY:
        ylights[0].on()
        ylights[1].off()
    elif y < -SENSITIVITY:
        ylights[1].on()
        ylights[0].off()
    else:
        ylights[0].off()
        ylights[1].off()

    pyb.delay(100)
```

We start by creating a tuple of LED objects for the x and y directions. Tuples are immutable objects in python which means they can't be modified once they are created. We then proceed as before but turn on a different LED for positive and negative x values. We then do the same for the y direction. This isn't particularly sophisticated but it does the job. Run this on your pyboard and you should see different LEDs turning on depending on how you tilt the board.

6.2.7 Safe mode and factory reset

If something goes wrong with your pyboard, don't panic! It is almost impossible for you to break the pyboard by programming the wrong thing.

The first thing to try is to enter safe mode: this temporarily skips execution of `boot.py` and `main.py` and gives default USB settings.

If you have problems with the filesystem you can do a factory reset, which restores the filesystem to its original state.

For more information, see *Reset and Boot Sequence*.

Safe mode

To enter safe mode, do the following steps:

1. Connect the pyboard to USB so it powers up.
2. Hold down the USR switch.
3. While still holding down USR, press and release the RST switch.
4. The LEDs will then cycle green to orange to green+orange and back again.
5. Keep holding down USR until *only the orange LED is lit*, and then let go of the USR switch.
6. The orange LED should flash quickly 4 times, and then turn off.
7. You are now in safe mode.

In safe mode, the `boot.py` and `main.py` files are not executed, and so the pyboard boots up with default settings. This means you now have access to the filesystem (the USB drive should appear), and you can edit `boot.py` and `main.py` to fix any problems.

Entering safe mode is temporary, and does not make any changes to the files on the pyboard.

Factory reset the filesystem

If you pyboard's filesystem gets corrupted (for example, you forgot to eject/unmount it), or you have some code in `boot.py` or `main.py` which you can't escape from, then you can reset the filesystem.

Resetting the filesystem deletes all files on the internal pyboard storage (not the SD card), and restores the files `boot.py`, `main.py`, `README.txt` and `pybcdc.inf` back to their original state.

To do a factory reset of the filesystem you follow a similar procedure as you did to enter safe mode, but release USR on green+orange:

1. Connect the pyboard to USB so it powers up.
2. Hold down the USR switch.
3. While still holding down USR, press and release the RST switch.
4. The LEDs will then cycle green to orange to green+orange and back again.
5. Keep holding down USR until *both the green and orange LEDs are lit*, and then let go of the USR switch.

6. The green and orange LEDs should flash quickly 4 times.
7. The red LED will turn on (so red, green and orange are now on).
8. The pyboard is now resetting the filesystem (this takes a few seconds).
9. The LEDs all turn off.
10. You now have a reset filesystem, and are in safe mode.
11. Press and release the RST switch to boot normally.

6.2.8 Making the pyboard act as a USB mouse

The pyboard is a USB device, and can configured to act as a mouse instead of the default USB flash drive.

To do this we must first edit the `boot.py` file to change the USB configuration. If you have not yet touched your `boot.py` file then it will look something like this:

```
# boot.py -- run on boot to configure USB and filesystem
# Put app code in main.py

import pyb
#pyb.main('main.py') # main script to run after this one
#pyb.usb_mode('VCP+MSC') # act as a serial and a storage device
#pyb.usb_mode('VCP+HID') # act as a serial device and a mouse
```

To enable the mouse mode, uncomment the last line of the file, to make it look like:

```
pyb.usb_mode('VCP+HID') # act as a serial device and a mouse
```

If you already changed your `boot.py` file, then the minimum code it needs to work is:

```
import pyb
pyb.usb_mode('VCP+HID')
```

This tells the pyboard to configure itself as a VCP (Virtual COM Port, ie serial port) and HID (human interface device, in our case a mouse) USB device when it boots up.

Eject/unmount the pyboard drive and reset it using the RST switch. Your PC should now detect the pyboard as a mouse!

Sending mouse events by hand

To get the py-mouse to do anything we need to send mouse events to the PC. We will first do this manually using the REPL prompt. Connect to your pyboard using your serial program and type the following (no need to type the # and text following it):

```
>>> hid = pyb.USB_HID()
>>> hid.send((0, 100, 0, 0)) # (button status, x-direction, y-direction, scroll)
```

Your mouse should move 100 pixels to the right! In the command above you are sending 4 pieces of information: **button status**, **x-direction**, **y-direction**, and **scroll**. The number 100 is telling the PC that the mouse moved 100 pixels in the x direction.

Let's make the mouse oscillate left and right:

```
>>> import math
>>> def osc(n, d):
...   for i in range(n):
...     hid.send((0, int(20 * math.sin(i / 10)), 0, 0))
...     pyb.delay(d)
...
>>> osc(100, 50)
```

The first argument to the function `osc` is the number of mouse events to send, and the second argument is the delay (in milliseconds) between events. Try playing around with different numbers.

Exercise: make the mouse go around in a circle.

Making a mouse with the accelerometer

Now lets make the mouse move based on the angle of the pyboard, using the accelerometer. The following code can be typed directly at the REPL prompt, or put in the `main.py` file. Here, we'll put in in `main.py` because to do that we will learn how to go into safe mode.

At the moment the pyboard is acting as a serial USB device and an HID (a mouse). So you cannot access the filesystem to edit your `main.py` file.

You also can't edit your `boot.py` to get out of HID-mode and back to normal mode with a USB drive…

To get around this we need to go into *safe mode*. This was described in the [safe mode tutorial](tut-reset), but we repeat the instructions here:

1. Hold down the USR switch.
2. While still holding down USR, press and release the RST switch.
3. The LEDs will then cycle green to orange to green+orange and back again.
4. Keep holding down USR until *only the orange LED is lit*, and then let go of the USR switch.
5. The orange LED should flash quickly 4 times, and then turn off.
6. You are now in safe mode.

In safe mode, the `boot.py` and `main.py` files are not executed, and so the pyboard boots up with default settings. This means you now have access to the filesystem (the USB drive should appear), and you can edit `main.py`. (Leave `boot.py` as-is, because we still want to go back to HID-mode after we finish editing `main.py`.)

In `main.py` put the following code:

```
import pyb

switch = pyb.Switch()
accel = pyb.Accel()
hid = pyb.USB_HID()

while not switch():
    hid.send((0, accel.x(), accel.y(), 0))
    pyb.delay(20)
```

Save your file, eject/unmount your pyboard drive, and reset it using the RST switch. It should now act as a mouse, and the angle of the board will move the mouse around. Try it out, and see if you can make the mouse stand still!

Press the USR switch to stop the mouse motion.

You'll note that the y-axis is inverted. That's easy to fix: just put a minus sign in front of the y-coordinate in the `hid.send()` line above.

Restoring your pyboard to normal

If you leave your pyboard as-is, it'll behave as a mouse every time you plug it in. You probably want to change it back to normal. To do this you need to first enter safe mode (see above), and then edit the `boot.py` file. In the `boot.py` file, comment out (put a # in front of) the line with the `VCP+HID` setting, so it looks like:

```
#pyb.usb_mode('VCP+HID') # act as a serial device and a mouse
```

Save your file, eject/unmount the drive, and reset the pyboard. It is now back to normal operating mode.

6.2.9 The Timers

The pyboard has 14 timers which each consist of an independent counter running at a user-defined frequency. They can be set up to run a function at specific intervals. The 14 timers are numbered 1 through 14, but 3 is reserved for internal use, and 5 and 6 are used for servo and ADC/DAC control. Avoid using these timers if possible.

Let's create a timer object:

```
>>> tim = pyb.Timer(4)
```

Now let's see what we just created:

```
>>> tim
Timer(4)
```

The pyboard is telling us that `tim` is attached to timer number 4, but it's not yet initialised. So let's initialise it to trigger at 10 Hz (that's 10 times per second):

```
>>> tim.init(freq=10)
```

Now that it's initialised, we can see some information about the timer:

```
>>> tim
Timer(4, prescaler=624, period=13439, mode=UP, div=1)
```

The information means that this timer is set to run at the peripheral clock speed divided by 624+1, and it will count from 0 up to 13439, at which point it triggers an interrupt, and then starts counting again from 0. These numbers are set to make the timer trigger at 10 Hz: the source frequency of the timer is 84MHz (found by running `tim.source_freq()`) so we get 84MHz / 625 / 13440 = 10Hz.

Timer counter

So what can we do with our timer? The most basic thing is to get the current value of its counter:

```
>>> tim.counter()
21504
```

This counter will continuously change, and counts up.

Timer callbacks

The next thing we can do is register a callback function for the timer to execute when it triggers (see the *switch tutorial* for an introduction to callback functions):

```
>>> tim.callback(lambda t:pyb.LED(1).toggle())
```

This should start the red LED flashing right away. It will be flashing at 5 Hz (2 toggle's are needed for 1 flash, so toggling at 10 Hz makes it flash at 5 Hz). You can change the frequency by re-initialising the timer:

```
>>> tim.init(freq=20)
```

You can disable the callback by passing it the value `None`:

```
>>> tim.callback(None)
```

The function that you pass to callback must take 1 argument, which is the timer object that triggered. This allows you to control the timer from within the callback function.

We can create 2 timers and run them independently:

```
>>> tim4 = pyb.Timer(4, freq=10)
>>> tim7 = pyb.Timer(7, freq=20)
>>> tim4.callback(lambda t: pyb.LED(1).toggle())
>>> tim7.callback(lambda t: pyb.LED(2).toggle())
```

Because the callbacks are proper hardware interrupts, we can continue to use the pyboard for other things while these timers are running.

Making a microsecond counter

You can use a timer to create a microsecond counter, which might be useful when you are doing something which requires accurate timing. We will use timer 2 for this, since timer 2 has a 32-bit counter (so does timer 5, but if you use timer 5 then you can't use the Servo driver at the same time).

We set up timer 2 as follows:

```
>>> micros = pyb.Timer(2, prescaler=83, period=0x3fffffff)
```

The prescaler is set at 83, which makes this timer count at 1 MHz. This is because the CPU clock, running at 168 MHz, is divided by 2 and then by prescaler+1, giving a frequency of 168 MHz/2/(83+1)=1 MHz for timer 2. The period is set to a large number so that the timer can count up to a large number before wrapping back around to zero. In this case it will take about 17 minutes before it cycles back to zero.

To use this timer, it's best to first reset it to 0:

```
>>> micros.counter(0)
```

and then perform your timing:

```
>>> start_micros = micros.counter()

... do some stuff ...

>>> end_micros = micros.counter()
```

6.2.10 Inline assembler

Here you will learn how to write inline assembler in MicroPython.

Note: this is an advanced tutorial, intended for those who already know a bit about microcontrollers and assembly language.

MicroPython includes an inline assembler. It allows you to write assembly routines as a Python function, and you can call them as you would a normal Python function.

Returning a value

Inline assembler functions are denoted by a special function decorator. Let's start with the simplest example:

```
@micropython.asm_thumb
def fun():
    movw(r0, 42)
```

You can enter this in a script or at the REPL. This function takes no arguments and returns the number 42. `r0` is a register, and the value in this register when the function returns is the value that is returned. MicroPython always interprets the `r0` as an integer, and converts it to an integer object for the caller.

If you run `print(fun())` you will see it print out 42.

Accessing peripherals

For something a bit more complicated, let's turn on an LED:

```
@micropython.asm_thumb
def led_on():
    movwt(r0, stm.GPIOA)
    movw(r1, 1 << 13)
    strh(r1, [r0, stm.GPIO_BSRRL])
```

This code uses a few new concepts:

- `stm` is a module which provides a set of constants for easy access to the registers of the pyboard's microcontroller. Try running `import stm` and then `help(stm)` at the REPL. It will give you a list of all the available constants.
- `stm.GPIOA` is the address in memory of the GPIOA peripheral. On the pyboard, the red LED is on port A, pin PA13.
- `movwt` moves a 32-bit number into a register. It is a convenience function that turns into 2 thumb instructions: `movw` followed by `movt`. The `movt` also shifts the immediate value right by 16 bits.
- `strh` stores a half-word (16 bits). The instruction above stores the lower 16-bits of `r1` into the memory location `r0 + stm.GPIO_BSRRL`. This has the effect of setting high all those pins on port A for which the corresponding bit in `r0` is set. In our example above, the 13th bit in `r0` is set, so PA13 is pulled high. This turns on the red LED.

Accepting arguments

Inline assembler functions can accept up to 4 arguments. If they are used, they must be named `r0`, `r1`, `r2` and `r3` to reflect the registers and the calling conventions.

Here is a function that adds its arguments:

```
@micropython.asm_thumb
def asm_add(r0, r1):
    add(r0, r0, r1)
```

This performs the computation `r0 = r0 + r1`. Since the result is put in `r0`, that is what is returned. Try `asm_add(1, 2)`, it should return 3.

Loops

We can assign labels with `label(my_label)`, and branch to them using `b(my_label)`, or a conditional branch like `bgt(my_label)`.

The following example flashes the green LED. It flashes it `r0` times.

```
@micropython.asm_thumb
def flash_led(r0):
    # get the GPIOA address in r1
    movwt(r1, stm.GPIOA)

    # get the bit mask for PA14 (the pin LED #2 is on)
    movw(r2, 1 << 14)

    b(loop_entry)

    label(loop1)

    # turn LED on
    strh(r2, [r1, stm.GPIO_BSRRL])

    # delay for a bit
    movwt(r4, 5599900)
    label(delay_on)
    sub(r4, r4, 1)
    cmp(r4, 0)
    bgt(delay_on)

    # turn LED off
    strh(r2, [r1, stm.GPIO_BSRRH])

    # delay for a bit
    movwt(r4, 5599900)
    label(delay_off)
    sub(r4, r4, 1)
    cmp(r4, 0)
    bgt(delay_off)

    # loop r0 times
    sub(r0, r0, 1)
    label(loop_entry)
```

(continues on next page)

(continued from previous page)

```
    cmp(r0, 0)
    bgt(loop1)
```

Further reading

For further information about supported instructions of the inline assembler, see the *reference documentation*.

6.2.11 Power control

pyb.wfi() is used to reduce power consumption while waiting for an event such as an interrupt. You would use it in the following situation:

```
while True:
    do_some_processing()
    pyb.wfi()
```

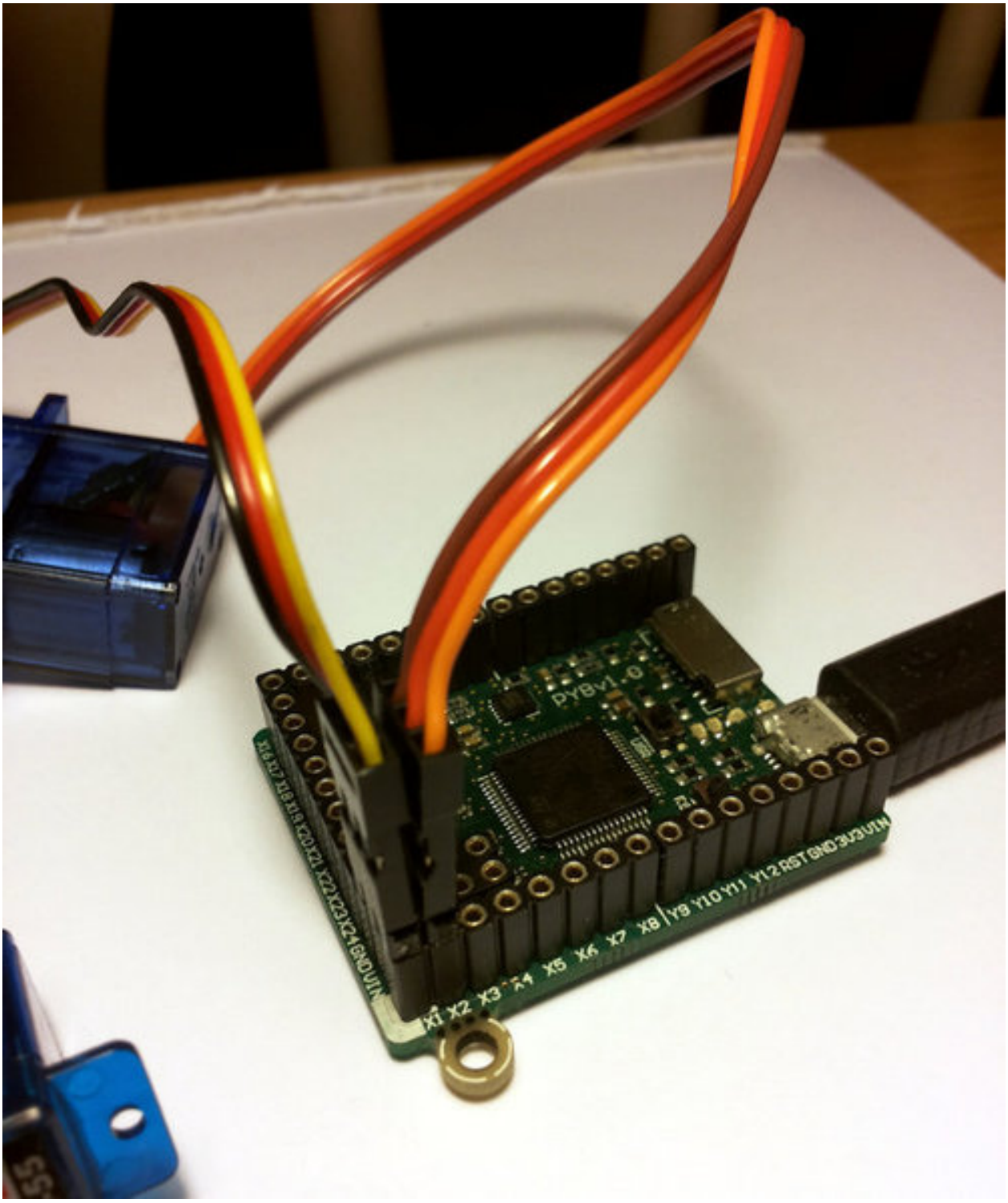
Control the frequency using *pyb.freq()*:

```
pyb.freq(30000000) # set CPU frequency to 30MHz
```

6.2.12 Tutorials requiring extra components

Controlling hobby servo motors

There are 4 dedicated connection points on the pyboard for connecting up hobby servo motors (see eg Wikipedia). These motors have 3 wires: ground, power and signal. On the pyboard you can connect them in the bottom right corner, with the signal pin on the far right. Pins X1, X2, X3 and X4 are the 4 dedicated servo signal pins.



In this picture there are male-male double adaptors to connect the servos to the header pins on the pyboard.

The ground wire on a servo is usually the darkest coloured one, either black or dark brown. The power wire will most likely be red.

The power pin for the servos (labelled VIN) is connected directly to the input power source of the pyboard. When powered via USB, VIN is powered through a diode by the 5V USB power line. Connect to USB, the pyboard can power at least 4 small to medium sized servo motors.

If using a battery to power the pyboard and run servo motors, make sure it is not greater than 6V, since this is the maximum voltage most servo motors can take. (Some motors take only up to 4.8V, so check what type you are using.)

Creating a Servo object

Plug in a servo to position 1 (the one with pin X1) and create a servo object using:

```
>>> servo1 = pyb.Servo(1)
```

To change the angle of the servo use the `angle` method:

```
>>> servo1.angle(45)
>>> servo1.angle(-60)
```

The angle here is measured in degrees, and ranges from about -90 to +90, depending on the motor. Calling `angle` without parameters will return the current angle:

```
>>> servo1.angle()
-60
```

Note that for some angles, the returned angle is not exactly the same as the angle you set, due to rounding errors in setting the pulse width.

You can pass a second parameter to the `angle` method, which specifies how long to take (in milliseconds) to reach the desired angle. For example, to take 1 second (1000 milliseconds) to go from the current position to 50 degrees, use

```
>>> servo1.angle(50, 1000)
```

This command will return straight away and the servo will continue to move to the desired angle, and stop when it gets there. You can use this feature as a speed control, or to synchronise 2 or more servo motors. If we have another servo motor (`servo2 = pyb.Servo(2)`) then we can do

```
>>> servo1.angle(-45, 2000); servo2.angle(60, 2000)
```

This will move the servos together, making them both take 2 seconds to reach their final angles.

Note: the semicolon between the 2 expressions above is used so that they are executed one after the other when you press enter at the REPL prompt. In a script you don't need to do this, you can just write them one line after the other.

Continuous rotation servos

So far we have been using standard servos that move to a specific angle and stay at that angle. These servo motors are useful to create joints of a robot, or things like pan-tilt mechanisms. Internally, the motor has a variable resistor (potentiometer) which measures the current angle and applies power to the motor proportional to how far it is from the desired angle. The desired angle is set by the width of a high-pulse on the servo signal wire. A pulse width of 1500 microsecond corresponds to the centre position (0 degrees). The pulses are sent at 50 Hz, ie 50 pulses per second.

You can also get **continuous rotation** servo motors which turn continuously clockwise or counterclockwise. The direction and speed of rotation is set by the pulse width on the signal wire. A pulse width of 1500 microseconds corresponds to a stopped motor. A pulse width smaller or larger than this means rotate one way or the other, at a given speed.

On the pyboard, the servo object for a continuous rotation motor is the same as before. In fact, using `angle` you can set the speed. But to make it easier to understand what is intended, there is another method called `speed` which sets the speed:

```
>>> servo1.speed(30)
```

`speed` has the same functionality as `angle`: you can get the speed, set it, and set it with a time to reach the final speed.

```
>>> servo1.speed()
30
>>> servo1.speed(-20)
>>> servo1.speed(0, 2000)
```

The final command above will set the motor to stop, but take 2 seconds to do it. This is essentially a control over the acceleration of the continuous servo.

A servo speed of 100 (or -100) is considered maximum speed, but actually you can go a bit faster than that, depending on the particular motor.

The only difference between the `angle` and `speed` methods (apart from the name) is the way the input numbers (angle or speed) are converted to a pulse width.

Calibration

The conversion from angle or speed to pulse width is done by the servo object using its calibration values. To get the current calibration, use

```
>>> servo1.calibration()
(640, 2420, 1500, 2470, 2200)
```

There are 5 numbers here, which have meaning:

1. Minimum pulse width; the smallest pulse width that the servo accepts.
2. Maximum pulse width; the largest pulse width that the servo accepts.
3. Centre pulse width; the pulse width that puts the servo at 0 degrees or 0 speed.
4. The pulse width corresponding to 90 degrees. This sets the conversion in the method `angle` of angle to pulse width.
5. The pulse width corresponding to a speed of 100. This sets the conversion in the method `speed` of speed to pulse width.

You can recalibrate the servo (change its default values) by using:

```
>>> servo1.calibration(700, 2400, 1510, 2500, 2000)
```

Of course, you would change the above values to suit your particular servo motor.

Fading LEDs

In addition to turning LEDs on and off, it is also possible to control the brightness of an LED using Pulse-Width Modulation (PWM), a common technique for obtaining variable output from a digital pin. This allows us to fade an LED:

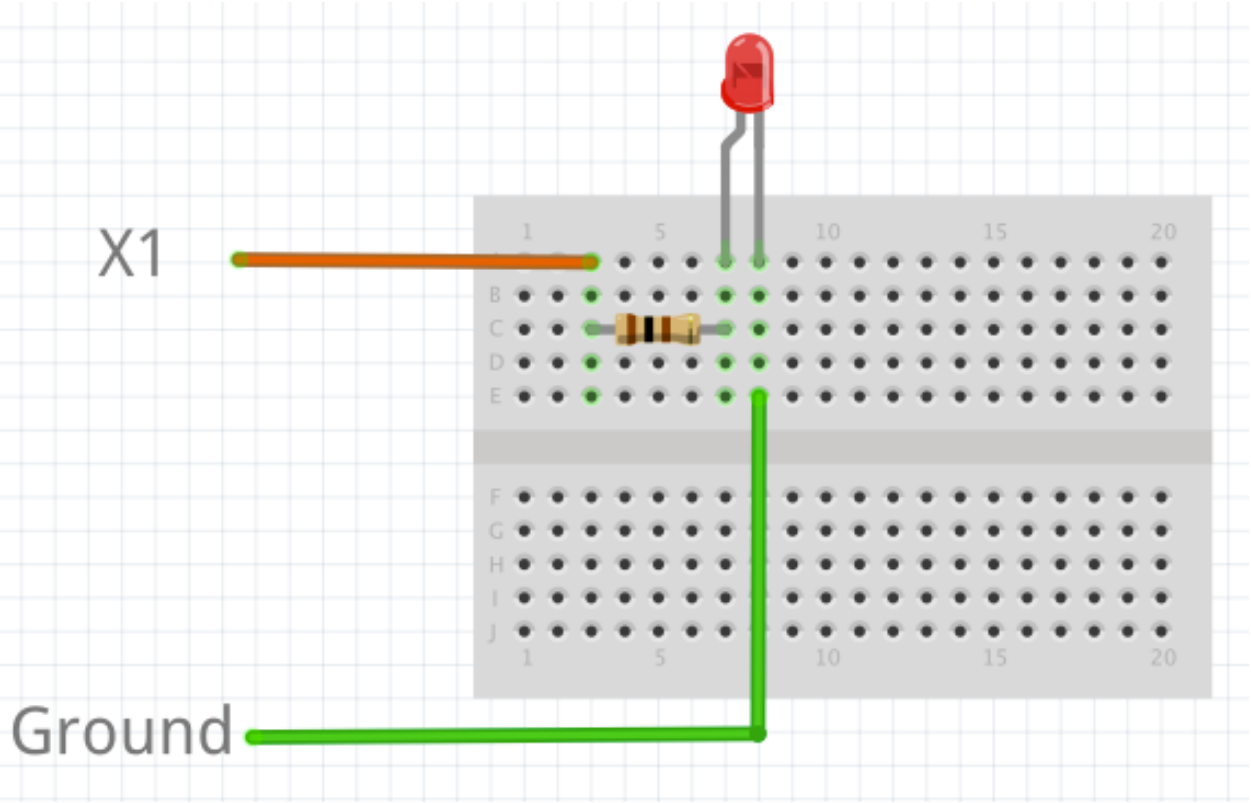
Components

You will need:

- Standard 5 or 3 mm LED
- 100 Ohm resistor
- Wires
- Breadboard (optional, but makes things easier)

Connecting Things Up

For this tutorial, we will use the `X1` pin. Connect one end of the resistor to `X1`, and the other end to the **anode** of the LED, which is the longer leg. Connect the **cathode** of the LED to ground.



Code

By examining the *Quick reference for the pyboard*, we see that `X1` is connected to channel 1 of timer 5 (`TIM5 CH1`). Therefore we will first create a `Timer` object for timer 5, then create a `TimerChannel` object for channel 1:

```
from pyb import Timer
from time import sleep

# timer 5 will be created with a frequency of 100 Hz
tim = pyb.Timer(5, freq=100)
tchannel = tim.channel(1, Timer.PWM, pin=pyb.Pin.board.X1, pulse_width=0)
```

Brightness of the LED in PWM is controlled by controlling the pulse-width, that is the amount of time the LED is on every cycle. With a timer frequency of 100 Hz, each cycle takes 0.01 second, or 10 ms.

To achieve the fading effect shown at the beginning of this tutorial, we want to set the pulse-width to a small value, then slowly increase the pulse-width to brighten the LED, and start over when we reach some maximum brightness:

```
# maximum and minimum pulse-width, which corresponds to maximum
# and minimum brightness
max_width = 200000
min_width = 20000

# how much to change the pulse-width by each step
wstep = 1500
cur_width = min_width

while True:
  tchannel.pulse_width(cur_width)

  # this determines how often we change the pulse-width. It is
  # analogous to frames-per-second
  sleep(0.01)

  cur_width += wstep

  if cur_width > max_width:
    cur_width = min_width
```

Breathing Effect

If we want to have a breathing effect, where the LED fades from dim to bright then bright to dim, then we simply need to reverse the sign of `wstep` when we reach maximum brightness, and reverse it again at minimum brightness. To do this we modify the `while` loop to be:

```
while True:
  tchannel.pulse_width(cur_width)

  sleep(0.01)

  cur_width += wstep

  if cur_width > max_width:
    cur_width = max_width
    wstep *= -1
  elif cur_width < min_width:
    cur_width = min_width
    wstep *= -1
```

Advanced Exercise

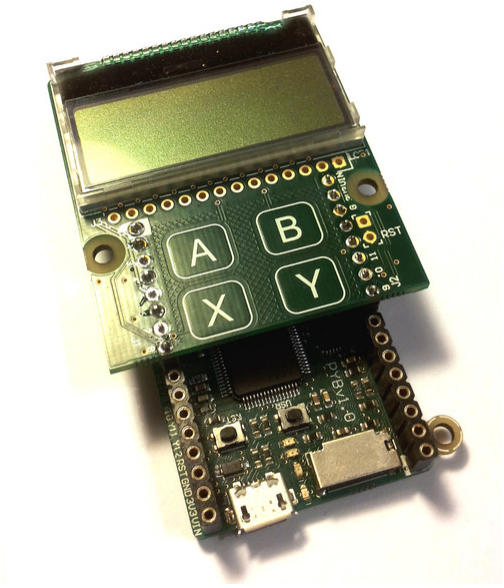
You may have noticed that the LED brightness seems to fade slowly, but increases quickly. This is because our eyes interprets brightness logarithmically (Weber's Law), while the LED's brightness changes linearly, that is by the same amount each time. How do you solve this problem? (Hint: what is the opposite of the logarithmic function?)

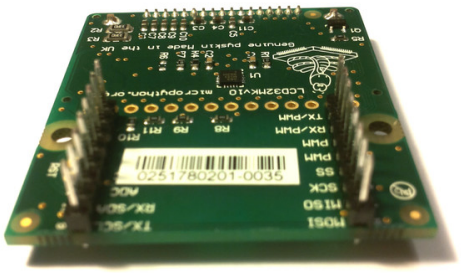
Addendum

We could have also used the digital-to-analog converter (DAC) to achieve the same effect. The PWM method has the advantage that it drives the LED with the same current each time, but for different lengths of time. This allows better control over the brightness, because LEDs do not necessarily exhibit a linear relationship between the driving current and brightness.

The LCD and touch-sensor skin

Soldering and using the LCD and touch-sensor skin.





The following video shows how to solder the headers onto the LCD skin. At the end of the video, it shows you how to correctly connect the LCD skin to the pyboard.

For circuit schematics and datasheets for the components on the skin see *The pyboard hardware*.

Using the LCD

To get started using the LCD, try the following at the MicroPython prompt. Make sure the LCD skin is attached to the pyboard as pictured at the top of this page.

```
>>> import pyb
>>> lcd = pyb.LCD('X')
>>> lcd.light(True)
>>> lcd.write('Hello MPy!\n')
```

You can make a simple animation using the code:

```
import pyb
lcd = pyb.LCD('X')
lcd.light(True)
for x in range(-80, 128):
    lcd.fill(0)
    lcd.text('Hello MPy!', x, 10, 1)
    lcd.show()
    pyb.delay(25)
```

Using the touch sensor

To read the touch-sensor data you need to use the I2C bus. The MPR121 capacitive touch sensor has address 90.

To get started, try:

```
>>> import pyb
>>> i2c = pyb.I2C(1, pyb.I2C.CONTROLLER)
>>> i2c.mem_write(4, 90, 0x5e)
>>> touch = i2c.mem_read(1, 90, 0)[0]
```

The first line above makes an I2C object, and the second line enables the 4 touch sensors. The third line reads the touch status and the `touch` variable holds the state of the 4 touch buttons (A, B, X, Y).

There is a simple driver here which allows you to set the threshold and debounce parameters, and easily read the touch status and electrode voltage levels. Copy this script to your pyboard (either flash or SD card, in the top directory or `lib/` directory) and then try:

```
>>> import pyb
>>> import mpr121
>>> m = mpr121.MPR121(pyb.I2C(1, pyb.I2C.CONTROLLER))
>>> for i in range(100):
...   print(m.touch_status())
...   pyb.delay(100)
...
```

This will continuously print out the touch status of all electrodes. Try touching each one in turn.

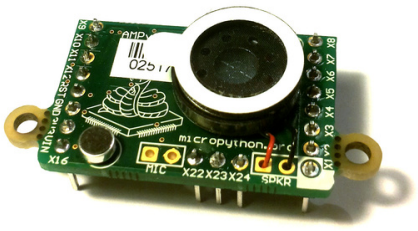
Note that if you put the LCD skin in the Y-position, then you need to initialise the I2C bus using:

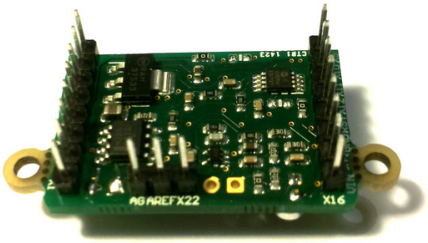
```
>>> m = mpr121.MPR121(pyb.I2C(2, pyb.I2C.CONTROLLER))
```

There is also a demo which uses the LCD and the touch sensors together, and can be found here.

The AMP audio skin

Soldering and using the AMP audio skin.





The following video shows how to solder the headers, microphone and speaker onto the AMP skin.

For circuit schematics and datasheets for the components on the skin see *The pyboard hardware*.

Example code

The AMP skin has a speaker which is connected to `DAC(1)` via a small power amplifier. The volume of the amplifier is controlled by a digital potentiometer, which is an I2C device with address 46 on the `IC2(1)` bus.

To set the volume, define the following function:

```
import pyb
def volume(val):
    pyb.I2C(1, pyb.I2C.CONTROLLER).mem_write(val, 46, 0)
```

Then you can do:

```
>>> volume(0)    # minimum volume
>>> volume(127)  # maximum volume
```

To play a sound, use the `write_timed` method of the `DAC` object. For example:

```
import math
from pyb import DAC

# create a buffer containing a sine-wave
buf = bytearray(100)
for i in range(len(buf)):
    buf[i] = 128 + int(127 * math.sin(2 * math.pi * i / len(buf)))

# output the sine-wave at 400Hz
```

(continues on next page)

(continued from previous page)

```
dac = DAC(1)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

You can also play WAV files using the Python `wave` module. You can get the wave module here and you will also need the chunk module available here. Put these on your pyboard (either on the flash or the SD card in the top-level directory). You will need an 8-bit WAV file to play, such as this one, or to convert any file you have with the command:

```
avconv -i original.wav -ar 22050 -codec pcm_u8 test.wav
```

Then you can do:

```
>>> import wave
>>> from pyb import DAC
>>> dac = DAC(1)
>>> f = wave.open('test.wav')
>>> dac.write_timed(f.readframes(f.getnframes()), f.getframerate())
```

This should play the WAV file. Note that this will read the whole file into RAM so it has to be small enough to fit in it.

To play larger wave files you will have to use the micro-SD card to store it. Also the file must be read and sent to the DAC in small chunks that will fit the RAM limit of the microcontroller. Here is an example function that can play 8-bit wave files with up to 16kHz sampling:

```
import wave
from pyb import DAC
from pyb import delay
dac = DAC(1)

def play(filename):
    f = wave.open(filename, 'r')
    total_frames = f.getnframes()
    framerate = f.getframerate()

    for position in range(0, total_frames, framerate):
        f.setpos(position)
        dac.write_timed(f.readframes(framerate), framerate)
        delay(1000)
```

This function reads one second worth of data and sends it to DAC. It then waits one second and moves the file cursor to the new position to read the next second of data in the next iteration of the for-loop. It plays one second of audio at a time every one second.

The LCD160CR skin

This tutorial shows how to get started using the LCD160CR skin.

For detailed documentation of the driver for the display see the `lcd160cr` module.

Plugging in the display

The display can be plugged directly into a pyboard (all pyboard versions are supported). You plug the display onto the top of the pyboard either in the X or Y positions. The display should cover half of the pyboard. See the picture above for how to achieve this; the left half of the picture shows the X position, and the right half shows the Y position.

Getting the driver

You can control the display directly using a power/enable pin and an I2C bus, but it is much more convenient to use the driver provided by the `lcd160cr` module. This driver is included in recent version of the pyboard firmware (see here). You can also find the driver in the GitHub repository here, and to use this version you will need to copy the file to your board, into a directory that is searched by import (usually the lib/ directory).

Once you have the driver installed you need to import it to use it:

```
import lcd160cr
```

Testing the display

There is a test program which you can use to test the features of the display, and which also serves as a basis to start creating your own code that uses the LCD. This test program is available on GitHub here. Copy it to the board over USB mass storage, or by using *MicroPython remote control: mpremote*.

To run the test from the MicroPython prompt do:

```
>>> import lcd160cr_test
```

It will then print some brief instructions. You will need to know which position your display is connected to (X or Y) and then you can run (assuming you have the display on position X):

```
>>> test_all('X')
```

Drawing some graphics

You must first create an LCD160CR object which will control the display. Do this using:

```
>>> import lcd160cr
>>> lcd = lcd160cr.LCD160CR('X')
```

This assumes your display is connected in the X position. If it's in the Y position then use `lcd = lcd160cr.LCD160CR('Y')` instead.

To erase the screen and draw a line, try:

```
>>> lcd.set_pen(lcd.rgb(255, 0, 0), lcd.rgb(64, 64, 128))
>>> lcd.erase()
>>> lcd.line(10, 10, 50, 80)
```

The next example draws random rectangles on the screen. You can copy-and-paste it into the MicroPython prompt by first pressing "Ctrl-E" at the prompt, then "Ctrl-D" once you have pasted the text.

```
from random import randint
for i in range(1000):
    fg = lcd.rgb(randint(128, 255), randint(128, 255), randint(128, 255))
    bg = lcd.rgb(randint(0, 128), randint(0, 128), randint(0, 128))
    lcd.set_pen(fg, bg)
    lcd.rect(randint(0, lcd.w), randint(0, lcd.h), randint(10, 40), randint(10, 40))
```

Using the touch sensor

The display includes a resistive touch sensor that can report the position (in pixels) of a single force-based touch on the screen. To see if there is a touch on the screen use:

```
>>> lcd.is_touched()
```

This will return either `False` or `True`. Run the above command while touching the screen to see the result.

To get the location of the touch you can use the method:

```
>>> lcd.get_touch()
```

This will return a 3-tuple, with the first entry being 0 or 1 depending on whether there is currently anything touching the screen (1 if there is), and the second and third entries in the tuple being the x and y coordinates of the current (or most recent) touch.

Directing the MicroPython output to the display

The display supports input from a UART and implements basic VT100 commands, which means it can be used as a simple, general purpose terminal. Let's set up the pyboard to redirect its output to the display.

First you need to create a UART object:

```
>>> import pyb
>>> uart = pyb.UART('XA', 115200)
```

This assumes your display is connected to position X. If it's on position Y then use `uart = pyb.UART('YA', 115200)` instead.

Now, connect the REPL output to this UART:

```
>>> pyb.repl_uart(uart)
```

From now on anything you type at the MicroPython prompt, and any output you receive, will appear on the display.

No set-up commands are required for this mode to work and you can use the display to monitor the output of any UART, not just from the pyboard. All that is needed is for the display to have power, ground and the power/enable pin driven high. Then any characters on the display's UART input will be printed to the screen. You can adjust the UART baudrate from the default of 115200 using the *set_uart_baudrate* method.

6.2.13 Tips, tricks and useful things to know

Debouncing a pin input

A pin used as input from a switch or other mechanical device can have a lot of noise on it, rapidly changing from low to high when the switch is first pressed or released. This noise can be eliminated using a capacitor (a debouncing circuit). It can also be eliminated using a simple function that makes sure the value on the pin is stable.

The following function does just this. It gets the current value of the given pin, and then waits for the value to change. The new pin value must be stable for a continuous 20ms for it to register the change. You can adjust this time (to say 50ms) if you still have noise.

```
import pyb

def wait_pin_change(pin):
    # wait for pin to change value
    # it needs to be stable for a continuous 20ms
    cur_value = pin.value()
    active = 0
    while active < 20:
        if pin.value() != cur_value:
            active += 1
        else:
            active = 0
        pyb.delay(1)
```

Use it something like this:

```
import pyb

pin_x1 = pyb.Pin('X1', pyb.Pin.IN, pyb.Pin.PULL_DOWN)
while True:
    wait_pin_change(pin_x1)
    pyb.LED(4).toggle()
```

Making a UART - USB pass through

It's as simple as:

```
import pyb
import select

def pass_through(usb, uart):
    usb.setinterrupt(-1)
    while True:
        select.select([usb, uart], [], [])
        if usb.any():
            uart.write(usb.read(256))
        if uart.any():
            usb.write(uart.read(256))

pass_through(pyb.USB_VCP(), pyb.UART(1, 9600, timeout=0))
```

6.3 General board control

See *pyb*.

```
import pyb

pyb.repl_uart(pyb.UART(1, 9600)) # duplicate REPL on UART(1)
pyb.wfi() # pause CPU, waiting for interrupt
pyb.freq() # get CPU and bus frequencies
pyb.freq(60000000) # set CPU freq to 60MHz
pyb.stop() # stop CPU, waiting for external interrupt
```

6.4 Delay and timing

Use the *time* module:

```
import time

time.sleep(1)           # sleep for 1 second
time.sleep_ms(500)      # sleep for 500 milliseconds
time.sleep_us(10)       # sleep for 10 microseconds
start = time.ticks_ms() # get value of millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start) # compute time difference
```

6.5 Internal LEDs

See *pyb.LED*.

```
from pyb import LED

led = LED(1) # 1=red, 2=green, 3=yellow, 4=blue
led.toggle()
led.on()
led.off()

# LEDs 3 and 4 support PWM intensity (0-255)
LED(4).intensity()    # get intensity
LED(4).intensity(128) # set intensity to half
```

6.6 Internal switch

See *pyb.Switch*.

```
from pyb import Switch

sw = Switch()
sw.value() # returns True or False
sw.callback(lambda: pyb.LED(1).toggle())
```

6.7 Pins and GPIO

See *pyb.Pin*.

```
from pyb import Pin

p_out = Pin('X1', Pin.OUT_PP)
p_out.high()
p_out.low()

p_in = Pin('X2', Pin.IN, Pin.PULL_UP)
p_in.value() # get value, 0 or 1
```

6.8 Servo control

See *pyb.Servo*.

```
from pyb import Servo

s1 = Servo(1) # servo on position 1 (X1, VIN, GND)
s1.angle(45) # move to 45 degrees
s1.angle(-60, 1500) # move to -60 degrees in 1500ms
s1.speed(50) # for continuous rotation servos
```

6.9 External interrupts

See *pyb.ExtInt*.

```
from pyb import Pin, ExtInt

callback = lambda e: print("intr")
ext = ExtInt(Pin('Y1'), ExtInt.IRQ_RISING, Pin.PULL_NONE, callback)
```

6.10 Timers

See *pyb.Timer*.

```
from pyb import Timer

tim = Timer(1, freq=1000)
tim.counter() # get counter value
tim.freq(0.5) # 0.5 Hz
tim.callback(lambda t: pyb.LED(1).toggle())
```

6.11 RTC (real time clock)

See *pyb.RTC*

```
from pyb import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 0, 1, 12, 48, 0)) # set a specific date and
                                             # time, eg. 2017/8/23 1:12:48
                                             # the day-of-week value is ignored
rtc.datetime() # get date and time
```

6.12 PWM (pulse width modulation)

See *pyb.Pin* and *pyb.Timer*.

```
from pyb import Pin, Timer

p = Pin('X1') # X1 has TIM2, CH1
tim = Timer(2, freq=1000)
ch = tim.channel(1, Timer.PWM, pin=p)
ch.pulse_width_percent(50)
```

6.13 ADC (analog to digital conversion)

See *pyb.Pin* and *pyb.ADC*.

```
from pyb import Pin, ADC

adc = ADC(Pin('X19'))
adc.read() # read value, 0-4095
```

6.14 DAC (digital to analog conversion)

See *pyb.Pin* and *pyb.DAC*.

```
from pyb import Pin, DAC

dac = DAC(Pin('X5'))
dac.write(120) # output between 0 and 255
```

6.15 UART (serial bus)

See *pyb.UART*.

```
from pyb import UART

uart = UART(1, 9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

6.16 SPI bus

See *pyb.SPI*.

```
from pyb import SPI

spi = SPI(1, SPI.CONTROLLER, baudrate=200000, polarity=1, phase=0)
spi.send('hello')
spi.recv(5) # receive 5 bytes on the bus
spi.send_recv('hello') # send and receive 5 bytes
```

6.17 I2C bus

Hardware I2C is available on the X and Y halves of the pyboard via `I2C('X')` and `I2C('Y')`. Alternatively pass in the integer identifier of the peripheral, eg `I2C(1)`. Software I2C is also available by explicitly specifying the `scl` and `sda` pins instead of the bus name. For more details see *machine.I2C*.

```
from machine import I2C

i2c = I2C('X', freq=400000)                 # create hardware I2c object
i2c = I2C(scl='X1', sda='X2', freq=100000)  # create software I2C object

i2c.scan()                            # returns list of peripheral addresses
i2c.writeto(0x42, 'hello')            # write 5 bytes to peripheral with address 0x42
i2c.readfrom(0x42, 5)                 # read 5 bytes from peripheral

i2c.readfrom_mem(0x42, 0x10, 2)       # read 2 bytes from peripheral 0x42, peripheral␣
↪memory 0x10
i2c.writeto_mem(0x42, 0x10, 'xy')     # write 2 bytes to peripheral 0x42, peripheral␣
↪memory 0x10
```

Note: for legacy I2C support see *pyb.I2C*.

6.18 I2S bus

See *machine.I2S*.

```
from machine import I2S, Pin

i2s = I2S(2, sck=Pin('Y6'), ws=Pin('Y5'), sd=Pin('Y8'), mode=I2S.TX, bits=16,␣
↪format=I2S.STEREO, rate=44100, ibuf=40000) # create I2S object
i2s.write(buf)              # write buffer of audio samples to I2S device

i2s = I2S(1, sck=Pin('X5'), ws=Pin('X6'), sd=Pin('Y4'), mode=I2S.RX, bits=16,␣
↪format=I2S.MONO, rate=22050, ibuf=40000) # create I2S object
i2s.readinto(buf)           # fill buffer with audio samples from I2S device
```

The I2S class is currently available as a Technical Preview. During the preview period, feedback from users is encouraged. Based on this feedback, the I2S class API and implementation may be changed.

PYBv1.0/v1.1 has one I2S bus with id=2. PYBD-SFxW has two I2S buses with id=1 and id=2. I2S is shared with SPI.

6.19 CAN bus (controller area network)

See *pyb.CAN*.

```
from pyb import CAN

can = CAN(1, CAN.LOOPBACK)
can.setfilter(0, CAN.LIST16, 0, (123, 124, 125, 126))
can.send('message!', 123)   # send a message with id 123
can.recv(0)                 # receive message on FIFO 0
```

6.20 Internal accelerometer

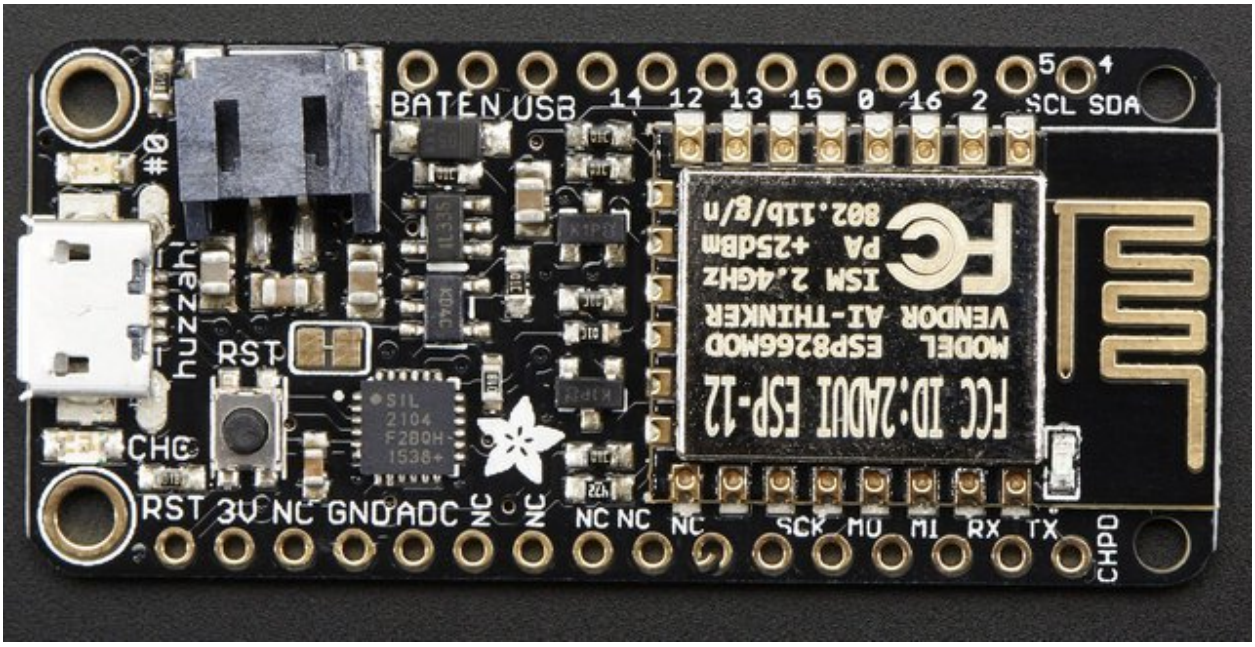
See *pyb.Accel*.

```
from pyb import Accel

accel = Accel()
print(accel.x(), accel.y(), accel.z(), accel.tilt())
```

CHAPTER

SEVEN

QUICK REFERENCE FOR THE ESP8266



The Adafruit Feather HUZZAH board (image attribution: Adafruit).

Below is a quick reference for ESP8266-based boards. If it is your first time working with this board please consider reading the following sections first:

7.1 General information about the ESP8266 port

ESP8266 is a popular WiFi-enabled System-on-Chip (SoC) by Espressif Systems.

7.1.1 Multitude of boards

There is a multitude of modules and boards from different sources which carry the ESP8266 chip. MicroPython tries to provide a generic port which would run on as many boards/modules as possible, but there may be limitations. Adafruit Feather HUZZAH board is taken as a reference board for the port (for example, testing is performed on it). If you have another board, please make sure you have a datasheet, schematics and other reference materials for your board handy to look up various aspects of your board functioning.

To make a generic ESP8266 port and support as many boards as possible, the following design and implementation decision were made:

- GPIO pin numbering is based on ESP8266 chip numbering, not some "logical" numbering of a particular board. Please have the manual/pin diagram of your board at hand to find correspondence between your board pins and actual ESP8266 pins. We also encourage users of various boards to share this mapping via MicroPython forum, with the idea to collect community-maintained reference materials eventually.
- All pins which make sense to support, are supported by MicroPython (for example, pins which are used to connect SPI flash are not exposed, as they're unlikely useful for anything else, and operating on them will lead to board lock-up). However, any particular board may expose only subset of pins. Consult your board reference manual.
- Some boards may lack external pins/internal connectivity to support ESP8266 deepsleep mode.

7.1.2 Technical specifications and SoC datasheets

The datasheets and other reference material for ESP8266 chip are available from the vendor site: http://bbs.espressif.com/viewtopic.php?f=67&t=225 . They are the primary reference for the chip technical specifications, capabilities, operating modes, internal functioning, etc.

For your convenience, some of technical specifications are provided below:

- Architecture: Xtensa lx106
- CPU frequency: 80MHz overclockable to 160MHz
- Total RAM available: 96KB (part of it reserved for system)
- BootROM: 64KB
- Internal FlashROM: None
- External FlashROM: code and data, via SPI Flash. Normal sizes 512KB-4MB.
- GPIO: 16 + 1 (GPIOs are multiplexed with other functions, including external FlashROM, UART, deep sleep wake-up, etc.)
- UART: One RX/TX UART (no hardware handshaking), one TX-only UART.
- SPI: 2 SPI interfaces (one used for FlashROM).
- I2C: No native external I2C (bitbang implementation available on any pins).
- I2S: 1.
- Programming: using BootROM bootloader from UART. Due to external FlashROM and always-available BootROM bootloader, ESP8266 is not brickable.

7.1.3 Scarcity of runtime resources

ESP8266 has very modest resources (first of all, RAM memory). So, please avoid allocating too big container objects (lists, dictionaries) and buffers. There is also no full-fledged OS to keep track of resources and automatically clean them up, so that's the task of a user/user application: please be sure to close open files, sockets, etc. as soon as possible after use.

7.1.4 Boot process

See *Reset and Boot Sequence*.

7.1.5 Known Issues

Real-time clock

RTC in ESP8266 has very bad accuracy, drift may be seconds per minute. As a workaround, to measure short enough intervals you can use `time.time()`, etc. functions, and for wall clock time, synchronize from the net using included `ntptime.py` module.

Due to limitations of the ESP8266 chip the internal real-time clock (RTC) will overflow every 7:45h. If a long-term working RTC time is required then `time()` or `localtime()` must be called at least once within 7 hours. MicroPython will then handle the overflow.

Simultaneous operation of STA_IF and AP_IF

Simultaneous operation of STA_IF and AP_IF interfaces is supported.

However, due to restrictions of the hardware, there may be performance issues in the AP_IF, if the STA_IF is not connected and searching. An application should manage these interfaces and for example deactivate the STA_IF in environments where only the AP_IF is used.

Sockets and WiFi buffers overflow

Socket instances remain active until they are explicitly closed. This has two consequences. Firstly they occupy RAM, so an application which opens sockets without closing them may eventually run out of memory. Secondly not properly closed socket can cause the low-level part of the vendor WiFi stack to emit `Lmac` errors. This occurs if data comes in for a socket and is not processed in a timely manner. This can overflow the WiFi stack input queue and lead to a deadlock. The only recovery is by a hard reset.

The above may also happen after an application terminates and quits to the REPL for any reason including an exception. Subsequent arrival of data provokes the failure with the above error message repeatedly issued. So, sockets should be closed in any case, regardless whether an application terminates successfully or by an exception, for example using try/finally:

```
sock = socket(...)
try:
    # Use sock
finally:
    sock.close()
```

SSL/TLS limitations

ESP8266 uses axTLS library, which is one of the smallest TLS libraries with compatible licensing. However, it also has some known issues/limitations:

1. No support for Diffie-Hellman (DH) key exchange and Elliptic-curve cryptography (ECC). This means it can't work with sites which require the use of these features (it works ok with the typical sites that use RSA certificates).
2. Half-duplex communication nature. axTLS uses a single buffer for both sending and receiving, which leads to considerable memory saving and works well with protocols like HTTP. But there may be problems with protocols which don't follow classic request-response model.

Besides axTLS's own limitations, the configuration used for MicroPython is highly optimized for code size, which leads to additional limitations (these may be lifted in the future):

3. Optimized RSA algorithms are not enabled, which may lead to slow SSL handshakes.
4. Session Reuse is not enabled, which means every connection must undergo the full, expensive SSL handshake.

Besides axTLS specific limitations described above, there's another generic limitation with usage of TLS on the low-memory devices:

5. The TLS standard specifies the maximum length of the TLS record (unit of TLS communication, the entire record must be buffered before it can be processed) as 16KB. That's almost half of the available ESP8266 memory, and inside a more or less advanced application would be hard to allocate due to memory fragmentation issues. As a compromise, a smaller buffer is used, with the idea that the most interesting usage for SSL would be accessing various REST APIs, which usually require much smaller messages. The buffers size is on the order of 5KB, and is adjusted from time to time, taking as a reference being able to access https://google.com . The smaller buffer however means that some sites can't be accessed using it, and it's not possible to stream large amounts of data. axTLS does have support for TLS's Max Fragment Size extension, but no HTTPS website does, so use of the extension is really only effective for local communication with other devices.

There are also some not implemented features specifically in MicroPython's `ssl` module based on axTLS:

6. Certificates are not validated (this makes connections susceptible to man-in-the-middle attacks).
7. There is no support for client certificates (scheduled to be fixed in 1.9.4 release).

7.2 MicroPython tutorial for ESP8266

This tutorial is intended to get you started using MicroPython on the ESP8266 system-on-a-chip. If it is your first time it is recommended to follow the tutorial through in the order below. Otherwise the sections are mostly self contained, so feel free to skip to those that interest you.

The tutorial does not assume that you know Python, but it also does not attempt to explain any of the details of the Python language. Instead it provides you with commands that are ready to run, and hopes that you will gain a bit of Python knowledge along the way. To learn more about Python itself please refer to https://www.python.org.

7.2.1 Getting started with MicroPython on the ESP8266

Using MicroPython is a great way to get the most of your ESP8266 board. And vice versa, the ESP8266 chip is a great platform for using MicroPython. This tutorial will guide you through setting up MicroPython, getting a prompt, using WebREPL, connecting to the network and communicating with the Internet, using the hardware peripherals, and controlling some external components.

Let's get started!

Requirements

The first thing you need is a board with an ESP8266 chip. The MicroPython software supports the ESP8266 chip itself and any board should work. The main characteristic of a board is how much flash it has, how the GPIO pins are connected to the outside world, and whether it includes a built-in USB-serial converter to make the UART available to your PC.

MicroPython is distributed as several firmware variants to suit the amount of flash on your board:

- The standard build (`ESP8266_GENERIC`) targets boards with **2MiB or more** of flash. This is the recommended build and the best choice for most users.
- The `FLASH_1M` variant is for boards with **1MiB** of flash. It removes asyncio and FAT-filesystem support, as well as some modules from micropython-lib.
- The `FLASH_2M_ROMFS` variant targets **2MiB** boards and reserves part of the flash for a read-only ROMFS filesystem.
- The `FLASH_512K` variant is for boards with only **512kiB** of flash. It is highly limited compared to the other builds: there is no filesystem support, and so features that depend on it won't work (WebREPL, mip, etc.). It also drops framebuffer support, some Python language features, and has less detailed error messages. This variant is mainly of interest to users who build from source and fine-tune parameters for their particular application.

The minimum recommended flash size is therefore 1MiB, with 2MiB or more giving the best experience.

Names of pins will be given in this tutorial using the chip names (eg GPIO0) and it should be straightforward to find which pin this corresponds to on your particular board.

Powering the board

If your board has a USB connector on it then most likely it is powered through this when connected to your PC. Otherwise you will need to power it directly. Please refer to the documentation for your board for further details.

Getting the firmware

The first thing you need to do is download the most recent MicroPython firmware .bin file to load onto your ESP8266 device. You can download it from the ESP8266 download page.

The download page offers the firmware variants described above. Pick the one that matches your board's flash size:

Firmware file	Board flash size
ESP8266_GENERIC-<date>-<version>.bin	2MiB-4MiB, 8MiB-16MiB** (the standard build)
ESP8266_GENERIC-FLASH_1M-<date>-<version>.bin	1MiB
ESP8266_GENERIC-FLASH_2M_ROMFS-<date>-<version>.bin	2MiB-4MiB, 8MiB-16MiB** (includes a ROMFS)
ESP8266_GENERIC-FLASH_512K-<date>-<version>.bin	512kiB

** Boards with 8MiB or 16MiB of flash can use the standard build, but require an manual step to set up the RF calibration data, see *Boards with more than 4MB of flash* below.

For each variant the page lists *release* builds and *preview* builds. If you are just starting with MicroPython, choose the latest release build. If you are an experienced user who would like to follow development closely and help with testing new features, the preview builds are automatic builds of the development branch.

Throughout the rest of this tutorial the example commands use the standard `ESP8266_GENERIC` firmware; substitute the exact filename you downloaded.

Deploying the firmware

Once you have the MicroPython firmware (compiled code), you need to load it onto your ESP8266 device. There are two main steps to do this: first you need to put your device in boot-loader mode, and second you need to copy across the firmware. The exact procedure for these steps is highly dependent on the particular board and you will need to refer to its documentation for details.

If you have a board that has a USB connector, a USB-serial converter, and has the DTR and RTS pins wired in a special way then deploying the firmware should be easy as all steps can be done automatically. Boards that have such features include the Adafruit Feather HUZZAH and NodeMCU boards.

If you do not have such a board, you need keep GPIO0 pulled to ground and reset the device by pulling the reset pin to ground and releasing it again to enter programming mode.

For best results it is recommended to first erase the entire flash of your device before putting on new MicroPython firmware.

Currently we only support esptool.py to copy across the firmware. You can find this tool here: https://github.com/espressif/esptool/, or install it using pip:

```
pip install esptool
```

Any other flashing program should work, so feel free to try them out or refer to the documentation for your board to see its recommendations.

Using esptool.py you can erase the flash with the command:

```
esptool.py --port /dev/ttyUSB0 erase_flash
```

And then deploy the new firmware using:

```
esptool.py --port /dev/ttyUSB0 --baud 460800 write_flash --flash_size=detect 0␣
↪ESP8266_GENERIC-20260406-v1.28.0.bin
```

You might need to change the "port" setting to something else relevant for your PC. You may also need to reduce the baudrate if you get errors when flashing (eg down to 115200). The filename of the firmware should also match the file that you have.

The `--flash_size=detect` option tells esptool.py to read the flash size from the chip's JEDEC ID. MicroPython itself also autodetects the flash size at runtime for chips up to **4MB**, so a single firmware build adapts to the actual flash on your board without any extra configuration. The filesystem is automatically sized to use all of the available flash.

The ESP8266 needs a small block of RF calibration data, known as `esp_init_data`, near the end of the flash before WiFi will start. On first boot MicroPython checks this region and, if it is blank (for example because you just ran `erase_flash`), it automatically writes the default calibration data for you. For boards with **4MB of flash or less**, no manual step is needed. See *Boards with more than 4MB of flash* below for boards larger than 4MB.

For some boards with a particular FlashROM configuration (e.g. some variants of a NodeMCU board) you may need to manually set a compatible SPI Flash Mode. You'd usually pick the fastest option that is compatible with your device, but the `-fm dout` option (the slowest option) should have the best compatibility:

```
esptool.py --port /dev/ttyUSB0 --baud 460800 write_flash --flash_size=detect -fm dout␣
↪0 ESP8266_GENERIC-20260406-v1.28.0.bin
```

If the above commands run without error then MicroPython should be installed on your board!

If you pulled GPIO0 manually to ground to enter programming mode, release it now and reset the device by again pulling the reset pin to ground for a short duration.

Boards with more than 4MB of flash

Boards with **8MB or 16MB** of flash need one extra step. The flash routines built into the ESP8266 boot ROM (which MicroPython uses to write the calibration data on first boot) cannot address flash beyond 4MB: the ROM reads the chip's device ID but clamps the reported size to 4MB, so any access above that offset fails. As a result MicroPython can only write the `esp_init_data` RF calibration block automatically within the first 4MB of flash. So after a full flash erase MicroPython cannot place the calibration data at the correct end-of-flash address by itself. Without it the WiFi subsystem will not start (typically showing up as a continuous reset loop or `rf_cal` errors). For more background on this ROM limitation see ESP8266 16MB Flash Handling.

To fix this, flash the `esp_init_data_default.bin` file (shipped with the Espressif NONOS SDK) to the calibration address, which is the flash size minus `0x4000`, see table below.

You can download `esp_init_data_default.bin` from the Espressif repository (open the link and click "View raw" to download the file): https://github.com/espressif/ESP8266_AT/blob/master/bin/esp_init_data_default.bin

The calibration addresses are:

Flash size	`esp_init_data` address
8MB	`0x7FC000`
16MB	`0xFFC000`

The full procedure for a 16MB board is:

```
# 1. Make sure esptool is up to date
pip install --upgrade esptool

# 2. Erase the flash
esptool.py --port /dev/ttyUSB0 --baud 460800 erase_flash

# 3. Flash the RF calibration blob at the end of flash (16MB example)
```

(continues on next page)

(continued from previous page)

```
esptool.py --port /dev/ttyUSB0 write_flash 0xFFC000 esp_init_data_default.bin

# 4. Flash MicroPython, telling esptool the real flash size
esptool.py --port /dev/ttyUSB0 --baud 460800 \
    write_flash -fm dio --flash_size 16MB 0 ESP8266_GENERIC-20260406-v1.28.0.bin

# 5. Check detected flash size in the REPL
mpremote exec "import esp;print(f'Detected flash: {esp.flash_size():_}')"
```

For an 8MB board, use `0x7FC000` in step 3 and `--flash_size 8MB` in step 4.

The filesystem is sized automatically from the detected flash size, so it will use all of the available space once the board boots.

Serial prompt

Once you have the firmware on the device you can access the REPL (Python prompt) over UART0 (GPIO1=TX, GPIO3=RX), which might be connected to a USB-serial converter, depending on your board. The baudrate is 115200. The next part of the tutorial will discuss the prompt in more detail.

WiFi

After a fresh install and boot the device configures itself as a WiFi access point (AP) that you can connect to. The ESSID is of the form MicroPython-xxxxxx where the x's are replaced with part of the MAC address of your device (so will be the same every time, and most likely different for all ESP8266 chips). The password for the WiFi is micropythoN (note the upper-case N). Its IP address will be 192.168.4.1 once you connect to its network. WiFi configuration will be discussed in more detail later in the tutorial.

Troubleshooting installation problems

If you experience problems during flashing or with running firmware immediately after it, here are troubleshooting recommendations:

- Be aware of and try to exclude hardware problems. There are 2 common problems: bad power source quality and worn-out/defective FlashROM. Speaking of power source, not just raw amperage is important, but also low ripple and noise/EMI in general. If you experience issues with self-made or wall-wart style power supply, try USB power from a computer. Unearthed power supplies are also known to cause problems as they source of increased EMI (electromagnetic interference) - at the very least, and may lead to electrical devices breakdown. So, you are advised to avoid using unearthed power connections when working with ESP8266 and other boards. In regard to FlashROM hardware problems, there are independent (not related to MicroPython in any way) reports (e.g.) that on some ESP8266 modules, FlashROM can be programmed as little as 20 times before programming errors occur. This is *much* less than 100,000 programming cycles cited for FlashROM chips of a type used with ESP8266 by reputable vendors, which points to either production rejects, or second-hand worn-out flash chips to be used on some (apparently cheap) modules/boards. You may want to use your best judgement about source, price, documentation, warranty, post-sales support for the modules/boards you purchase.
- The flashing instructions above use flashing speed of 460800 baud, which is good compromise between speed and stability. However, depending on your module/board, USB-UART converter, cables, host OS, etc., the above baud rate may be too high and lead to errors. Try a more common 115200 baud rate instead in such cases.
- If lower baud rate didn't help, you may want to try a different version of esptool.py, which may use a different programming algorithm.
- The `--flash_size` option in the commands above is mandatory. Omitting it will lead to a corrupted firmware.

- To catch incorrect flash content (e.g. from a defective sector on a chip), add `--verify` switch to the commands above.
- Additionally, you can check the firmware integrity from a MicroPython REPL prompt (assuming you were able to flash it and `--verify` option doesn't report errors):

```
import esp
esp.check_fw()
```

  If the last output value is True, the firmware is OK. Otherwise, it's corrupted and need to be reflashed correctly.
- If you experience any issues with another flashing application (not esptool.py), try esptool.py, it is a generally accepted flashing application in the ESP8266 community.
- If you still experience problems with even flashing the firmware, please refer to esptool.py project page, https://github.com/espressif/esptool for additional documentation and bug tracker where you can report problems.
- If you are able to flash firmware, but `--verify` option or `esp.check_fw()` return errors even after multiple retries, you may have a defective FlashROM chip, as explained above.

7.2.2 Getting a MicroPython REPL prompt

REPL stands for Read Evaluate Print Loop, and is the name given to the interactive MicroPython prompt that you can access on the ESP8266. Using the REPL is by far the easiest way to test out your code and run commands.

There are two ways to access the REPL: either via a wired connection through the UART serial port, or via WiFi.

REPL over the serial port

The REPL is always available on the UART0 serial peripheral, which is connected to the pins GPIO1 for TX and GPIO3 for RX. The baudrate of the REPL is 115200. If your board has a USB-serial converter on it then you should be able to access the REPL directly from your PC. Otherwise you will need to have a way of communicating with the UART.

To access the prompt over USB-serial you need to use a terminal emulator program. On Windows TeraTerm is a good choice, on Mac you can use the built-in `screen` program, and Linux has `picocom` and `minicom`. Of course, there are many other terminal programs that will work, so pick your favourite!

For example, on Linux you can try running:

```
picocom /dev/ttyUSB0 -b115200
```

Once you have made the connection over the serial port you can test if it is working by hitting enter a few times. You should see the Python REPL prompt, indicated by >>>.

WebREPL - a prompt over WiFi

WebREPL allows you to use the Python prompt over WiFi, connecting through a browser. The latest versions of Firefox and Chrome are supported.

For your convenience, WebREPL client is hosted at http://micropython.org/webrepl. Alternatively, you can install it locally from the GitHub repository https://github.com/micropython/webrepl.

Before connecting to WebREPL, you should set a password and enable it via a normal serial connection. Initial versions of MicroPython for ESP8266 came with WebREPL automatically enabled on the boot and with the ability to set a password via WiFi on the first connection, but as WebREPL was becoming more widely known and popular, the initial setup has switched to a wired connection for improved security:

```
import webrepl_setup
```

Follow the on-screen instructions and prompts. To make any changes active, you will need to reboot your device.

To use WebREPL connect your computer to the ESP8266's access point (MicroPython-xxxxxx, see the previous section about this). If you have already reconfigured your ESP8266 to connect to a router then you can skip this part.

Once you are on the same network as the ESP8266 you click the "Connect" button (if you are connecting via a router then you may need to change the IP address, by default the IP address is correct when connected to the ESP8266's access point). If the connection succeeds then you should see a password prompt.

Once you type the password configured at the setup step above, press Enter once more and you should get a prompt looking like >>>. You can now start typing Python commands!

Using the REPL

Once you have a prompt you can start experimenting! Anything you type at the prompt will be executed after you press the Enter key. MicroPython will run the code that you enter and print the result (if there is one). If there is an error with the text that you enter then an error message is printed.

Try typing the following at the prompt:

```
>>> print('hello esp8266!')
hello esp8266!
```

Note that you shouldn't type the >>> arrows, they are there to indicate that you should type the text after it at the prompt. And then the line following is what the device should respond with. In the end, once you have entered the text `print("hello esp8266!")` and pressed the Enter key, the output on your screen should look exactly like it does above.

If you already know some python you can now try some basic commands here. For example:

```
>>> 1 + 2
3
>>> 1 / 2
0.5
>>> 12**34
49222352429520267040371132431220080064
```

If your board has an LED attached to GPIO2 (the ESP-12 modules do) then you can turn it on and off using the following code:

```
>>> import machine
>>> pin = machine.Pin(2, machine.Pin.OUT)
>>> pin.on()
>>> pin.off()
```

Note that `on` method of a Pin might turn the LED off and `off` might turn it on (or vice versa), depending on how the LED is wired on your board. To resolve this, machine.Signal class is provided.

Line editing

You can edit the current line that you are entering using the left and right arrow keys to move the cursor, as well as the delete and backspace keys. Also, pressing Home or ctrl-A moves the cursor to the start of the line, and pressing End or ctrl-E moves to the end of the line.

Input history

The REPL remembers a certain number of previous lines of text that you entered (up to 8 on the ESP8266). To recall previous lines use the up and down arrow keys.

Tab completion

Pressing the Tab key will do an auto-completion of the current word that you are entering. This can be very useful to find out functions and methods that a module or object has. Try it out by typing "ma" and then pressing Tab. It should complete to "machine" (assuming you imported machine in the above example). Then type "." and press Tab again to see a list of all the functions that the machine module has.

Line continuation and auto-indent

Certain things that you type will need "continuing", that is, will need more lines of text to make a proper Python statement. In this case the prompt will change to `...` and the cursor will auto-indent the correct amount so you can start typing the next line straight away. Try this by defining the following function:

```
>>> def toggle(p):
...     p.value(not p.value())
...
...
...
>>>
```

In the above, you needed to press the Enter key three times in a row to finish the compound statement (that's the three lines with just dots on them). The other way to finish a compound statement is to press backspace to get to the start of the line, then press the Enter key. (If you did something wrong and want to escape the continuation mode then press ctrl-C; all lines will be ignored.)

The function you just defined allows you to toggle a pin. The pin object you created earlier should still exist (recreate it if it doesn't) and you can toggle the LED using:

```
>>> toggle(pin)
```

Let's now toggle the LED in a loop (if you don't have an LED then you can just print some text instead of calling toggle, to see the effect):

```
>>> import time
>>> while True:
...     toggle(pin)
...     time.sleep_ms(500)
...
...
...
>>>
```

This will toggle the LED at 1Hz (half a second on, half a second off). To stop the toggling press ctrl-C, which will raise a KeyboardInterrupt exception and break out of the loop.

The time module provides some useful functions for making delays and doing timing. Use tab completion to find out what they are and play around with them!

Paste mode

Pressing ctrl-E will enter a special paste mode. This allows you to copy and paste a chunk of text into the REPL. If you press ctrl-E you will see the paste-mode prompt:

```
paste mode; Ctrl-C to cancel, Ctrl-D to finish
===
```

You can then paste (or type) your text in. Note that none of the special keys or commands work in paste mode (eg Tab or backspace), they are just accepted as-is. Press ctrl-D to finish entering the text and execute it.

Other control commands

There are four other control commands:

- Ctrl-A on a blank line will enter raw REPL mode. This is like a permanent paste mode, except that characters are not echoed back.
- Ctrl-B on a blank like goes to normal REPL mode.
- Ctrl-C cancels any input, or interrupts the currently running code.
- Ctrl-D on a blank line will do a soft reset.

Note that ctrl-A and ctrl-D do not work with WebREPL.

7.2.3 The internal filesystem

If your devices has 1Mbyte or more of storage then it will be set up (upon first boot) to contain a filesystem. This filesystem uses the FAT format and is stored in the flash after the MicroPython firmware.

Creating and reading files

MicroPython on the ESP8266 supports the standard way of accessing files in Python, using the built-in `open()` function.

To create a file try:

```
>>> f = open('data.txt', 'w')
>>> f.write('some data')
9
>>> f.close()
```

The "9" is the number of bytes that were written with the `write()` method. Then you can read back the contents of this new file using:

```
>>> f = open('data.txt')
>>> f.read()
'some data'
>>> f.close()
```

Note that the default mode when opening a file is to open it in read-only mode, and as a text file. Specify `'wb'` as the second argument to `open()` to open for writing in binary mode, and `'rb'` to open for reading in binary mode.

Listing file and more

The os module can be used for further control over the filesystem. First import the module:

```
>>> import os
```

Then try listing the contents of the filesystem:

```
>>> os.listdir()
['boot.py', 'port_config.py', 'data.txt']
```

You can make directories:

```
>>> os.mkdir('dir')
```

And remove entries:

```
>>> os.remove('data.txt')
```

Start up scripts

There are two files that are treated specially by the ESP8266 when it starts up: boot.py and main.py. The boot.py script is executed first (if it exists) and then once it completes the main.py script is executed. You can create these files yourself and populate them with the code that you want to run when the device starts up.

Accessing the filesystem via WebREPL

You can access the filesystem over WebREPL using the web client in a browser or via the command-line tool. Please refer to Quick Reference and Tutorial sections for more information about WebREPL.

7.2.4 Network basics

The *network.WLAN* class in the *network* module is used to configure the WiFi connection. There are two WiFi interfaces, one for the station (when the ESP8266 connects to a router) and one for the access point (for other devices to connect to the ESP8266). Create instances of these objects using:

```
>>> import network
>>> sta_if = network.WLAN(network.WLAN.IF_STA)
>>> ap_if = network.WLAN(network.WLAN.IF_AP)
```

You can check if the interfaces are active by:

```
>>> sta_if.active()
False
>>> ap_if.active()
True
```

You can also check the network settings of the interface by:

```
>>> ap_if.ipconfig('addr4')
('192.168.4.1', '255.255.255.0')
```

The returned values are: IP address and netmask.

Configuration of the WiFi

Upon a fresh install the ESP8266 is configured in access point mode, so the AP_IF interface is active and the STA_IF interface is inactive. You can configure the module to connect to your own network using the STA_IF interface.

First activate the station interface:

```
>>> sta_if.active(True)
```

Then connect to your WiFi network:

```
>>> sta_if.connect('<your SSID>', '<your key>')
```

To check if the connection is established use:

```
>>> sta_if.isconnected()
```

Once established you can check the IP address:

```
>>> sta_if.ipconfig('addr4')
('192.168.0.2', '255.255.255.0')
```

You can then disable the access-point interface if you no longer need it:

```
>>> ap_if.active(False)
```

Here is a function you can run (or put in your boot.py file) to automatically connect to your WiFi network:

```
def do_connect():
    import network
    sta_if = network.WLAN(network.WLAN.IF_STA)
    if not sta_if.isconnected():
        print('connecting to network...')
        sta_if.active(True)
        sta_if.connect('<ssid>', '<key>')
        while not sta_if.isconnected():
            pass
    print('network config:', sta_if.ipconfig('addr4'))
```

Sockets

Once the WiFi is set up the way to access the network is by using sockets. A socket represents an endpoint on a network device, and when two sockets are connected together communication can proceed. Internet protocols are built on top of sockets, such as email (SMTP), the web (HTTP), telnet, ssh, among many others. Each of these protocols is assigned a specific port, which is just an integer. Given an IP address and a port number you can connect to a remote device and start talking with it.

The next part of the tutorial discusses how to use sockets to do some common and useful network tasks.

7.2.5 Network - TCP sockets

The building block of most of the internet is the TCP socket. These sockets provide a reliable stream of bytes between the connected network devices. This part of the tutorial will show how to use TCP sockets in a few different cases.

Star Wars Asciimation

The simplest thing to do is to download data from the internet. In this case we will use the Star Wars Asciimation service provided by the blinkenlights.nl website. It uses the telnet protocol on port 23 to stream data to anyone that connects. It's very simple to use because it doesn't require you to authenticate (give a username or password), you can just start downloading data straight away.

The first thing to do is make sure we have the socket module available:

```
>>> import socket
```

Then get the IP address of the server:

```
>>> addr_info = socket.getaddrinfo("towel.blinkenlights.nl", 23)
```

The `getaddrinfo` function actually returns a list of addresses, and each address has more information than we need. We want to get just the first valid address, and then just the IP address and port of the server. To do this use:

```
>>> addr = addr_info[0][-1]
```

If you type `addr_info` and `addr` at the prompt you will see exactly what information they hold.

Using the IP address we can make a socket and connect to the server:

```
>>> s = socket.socket()
>>> s.connect(addr)
```

Now that we are connected we can download and display the data:

```
>>> while True:
...     data = s.recv(500)
...     print(str(data, 'utf8'), end='')
...
```

When this loop executes it should start showing the animation (use ctrl-C to interrupt it).

You should also be able to run this same code on your PC using normal Python if you want to try it out there.

HTTP GET request

The next example shows how to download a webpage. HTTP uses port 80 and you first need to send a "GET" request before you can download anything. As part of the request you need to specify the page to retrieve.

Let's define a function that can download and print a URL:

```
def http_get(url):
    import socket
    _, _, host, path = url.split('/', 3)
    addr = socket.getaddrinfo(host, 80)[0][-1]
    s = socket.socket()
    s.connect(addr)
```

(continues on next page)

(continued from previous page)

```
    s.send(bytes('GET /%s HTTP/1.0\r\nHost: %s\r\n\r\n' % (path, host), 'utf8'))
    while True:
        data = s.recv(100)
        if data:
            print(str(data, 'utf8'), end='')
        else:
            break
    s.close()
```

Then you can try:

```
>>> http_get('http://micropython.org/ks/test.html')
```

This should retrieve the webpage and print the HTML to the console.

Simple HTTP server

The following code creates an simple HTTP server which serves a single webpage that contains a table with the state of all the GPIO pins:

```
import machine
pins = [machine.Pin(i, machine.Pin.IN) for i in (0, 2, 4, 5, 12, 13, 14, 15)]

html = """<!DOCTYPE html>
<html>
    <head> <title>ESP8266 Pins</title> </head>
    <body> <h1>ESP8266 Pins</h1>
        <table border="1"> <tr><th>Pin</th><th>Value</th></tr> %s </table>
    </body>
</html>
"""

import socket
addr = socket.getaddrinfo('0.0.0.0', 80)[0][-1]

s = socket.socket()
s.bind(addr)
s.listen(1)

print('listening on', addr)

while True:
    cl, addr = s.accept()
    print('client connected from', addr)
    cl_file = cl.makefile('rwb', 0)
    while True:
        line = cl_file.readline()
        if not line or line == b'\r\n':
            break
    rows = ['<tr><td>%s</td><td>%d</td></tr>' % (str(p), p.value()) for p in pins]
    response = html % '\n'.join(rows)
    cl.send('HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n')
    cl.send(response)
    cl.close()
```

7.2.6 GPIO Pins

The way to connect your board to the external world, and control other components, is through the GPIO pins. Not all pins are available to use, in most cases only pins 0, 2, 4, 5, 12, 13, 14, 15, and 16 can be used.

The pins are available in the machine module, so make sure you import that first. Then you can create a pin using:

```
>>> pin = machine.Pin(0)
```

Here, the "0" is the pin that you want to access. Usually you want to configure the pin to be input or output, and you do this when constructing it. To make an input pin use:

```
>>> pin = machine.Pin(0, machine.Pin.IN, machine.Pin.PULL_UP)
```

You can either use PULL_UP or None for the input pull-mode. If it's not specified then it defaults to None, which is no pull resistor. GPIO16 has no pull-up mode. You can read the value on the pin using:

```
>>> pin.value()
0
```

The pin on your board may return 0 or 1 here, depending on what it's connected to. To make an output pin use:

```
>>> pin = machine.Pin(0, machine.Pin.OUT)
```

Then set its value using:

```
>>> pin.value(0)
>>> pin.value(1)
```

Or:

```
>>> pin.off()
>>> pin.on()
```

External interrupts

All pins except number 16 can be configured to trigger a hard interrupt if their input changes. You can set code (a callback function) to be executed on the trigger.

Let's first define a callback function, which must take a single argument, being the pin that triggered the function. We will make the function just print the pin:

```
>>> def callback(p):
...     print('pin change', p)
```

Next we will create two pins and configure them as inputs:

```
>>> from machine import Pin
>>> p0 = Pin(0, Pin.IN)
>>> p2 = Pin(2, Pin.IN)
```

An finally we need to tell the pins when to trigger, and the function to call when they detect an event:

```
>>> p0.irq(trigger=Pin.IRQ_FALLING, handler=callback)
>>> p2.irq(trigger=Pin.IRQ_RISING | Pin.IRQ_FALLING, handler=callback)
```

We set pin 0 to trigger only on a falling edge of the input (when it goes from high to low), and set pin 2 to trigger on both a rising and falling edge. After entering this code you can apply high and low voltages to pins 0 and 2 to see the interrupt being executed.

A hard interrupt will trigger as soon as the event occurs and will interrupt any running code, including Python code. As such your callback functions are limited in what they can do (they cannot allocate memory, for example) and should be as short and simple as possible.

7.2.7 Pulse Width Modulation

Pulse width modulation (PWM) is a way to get an artificial analog output on a digital pin. It achieves this by rapidly toggling the pin from low to high. There are two parameters associated with this: the frequency of the toggling, and the duty cycle. The duty cycle is defined to be how long the pin is high compared with the length of a single period (low plus high time). Maximum duty cycle is when the pin is high all of the time, and minimum is when it is low all of the time.

On the ESP8266 the pins 0, 2, 4, 5, 12, 13, 14 and 15 all support PWM. The limitation is that they must all be at the same frequency, and the frequency must be between 1Hz and 1kHz.

To use PWM on a pin you must first create the pin object, for example:

```
>>> import machine
>>> p12 = machine.Pin(12)
```

Then create the PWM object using:

```
>>> pwm12 = machine.PWM(p12)
```

You can set the frequency and duty cycle using:

```
>>> pwm12.freq(500)
>>> pwm12.duty(512)
```

Note that the duty cycle is between 0 (all off) and 1023 (all on), with 512 being a 50% duty. Values beyond this min/max will be clipped. If you print the PWM object then it will tell you its current configuration:

```
>>> pwm12
PWM(12, freq=500, duty=512)
```

You can also call the `freq()` and `duty()` methods with no arguments to get their current values.

The pin will continue to be in PWM mode until you deinitialise it using:

```
>>> pwm12.deinit()
```

Fading an LED

Let's use the PWM feature to fade an LED. Assuming your board has an LED connected to pin 2 (ESP-12 modules do) we can create an LED-PWM object using:

```
>>> led = machine.PWM(machine.Pin(2), freq=1000)
```

Notice that we can set the frequency in the PWM constructor.

For the next part we will use timing and some math, so import these modules:

```
>>> import time, math
```

Then create a function to pulse the LED:

```
>>> def pulse(l, t):
...     for i in range(20):
...         l.duty(int(math.sin(i / 10 * math.pi) * 500 + 500))
...         time.sleep_ms(t)
```

You can try this function out using:

```
>>> pulse(led, 50)
```

For a nice effect you can pulse many times in a row:

```
>>> for i in range(10):
...     pulse(led, 20)
```

Remember you can use ctrl-C to interrupt the code.

Control a hobby servo

Hobby servo motors can be controlled using PWM. They require a frequency of 50Hz and then a duty between about 40 and 115, with 77 being the centre value. If you connect a servo to the power and ground pins, and then the signal line to pin 12 (other pins will work just as well), you can control the motor using:

```
>>> servo = machine.PWM(machine.Pin(12), freq=50)
>>> servo.duty(40)
>>> servo.duty(115)
>>> servo.duty(77)
```

7.2.8 Analog to Digital Conversion

The ESP8266 has a single pin (separate to the GPIO pins) which can be used to read analog voltages and convert them to a digital value. You can construct such an ADC pin object using:

```
>>> import machine
>>> adc = machine.ADC(0)
```

Then read its value with:

```
>>> adc.read()
58
```

The values returned from the `read()` function are between 0 (for 0.0 volts) and 1024 (for 1.0 volts). Please note that this input can only tolerate a maximum of 1.0 volts and you must use a voltage divider circuit to measure larger voltages.

7.2.9 Power control

The ESP8266 provides the ability to change the CPU frequency on the fly, and enter a deep-sleep state. Both can be used to manage power consumption.

Changing the CPU frequency

The machine module has a function to get and set the CPU frequency. To get the current frequency use:

```
>>> import machine
>>> machine.freq()
80000000
```

By default the CPU runs at 80MHz. It can be changed to 160MHz if you need more processing power, at the expense of current consumption:

```
>>> machine.freq(160000000)
>>> machine.freq()
160000000
```

You can change to the higher frequency just while your code does the heavy processing and then change back when it's finished.

Deep-sleep mode

The deep-sleep mode will shut down the ESP8266 and all its peripherals, including the WiFi (but not including the real-time-clock, which is used to wake the chip). This drastically reduces current consumption and is a good way to make devices that can run for a while on a battery.

To be able to use the deep-sleep feature you must connect GPIO16 to the reset pin (RST on the Adafruit Feather HUZZAH board). Then the following code can be used to sleep and wake the device:

```
import machine

# configure RTC.ALARM0 to be able to wake the device
rtc = machine.RTC()
rtc.irq(trigger=rtc.ALARM0, wake=machine.DEEPSLEEP)

# set RTC.ALARM0 to fire after 10 seconds (waking the device)
rtc.alarm(rtc.ALARM0, 10000)

# put the device to sleep
machine.deepsleep()
```

Note that when the chip wakes from a deep-sleep it is completely reset, including all of the memory. The boot scripts will run as usual and you can put code in them to check the reset cause to perhaps do something different if the device just woke from a deep-sleep. For example, to print the reset cause you can use:

```
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
    print('woke from a deep sleep')
else:
    print('power on or hard reset')
```

7.2.10 Controlling 1-wire devices

The 1-wire bus is a serial bus that uses just a single wire for communication (in addition to wires for ground and power). The DS18B20 temperature sensor is a very popular 1-wire device, and here we show how to use the onewire module to read from such a device.

For the following code to work you need to have at least one DS18S20 or DS18B20 temperature sensor with its data line connected to GPIO12. You must also power the sensors and connect a 4.7k Ohm resistor between the data pin and the power pin.

```
import time
import machine
import onewire, ds18x20

# the device is on GPIO12
dat = machine.Pin(12)

# create the onewire object
ds = ds18x20.DS18X20(onewire.OneWire(dat))

# scan for devices on the bus
roms = ds.scan()
print('found devices:', roms)

# loop 10 times and print all temperatures
for i in range(10):
    print('temperatures:', end=' ')
    ds.convert_temp()
    time.sleep_ms(750)
    for rom in roms:
        print(ds.read_temp(rom), end=' ')
    print()
```

Note that you must execute the `convert_temp()` function to initiate a temperature reading, then wait at least 750ms before reading the value.

7.2.11 Controlling NeoPixels

NeoPixels, also known as WS2812 LEDs, are full-colour LEDs that are connected in serial, are individually addressable, and can have their red, green and blue components set between 0 and 255. They require precise timing to control them and there is a special neopixel module to do just this.

To create a NeoPixel object do the following:

```
>>> import machine, neopixel
>>> np = neopixel.NeoPixel(machine.Pin(4), 8)
```

This configures a NeoPixel strip on GPIO4 with 8 pixels. You can adjust the “4” (pin number) and the “8” (number of pixel) to suit your set up.

To set the colour of pixels use:

```
>>> np[0] = (255, 0, 0) # set to red, full brightness
>>> np[1] = (0, 128, 0) # set to green, half brightness
>>> np[2] = (0, 0, 64)  # set to blue, quarter brightness
```

For LEDs with more than 3 colours, such as RGBW pixels or RGBY pixels, the NeoPixel class takes a `bpp` parameter. To setup a NeoPixel object for an RGBW Pixel, do the following:

```
>>> import machine, neopixel
>>> np = neopixel.NeoPixel(machine.Pin(4), 8, bpp=4)
```

In a 4-bpp mode, remember to use 4-tuples instead of 3-tuples to set the colour. For example to set the first three pixels use:

```
>>> np[0] = (255, 0, 0, 128) # Orange in an RGBY Setup
>>> np[1] = (0, 255, 0, 128) # Yellow-green in an RGBY Setup
>>> np[2] = (0, 0, 255, 128) # Green-blue in an RGBY Setup
```

Then use the `write()` method to output the colours to the LEDs:

```
>>> np.write()
```

The following demo function makes a fancy show on the LEDs:

```
import time

def demo(np):
    n = np.n

    # cycle
    for i in range(4 * n):
        for j in range(n):
            np[j] = (0, 0, 0)
        np[i % n] = (255, 255, 255)
        np.write()
        time.sleep_ms(25)

    # bounce
    for i in range(4 * n):
        for j in range(n):
            np[j] = (0, 0, 128)
        if (i // n) % 2 == 0:
            np[i % n] = (0, 0, 0)
        else:
            np[n - 1 - (i % n)] = (0, 0, 0)
        np.write()
        time.sleep_ms(60)

    # fade in/out
    for i in range(0, 4 * 256, 8):
        for j in range(n):
            if (i // 256) % 2 == 0:
                val = i & 0xff
            else:
                val = 255 - (i & 0xff)
            np[j] = (val, 0, 0)
        np.write()

    # clear
    for i in range(n):
        np[i] = (0, 0, 0)
    np.write()
```

Execute it using:

```
>>> demo(np)
```

7.2.12 Controlling APA102 LEDs

APA102 LEDs, also known as DotStar LEDs, are individually addressable full-colour RGB LEDs, generally in a string formation. They differ from NeoPixels in that they require two pins to control - both a Clock and Data pin. They can operate at a much higher data and PWM frequencies than NeoPixels and are more suitable for persistence-of-vision effects.

To create an APA102 object do the following:

```
>>> import machine, apa102
>>> strip = apa102.APA102(machine.Pin(5), machine.Pin(4), 60)
```

This configures an 60 pixel APA102 strip with clock on GPIO5 and data on GPIO4. You can adjust the pin numbers and the number of pixels to suit your needs.

The RGB colour data, as well as a brightness level, is sent to the APA102 in a certain order. Usually this is `(Red, Green, Blue, Brightness)`. If you are using one of the newer APA102C LEDs the green and blue are swapped, so the order is `(Red, Blue, Green, Brightness)`. The APA102 has more of a square lens while the APA102C has more of a round one. If you are using a APA102C strip and would prefer to provide colours in RGB order instead of RBG, you can customise the tuple colour order like so:

```
>>> strip.ORDER = (0, 2, 1, 3)
```

To set the colour of pixels use:

```
>>> strip[0] = (255, 255, 255, 31) # set to white, full brightness
>>> strip[1] = (255, 0, 0, 31) # set to red, full brightness
>>> strip[2] = (0, 255, 0, 15) # set to green, half brightness
>>> strip[3] = (0, 0, 255, 7)  # set to blue, quarter brightness
```

Use the `write()` method to output the colours to the LEDs:

```
>>> strip.write()
```

Demonstration:

```
import time
import machine, apa102

# 1M strip with 60 LEDs
strip = apa102.APA102(machine.Pin(5), machine.Pin(4), 60)

brightness = 1  # 0 is off, 1 is dim, 31 is max

# Helper for converting 0-255 offset to a colour tuple
def wheel(offset, brightness):
    # The colours are a transition r - g - b - back to r
    offset = 255 - offset
    if offset < 85:
        return (255 - offset * 3, 0, offset * 3, brightness)
    if offset < 170:
        offset -= 85
        return (0, offset * 3, 255 - offset * 3, brightness)
```

(continues on next page)

(continued from previous page)

```
    offset -= 170
    return (offset * 3, 255 - offset * 3, 0, brightness)

# Demo 1: RGB RGB RGB
red = 0xff0000
green = red >> 8
blue = red >> 16
for i in range(strip.n):
    colour = red >> (i % 3) * 8
    strip[i] = ((colour & red) >> 16, (colour & green) >> 8, (colour & blue),␣
↪brightness)
strip.write()

# Demo 2: Show all colours of the rainbow
for i in range(strip.n):
    strip[i] = wheel((i * 256 // strip.n) % 255, brightness)
strip.write()

# Demo 3: Fade all pixels together through rainbow colours, offset each pixel
for r in range(5):
    for n in range(256):
        for i in range(strip.n):
            strip[i] = wheel(((i * 256 // strip.n) + n) & 255, brightness)
        strip.write()
    time.sleep_ms(25)

# Demo 4: Same colour, different brightness levels
for b in range(31,-1,-1):
    strip[0] = (255, 153, 0, b)
    strip.write()
    time.sleep_ms(250)

# End: Turn off all the LEDs
strip.fill((0, 0, 0, 0))
strip.write()
```

7.2.13 Temperature and Humidity

DHT (Digital Humidity & Temperature) sensors are low cost digital sensors with capacitive humidity sensors and thermistors to measure the surrounding air. They feature a chip that handles analog to digital conversion and provide a 1-wire interface. Newer sensors additionally provide an I2C interface.

The DHT11 (blue) and DHT22 (white) sensors provide the same 1-wire interface, however, the DHT22 requires a separate object as it has more complex calculation. DHT22 have 1 decimal place resolution for both humidity and temperature readings. DHT11 have whole number for both.

A custom 1-wire protocol, which is different to Dallas 1-wire, is used to get the measurements from the sensor. The payload consists of a humidity value, a temperature value and a checksum.

To use the 1-wire interface, construct the objects referring to their data pin:

```
>>> import dht
>>> import machine
>>> d = dht.DHT11(machine.Pin(4))

>>> import dht
```

(continues on next page)

(continued from previous page)

```
>>> import machine
>>> d = dht.DHT22(machine.Pin(4))
```

Then measure and read their values with:

```
>>> d.measure()
>>> d.temperature()
>>> d.humidity()
```

Values returned from `temperature()` are in degrees Celsius and values returned from `humidity()` are a percentage of relative humidity.

The DHT11 can be called no more than once per second and the DHT22 once every two seconds for most accurate results. Sensor accuracy will degrade over time. Each sensor supports a different operating range. Refer to the product datasheets for specifics.

In 1-wire mode, only three of the four pins are used and in I2C mode, all four pins are used. Older sensors may still have 4 pins even though they do not support I2C. The 3rd pin is simply not connected.

Pin configurations:

Sensor without I2C in 1-wire mode (eg. DHT11, DHT22, AM2301, AM2302):

1=VDD, 2=Data, 3=NC, 4=GND

Sensor with I2C in 1-wire mode (eg. DHT12, AM2320, AM2321, AM2322):

1=VDD, 2=Data, 3=GND, 4=GND

Sensor with I2C in I2C mode (eg. DHT12, AM2320, AM2321, AM2322):

1=VDD, 2=SDA, 3=GND, 4=SCL

You should use pull-up resistors for the Data, SDA and SCL pins.

To make newer I2C sensors work in backwards compatible 1-wire mode, you must connect both pins 3 and 4 to GND. This disables the I2C interface.

DHT22 sensors are now sold under the name AM2302 and are otherwise identical.

7.2.14 Using a SSD1306 OLED display

The SSD1306 OLED display uses either a SPI or I2C interface and comes in a variety of sizes (128x64, 128x32, 72x40, 64x48) and colours (white, yellow, blue, yellow + blue).

Hardware SPI interface:

```
from machine import Pin, SPI
import ssd1306

hspi = SPI(1)  # sck=14 (scl), mosi=13 (sda), miso=12 (unused)

dc = Pin(4)    # data/command
rst = Pin(5)   # reset
cs = Pin(15)   # chip select, some modules do not have a pin for this

display = ssd1306.SSD1306_SPI(128, 64, hspi, dc, rst, cs)
```

Software SPI interface:

```
from machine import Pin, SoftSPI
import ssd1306

spi = SoftSPI(baudrate=500000, polarity=1, phase=0, sck=Pin(14), mosi=Pin(13),␣
↪miso=Pin(12))

dc = Pin(4)    # data/command
rst = Pin(5)   # reset
cs = Pin(15)   # chip select, some modules do not have a pin for this

display = ssd1306.SSD1306_SPI(128, 64, spi, dc, rst, cs)
```

I2C interface:

```
from machine import Pin, I2C
import ssd1306

# using default address 0x3C
i2c = I2C(sda=Pin(4), scl=Pin(5))
display = ssd1306.SSD1306_I2C(128, 64, i2c)
```

Print Hello World on the first line:

```
display.text('Hello, World!', 0, 0, 1)
display.show()
```

Basic functions:

```
display.poweroff()      # power off the display, pixels persist in memory
display.poweron()       # power on the display, pixels redrawn
display.contrast(0)     # dim
display.contrast(255)   # bright
display.invert(1)       # display inverted
display.invert(0)       # display normal
display.rotate(True)    # rotate 180 degrees
display.rotate(False)   # rotate 0 degrees
display.show()          # write the contents of the FrameBuffer to display memory
```

Subclassing FrameBuffer provides support for graphics primitives:

```
display.fill(0)                         # fill entire screen with colour=0
display.pixel(0, 10)                    # get pixel at x=0, y=10
display.pixel(0, 10, 1)                 # set pixel at x=0, y=10 to colour=1
display.hline(0, 8, 4, 1)               # draw horizontal line x=0, y=8, width=4,␣
↪colour=1
display.vline(0, 8, 4, 1)               # draw vertical line x=0, y=8, height=4,␣
↪colour=1
display.line(0, 0, 127, 63, 1)          # draw a line from 0,0 to 127,63
display.rect(10, 10, 107, 43, 1)        # draw a rectangle outline 10,10 to 117,53,␣
↪colour=1
display.fill_rect(10, 10, 107, 43, 1)   # draw a solid rectangle 10,10 to 117,53,␣
↪colour=1
display.text('Hello World', 0, 0, 1)    # draw some text at x=0, y=0, colour=1
display.scroll(20, 0)                   # scroll 20 pixels to the right

# draw another FrameBuffer on top of the current one at the given coordinates
import framebuf
```

(continues on next page)

(continued from previous page)

```
fbuf = framebuf.FrameBuffer(bytearray(8 * 8 * 1), 8, 8, framebuf.MONO_VLSB)
fbuf.line(0, 0, 7, 7, 1)
display.blit(fbuf, 10, 10, 0)           # draw on top at x=10, y=10, key=0
display.show()
```

Draw the MicroPython logo and print some text:

```
display.fill(0)
display.fill_rect(0, 0, 32, 32, 1)
display.fill_rect(2, 2, 28, 28, 0)
display.vline(9, 8, 22, 1)
display.vline(16, 2, 22, 1)
display.vline(23, 8, 22, 1)
display.fill_rect(26, 24, 2, 4, 1)
display.text('MicroPython', 40, 0, 1)
display.text('SSD1306', 40, 12, 1)
display.text('OLED 128x64', 40, 24, 1)
display.show()
```

7.2.15 Next steps

That brings us to the end of the tutorial! Hopefully by now you have a good feel for the capabilities of MicroPython on the ESP8266 and understand how to control both the WiFi and IO aspects of the chip.

There are many features that were not covered in this tutorial. The best way to learn about them is to read the full documentation of the modules, and to experiment!

Good luck creating your Internet of Things devices!

7.3 Installing MicroPython

See the corresponding section of tutorial: *Getting started with MicroPython on the ESP8266*. It also includes a troubleshooting subsection.

7.4 General board control

The MicroPython REPL is on UART0 (GPIO1=TX, GPIO3=RX) at baudrate 115200. Tab-completion is useful to find out what methods an object has. Paste mode (ctrl-E) is useful to paste a large slab of Python code into the REPL.

The *machine* module:

```
import machine

machine.freq()          # get the current frequency of the CPU
machine.freq(160000000) # set the CPU frequency to 160 MHz
```

The *esp* module:

```
import esp
```

(continues on next page)

(continued from previous page)

```
esp.osdebug(None)       # turn off vendor O/S debugging messages
esp.osdebug(0)          # redirect vendor O/S debugging messages to UART(0)
```

7.5 Networking

The *network.WLAN* class in the *network* module:

```
import network

wlan = network.WLAN(network.WLAN.IF_STA) # create station interface
wlan.active(True)       # activate the interface
wlan.scan()             # scan for access points
wlan.isconnected()      # check if the station is connected to an AP
wlan.connect('ssid', 'key') # connect to an AP
wlan.config('mac')      # get the interface's MAC address
wlan.ipconfig('addr4')  # get the interface's IPv4 addresses

ap = network.WLAN(network.WLAN.IF_AP) # create access-point interface
ap.active(True)         # activate the interface
ap.config(ssid='ESP-AP') # set the SSID of the access point
```

A useful function for connecting to your local WiFi network is:

```
def do_connect():
    import network
    wlan = network.WLAN(network.WLAN.IF_STA)
    wlan.active(True)
    if not wlan.isconnected():
        print('connecting to network...')
        wlan.connect('ssid', 'key')
        while not wlan.isconnected():
            pass
    print('network config:', wlan.ipconfig('addr4'))
```

Once the network is established the *socket* module can be used to create and use TCP/UDP sockets as usual.

7.6 Delay and timing

Use the *time* module:

```
import time

time.sleep(1)           # sleep for 1 second
time.sleep_ms(500)      # sleep for 500 milliseconds
time.sleep_us(10)       # sleep for 10 microseconds
start = time.ticks_ms() # get millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start) # compute time difference
```

7.7 Timers

Virtual (RTOS-based) timers are supported. Use the *machine.Timer* class with timer ID of -1:

```
from machine import Timer

tim = Timer(-1)
tim.init(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(1))
tim.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(2))
```

The period is in milliseconds.

By default, timer callbacks are scheduled as soft interrupts on this port. Specify `hard=True` to run them in hard interrupt context instead.

7.8 Pins and GPIO

Use the *machine.Pin* class:

```
from machine import Pin

p0 = Pin(0, Pin.OUT)    # create output pin on GPIO0
p0.on()                 # set pin to "on" (high) level
p0.off()                # set pin to "off" (low) level
p0.value(1)             # set pin to on/high

p2 = Pin(2, Pin.IN)     # create input pin on GPIO2
print(p2.value())       # get value, 0 or 1

p4 = Pin(4, Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor
p5 = Pin(5, Pin.OUT, value=1) # set pin high on creation
```

Available pins are: 0, 1, 2, 3, 4, 5, 12, 13, 14, 15, 16, which correspond to the actual GPIO pin numbers of ESP8266 chip. Note that many end-user boards use their own adhoc pin numbering (marked e.g. D0, D1, ...). As MicroPython supports different boards and modules, physical pin numbering was chosen as the lowest common denominator. For mapping between board logical pins and physical chip pins, consult your board documentation.

Note that Pin(1) and Pin(3) are REPL UART TX and RX respectively. Also note that Pin(16) is a special pin (used for wakeup from deepsleep mode) and may be not available for use with higher-level classes like `Neopixel`.

There's a higher-level abstraction *machine.Signal* which can be used to invert a pin. Useful for illuminating active-low LEDs using `on()` or `value(1)`.

7.9 UART (serial bus)

See *machine.UART*.

```
from machine import UART
uart = UART(0, baudrate=9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

Two UARTs are available. UART0 is on Pins 1 (TX) and 3 (RX). UART0 is bidirectional, and by default is used for the REPL. UART1 is on Pins 2 (TX) and 8 (RX) however Pin 8 is used to connect the flash chip, so UART1 is TX only.

When UART0 is attached to the REPL, all incoming chars on UART(0) go straight to stdin so uart.read() will always return None. Use sys.stdin.read() if it's needed to read characters from the UART(0) while it's also used for the REPL (or detach, read, then reattach). When detached the UART(0) can be used for other purposes.

If there are no objects in any of the dupterm slots when the REPL is started (on *hard or soft reset*) then UART(0) is automatically attached. Without this, the only way to recover a board without a REPL would be to completely erase and reflash (which would install the default boot.py which attaches the REPL).

To detach the REPL from UART0, use:

```
import os
os.dupterm(None, 1)
```

The REPL is attached by default. If you have detached it, to reattach it use:

```
import os, machine
uart = machine.UART(0, 115200)
os.dupterm(uart, 1)
```

7.10 PWM (pulse width modulation)

PWM can be enabled on all pins except Pin(16). There is a single frequency for all channels, with range between 1 and 1000 (measured in Hz). The duty cycle is between 0 and 1023 inclusive.

Use the `machine.PWM` class:

```
from machine import Pin, PWM

pwm0 = PWM(Pin(0))      # create PWM object from a pin
pwm0.freq()             # get current frequency
pwm0.freq(1000)         # set frequency
pwm0.duty()             # get current duty cycle
pwm0.duty(200)          # set duty cycle
pwm0.deinit()           # turn off PWM on the pin

pwm2 = PWM(Pin(2), freq=500, duty=512) # create and configure in one go
```

7.11 ADC (analog to digital conversion)

ADC is available on a dedicated pin. Note that input voltages on the ADC pin must be between 0v and 1.0v.

Use the *machine.ADC* class:

```
from machine import ADC

adc = ADC(0)            # create ADC object on ADC pin
adc.read()              # read value, 0-1024
```

7.12 Software SPI bus

There are two SPI drivers. One is implemented in software (bit-banging) and works on all pins, and is accessed via the *machine.SoftSPI* class:

```
from machine import Pin, SoftSPI

# construct an SPI bus on the given pins
# polarity is the idle state of SCK
# phase=0 means sample on the first edge of SCK, phase=1 means the second
spi = SoftSPI(baudrate=100000, polarity=1, phase=0, sck=Pin(0), mosi=Pin(2),␣
↪miso=Pin(4))

spi.init(baudrate=200000) # set the baudrate

spi.read(10)            # read 10 bytes on MISO
spi.read(10, 0xff)      # read 10 bytes while outputting 0xff on MOSI

buf = bytearray(50)     # create a buffer
spi.readinto(buf)       # read into the given buffer (reads 50 bytes in this case)
spi.readinto(buf, 0xff) # read into the given buffer and output 0xff on MOSI

spi.write(b'12345')     # write 5 bytes on MOSI

buf = bytearray(4)      # create a buffer
spi.write_readinto(b'1234', buf) # write to MOSI and read from MISO into the buffer
spi.write_readinto(buf, buf) # write buf to MOSI and read MISO back into buf
```

7.13 Hardware SPI bus

The hardware SPI is faster (up to 80Mhz), but only works on following pins: `MISO` is GPIO12, `MOSI` is GPIO13, and `SCK` is GPIO14. It has the same methods as the bitbanging SPI class above, except for the pin parameters for the constructor and init (as those are fixed):

```
from machine import Pin, SPI

hspi = SPI(1, baudrate=80000000, polarity=0, phase=0)
```

(`SPI(0)` is used for FlashROM and not available to users.)

7.14 I2C bus

The I2C driver is implemented in software and works on all pins, and is accessed via the *machine.I2C* class (which is an alias of *machine.SoftI2C*):

```
from machine import Pin, I2C

# construct an I2C bus
i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)

i2c.readfrom(0x3a, 4)   # read 4 bytes from peripheral device with address 0x3a
i2c.writeto(0x3a, '12') # write '12' to peripheral device with address 0x3a
```

(continues on next page)

(continued from previous page)

```
buf = bytearray(10)      # create a buffer with 10 bytes
i2c.writeto(0x3a, buf)  # write the given buffer to the peripheral
```

7.15 Real time clock (RTC)

See *machine.RTC*

```
from machine import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 0, 1, 12, 48, 0)) # set a specific date and
                                             # time, eg. 2017/8/23 1:12:48
                                             # the day-of-week value is ignored
rtc.datetime() # get date and time

# synchronize with ntp
# need to be connected to wifi
import ntptime
ntptime.settime() # set the rtc datetime from the remote server
rtc.datetime()    # get the date and time in UTC
```

Note: Not all methods are implemented: *RTC.now()*, *RTC.irq(handler=*)* (using a custom handler), *RTC.init()* and *RTC.deinit()* are currently not supported.

7.16 WDT (Watchdog timer)

See *machine.WDT*.

```
from machine import WDT

# enable the WDT
wdt = WDT()
wdt.feed()
```

7.17 Deep-sleep mode

Connect GPIO16 to the reset pin (RST on HUZZAH). Then the following code can be used to sleep, wake and check the reset cause:

```
import machine

# configure RTC.ALARM0 to be able to wake the device
rtc = machine.RTC()
rtc.irq(trigger=rtc.ALARM0, wake=machine.DEEPSLEEP)

# check if the device woke from a deep sleep
```

(continues on next page)

(continued from previous page)

```
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
    print('woke from a deep sleep')

# set RTC.ALARM0 to fire after 10 seconds (waking the device)
rtc.alarm(rtc.ALARM0, 10000)

# put the device to sleep
machine.deepsleep()
```

7.18 OneWire driver

The OneWire driver is implemented in software and works on all pins:

```
from machine import Pin
import onewire

ow = onewire.OneWire(Pin(12)) # create a OneWire bus on GPIO12
ow.scan()               # return a list of devices on the bus
ow.reset()              # reset the bus
ow.readbyte()           # read a byte
ow.writebyte(0x12)      # write a byte on the bus
ow.write('123')         # write bytes on the bus
ow.select_rom(b'12345678') # select a specific device by its ROM code
```

There is a specific driver for DS18S20 and DS18B20 devices:

```
import time, ds18x20
ds = ds18x20.DS18X20(ow)
roms = ds.scan()
ds.convert_temp()
time.sleep_ms(750)
for rom in roms:
    print(ds.read_temp(rom))
```

Be sure to put a 4.7k pull-up resistor on the data line. Note that the `convert_temp()` method must be called each time you want to sample the temperature.

7.19 NeoPixel driver

Use the `neopixel` module:

```
from machine import Pin
from neopixel import NeoPixel

pin = Pin(0, Pin.OUT)    # set GPIO0 to output to drive NeoPixels
np = NeoPixel(pin, 8)    # create NeoPixel driver on GPIO0 for 8 pixels
np[0] = (255, 255, 255) # set the first pixel to white
np.write()              # write data to all pixels
r, g, b = np[0]         # get first pixel colour
```

Warning: By default `NeoPixel` is configured to control the more popular *800kHz* units. It is possible to use alternative timing to control other (typically 400kHz) devices by passing `timing=0` when constructing the `NeoPixel` object.

For low-level driving of a NeoPixel see *machine.bitstream*.

7.20 APA102 driver

Use the `apa102` module:

```
from machine import Pin
from apa102 import APA102

clock = Pin(14, Pin.OUT)      # set GPIO14 to output to drive the clock
data = Pin(13, Pin.OUT)       # set GPIO13 to output to drive the data
apa = APA102(clock, data, 8) # create APA102 driver on the clock and the data pin for␣
↪8 pixels
apa[0] = (255, 255, 255, 31) # set the first pixel to white with a maximum brightness␣
↪of 31
apa.write()                   # write data to all pixels
r, g, b, brightness = apa[0] # get first pixel colour
```

For low-level driving of an APA102:

```
import esp
esp.apa102_write(clock_pin, data_pin, rgbi_buf)
```

7.21 DHT driver

The DHT driver is implemented in software and works on all pins:

```
import dht
import machine

d = dht.DHT11(machine.Pin(4))
d.measure()
d.temperature() # eg. 23 (°C)
d.humidity()    # eg. 41 (% RH)

d = dht.DHT22(machine.Pin(4))
d.measure()
d.temperature() # eg. 23.6 (°C)
d.humidity()    # eg. 41.3 (% RH)
```

7.22 SSD1306 driver

Driver for SSD1306 monochrome OLED displays. See tutorial *Using a SSD1306 OLED display*.

```
from machine import Pin, I2C
import ssd1306

i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)
display = ssd1306.SSD1306_I2C(128, 64, i2c)

display.text('Hello World', 0, 0, 1)
display.show()
```

7.23 WebREPL (web browser interactive prompt)

WebREPL (REPL over WebSockets, accessible via a web browser) is an experimental feature available in ESP8266 port. Download web client from https://github.com/micropython/webrepl (hosted version available at http://micropython.org/webrepl), and configure it by executing:

```
import webrepl_setup
```

and following on-screen instructions. After reboot, it will be available for connection. If you disabled automatic start-up on boot, you may run configured daemon on demand using:

```
import webrepl
webrepl.start()
```

The supported way to use WebREPL is by connecting to ESP8266 access point, but the daemon is also started on STA interface if it is active, so if your router is set up and works correctly, you may also use WebREPL while connected to your normal Internet access point (use the ESP8266 AP connection method if you face any issues).

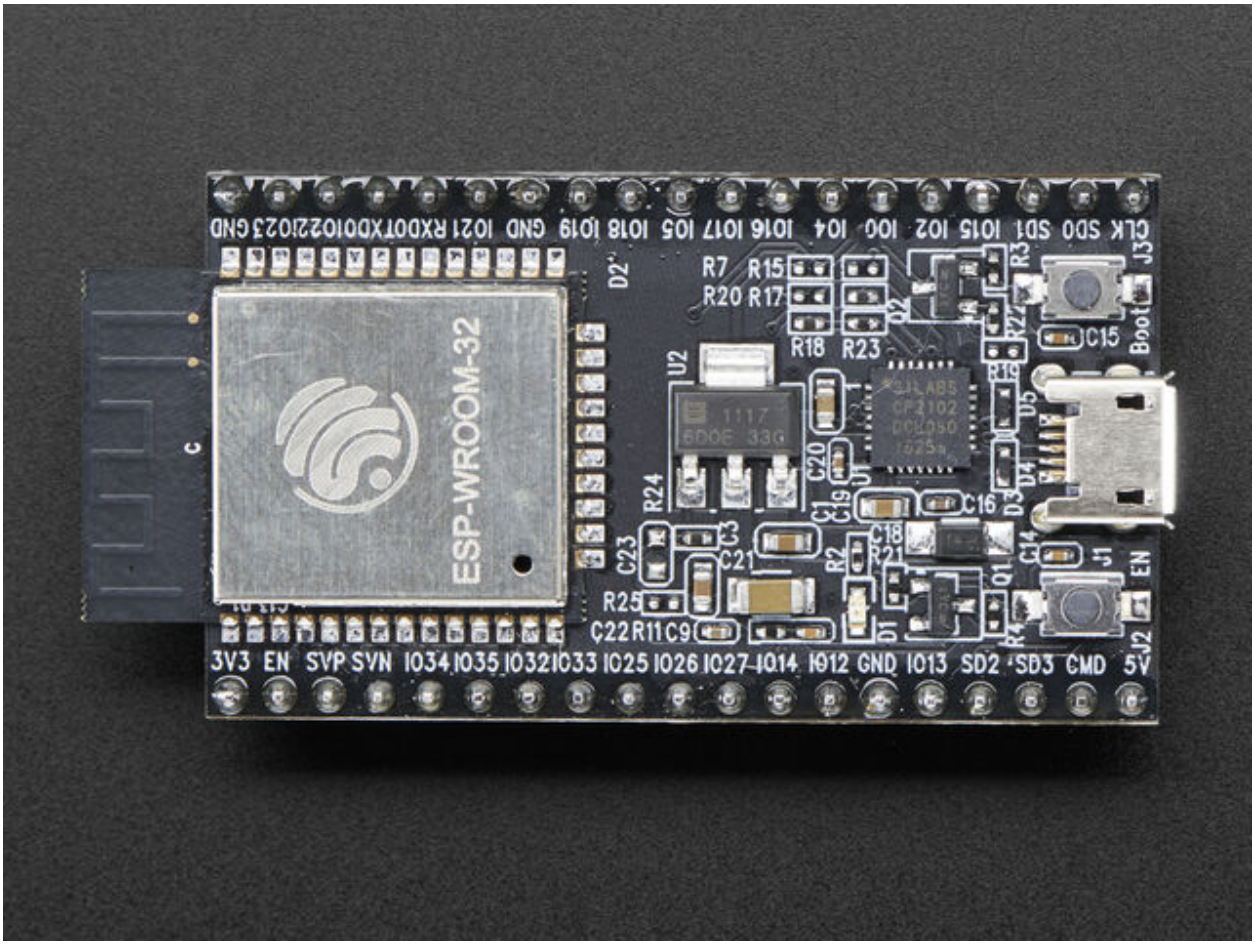
Besides terminal/command prompt access, WebREPL also has provision for file transfer (both upload and download). Web client has buttons for the corresponding functions, or you can use command-line client `webrepl_cli.py` from the repository above.

See the MicroPython forum for other community-supported alternatives to transfer files to ESP8266.

CHAPTER

EIGHT

QUICK REFERENCE FOR THE ESP32



The Espressif ESP32 Development Board (image attribution: Adafruit).

Below is a quick reference for ESP32-based boards. If it is your first time working with this board it may be useful to get an overview of the microcontroller:

8.1 General information about the ESP32 port

The ESP32 is a popular WiFi and Bluetooth enabled System-on-Chip (SoC) by Espressif Systems.

8.1.1 Multitude of boards

There is a multitude of modules and boards from different sources which carry the ESP32 chip. MicroPython tries to provide a generic port which would run on as many boards/modules as possible, but there may be limitations. Espressif development boards are taken as reference for the port (for example, testing is performed on them). For any board you are using please make sure you have a datasheet, schematics and other reference materials so you can look up any board-specific functions.

To make a generic ESP32 port and support as many boards as possible the following design and implementation decision were made:

- GPIO pin numbering is based on ESP32 chip numbering. Please have the manual/pin diagram of your board at hand to find correspondence between your board pins and actual ESP32 pins.
- All pins are supported by MicroPython but not all are usable on any given board. For example pins that are connected to external SPI flash should not be used, and a board may only expose a certain selection of pins.

8.1.2 Technical specifications and SoC datasheets

The datasheets and other reference material for ESP32 chip are available from the vendor site: https://www.espressif.com/en/support/download/documents?keys=esp32 . They are the primary reference for the chip technical specifications, capabilities, operating modes, internal functioning, etc.

For your convenience, some of technical specifications are provided below:

- Architecture: Xtensa Dual-Core 32-bit LX6
- CPU frequency: up to 240MHz
- Total RAM available: 528KB (part of it reserved for system)
- BootROM: 448KB
- Internal FlashROM: none
- External FlashROM: code and data, via SPI Flash; usual size 4MB
- GPIO: 34 (GPIOs are multiplexed with other functions, including external FlashROM, UART, etc.)
- UART: 3 RX/TX UART (no hardware handshaking), one TX-only UART
- SPI: 4 SPI interfaces (one used for FlashROM)
- I2C: 2 I2C (bitbang implementation available on any pins)
- I2S: 2
- ADC: 12-bit SAR ADC up to 18 channels
- DAC: 2 8-bit DACs
- RMT: 8 channels allowing accurate pulse transmit/receive
- Programming: using BootROM bootloader from UART - due to external FlashROM and always-available BootROM bootloader, the ESP32 is not brickable

For more information see the ESP32 datasheet: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

MicroPython is implemented on top of the ESP-IDF, Espressif's development framework for the ESP32. This is a FreeRTOS based system. See the ESP-IDF Programming Guide for details.

8.2 MicroPython tutorial for ESP32

This tutorial is intended to get you started using MicroPython on the ESP32 system-on-a-chip. If it is your first time it is recommended to follow the tutorial through in the order below. Otherwise the sections are mostly self contained, so feel free to skip to those that interest you.

The tutorial does not assume that you know Python, but it also does not attempt to explain any of the details of the Python language. Instead it provides you with commands that are ready to run, and hopes that you will gain a bit of Python knowledge along the way. To learn more about Python itself please refer to https://www.python.org.

8.2.1 Getting started with MicroPython on the ESP32

Using MicroPython is a great way to get the most of your ESP32 board. And vice versa, the ESP32 chip is a great platform for using MicroPython. This tutorial will guide you through setting up MicroPython, getting a prompt, using WebREPL, connecting to the network and communicating with the Internet, using the hardware peripherals, and controlling some external components.

Let's get started!

Requirements

The first thing you need is a board with an ESP32 chip. The MicroPython software supports the ESP32 chip itself and any board should work. The main characteristic of a board is how the GPIO pins are connected to the outside world, and whether it includes a built-in USB-serial converter to make the UART available to your PC.

Names of pins will be given in this tutorial using the chip names (eg GPIO2) and it should be straightforward to find which pin this corresponds to on your particular board.

Powering the board

If your board has a USB connector on it then most likely it is powered through this when connected to your PC. Otherwise you will need to power it directly. Please refer to the documentation for your board for further details.

Getting the firmware

The first thing you need to do is download the most recent MicroPython firmware .bin file to load onto your ESP32 device. You can download it from the MicroPython download page. Search for your particular board on this page.

Note: If you don't see your specific board on the download page, then it's very likely that one of the generic firmwares will work. These are listed at the top of the download page and have names matching the onboard Espressif chip (i.e. ESP32 / WROOM, ESP32-C3, ESP32-S3, etc).

However, you may need to double check with the vendor you purchased the board from.

From here, you have a choice to make:

- Download a stable firmware release.
- Download a daily firmware "Preview" build.

If you are just starting with MicroPython, the best bet is to go for the stable Release firmware builds. If you are an advanced, experienced MicroPython ESP32 user who would like to follow development closely and help with testing new features, then you may find the Preview builds useful.

Deploying the firmware

Once you have the MicroPython firmware you need to load it onto your ESP32 device. There are two main steps to do this: first you need to put your device in bootloader mode, and second you need to copy across the firmware. The exact procedure for these steps is highly dependent on the particular board.

Detailed steps can be found on the same MicroPython download page for your board. It's recommended that you follow the steps on the download page, as they are customised for your particular board.

If the above commands run without error then MicroPython should be installed on your board! Skip ahead to *Serial prompt*.

Troubleshooting installation problems

If you experience problems during flashing or with running firmware immediately after flashing, here are some troubleshooting recommendations:

- Esptool will try to detect the serial port where your ESP32 is connected. If this doesn't work, or you have multiple serial ports, then you may need to manually specify the port by adding the `--port` option to the start of the `esptool` command line. For example, `esptool --port /dev/ttyUSB0 <rest of line>` for Linux or `esptool --port COM4 <rest of line>` for Windows.
- If the board isn't responding to esptool at all, it may need to be manually reset into the bootloader download mode. Look for a button marked "BOOT" or "IO0" on your board and a second button marked "RESET" or "RST". If you have both buttons, try these steps:

  1. Press "BOOT" (or "IO0") and hold it down.
  2. Press "RESET" (or "RST") and immediately release it.
  3. Release "BOOT" (or "IO0").
  4. Re-run the flashing steps from the download page.

  If your board doesn't have these buttons, consult the board manufacturer's documentation about entering bootloader download mode.
- If you get errors part-way through the flashing process then try reducing the speed of data transfer by removing the `--baud 460800` argument.
- Hardware problems can cause flashing to fail. There are two common problems: bad power source quality, and defective hardware (especially very low cost unbranded development boards). Speaking of power source, not just raw amperage is important, but also low ripple and noise/EMI in general. The most reliable and convenient power source is a USB port.
- If you still experience problems with flashing the firmware then please also refer to the esptool Troubleshooting documentation.

Serial prompt

Once you have the firmware on the device you can access the REPL (Python prompt) over either UART0, which might be connected to a USB-serial converter depending on your board, or the chip's built-in USB device. The baudrate is 115200.

From here you can now follow the ESP8266 tutorial, because these two Espressif chips are very similar when it comes to using MicroPython on them. The ESP8266 tutorial is found at *MicroPython tutorial for ESP8266* (but skip the Introduction section).

8.2.2 Pulse Width Modulation

Pulse width modulation (PWM) is a way to get an artificial analog output on a digital pin. It achieves this by rapidly toggling the pin from low to high. There are two parameters associated with this: the frequency of the toggling, and the duty cycle. The duty cycle is defined to be how long the pin is high compared with the length of a single period (low plus high time). Maximum duty cycle is when the pin is high all of the time, and minimum is when it is low all of the time.

- More comprehensive example with all **16 PWM channels and 8 timers**:

```
from time import sleep
from machine import Pin, PWM
try:
    F = 10000  # Hz
    D = 65536 // 16  # 6.25%
    pins = (2, 4, 12, 13, 14, 15, 16, 18, 19, 22, 23, 25, 26, 27, 32, 33)
    pwms = []
    for i, pin in enumerate(pins):
        f = F * (i // 2 + 1)
        d = min(65535, D * (i + 1))
        pwms.append(PWM(pin, freq=f, duty_u16=d))
        sleep(2 / f)
        print(pwms[i])
finally:
    for pwm in pwms:
        try:
            pwm.deinit()
        except:
            pass
```

Output is:

```
PWM(Pin(2), freq=10000, duty_u16=4096)
PWM(Pin(4), freq=10000, duty_u16=8192)
PWM(Pin(12), freq=20000, duty_u16=12288)
PWM(Pin(13), freq=20000, duty_u16=16384)
PWM(Pin(14), freq=30030, duty_u16=20480)
PWM(Pin(15), freq=30030, duty_u16=24576)
PWM(Pin(16), freq=40000, duty_u16=28672)
PWM(Pin(18), freq=40000, duty_u16=32768)
PWM(Pin(19), freq=50000, duty_u16=36864)
PWM(Pin(22), freq=50000, duty_u16=40960)
PWM(Pin(23), freq=60060, duty_u16=45056)
PWM(Pin(25), freq=60060, duty_u16=49152)
PWM(Pin(26), freq=69930, duty_u16=53248)
PWM(Pin(27), freq=69930, duty_u16=57344)
PWM(Pin(32), freq=80000, duty_u16=61440)
PWM(Pin(33), freq=80000, duty_u16=65535)
```

- Example of a **smooth frequency change**:

```
from time import sleep
from machine import Pin, PWM

F_MIN = 1000
F_MAX = 10000

f = F_MIN
delta_f = F_MAX // 50

pwm = PWM(Pin(27), f)

while True:
    pwm.freq(f)
    sleep(1 / f)
    sleep(0.1)
    print(pwm)

    f += delta_f
    if f > F_MAX or f < F_MIN:
        delta_f = -delta_f
        print()
        if f > F_MAX:
            f = F_MAX
        elif f < F_MIN:
            f = F_MIN
```

See PWM wave on Pin(27) with an oscilloscope.

Output is:

```
PWM(Pin(27), freq=998, duty_u16=32768)
PWM(Pin(27), freq=1202, duty_u16=32768)
PWM(Pin(27), freq=1401, duty_u16=32768)
PWM(Pin(27), freq=1598, duty_u16=32768)
...
PWM(Pin(27), freq=9398, duty_u16=32768)
PWM(Pin(27), freq=9615, duty_u16=32768)
PWM(Pin(27), freq=9804, duty_u16=32768)
PWM(Pin(27), freq=10000, duty_u16=32768)

PWM(Pin(27), freq=10000, duty_u16=32768)
PWM(Pin(27), freq=9804, duty_u16=32768)
PWM(Pin(27), freq=9615, duty_u16=32768)
PWM(Pin(27), freq=9398, duty_u16=32768)
...
PWM(Pin(27), freq=1598, duty_u16=32768)
PWM(Pin(27), freq=1401, duty_u16=32768)
PWM(Pin(27), freq=1202, duty_u16=32768)
PWM(Pin(27), freq=998, duty_u16=32768)
```

- Example of a **smooth duty change**:

```
from time import sleep
from machine import Pin, PWM

DUTY_MAX = 65535
```

(continues on next page)

(continued from previous page)

```
duty_u16 = 0
delta_d = 256

pwm = PWM(Pin(27), freq=1000, duty_u16=duty_u16)

while True:
    pwm.duty_u16(duty_u16)
    sleep(2 / pwm.freq())
    print(pwm)

    if duty_u16 >= DUTY_MAX:
        print()
        sleep(2)
    elif duty_u16 <= 0:
        print()
        sleep(2)

    duty_u16 += delta_d
    if duty_u16 >= DUTY_MAX:
        duty_u16 = DUTY_MAX
        delta_d = -delta_d
    elif duty_u16 <= 0:
        duty_u16 = 0
        delta_d = -delta_d
```

PWM wave on Pin(27) with an oscilloscope.

Output is:

```
PWM(Pin(27), freq=998, duty_u16=0)
PWM(Pin(27), freq=998, duty_u16=256)
PWM(Pin(27), freq=998, duty_u16=512)
PWM(Pin(27), freq=998, duty_u16=768)
PWM(Pin(27), freq=998, duty_u16=1024)
...
PWM(Pin(27), freq=998, duty_u16=64512)
PWM(Pin(27), freq=998, duty_u16=64768)
PWM(Pin(27), freq=998, duty_u16=65024)
PWM(Pin(27), freq=998, duty_u16=65280)
PWM(Pin(27), freq=998, duty_u16=65535)

PWM(Pin(27), freq=998, duty_u16=65279)
PWM(Pin(27), freq=998, duty_u16=65023)
PWM(Pin(27), freq=998, duty_u16=64767)
PWM(Pin(27), freq=998, duty_u16=64511)
...
PWM(Pin(27), freq=998, duty_u16=1023)
PWM(Pin(27), freq=998, duty_u16=767)
PWM(Pin(27), freq=998, duty_u16=511)
PWM(Pin(27), freq=998, duty_u16=255)
PWM(Pin(27), freq=998, duty_u16=0)
```

- Example of a **smooth duty change and PWM output inversion**:

```
from utime import sleep
from machine import Pin, PWM
```

(continues on next page)

(continued from previous page)

```
try:
    DUTY_MAX = 65535

    duty_u16 = 0
    delta_d = 65536 // 32

    pwm = PWM(Pin(27))
    pwmi = PWM(Pin(32), invert=1)

    while True:
        pwm.duty_u16(duty_u16)
        pwmi.duty_u16(duty_u16)

        duty_u16 += delta_d
        if duty_u16 >= DUTY_MAX:
            duty_u16 = DUTY_MAX
            delta_d = -delta_d
        elif duty_u16 <= 0:
            duty_u16 = 0
            delta_d = -delta_d

        sleep(.01)
        print(pwm)
        print(pwmi)

finally:
    try:
        pwm.deinit()
    except:
        pass
    try:
        pwmi.deinit()
    except:
        pass
```

Output is:

```
PWM(Pin(27), freq=5000, duty_u16=0)
PWM(Pin(32), freq=5000, duty_u16=32768, invert=1)
PWM(Pin(27), freq=5000, duty_u16=2048)
PWM(Pin(32), freq=5000, duty_u16=2048, invert=1)
PWM(Pin(27), freq=5000, duty_u16=4096)
PWM(Pin(32), freq=5000, duty_u16=4096, invert=1)
PWM(Pin(27), freq=5000, duty_u16=6144)
PWM(Pin(32), freq=5000, duty_u16=6144, invert=1)
PWM(Pin(27), freq=5000, duty_u16=8192)
PWM(Pin(32), freq=5000, duty_u16=8192, invert=1)
...
```

See PWM waves on Pin(27) and Pin(32) with an oscilloscope.

Note: New PWM parameters take effect in the next PWM cycle.

pwm = PWM(2, duty=512) print(pwm) >>> PWM(Pin(2), freq=5000, duty=1023) # the duty is not relevant pwm.init(freq=2, duty=64) print(pwm) >>> PWM(Pin(2), freq=2, duty=16) # the duty is not relevant time.sleep(1 / 2) # wait one PWM period print(pwm) >>> PWM(Pin(2), freq=2, duty=64) # the duty is actual

Note: machine.freq(20_000_000) reduces the highest PWM frequency to 10 MHz.

Note: the Pin.OUT mode does not need to be specified. The channel is initialized to PWM mode internally once for each Pin that is passed to the PWM constructor.

The following code is wrong:

```
pwm = PWM(Pin(5, Pin.OUT), freq=1000, duty=512)  # Pin(5) in PWM mode here
pwm = PWM(Pin(5, Pin.OUT), freq=500, duty=256)  # Pin(5) in OUT mode here, PWM is off
```

Use this code instead:

```
pwm = PWM(Pin(5), freq=1000, duty=512)
pwm.init(freq=500, duty=256)
```

8.2.3 Accessing peripherals directly via registers

The ESP32's peripherals can be controlled via direct register reads and writes. This requires reading the datasheet to know what registers to use and what values to write to them. The following example shows how to turn on and change the prescaler of the MCPWM0 peripheral.

```
from micropython import const
from machine import mem32

# Define the register addresses that will be used.
DR_REG_DPORT_BASE = const(0x3FF00000)
DPORT_PERIP_CLK_EN_REG = const(DR_REG_DPORT_BASE + 0x0C0)
DPORT_PERIP_RST_EN_REG = const(DR_REG_DPORT_BASE + 0x0C4)
DPORT_PWM0_CLK_EN = const(1 << 17)
MCPWM0 = const(0x3FF5E000)
MCPWM1 = const(0x3FF6C000)

# Enable CLK and disable RST.
print(hex(mem32[DPORT_PERIP_CLK_EN_REG] & 0xffffffff))
print(hex(mem32[DPORT_PERIP_RST_EN_REG] & 0xffffffff))
mem32[DPORT_PERIP_CLK_EN_REG] |= DPORT_PWM0_CLK_EN
mem32[DPORT_PERIP_RST_EN_REG] &= ~DPORT_PWM0_CLK_EN
print(hex(mem32[DPORT_PERIP_CLK_EN_REG] & 0xffffffff))
print(hex(mem32[DPORT_PERIP_RST_EN_REG] & 0xffffffff))

# Change the MCPWM0 prescaler.
print(hex(mem32[MCPWM0])) # read PWM_CLK_CFG_REG (reset value = 0)
mem32[MCPWM0] = 0x55      # change PWM_CLK_PRESCALE
print(hex(mem32[MCPWM0])) # read PWM_CLK_CFG_REG
```

The specific addresses will be different on different ESP32 models. For example, ESP32-S3 uses these values:

```
DR_REG_DPORT_BASE = const(0x600C_0000)
DPORT_PERIP_CLK_EN0_REG = const(DR_REG_DPORT_BASE + 0x0018)
DPORT_PERIP_RST_EN0_REG = const(DR_REG_DPORT_BASE + 0x0020)
DPORT_PWM0_CLK_EN = const(1 << 17)
MCPWM0 = const(0x6001_E000 + 0x0004)
...
```

Note that before a peripheral can be used its clock must be enabled and it must be taken out of reset. In the above example the following registers are used for this:

- `DPORT_PERI_CLK_EN_REG`: used to enable a peripheral clock

- DPORT_PERI_RST_EN_REG: used to reset (or take out of reset) a peripheral

The MCPWM0 peripheral is in bit position 17 of the above two registers, hence the value of DPORT_PWM0_CLK_EN.

Synchronous access to pins directly via registers

The following code shows how to access pins directly via registers. It has been tested on a generic ESP32 board. It configures pins 16, 17, 32 and 33 in output mode via registers, and switches pin output values via registers. Pins 16 and 17 are switched simultaneously.

```
from micropython import const
from machine import mem32, Pin

GPIO_OUT_REG = const(0x3FF44004)  # GPIO 0-31 output register
GPIO_OUT1_REG = const(0x3FF44010)  # GPIO 32-39 output register

GPIO_ENABLE_REG = const(0x3FF44020)  # GPIO 0-31 output enable register
GPIO_ENABLE1_REG = const(0x3FF4402C)  # GPIO 32-39 output enable register

M16 = 1 << 16  # Pin(16) bit mask
M17 = 1 << 17  # Pin(17) bit mask

M32 = 1 << (32-32)  # Pin(32) bit mask
M33 = 1 << (33-32)  # Pin(33) bit mask

# Enable pin output mode like
# p16 = Pin(16, mode=Pin.OUT)
# p17 = Pin(17, mode=Pin.OUT)
# p32 = Pin(32, mode=Pin.OUT)
# p33 = Pin(33, mode=Pin.OUT)
mem32[GPIO_ENABLE_REG] = mem32[GPIO_ENABLE_REG] | M16 | M17
mem32[GPIO_ENABLE1_REG] = mem32[GPIO_ENABLE1_REG] | M32 | M33

print(hex(mem32[GPIO_OUT_REG]), hex(mem32[GPIO_OUT1_REG]))

# Set outputs to 1 like
# p16(1)
# p17(1)
# p32(1)
# p33(1)
mem32[GPIO_OUT_REG] = mem32[GPIO_OUT_REG] | M16 | M17
mem32[GPIO_OUT1_REG] = mem32[GPIO_OUT1_REG] | M32 | M33

print(hex(mem32[GPIO_OUT_REG]), hex(mem32[GPIO_OUT1_REG]))

# Set outputs to 0 like
# p16(0)
# p17(0)
# p32(0)
# p33(0)
mem32[GPIO_OUT_REG] = mem32[GPIO_OUT_REG] & ~(M16 | M17)
mem32[GPIO_OUT1_REG] = mem32[GPIO_OUT1_REG] & ~(M32 | M33)

print(hex(mem32[GPIO_OUT_REG]), hex(mem32[GPIO_OUT1_REG]))

while True:
    # Set outputs to 1
```

(continues on next page)

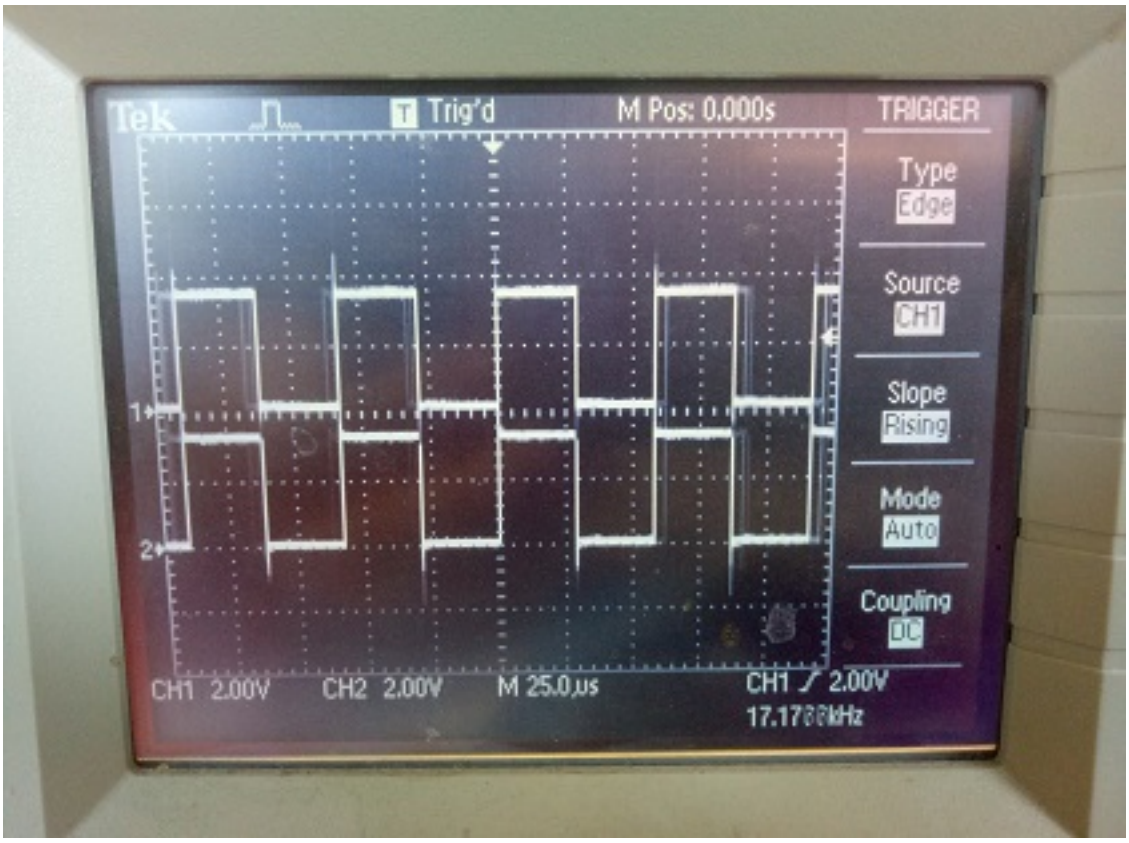
(continued from previous page)

```
    mem32[GPIO_OUT_REG] = mem32[GPIO_OUT_REG] | M16 | M17
    mem32[GPIO_OUT1_REG] = mem32[GPIO_OUT1_REG] | M32 | M33

    # Set outputs to 0
    mem32[GPIO_OUT_REG] = mem32[GPIO_OUT_REG] & ~(M16 | M17)
    mem32[GPIO_OUT1_REG] = mem32[GPIO_OUT1_REG] & ~(M32 | M33)
```

Output is:

```
0x0 0x0
0x30000 0x3
0x0 0x0
```

Pins 16 and 17 are switched synchronously:



Same image on pins 32 and 33.

Note that pins 34-36 and 39 are inputs only. Also pins 1 and 3 are Tx, Rx of the REPL UART, pins 6-11 are connected to the built-in SPI flash.

8.2.4 Factory reset

If something unexpected happens and your ESP32-based board no longer boots MicroPython, then you may have to factory reset it. For more details, see *Soft Bricking (failure to boot)*.

Factory resetting the MicroPython esp32 port involves fully erasing the flash and resetting the flash memory, so you will need to re-flash the MicroPython firmware afterwards and copy any Python files to the filesystem again.

1. You will need the Espressif esptool installed on your system. This is the same tool that you may have used to initially install MicroPython on your board (see *installation instructions*).
2. Find the serial port name of your board, and then use esptool to erase the entire flash contents:

```
esptool.py -p PORTNAME erase_flash
```

3. Use esptool to flash the MicroPython file to your board again. If needed, this file and flashing instructions can be found on the MicroPython downloads page.

Note that there are several varieties of ESP32 – ESP32, ESP32C3, ESP32C6, ESP32S2, ESP32S3 – supported by MicroPython, with some differences in functionality between them.

8.3 Installing MicroPython

See the corresponding section of tutorial: *Getting started with MicroPython on the ESP32*. It also includes a troubleshooting subsection.

8.4 General board control

The MicroPython REPL is on UART0 (GPIO1=TX, GPIO3=RX) at baudrate 115200. Tab-completion is useful to find out what methods an object has. Paste mode (ctrl-E) is useful to paste a large slab of Python code into the REPL.

The *machine* module:

```
import machine

machine.freq()          # get the current frequency of the CPU
machine.freq(240000000) # set the CPU frequency to 240 MHz
```

The *esp* module:

```
import esp

esp.osdebug(None)       # turn off vendor O/S debugging messages
esp.osdebug(0)          # redirect vendor O/S debugging messages to UART(0)

# low level methods to interact with flash storage
esp.flash_size()
esp.flash_user_start()
esp.flash_erase(sector_no)
esp.flash_write(byte_offset, buffer)
esp.flash_read(byte_offset, buffer)
```

The *esp32* module:

```
import esp32

esp32.raw_temperature() # read the internal temperature of the MCU, in Fahrenheit
esp32.ULP()             # access to the Ultra-Low-Power Co-processor, not on ESP32C3/
↪C6
```

Note that the temperature sensor in the ESP32 will typically read higher than ambient due to the IC getting warm while it runs. This effect can be minimised by reading the temperature sensor immediately after waking up from sleep.

ESP32C3, ESP32C6, ESP32S2, and ESP32S3 also have an internal temperature sensor available. It is implemented a bit differently to the ESP32 and returns the temperature in Celsius:

```
esp32.mcu_temperature() # read the internal temperature of the MCU, in Celsius
```

8.5 Networking

8.5.1 WLAN

The *network.WLAN* class in the *network* module:

```
import network

wlan = network.WLAN()       # create station interface (the default, see below for an␣
↪access point interface)
wlan.active(True)           # activate the interface
wlan.scan()                 # scan for access points
wlan.isconnected()          # check if the station is connected to an AP
wlan.connect('ssid', 'key') # connect to an AP
wlan.config('mac')          # get the interface's MAC address
wlan.ipconfig('addr4')      # get the interface's IPv4 addresses

ap = network.WLAN(network.WLAN.IF_AP) # create access-point interface
ap.config(ssid='ESP-AP')              # set the SSID of the access point
ap.config(max_clients=10)             # set how many clients can connect to the␣
↪network
ap.active(True)                       # activate the interface
```

A useful function for connecting to your local WiFi network is:

```
def do_connect():
    import machine, network
    wlan = network.WLAN()
    wlan.active(True)
    if not wlan.isconnected():
        print('connecting to network...')
        wlan.connect('ssid', 'key')
        while not wlan.isconnected():
            machine.idle()
    print('network config:', wlan.ipconfig('addr4'))
```

Once the network is established the *socket* module can be used to create and use TCP/UDP sockets as usual, and the `requests` module for convenient HTTP requests.

After a call to `wlan.connect()`, the device will by default retry to connect **forever**, even when the authentication failed or no AP is in range. `wlan.status()` will return `network.STAT_CONNECTING` in this state until a connection

succeeds or the interface gets disabled. This can be changed by calling `wlan.config(reconnects=n)`, where n are the number of desired reconnect attempts (0 means it won't retry, -1 will restore the default behaviour of trying to reconnect forever).

8.5.2 LAN

Built-in MAC (original ESP32)

The original ESP32 SoC has a built-in Ethernet MAC. Using this MAC requires an external Ethernet PHY to be wired to the chip's EMAC pins. Most of the EMAC pin assignments are fixed, consult the ESP32 datasheet for details.

If the PHY is connected, the internal Ethernet MAC can be configured via the *network.LAN* constructor:

```
import network

lan = network.LAN(mdc=PIN_MDC, ...)    # Set the pin and mode configuration
lan.active(True)                       # activate the interface
lan.ipconfig('addr4')                  # get the interface's IPv4 addresses
```

Required keyword arguments for the constructor:

- `mdc` and `mdio` - *machine.Pin* objects (or integers) specifying the MDC and MDIO pins.
- `phy_type` - Select the PHY device type. Supported devices are `PHY_GENERIC`, `PHY_LAN8710`, `PHY_LAN8720`, `PHY_IP101`, `PHY_RTL8201`, `PHY_DP83848`, `PHY_KSZ8041` and `PHY_KSZ8081`. These values are all constants defined in the `network` module.
- `phy_addr` - The address number of the PHY device. Must be an integer in the range 0x00 to 0x1f, inclusive. Common values are `0` and `1`.

All of the above keyword arguments must be present to configure the interface.

Optional keyword arguments:

- `reset` - *machine.Pin* object (or integer) specifying the PHY reset pin.
- `power` - *machine.Pin* object (or integer) specifying a pin which switches the power of the PHY device.
- `ref_clk` - *machine.Pin* object (or integer) specifying the pin used for the EMAC `ref_clk` signal. If not specified, the board default is used (typically GPIO 0, but may be different if a particular board has Ethernet.)
- `ref_clk_mode` - Defines whether the EMAC `ref_clk` pin of the ESP32 should be an input or an output. Suitable values are `machine.Pin.IN` and `machine.Pin.OUT`. If not specified, the board default is used (typically input, but may be different if a particular board has Ethernet.)

These are working configurations for LAN interfaces of some popular ESP32 boards:

```
# Olimex ESP32-GATEWAY: power controlled by Pin(5)
# Olimex ESP32 PoE and ESP32-PoE ISO: power controlled by Pin(12)

lan = network.LAN(mdc=machine.Pin(23), mdio=machine.Pin(18), power=machine.Pin(5),
                  phy_type=network.PHY_LAN8720, phy_addr=0,
                  ref_clk=machine.Pin(17), ref_clk_mode=machine.Pin.OUT)

# Wireless-Tag's WT32-ETH01

lan = network.LAN(mdc=machine.Pin(23), mdio=machine.Pin(18),
                  phy_type=network.PHY_LAN8720, phy_addr=1, power=None)
```

(continues on next page)

(continued from previous page)

```
# Wireless-Tag's WT32-ETH01 v1.4

lan = network.LAN(mdc=machine.Pin(23), mdio=machine.Pin(18),
                  phy_type=network.PHY_LAN8720, phy_addr=1,
                  power=machine.Pin(16))

# Espressif ESP32-Ethernet-Kit_A_V1.2

lan = network.LAN(id=0, mdc=machine.Pin(23), mdio=machine.Pin(18), power=machine.
↪Pin(5),
                  phy_type=network.PHY_IP101, phy_addr=1)

# ESP32-WROOM-32UE with KSZ8863RLL (Integrated 3-Port 10/100 Managed Switch with PHYs)

lan = network.LAN(mdc=machine.Pin(23),  # connected to SCL_MDC pin of KSZ8863RLL
                  mdio=machine.Pin(15),  # connected to SDA_MDIO pin of KSZ8863RLL
                  power=machine.Pin(16),  # connected to RSTN pin of KSZ8863RLL
                  phy_type=network.PHY_GENERIC,
                  phy_addr=3,
                  ref_clk_mode=machine.Pin.IN,
                  ref_clk=machine.Pin(0))  # REF_CLK 50MHz from KSZ8863RLL
```

SPI Ethernet Interface

All ESP32 SoCs support external SPI Ethernet interface chips. These are Ethernet interfaces that connect via a SPI bus, rather than an Ethernet RMII interface.

Note: The only exception is the ESP32 `d2wd` variant, where this feature is disabled to save code size.

SPI Ethernet uses the same *`network.LAN`* constructor, with a different set of keyword arguments:

```
import machine, network

spi = machine.SPI(1, sck=SCK_PIN, mosi=MOSI_PIN, miso=MISO_PIN)
lan = network.LAN(spi=spi, cs=CS_PIN, ...)    # Set the pin and mode configuration
lan.active(True)                              # activate the interface
lan.ipconfig('addr4')                         # get the interface's IPv4 addresses
```

Required keyword arguments for the constructor:

- `spi` - Should be a *`machine.SPI`* object configured for this connection. Note that any clock speed configured on the SPI object is ignored, the SPI Ethernet clock speed is configured at compile time.
- `cs` - *`machine.Pin`* object (or integer) specifying the CS pin connected to the interface.
- `int` - *`machine.Pin`* object (or integer) specifying the INT pin connected to the interface.
- `phy_type` - Select the SPI Ethernet interface type. Supported devices are `PHY_KSZ8851SNL`, `PHY_DM9051`, `PHY_W5500`. These values are all constants defined in the `network` module.
- `phy_addr` - The address number of the PHY device. Must be an integer in the range 0x00 to 0x1f, inclusive. This is usually `0` for SPI Ethernet devices.

All of the above keyword arguments must be present to configure the interface.

Optional keyword arguments for the constructor:

- `reset` - *machine.Pin* object (or integer) specifying the SPI Ethernet interface reset pin.
- `power` - *machine.Pin* object (or integer) specifying a pin which switches the power of the SPI Ethernet interface.

Here is a sample configuration for a WIZNet W5500 chip connected to pins on an ESP32-S3 development board:

```
import machine, network
from machine import Pin, SPI

spi = SPI(1, sck=Pin(12), mosi=Pin(13), miso=Pin(14))
lan = network.LAN(spi=spi, phy_type=network.PHY_W5500, phy_addr=0,
                  cs=Pin(10), int=Pin(11))
```

Note: WIZnet W5500 Ethernet is also supported on some other MicroPython ports, but using a *different software interface*.

8.6 Delay and timing

Use the *time* module:

```
import time

time.sleep(1)           # sleep for 1 second
time.sleep_ms(500)      # sleep for 500 milliseconds
time.sleep_us(10)       # sleep for 10 microseconds
start = time.ticks_ms() # get millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start) # compute time difference
```

8.7 Timers

The ESP32 port has one, two or four hardware timers, depending on the ESP32 device type. There is 1 timer for ESP32C2, 2 timers for ESP32C4, ESP32C6 and ESP32H4, and 4 timers otherwise. Use the *machine.Timer* class with a timer ID of 0, 0 and 1, or from 0 to 3 (inclusive):

```
from machine import Timer

tim0 = Timer(0)
tim0.init(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(0))

tim1 = Timer(1)
tim1.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(1))
```

The period is in milliseconds.

Timer callbacks are scheduled as soft interrupts on this port; hard callbacks are not implemented. Specifying `hard=True` will raise a ValueError.

Virtual timers are not currently supported on this port.

8.8 Pins and GPIO

Use the *machine.Pin* class:

```
from machine import Pin

p0 = Pin(0, Pin.OUT)    # create output pin on GPIO0
p0.on()                 # set pin to "on" (high) level
p0.off()                # set pin to "off" (low) level
p0.value(1)             # set pin to on/high

p2 = Pin(2, Pin.IN)     # create input pin on GPIO2
print(p2.value())       # get value, 0 or 1

p4 = Pin(4, Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor
p5 = Pin(5, Pin.OUT, value=1) # set pin high on creation
p6 = Pin(6, Pin.OUT, drive=Pin.DRIVE_3) # set maximum drive strength
```

Available Pins are from the following ranges (inclusive): 0-19, 21-23, 25-27, 32-39. These correspond to the actual GPIO pin numbers of ESP32 chip. Note that many end-user boards use their own adhoc pin numbering (marked e.g. D0, D1, ...). For mapping between board logical pins and physical chip pins consult your board documentation.

Four drive strengths are supported, using the `drive` keyword argument to the `Pin()` constructor or `Pin.init()` method, with different corresponding safe maximum source/sink currents and approximate internal driver resistances:

- `Pin.DRIVE_0`: 5mA / 130 ohm
- `Pin.DRIVE_1`: 10mA / 60 ohm
- `Pin.DRIVE_2`: 20mA / 30 ohm (default strength if not configured)
- `Pin.DRIVE_3`: 40mA / 15 ohm

The `hold=` keyword argument to `Pin()` and `Pin.init()` will enable the ESP32 "pad hold" feature. When set to `True`, the pin configuration (direction, pull resistors and output value) will be held and any further changes (including changing the output level) will not be applied. Setting `hold=False` will immediately apply any outstanding pin configuration changes and release the pin. Using `hold=True` while a pin is already held will apply any configuration changes and then immediately reapply the hold.

Notes:

- Pins 1 and 3 are REPL UART TX and RX respectively
- Pins 6, 7, 8, 11, 16, and 17 are used for connecting the embedded flash, and are not recommended for other uses
- Pins 34-39 are input only, and also do not have internal pull-up resistors
- See *Deep-sleep mode* for a discussion of pin behaviour during sleep

There's a higher-level abstraction *machine.Signal* which can be used to invert a pin. Useful for illuminating active-low LEDs using `on()` or `value(1)`.

8.9 UART (serial bus)

See *machine.UART*.

```
from machine import UART

uart1 = UART(1, baudrate=9600, tx=33, rx=32)
uart1.write('hello')  # write 5 bytes
uart1.read(5)         # read up to 5 bytes
```

The ESP32 has three hardware UARTs: UART0, UART1 and UART2. They each have default GPIO assigned to them, however depending on your ESP32 variant and board, these pins may conflict with embedded flash, onboard PSRAM or peripherals.

Any GPIO can be used for hardware UARTs using the GPIO matrix, except for input-only pins 34-39 that can be used as `rx`. To avoid conflicts simply provide `tx` and `rx` pins when constructing. The default pins listed below.

	UART0	UART1	UART2
tx	1	10	17
rx	3	9	16

On ESP32 with SPIRAM, the default pins for UART1 are `tx=5` and `rx=4` to avoid possible conflicts with the SPIRAM pins.

8.10 PWM (pulse width modulation)

PWM can be enabled on all output-enabled pins. The base frequency can range from 1Hz to 40MHz but there is a tradeoff; as the base frequency *increases* the duty resolution *decreases*. See LED Control for more details.

Use the *machine.PWM* class:

```
from machine import Pin, PWM, lightsleep

pwm0 = PWM(Pin(0), freq=5000, duty_u16=32768) # create PWM object from a pin
freq = pwm0.freq()          # get current frequency
pwm0.freq(1000)             # set PWM frequency from 1Hz to 40MHz

duty = pwm0.duty()          # get current duty cycle, range 0-1023 (default 512, 50%)
pwm0.duty(256)              # set duty cycle from 0 to 1023 as a ratio duty/1023, (now␣
↪25%)

duty_u16 = pwm0.duty_u16() # get current duty cycle, range 0-65535
pwm0.duty_u16(65536*3//4)  # set duty cycle from 0 to 65535 as a ratio duty_u16/65535,
↪ (now 75%)

duty_ns = pwm0.duty_ns()    # get current pulse width in ns
pwm0.duty_ns(250_000)       # set pulse width in nanoseconds from 0 to 1_000_000_000/
↪freq, (now 25%)

pwm0.deinit()               # turn off PWM on the pin

pwm2 = PWM(Pin(2), freq=20000, duty=512)  # create and configure in one go
```

(continues on next page)

(continued from previous page)

```
print(pwm2)                                 # view PWM settings
pwm2.deinit()                               # turn off PWM on the pin

pwm0 = PWM(Pin(0), duty_u16=16384)              # The output is at a high level 25% of␣
↪the time.
pwm2 = PWM(Pin(2), duty_u16=16384, invert=1)   # The output is at a low level 25% of␣
↪the time.

pwm4 = PWM(Pin(4), lightsleep=True)             # Allow PWM during light sleep mode

lightsleep(10*1000) # pwm0, pwm2 goes off, pwm4 stays on during 10s light sleep
                    # pwm0, pwm2, pwm4 on after 10s light sleep
```

ESP chips have different hardware peripherals:

Hardware specification	ESP32	ESP32-S2, ESP32-S3, ESP32-P4	ESP32-C2, ESP32-C3, ESP32-C5, ESP32-C6, ESP32-H2
Number of groups (speed modes)	2	1	1
Number of timers per group	4	4	4
Number of channels per group	8	8	6
Different PWM frequencies = (groups * timers)	8	4	4
Total PWM channels (Pins, duties) = (groups * channels)	16	8	6

In light sleep, the ESP32 PWM can only operate in low speed mode, so only 4 timers and 8 channels are available.

A maximum number of PWM channels (Pins) are available on the ESP32 - 16 channels, but only 8 different PWM frequencies are available, the remaining 8 channels must have the same frequency. On the other hand, 16 independent PWM duty cycles are possible at the same frequency.

See more examples in the *Pulse Width Modulation* tutorial.

8.11 DAC (digital to analog conversion)

On the ESP32, DAC functionality is available on pins 25, 26. On the ESP32S2, DAC functionality is available on pins 17, 18.

Use the DAC:

```
from machine import DAC, Pin

dac = DAC(Pin(25))  # create an DAC object acting on a pin
dac.write(128)      # set a raw analog value in the range 0-255, 50% now
```

8.12 ADC (analog to digital conversion)

On the ESP32, ADC functionality is available on pins 32-39 (ADC block 1) and pins 0, 2, 4, 12-15 and 25-27 (ADC block 2).

Use the *machine.ADC* class:

```
from machine import ADC

adc = ADC(pin)         # create an ADC object acting on a pin
val = adc.read_u16()   # read a raw analog value in the range 0-65535
val = adc.read_uv()    # read an analog value in microvolts
```

ADC block 2 is also used by WiFi and so attempting to read analog values from block 2 pins when WiFi is active will raise an exception.

The internal ADC reference voltage is typically 1.1V, but varies slightly from package to package. The ADC is less linear close to the reference voltage (particularly at higher attenuations) and has a minimum measurement voltage around 100mV, voltages at or below this will read as 0. To read voltages accurately, it is recommended to use the `read_uv()` method (see below).

ESP32-specific ADC class method reference:

class ADC(*pin*, *, *atten*)

Return the ADC object for the specified pin. ESP32 does not support different timings for ADC sampling and so the `sample_ns` keyword argument is not supported.

To read voltages above the reference voltage, apply input attenuation with the `atten` keyword argument. Valid values (and approximate linear measurement ranges) are:

- `ADC.ATTN_0DB`: No attenuation (100mV - 950mV)
- `ADC.ATTN_2_5DB`: 2.5dB attenuation (100mV - 1250mV)
- `ADC.ATTN_6DB`: 6dB attenuation (150mV - 1750mV)
- `ADC.ATTN_11DB`: 11dB attenuation (150mV - 2450mV)

Warning: Note that the absolute maximum voltage rating for input pins is 3.6V. Going near to this boundary risks damage to the IC!

`ADC.`**read_uv**()

This method uses the known characteristics of the ADC and per-package eFuse values - set during manufacture - to return a calibrated input voltage (before attenuation) in microvolts. The returned value has only millivolt resolution (i.e., will always be a multiple of 1000 microvolts).

The calibration is only valid across the linear range of the ADC. In particular, an input tied to ground will read as a value above 0 microvolts. Within the linear range, however, more accurate and consistent results will be obtained than using *read_u16()* and scaling the result with a constant.

The ESP32 port also supports the *machine.ADC* API:

class ADCBlock(*id*, *, *bits*)

Return the ADC block object with the given `id` (1 or 2) and initialize it to the specified resolution (9 to 12-bits depending on the ESP32 series) or the highest supported resolution if not specified.

`ADCBlock.`**connect**(*pin*)

`ADCBlock.`**connect**(*channel*)

ADCBlock.**connect**(*channel*, *pin*)

Return the ADC object for the specified ADC pin or channel number. Arbitrary connection of ADC channels to GPIO is not supported and so specifying a pin that is not connected to this block, or specifying a mismatched channel and pin, will raise an exception.

Legacy methods:

ADC.**read**()

This method returns the raw ADC value ranged according to the resolution of the block, e.g., 0-4095 for 12-bit resolution.

ADC.**atten**(*atten*)

Equivalent to `ADC.init(atten=atten)`.

ADC.**width**(*bits*)

Equivalent to `ADC.block().init(bits=bits)`.

The only chip that can switch resolution to a lower one is the normal esp32. The C2 & S3 are stuck at 12 bits, while the S2 is at 13 bits.

For compatibility, the ADC object also provides constants matching the supported ADC resolutions, per chip:

ESP32:

- `ADC.WIDTH_9BIT` = 9
- `ADC.WIDTH_10BIT` = 10
- `ADC.WIDTH_11BIT` = 11
- `ADC.WIDTH_12BIT` = 12

ESP32 C3 & S3:

- `ADC.WIDTH_12BIT` = 12

ESP32 S2:

- `ADC.WIDTH_13BIT` = 13

ADC.**deinit**()

Provided to deinit the adc driver.

8.13 Pulse Counter (pin pulse/edge counting)

The ESP32 provides up to 8 pulse counter peripherals depending on the hardware, with id 0..7. These can be configured to count rising and/or falling edges on any input pin.

Use the *esp32.PCNT* class:

```
from machine import Pin
from esp32 import PCNT

counter = PCNT(0, pin=Pin(2), rising=PCNT.INCREMENT)         # create counter
counter.start()                                              # start counter
count = counter.value()                                      # read count, -32768..
↪32767
counter.value(0)                                             # reset counter
count = counter.value(0)                                     # read and reset
```

The PCNT hardware supports monitoring multiple pins in a single unit to implement quadrature decoding or up/down signal counters.

See the *machine.Counter* and *machine.Encoder* classes for simpler abstractions of common pulse counting applications:

```
from machine import Pin, Counter

counter = Counter(0, Pin(2))     # create a counter as above and start it
count = counter.value()          # read the count as an arbitrary precision signed␣
↪integer

encoder = Encoder(0, Pin(12), Pin(14))     # create an encoder and begin counting
count = encoder.value()                    # read the count as an arbitrary precision␣
↪signed integer
```

Note that the id passed to these `Counter()` and `Encoder()` objects must be a PCNT id.

8.14 Software SPI bus

Software SPI (using bit-banging) works on all pins, and is accessed via the *machine.SoftSPI* class:

```
from machine import Pin, SoftSPI

# construct a SoftSPI bus on the given pins
# polarity is the idle state of SCK
# phase=0 means sample on the first edge of SCK, phase=1 means the second
spi = SoftSPI(baudrate=100000, polarity=1, phase=0, sck=Pin(0), mosi=Pin(2),␣
↪miso=Pin(4))

spi.init(baudrate=200000) # set the baudrate

spi.read(10)            # read 10 bytes on MISO
spi.read(10, 0xff)      # read 10 bytes while outputting 0xff on MOSI

buf = bytearray(50)     # create a buffer
spi.readinto(buf)       # read into the given buffer (reads 50 bytes in this case)
spi.readinto(buf, 0xff) # read into the given buffer and output 0xff on MOSI

spi.write(b'12345')     # write 5 bytes on MOSI

buf = bytearray(4)      # create a buffer
spi.write_readinto(b'1234', buf) # write to MOSI and read from MISO into the buffer
spi.write_readinto(buf, buf) # write buf to MOSI and read MISO back into buf
```

Warning: Currently *all* of `sck`, `mosi` and `miso` *must* be specified when initialising Software SPI.

8.15 Hardware SPI bus

There are two hardware SPI channels that allow faster transmission rates (up to 80Mhz). These may be used on any IO pins that support the required direction and are otherwise unused (see *Pins and GPIO*) but if they are not configured to their default pins then they need to pass through an extra layer of GPIO multiplexing, which can impact their reliability at high speeds. Hardware SPI channels are limited to 40MHz when used on pins other than the default ones listed below.

	HSPI (id=1)	VSPI (id=2)
sck	14	18
mosi	13	23
miso	12	19

Hardware SPI is accessed via the *machine.SPI* class and has the same methods as software SPI above:

```
from machine import Pin, SPI

hspi = SPI(1, 10000000)
hspi = SPI(1, 10000000, sck=Pin(14), mosi=Pin(13), miso=Pin(12))
vspi = SPI(2, baudrate=80000000, polarity=0, phase=0, bits=8, firstbit=0, sck=Pin(18),
↪ mosi=Pin(23), miso=Pin(19))
```

8.16 Software I2C bus

Software I2C (using bit-banging) works on all output-capable pins, and is accessed via the *machine.SoftI2C* class:

```
from machine import Pin, SoftI2C

i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)

i2c.scan()              # scan for devices

i2c.readfrom(0x3a, 4)   # read 4 bytes from device with address 0x3a
i2c.writeto(0x3a, '12') # write '12' to device with address 0x3a

buf = bytearray(10)     # create a buffer with 10 bytes
i2c.writeto(0x3a, buf)  # write the given buffer to the peripheral
```

8.17 Hardware I2C bus

There are two hardware I2C peripherals with identifiers 0 and 1. Any available output-capable pins can be used for SCL and SDA but the defaults are given below.

	I2C(0)	I2C(1)
scl	18	25
sda	19	26

The driver is accessed via the *machine.I2C* class and has the same methods as software I2C above:

```
from machine import Pin, I2C

i2c = I2C(0)
i2c = I2C(1, scl=Pin(5), sda=Pin(4), freq=400000)
```

8.18 I2S bus

See *machine.I2S*.

```
from machine import I2S, Pin

i2s = I2S(0, sck=Pin(13), ws=Pin(14), sd=Pin(34), mode=I2S.TX, bits=16, format=I2S.
↪STEREO, rate=44100, ibuf=40000) # create I2S object
i2s.write(buf)              # write buffer of audio samples to I2S device

i2s = I2S(1, sck=Pin(33), ws=Pin(25), sd=Pin(32), mode=I2S.RX, bits=16, format=I2S.
↪MONO, rate=22050, ibuf=40000) # create I2S object
i2s.readinto(buf)           # fill buffer with audio samples from I2S device
```

The I2S class is currently available as a Technical Preview. During the preview period, feedback from users is encouraged. Based on this feedback, the I2S class API and implementation may be changed.

ESP32 has two I2S buses with id=0 and id=1

8.19 Real time clock (RTC)

See *machine.RTC*

```
from machine import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 0, 1, 12, 48, 0)) # set a specific date and
                                             # time, eg. 2017/8/23 1:12:48
                                             # the day-of-week value is ignored
rtc.datetime() # get date and time
```

8.20 WDT (Watchdog timer)

See *machine.WDT*.

```
from machine import WDT

# enable the WDT with a timeout of 5s (1s is the minimum)
wdt = WDT(timeout=5000)
wdt.feed()
```

8.21 Deep-sleep mode

The following code can be used to sleep, wake and check the reset cause:

```
import machine

# check if the device woke from a deep sleep
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
    print('woke from a deep sleep')

# put the device to sleep for 10 seconds
machine.deepsleep(10000)
```

Notes:

- Calling `deepsleep()` without an argument will put the device to sleep indefinitely
- A software reset does not change the reset cause

Some ESP32 pins (0, 2, 4, 12-15, 25-27, 32-39) are connected to the RTC during deep-sleep and can be used to wake the device with the `wake_on_` functions in the *esp32* module. The output-capable RTC pins (all except 34-39) will also retain their pull-up or pull-down resistor configuration when entering deep-sleep.

If the pull resistors are not actively required during deep-sleep and are likely to cause current leakage (for example a pull-up resistor is connected to ground through a switch), then they should be disabled to save power before entering deep-sleep mode:

```
from machine import Pin, deepsleep

# configure input RTC pin with pull-up on boot
pin = Pin(2, Pin.IN, Pin.PULL_UP)

# disable pull-up and put the device to sleep for 10 seconds
pin.init(pull=None)
machine.deepsleep(10000)
```

Output-configured RTC pins will also retain their output direction and level in deep-sleep if pad hold is enabled with the `hold=True` argument to `Pin.init()`.

Non-RTC GPIO pins will be disconnected by default on entering deep-sleep. Configuration of non-RTC pins - including output level - can be retained by enabling pad hold on the pin and enabling GPIO pad hold during deep-sleep:

```
from machine import Pin, deepsleep
import esp32

opin = Pin(19, Pin.OUT, value=1, hold=True) # hold output level
ipin = Pin(21, Pin.IN, Pin.PULL_UP, hold=True) # hold pull-up

# enable pad hold in deep-sleep for non-RTC GPIO
esp32.gpio_deep_sleep_hold(True)

# put the device to sleep for 10 seconds
deepsleep(10000)
```

The pin configuration - including the pad hold - will be retained on wake from sleep. See *Pins and GPIO* above for a further discussion of pad holding.

8.22 SD card

See *machine.SDCard*.

```
import machine, os, vfs

# On original ESP32, slot 2 uses pins sck=18, cs=5, miso=19, mosi=23
sd = machine.SDCard(slot=2)
vfs.mount(sd, '/sd') # mount

os.listdir('/sd')    # list directory contents

vfs.umount('/sd')    # eject
```

8.23 RMT

The RMT is ESP32-specific and allows generation of accurate digital pulses with 12.5ns resolution. See *esp32.RMT* for details. Usage is:

```
import esp32
from machine import Pin

r = esp32.RMT(pin=Pin(18), resolution_hz=10000000)
r   # RMT(pin=18, source_freq=80000000, resolution_hz=10000000)
# The channel resolution is based on resolution_hz, i.e. 100ns for 10000000
r.write_pulses((1, 20, 2, 40), 0) # Send 0 for 100ns, 1 for 2000ns, 0 for 200ns, 1␣
↪for 4000ns
```

The ESP32-C2 family does not include any RMT peripheral, so this class is unavailable on those SoCs.

8.24 OneWire driver

The OneWire driver is implemented in software and works on all pins:

```
from machine import Pin
import onewire

ow = onewire.OneWire(Pin(12)) # create a OneWire bus on GPIO12
ow.scan()               # return a list of devices on the bus
ow.reset()              # reset the bus
ow.readbyte()           # read a byte
ow.writebyte(0x12)      # write a byte on the bus
ow.write('123')         # write bytes on the bus
ow.select_rom(b'12345678') # select a specific device by its ROM code
```

There is a specific driver for DS18S20 and DS18B20 devices:

```
import time, ds18x20
ds = ds18x20.DS18X20(ow)
roms = ds.scan()
ds.convert_temp()
time.sleep_ms(750)
```

(continues on next page)

(continued from previous page)

```
for rom in roms:
    print(ds.read_temp(rom))
```

Be sure to put a 4.7k pull-up resistor on the data line. Note that the `convert_temp()` method must be called each time you want to sample the temperature.

8.25 NeoPixel and APA106 driver

Use the `neopixel` and `apa106` modules:

```
from machine import Pin
from neopixel import NeoPixel

pin = Pin(0, Pin.OUT)   # set GPIO0 to output to drive NeoPixels
np = NeoPixel(pin, 8)   # create NeoPixel driver on GPIO0 for 8 pixels
np[0] = (255, 255, 255) # set the first pixel to white
np.write()              # write data to all pixels
r, g, b = np[0]         # get first pixel colour
```

The APA106 driver extends NeoPixel, but internally uses a different colour order:

```
from apa106 import APA106
ap = APA106(pin, 8)
r, g, b = ap[0]
```

Warning: By default `NeoPixel` is configured to control the more popular *800kHz* units. It is possible to use alternative timing to control other (typically 400kHz) devices by passing `timing=0` when constructing the `NeoPixel` object.

For low-level driving of a NeoPixel see *`machine.bitstream`*. This low-level driver uses an RMT channel by default.

APA102 (DotStar) uses a different driver as it has an additional clock pin.

8.26 Capacitive touch

ESP32, ESP32-S2 and ESP32-S3 support capacitive touch via the `TouchPad` class in the `machine` module:

```
from machine import TouchPad, Pin

t = TouchPad(Pin(14))
t.read()              # Returns a smaller number when touched
```

`TouchPad.read` returns a value proportional to the capacitance between the pin and the board's Ground connection. On ESP32 the number becomes smaller when the pin (or connected touch pad) is touched, on ESP32-S2 and ESP32-S3 the number becomes larger when the pin is touched.

In all cases, a touch causes a significant change in the return value. Note the returned values are *relative* and can vary depending on the board and surrounding environment so some calibration (i.e. comparison to a baseline or rolling average) may be required.

Chip	Touch-enabled pins
ESP32	0, 2, 4, 12, 13, 14, 15, 27, 32, 33
ESP32-S2	1 to 14 inclusive
ESP32-S3	1 to 14 inclusive

Trying to assign to any other pins will result in a `ValueError`.

Note that TouchPads can be used to wake an ESP32 from sleep:

```
import machine
from machine import TouchPad, Pin
import esp32

t = TouchPad(Pin(14))
t.config(500)                # configure the threshold at which the pin is considered
↪touched
esp32.wake_on_touch(True)
machine.lightsleep()         # put the MCU to sleep until a touchpad is touched
```

For more details on touchpads refer to Espressif Touch Sensor.

8.27 DHT driver

The DHT driver is implemented in software and works on all pins:

```
import dht
import machine

d = dht.DHT11(machine.Pin(4))
d.measure()
d.temperature() # eg. 23 (°C)
d.humidity()    # eg. 41 (% RH)

d = dht.DHT22(machine.Pin(4))
d.measure()
d.temperature() # eg. 23.6 (°C)
d.humidity()    # eg. 41.3 (% RH)
```

8.28 WebREPL (web browser interactive prompt)

WebREPL (REPL over WebSockets, accessible via a web browser) is an experimental feature available in ESP32 port. Download web client from https://github.com/micropython/webrepl (hosted version available at http://micropython.org/webrepl), and configure it by executing:

```
import webrepl_setup
```

and following on-screen instructions. After reboot, it will be available for connection. If you disabled automatic start-up on boot, you may run configured daemon on demand using:

```
import webrepl
webrepl.start()

# or, start with a specific password
webrepl.start(password='mypass')
```

The WebREPL daemon listens on all active interfaces, which can be STA or AP. This allows you to connect to the ESP32 via a router (the STA interface) or directly when connected to its access point.

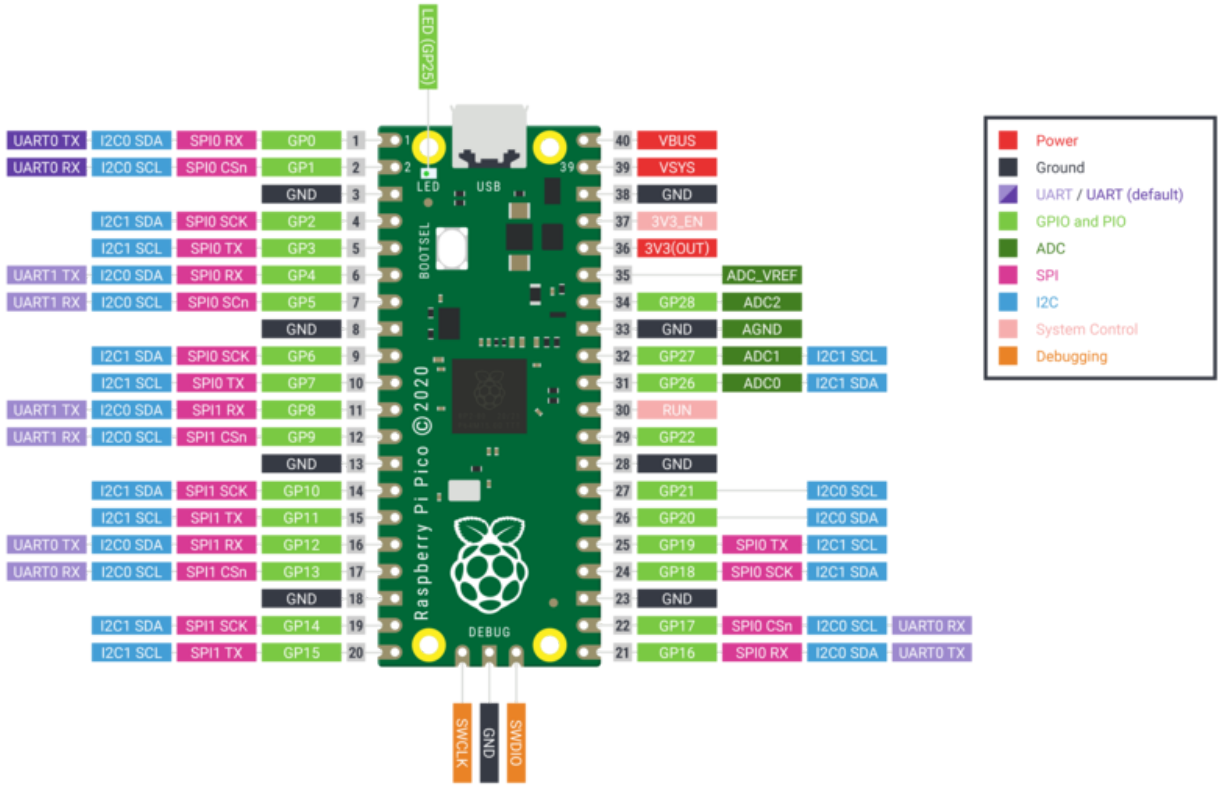
In addition to terminal/command prompt access, WebREPL also has provision for file transfer (both upload and download). The web client has buttons for the corresponding functions, or you can use the command-line client `webrepl_cli.py` from the repository above.

See the MicroPython forum for other community-supported alternatives to transfer files to an ESP32 board.

CHAPTER

NINE

QUICK REFERENCE FOR THE RP2



The Raspberry Pi Pico Development Board (image attribution: Raspberry Pi Foundation).

Below is a quick reference for Raspberry Pi RP2xxx boards. If it is your first time working with this board it may be useful to get an overview of the microcontroller:

9.1 General information about the RP2xxx port

The rp2 port supports boards powered by the Raspberry Pi Foundation's RP2xxx family of microcontrollers, including the RP2040 and RP2350.

9.1.1 Technical specifications and SoC datasheets

RP2040

The RP2040 microcontroller is manufactured on a 40 nm silicon process in a 7x7mm QFN-56 SMD package.

The key features include:

- 133 MHz dual ARM Cortex-M0+ cores (overclockable to over 400 MHz)
- 264KB SRAM in six independent banks
- No internal Flash or EEPROM memory (after reset, the bootloader loads firmware from either the external flash memory or USB bus into internal SRAM)
- QSPI bus controller, which supports up to 16 MB of external Flash memory
- On-chip programmable LDO to generate core voltage
- 2 on-chip PLLs to generate USB and core clocks
- 30 GPIO pins, of which 4 can optionally be used as analog inputs

The peripherals include:

- 2 UARTs
- 2 SPI controllers
- 2 I2C controllers
- 16 PWM channels
- USB 1.1 controller
- 8 PIO state machines

For detailed technical specifications, please refer to the rp2040-datasheet

RP2350

The RP2350 microcontroller is manufactured on a 40 nm silicon process and is available in QFN-60 (RP2350A) or QFN-80 (RP2350B) packages.

The key features include:

- Dual-core Arm Cortex-M33 or Hazard3 RISC-V processors at up to 150MHz
- 520KB on-chip SRAM
- Support for up to 16MB of off-chip Flash memory via QSPI
- USB 2.0 Full-Speed/Low-Speed controller
- 30 GPIO pins (RP2350A) or 48 GPIO pins (RP2350B)

The peripherals include:

- 2 UARTs
- 2 SPI controllers
- 2 I2C controllers
- 24 PWM channels
- 12 PIO state machines
- HSTX high-speed transmitter

For detailed technical specifications, please refer to the rp2350-datasheet

9.2 Getting started with MicroPython on the RP2xxx

Let's get started!

9.2.1 Factory reset

If something unexpected happens and your RP2xxx-based board no longer boots MicroPython, then you may have to factory reset it. For more details, see *Soft Bricking (failure to boot)*.

Factory resetting the MicroPython rp2 port involves fully erasing the flash and resetting the flash memory, so you will need to re-flash the MicroPython firmware afterwards and copy any Python files to the filesystem again.

1. Follow the instructions on the Raspberry Pi website for resetting flash memory.
2. Copy the MicroPython .uf2 firmware file to your board. If needed, this file can be found on the MicroPython downloads page.

9.2.2 Programmable IO

The RP2xxx family of microcontrollers provides hardware support for standard communication protocols like I2C, SPI and UART. For protocols where there is no hardware support, or where there is a requirement of custom I/O behaviour, Programmable Input Output (PIO) comes into play. Also, some MicroPython applications make use of a technique called bit banging in which pins are rapidly turned on and off to transmit data. This can make the entire process slow as the processor concentrates on bit banging rather than executing other logic. However, PIO allows bit banging to happen in the background while the CPU is executing the main work.

PIO State Machines

State machines transfer data to/from other entities using First-In-First-Out (FIFO) buffers, which allow the state machine and main processor to work independently yet also synchronise their data. The FIFOs can be linked to the DMA to transfer larger amounts of data without extensive CPU usage.

RP2040

The RP2040 has two PIO blocks, each with four independent state machines (total of 8).

Each FIFO has four words (each of 32 bits) per state machine (TX and RX).

RP2350

The RP2350 has three PIO blocks, each with four independent state machines (total of 12).

Similar to the RP2040, each state machine has a four-word FIFO per state machine (TX and RX), but they have some additional functions

For each state machine:

- TX or RX FIFOs can be combined into a single 8-word FIFO for higher bandwidth.
- The RX FIFO supports random read/write access to its registers.

Compatibility

RP2350 PIO is backward-compatible with RP2040, with some additional enhancements.

At the moment, native support for RP2350 PIO enhancements is very limited, but work for some features are already in progress

Instructions

Each PIO block has 32 instructions of memory shared among its four state machines.

All PIO instructions follow a common pattern:

```
<instruction> .side(<side_set_value>) [<delay_value>]
```

The side-set `.side(...)` and delay `[...]` parts are both optional, and if specified allow the instruction to perform more than one operation. This keeps PIO programs small and efficient.

There are nine instructions which perform the following tasks:

- `jmp()` transfers control to a different part of the code
- `wait()` pauses until a particular action happens
- `in_()` shifts the bits from a source (scratch register or set of pins) to the input shift register
- `out()` shifts the bits from the output shift register to a destination
- `push()` sends data to the RX FIFO
- `pull()` receives data from the TX FIFO
- `mov()` moves data from a source to a destination
- `irq()` sets or clears an IRQ flag
- `set()` writes a literal value to a destination

The instruction modifiers are:

- `.side()` sets the side-set pins at the start of the instruction

- `[]` delays for a certain number of cycles after execution of the instruction

There are also directives:

- `wrap_target()` specifies where the program execution will get continued from
- `wrap()` specifies the instruction where the control flow of the program will get wrapped from
- `label()` sets a label for use with `jmp()` instructions
- `word()` emits a raw 16-bit value which acts as an instruction in the program

An example

Take the `pio_1hz.py` example for a simple understanding of how to use the PIO and state machines. Below is the code for reference.

```
# Example using PIO to blink an LED and raise an IRQ at 1Hz.

import time
from machine import Pin
import rp2


@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink_1hz():
    # Cycles: 1 + 1 + 6 + 32 * (30 + 1) = 1000
    irq(rel(0))
    set(pins, 1)
    set(x, 31)                  [5]
    label("delay_high")
    nop()                       [29]
    jmp(x_dec, "delay_high")

    # Cycles: 1 + 1 + 6 + 32 * (30 + 1) = 1000
    nop()
    set(pins, 0)
    set(x, 31)                  [5]
    label("delay_low")
    nop()                       [29]
    jmp(x_dec, "delay_low")


# Create the StateMachine with the blink_1hz program, outputting on Pin(25).
sm = rp2.StateMachine(0, blink_1hz, freq=2000, set_base=Pin(25))

# Set the IRQ handler to print the millisecond timestamp.
sm.irq(lambda p: print(time.ticks_ms()))

# Start the StateMachine.
sm.active(1)
```

This creates an instance of class *rp2.StateMachine* which runs the `blink_1hz` program at 2000Hz, and connects to pin 25. The `blink_1hz` program uses the PIO to blink an LED connected to this pin at 1Hz, and also raises an IRQ as the LED turns on. This IRQ then calls the `lambda` function which prints out a millisecond timestamp.

The `blink_1hz` program is a PIO assembler routine. It connects to a single pin which is configured as an output and starts out low. The instructions do the following:

- `irq(rel(0))` raises the IRQ associated with the state machine.

- The LED is turned on via the `set(pins, 1)` instruction.
- The value 31 is put into register X, and then there is a delay for 5 more cycles, specified by the `[5]`.
- The `nop() [29]` instruction waits for 30 cycles.
- The `jmp(x_dec, "delay_high")` will keep looping to the `delay_high` label as long as the register X is non-zero, and will also post-decrement X. Since X starts with the value 31 this jump will happen 31 times, so the `nop() [29]` runs 32 times in total (note there is also one instruction cycle taken by the `jmp` for each of these 32 loops).
- The single `nop()` correlates with the cycle used for IRQ raise, and ensures the same number of cycles are used for LED on and LED off.
- `set(pins, 0)` will turn the LED off by setting pin 25 low.
- Another 32 loops of `nop() [29]` and `jmp(...)` will execute.
- Because `wrap_target()` and `wrap()` are not specified, their default will be used and execution of the program will wrap around from the bottom to the top. This wrapping does not cost any execution cycles.

The entire routine takes exactly 2000 cycles of the state machine. Setting the frequency of the state machine to 2000Hz makes the LED blink at 1Hz.

This example works identically on both RP2040 and RP2350 (using StateMachine ID 0). On RP2350, higher `freq` values are possible due to the faster system clock.

Compatibility Notes

PIO programs using the basic instructions shown here are fully compatible between RP2040 and RP2350. The MicroPython `rp2` module API (including `@rp2.asm_pio`, `StateMachine`, etc.) is the same, though RP2350 supports StateMachine IDs 0–11.

RP2350 offers additional advanced features (new instructions, FIFO modes, security, GPIO relocation) not covered in this introduction; see the RP2350 datasheet for details. Some programs may require minor adjustments due to differences in default GPIO reset states between the two chips.

9.3 Installing MicroPython

See the corresponding section of tutorial: *Getting started with MicroPython on the RP2xxx*. It also includes a troubleshooting subsection.

9.4 General board control

The MicroPython REPL is accessed via the USB serial port. Tab-completion is useful to find out what methods an object has. Paste mode (ctrl-E) is useful to paste a large slab of Python code into the REPL.

The *machine* module:

machine.freq() allows to change the MCU frequency and control the peripheral frequency for UART and SPI. Usage:

```
machine.freq(MCU_frequency[, peripheral_frequency=48_000_000])
```

The MCU frequency can be set in a range from less than 48 MHz to about 250MHz. The default at boot time is 125 MHz. The peripheral frequency must be either 48 MHz or identical to the MCU frequency, with 48 MHz as the default.

If the peripheral frequency is changed, any already existing instance of UART and SPI will change it's baud rate and may have to be re-configured:

```
import machine

machine.freq()          # get the current frequency of the CPU
machine.freq(240000000) # set the CPU frequency to 240 MHz and keep
                        # the UART frequency at 48MHz
machine.freq(125000000, 125000000) # set the CPU and UART frequency to 125 MHz
```

The *rp2* module:

```
import rp2
```

9.5 Networking

9.5.1 WLAN

Note: This section applies only to devices that include WiFi support, such as the Pico W and Pico 2 W.

The *network.WLAN* class in the *network* module:

```
import network

wlan = network.WLAN()       # create station interface (the default, see below for an␣
↪access point interface)
wlan.active(True)           # activate the interface
wlan.scan()                 # scan for access points
wlan.isconnected()          # check if the station is connected to an AP
wlan.connect('ssid', 'key') # connect to an AP
wlan.config('mac')          # get the interface's MAC address
wlan.ipconfig('addr4')      # get the interface's IPv4 addresses

ap = network.WLAN(network.WLAN.IF_AP) # create access-point interface
ap.config(ssid='RP2-AP')              # set the SSID of the access point
ap.config(max_clients=10)             # set how many clients can connect to the␣
↪network
ap.active(True)                       # activate the interface
```

A useful function for connecting to your local WiFi network is:

```
def do_connect():
    import machine, network
    wlan = network.WLAN()
    wlan.active(True)
    if not wlan.isconnected():
        print('connecting to network...')
        wlan.connect('ssid', 'key')
        while not wlan.isconnected():
            machine.idle()
    print('network config:', wlan.ipconfig('addr4'))
```

Once the network is established the *socket* module can be used to create and use TCP/UDP sockets as usual, and the `requests` module for convenient HTTP requests.

After a call to `wlan.connect()`, the device will by default retry to connect **forever**, even when the authentication failed or no AP is in range. `wlan.status()` will return `network.STAT_CONNECTING` in this state until a connection succeeds or the interface gets disabled.

9.6 Delay and timing

Use the *time* module:

```
import time

time.sleep(1)           # sleep for 1 second
time.sleep_ms(500)      # sleep for 500 milliseconds
time.sleep_us(10)       # sleep for 10 microseconds
start = time.ticks_ms() # get millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start) # compute time difference
```

9.7 Timers

RP2040's system timer peripheral provides a global microsecond timebase and generates interrupts for it. The software timer is available currently, and there are unlimited number of them (memory permitting). There is no need to specify the timer id (id=-1 is supported at the moment) as it will default to this.

Use the *machine.Timer* class:

```
from machine import Timer

tim = Timer(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(1))
tim.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(2))
```

By default, timer callbacks run as soft IRQs so they can allocate but are prone to GC jitter and delays. Pass `hard=True` to the `Timer()` constructor or `init()` method to run the callback in hard-IRQ context instead. This reduces delay and jitter, but see *Writing interrupt handlers* for the restrictions that apply to hard-IRQ handlers.

9.8 Pins and GPIO

Use the *machine.Pin* class:

```
from machine import Pin

p0 = Pin(0, Pin.OUT)    # create output pin on GPIO0
p0.on()                 # set pin to "on" (high) level
p0.off()                # set pin to "off" (low) level
p0.value(1)             # set pin to on/high

p2 = Pin(2, Pin.IN)     # create input pin on GPIO2
print(p2.value())       # get value, 0 or 1

p4 = Pin(4, Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor
p5 = Pin(5, Pin.OUT, value=1) # set pin high on creation
```

9.9 Programmable IO (PIO)

PIO is useful to build low-level IO interfaces from scratch. See the *rp2* module for detailed explanation of the assembly instructions.

Example using PIO to blink an LED at 1Hz:

```
from machine import Pin
import rp2

@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink_1hz():
    # Cycles: 1 + 7 + 32 * (30 + 1) = 1000
    set(pins, 1)
    set(x, 31)                  [6]
    label("delay_high")
    nop()                       [29]
    jmp(x_dec, "delay_high")

    # Cycles: 1 + 7 + 32 * (30 + 1) = 1000
    set(pins, 0)
    set(x, 31)                  [6]
    label("delay_low")
    nop()                       [29]
    jmp(x_dec, "delay_low")

# Create and start a StateMachine with blink_1hz, outputting on Pin(25)
sm = rp2.StateMachine(0, blink_1hz, freq=2000, set_base=Pin(25))
sm.active(1)
```

9.10 UART (serial bus)

There are two UARTs, UART0 and UART1. UART0 can be mapped to GPIO 0/1, 12/13 and 16/17, and UART1 to GPIO 4/5 and 8/9.

See *machine.UART*.

```
from machine import UART, Pin
uart1 = UART(1, baudrate=9600, tx=Pin(4), rx=Pin(5))
uart1.write('hello')  # write 5 bytes
uart1.read(5)         # read up to 5 bytes
```

It is possible to access the REPL over UART, but this is disabled by default. To duplicate the REPL stream over UART, use *os.dupterm()*. This code should be in *boot.py* or *main.py* to establish UART on boot.

```
from machine import UART
import os
uart = UART(0)
os.dupterm(uart, 0)
uart.irq(os.dupterm_notify, UART.IRQ_RXIDLE) # ensure inputs are handled
```

To use UART for REPL instead of the standard USB interface (for example if you are using *machine.USBDevice*), you will need to *build MicroPython from source*. Modify `ports/rp2/mpconfigport.h`, and change the `MICROPY_HW_ENABLE_UART_REPL` variable to `1`:

```
#define MICROPY_HW_ENABLE_UART_REPL             (1) // useful if there is no USB
```

9.11 PWM (pulse width modulation)

There are 8 independent PWM generators called slices, which each have two channels making it 16 PWM channels in total which can be clocked from 8Hz to 62.5Mhz at a machine.freq() of 125Mhz. The two channels of a slice run at the same frequency, but can have a different duty rate. The two channels are usually assigned to adjacent GPIO pin pairs with even/odd numbers. So GPIO0 and GPIO1 are at slice 0, GPIO2 and GPIO3 are at slice 1, and so on. A certain channel can be assigned to different GPIO pins (see Pinout). For instance slice 0, channel A can be assigned to both GPIO0 and GPIO16.

Use the *machine.PWM* class:

```
from machine import Pin, PWM

# create PWM object from a pin and set the frequency of slice 0
# and duty cycle for channel A
pwm0 = PWM(Pin(0), freq=2000, duty_u16=32768)
pwm0.freq()             # get the current frequency of slice 0
pwm0.freq(1000)         # set/change the frequency of slice 0
pwm0.duty_u16()         # get the current duty cycle of channel A, range 0-65535
pwm0.duty_u16(200)      # set the duty cycle of channel A, range 0-65535
pwm0.duty_u16(0)        # stop the output at channel A
print(pwm0)             # show the properties of the PWM object.
pwm0.deinit()           # turn off PWM of slice 0, stopping channels A and B
```

9.12 ADC (analog to digital conversion)

RP2040 has five ADC channels in total, four of which are 12-bit SAR based ADCs: GP26, GP27, GP28 and GP29. The input signal for ADC0, ADC1, ADC2 and ADC3 can be connected with GP26, GP27, GP28, GP29 respectively (On Pico board, GP29 is connected to VSYS). The standard ADC range is 0-3.3V. The fifth channel is connected to the in-built temperature sensor and can be used for measuring the temperature.

Use the *machine.ADC* class:

```
from machine import ADC, Pin
adc = ADC(Pin(26))      # create ADC object on ADC pin
adc.read_u16()          # read value, 0-65535 across voltage range 0.0v - 3.3v
```

The argument of the constructor ADC specifies either a Pin by number, name of as Pin object, or a channel number in the range 0 - 3 or ADC.CORE_TEMP for the internal temperature sensor. If a pin is specified, the pin is initialized in high-Z mode. If a channel number is used, the pin is not initialized and configuring is left to the user code. After hard reset, RP2040 pins operate in current sink mode at about 60µA. If the pin is not otherwise configured, that may lead to wrong ADC readings.

9.13 Software SPI bus

Software SPI (using bit-banging) works on all pins, and is accessed via the *machine.SoftSPI* class:

```
from machine import Pin, SoftSPI

# construct a SoftSPI bus on the given pins
# polarity is the idle state of SCK
# phase=0 means sample on the first edge of SCK, phase=1 means the second
spi = SoftSPI(baudrate=100_000, polarity=1, phase=0, sck=Pin(0), mosi=Pin(2),␣
↪miso=Pin(4))

spi.init(baudrate=200000) # set the baudrate

spi.read(10)            # read 10 bytes on MISO
spi.read(10, 0xff)      # read 10 bytes while outputting 0xff on MOSI

buf = bytearray(50)     # create a buffer
spi.readinto(buf)       # read into the given buffer (reads 50 bytes in this case)
spi.readinto(buf, 0xff) # read into the given buffer and output 0xff on MOSI

spi.write(b'12345')     # write 5 bytes on MOSI

buf = bytearray(4)      # create a buffer
spi.write_readinto(b'1234', buf) # write to MOSI and read from MISO into the buffer
spi.write_readinto(buf, buf) # write buf to MOSI and read MISO back into buf
```

Warning: Currently *all* of `sck`, `mosi` and `miso` *must* be specified when initialising Software SPI.

9.14 Hardware SPI bus

The RP2040 has 2 hardware SPI buses which is accessed via the *machine.SPI* class and has the same methods as software SPI above:

```
from machine import Pin, SPI

spi = SPI(1, 10_000_000)  # Default assignment: sck=Pin(10), mosi=Pin(11), miso=Pin(8)
spi = SPI(1, 10_000_000, sck=Pin(14), mosi=Pin(15), miso=Pin(12))
spi = SPI(0, baudrate=80_000_000, polarity=0, phase=0, bits=8, sck=Pin(6),␣
↪mosi=Pin(7), miso=Pin(4))
```

9.15 Software I2C bus

Software I2C (using bit-banging) works on all output-capable pins, and is accessed via the *machine.SoftI2C* class:

```
from machine import Pin, SoftI2C

i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100_000)

i2c.scan()              # scan for devices
```

(continues on next page)

(continued from previous page)

```
i2c.readfrom(0x3a, 4)    # read 4 bytes from device with address 0x3a
i2c.writeto(0x3a, '12') # write '12' to device with address 0x3a

buf = bytearray(10)      # create a buffer with 10 bytes
i2c.writeto(0x3a, buf)  # write the given buffer to the peripheral
```

9.16 Hardware I2C bus

The driver is accessed via the *machine.I2C* class and has the same methods as software I2C above:

```
from machine import Pin, I2C

i2c = I2C(0)   # default assignment for Pico: scl=Pin(5), sda=Pin(4)
i2c = I2C(1, scl=Pin(3), sda=Pin(2), freq=400_000)
```

9.17 I2S bus

See *machine.I2S*.

```
from machine import I2S, Pin

i2s = I2S(0, sck=Pin(16), ws=Pin(17), sd=Pin(18), mode=I2S.TX, bits=16, format=I2S.
↪STEREO, rate=44100, ibuf=40000) # create I2S object
i2s.write(buf)              # write buffer of audio samples to I2S device

i2s = I2S(1, sck=Pin(0), ws=Pin(1), sd=Pin(2), mode=I2S.RX, bits=16, format=I2S.MONO,␣
↪rate=22050, ibuf=40000) # create I2S object
i2s.readinto(buf)           # fill buffer with audio samples from I2S device
```

The `ws` pin number must be one greater than the `sck` pin number.

The I2S class is currently available as a Technical Preview. During the preview period, feedback from users is encouraged. Based on this feedback, the I2S class API and implementation may be changed.

Two I2S buses are supported with id=0 and id=1.

9.18 Real time clock (RTC)

See *machine.RTC*

```
from machine import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 0, 1, 12, 48, 0)) # set a specific date and
                                             # time, eg. 2017/8/23 1:12:48
                                             # the day-of-week value is ignored
rtc.datetime() # get date and time
```

9.19 WDT (Watchdog timer)

The RP2040 has a watchdog which is a countdown timer that can restart parts of the chip if it reaches zero.

See *machine.WDT*.

```
from machine import WDT

# enable the WDT with a timeout of 5s (1s is the minimum)
wdt = WDT(timeout=5000)
wdt.feed()
```

The maximum value for timeout is 8388 ms.

9.20 OneWire driver

The OneWire driver is implemented in software and works on all pins:

```
from machine import Pin
import onewire

ow = onewire.OneWire(Pin(12)) # create a OneWire bus on GPIO12
ow.scan()               # return a list of devices on the bus
ow.reset()              # reset the bus
ow.readbyte()           # read a byte
ow.writebyte(0x12)      # write a byte on the bus
ow.write('123')         # write bytes on the bus
ow.select_rom(b'12345678') # select a specific device by its ROM code
```

There is a specific driver for DS18S20 and DS18B20 devices:

```
import time, ds18x20
ds = ds18x20.DS18X20(ow)
roms = ds.scan()
ds.convert_temp()
time.sleep_ms(750)
for rom in roms:
    print(ds.read_temp(rom))
```

Be sure to put a 4.7k pull-up resistor on the data line. Note that the `convert_temp()` method must be called each time you want to sample the temperature.

9.21 NeoPixel and APA106 driver

Use the `neopixel` and `apa106` modules:

```
from machine import Pin
from neopixel import NeoPixel

pin = Pin(0, Pin.OUT)   # set GPIO0 to output to drive NeoPixels
np = NeoPixel(pin, 8)   # create NeoPixel driver on GPIO0 for 8 pixels
np[0] = (255, 255, 255) # set the first pixel to white
```

(continues on next page)

(continued from previous page)

```
np.write()              # write data to all pixels
r, g, b = np[0]         # get first pixel colour
```

The APA106 driver extends NeoPixel, but internally uses a different colour order:

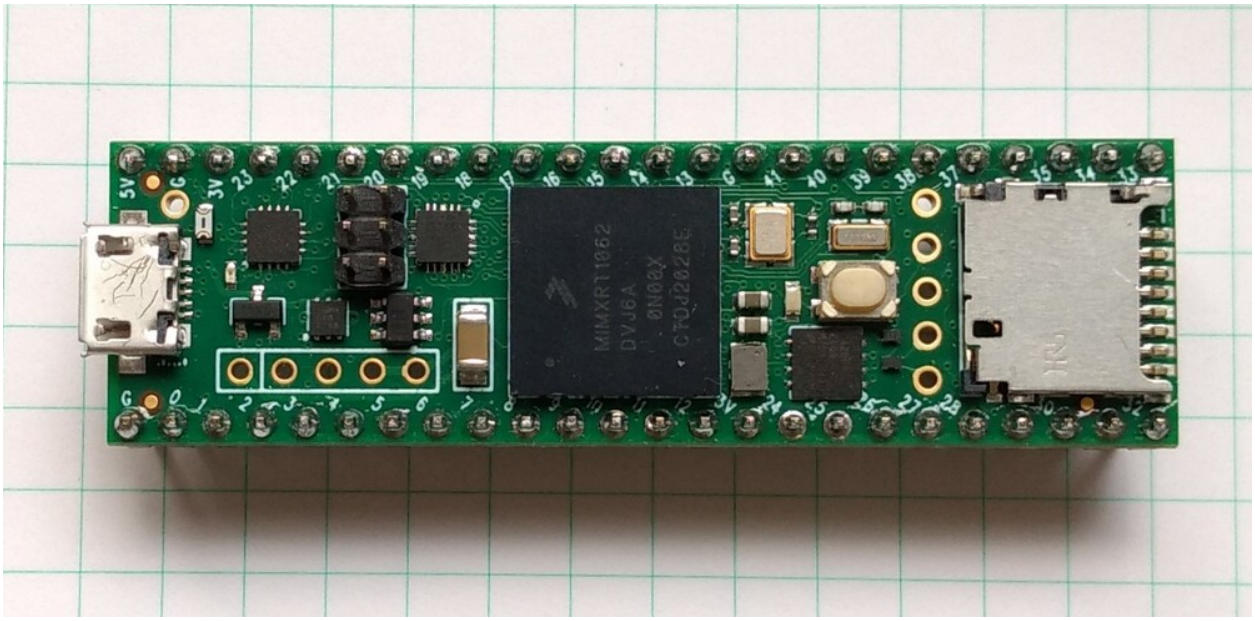
```
from apa106 import APA106
ap = APA106(pin, 8)
r, g, b = ap[0]
```

APA102 (DotStar) uses a different driver as it has an additional clock pin.

CHAPTER

TEN

QUICK REFERENCE FOR THE I.MXRT FAMILY



The Teensy 4.1 board.

Below is a quick reference for i.MXRT-based boards. If it is your first time working with this board it may be useful to get an overview of the microcontroller:

10.1 General information about the MIMXRT port

The i.MXRT MCU family is a high performance family of devices made by NXP. Based on an ARM7 core, they provide many on-chip I/O units for building small to medium sized devices.

10.1.1 Multitude of boards

There is a multitude of modules and boards from different sources which carry an i.MXRT chip. MicroPython aims to provide a generic port which runs on as many boards/modules as possible, but there may be limitations. The NXP IMXRT1020-EVK and the Teensy 4.0 and Teensy 4.1 development boards are taken as reference for the port (for example, testing is performed on them). For any board you are using please make sure you have a data sheet, schematics and other reference materials so you can look up any board-specific functions.

The following boards are supported by the port:

- MIMXRT1010-EVK
- MIMXRT1020-EVK
- MIMXRT1050-EVK
- MIMXRT1060-EVK
- MIMXRT1064-EVK
- Teensy 4.0
- Teensy 4.1

10.1.2 Supported MCUs

Product	CPU	Memory
i.MX RT1064	Cortex-M7 @600 MHz	1 MB SRAM, 4 MB Flash
i.MX RT1061	Cortex-M7 @600 MHz	1 MB SRAM
i.MX RT1062	Cortex-M7 @600 MHz	1 MB SRAM
i.MX RT1050	Cortex-M7 @600 MHz	512 kB SRAM
i.MX RT1020	Cortex-M7 @500 MHz	256 kB SRAM
i.MX RT1015	Cortex-M7 @500 MHz	128 kB SRAM
i.MX RT1010	Cortex-M7 @500 MHz	128 kB SRAM

Note: Most of the controllers do not have internal flash memory. Therefore their flash capacity is dependent on an external flash chip.

To make a generic MIMXRT port and support as many boards as possible the following design and implementation decision were made:

- GPIO pin numbering is based on the board numbering as well as on the MCU numbering. Please have the manual/pin diagram of your board at hand to find correspondence between your board pins and actual i.MXRT pins.
- All MCU pins are supported by MicroPython but not all are usable on any given board.

10.1.3 Technical specifications and SoC datasheets

The data sheets and other reference material for i.MXRT chip are available from the vendor site: https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus:IMX-RT-SERIES . They are the primary reference for the chip technical specifications, capabilities, operating modes, internal functioning, etc.

For your convenience, a few technical specifications are provided below:

- Architecture: ARM Cortex M7
- CPU frequency: up to 600MHz
- Total RAM available: up to 1 MByte (see table)
- BootROM: 96KB
- External FlashROM: code and data, via SPI Flash; usual size 2 - 16 MB Some boards provide additional external RAM and SPI flash.
- GPIO: up to 124 (GPIOs are multiplexed with other functions, including external FlashROM, UART, etc.)
- UART: 4 or 8 RX/TX UART. Hardware handshaking is supported by the MCU, but the boards used for testing do not expose the signals.
- SPI: 2 or 4 low power SPI interfaces (software implementation available on every pin)
- I2C: 2 or 4 low power I2C interfaces (software implementation available on every pin)
- I2S: 1 to 3 I2S interfaces
- ADC: one or two 12-bit SAR ADC converters
- Ethernet controller (except i.MX RT1010/-1015)
- Programming: using BootROM bootloader from USB - due to external FlashROM and always-available BootROM bootloader, the MIMXRT is not brickable

The lower numbers for UART, SPI and I2C apply to the i.MXRT 101x MCU.

For more information see the i.MXRT data sheets or reference manuals. NXP provides software support through it's SDK packages.

10.2 Getting started with MicroPython on the i.MXRT

Using MicroPython is a great way to get the most of your i.MXRT board. And vice versa, the i.MXRT chip is a great platform for using MicroPython. This tutorial will guide you through setting up MicroPython, getting a prompt, using the hardware peripherals, and controlling some external components.

Let's get started!

10.2.1 Requirements

The first thing you need is a board with an i.MXRT chip. The MicroPython software supports the i.MXRT chip itself and any board should work. The main characteristic of a board is how the GPIO pins are connected to the outside world, and whether it includes a built-in USB-serial converter to make the UART available to your PC.

Names of pins will be given in this tutorial using the chip names (eg GPIO2) and it should be straightforward to find which pin this corresponds to on your particular board.

10.2.2 Powering the board

If your board has a USB connector on it then most likely it is powered through this when connected to your PC. Otherwise you will need to power it directly. Please refer to the documentation for your board for further details.

10.2.3 Getting the firmware

Firmware versions are provided at the MicroPython download page. You can download the most recent MicroPython firmware .hex or .bin file to load onto your i.MXRT device. From that download page you have two main choices:

- stable firmware builds
- daily firmware builds

If you are just starting with MicroPython, the best bet is to go for the stable firmware builds. If you are an advanced, experienced MicroPython i.MXRT user who would like to follow development closely and help with testing new features, there are daily builds.

10.2.4 Deploying the firmware

Once you have the MicroPython firmware you need to load it onto your i.MXRT device. The exact procedure for these steps is highly dependent on the particular board and you will need to refer to its documentation for details.

Teensy 4.0 and 4.1

For Teensy 4.0 and 4.1 you have to use the built-in loader together with the PC loader provided by PJRC. The built-in loader will be activated by pushing the button on the board. Then you can upload the firmware with the command:

```
teensy_loader_cli --mcu=imxrt1062 -v -w firmware.hex
```

IMXRT10xx-EVK boards

The IMXRT10xx-EVK boards have a second USB port connected to a support MCU. Connecting that USB port to your PC will register a disk drive with the name of the board. Just copy the firmware.bin file to this drive, and that will start the flashing procedure. You will know that the flash was complete, if that drive disappears and reappears. If you decided to install the very useful Segger open-SDA firmware on that sidekick MCU, then you have to use the debugger software to upload the i.MXRT firmware.

Seed ARCH MIX board

Firmware upload to the Seed ARCH MIX board is less convenient. The vendor suggests using J-Link as a method and tool. For that, follow the instructions given by Seed in their Wiki at https://wiki.seeedstudio.com/Arch_Mix/#flashing-arduino-bootloader-to-arch-mix. You will need a J-Link debug probe and software. You may find Segger JLink edu or Segger JLink edu mini convenient. As a matching loader you can use JFlashLite. The target address for loading is 0x60000000.

The MIMXRT family also support a serial upload method. The software for serial upload is provided by NXP. The steps to use it are:

- Connect J3, Pin 19 to 3.3V (GPIO_AD_B0_05).
- Change the DIP-Switch settings from off-off-on-off to off-off-off-on
- Push Reset
- Run the upload with: ./FLASH.sh <firmware_image_file name>
- Once the upload has finished, set the DIP-switch back to off-off-on-off.
- Remove the Jumper to J3, Pin19 and push reset

To avoid running the Flash loader as superuser, you can copy the provided udev-rules script to /etc/udev/rules.d/. FLASH.sh calls two binaries, blhost and sdphost, which are provided by NXP under the BSD-3-Clause License. A version of these binaries and the script can be downloaded at https://github.com/robert-hh/Shared-Stuff/blob/master/mimxrt_serial_downloader.zip.

Serial downloading can be used for the NXP MIMXRT boards as well. But the built-in loader is much more convenient to use.

10.2.5 Serial prompt

Once you have the firmware on the device you can access the REPL (Python prompt) over USB.

From there you can follow the i.MXRT tutorial.

10.2.6 Troubleshooting installation problems

If you experience problems during flashing or with running firmware immediately after it, here are some troubleshooting recommendations:

- Be aware of and try to exclude hardware problems. There are two common problems: bad power source quality, and worn-out/defective Flash ROM. Speaking of power source, not just raw amperage is important, but also low ripple and noise/EMI in general. The most reliable and convenient power source is a USB port.

10.3 Pinout for the i.MXRT machine modules

The Teensy 4.0 and 4.1 board show pin numbers **nn** at the board's silkscreen. These are denoted in the tables below as **Dnn**. E.g. a silkscreen number **1** is shown as **D1**. Whenever a Pin has to be specified in a script, **Dnn** must be used, not **nn**.

10.3.1 UART pin assignment

The pin assignment of UARTs to pins is fixed. The UARTs are numbered 0..8. The rx/tx pins are assigned according to the tables below:

Board / Pin	UART0	UART1	UART2	UART3
Teensy 4.0	•	D0/D1	D7/D8	D14/D15
Teensy 4.1	•	D0/D1	D7/D8	D14/D15
MIMXRT1010-EVK	Debug USB	D0/D1	D7/D6	•
MIMXRT1015-EVK	Debug USB	D0/D1	D7/A1	•
MIMXRT1020-EVK	Debug USB	D0/D1	D9/D6	D10/D13
MIMXRT1050-EVK	Debug USB	D0/D1	D7/D6	D8/D9
MIMXRT1050-EVKB	Debug USB	D0/D1	D7/D6	D8/D9
MIMXRT1060-EVK	Debug USB	D0/D1	D7/D6	D8/D9
MIMXRT1064-EVK	Debug USB	D0/D1	D7/D6	D8/D9
MIMXRT1170-EVK	Debug USB	D0/D1	D12/D11	D10/D13
Adafruit Metro M7	•	D0/D1	D7/D3	A1/A0
Olimex RT1010Py	•	RxD/TxD	D7/D8	D5/D6
Seeed ARCH MIX	•	J3_19/J3_20	J4_16/J4_17	J4_06/J4_07
Makerdiary RT1011	•	D9/D10	D13/A0	D11/D12

Board / Pin	UART4	UART5	UART6	UART7	UART8
Teensy 4.0	D16/D17	D21/D20	D25/D24	D28/D29	•
Teensy 4.1	D16/D17	D21/D20	D25/D24	D28/D29	D34/D35
MIMXRT1010-EVK	•	•	•	•	•
MIMXRT1015-EVK	•	•	•	•	•
MIMXRT1020-EVK	D15/D14	A1/A0	•	•	•
MIMXRT1050-EVK	A1/A0	•	•	•	•
MIMXRT1050-EVKB	A1/A0	•	•	•	•
MIMXRT1060-EVK	A1/A0	•	•	•	•
MIMXRT1064-EVK	A1/A0	•	•	•	•
MIMXRT1170-EVK	D15/D14	D25/D26	D33/D34	D35/D36	•
Olimex RT1010Py	•	•	•	•	•
Seeed ARCH MIX	J4_10/J4_11	J5_08/J5_12	•	•	•
Makerdiary RT1011	A1/A2	•	•	•	•

10.3.2 PWM pin assignment

Pins are specified in the same way as for the Pin class. The following tables show the assignment of the board Pins to PWM modules:

Pin/ MIMXRT	1010	1015	1020	1050/60/64	1170	Metro M7
D0	•	Q1/1	F1/1/B	•	•	•
D1	•	Q1/0	F1/1/A	•	•	•
D2	F1/3/B	F1/3/A	•	F1/3/B	•	•
D3	F1/3/A	F1/0/A	F2/3/B	F4/0/A	F1/2/A (&)	•
D4	F1/3/A (*)	Q1/2	Q2/1	F2/3/A	F1/0/X (&)	F1/0/B
D5	F1/0/B (*)	F1/0/B	F2/3/A	F1/3/A	F1/2/B (&)	F1/0/A
D6	•	F1/2/B	F2/0/A	Q3/2	F1/0/A	•
D7	•	•	F1/0/A	Q3/3	F3/0/X	•
D8	F1/0/A	F1/1/B	F1/0/B	F1/1/X	F1/1/X (&)	F1/3/A
D9	F1/1/B (*)	F1/2/A	F2/0/B	F1/0/X	F1/0/B	F1/3/B
D10	F1/3/B	•	F2/2/B	F1/0/B (*)	F2/2/B	F1/2/A
D11	F1/2/A	•	F2/1/A	F1/1/A (*)	•	F1/2/B
D12	F1/2/B	•	F2/1/B	F1/1/B (*)	•	F1/1/A
D13	F1/3/A	•	F2/2/A	F1/0/A (*)	F2/2/A	F1/1/B
D14	F1/0/B	•	•	F2/3/B	•	F1/0/B
D15	F1/0/A	•	•	F2/3/A	•	F1/0/A
A0	•	•	F1/2/A	•	F2/0/X	•
A1	F1/3/X	F1/3/B	F1/2/B	•	F2/1/X	•
A2	F1/2/X	F1/3/A	F1/3/A	•	F2/2/X	•
A3	•	F1/2/A	F1/3/B	•	F2/3/X	F1/3/B
A4	•	•	•	Q3/1	F1/3/X	F1/2/X
A5	•	•	•	Q3/0	F1/2/X	•
D31	•	•	•	•	F1/2/B (&)	•
D32	•	•	•	•	F1/2/A (&)	•
D33	•	•	•	•	F1/1/B	•
D34	•	•	•	•	F1/1/A	•

Pins denoted with (*) are by default not wired at the board.

Pins denoted with (&) have alternative PWM options available.

Pin	Teensy 4.0	Pin	Teensy 4.1
D0	F1/1/X	D0	F1/1/X
D1	F1/0/X	D1	F1/0/X
D2	F4/2/A	D2	F4/2/A
D3	F4/2/B	D3	F4/2/B
D4	F2/0/A	D4	F2/0/A
D5	F2/1/A	D5	F2/1/A
D6	F2/2/A	D6	F2/2/A
D7	F1/3/B	D7	F1/3/B
D8	F1/3/A	D8	F1/3/A
D9	F2/2/B	D9	F2/2/B
D10	Q1/0	D10	Q1/0
D11	Q1/2	D11	Q1/2
D12	Q1/1	D12	Q1/1
D13	Q2/0	D13	Q2/0
D14	Q3/2	D14	Q3/2
D15	Q3/3	D15	Q3/3
D18	Q3/1	D18	Q3/1
D19	Q3/0	D19	Q3/0
D22	F4/0/A	D22	F4/0/A
D23	F4/1/A	D23	F4/1/A
D24	F1/2/X	D24	F1/2/X
D25	F1/3/X	D25	F1/3/X
D28	F3/1/B	D28	F3/1/B
D29	F3/1/A	D29	F3/1/A
D33	F2/0/B	D33	F2/0/B
•	•	D36	F2/3/A
•	•	D37	F2/3/B
DAT1	F1/1/B	D42	F1/1/B
DAT0	F1/1/A	D43	F1/1/A
CLK	F1/0/B	D44	F1/0/B
CMD	F1/0/A	D45	F1/0/A
DAT2	F1/2/A	D46	F1/2/A
DAT3	F1/2/B	D47	F1/2/B
•	•	D48	F1/0/B
•	•	D49	F1/2/A
•	•	D50	F1/2/B
•	•	D51	F3/3/B
•	•	D52	F1/1/B

continues on next page

Table 1 – continued from previous page

Pin	Teensy 4.0	Pin	Teensy 4.1
•	•	D53	F1/1/A
•	•	D54	F3/0/A

Pin	Seeed ARCH MIX
J3_04	Q4/3
J3_10	Q1/3
J3_12	Q2/3
J3_13	Q3/3
J3_16	Q3/0
J3_17	Q3/1
J3_19	F1/3/X
J3_20	F1/2/X
J4_08	F4/0/A
J4_09	F4/1/A
J4_16	Q3/2
J4_17	Q3/3
J5_32	Q1/0
J5_28	Q1/1
J5_29	Q1/2
J5_30	Q2/0
J5_04	Q2/1
J5_05	Q2/3
J5_06	F2/0/A
J5_07	F2/0/B
J5_08	F2/1/A
J5_12	F2/1/B
J5_13	F2/2/A
J5_14	F2/2/B
J5_23	F1/3/A
J5_24	F1/3/B
J5_25	F2/3/A
J5_26	F2/3/B
J5_42	Q3/0
J5_43	Q3/1
J5_50	F1/0/X
LED_RED	F2/3/A
LED_GREEN	F1/3/A
LED_BLUE	F1/3/B

Pin	Olimex RT1010PY
D1	F1/0/B
D2	F1/0/A
D3	F1/1/B
D4	F1/1/A
D5	F1/2/B
D6	F1/2/A
D7	F1/3/B
D8	F1/3/A
D10	F1/0/B
D11	F1/0/A
D12	F1/1/B
D13	F1/1/A
A1	F1/2/B
A2	F1/2/A
A3	F1/3/B
A4	F1/3/A
SDI	F1/3/X
SDO	F1/2/X
CS0	F1/1/X
SCK	F1/0/X

Pin	Makerdiary RT1011
D1	F1/0/B
D2	F1/0/A
D3	F1/1/B
D4	F1/1/A
D5	F1/2/B
D6	F1/2/A
D7	F1/3/B
D8	F1/3/A
A3	F1/2/B
A4	F1/2/A
A5	F1/3/B
A6	F1/3/A
A9	F1/3/X
A10	F1/2/X
A11	F1/1/X
SD1	F1/0/B
SD2	F1/0/A
LED	F1/1/B
DIO	F1/0/X

Legend:

- Qm/n: QTMR module m, channel n

- Fm/n/l: FLEXPWM module m, submodule n, channel l. The pulse at a X channel is always aligned to the period start.

Pins without a PWM signal are not listed. A signal may be available at more than one Pin. FlexPWM pins may also be pure CPU pin, not assigned to a board signal. In that case the PWM output is disabled. The PWM channel of a submodule 0 may still be used as synchronization source for other channels of the same module, unless used by another peripheral.

Submodule 0 pins for i.MX RT1011:

Pin	Channel
Pin.cpu.GPIO_01	B
Pin.cpu.GPIO_02	A
Pin.cpu.GPIO_AD_12	X
Pin.cpu.GPIO_SD_01	B
Pin.cpu.GPIO_SD_02	A

Submodule 0 pins for i.MX RT1021:

Pin	Module & Channel
Pin.cpu.GPIO_AD_B1_06	FLEXPWM1 Channel A
Pin.cpu.GPIO_AD_B1_07	FLEXPWM1 Channel B
Pin.cpu.GPIO_EMC_26	FLEXPWM1 Channel A
Pin.cpu.GPIO_EMC_27	FLEXPWM1 Channel B
Pin.cpu.GPIO_AD_B0_14	FLEXPWM2 Channel A
Pin.cpu.GPIO_AD_B0_15	FLEXPWM2 Channel B
Pin.cpu.GPIO_EMC_10	FLEXPWM2 Channel X
Pin.cpu.GPIO_EMC_38	FLEXPWM2 Channel A
Pin.cpu.GPIO_EMC_39	FLEXPWM2 Channel B

Submodule 0 pins for i.MX RT1052, i.MX RT1062 and i.MX RT1064:

Pin	Module & Channel
Pin.cpu.GPIO_AD_B0_02	FLEXPWM1 Channel X
Pin.cpu.GPIO_EMC_23	FLEXPWM1 Channel A
Pin.cpu.GPIO_EMC_24	FLEXPWM1 Channel B
Pin.cpu.GPIO_SD_B0_00	FLEXPWM1 Channel A
Pin.cpu.GPIO_SD_B0_01	FLEXPWM1 Channel B
Pin.cpu.GPIO_B0_06	FLEXPWM2 Channel A
Pin.cpu.GPIO_B0_07	FLEXPWM2 Channel B
Pin.cpu.GPIO_EMC_06	FLEXPWM2 Channel A
Pin.cpu.GPIO_EMC_07	FLEXPWM2 Channel B
Pin.cpu.GPIO_EMC_29	FLEXPWM3 Channel A
Pin.cpu.GPIO_EMC_30	FLEXPWM3 Channel B
Pin.cpu.GPIO_AD_B1_08	FLEXPWM4 Channel A
Pin.cpu.GPIO_EMC_00	FLEXPWM4 Channel A
Pin.cpu.GPIO_EMC_01	FLEXPWM4 Channel B

Submodule 0 pins for i.MX RT1176

Pin	Module & Channel
Pin.cpu.GPIO_EMC_B1_00	FLEXPWM4 Channel A (*)
Pin.cpu.GPIO_EMC_B1_01	FLEXPWM4 Channel B (*)
Pin.cpu.GPIO_EMC_B1_06	FLEXPWM2 Channel A (*)
Pin.cpu.GPIO_EMC_B1_07	FLEXPWM2 Channel B (*)
Pin.cpu.GPIO_EMC_B1_23	FLEXPWM1 Channel A (*)
Pin.cpu.GPIO_EMC_B1_24	FLEXPWM1 Channel B (*)
Pin.cpu.GPIO_EMC_B1_29	FLEXPWM3 Channel A (*)
Pin.cpu.GPIO_EMC_B1_30	FLEXPWM3 Channel B (*)
Pin.cpu.GPIO_AD_00	FLEXPWM1 Channel A
Pin.cpu.GPIO_AD_01	FLEXPWM1 Channel B
Pin.cpu.GPIO_AD_06	FLEXPWM1 Channel X
Pin.cpu.GPIO_AD_10	FLEXPWM2 Channel X
Pin.cpu.GPIO_AD_14	FLEXPWM3 Channel X
Pin.cpu.GPIO_AD_18	FLEXPWM4 Channel X
Pin.cpu.GPIO_AD_24	FLEXPWM2 Channel A
Pin.cpu.GPIO_AD_25	FLEXPWM2 Channel B

10.3.3 Hardware SPI pin assignment

The SPI signals have fixed assignments to GPIO pins. It depends on the board design, which SPI's signals are exposed to the user, as detailed in the table below. The signal order in the table is: CS0, CS1, MOSI, MISO, CLK.

Board / Pin	SPI0	SPI1	SPI2
Teensy 4.0	D10/-/D11/D12/D13	D0/-/D26/D1/D27	•
Teensy 4.1	D10/D37/D11/D12/D13	D0/-/D26/D1/D27	-/D29/D50/D54/D49
MIXMXRT1010-EVK	D10/D7/D11/D12/D13	•	•
MIXMXRT1015-EVK	D10/-/D11/D12/D13	•	•
MIXMXRT1020-EVK	D10/-/D11/D12/D13	A3/D0/A5/A4/A0	•
MIXMXRT1050-EVK	D10/-/D11/D12/D13 (*)	•	•
MIXMXRT1050-EVKB	D10/-/D11/D12/D13 (*)	•	•
MIXMXRT1060-EVK	D10/-/D11/D12/D13 (*)	•	•
MIXMXRT1064-EVK	D10/-/D11/D12/D13 (*)	•	•
MIXMXRT1170-EVK	D10/-/D11/D12/D13	D28/-/D25/D24/D26	-/-/D14/D15/D24
Adafruit Metro M7	-/-/MOSI/MISO/SCK	•	•
Olimex RT1010Py	•	CS0/-/SDO/SDI/SCK	SDCARD with CS1
Seeed ARCH MIX	J4_12/-/J4_14/J4_13/J4_15	J3_09/J3_05/J3_08_J3_11	
Makerdiary RT1011	A5/A2/A4/A3/A6	A11/A1/A10/A9/CLK	

Pins denoted with (*) are by default not wired at the board. The CS0 and CS1 signals are enabled with the keyword option cs=0 or cs=1 of the SPI object constructor.

10.3.4 Hardware I2C pin assignment

The I2C signals have fixed assignments to GPIO pins. It depends on the board design, which I2C's signals are exposed to the user, as detailed in the table below. The signal order in the table is: SDA, SCL.

Board / Pin	I2C 0	I2C 1	I2C 2	I2C 3	I2C 4
Teensy 4.0	D18/D19	D17/D16	D25/D24	•	•
Teensy 4.1	D18/D19	D17/D16	D25/D24	•	•
MIXMXRT1010-EVK	D14/D15	D0/D1	•	•	•
MIXMXRT1015-EVK	D14/D15	•	•	•	•
MIXMXRT1020-EVK	D14/D15	A4/A5	D0/D1	•	•
MIXMXRT1050-EVK	A4/A5	D1/D0	•	•	•
MIXMXRT1050-EVKB	A4/A5	D1/D0	•	•	•
MIXMXRT1060-EVK	A4/A5	D1/D0	•	•	•
MIXMXRT1064-EVK	A4/A5	D1/D0	•	•	•
MIXMXRT1170-EVK	D14/D15	D1/D0	A4/A5	D26/D25	D19/D18
Adafruit Metro M7	D14/D15	D0/D1			
Olimex RT1010Py	•	SDA1/SCL1	SDA2/SCL2	•	•
Seeed ARCH MIX	J3_17/J3_16	J4_06/J4_07	J5_05/J5_04	•	•
Makerdiary RT1011	D1/D2	A7/A8			

10.3.5 Hardware I2S pin assignment

Pin assignments for a few MIMXRT boards:

Board	ID	MCK	SCK_TX	WS_TX	SD_TX	SCK_RX	WS_RX	SD_RX
Teensy 4.0	1	D23	D26	D27	D7	D21	D20	D8
Teensy 4.0	2	D33	D4	D3	D2	•	•	D5
Teensy 4.1	1	D23	D26	D27	D7	D21	D20	D8
Teensy 4.1	2	D33	D4	D3	D2	•	•	D5
Seeed Arch MIX	1	J4_09	J4_14	J4_15	J14_13	J4_11	J4_10	J4_10
Adafruit Metro M7	1	D8	D10	D9	D12	D14	D15	D13
Olimex RT1010Py	1	D8	D6	D7	D4	D1	D2	D3
Olimex RT1010Py	3	•	D10	D9	D11	•	•	•
MIMXRT_	1	“MCK”	“SCK_TX”	“WS_TX”	“SD_TX”	“SCK_RX”	“WS_RX”	“SD_RX”
Mak-erdiary RT1011	1	D8	SD1	D7	D4	D1	D2	D3

Symbolic pin names are provided for the MIMXRT_10xx_DEV boards. These are provided for the other boards as well.

10.3.6 Pin Assignment

Pins are specified in the same way as for the Pin class. The pins available for an assignment to the Encoder or Counter are:

IMXRT1010_EVK, **iMX RT1011 Nano Kit**, **Olimex RT1010Py**:

Not supported.

IMXRT1015_EVK:

J30, pins 1 and 3, with the pin names “ENC1” and “ENC2”.

IMXRT1020_EVK:

Pins D0 and D1.

IMXRT1050_EVK, **IMXRT1050_EVKB**, **IMXRT1060_EVK**, **IMXRT1064_EBK**:

Pins D2, D4, D5, D8, D9, D10, D11, D12, D13, D14, D15, A4, A5. Depending on the board configuration, not all pins may be wired. Pins D2, D4 and D5 cannot be used for the match output.

IMXRT1170_EVK:

Pins D0, D1, D2.

D2 is connected to the 1G PHY chip as well. So levels may be distorted.

Teensy 4.0:

Pins D0, D1, D2, D3, D4, D5, D7, D8, D26, D27, D30, D31, D32, D33. Pin D0 and D5 share the same signal and cannot be used independently. Pins D26, D27, D30 and D31 cannot be used for the match output.

Teensy 4.1:

Pins D0, D1, D2, D3, D4, D5, D7, D8, D26, D27, D30, D31, D32, D33, D37, D42, D43, D44, D45, D46 and D47. Pins D26, D27, D30 and D31 cannot be used for the match output. Some pins are assigned to the same signal and cannot be used independently. These are:

- Pins D0, D5 and D37,
- Pins D2 and D43,
- Pins D3 and D42, and
- Pins D4 and D47.

Seeed ARCH MIX

Pins J3_14, J3_15, J4_19, J4_20, J5_15, J5_16, J5_17, J5_22, J5_23, J5_24, J5_25 and J5_26. Pins J3_14 and J3_15 cannot be used for the match output.

10.3.7 Hardware CAN pin assignment

Pin assignments for a few MIMXRT boards. The list show the MCU TX/RX pins.

Board	CAN1	CAN2	CAN3
Teensy 4.0	A8/A9	D1/D0	•
Teensy 4.1	A8/A9	D1/D0	D31/D30
Seeed Arch MIX	J4_08/J4_09	J3_14/J3_15	
MIMXRT1020_DEV	Transceiver	•	•
MIMXRT1050_DEV	Transceiver	•	•
MIMXRT1060_DEV	Transceiver	•	•
MIMXRT1064_DEV	Transceiver	•	•
MIMXRT1170_DEV	Transceiver	D4/D8	•
phyBOARD-RT1170	CAN port	•	•

All supported MIMXRT Developments boards are equipped with a CAN transceiver and do expose it's MCU pins. At the 3 pin transceiver connector CAN_H is as pin 1, CAN_L at Pin 3. Pin 2 is connected to GND. A documentation showing the CAN pin assignments of the phyBOARD-RT1170 does not seem to be accessible.

10.4 Installing MicroPython

See the corresponding section of tutorial: *Getting started with MicroPython on the i.MXRT*. It also includes a troubleshooting subsection.

10.5 General board control

The MicroPython REPL is on the USB port, configured in VCP mode. Tab-completion is useful to find out what methods an object has. Paste mode (ctrl-E) is useful to paste a large slab of Python code into the REPL.

The `machine` module:

```
import machine

machine.freq()          # get the current frequency of the CPU
```

10.6 Delay and timing

Use the `time` module:

```
import time

time.sleep(1)           # sleep for 1 second
time.sleep_ms(500)      # sleep for 500 milliseconds
time.sleep_us(10)       # sleep for 10 microseconds
start = time.ticks_ms() # get millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start) # compute time difference
```

10.7 Timers

The i.MXRT port supports virtual Timers. Example of usage:

```
from machine import Timer

tim0 = Timer(-1)
tim0.init(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(0))

tim1 = Timer(-1)
tim1.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(1))
```

The period is in milliseconds.

10.8 Pins and GPIO

Use the *machine.Pin* class:

```
from machine import Pin

p0 = Pin('D0', Pin.OUT) # create output pin on GPIO0
p0.on()                 # set pin to "on" (high) level
p0.off()                # set pin to "off" (low) level
p0.value(1)             # set pin to on/high

p2 = Pin('D2', Pin.IN)  # create input pin on GPIO2
print(p2.value())       # get value, 0 or 1

p4 = Pin('D4', Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor
p5 = Pin('D5', Pin.OUT, value=1) # set pin high on creation

p6 = Pin(pin.cpu.GPIO_B1_15, Pin.OUT) # Use the cpu pin name.
```

Available Pins follow the ranges and labelling of the respective board, like:

- 'D0-Dxx', or 'A0-Ann' for Teensy 4.0, MIMXRT10xx-EVK ns Olimex board,
- 'J3_xx', 'J4_xx', 'J5_xx' for the Seeed ARCH MIX board,

or the pin names of the Pin.board or Pin.cpu classes.

Notes:

- The MIMXRT1xxx-EVK boards may have other on-board devices connected to these pins, limiting it's use for input or output.
- At the MIMXRT1010_EVK, pins D4, D5 and D9 of the Arduino connector are by default not connected to the MCU. For details refer to the schematics.
- **At the MIMXRT1170_EVK board, the inner rows of the Arduino connectors are assigned as follows:**
  - 'D16' - 'D23': J9, odd pin numbers; 'D17' is by default not connected.
  - 'D24' - 'D27': J26, odd pin numbers; J63-J66 have to be closed to enable these pins.
  - 'D29' - 'D36': J25, odd pin numbers; 'D29' and 'D30' are by default not connected.

There's a higher-level abstraction *machine.Signal* which can be used to invert a pin. Useful for illuminating active-low LEDs using `on()` or `value(1)`.

10.9 UART (serial bus)

See *machine.UART*.

```
from machine import UART

uart1 = UART(1, baudrate=115200)
uart1.write('hello')  # write 5 bytes
uart1.read(5)         # read up to 5 bytes
uart1 = UART(baudrate=19200) # open UART 1 at 19200 baud
```

The i.MXRT has up to eight hardware UARTs, but not every board exposes all TX and RX pins for users. For the assignment of Pins to UART signals, refer to the *UART pinout*. If the UART ID is omitted, UART(1) is selected. Then, the keyword option for baudrate must be used to change it from the default value.

10.10 PWM (pulse width modulation)

The i.MXRT has up to four dedicated PWM modules with four FLEXPWM submodules each and up to four QTMR modules with four channels, which can be used to generate a PWM signal or signal pair.

The PWM functions are provided by the *machine.PWM* class. It supports all basic methods listed for that class and a few additional methods for handling signal groups.

```
# Samples for Teensy
#

from machine import Pin, PWM

# create PWM object from a pin and set the frequency and duty cycle
pwm2 = PWM(Pin('D2'), freq=2000, duty_u16=32768)
pwm2.freq()             # get the current frequency
pwm2.freq(1000)         # set/change the frequency
pwm2.duty_u16()         # get the current duty cycle, range 0-65535
pwm2.duty_u16(200)      # set the duty cycle, range 0-65535
pwm2.deinit()           # turn off PWM on the pin
# create a complementary signal pair on Pin 2 and 3
pwm2 = PWM(('D2', 'D3'), freq=2000, duty_ns=20000)

# Create a group of four synchronized signals.
# Start with Pin('D4') at submodule 0, which creates the sync pulse.
pwm4 = PWM(Pin('D4'), freq=1000, align=PWM.HEAD)
# Pins D5, D6, and D9 are pins at the same module
pwm5 = PWM(Pin('D5'), freq=1000, duty_u16=10000, align=PWM.HEAD, sync=True)
pwm6 = PWM(Pin('D6', freq=1000, duty_u16=20000, align=PWM.HEAD, sync=True)
pwm9 = PWM(Pin('D9'), freq=1000, duty_u16=30000, align=PWM.HEAD, sync=True)

pwm3                    # show the PWM objects properties
```

10.10.1 PWM Constructor

class PWM(*dest*, *freq*, *duty_u16*, *duty_ns*, ***, *center*, *align*, *invert*, *sync*, *xor*, *deadtime*)

Construct and return a new PWM object using the following parameters:

- *dest* is the entity on which the PWM is output, which is usually a *machine.Pin* object, but a port may allow other values, like integers or strings, which designate a Pin in the machine.PIN class. *dest* is either a single object or a two element object tuple. If the object tuple is specified, the two pins act in complementary mode. These two pins must be the A/B channels of the same submodule.

PWM objects are either provided by a FLEXPWM module or a QTMR module. The i.MXRT devices have either two or four FLEXPWM and QTMR modules. Each FLEXPWM module has four submodules with three channels, each, called A, B and X. Each QTMR module has four channels. Each FLEXPWM submodule or QTMR channel may be set to different parameters. Not every channel is routed to a board pin. Details are listed below.

Setting *freq* affects the three channels of the same FLEXPWM submodule. Only one of *duty_u16* and *duty_ns* should be specified at a time.

Keyword arguments:

- *freq* should be an integer which sets the frequency in Hz for the PWM cycle. The valid frequency range is 15 Hz resp. 18Hz resp. 24Hz up to > 1 MHz.
- *duty_u16* sets the duty cycle as a ratio `duty_u16 / 65535`. The duty cycle of a X channel can only be changed, if the A and B channel of the respective submodule is not used. Otherwise the duty_16 value of the X channel is 32768 (50%).
- *duty_ns* sets the pulse width in nanoseconds. The limitation for X channels apply as well.
- *center*=value. An integer sets the center of the pulse within the pulse period. The range is 0-65535. The resulting pulse will last from center - duty_u16/2 to center + duty_u16/2.
- *align*=value. Shortcuts for the pulse center setting, causing the pulse either at the center of the frame (value=0), the leading edge at the begin (value=1) or the trailing edge at the end of a pulse period (value=2).
- *invert*=value channel_mask. Setting a bit in the mask inverts the respective channel. Bit 0 inverts the first specified channel, bit 1 the second. The default is 0. For a PWM object with a single channel, True and False may be used as values.
- *sync*=True|False. If a channel of a module's submodule 0 is already active, other submodules of the same module can be forced to be synchronous to submodule 0. Their pulse period start then at at same clock cycle. The default is False.
- *xor*=0|1|2. If set to 1 or 2, the channel will output the XOR'd signal from channels A or B. If set to 1 on channel A or B, both A and B will show the same signal. If set to 2, A and B will show alternating signals. For details and an illustration, please refer to the MCU's reference manual, chapter "Double Switching PWMs".
- *deadtime*=time_ns. This setting affects complementary channels and defines a deadtime between an edge of a first channel and the edge of the next channel, in which both channels are set to low. That allows connected H-bridges to switch off one side of a push-pull driver before switching on the other side.

10.10.2 PWM Methods

The methods are identical to the generic *machine.PWM* class, with additional keyword arguments to the init() method, matchings those of the constructor.

Each FLEX submodule or QTMR module may run at different frequencies. The PWM signal is created by dividing the pwm_clk signal by an integral factor, according to the formula:

```
f = pwm_clk / (2**n * m)
```

with n being in the range of 0..7, and m in the range of 2..65535. pmw_clk is 125Mhz for MIMXRT1010/1015/1020, 150 MHz for MIMXRT1050/1060/1064 and 160MHz for MIMXRT1170. The lowest frequency is pwm_clk/2**23 (15, 18, 20Hz). The highest frequency with U16 resolution is pwm_clk/2**16 (1907, 2288, 2441 Hz), the highest frequency with 1 percent resolution is pwm_clk/100 (1.25, 1.5, 1.6 MHz). The highest achievable frequency is pwm_clk/3 for the A/B channels, and pwm_clk/2 for the X channels and QTMR signal.

10.10.3 PWM Pin Assignment

Pins are specified in the same way as for the Pin class. For the assignment of Pins to PWM signals, refer to the *PWM pinout*.

10.11 ADC (analog to digital conversion)

On the i.MXRT ADC functionality is available on Pins labeled 'Ann'.

Use the *machine.ADC* class:

```
from machine import ADC

adc = ADC(Pin('A2'))          # create ADC object on ADC pin
adc.read_u16()                # read value, 0-65535 across voltage range 0.0v - 3.3v
```

The resolution of the ADC is 12 bit with 10 to 11 bit accuracy, irrespective of the value returned by read_u16(). If you need a higher resolution or better accuracy, use an external ADC.

10.12 Software SPI bus

Software SPI (using bit-banging) works on all pins, and is accessed via the *machine.SoftSPI* class.

```
from machine import Pin, SoftSPI

# construct a SoftSPI bus on the given pins
# polarity is the idle state of SCK
# phase=0 means sample on the first edge of SCK, phase=1 means the second
spi = SoftSPI(baudrate=100000, polarity=1, phase=0, sck=Pin('D0'), mosi=Pin('D2'),␣
↪miso=Pin('D4'))

spi.init(baudrate=200000) # set the baudrate

spi.read(10)            # read 10 bytes on MISO
spi.read(10, 0xff)      # read 10 bytes while outputting 0xff on MOSI

buf = bytearray(50)     # create a buffer
spi.readinto(buf)       # read into the given buffer (reads 50 bytes in this case)
spi.readinto(buf, 0xff) # read into the given buffer and output 0xff on MOSI

spi.write(b'12345')     # write 5 bytes on MOSI

buf = bytearray(4)      # create a buffer
spi.write_readinto(b'1234', buf) # write to MOSI and read from MISO into the buffer
spi.write_readinto(buf, buf) # write buf to MOSI and read MISO back into buf
```

The highest supported baud rate is 500000.

10.13 Hardware SPI bus

There are up to four hardware SPI channels that allow faster transmission rates (up to 30Mhz). Hardware SPI is accessed via the *machine.SPI* class and has the same methods as software SPI above:

```
from machine import SPI, Pin

spi = SPI(0, 10000000)
cs_pin = Pin('D6', Pin.OUT, value=1)
cs_pin(0)
spi.write('Hello World')
cs_pin(1)
spi = SPI(baudrate=4_000_000)  # Use SPI(0) at a baudrate of 4 MHz
```

For the assignment of Pins to SPI signals, refer to *Hardware SPI pinout*. The keyword option cs=n can be used to enable the cs pin 0 or 1 for an automatic cs signal. The default is cs=-1. Using cs=-1 the automatic cs signal is not created. In that case, cs has to be set by the script. Clearing that assignment requires a power cycle. If the SPI ID is omitted, SPI(0) is selected. Then, the keyword option for baudrate must be used to change it from the default value.

Notes:

1. Even if the highest reliable baud rate at the moment is about 30 Mhz, setting a baud rate will not always result in exactly that frequency, especially at high baud rates.
2. Sending at higher baud rate is possible. In the tests receiving worked up to 60 MHz, sending up to 90 MHz.

10.14 Software I2C bus

Software I2C (using bit-banging) works on all output-capable pins, and is accessed via the *machine.SoftI2C* class:

```
from machine import Pin, SoftI2C

i2c = SoftI2C(scl=Pin('D5'), sda=Pin('D4'), freq=100000)

i2c.scan()              # scan for devices

i2c.readfrom(0x3a, 4)   # read 4 bytes from device with address 0x3a
i2c.writeto(0x3a, '12') # write '12' to device with address 0x3a

buf = bytearray(10)     # create a buffer with 10 bytes
i2c.writeto(0x3a, buf)  # write the given buffer to the slave
```

The highest supported freq is 400000.

10.15 Hardware I2C bus

There are up to four hardware I2C channels that allow faster transmission rates and support the full I2C protocol. The I2C signals have fixed assignments to GPIO pins. For the assignment of Pins to I2C signals, refer to *Hardware I2C pinout*.

Hardware I2C is accessed via the *machine.I2C* class and has the same methods as software SPI above:

```
from machine import I2C
```

(continues on next page)

(continued from previous page)

```
i2c = I2C(0, 400_000)
i2c.writeto(0x76, b"Hello World")
i2c = I2C(freq=100_000)  # use I2C(0) at 100kHz
```

If the I2C ID is omitted, I2C(0) is selected. Then, the keyword option for freq must be used to change the freq from the default value.

10.16 I2S bus

See *machine.I2S*. Example using a Teensy 4.1 board with a simple external Codec like UDA1334.:

```
from machine import I2S, Pin
i2s = I2S(2, sck=Pin('D26'), ws=Pin('D27'), sd=Pin('D7'),
    mode=I2S.TX, bts=16,format=I2S.STEREO,
    rate=44100,ibuf=40000)
i2s.write(buf)              # write buffer of audio samples to I2S device
```

Example for using I2S with a MIMXRT10xx_DEV board:

```
from machine import I2S, I2C, Pin
import wm8960

i2c=I2C(0)

wm=wm8960.WM8960(i2c, sample_rate=SAMPLE_RATE_IN_HZ,
    adc_sync=wm8960.sync_dac,
    swap=wm8960.swap_input)

i2s = I2S(1, sck=Pin("SCK_TX"), ws=Pin("WS_TX"), sd=Pin("SD_RX"),
    mck=Pin("MCK),mode=I2S.RX, bts=16,format=I2S.MONO,
    rate=32000,ibuf=10000)
i2s.readinto(buf)           # fill buffer with audio samples from I2S device
```

In this example, the input channels are swapped in the WM8960 driver, since the on-board microphone is connected to the right channel, but mono audio is taken from the left channel. Note, that the sck and ws pins are connected to the TX signals of the I2S bus. That is intentional, since at the MW8960 codec these signals are shared for RX and TX.

Example using the Teensy audio shield:

```
from machine import I2C, I2S, Pin
from sgtl5000 import CODEC
i2s = I2S(1, sck=Pin('D21'), ws=Pin('D20'), sd=Pin('D7'), mck=Pin('D23'),
    mode=I2S.TX, bits=16,rate=44100,format=I2S.STEREO,
    ibuf=40000,
)

# configure the SGTL5000 codec
i2c = I2C(0, freq=400000)
codec = CODEC(0x0A, i2c)
codec.mute_dac(False)
codec.dac_volume(0.9, 0.9)
codec.headphone_select(0)
codec.mute_headphone(False)
codec.volume(0.7, 0.7)
```

(continues on next page)

(continued from previous page)

```
i2s.write(buf)              # write buffer of audio samples to I2S device
```

The SGTL5000 codec used by the Teensy Audio shield uses the RX signals for both RX and TX. Note that the codec is initialized after the I2S device. That is essential since MCK is needed for its I2C operation and is provided by the I2S controller.

MIMXRT boards may have 1 or 2 I2S buses available at the board connectors. On MIMXRT1010 devices the bus numbers are 1 and 3. The I2S signals have fixed assignments to GPIO pins. For the assignment of Pins to I2S signals, refer to *I2S pinout*.

10.17 Real time clock (RTC)

See *machine.RTC*:

```
from machine import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 0, 1, 12, 48, 0)) # set a specific date and
                                             # time, eg. 2017/8/23 1:12:48
                                             # the day-of-week value is ignored
rtc.datetime() # get date and time
rtc.now() # return date and time in CPython format.
```

The i.MXRT MCU supports battery backup of the RTC. By connecting a battery of 1.5-3.6V, time and date are maintained in the absence of the main power. The current drawn from the battery is ~20µA, which is rather high. A CR2032 coin cell will last for about one year.

Note: In v1.23.0 the support for subseconds was removed. When reading the RTC, 0 will be returned as value for subsecond, When setting the RTC time, the subsecond field is ignored. The RTC itself does not provide a microsecond value.

10.18 SD card

See *machine.SDCard*:

```
import machine, os, vfs

sd = machine.SDCard()
fs = vfs.VfsFat(sd)
vfs.mount(fs, "/sd") # mount
os.listdir('/sd')    # list directory contents
vfs.umount('/sd')    # eject
```

Note: The i.mx-rt 1011 and 1015 based boards do not support the `machine.SDCard` class. For these, the SPI based driver `sdcard.py` from the MicroPython drivers can be used. When using it, you have to overdrive the CS pin of the SPI hardware module. Example:

```
import vfs, sdcard, machine

cs_pin = "D10"
spi = machine.SPI(0) # SPI0 with cs at Pin "D10" used for SDCARD
```

(continues on next page)

(continued from previous page)

```
cs = machine.Pin(cs_pin, machine.Pin.OUT, value=1)
sd = sdcard.SDCard(spi, cs)
fs = vfs.VfsFat(sd)
vfs.mount(fs, "/sdcard")
```

10.19 OneWire driver

The OneWire driver is implemented in software and works on all pins:

```
from machine import Pin
import onewire

ow = onewire.OneWire(Pin('D12')) # create a OneWire bus on GPIO12
ow.scan()                        # return a list of devices on the bus
ow.reset()                       # reset the bus
ow.readbyte()                    # read a byte
ow.writebyte(0x12)               # write a byte on the bus
ow.write('123')                  # write bytes on the bus
ow.select_rom(b'12345678')       # select a specific device by its ROM code
```

There is a specific driver for DS18S20 and DS18B20 devices:

```
import time, ds18x20
ds = ds18x20.DS18X20(ow)
roms = ds.scan()
ds.convert_temp()
time.sleep_ms(750)
for rom in roms:
    print(ds.read_temp(rom))
```

Be sure to put a 4.7k pull-up resistor on the data line. Note that the `convert_temp()` method must be called each time you want to sample the temperature.

10.20 DHT driver

The DHT driver is implemented in software and works on all pins:

```
import dht
import machine

d = dht.DHT11(machine.Pin('D4'))
d.measure()
d.temperature() # eg. 23 (°C)
d.humidity()    # eg. 41 (% RH)

d = dht.DHT22(machine.Pin('D4'))
d.measure()
d.temperature() # eg. 23.6 (°C)
d.humidity()    # eg. 41.3 (% RH)
```

Be sure to have a 4.7k pull-up resistor on the data line. Some DHT modules may already have one.

10.21 Ethernet driver

All MIMXRT boards except the MIMXRT1011 based boards and Teensy 4.0 support Ethernet. Example usage:

```
import network

lan = network.LAN(0)
lan.active(True)
```

If there is a DHCP server in the LAN, the IP address is supplied by that server. Otherwise, the IP address can be set with lan.ipconfig(addr4="…"). The default address is 192.168.0.1.

Teensy 4.1 does not have an Ethernet jack on the board, but PJRC offers an adapter for self-assembly. The Seeed ARCH MIX board has no PHY hardware on the board, however you can attach external PHY interfaces. By default, the firmware for Seeed Arch Mix uses the driver for a LAN8720 PHY. The MIMXRT1170_EVK is equipped with two Ethernet ports, which are addressed as LAN(0) for the 100M port and LAN(1) for the 1G port.

For details of the network interface refer to the class *network.LAN*.

10.22 class Counter – Signal counter for i.MXRT MCUs

This class provides a Counter service using the Quadrature Encoder module

Example usage:

```
# Samples for Teensy

from machine import Pin, Counter

counter = Counter(0, Pin("D0"))        # create Counter object
counter.value()                        # get current counter value
counter.value(0)                       # set the counter to 0
counter.init(max=128)                  # set the upper counting range
counter.deinit()                       # turn off the Counter
counter.init(match=1000)               # create a match event at count 1000
counter.irq(handler, Counter.IRQ_MATCH)  # call the function handler at a counter␣
↪match
counter                                # show the Counter object properties
```

The Counter is hardware based. It is available at all MIMXRT devices except the ones based on the i.MX RT 1010 MCU. For details about using the Counter with a MIMXRT board see *machine.Counter*:

10.23 class Encoder – Quadrature Encoder for i.MXRT MCUs

This class provides the Quadrature Encoder Service.

Example usage:

```
# Samples for Teensy

from machine import Pin, Encoder

qe = Encoder(0, Pin("D0"), Pin("D1")) # create Quadrature Encoder object
```

(continues on next page)

(continued from previous page)

```
qe.value()                          # get current counter values
qe.value(0)                         # set the counter value to 0
qe.init(max=128)                    # specify 128 counts as upper range
qe.init(index=Pin("D3"))            # specify Pin 3 as Index pulse input
qe.deinit()                         # turn off the Quadrature Encoder
qe.init(match=64)                   # set a match event at count 64
qe.irq(handler, qe.IRQ_MATCH)       # call the function handler at a match event
qe                                  # show the Encoder object properties
```

The Quadrature Encoder is hardware based. It is available at all MIMXRT devices except the ones based on the i.MX RT 1010 MCU. For details about using the Encoder with a MIMXRT board see *machine.Encoder*:

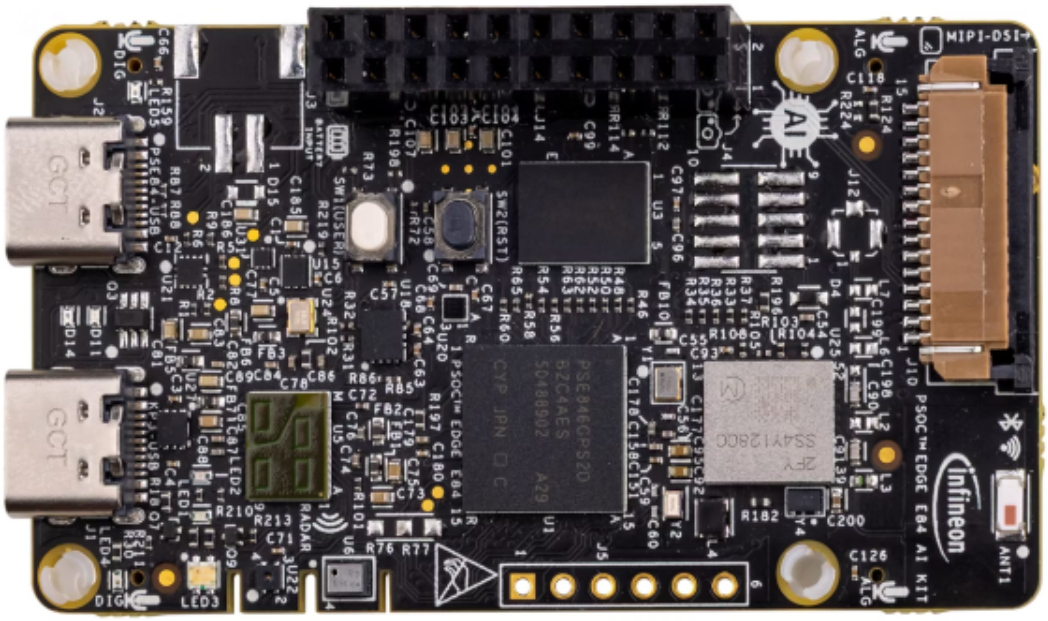
10.24 Transferring files

Files can be transferred to the i.MXRT devices for instance with the `mpremote` tool or using an SD card. If Ethernet is available, you can also use ftp. See the MicroPython forum for the FTP server or other community-supported alternatives to transfer files to an i.MXRT board, like rshell or Thonny.

CHAPTER

ELEVEN

QUICK REFERENCE FOR THE PSOC™ EDGE



The PSOC™ Edge E84 AI Kit.

Below is a quick reference for PSOC™ Edge E84 boards. If it is your first time working with this port it may be useful to get an overview of the microcontroller:

11.1 General information about the PSOC™ Edge port

The PSOC™ Edge is designed for next generation responsive compute and control applications, featuring hardware-assisted machine learning (ML) acceleration. The PSOC™ Edge devices are based on high performance Arm® Cortex®-M55, including Helium™ DSP support paired with Arm® Ethos™-U55 and Cortex®-M33 paired Infineon's ultra-low power NNLite neural network accelerator.

The PSOC™ Edge port supports boards powered by the Infineon PSOC™ Edge family of microcontrollers and currently supports the following MCUs:

- PSOC™ Edge E84

11.1.1 Supported boards

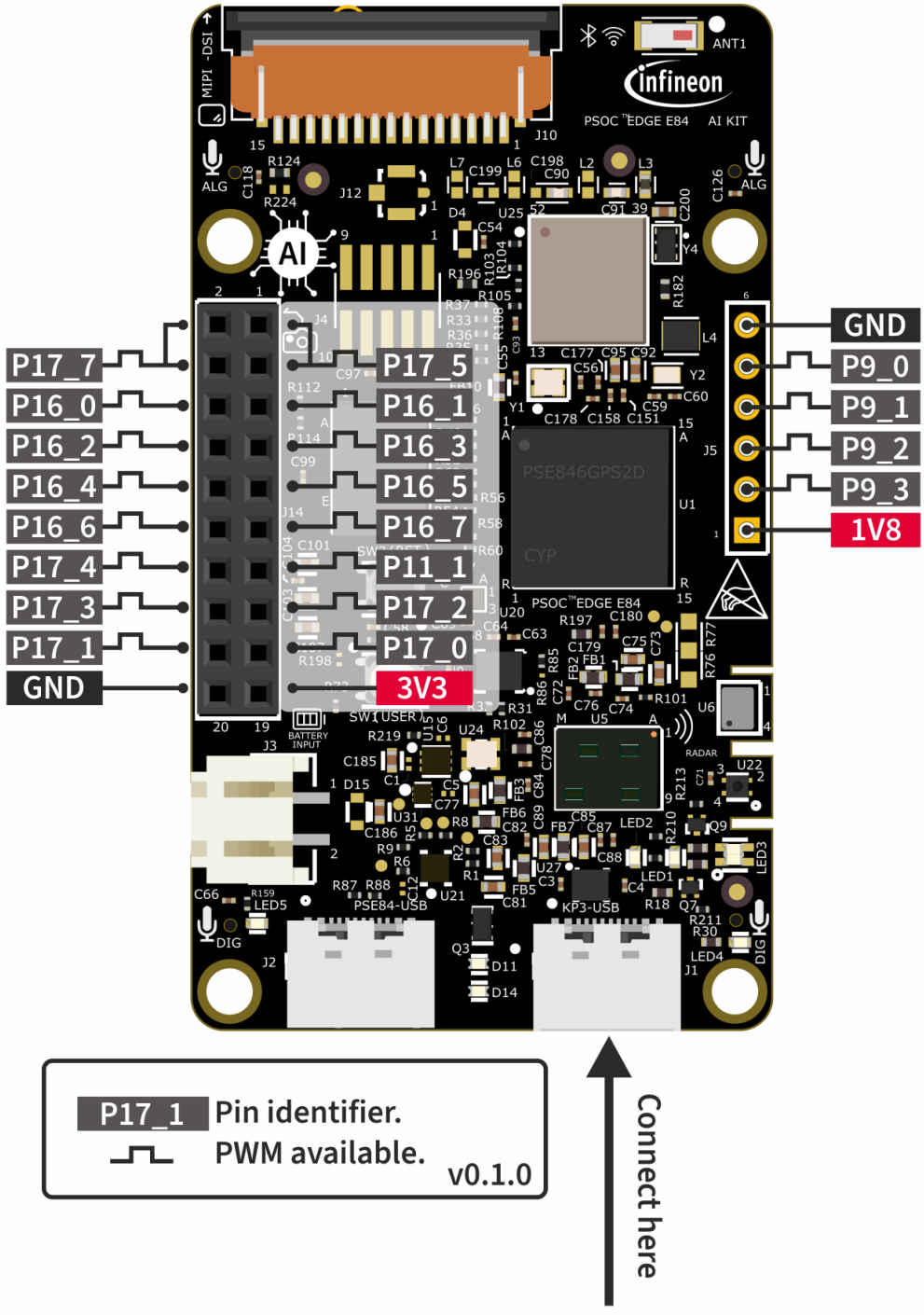
The following boards are supported on this port:

Board name	Connectivity	Special feature
KIT_PSE84_AI	Wi-Fi, BT	60GHz radar, analog microphone, digital microphone, barometric pressure sensor, external flash, speaker interface, IMU sensor, magnetometer, image sensor

For more hardware-related details, refer to the following sections.

KIT_PSE84_AI

Pinout


P17_7
P16_0
P16_2
P16_4
P16_6
P17_4
P17_3
P17_1
GND
P17_5
P16_1
P16_3
P16_5
P16_7
P11_1
P17_2
P17_0
3V3
GND
P9_0
P9_1
P9_2
P9_3
1V8
Infineon
PSOC™EDGE E84 AI KIT
PSE846GPS2D
PSOC™EDGE E84
P17_1 Pin identifier.
PWM available.
v0.1.0
Connect here

Technical specifications

Links for product details:

- KIT_PSE84_AI PSOC™ Edge E84 AI product page
- KIT_PSE84_AI PSOC™ Edge E84 AI Kit guide
- PSOC™ Edge E8x2, E8x3, E8x5, E8x6 Consumer Datasheet
- PSOC™ Edge E8x2, E8x3, E8x5, E8x6 Architecture Reference Manual

11.2 Installing MicroPython

To facilitate the installation of the MicroPython PSOC™ Edge port, the `mp-ifx-flash.py` Python script is provided. It is compatible with Windows, Linux, and macOS.

11.2.1 Prerequisites

Before downloading and running the script, it is recommended to create a new folder to keep all the related files together. For example:

```
$ mkdir mp-install
$ cd mp-install
```

You can easily download the script from the terminal using the following command:

```
$ curl -s -L https://raw.githubusercontent.com/micropython/micropython/master/ports/
↪psoc-edge/tools/mp-ifx-flash.py > mp-ifx-flash.py
```

Ensure you have a recent version of Python 3.x and the pip package installed. Then install the following packages:

```
$ pip install requests
```

Getting the firmware

Download the desired MicroPython firmware version for the PSOC™ Edge board from the MicroPython download page. The downloaded file is a `.zip` package containing the firmware binary and the necessary files for flashing.

11.2.2 Flashing

To flash the firmware, use the `from-package` command. Specify the board using the `--board` flag and the path to the downloaded `.zip` package using the `--zip-package` flag, as shown below:

```
$ python mp-ifx-flash.py from-package --board KIT_PSE84_AI --zip-package pathtodir/
↪psoc-edge-package.zip
```

Multiple connected devices

If you have multiple PSOC™ Edge boards connected to your computer, you can identify them by their serial number and flash each one individually. To do so, use the `-n` flag to provide the serial number of the target board as shown below:

```
$ python mp-ifx-flash.py from-package --board KIT_PSE84_AI --zip-package pathtodir/
↪psoc-edge-package.zip -n 181F0D5A01212300
```

Note: You can follow the latest progress for this port in the Infineon MicroPython PSOC™ Edge fork. We are working to bring those features here. Meanwhile, you can try them early by installing the fork version.

11.3 Pins and GPIO

See *machine.Pin* for the complete Pin API reference. This section focuses on the specific PSOC™ Edge port variations and particularities.

11.3.1 The constructor

The controller pin naming follows the nomenclature `P<port>_<pin>`, where:

- `<port>` is a numeric identifier for the port (e.g., 0-21 for the PSOC™ Edge E84)
- `<pin>` is the pin number within that port.

Use the respective board pinout diagram to find the available pins and their locations.

This is the `id` that needs to be passed to the constructor in one of the following formats:

- As a **string label**, single or double quoted: `'P<port>_<pin>'` or `"P<port>_<pin>"`
- A **pre-instantiated object** `Pin.cpu.<pin>` or `Pin.board.<pin>`.

```
from machine import Pin

p_in  = Pin('P0_0', Pin.IN)
p_out = Pin("P7_0", Pin.OUT, value=False)

p = Pin(Pin.cpu.P17_1, Pin.OPEN_DRAIN)
```

The pre-instantiated object can be used directly without calling the constructor. Instead, you can use `init()` to configure it.

```
from machine import Pin

pin = Pin.cpu.P17_0
pin.init(mode=Pin.IN)
```

Tip: Use the REPL interface to discover the available user pins, using tab for completion:

```
>>> from machine import Pin
>>> Pin.cpu.P
P10_5           P10_7           P11_3           P12_3
```

(continues on next page)

(continued from previous page)

```
P13_0           P13_1           P13_2           P13_3
P13_4           P13_5           P13_6           P13_7
P14_0           P14_1           P14_2           P14_3
P14_4           P14_5           P14_6           P14_7
P15_0           P15_1           P15_2           P15_3
P15_4           P15_5           P15_6           P15_7
P16_0           P16_1           P16_2           P16_3
P16_4           P16_5           P16_6           P16_7
P17_0           P17_1           P17_2           P17_3
P17_4           P17_5           P17_7           P20_3
P20_4           P20_5           P20_6           P20_7
P21_1           P21_2           P21_3           P21_4
P21_5           P21_6           P21_7           P3_0
P3_1            P6_4            P6_6            P7_0
P7_7            P8_0            P8_1            P8_5
P8_6            P9_0            P9_1            P9_2
P9_3
```

```
>>> from machine import Pin
>>> Pin.board.
AMIC1_CTB_INN   AMIC1_CTB_INP   AMIC1_CTB_OUT   AMIC1_CTB_REF
AMIC2_CTB_INN   AMIC2_CTB_INP   AMIC2_CTB_OUT   AMIC2_CTB_REF
I2C_SCL_1V8     I2C_SCL_3V3     I2C_SDA_1V8     I2C_SDA_3V3
I2S_TX_FYSYNC   I2S_TX_MCK      I2S_TX_SCK      I2S_TX_SD
I3C_SCL         I3C_SDA         IMU0_INT        IMU1_INT
MAG_INT         PDM_CLK         PDM_DATA        PRESS_SENS_INT
RADAR_INT       RADAR_RESET     RADAR_SPI_CLK   RADAR_SPI_CS
RADAR_SPI_MISO  RADAR_SPI_MOSI  SERIAL_INT0     SERIAL_INT1
SERIAL_INT2     SERIAL_INT3     USER_BUTTON     USER_LED1
USER_LED2       USER_LED_B      USER_LED_G      USER_LED_R
```

The `drive` parameter accepts up to 8 levels, which set the following drive strength for the pin:

- `DRIVE_0`: 1mA/2mA drive current (normal/high speed IO)
- `DRIVE_1`: 2mA/4mA drive current (normal/high speed IO)
- `DRIVE_2`: 3mA/6mA drive current (normal/high speed IO)
- `DRIVE_3`: 4mA/8mA drive current (normal/high speed IO)
- `DRIVE_4`: 5mA/10mA drive current (normal/high speed IO)
- `DRIVE_5`: 6mA/12mA drive current (normal/high speed IO)
- `DRIVE_6`: 7mA/14mA drive current (normal/high speed IO)
- `DRIVE_7`: 8mA/16mA drive current (normal/high speed IO)

For more information about drive strength, check the PSOC™ Edge Datasheet and Architecture Reference Manual.

Note: The following constructor arguments and/or configuration values are NOT supported in this port:

- `alt`: Alternate functionality is not supported.
- `mode`: `Pin.ALT`, `Pin.ALT_OPEN_DRAIN`, and `Pin.ANALOG` modes are not supported.

The following `mode`- `pull` combinations are not supported in this port:

- `Pin.OUT` with `Pin.PULL_UP` or `Pin.PULL_DOWN`

- `Pin.OPEN_DRAIN` with `Pin.PULL_DOWN`

11.3.2 Methods

`Pin.`**`irq`**(*handler=None*, *trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING*, *priority=7*)

The following parameters have port-specific behavior:

- `priority`: Priority values range from 7 (lowest) to 0 (highest). Default is 7.

  Note: All pins on the same port share the same interrupt line. Therefore, only one priority can be set for all pins on the same port. If multiple pins configure interrupts for the same port, the highest priority will be used. If only one pin is configured for an interrupt, its priority can be reconfigured to any value.

Note: The following `irq()` features are not supported in this port:

- `trigger`: The `Pin.IRQ_LOW_LEVEL` and `Pin.IRQ_HIGH_LEVEL` triggers are not supported.
- `wake`: The wake parameter is currently not supported.
- `hard`: This parameter is ignored. It can be passed but currently has no effect.

Note: **None** of the non-core methods from the Pin API are currently implemented for this port.

11.3.3 Signal

There's a higher-level abstraction *machine.Signal* which can be used to invert a pin. Useful for illuminating active-low LEDs using `Signal.on()` or `Signal.value(1)`.

11.4 Real time clock (RTC)

See *machine.RTC*:

```
from machine import RTC
import time

irq_counter = 0

def cback(event):
    global irq_counter
    irq_counter += 1

rtc = RTC()
rtc.init((2023, 1, 1, 0, 0, 0, 0, 0)) # initialise rtc with specific date and time,
                                      # eg. 2023/1/1 00:00:00
rtc.datetime((2017, 8, 23, 2, 12, 48, 0, 0)) # set a specific date and
                                             # time, eg. 2017/8/23 1:12:48
rtc.datetime() # get date and time
```

(continues on next page)

(continued from previous page)

```
rtc.irq(trigger=RTC.ALARM0, handler=cback)
rtc.alarm(1000, repeat=False) # set one-shot short alarm in ms
rtc.alarm_left() # Read the time left for the alarm to expire
time.sleep_ms(1008) # wait sufficient time
print(irq_counter) # Check irq counter

rtc.irq(trigger=RTC.ALARM0, handler=cback)
rtc.alarm(3000, repeat=True) # set periodic short alarm in ms
rtc.cancel() # cancel the alarm

rtc.irq(trigger=RTC.ALARM0, handler=cback)
rtc.alarm((2023, 1, 1, 0, 0, 1, 0, 0), repeat=False) # set one-shot longer duration␣
↪alarm

rtc.memory(b"hello") # write bytes into RTC user memory
rtc.memory() # read bytes from RTC user memory
```

Note: Setting a random week day in 'wday' field is not valid. The underlying library implements the logic to always calculate the right weekday based on the year, date and month passed. However, datetime() will not raise an error for this but rather re-write the field with the last calculated actual value.

Note: RTC API behavior on this port has the following specifics:

- `RTC()` is a singleton constructor with no `id` or additional constructor arguments.
- `rtc.irq()` accepts alarm trigger `0` (`RTC.ALARM0`); `wake` is not implemented.
- `rtc.alarm()` accepts `time` and optional `repeat`; no positional alarm `id` argument is used.
- Input `weekday` in datetime tuples is ignored and hardware computes the weekday from date fields.
- The current `rtc.memory([data])` maximum payload on KIT_PSE84_AI is 28 bytes.

Warning: RTC alarm timing on this port has second-level resolution. Millisecond alarm values are accepted, but are rounded up to whole seconds internally.

11.5 UART

See *machine.UART*.

The following specialization applies to this port:

11.5.1 Constructor

class UART(*id*)

The following parameters are supported with limited configuration:

- `bits`. Only 8 bits.

Note: These parameters are not implemented:

- `txbuf`
- `invert`

11.5.2 Methods

`UART.`**init**(*baudrate=9600*, *bits=8*, *parity=None*, *stop=1*, ***, *...*)

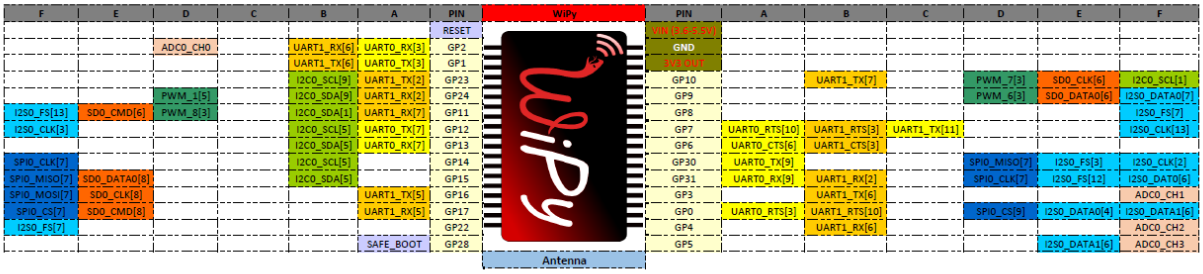
The same parameters as the constructor are supported, with the same limitations.

CHAPTER

TWELVE

QUICK REFERENCE FOR THE WIPY

WiPy pinout and alternate functions table



Timer	Channel	PWM pin
0	A	PWM_1
	B	
1	A	PWM_3
	B	
2	A	
	B	PWM_6
3	A	PWM_7
	B	PWM_8

Remarks:

- *The number in brackets next to each function is the one to be used when remapping the pin. In order to use the pin in GPIO mode, alternate function 0 must be selected*
- *ADC pin input range is 0-1.4V (being 1.8V the absolute maximum that it can withstand). When GP2, GP3, GP4 or GP5 are remapped to the ADC block, 1.8 V is the maximum. If these pins are used in digital mode, then the maximum allowed input is 3.6V.*
- *The heart beat LED is connected to GP25 and also has PWM_3 functionality with the alternate function 9.*

Below is a quick reference for CC3200/WiPy. If it is your first time working with this board please consider reading the following sections first:

12.1 General information about the WiPy

12.1.1 No floating point support

Due to space reasons, there's no floating point support, and no math module. This means that floating point numbers cannot be used anywhere in the code, and that all divisions must be performed using '//' instead of '/'. Example:

```
>>> r = 4 // 2  # this will work
>>> r = 4 / 2   # this WON'T
```

12.1.2 Before applying power

Warning: The GPIO pins of the WiPy are NOT 5V tolerant, connecting them to voltages higher than 3.6V will cause irreparable damage to the board. ADC pins, when configured in analog mode cannot withstand voltages above 1.8V. Keep these considerations in mind when wiring your electronics.

12.1.3 WLAN default behaviour

When the WiPy boots with the default factory configuration starts in Access Point mode with `ssid` that starts with: `wipy-wlan` and `key: www.wipy.io`. Connect to this network and the WiPy will be reachable at `192.168.1.1`. In order to gain access to the interactive prompt, open a telnet session to that IP address on the default port (23). You will be asked for credentials: `login: micro` and `password: python`

12.1.4 Telnet REPL

Linux stock telnet works like a charm (also on OSX), but other tools like putty work quite well too. The default credentials are: **user:** `micro`, **password:** `python`. See *network.Server* for info on how to change the defaults. For instance, on a linux shell (when connected to the WiPy in AP mode):

```
$ telnet 192.168.1.1
```

12.1.5 Local file system and FTP access

There is a small internal file system (a drive) on the WiPy, called `/flash`, which is stored within the external serial flash memory. If a micro SD card is hooked-up and mounted, it will be available as well.

When the WiPy starts up, it always boots from the `boot.py` located in the `/flash` file system. On boot up, the current directory is `/flash`.

The file system is accessible via the native FTP server running in the WiPy. Open your FTP client of choice and connect to:

url: `ftp://192.168.1.1`, **user:** `micro`, **password:** `python`

See *network.Server* for info on how to change the defaults. The recommended clients are: Linux stock FTP (also in OSX), Filezilla and FireFTP. For example, on a linux shell:

```
$ ftp 192.168.1.1
```

The FTP server on the WiPy doesn't support active mode, only passive, therefore, if using the native unix ftp client, just after logging in do:

```
ftp> passive
```

Besides that, the FTP server only supports one data connection at a time. Check out the Filezilla settings section below for more info.

12.1.6 FileZilla settings

Do not use the quick connect button, instead, open the site manager and create a new configuration. In the `General` tab make sure that encryption is set to: `Only use plain FTP (insecure)`. In the Transfer Settings tab limit the max number of connections to one, otherwise FileZilla will try to open a second command connection when retrieving and saving files, and for simplicity and to reduce code size, only one command and one data connections are possible. Other FTP clients might behave in a similar way.

12.1.7 Upgrading the firmware Over The Air

OTA software updates can be performed through the FTP server. Upload the `mcuimg.bin` file to: `/flash/sys/mcuimg.bin` it will take around 6s. You won't see the file being stored inside `/flash/sys/` because it's actually saved bypassing the user file system, so it ends up inside the internal **hidden** file system, but rest assured that it was successfully transferred, and it has been signed with a MD5 checksum to verify its integrity. Now, reset the WiPy by pressing the switch on the board, or by typing:

```
>>> import machine
>>> machine.reset()
```

Software updates can be found in: https://github.com/wipy/wipy/releases (**Binaries.zip**). It's always recommended to update to the latest software, but make sure to read the **release notes** before.

Note: The `bootloader.bin` found inside `Binaries.zip` is there only for reference, it's not needed for the Over The Air update.

In order to check your software version, do:

```
>>> import os
>>> os.uname().release
```

If the version number is lower than the latest release found in the releases, go ahead and update your WiPy!

12.1.8 Boot modes and safe boot

If you power up normally, or press the reset button, the WiPy will boot into standard mode; the `boot.py` file will be executed first, then `main.py` will run.

You can override this boot sequence by pulling `GP28` **up** (connect it to the 3v3 output pin) during reset. This procedure also allows going back in time to old firmware versions. The WiPy can hold up to 3 different firmware versions, which are: the factory firmware plus 2 user updates.

After reset, if `GP28` is held high, the heartbeat LED will start flashing slowly, if after 3 seconds the pin is still being held high, the LED will start blinking a bit faster and the WiPy will select the previous user update to boot. If the previous user update is the desired firmware image, `GP28` must be released before 3 more seconds elapse. If 3 seconds later the pin is still high, the factory firmware will be selected, the LED will flash quickly for 1.5 seconds and the WiPy will proceed to boot. The firmware selection mechanism is as follows:

Safe Boot Pin `GP28` released during:

1st 3 secs window	2nd 3 secs window	Final 1.5 secs window
Safe boot, *latest* firmware is selected	Safe boot, *previous* user update selected	Safe boot, the *factory* firmware is selected

On all of the above 3 scenarios, safe boot mode is entered, meaning that the execution of both `boot.py` and `main.py` is skipped. This is useful to recover from crash situations caused by the user scripts. The selection made during safe boot is not persistent, therefore after the next normal reset the latest firmware will run again.

12.1.9 The heartbeat LED

By default the heartbeat LED flashes once every 4s to signal that the system is alive. This can be overridden through the *wipy* module:

```
>>> import wipy
>>> wipy.heartbeat(False)
```

There are currently 2 kinds of errors that you might see:

1. If the heartbeat LED flashes quickly, then a Python script (eg `main.py`) has an error. Use the REPL to debug it.
2. If the heartbeat LED stays on, then there was a hard fault, you cannot recover from this, the only way out is to press the reset switch.

12.1.10 Details on sleep modes

- `machine.idle()`: Power consumption: ~12mA (in WLAN STA mode). Wake sources: any hardware interrupt (including systick with period of 1ms), no special configuration required.
- `machine.lightsleep()`: 950uA (in WLAN STA mode). Wake sources are `Pin`, `RTC` and `WLAN`
- `machine.deepsleep()`: ~350uA. Wake sources are `Pin` and `RTC`.

12.1.11 Additional details for machine.Pin

On the WiPy board the pins are identified by their string id:

```
from machine import Pin
g = machine.Pin('GP9', mode=Pin.OUT, pull=None, drive=Pin.MED_POWER, alt=-1)
```

You can also configure the Pin to generate interrupts. For instance:

```
from machine import Pin

def pincb(pin):
    print(pin.id())

pin_int = Pin('GP10', mode=Pin.IN, pull=Pin.PULL_DOWN)
pin_int.irq(trigger=Pin.IRQ_RISING, handler=pincb)
# the callback can be triggered manually
```

(continues on next page)

(continued from previous page)

```
pin_int.irq()()
# to disable the callback
pin_int.irq().disable()
```

Now every time a falling edge is seen on the gpio pin, the callback will be executed. Caution: mechanical push buttons have "bounce" and pushing or releasing a switch will often generate multiple edges. See: http://www.eng.utah.edu/~cs5780/debouncing.pdf for a detailed explanation, along with various techniques for debouncing.

All pin objects go through the pin mapper to come up with one of the gpio pins.

For the `drive` parameter the strengths are:

- `Pin.LOW_POWER` - 2mA drive capability.
- `Pin.MED_POWER` - 4mA drive capability.
- `Pin.HIGH_POWER` - 6mA drive capability.

For the `alt` parameter please refer to the pinout and alternate functions table at <https://raw.githubusercontent.com/wipy/wipy/master/docs/PinOUT.png>`_ for the specific alternate functions that each pin supports.

For interrupts, the `priority` can take values in the range 1-7. And the `wake` parameter has the following properties:

- If `wake_from=machine.Sleep.ACTIVE` any pin can wake the board.
- If `wake_from=machine.Sleep.SUSPENDED` pins `GP2`, `GP4`, `GP10`, `GP11`, GP17`` or `GP24` can wake the board. Note that only 1 of this pins can be enabled as a wake source at the same time, so, only the last enabled pin as a `machine.Sleep.SUSPENDED` wake source will have effect.
- If `wake_from=machine.Sleep.SUSPENDED` pins `GP2`, `GP4`, `GP10`, `GP11`, `GP17` and `GP24` can wake the board. In this case all of the 6 pins can be enabled as a `machine.Sleep.HIBERNATE` wake source at the same time.

Additional Pin methods:

`machine.Pin.`**`alt_list`**`()`

Returns a list of the alternate functions supported by the pin. List items are a tuple of the form: `('ALT_FUN_NAME', ALT_FUN_INDEX)`

12.1.12 Additional details for machine.I2C

On the WiPy there is a single hardware I2C peripheral, identified by "0". By default this is the peripheral that is used when constructing an I2C instance. The default pins are GP23 for SCL and GP13 for SDA, and one can create the default I2C peripheral simply by doing:

```
i2c = machine.I2C()
```

The pins and frequency can be specified as:

```
i2c = machine.I2C(freq=400000, scl='GP23', sda='GP13')
```

Only certain pins can be used as SCL/SDA. Please refer to the pinout for further information.

12.1.13 Known issues

Incompatible way to create SSL sockets

SSL sockets need to be created the following way before wrapping them with. `ssl.wrap_socket`:

```
import socket
import ssl
s = socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_SEC)
ss = ssl.wrap_socket(s)
```

Certificates must be used in order to validate the other side of the connection, and also to authenticate ourselves with the other end. Such certificates must be stored as files using the FTP server, and they must be placed in specific paths with specific names.

- The certificate to validate the other side goes in: **'/flash/cert/ca.pem'**
- The certificate to authenticate ourselves goes in: **'/flash/cert/cert.pem'**
- The key for our own certificate goes in: **'/flash/cert/private.key'**

Note: When these files are stored, they are placed inside the internal **hidden** file system (just like firmware updates), and therefore they are never visible.

For instance to connect to the Blynk servers using certificates, take the file `ca.pem` located in the blynk examples folder. and put it in '/flash/cert/'. Then do:

```
import socket
import ssl
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_SEC)
ss = ssl.wrap_socket(s, cert_reqs=ssl.CERT_REQUIRED, ca_certs='/flash/cert/ca.pem')
ss.connect(socket.getaddrinfo('cloud.blynk.cc', 8441)[0][-1])
```

Incompatibilities in hashlib module

Due to hardware implementation details of the WiPy, data must be buffered before being digested, which would make it impossible to calculate the hash of big blocks of data that do not fit in RAM. In this case, since most likely the total size of the data is known in advance, the size can be passed to the constructor and hence the HASH hardware engine of the WiPy can be properly initialized without needing buffering. If `block_size` is to be given, an initial chunk of `data` must be passed as well. **When using this extension, care must be taken to make sure that the length of all intermediate chunks (including the initial one) is a multiple of 4 bytes.** The last chunk may be of any length.

Example:

```
hash = hashlib.sha1('abcd1234', 1001)    # length of the initial piece is multiple of␣
↪4 bytes
hash.update('1234')                       # also multiple of 4 bytes
...
hash.update('12345')                      # last chunk may be of any length
hash.digest()
```

Unrelated function in machine module

main(*filename*)

Set the filename of the main script to run after boot.py is finished. If this function is not called then the default file main.py will be executed.

It only makes sense to call this function from within boot.py.

Adhoc way to control telnet/FTP server via network module

The `Server` class controls the behaviour and the configuration of the FTP and telnet services running on the WiPy. Any changes performed using this class' methods will affect both.

Example:

```
import network
server = network.Server()
server.deinit() # disable the server
# enable the server again with new settings
server.init(login=('user', 'password'), timeout=600)
```

class network.**Server**(*id*, ...)

Create a server instance, see `init` for parameters of initialization.

server.**init**(*, *login=('micro', 'python')*, *timeout=300*)

Init (and effectively start the server). Optionally a new `user`, `password` and `timeout` (in seconds) can be passed.

server.**deinit**()

Stop the server

server.**timeout**([*timeout_in_seconds*])

Get or set the server timeout.

server.**isrunning**()

Returns `True` if the server is running, `False` otherwise.

Adhoc VFS-like support

WiPy doesn't implement full MicroPython VFS support, instead following functions are defined in `os` module:

mount(*block_device*, *mount_point*, *, *readonly=False*)

Mounts a block device (like an `SD` object) in the specified mount point. Example:

```
vfs.mount(sd, '/sd')
```

unmount(*path*)

Unmounts a previously mounted block device from the given path.

mkfs(*block_device or path*)

Formats the specified path, must be either `/flash` or `/sd`. A block device can also be passed like an `SD` object before being mounted.

12.2 WiPy tutorials and examples

Before starting, make sure that you are running the latest firmware, for instructions see *OTA How-To*.

12.2.1 Introduction to the WiPy

To get the most out of your WiPy, there are a few basic things to understand about how it works.

Caring for your WiPy and expansion board

Because the WiPy/expansion board does not have a housing it needs a bit of care:

- Be gentle when plugging/unplugging the USB cable. Whilst the USB connector is well soldered and is relatively strong, if it breaks off it can be very difficult to fix.
- Static electricity can shock the components on the WiPy and destroy them. If you experience a lot of static electricity in your area (eg dry and cold climates), take extra care not to shock the WiPy. If your WiPy came in a ESD bag, then this bag is the best way to store and carry the WiPy as it will protect it against static discharges.

As long as you take care of the hardware, you should be okay. It's almost impossible to break the software on the WiPy, so feel free to play around with writing code as much as you like. If the filesystem gets corrupt, see below on how to reset it. In the worst case you might need to do a safe boot, which is explained in detail in *Boot modes and safe boot*.

Plugging into the expansion board and powering on

The expansion board can power the WiPy via USB. The WiPy comes with a sticker on top of the RF shield that labels all pins, and this should match the label numbers on the expansion board headers. When plugging it in, the WiPy antenna will end up on top of the SD card connector of the expansion board. A video showing how to do this can be found here on YouTube.

Expansion board hardware guide

The document explaining the hardware details of the expansion board can be found in this PDF.

Powering by an external power source

The WiPy can be powered by a battery or other external power source.

Be sure to connect the positive lead of the power supply to VIN, and ground to GND. There is no polarity protection on the WiPy so you must be careful when connecting anything to VIN.

- When powering via `VIN`:

  The input voltage must be between 3.6V and 5.5V.

- When powering via `3V3`:

  The input voltage must be exactly 3V3, ripple free and from a supply capable of sourcing at least 300mA of current

Performing firmware upgrades

For detailed instructions see *OTA How-To*.

12.2.2 Getting a MicroPython REPL prompt

REPL stands for Read Evaluate Print Loop, and is the name given to the interactive MicroPython prompt that you can access on the WiPy. Using the REPL is by far the easiest way to test out your code and run commands. You can use the REPL in addition to writing scripts in `main.py`.

To use the REPL, you must connect to the WiPy either via *telnet*, or with a USB to serial converter wired to one of the two UARTs on the WiPy. To enable REPL duplication on UART0 (the one accessible via the expansion board) do:

```
>>> from machine import UART
>>> import os
>>> uart = UART(0, 115200)
>>> os.dupterm(uart)
```

Place this piece of code inside your `boot.py` so that it's done automatically after reset.

Windows

First you need to install the FTDI drivers for the expansion board's USB to serial converter. Then you need a terminal software. The best option is to download the free program PuTTY: putty.exe.

In order to get to the telnet REPL:

Using putty, select `Telnet` as connection type, leave the default port (23) and enter the IP address of your WiPy (192.168.1.1 when in `WLAN.AP` mode), then click open.

In order to get to the REPL UART:

Using your serial program you must connect to the COM port that you found in the previous step. With PuTTY, click on "Session" in the left-hand panel, then click the "Serial" radio button on the right, then enter you COM port (eg COM4) in the "Serial Line" box. Finally, click the "Open" button.

Mac OS X

Open a terminal and run:

```
$ telnet 192.168.1.1
```

or:

```
$ screen /dev/tty.usbmodem* 115200
```

When you are finished and want to exit `screen`, type CTRL-A CTRL-\. If your keyboard does not have a \-key (i.e. you need an obscure combination for \ like ALT-SHIFT-7) you can remap the `quit` command:

- create `~/.screenrc`
- add `bind q quit`

This will allow you to quit `screen` by hitting CTRL-A Q.

Linux

Open a terminal and run:

```
$ telnet 192.168.1.1
```

or:

```
$ screen /dev/ttyUSB0 115200
```

You can also try `picocom` or `minicom` instead of screen. You may have to use `/dev/ttyUSB01` or a higher number for `ttyUSB`. And, you may need to give yourself the correct permissions to access this devices (eg group `uucp` or `dialout`, or use sudo).

Using the REPL prompt

Now let's try running some MicroPython code directly on the WiPy.

With your serial program open (PuTTY, screen, picocom, etc) you may see a blank screen with a flashing cursor. Press Enter and you should be presented with a MicroPython prompt, i.e. `>>>`. Let's make sure it is working with the obligatory test:

```
>>> print("hello WiPy!")
hello WiPy!
```

In the above, you should not type in the `>>>` characters. They are there to indicate that you should type the text after it at the prompt. In the end, once you have entered the text `print("hello WiPy!")` and pressed Enter, the output on your screen should look like it does above.

If you already know some Python you can now try some basic commands here.

If any of this is not working you can try either a hard reset or a soft reset; see below.

Go ahead and try typing in some other commands. For example:

```
>>> from machine import Pin
>>> import wipy
>>> wipy.heartbeat(False)  # disable the heartbeat
>>> led = Pin('GP25', mode=Pin.OUT)
>>> led(1)
>>> led(0)
>>> led.toggle()
>>> 1 + 2
3
>>> 4 // 2
2
>>> 20 * 'py'
'pypypypypypypypypypypypypypypypypypypypy'
```

Resetting the board

If something goes wrong, you can reset the board in two ways. The first is to press CTRL-D at the MicroPython prompt, which performs a soft reset. You will see a message something like:

```
>>>
MPY: soft reboot
MicroPython v1.4.6-146-g1d8b5e5 on 2015-10-21; WiPy with CC3200
Type "help()" for more information.
>>>
```

If that isn't working you can perform a hard reset (turn-it-off-and-on-again) by pressing the RST switch (the small black button next to the heartbeat LED). During telnet, this will end your session, disconnecting whatever program that you used to connect to the WiPy.

12.2.3 Getting started with Blynk and the WiPy

Blynk provides iOS and Android apps to control any hardware over the Internet or directly using Bluetooth. You can easily build graphic interfaces for all your projects by simply dragging and dropping widgets, right on your smartphone.

Before anything else, make sure that your WiPy is running the latest software, check *OTA How-To* for instructions.

1. Get the Blynk library and put it in `/flash/lib/` via FTP.
2. Get the Blynk example for WiPy, edit the network settings, and afterwards upload it to `/flash/` via FTP as well.
3. Follow the instructions on each example to setup the Blynk dashboard on your smartphone or tablet.
4. Give it a try, for instance:

```
>>> execfile('sync_virtual.py')
```

12.2.4 WLAN step by step

The WLAN is a system feature of the WiPy, therefore it is always enabled (even while in `machine.SLEEP`), except when deepsleep mode is entered.

In order to retrieve the current WLAN instance, do:

```
>>> from network import WLAN
>>> wlan = WLAN() # we call the constructor without params
```

You can check the current mode (which is always `WLAN.AP` after power up):

```
>>> wlan.mode()
```

Warning: When you change the WLAN mode following the instructions below, your WLAN connection to the WiPy will be broken. This means you will not be able to run these commands interactively over the WLAN.

There are two ways around this::

1. put this setup code into your *boot.py file* so that it gets executed automatically after reset.
2. *duplicate the REPL on UART*, so that you can run commands via USB.

Connecting to your home router

The WLAN network card always boots in `WLAN.AP` mode, so we must first configure it as a station:

```
from network import WLAN
wlan = WLAN(mode=WLAN.STA)
```

Now you can proceed to scan for networks:

```
nets = wlan.scan()
for net in nets:
    if net.ssid == 'mywifi':
        print('Network found!')
        wlan.connect(net.ssid, auth=(net.sec, 'mywifikey'), timeout=5000)
        while not wlan.isconnected():
            machine.idle() # save power while waiting
        print('WLAN connection succeeded!')
        break
```

Assigning a static IP address when booting

If you want your WiPy to connect to your home router after boot-up, and with a fixed IP address so that you can access it via telnet or FTP, use the following script as /flash/boot.py:

```
import machine, network
from network import WLAN
wlan = WLAN() # get current object, without changing the mode

if machine.reset_cause() != machine.SOFT_RESET:
    wlan.init(WLAN.STA)
    # configuration below MUST match your home router settings!!
    network.ipconfig(dns='8.8.8.8')
    wlan.ipconfig(addr4='192.168.0.107/24', gw4='192.168.0.1')

if not wlan.isconnected():
    # change the line below to match your network ssid, security and password
    wlan.connect('mywifi', auth=(WLAN.WPA2, 'mywifikey'), timeout=5000)
    while not wlan.isconnected():
        machine.idle() # save power while waiting
```

Note: Notice how we check for the reset cause and the connection status, this is crucial in order to be able to soft reset the WiPy during a telnet session without breaking the connection.

12.2.5 Hardware timers

Timers can be used for a great variety of tasks, calling a function periodically, counting events, and generating a PWM signal are among the most common use cases. Each timer consists of two 16-bit channels and this channels can be tied together to form one 32-bit timer. The operating mode needs to be configured per timer, but then the period (or the frequency) can be independently configured on each channel. By using the callback method, the timer event can call a Python function.

Example usage to toggle an LED at a fixed frequency:

```
from machine import Timer
from machine import Pin
led = Pin('GP16', mode=Pin.OUT)                    # enable GP16 as output to drive the␣
↪LED
tim = Timer(3)                                     # create a timer object using timer 3
tim.init(mode=Timer.PERIODIC)                      # initialize it in periodic mode
tim_ch = tim.channel(Timer.A, freq=5)              # configure channel A at a frequency␣
↪of 5Hz
tim_ch.irq(handler=lambda t:led.toggle(), trigger=Timer.TIMEOUT)        # toggle a␣
↪LED on every cycle of the timer
```

Example using named function for the callback:

```
from machine import Timer
from machine import Pin
tim = Timer(1, mode=Timer.PERIODIC, width=32)
tim_a = tim.channel(Timer.A | Timer.B, freq=1)    # 1 Hz frequency requires a 32 bit␣
↪timer

led = Pin('GP16', mode=Pin.OUT) # enable GP16 as output to drive the LED

def tick(timer):                # we will receive the timer object when being called
    global led
    led.toggle()                # toggle the LED

tim_a.irq(handler=tick, trigger=Timer.TIMEOUT)         # create the interrupt
```

Further examples:

```
from machine import Timer
tim1 = Timer(1, mode=Timer.ONE_SHOT)                                    # initialize it in␣
↪one shot mode
tim2 = Timer(2, mode=Timer.PWM)                                         # initialize it in␣
↪PWM mode
tim1_ch = tim1.channel(Timer.A, freq=10, polarity=Timer.POSITIVE)  # start the event␣
↪counter with a frequency of 10Hz and triggered by positive edges
tim2_ch = tim2.channel(Timer.B, freq=10000, duty_cycle=5000)       # start the PWM on␣
↪channel B with a 50% duty cycle
tim2_ch.freq(20)                                                        # set the␣
↪frequency (can also get)
tim2_ch.duty_cycle(3010)                                                # set the duty␣
↪cycle to 30.1% (can also get)
tim2_ch.duty_cycle(3020, Timer.NEGATIVE)                                # set the duty␣
↪cycle to 30.2% and change the polarity to negative
tim2_ch.period(2000000)                                                 # change the␣
↪period to 2 seconds
```

Additional constants for Timer class

`Timer.`**`PWM`**
PWM timer operating mode.

`Timer.`**`A`**

`Timer.`**`B`**
Selects the timer channel. Must be ORed (`Timer.A | Timer.B`) when using a 32-bit timer.

`Timer.`**`POSITIVE`**

`Timer.`**`NEGATIVE`**
Timer channel polarity selection (only relevant in PWM mode).

`Timer.`**`TIMEOUT`**

`Timer.`**`MATCH`**
Timer channel IRQ triggers.

12.2.6 Reset and boot modes

There are soft resets and hard resets.

- A soft reset simply clears the state of the MicroPython virtual machine, but leaves hardware peripherals unaffected. To do a soft reset, simply press **Ctrl+D** on the REPL, or within a script do:

```
import sys
sys.exit()
```

- A hard reset is the same as performing a power cycle to the board. In order to hard reset the WiPy, press the switch on the board or:

```
import machine
machine.reset()
```

For more information, see *Reset and Boot Sequence*.

Safe boot

If something goes wrong with your WiPy, don't panic! It is almost impossible for you to break the WiPy by programming the wrong thing.

The first thing to try is to boot in safe mode: this temporarily skips execution of `boot.py` and `main.py` and gives default WLAN settings.

If you have problems with the filesystem you can *format the internal flash drive*.

To boot in safe mode, follow the detailed instructions described *here*.

In safe mode, the `boot.py` and `main.py` files are not executed, and so the WiPy boots up with default settings. This means you now have access to the filesystem, and you can edit `boot.py` and `main.py` to fix any problems.

Entering safe mode is temporary, and does not make any changes to the files on the WiPy.

Factory reset the filesystem

If you WiPy's filesystem gets corrupted (very unlikely, but possible), you can format it very easily by doing:

```
>>> import os
>>> os.mkfs('/flash')
```

Resetting the filesystem deletes all files on the internal WiPy storage (not the SD card), and restores the files `boot.py` and `main.py` back to their original state after the next reset.

12.3 General board control (including sleep modes)

See the *machine* module:

```
import machine

help(machine) # display all members from the machine module
machine.freq() # get the CPU frequency
machine.unique_id() # return the 6-byte unique id of the board (the WiPy's MAC␣
↪address)

machine.idle()          # average current decreases to (~12mA), any interrupts wake it␣
↪up
machine.lightsleep()    # everything except for WLAN is powered down (~950uA avg.␣
↪current)
                        # wakes from Pin, RTC or WLAN
machine.deepsleep()     # deepest sleep mode, MCU starts from reset. Wakes from Pin and␣
↪RTC.
```

12.4 Pins and GPIO

See *machine.Pin*.

```
from machine import Pin

# initialize GP2 in gpio mode (alt=0) and make it an output
p_out = Pin('GP2', mode=Pin.OUT)
p_out.value(1)
p_out.value(0)
p_out.toggle()
p_out(True)

# make GP1 an input with the pull-up enabled
p_in = Pin('GP1', mode=Pin.IN, pull=Pin.PULL_UP)
p_in() # get value, 0 or 1
```

12.5 Timers

See *machine.TimerWiPy* and *machine.Pin*. Timer `id`'s take values from 0 to 3.:

```
from machine import Timer
from machine import Pin

tim = Timer(0, mode=Timer.PERIODIC)
tim_a = tim.channel(Timer.A, freq=1000)
tim_a.freq(5) # 5 Hz

p_out = Pin('GP2', mode=Pin.OUT)
tim_a.irq(trigger=Timer.TIMEOUT, handler=lambda t: p_out.toggle())
```

12.6 PWM (pulse width modulation)

See *machine.Pin* and *machine.Timer*.

```
from machine import Timer

# timer 1 in PWM mode and width must be 16 buts
tim = Timer(1, mode=Timer.PWM, width=16)

# enable channel A @1KHz with a 50.55% duty cycle
tim_a = tim.channel(Timer.A, freq=1000, duty_cycle=5055)
```

12.7 ADC (analog to digital conversion)

See *machine.ADCWiPy*.

```
from machine import ADC

adc = ADC()
apin = adc.channel(pin='GP3')
apin() # read value, 0-4095
```

12.8 UART (serial bus)

See *machine.UART*.

```
from machine import UART
uart = UART(0, baudrate=9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

12.9 SPI bus

See *machine.SPI*.

```
from machine import SPI

# configure the SPI controller @ 2MHz
spi = SPI(0, SPI.CONTROLLER, baudrate=2_000_000, polarity=0, phase=0)
spi.write('hello')
spi.read(5) # receive 5 bytes on the bus
rbuf = bytearray(5)
spi.write_readinto('hello', rbuf) # send and receive 5 bytes
```

12.10 I2C bus

See *machine.I2C*.

```
from machine import I2C
# configure the I2C bus
i2c = I2C(baudrate=100000)
i2c.scan() # returns list of peripheral addresses
i2c.writeto(0x42, 'hello') # send 5 bytes to peripheral with address 0x42
i2c.readfrom(0x42, 5) # receive 5 bytes from peripheral
i2c.readfrom_mem(0x42, 0x10, 2) # read 2 bytes from peripheral 0x42, peripheral␣
↪memory 0x10
i2c.writeto_mem(0x42, 0x10, 'xy') # write 2 bytes to peripheral 0x42, peripheral␣
↪memory 0x10
```

12.11 Watchdog timer (WDT)

See *machine.WDT*.

```
from machine import WDT

# enable the WDT with a timeout of 5s (1s is the minimum)
wdt = WDT(timeout=5000)
wdt.feed()
```

12.12 Real time clock (RTC)

See *machine.RTC*

```
from machine import RTC

rtc = RTC() # init with default time and date
rtc = RTC(datetime=(2015, 8, 29, 9, 0, 0, 0, None)) # init with a specific time and␣
↪date
print(rtc.now())
```

(continues on next page)

(continued from previous page)

```
def alarm_handler (rtc_o):
    pass
    # do some non blocking operations
    # warning printing on an irq via telnet is not
    # possible, only via UART

# create a RTC alarm that expires after 5 seconds
rtc.alarm(time=5000, repeat=False)

# enable RTC interrupts
rtc_i = rtc.irq(trigger=RTC.ALARM0, handler=alarm_handler, wake=machine.SLEEP)

# go into suspended mode waiting for the RTC alarm to expire and wake us up
machine.lightsleep()
```

12.13 SD card

See *machine.SD*.

```
from machine import SD
import vfs

# clock pin, cmd pin, data0 pin
sd = SD(pins=('GP10', 'GP11', 'GP15'))
# or use default ones for the expansion board
sd = SD()
vfs.mount(sd, '/sd')
```

12.14 WLAN (WiFi)

See *network.WLAN* and *machine*.

```
import machine, network
from network import WLAN

# configure the WLAN subsystem in station mode (the default is AP)
wlan = WLAN(mode=WLAN.STA)
# go for fixed IP settings
network.ipconfig(dns='8.8.8.8')
wlan.ipconfig(addr4='192.168.0.107/24', gw4='192.168.0.1')
wlan.scan()     # scan for available networks
wlan.connect(ssid='mynetwork', auth=(WLAN.WPA2, 'mynetworkkey'))
while not wlan.isconnected():
    pass
# enable wake on WLAN
wlan.irq(trigger=WLAN.ANY_EVENT, wake=machine.SLEEP)
# go to sleep
machine.lightsleep()
# now, connect to the FTP or the Telnet server and the WiPy will wake-up
```

12.15 Telnet and FTP server

See *network.Server*

```
from network import Server

# init with new user, password and seconds timeout
server = Server(login=('user', 'password'), timeout=60)
server.timeout(300) # change the timeout
server.timeout() # get the timeout
server.isrunning() # check whether the server is running or not
```

12.16 Heart beat LED

See *wipy*.

```
import wipy

wipy.heartbeat(False)   # disable the heartbeat LED
wipy.heartbeat(True)    # enable the heartbeat LED
wipy.heartbeat()        # get the heartbeat state
```

CHAPTER

THIRTEEN

QUICK REFERENCE FOR THE UNIX AND WINDOWS PORTS

13.1 Command line options

Usage:

```
micropython [ -h ] [ -i ] [ -O<level> ] [ -v ] [ -X <option> ] [ -c <command> | -m
↪<module> | <script> ] [ <args> ]
```

Invocation options:

-c <command>

Runs the code in <command>. The code can be one or more Python statements.

-m <module>

Runs the module <module>. The module must be in sys.path.

<script>

Runs the file <script>. The script must be a valid MicroPython source code file.

If none of the 3 options above are given, then MicroPython is run in an interactive REPL mode.

<args>

Any additional arguments after the module or script will be passed to sys.argv (not supported with the *-c* option).

General options:

-h

Prints a help message containing the command line usage and exits.

-i

Enables inspection. When this flag is set, MicroPython will enter the interactive REPL mode after the command, module or script has finished. This can be useful for debugging the state after an unhandled exception. Also see the *MICROPYINSPECT* environment variable.

-O | -O<level> | -OO...

Sets the optimization level. The O can be followed by a number or can be repeated multiple times to indicate the level. E.g. -O3 is the same as -OOO.

-v

Increases the verbosity level. This option can be given multiple times. This option only has an effect if MICROPY_DEBUG_PRINTERS was enabled when MicroPython itself was compiled.

-X <option>

Specifies additional implementation-specific options. Possible options are:

- `-X compile-only` compiles the command, module or script but does not run it.
- `-X emit={bytecode,native,viper}` sets the default code emitter. Native emitters may not be available depending on the settings when MicroPython itself was compiled.
- `-X heapsize=<n>[w][K|M]` sets the heap size for the garbage collector. The suffix `w` means words instead of bytes. `K` means x1024 and `M` means x1024x1024.
- `-X realtime` sets thread priority to realtime. This can be used to improve timer precision. Only available on macOS.

13.2 Environment variables

MICROPYPATH

Overrides the default search path for MicroPython libraries. `MICROPYPATH` should be set to a colon (semi-colon for Windows port) separated list of directories. If `MICROPYPATH` is not defined, the search path will be `~/.micropython/lib:/usr/lib/micropython` (`~/.micropython/lib` for Windows port) or the value of the `MICROPY_PY_SYS_PATH_DEFAULT` option if it was set when MicroPython itself was compiled.

MICROPYINSPECT

Enables inspection. If `MICROPYINSPECT` is set to a non-empty string, it has the same effect as setting the *-i* command line option.

CHAPTER

FOURTEEN

QUICK REFERENCE FOR THE ZEPHYR PORT

Below is a quick reference for the Zephyr port. If it is your first time working with this port please consider reading the following sections first:

14.1 General information about the Zephyr port

The Zephyr Project is a Linux Foundation hosted Collaboration Project. It's an open source collaborative effort uniting developers and users in building a small, scalable, real-time operating system (RTOS) optimized for resource-constrained devices, across multiple architectures.

14.1.1 Multitude of boards

There is a multitude of modules and boards from different sources that are supported by the Zephyr OS. All boards supported by Zephyr (with standard level of features support, like UART console) should work with MicroPython (but not all were tested). The FRDM-K64f board is taken as a reference board for the port for this documentation. If you have another board, please make sure you have a datasheet, schematics and other reference materials for your board handy to look up various aspects of your board functioning.

For a full list of Zephyr supported boards click here (external link)

14.2 MicroPython tutorial for the Zephyr port

This tutorial is intended to get you started with the Zephyr port.

14.2.1 Getting started with MicroPython on the Zephyr port

Let's get started!

Requirements

To use the MicroPython Zephyr port, you will need a Zephyr supported board (for a list of acceptable boards see *General information about the Zephyr port*).

Powering up

If your board has a USB connector on it then most likely it is powered through this when connected to your PC. Otherwise you will need to power it directly. Please refer to the documentation for your board for further details.

Getting and deploying the firmware

The first step you will need to do is either clone the MicroPython repository or download it from the MicroPython downloads page. If you are an end user of MicroPython, it is recommended to start with the stable firmware builds. If you would like to work on development, you may follow the daily builds on git.

Next, follow the Zephyr port readme document (`ports/zephyr/README.md`) to build and run the application on your board.

14.2.2 Getting a MicroPython REPL prompt

REPL stands for Read Evaluate Print Loop, and is the name given to the interactive MicroPython prompt that you can access on your board through Zephyr. It is recommended to use REPL to test out your code and run commands.

REPL over the serial port

The REPL is available on a UART serial peripheral specified for the board by the `zephyr,console` devicetree node. The baudrate of the REPL is 115200. If your board has a USB-serial converter on it then you should be able to access the REPL directly from your PC.

To access the prompt over USB-serial you will need to use a terminal emulator program. For a Linux or Mac machine, open a terminal and run:

```
screen /dev/ttyACM0 115200
```

You can also try `picocom` or `minicom` instead of screen. You may have to use `/dev/ttyACM1` or a higher number for `ttyACM`. Additional permissions may be necessary to access this device (eg group `uucp` or `dialout`, or use sudo). For Windows, get a terminal software, such as puTTY and connect via a serial session using the proper COM port.

Using the REPL

With your serial program open (PuTTY, screen, picocom, etc) you may see a blank screen with a flashing cursor. Press Enter (or reset the board) and you should be presented with the following text:

```
*** Booting Zephyr OS build v4.4.0 ***
MicroPython v1.29.0-preview.69.g8a56be6660 on 2026-04-27; zephyr-frdm_k64f with␣
↪mk64f12
Type "help()" for more information.
>>>
```

Now you can try running MicroPython code directly on your board.

Anything you type at the prompt, indicated by >>>, will be executed after you press the Enter key. If there is an error with the text that you enter then an error message is printed.

Start by typing the following at the prompt to make sure it is working:

```
>>> print("hello world!")
hello world!
```

If you already know some python you can now try some basic commands here. For example:

```
>>> 1 + 2
3
>>> 1 / 2
0.5
>>> 3 * 'Zephyr'
ZephyrZephyrZephyr
```

If your board has an LED, you can blink it using the following code:

```
>>>import time
>>>from machine import Pin

>>>LED = Pin(("GPIO_1", 21), Pin.OUT)
>>>while True:
...     LED.value(1)
...     time.sleep(0.5)
...     LED.value(0)
...     time.sleep(0.5)
```

The above code uses an LED location for a FRDM-K64F board (port B, pin 21; following Zephyr conventions ports are identified by "GPIO_x", where *x* starts from 0). You will need to adjust it for another board using the board's reference materials.

14.2.3 Filesystems and Storage

Storage modules support virtual filesystem with FAT and littlefs formats, backed by either Zephyr DiskAccess or FlashArea (flash map) APIs depending on which the board supports.

See vfs Filesystem Mounting.

Automatic Filesystem Mounting

MicroPython on Zephyr automatically attempts to mount available storage at startup:

1. **Flash storage** (if available): Automatically mounted at `/flash` using the littlefs2 filesystem. If the filesystem doesn't exist, it will be created automatically.
2. **Disk storage** (such as SD cards): Automatically detected and mounted at `/<disk_name>` (e.g., `/sd` for SD cards).
3. **Working directory**: The working directory is set to the mounted disk if available, otherwise to `/flash` if flash storage is available.
4. **Module search path**: The `sys.path` is automatically updated to include `/<disk>/lib` for each mounted disk and `/flash/lib` for flash storage.

This automatic mounting is performed by the `_boot.py` module at startup. For most use cases, you don't need to manually mount filesystems.

Disk Access

The *zephyr.DiskAccess* class can be used to access storage devices, such as SD cards. This class uses Zephyr Disk Access API and implements the *vfs.AbstractBlockDev* protocol.

For use with SD card controllers, SD cards must be present at boot & not removed; they will be auto detected and initialized by filesystem at boot. Use the disk driver interface and a file system to access SD cards via disk access (see below).

The available disk names can be accessed via the `DiskAccess.disks` attribute, which contains a tuple of available disk names (e.g., `('SDHC',)` or `('SD',)`).

Example usage of FatFS with an SD card on the mimxrt1050_evk board:

```
import vfs
from zephyr import DiskAccess

# List available disks
print(DiskAccess.disks)                 # prints available disk names, e.g., ('SDHC',)

bdev = DiskAccess('SDHC')               # create block device object using DiskAccess
vfs.VfsFat.mkfs(bdev)                   # create FAT filesystem object using the disk␣
↪storage block
vfs.mount(bdev, '/sd')                  # mount the filesystem at the SD card␣
↪subdirectory
with open('/sd/hello.txt','w') as f:    # open a new file in the directory
    f.write('Hello world')              # write to the file
print(open('/sd/hello.txt').read())     # print contents of the file
```

Note: In most cases, disks are automatically mounted at startup and manual mounting is not necessary.

Flash Area

The *zephyr.FlashArea* class can be used to implement a low-level storage system or customize filesystem configurations. To store persistent data on the device, using a higher-level filesystem API is recommended (see below).

This class uses Zephyr Flash map API and implements the *vfs.AbstractBlockDev* protocol.

The available flash area names can be accessed via the `FlashArea.areas` dictionary, which maps partition labels to their IDs (e.g., `{'storage': 1, 'scratch': 4}`).

Example usage with the internal flash on the reel_board or the rv32m1_vega_ri5cy board:

```
import vfs
from zephyr import FlashArea

# List available flash areas
print(FlashArea.areas)                          # prints available areas, e.g., {
↪'storage': 1, 'scratch': 4}

bdev = FlashArea(FlashArea.areas['storage'], 4096)    # create block device object␣
↪using FlashArea
vfs.VfsLfs2.mkfs(bdev)                                # create Little filesystem␣
↪object using the flash area block
```

(continues on next page)

(continued from previous page)

```
vfs.mount(bdev, '/flash')                          # mount the filesystem at the␣
↪flash storage subdirectory
with open('/flash/hello.txt','w') as f:            # open a new file in the␣
↪directory
    f.write('Hello world')                         # write to the file
print(open('/flash/hello.txt').read())             # print contents of the file
```

Note: In most cases, the flash storage partition is automatically mounted at `/flash` at startup with automatic filesystem creation if needed, so manual mounting is not necessary.

For boards such as the frdm_k64f in which the MicroPython application spills into the default flash storage partition, use the scratch partition by replacing `'storage'` with `'scratch'`.

14.2.4 GPIO Pins

Use *machine.Pin* to control I/O pins.

For Zephyr, pins are initialized using a tuple of port and pin number `(\"GPIO_x\", pin#)` for the `id` value. For example to initialize a pin for the red LED on a FRDM-k64 board:

```
LED = Pin(("GPIO_1", 22), Pin.OUT)
```

Reference your board's datasheet or Zephyr documentation for pin numbers, see below for more examples.

Table 1: Pin Formatting

Board	Pin	Format
frdm_k64f	Red LED = PTB22	("GPIO_1", 22)
96b_carbon	LED1 = PD2	("GPIOD", 2)
mimxrt685_evk_cm33	Green LED = PIO0_14	("GPIO0", 14)

Interrupts

The Zephyr port also supports interrupt handling for Pins using *machine.Pin.irq()*. To respond to Pin change IRQs run:

```
from machine import Pin

SW2 = Pin(("GPIO_2", 6), Pin.IN)             # create Pin object for switch 2
SW3 = Pin(("GPIO_0", 4), Pin.IN)             # create Pin object for switch 3

SW2.irq(lambda t: print("SW2 changed"))      # print message when SW2 state is changed␣
↪(triggers change IRQ)
SW3.irq(lambda t: print("SW3 changed"))      # print message when SW3 state is changed␣
↪(triggers change IRQ)

while True:                                  # wait
    pass
```

14.3 Running MicroPython

See the corresponding section of the tutorial: *Getting started with MicroPython on the ESP8266*.

14.4 Delay and timing

Use the `time` module:

```
import time

time.sleep(1)               # sleep for 1 second
time.sleep_ms(500)          # sleep for 500 milliseconds
time.sleep_us(10)           # sleep for 10 microseconds
start = time.ticks_ms()     # get millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start) # compute time difference
```

14.5 Pins and GPIO

Use the *machine.Pin* class:

```
from machine import Pin

pin = Pin(("gpiob", 21), Pin.IN)    # create input pin on GPIO port B
print(pin)                          # print pin port and number

pin.init(Pin.OUT, Pin.PULL_UP, value=1)     # reinitialize pin

pin.value(1)                        # set pin to high
pin.value(0)                        # set pin to low

pin.on()                            # set pin to high
pin.off()                           # set pin to low

pin = Pin(("gpiob", 21), Pin.IN)              # create input pin on GPIO port B

pin = Pin(("gpiob", 21), Pin.OUT, value=1)    # set pin high on creation

pin = Pin(("gpiob", 21), Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor

switch = Pin(("gpioc", 6), Pin.IN)            # create input pin for a switch
switch.irq(lambda t: print("SW2 changed"))    # enable an interrupt when switch state␣
↪is changed
```

14.6 PWM

Use the *machine.PWM* class:

```
from machine import PWM

pwm = PWM(("pwm0", 0), freq=3921568, duty_ns=200, invert=True)     # create pwm on PWM0
print(pwm)                                                         # print pwm

print(pwm.duty_ns())                                               # print pwm duty␣
↪cycle in nanoseconds
pwm.duty_ns(255)                                                   # set new pwm duty␣
↪cycle in nanoseconds

pwm.deinit()
```

14.7 Hardware I2C bus

Hardware I2C is accessed via the *machine.I2C* class:

```
from machine import I2C

i2c = I2C("i2c0")           # construct an i2c bus
print(i2c)                  # print device name

i2c.scan()                  # scan the device for available I2C slaves

i2c.readfrom(0x1D, 4)                # read 4 bytes from slave 0x1D
i2c.readfrom_mem(0x1D, 0x0D, 1)      # read 1 byte from slave 0x1D at slave memory␣
↪0x0D

i2c.writeto(0x1D, b'abcd')           # write to slave with address 0x1D
i2c.writeto_mem(0x1D, 0x0D, b'ab')   # write to slave 0x1D at slave memory 0x0D

buf = bytearray(8)                  # create buffer of size 8
i2c.writeto(0x1D, b'abcd')          # write buf to slave 0x1D
```

14.8 Hardware SPI bus

Hardware SPI is accessed via the *machine.SPI* class:

```
from machine import SPI

spi = SPI("spi0")           # construct a SPI bus with default configuration
spi.init(baudrate=100000, polarity=0, phase=0, bits=8, firstbit=SPI.MSB) # set␣
↪configuration

# equivalently, construct the SPI bus and set configuration at the same time
spi = SPI("spi0", baudrate=100000, polarity=0, phase=0, bits=8, firstbit=SPI.MSB)
print(spi)                  # print device name and bus configuration

spi.read(4)                 # read 4 bytes on MISO
```

(continues on next page)

(continued from previous page)

```
spi.read(4, write=0xF)      # read 4 bytes while writing 0xF on MOSI

buf = bytearray(8)          # create a buffer of size 8
spi.readinto(buf)           # read into the buffer (reads number of bytes equal to␣
↪the buffer size)
spi.readinto(buf, 0xF)      # read into the buffer while writing 0xF on MOSI

spi.write(b'abcd')          # write 4 bytes on MOSI

buf = bytearray(4)                  # create buffer of size 8
spi.write_readinto(b'abcd', buf)    # write to MOSI and read from MISO into the buffer
spi.write_readinto(buf, buf)        # write buf to MOSI and read back into the buf
```

14.9 Analog to Digital Converter (ADC)

Use the *machine.ADC* class.

Example of using ADC to read a pin's analog value (the `zephyr,user` node must contain the `io-channels` property containing all the ADC channels):

```
from machine import ADC

adc = ADC(("adc", 0))
adc.read_uv()
```

14.10 Disk Access

Storage devices such as SD cards are automatically mounted at startup (e.g., at `/sd`). For manual mounting, use the *zephyr.DiskAccess* class:

```
import vfs
from zephyr import DiskAccess

print(DiskAccess.disks)             # list available disk names, e.g., ('SDHC',)

block_dev = DiskAccess('SDHC')      # create a block device object for an SD card
vfs.VfsFat.mkfs(block_dev)          # create FAT filesystem object using the disk␣
↪storage block
vfs.mount(block_dev, '/sd')         # mount the filesystem at the SD card subdirectory

# with the filesystem mounted, files can be manipulated as normal
with open('/sd/hello.txt','w') as f:     # open a new file in the directory
    f.write('Hello world')                  # write to the file
print(open('/sd/hello.txt').read())      # print contents of the file
```

14.11 Flash Area

Flash storage is automatically mounted at `/flash` at startup with automatic filesystem creation. For manual mounting, use the *zephyr.FlashArea* class:

```
import vfs
from zephyr import FlashArea

print(FlashArea.areas)              # list available areas, e.g., {'storage': 1,
↪'scratch': 4}

block_dev = FlashArea(FlashArea.areas['scratch'], 4096)  # creates a block device␣
↪object using the scratch partition
vfs.VfsLfs2.mkfs(block_dev)         # create filesystem in lfs2 format using the␣
↪flash block device
vfs.mount(block_dev, '/flash')      # mount the filesystem at the flash subdirectory

# with the filesystem mounted, files can be manipulated as normal
with open('/flash/hello.txt','w') as f:     # open a new file in the directory
    f.write('Hello world')                  # write to the file
print(open('/flash/hello.txt').read())      # print contents of the file
```

14.12 Sensor

Use the *zsensor.Sensor* class to access sensor data:

```
import zsensor
from zsensor import Sensor

accel = Sensor("fxos8700")    # create sensor object for the accelerometer

accel.measure()               # obtain a measurement reading from the accelerometer

# each of these prints the value taken by measure()
accel.get_float(zsensor.ACCEL_X)  # print measurement value for accelerometer X-axis␣
↪sensor channel as float
accel.get_millis(zsensor.ACCEL_Y) # print measurement value for accelerometer Y-axis␣
↪sensor channel in millionths
accel.get_micro(zsensor.ACCEL_Z)  # print measurement value for accelerometer Z-axis␣
↪sensor channel in thousandths
accel.get_int(zsensor.ACCEL_X)    # print measurement integer value only for␣
↪accelerometer X-axis sensor channel
```

The channel IDs that are used as arguments to the *zsensor.Sensor.get_int()*, *zsensor.Sensor.get_float()*, *zsensor.Sensor.get_millis()*, and *zsensor.Sensor.get_micros()* methods are constants in the *zsensor* module.

You can use the *zsensor.Sensor.attr_set()* method to set sensor attributes like full-scale range and update rate:

```
# Example for XIAO BLE NRF52840 SENSE
from zsensor import *
accel = Sensor('lsm6ds3tr_c')  # name from Devicetree
# Set full-scale to 2g (19.613300 m/sec^2)
```

(continues on next page)

(continued from previous page)

```
# units are micro-m/s^2 (given as a float)
accel.attr_set(ACCEL_XYZ, ATTR_FULL_SCALE, 19.613300)
# Set sampling frequency to 104 Hz (as a pair of integers)
accel.attr_set(ACCEL_XYZ, ATTR_SAMPLING_FREQUENCY, 104, 0)
accel.measure()
accel.get_float(ACCEL_X) # -0.508 (m/s^2)
accel.get_float(ACCEL_Y) # -3.62 (m/s^2)
accel.get_float(ACCEL_Z) # 9.504889 (m/s^2)
```

There are also the *zsensor.Sensor.attr_get_float()*, *zsensor.Sensor.attr_get_int()*, *zsensor.Sensor.attr_get_millis()*, and *zsensor.Sensor.attr_get_micros()* methods, but many sensors do not support these:

```
full_scale = accel.attr_get_float(ATTR_FULL_SCALE)
```

The attribute IDs that are used as arguments to the *zsensor.Sensor.attr_set()*, *zsensor.Sensor.attr_get_float()*, *zsensor.Sensor.attr_get_int()*, *zsensor.Sensor.attr_get_millis()*, and *zsensor.Sensor.attr_get_micros()* methods are constants in the *zsensor* module named `ATTR_*`.

14.13 Display

Use the *zephyr.Display* class to control displays provided by Zephyr:

```
import zephyr, framebuf

# Create Display instance from Node identifier
display = zephyr.Display("ssd1306@3c")

# Create a FrameBuffer of the correct format for the display
fb = framebuf.FrameBuffer(bytearray(int(64*128/8)), 128, 64, framebuf.MVLSB)

# Disable blanking
display.blanking(False)

# Write text to framebuf
fb.text('Hello World', 0, 0, 1)

# Write framebuf buffer to Display
display.write(fb)
```

Or:

```
# framebuf module must be available for as_framebuf to generate the augmented␣
↪FrameBuffer class
import zephyr, framebuf

# Create Display instance for the first display configured
display = zephyr.Display(0)

# Disable blanking
display.blanking(False)

# Get a FrameBuffer for the Display as currently configured
```

(continues on next page)

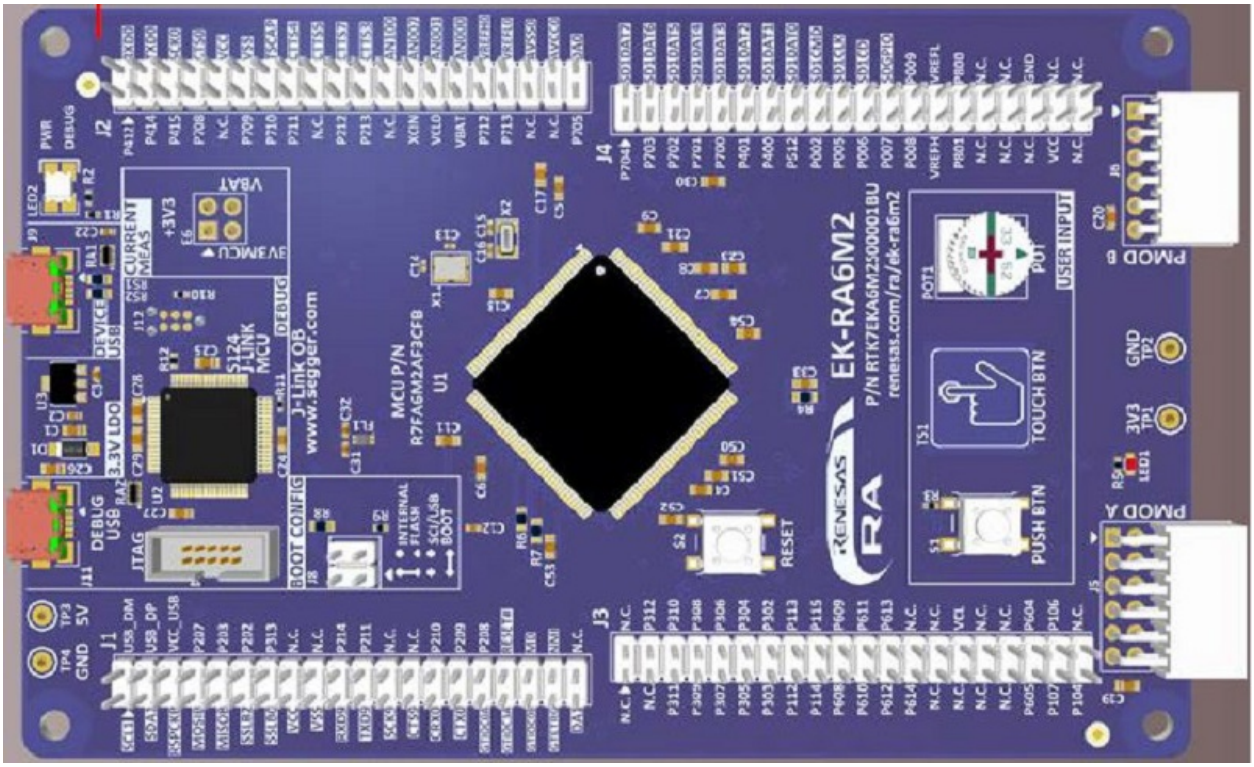
(continued from previous page)

```
fb = display.as_framebuf()

# Write text to framebuf
fb.text('Hello World', 0, 0, 1)

# Write framebuf buffer to Display
fb.show()
```

CHAPTER

FIFTEEN

QUICK REFERENCE FOR THE RENESAS RA



The Renesas EK-RA6M2 board.

Below is a quick reference for the Renesas RA boards. If it is your first time working with this board, it may be useful to get an overview of the microcontroller and the board:

15.1 General information about Renesas RA port

15.1.1 Overview

The renesas-ra port supports boards powered by Renesas Electronics's the flexible Renesas Advanced (RA) 32-bit microcontrollers (MCUs) family, industry-leading 32-bit MCUs with the Arm® Cortex®-M33, -M23 and -M4 processor cores and PSA certification.

The ported hardware functionalities are Pin, SPI, I2C, UART and RTC at this present. Other hardware functionalities such as Timer, PWM, USB, CAN and WDT are not implemented yet.

The following boards are officially supported.

- RA4M1-CLICKER

For the manual and other references for the board and RA Family MCU, please refer to the web page: RA4M1 CLICKER

- EK-RA6M2

For the manual and other references for the board and RA Family MCU, please refer to the web page: EK-RA6M2

The following boards are unofficially supported.

- EK-RA4M1

For the manual and other references for the board and RA Family MCU, please refer to the web page: EK-RA4M1

- EK-RA4W1

For the manual and other references for the board and RA Family MCU, please refer to the web page: EK-RA4W1

- EK-RA6M1

For the manual and other references for the board and RA Family MCU, please refer to the web page: EK-RA6M1

For more information, please visit the web page: https://github.com/renesas/micropython/wiki

15.2 MicroPython tutorial for Renesas RA port

This tutorial is intended to get you started using MicroPython on the Renesas RA port.

15.2.1 Getting started with MicroPython on the Renesas RA

This tutorial will guide you through setting up MicroPython, getting a prompt, using the hardware peripherals, using internal flash file system, reset and boot modes, and Factory reset the internal file system.

Requirements

You need a board. For the information of available boards, please refer to the general information about Renesas RA port: *General information about Renesas RA port*.

You need a USB-Serial conversion cable to connect the board and your PC. Please get a type with separate signal pins so that you can connect to the UART TX and RX pins on the board.

Flashing the MicroPython image with J-Link OB

The board has a builtin programmer interface called J-Link OB. Using the J-Link Software, you can flash the binary image “firmware.hex” of MicroPython on the board via J-Link OB.

You can download the J-Link Software and Documentation pack from https://www.segger.com/downloads/jlink/.

After installing J-Link Software, start J-Flash-Lite program. Then specify following device in Device select menu in J-Flash-Lite.

Board	Device
EK-RA4M1	R7FA4M1AB
EK-RA4W1	R7FA4W1AD2CNG
EK-RA6M1	R7FA6M1AD
EK-RA6M2	R7FA6M2AF
RA4M1 CLICKER	R7FA4M1AB

Select a firmware hex file in Data File of J-Link-Lite, and push Program Device button to flash the firmware.

Getting a prompt of MicroPython

Cross connect USB-Serial conversion cable RX/TX/GND pins to TX/RX/GND pins on the board.

Board	USB Serial RX	USB Serial TX
EK-RA4M1	P411	P410
EK-RA4W1	P205	P206
EK-RA6M1	P411	P410
EK-RA6M2	P411	P410
RA4M1 CLICKER	P401	P402

Access the MicroPython REPL (the Python prompt) via USB serial or UART with 115200 baud rate, 1 stop bit and no parity bit using your favorite terminal software, picocom on Linux or Tera Term on Windows. You can try on Linux:

```
$ picocom /dev/ttyACM0
```

You can see the MicroPython REPL prompt like below:

```
MicroPython v1.20.0 on 2023-04-27; EK-RA6M2 with RA6M2
Type "help()" for more information.
>>>
```

15.2.2 Using peripherals

For quick help information, please enter:

```
help()
```

You can access RA MCU's peripherals using MicroPython modules. To list supported modules, please enter:

```
help('modules')
```

Especially *machine* module and class *machine.Pin* are very important for using peripherals.

Using "from machine import Pin", Pin name is available corresponding to the RA MCU's pin name which are Pin.cpu.P000 and 'P000'. In addition, you can use 'LED1', 'LED2', 'SW1', and 'SW2' name if the board has these LEDs and switches.

LED blinking

As simple example, you can enter following program to blink LED1. Please enter key 4 times after the input of last time.sleep(1).

```
import time
from machine import Pin
led1 = Pin('LED1')
print(led1)
while True:
    led1.on()
    time.sleep(1)
    led1.off()
    time.sleep(1)
```

You can see the LED1 blinking per 1 second.

If you want to stop the program, please enter CTRL-C.

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
KeyboardInterrupt:
```

This message is displayed, and the program stops. The message means the program was interrupted at line 5 “while” statement.

Using print(led1), you can confirm that LED1 is assigned to Pin.cpu.P106 on the board.:

```
Pin(Pin.cpu.P106, mode=Pin.OUT, pull=Pin.PULL_NONE, drive=Pin.LOW_POWER)
```

So you can get the same result if Pin(Pin.cpu.P106) is specified instead of Pin(‘LED1’).

```
import time
from machine import Pin
led1 = Pin(Pin.cpu.P106)
print(led1)
while True:
    led1.on()
    time.sleep(1)
    led1.off()
    time.sleep(1)
```

15.2.3 Write a program in internal file system

Internal file system

The FAT file system is created and initialized in the RA MCU’s internal flash when the MicroPython starts at the first time on the board. The file system is mounted as “/flash”, so you can access this flash system and create a program file into the /flash directory.

As the factory setting, following size is allocated for the file system:

Board	File System Size
EK-RA4M1	36KB (36864B)
EK-RA4W1	64KB (65536B)
EK-RA6M1	64KB (65536B)
EK-RA6M2	128KB (131072B)
RA4M1 CLICKER	36KB (36864B)

As the factory setting, following 2 files are created in the file system:

- boot.py : executed first when the system starts
- main.py : executed after boot.py completes

See *Reset and Boot Sequence* for more information.

Write a program in the internal file system

You can write a program in main.py which is executed automatically when the MicroPython starts. For example, you can write LED blinking program like below:

```
import os
os.getcwd()
f = open('main.py', 'rw+')
print(f.read())
f.write('import time\n')
f.write('from machine import Pin\n')
f.write('led1 = Pin(Pin.cpu.P106)\n')
f.write('while True:\n')
f.write('    led1.on()\n')
f.write('    time.sleep(1)\n')
f.write('    led1.off()\n')
f.write('    time.sleep(1)\n')
f.close()
f = open('main.py', 'r')
print(f.read())
f.close()
```

Entering CTRL-D for software reset, the MicroPython reboots, displays following messages:

```
MPY: sync filesystems
MPY: soft reboot
```

and main.py is executed and LED1 blinks per 1 second. If you want to stop the program, please enter CTRL-C.

15.2.4 Reset and boot mode

Resetting the board

If something goes wrong, you can reset the board in two ways. The first is to press CTRL-D at the MicroPython prompt, which performs a soft reset. You will see a message something like

```
MPY: sync filesystems
MPY: soft reboot
MicroPython v1.18-293-g339aa09b8-dirty on 2022-03-26; RA6M2_EK with RA6M2
```

(continues on next page)

(continued from previous page)

```
Type "help()" for more information.
>>>
```

If that isn't working you can perform a hard reset (turn-it-off-and-on-again) by pressing the RESET button. This will end your session, disconnecting whatever program (PuTTY, screen, etc) that you used to connect to the board.

For more details, see *Reset and Boot Sequence*.

boot mode

There are 3 boot modes:

- normal boot mode
- safe boot mode
- factory filesystem boot mode

boot.py and main.py are executed on "normal boot mode". See *Boot Sequence*.

The other modes can be used to recover from *Soft Bricking (failure to boot)*:

boot.py and main.py are *NOT* executed on "safe boot mode".

The file system of internal flash is initialized and *all files are erased* on "factory filesystem boot mode".

For changing boot mode, please push the RESET button with pressing USER SW1 on the board:

- For normal boot mode, release the USER SW1 after LED1 flashes 4 times or more
- For safe boot mode, release the USER SW1 after LED1 flashes 2 times
- For factory file system boot mode, release the USER SW1 after LED1 flashes 3 times.

You have created the main.py which executes LED1 blinking in the previous part. If you change the boot mode to safe boot mode, the MicroPython starts without the execution of main.py.

15.2.5 Trouble Shooting

Flash file system

- MicroPython REPL prompt is not displayed.
  - Re-program the MicroPython image file.
  - Confirm serial port connection. The connection must be crossed that the board TXD pin is connected with USB-serial conversion's RXD signal pin.
  - If the prompt is not displayed suddenly, try to do factory file system boot mode as the final method. Please note that all files are removed forever.

15.3 Installing MicroPython

See the corresponding section of tutorial: *Getting started with MicroPython on the Renesas RA*. It also includes a troubleshooting subsection.

15.4 General board control

The MicroPython REPL is accessed via the USB serial port. Tab-completion is useful to find out what methods an object has. Paste mode (ctrl-E) is useful to paste a large slab of Python code into the REPL. Some of features are not implemented for Renesas RA boards yet, please refer to the tutorial for more details.

The *machine* module:

```
import machine

machine.freq()     # get the current frequency of the CPU
```

Following functions are supported:

```
machine.freq()
machine.reset()
machine.soft_reset()
machine.unique_id()
```

Following functions are not supported at the present:

```
machine.reset_cause()
machine.bootloader([value])
machine.disable_irq()
machine.enable_irq(state)
machine.freq([hz])
machine.idle()
machine.sleep()
machine.lightsleep()
machine.lightsleep([time_ms])
machine.deepsleep()
machine.deepsleep([time_ms])
machine.wake_reason()
machine.time_pulse_us(pin, pulse_level, timeout_us=1000000,/)
machine.bitstream(pin, encoding, timing, data, /)
machine.rng()
```

15.5 Delay and timing

Use the *time* module:

```
import time

time.sleep(1)           # sleep for 1 second
time.sleep_ms(500)      # sleep for 500 milliseconds
time.sleep_us(10)       # sleep for 10 microseconds
start = time.ticks_ms() # get value of millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start) # compute time difference
```

15.6 Timers

The RA MCU's system timer peripheral provides a global microsecond timebase and generates interrupts for it. The software timer is available currently and there are unlimited number of them (memory permitting). There is no need to specify the timer id (id=-1 is supported at the moment) as it will default to this.

Use the *machine.Timer* class:

```
from machine import Timer

tim = Timer(-1)
tim.init(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(1))
tim.init(period=2000, mode=Timer.PERIODIC, callback=lambda t: print(2))
```

Following functions are not supported at the present::
Timer(id) # hardware timer is not supported.

15.7 Pins and GPIO

Use the *machine.Pin* class:

```
from machine import Pin

p0 = Pin('P000', Pin.OUT)       # create output pin on P000
p0.on()                         # set pin to "on" (high) level
p0.off()                        # set pin to "off" (low) level
p0.value(1)                     # set pin to on/high

p2 = Pin(Pin.cpu.P002, Pin.IN)  # create input pin on P002
print(p2.value())               # get value, 0 or 1

p4 = Pin('P004', Pin.PULL_UP)       # enable internal pull-up register
p5 = Pin('P005', Pin.OUT, value=1)  # set pin high on creation
```

Pin id is available corresponding to the RA MCU's pin name which are Pin.cpu.P106 and 'P106'. The RA MCU has many feature's pins. However, there are some cases that pin feature is fixed or not connected by the board. Please confirm the board manual for the pin mapping.

The following *drive* keyword argument are available if the port drive capability of the Pin is supported by the MCU:

```
Pin.DRIVE_0: Low drive
Pin.DRIVE_1: Middle drive
Pin.DRIVE_2: Middle drive for I2C Fast-mode
Pin.DRIVE_3: High drive
```

The *alt* keyword argument is not supported.

The following functions are not supported:

```
Pin.irq(priority=)  # priority keyword argument is not supported
Pin.irq(wake=)      # wake keyword argument is not supported
Pin.irq(hard=)      # hard keyword argument is ignored because hardware interrupt is␣
↪used
Pin.mode()
Pin.pull()
Pin.drive()
```

15.8 UART (serial bus)

The RA MCU has some hardware UARTs called SCI (Serial Communication Interface). UART id is available corresponding to the RA MCU's SCI number which are UART(0) as SCI0 and UART(1) as SCI1.

See *machine.UART*.

```
from machine import UART

uart1 = UART(1, 115200)
uart1.write('hello')     # write 5 bytes
uart1.read(5)            # read up to 5 bytes
```

Available UARTs and pins on the board are fixed and follows. One of these UARTs is used for REPL.

EK-RA4M1	UART0(REPL)	UART1	UART2
tx	P411	P401	P302
rx	P410	P402	P301

EK-RA4W1	UART1	UART4(REPL)	UART9
tx	P213	P204	P109
rx	P212	P206	P110

EK-RA6M1	UART0(REPL)	UART2	UART8
tx	P411	P302	P105
rx	P410	P301	P104

EK-RA6M2	UART0(REPL)	UART7	UART9
tx	P411	P401	P602
rx	P410	P402	P601

RA4M1-CLICKER	UART0	UART1(REPL)
tx	P411	P401
rx	P410	P402

Following functions are not supported at the present:

```
UART.init(baudrate)    # now only 115200 is confirmed
UART.init(cts, rts)    # Pins are fixed.
UART.init(invert)
UART.init(tx,rx)       # Pins are fixed.
UART.init(txbuf)
UART.init(flow)
UART.irq(handler)
UART.irq(trigger=RX_ANY)
```

(continues on next page)

(continued from previous page)

```
UART.irq(priority)
UART.irq(wake=machine.IDLE)
```

15.9 Real time clock (RTC)

See *machine.RTC*

```
from machine import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 0, 1, 12, 48, 0)) # set a specific date and
                                             # time, eg. 2017/8/23 1:12:48
                                             # the day-of-week value is ignored
rtc.datetime() # get date and time
```

Following functions are not supported at the present:

```
RTC.init(datetime)
RTC.now()
RTC.deinit()
RTC.alarm()
RTC.alarm_left()
RTC.cancel()
RTC.irq()
```

15.10 ADC (analog to digital conversion)

See *machine.ADC*

```
from machine import ADC

adc = ADC('P000')     # create an ADC object acting on a pin
adc.read_u16()        # read a raw analog value in the range 0-65535
```

Pin id is available corresponding to the RA MCU's pin name which are 'P000' as AN000 (analog channel 000). The RA MCU has many analog channels. However, there are some cases that pin feature is fixed or not available by the board. Please confirm the MCU and board manual for the pin mapping.

Following functions are not supported at the present:

```
ADC.init()
ADC(sample_ns)
ADC(atten)
ADC.read_uv()
ADC.block()
```

15.11 SPI bus

The RA MCU has some hardware SPIs (Serial Peripheral Interface). SPI id is available corresponding to the RA MCU's SPI number which are SPI(0) as SPI0 and SPI(1) as SPI1. If with no additional parameters, machine.SoftSPI() is called.

See *machine.SPI*.

```
from machine import SPI, Pin

spi = SPI(0, baudrate=500000)
cs = Pin.cpu.P103
cs(0)
spi.write(b"12345678")
cs(1)
```

Available SPIs and pins on the board are fixed and follows.

EK-RA4M1	SPI0
sck	P102
mosi	P101
miso	P100
cs	P206

EK-RA4W1	SPI0
sck	P102
mosi	P101
miso	P100
cs	P103

EK-RA6M1	SPI0
sck	P102
mosi	P101
miso	P100
cs	P103

EK-RA6M2	SPI0	SPI1
sck	P102	P702
mosi	P101	P701
miso	P100	P700
cs	P103	P703

RA4M1-CLICKER	SPI0
sck	P102
mosi	P101
miso	P100
cs	P103

Following functions are not supported at the present:

```
SPI.init(firstbit)     # now fixed with SPI.LSB
SPI.init(baudrate)     # now confirmed only 500000
```

15.12 I2C bus

The RA MCU has some hardware IIC (Inter-Integrated Circuit Bus). I2C id is available corresponding to the RA MCU's I2C number which are I2C(0) as IIC0 and I2C(1) as IIC1. If with no additional parameters, machine.SoftI2C() is called.

See *machine.I2C*

```
from machine import I2C

i2c = I2C(0)

i2c.scan()          # returns list of slave addresses
i2c.readfrom_mem(0x50, 0x10, 2, addrsize=16)  # read 2 bytes from slave 0x50, slave␣
↪memory 0x10
```

Available I2Cs and pins on the board are fixed and follows.

EK-RA4M1	•
scl	not supported
sda	not supported

EK-RA4W1	I2C0
scl	P204
sda	P407

EK-RA6M1	I2C0
scl	P400
sda	P401

EK-RA6M2	I2C2
scl	P512
sda	P511

RA4M1-CLICKER	I2C1
scl	P205
sda	P206

Following functions are not supported at the present:

```
I2C.init(freq)     # now confirmed only 400000
I2C.deinit()
I2C.start()
I2C.stop()
```

15.13 PWM (pulse width modulation)

PWM is not supported.

15.14 WDT (Watchdog timer)

WDT is not supported.

15.15 SDCard

The frozen sdcard driver (drivers/sdcard/sdcard.py) is available by connecting microSD card device to hardware SPI0 pins.:

```
from machine import Pin, SPI
import os, vfs, sdcard

spi = SPI(0, baudrate=500000)
cs = Pin.cpu.P103
sd = sdcard.SDCard(spi, cs)
vfs.mount(sd, '/sd')
os.listdir('/')
os.chdir('/sd')
vfs.umount('/sd')
```

15.16 OneWire driver

The OneWire driver is implemented in software and works on all pins:

```
from machine import Pin
import onewire

ow = onewire.OneWire(Pin(P012)) # create a OneWire bus on P012
ow.scan()               # return a list of devices on the bus
ow.reset()              # reset the bus
ow.readbyte()           # read a byte
ow.writebyte(0x12)      # write a byte on the bus
ow.write('123')         # write bytes on the bus
ow.select_rom(b'12345678') # select a specific device by its ROM code
```

There is a specific driver for DS18S20 and DS18B20 devices:

```
import time, ds18x20
ds = ds18x20.DS18X20(ow)
roms = ds.scan()
ds.convert_temp()
time.sleep_ms(750)
for rom in roms:
    print(ds.read_temp(rom))
```

Be sure to put a 4.7k pull-up resistor on the data line. Note that the `convert_temp()` method must be called each time you want to sample the temperature.

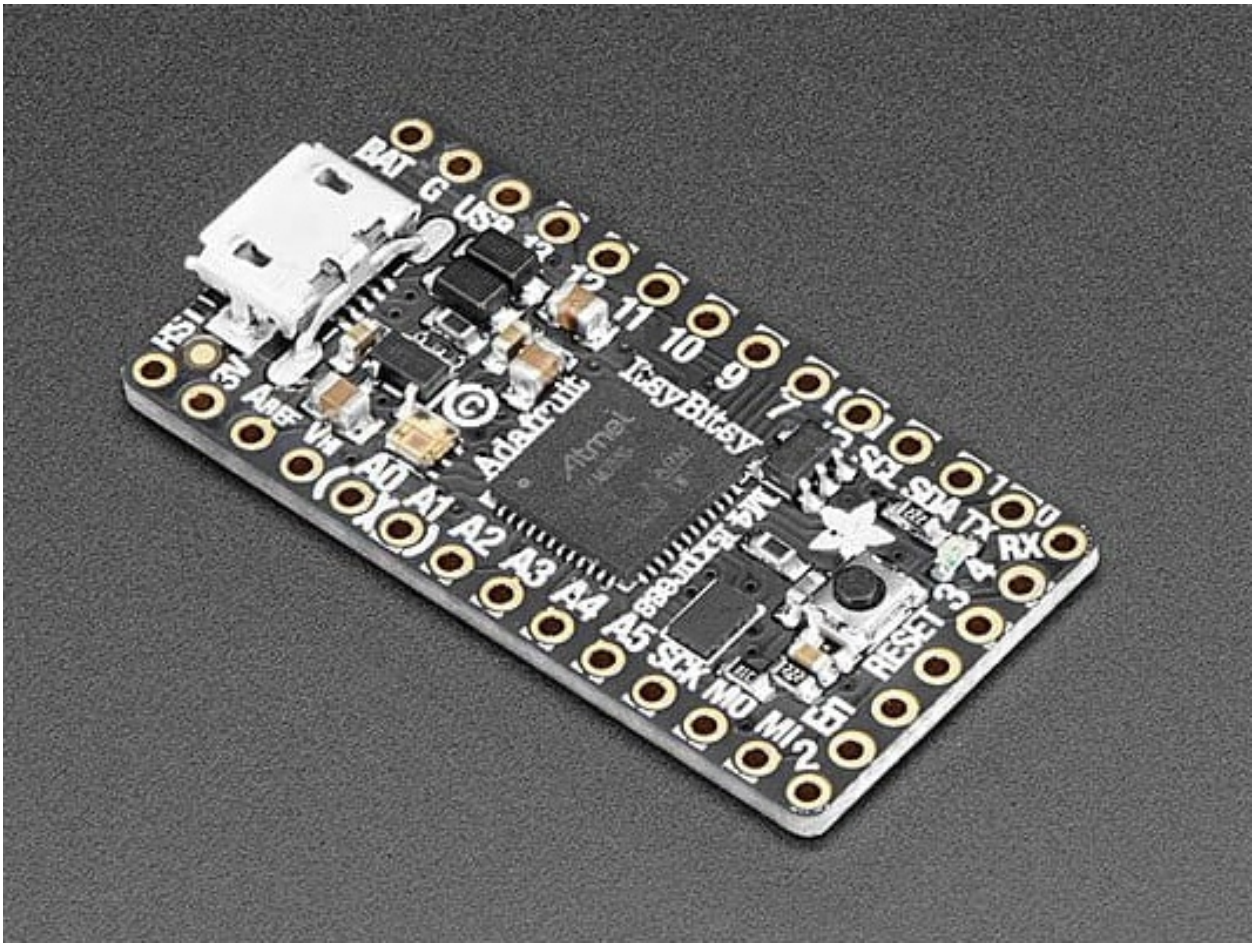
15.17 NeoPixel and APA106 driver

NeoPixel is not supported currently.

CHAPTER

SIXTEEN

QUICK REFERENCE FOR THE SAMD21/SAMD51 FAMILY



The Adafruit ItsyBitsy M4 Express board.

Below is a quick reference for SAMD21/SAMD51-based boards. If it is your first time working with this board it may be useful to get an overview of the microcontroller:

16.1 General information about the SAMD port

The SAMD21/SAMD51 MCU family is a high performance family of devices made by MicroChip. The SAMD21 devices are based on an ARM M0+ core, the SAMD51 device on ARM Cortex M4 core. They provide many on-chip I/O units for building small to medium sized devices.

16.1.1 Multitude of boards

There is a multitude of modules and boards from different sources which carry an SAMD21/SAMD51 chip. MicroPython aims to provide a generic port which runs on as many boards/modules as possible, but there may be limitations. The Adafruit ItsyBitsy M0 Express, Adafruit Feather M4 Express and the Adafruit ItsyBitsy M4 Express development boards are taken as reference for the port (for example, testing is performed on them). For any board you are using please make sure you have a data sheet, schematics and other reference materials so you can look up any board-specific functions.

The following boards are at the moment supported by the port:

- ADAFRUIT FEATHER M0 EXPRESS
- ADAFRUIT FEATHER M4 EXPRESS
- ADAFRUIT ITSYBITSY M0 EXPRESS
- ADAFRUIT ITSYBITSY M4 EXPRESS
- ADAFRUIT TRINKET M0
- MINISAM M4
- SAMD21 XPLAINED PRO
- SEEED WIO TERMINAL
- SEEED XIAO

To make a generic SAMD port and support as many boards as possible the following design and implementation decision were made:

- GPIO pin numbering is based on the board numbering. Please have the manual/pin diagram of your board at hand to find correspondence between your board pins and actual SAMD21/SAMD51 pins. For the boards listed above, the relation between the board pin number and the GPIO number can be found at *Pinout for the SAMD machine modules*.
- The pins that can be used by MicroPython are limited to those listed in the board definition files.

16.1.2 Technical specifications and SoC data sheets

The data sheets and other reference material for SAMD21/SAMD51 chip are available from the vendor site: https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/32-bit-mcus/sam-32-bit-mcus They are the primary reference for the chip technical specifications, capabilities, operating modes, internal functioning, etc.

For your convenience, a few technical specifications are provided below:

SAMD21:

- Architecture: ARM Cortex M0+
- CPU frequency: up to 48MHz
- Total RAM available: up to 32 kB (see table)
- Internal FlashROM: up to 256 kB Some boards provide additional external SPI flash.

- GPIO: up to 52 (GPIOs are multiplexed with other functions, including external FlashROM, UART, etc.)
- UART: up to 6 serial devices, which can used for UART, SPI or I2C.
- I2S: 1 I2S interfaces
- ADC: One 12-bit SAR ADC converter with 16 channels.
- Programming: using BootROM bootloader from USB.

SAMD51:

- Architecture: ARM Cortex M4
- CPU frequency: up to 120MHz
- Total RAM available: up to 256 kB
- Internal FlashROM: up to 1 MB Some boards provide additional external SPI flash.
- GPIO: up to 99 (GPIOs are multiplexed with other functions, including external FlashROM, UART, etc.)
- UART: up to 8 serial devices, which can used for UART, SPI or I2C.
- I2S: 1 I2S interfaces
- ADC: Dual 12-bit SAR ADC converter with 16 channels.
- Programming: using BootROM bootloader from USB.

For more information see the excellent SAMD21/SAMD51 data sheets or reference manuals.

At the moment, the SAM21 port of MicroPython requires 256kB flash, of which 64kB is used for a small file system. The SAM51 port requires 512 kB of flash, of which all flash beyond 384kB is used for a file system. Some boards have additional flash memory, which can be used as additional file space.

16.2 Getting started with MicroPython on the SAMD

Using MicroPython is a great way to get the most of your SAMD board. And vice versa, the SAMD chip is a great platform for using MicroPython. This tutorial will guide you through setting up MicroPython, getting a prompt, using the hardware peripherals, and controlling some external components.

Let's get started!

16.2.1 Requirements

The first thing you need is a board with an SAMD chip. The MicroPython software supports the SAMD chip itself and any board should work. The main characteristic of a board is how the GPIO pins are connected to the outside world, and whether it includes a built-in USB-serial converter to make the UART available to your PC.

Names of pins will be given in this tutorial using the board names ("D0") and it should be straightforward to find which pin this corresponds to on your particular board.

16.2.2 Powering the board

If your board has a USB connector on it then most likely it is powered through this when connected to your PC. Otherwise you will need to power it directly. Please refer to the documentation for your board for further details.

16.2.3 Getting the firmware

Firmware versions are provided at the MicroPython download page. You can download the most recent MicroPython firmware .uf2 file to load onto your SAMD device. From that download page you have two main choices:

- stable firmware builds
- daily firmware builds

If you are just starting with MicroPython, the best bet is to go for the stable firmware builds. If you are an advanced, experienced MicroPython SAMD user who would like to follow development closely and help with testing new features, there are daily builds. They run through the same rigid internal test sequence as the stable builds, but the firmware features may change.

16.2.4 Deploying the firmware

Once you have the MicroPython firmware you need to load it onto your SAMD device. Most of the boards support the following easy procedure for firmware upload.

- Connect the board by an USB-A/USB-Micro cable to your PC.
- Push the reset button twice. Then, a drive should pop up at the PC's file manager. Many boards have a on-board RGB LED, which first lights up red, and then changes it's colour to green. The timing of the double push is sometimes tricky. So you may have to try to get the proper rhythm.
- Copy the .uf2 file to that newly opened drive. A LED on the board may flash a while. When the copy is finished, the board drive will disappear and the RGB led will turn from green to the initial state.

16.2.5 Serial prompt

Once you have the firmware on the device you can access the REPL (Python prompt) over USB.

From there you can follow the SAMD tutorial.

16.2.6 Troubleshooting installation problems

If you experience problems during flashing or with running firmware immediately after it, here are some troubleshooting recommendations:

- Be aware of and try to exclude hardware problems. There are two common problems: bad power source quality, and worn-out/defective Flash ROM. Speaking of power source, not just raw amperage is important, but also low ripple and noise/EMI in general. The most reliable and convenient power source is a USB port.

16.3 Pinout for the SAMD machine modules

The assignment of device functions to pins is very flexible. The same function may be used at different pins. The representation of the assignment choices are given by a table, which is a subset of the MCU's Pin MUX table and is specific to each board, as the available pin set varies. The structure of the table is the same for each board, but the set of rows is different.

16.3.1 Adafruit ItsyBitsy M0 Express pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
2	PA02	A0	2	0	•	•	•	•
40	PB08	A1	8	2	•	4/0	4/0	•
41	PB09	A2	9	3	•	4/1	4/1	•
4	PA04	A3	4	4	•	0/0	0/0	•
5	PA05	A4	5	5	•	0/1	0/1	•
34	PB02	A5	2	10	•	5/0	6/0	•
11	PA11	D0	11	19	0/3	2/3	1/1	0/3
10	PA10	D1	10	18	0/2	2/2	1/0	0/2
14	PA14	D2	14	•	2/2	4/2	3/0	0/4
9	PA09	D3	9	17	0/1	2/1	0/1	1/3
8	PA08	D4	•	16	0/0	2/0	0/0	1/2
15	PA15	D5	15	•	2/3	4/3	3/1	0/5
21	PA21	D7	5	•	5/3	3/3	7/1	0/7
7	PA07	D9	7	7	•	0/3	1/1	•
18	PA18	D10	2	•	1/2	3/2	3/0	0/2
16	PA16	D11	0	•	1/0	3/0	2/0	0/6
19	PA19	D12	3	•	1/3	3/3	3/1	0/3
17	PA17	D13	1	•	1/1	3/1	2/1	0/7

continues on next page

Table 1 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
0	PA00	DOT-STAR_CLI	0	•	•	1/0	2/0	•
1	PA01	DOT-STAR_DA	1	•	•	1/1	2/1	•
27	PA27	FLASH_CS	15	•	•	•	•	•
35	PB03	FLASH_M	3	11	•	5/1	6/1	•
54	PB22	FLASH_M	6	•	•	5/2	7/0	•
55	PB23	FLASH_SC	7	•	•	5/3	7/1	•
11	PA11	RX	11	19	0/3	2/3	1/1	0/3
10	PA10	TX	10	18	0/2	2/2	1/0	0/2
12	PA12	MISO	12	•	2/0	4/0	2/0	0/6
42	PA12	MOSI	10	•	•	4/2	5/0	0/4
43	PA13	SCK	11	•	•	4/3	5/1	0/5
23	PA23	SCL	7	•	3/1	5/1	4/1	0/5
22	PA22	SDA	6	•	3/0	5/0	4/0	0/4
30	PA30	SWCLK	10	•	•	1/2	1/0	•
31	PA31	SWDIO	11	•	•	1/3	1/1	•
24	PA24	USB_DM	12	•	3/2	5/2	5/0	1/2
25	PA25	USB_DP	13	•	3/3	5/3	5/1	1/3
3	PA03		3	1	•	•	•	•
6	PA06		6	6	•	0/2	1/0	•
13	PA13		13	•	2/1	4/1	2/0	0/7
20	PA20		4	•	5/2	3/2	7/0	0/4
28	PA28		8	•	•	•	•	•

Description of the columns:

- *Pin* - The number that is expected at `machine.Pin(n)`, if the pin is given as a number.
- *GPIO* - The GPIO name, which can be used as argument to `machine.Pin("name")`.
- *Pin Name* - The boards name, which can be used as argument to `machine.Pin("name")`.
- *IRQ* - The IRQ number assigned to that GPIO, used internally by `Pin.irq()`. When using `Pin.irq()`, different pins must use different IRQs
- *ADC* - The ADC channel assigned to the pin. When using ADC, different pins must not use the same ADC channel.
- *Serial* - Two columns with optional Serial signal assignments. Both may be used. The cell content is device #/pad #. The pad # is the respective internal signal of that serial device. Details below.
- *TCC/TC* - Two columns with assignments of the TCC modules for PWM. The cell content is device #/output #. For PWM, devices 0, 1, and 2 are used. The TC device pair 3/4 is used for `ticks_us()`.

SAMD21 UART assignments

The UART devices and signals must be chosen according to the following rules:

- The TX signal must be at a Pin with pad numbers 2 or 0, like Pin D1 with serial device 0 or 2.
- The RX pin may be assigned to one of the other pads.

Examples for Adafruit ItsyBitsy M0 Express:

- uart 0 at pins D0/D1 This is the default UART at the RX/TX labelled pins
- uart 1 at pins D12/D10
- uart 2 at pins D0/D1
- uart 3 at pins D11/D13
- uart 4 at pins D2/D5
- uart 5 at pins SCL/SDA

or other combinations. For hardware flow control, tx must be at pad 0, rx at pad 1, rts at pad 2 and cts at pad 3. This applies for instance to UART 3 or UART 1 at the pins D13/D11/D10/D12 for rx/tx/rts/cts.

SAMD21 I2C assignments

The I2C devices and signals must be chosen according to the following rules:

- The SDA signal must be at a Pin with pad numbers 0.
- The SCL signal must be at a Pin with pad numbers 1.

Examples for Adafruit ItsyBitsy M0 Express:

- I2C 0 at Pin A3/A4
- I2C 1 at pins D11/D13
- I2C 2 at the pins D4/D3
- I2C 3 at the pins SDA/SCL This is the default I2C device at the SDA/SCl labelled pin
- I2C 4 at the pins A1/A2
- I2C 5 at the pins SDA/SCL,

or other combinations.

SAMD21 SPI assignments

The SPI devices and signals must be chosen according to the following rules:

- The following pad number pairs are suitable for MOSI/SCK: 0/1, 2/3, 3/1, and 0/3.
- The MISO signal must be at a Pin with a different pad number than MOSI or SCK.

Examples for Adafruit ItsyBitsy M0 Express:

- SPI 0 at pins D0/D4/D1
- SPI 1 at pins D11/D12/D13
- SPI 2 at pins D0/D4/D1
- SPI 3 at pins D11/D12/D13
- SPI 2 at Pin MOSI/MISO/SCK This is the default SPI device at the MOSI/MISO/SCK labelled pins.

or other combinations.

SAMD21 PWM assignments

The TCC/TC device numbers 0, 1 and 2 can be used for PWM. Device 0 has four channels, device 1 and 2 have two channels. So in total 3 different PWM frequencies can be used, and 8 different duty cycle values.

The DAC output for the Adafruit ItsyBitsy M0 Express board is available at the pin A0.

16.3.2 Adafruit ItsyBitsy M4 Express pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
2	PA02	A0	2	0	•	•	•	•	•	•
5	PA05	A1	5	5	•	•	0/1	0/1	•	•
40	PB08	A2	8	2	0	•	4/0	4/0	•	•
41	PB09	A3	9	3	1	•	4/1	4/1	•	•
4	PA04	A4	4	4	•	•	0/0	0/0	•	•
6	PA06	A5	6	6	•	•	0/2	1/0	•	•
16	PA16	D0	0	•	•	1/0	3/1	2/0	1/0	0/4
17	PA17	D1	1	•	•	1/1	3/0	2/1	1/1	0/5

continues on next page

Table 2 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
7	PA07	D2	7	7	•	•	0/3	1/1	•	•
54	PB22	D3	22	•	•	1/2	5/2	7/0	•	•
14	PA14	D4	14	•	•	2/2	4/2	3/0	2/0	1/2
15	PA15	D5	15	•	•	2/3	4/3	3/1	2/1	1/3
18	PA18	D7	2	•	•	1/2	3/2	3/0	1/2	0/6
19	PA19	D9	3	•	•	1/3	3/3	3/1	1/3	0/7
20	PA20	D10	4	•	•	5/2	3/2	7/0	1/4	0/0
21	PA21	D11	5	•	•	5/3	3/3	7/1	1/5	0/1
23	PA23	D12	7	•	•	3/1	5/0	4/1	1/7	0/3
22	PA22	D13	6	•	•	3/0	5/1	4/0	1/6	0/2
34	PB02	DOT-STAR_C	2	14	•	•	5/0	6/0	2/2	•
35	PB03	DOT-STAR_D	9	15	•	•	5/1	6/1	•	•
16	PA16	RX	0	•	•	1/0	3/1	2/0	1/0	0/4
17	PA17	TX	1	•	•	1/1	3/0	2/1	1/1	0/5
55	PB23	MISO	7	•	•	1/3	5/3	7/1	•	•
0	PA00	MOSI	0	•	•	•	1/0	2/0	•	•
43	PB11	QSPI_CS	12	•	•	•	4/3	5/1	0/5	1/1
8	PA08	QSPI_D(	•	8	2	0/0	2/1	0/0	0/0	1/4
9	PA09	QSPI_D1	9	9	3	0/1	2/0	0/1	0/1	1/5
10	PA10	QSPI_D2	10	10	•	0/2	2/2	1/0	0/2	1/6
11	PA11	QSPI_D3	11	11	•	0/3	2/3	1/1	0/3	1/7

continues on next page

Table 2 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
42	PB10	QSPI_SC	10	•	•	•	4/2	5/0	0/4	1/0
1	PA01	SCK	1	•	•	•	1/1	2/1	•	•
13	PA13	SCL	13	•	•	2/1	4/0	2/1	0/7	1/3
12	PA12	SDA	12	•	•	2/0	4/1	2/0	0/6	1/2
30	PA30	SWCLK	14	•	•	7/2	1/2	6/0	2/0	•
31	PA31	SWDIO	15	•	•	7/3	1/3	6/1	2/1	•
24	PA24	USB_DM	8	•	•	3/2	5/2	5/0	2/2	•
25	PA25	USB_DP	9	•	•	3/3	5/3	5/1	•	•
3	PA03	•	3	10	•	•	•	•	•	•
27	PA27	•	11	•	•	•	•	•	•	•

Description of the columns:

- *Pin* - The number that is expected at `machine.Pin(n)`, if the pin is given as a number.
- *GPIO* - The GPIO name, which can be used as argument to `machine.Pin("name")`.
- *Pin Name* - The boards name, which can be used as argument to `machine.Pin("name")`.
- *IRQ* - The IRQ number assigned to that GPIO, used internally by `Pin.irq()`. When using `Pin.irq()`, different pins must use different IRQs
- *ADC* - The ADC0/1 channel assigned to the pin. When using ADC, different pins must not use the same ADC device and channel.
- *Serial* - Two columns with optional Serial signal assignments. Both may be used. The cell content is device #/pad #. The pad # is the respective internal signal of that serial device. Details below.
- *TC* - These device are currently not assigned to Pin. the TC device pair 0/1 is used for `ticks_us()`.
- *PWM* - Two columns with assignments of the TCC modules for PWM The cell content is device #/output #. Details below.

SAMD51 UART assignments

The UART devices and signals must be chosen according to the following rules:

- The TX signal must be at a Pin with pad numbers 0, like Pin D1 with serial device 3.
- The RX pin may be assigned to one of the other pads.

Examples for Adafruit ItsyBitsy 4 Express:

- uart 0 at pins A4/A1
- uart 1 at pins D1/D0 This is the default UART at the RX/TX labelled pins
- uart 2 at pins SCL/SDA This is the default I2C device at the SDA/SCl labelled pin
- uart 3 at pins D0/D1
- uart 4 at pins SDA/SCL
- uart 5 at pins D12/D13

or other combinations. For hardware flow control, tx must be at pad 0, rx at pad 1, rts at pad 2 and cts at pad 3. This applies for instance to UART 5 at the pins D12/D13/D10/D11 for rx/tx/rts/cts.

SAMD51 I2C assignments

The I2C devices and signals must be chosen according to the following rules:

- The SDA signal must be at a Pin with pad numbers 0.
- The SCL signal must be at a Pin with pad numbers 1.

Examples for Adafruit ItsyBitsy M4 Express:

- I2C 0 at pins A3/A4
- I2C 1 at pins D0/D1
- I2C 2 at the pins SDA/SCL
- I2C 3 at the pins D1/D0
- I2C 4 at the pins A2/A3
- I2C 5 at the pins D12/D13

or other combinations.

SAMD51 SPI assignments

The SPI devices and signals must be chosen according to the following rules:

- The following pad number pairs are suitable for MOSI/SCK: 0/1 and 3/1.
- The MISO signal must be at a Pin with a different pad number than MOSI or SCK.

Examples for Adafruit ItsyBitsy M4 Express:

- SPI 1 at Pin MOSI/MISO/SCK This is the default SPI device at the MOSI/MISO/SCK labelled pins.
- SPI 3 at pins D13/D11/D12
- SPI 5 at pins D12/D3/D13

or other combinations.

SAMD51 PWM assignments

The TCC/PWM device numbers 0 through 4 can be used for PWM. Device 0 has six channels, device 1 has four channels, device 2 has three channels and devices 3 and 4 have two channels. So in total up to 5 different PWM frequencies can be used, and up to 17 different duty cycle values. Note that these numbers do not apply to every board.

The DAC outputs for the Adafruit ItsyBitsy M4 Express board are available at the pins A0 and A1.

16.3.3 Adafruit Feather M4 Express pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
2	PA02	A0	2	0	•	•	•	•	•	•
5	PA05	A1	5	5	•	•	0/1	0/1	•	•
40	PB08	A2	8	2	0	•	4/0	4/0	•	•
41	PB09	A3	9	3	1	•	4/1	4/1	•	•
4	PA04	A4	4	4	•	•	0/0	0/0	•	•
38	PB06	A5	6	•	8	•	•	•	•	•
49	PB17	D0	1	•	•	5/1	•	6/1	3/1	0/5
48	PB16	D1	0	•	•	5/0	•	6/0	3/0	0/4
14	PA14	D4	14	•	•	2/2	4/2	3/0	2/0	1/2
16	PA16	D5	0	•	•	1/0	3/1	2/0	1/0	0/4
18	PA18	D6	2	•	•	1/2	3/2	3/0	1/2	0/6
19	PA19	D9	3	•	•	1/3	3/3	3/1	1/3	0/7
3	PA03	AREF	3	10	•	•	•	•	•	•
20	PA20	D10	4	•	•	5/2	3/2	7/0	1/4	0/0
21	PA21	D11	5	•	•	5/3	3/3	7/1	1/5	0/1
22	PA22	D12	6	•	•	3/0	5/1	4/0	1/6	0/2

continues on next page

Table 3 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
23	PA23	D13	7	•	•	3/1	5/0	4/1	1/7	0/3
49	PB17	RX	1	•	•	5/1	•	6/1	3/1	0/5
48	PB16	TX	0	•	•	5/0	•	6/0	3/0	0/4
54	PB22	MISO	22	•	•	1/2	5/2	7/0	•	•
55	PB23	MOSI	7	•	•	1/3	5/3	7/1	•	•
35	PB03	NEOPIX	9	15	•	•	5/1	6/1	•	•
43	PB11	QSPI_CS	12	•	•	•	4/3	5/1	0/5	1/1
8	PA08	QSPI_D(	•	8	2	0/0	2/1	0/0	0/0	1/4
9	PA09	QSPI_D1	9	9	3	0/1	2/0	0/1	0/1	1/5
10	PA10	QSPI_D2	10	10	•	0/2	2/2	1/0	0/2	1/6
11	PA11	QSPI_D3	11	11	•	0/3	2/3	1/1	0/3	1/7
42	PB10	QSPI_SC	10	•	•	•	4/2	5/0	0/4	1/0
17	PA17	SCK	1	•	•	1/1	3/0	2/1	1/1	0/5
13	PA13	SCL	13	•	•	2/1	4/0	2/1	0/7	1/3
12	PA12	SDA	12	•	•	2/0	4/1	2/0	0/6	1/2
30	PA30	SWCLK	14	•	•	7/2	1/2	6/0	2/0	•
31	PA31	SWDIO	15	•	•	7/3	1/3	6/1	2/1	•
24	PA24	USB_DM	8	•	•	3/2	5/2	5/0	2/2	•
25	PA25	USB_DP	9	•	•	3/3	5/3	5/1	•	•
33	PB01	VDIV	1	13	•	•	5/3	7/1	•	•
0	PA00	•	0	•	•	•	1/0	2/0	•	•

continues on next page

Table 3 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
1	PA01	.	1	.	.	.	1/1	2/1	.	.
6	PA06	.	6	6	.	.	0/2	1/0	.	.
7	PA07	.	7	7	.	.	0/3	1/1	.	.
15	PA15	.	15	.	.	2/3	4/3	3/1	2/1	1/3
27	PA27	.	11	.	.	.	.	.	.	.
32	PB00	.	9	12	.	.	5/2	7/0	.	.
34	PB02	.	2	14	.	.	5/0	6/0	2/2	.
36	PB04	.	4	.	6	.	.	.	.	.
37	PB05	.	5	.	7	.	.	.	.	.
39	PB07	.	7	.	9	.	.	.	.	.
44	PB12	.	12	.	.	4/0	.	4/0	3/0	0/0
45	PB13	.	13	.	.	4/1	.	4/1	3/1	0/1
46	PB14	.	14	.	.	4/2	.	5/0	4/0	0/2
47	PB15	.	15	.	.	4/3	.	5/1	4/1	0/3
62	PB30	.	14	.	.	7/0	5/1	0/0	4/0	0/6
63	PB31	.	15	.	.	7/1	5/0	0/1	4/1	0/7

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M4 Express *Adafruit ItsyBitsy M4 Express pin assignment table*.

The default devices at the board are:

- UART 5 at pins D0/D1, labelled RX/TX
- I2C 2 at pins PA12/PA13, labelled SDA/SCL
- SPI 1 at pins PA23/PA22/PA17, labelled MOSI, MISO and SCK
- DAC output on pins PA02 and PA05, labelled A0 and A1

16.3.4 Adafruit Metro M4 Airlift pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
2	PA02	A0	2	0	•	•	•	•	•	•
5	PA05	A1	5	5	•	•	0/1	0/1	•	•
6	PA06	A2	6	6	•	•	0/2	1/0	•	•
32	PB00	A3	9	12	•	•	5/2	7/0	•	•
40	PB08	A4	8	2	0	•	4/0	4/0	•	•
41	PB09	A5	9	3	1	•	4/1	4/1	•	•
23	PA23	D0	7	•	•	3/1	5/0	4/1	1/7	0/3
22	PA22	D1	6	•	•	3/0	5/1	4/0	1/6	0/2
49	PB17	D2	1	•	•	5/1	•	6/1	3/1	0/5
48	PB16	D3	0	•	•	5/0	•	6/0	3/0	0/4
45	PB13	D4	13	•	•	4/1	•	4/1	3/1	0/1
46	PB14	D5	14	•	•	4/2	•	5/0	4/0	0/2
47	PB15	D6	15	•	•	4/3	•	5/1	4/1	0/3
44	PB12	D7	12	•	•	4/0	•	4/0	3/0	0/0
21	PA21	D8	5	•	•	5/3	3/3	7/1	1/5	0/1
20	PA20	D9	4	•	•	5/2	3/2	7/0	1/4	0/0
3	PA03	AREF	3	10	•	•	•	•	•	•
18	PA18	D10	2	•	•	1/2	3/2	3/0	1/2	0/6
19	PA19	D11	3	•	•	1/3	3/3	3/1	1/3	0/7
16	PA16	D13	0	•	•	1/0	3/1	2/0	1/0	0/4

continues on next page

Table 4 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
36	PB04	ESP_BU	4	•	6	•	•	•	•	•
15	PA15	ESP_CS	15	•	•	2/3	4/3	3/1	2/1	1/3
33	PB01	ESP_GP	1	13	•	•	5/3	7/1	•	•
37	PB05	ESP_RE	5	•	7	•	•	•	•	•
55	PB23	ESP_RT	7	•	•	1/3	5/3	7/1	•	•
7	PA07	ESP_RX	7	7	•	•	0/3	1/1	•	•
4	PA04	ESP_TX	4	4	•	•	0/0	0/0	•	•
43	PB11	FLASH_	12	•	•	•	4/3	5/1	0/5	1/1
11	PA11	FLASH_	11	11	•	0/3	2/3	1/1	0/3	1/7
9	PA09	FLASH_	9	9	3	0/1	2/0	0/1	0/1	1/5
8	PA08	FLASH_	•	8	2	0/0	2/1	0/0	0/0	1/4
42	PB10	FLASH_	10	•	•	•	4/2	5/0	0/4	1/0
10	PA10	FLASH_	10	10	•	0/2	2/2	1/0	0/2	1/6
23	PA23	RX	7	•	•	3/1	5/0	4/1	1/7	0/3
22	PA22	TX	6	•	•	3/0	5/1	4/0	1/6	0/2
14	PA14	MISO	14	•	•	2/2	4/2	3/0	2/0	1/2
12	PA12	MOSI	12	•	•	2/0	4/1	2/0	0/6	1/2
54	PB22	NEOPIX	22	•	•	1/2	5/2	7/0	•	•
38	PB06	RXLED	6	•	8	•	•	•	•	•
13	PA13	SCK	13	•	•	2/1	4/0	2/1	0/7	1/3
35	PB03	SCL	9	15	•	•	5/1	6/1	•	•

continues on next page

Table 4 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
34	PB02	SDA	2	14	•	•	5/0	6/0	2/2	•
30	PA30	SWCLK	14	•	•	7/2	1/2	6/0	2/0	•
31	PA31	SWDIO	15	•	•	7/3	1/3	6/1	2/1	•
62	PB30	SWO	14	•	•	7/0	5/1	0/0	4/0	0/6
39	PB07	TXLED	7	•	9	•	•	•	•	•
24	PA24	USB_DM	8	•	•	3/2	5/2	5/0	2/2	•
25	PA25	USB_DP	9	•	•	3/3	5/3	5/1	•	•
17	PA17	USB_HC	1	•	•	1/1	3/0	2/1	1/1	0/5
0	PA00	•	0	•	•	•	1/0	2/0	•	•
1	PA01	•	1	•	•	•	1/1	2/1	•	•
27	PA27	•	11	•	•	•	•	•	•	•
63	PB31	•	15	•	•	7/1	5/0	0/1	4/1	0/7

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M4 Express *Adafruit ItsyBitsy M4 Express pin assignment table*.

The default devices at the board are:

- UART 3 at pins PA23/PA22, labelled D0/D1 resp. RX/TX
- I2C 5 at pins PB02/PB03, labelled SDA/SCL
- SPI 4 at pins PA12/PA14/PA13, labelled MOSI, MISO and SCK
- DAC output on pins PA02 and PA05, labelled A0 and A1

16.3.5 SEEED XIAO pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
2	PA02	A0/D0	2	0	•	•	•	•
4	PA04	A1/D1	4	4	•	0/0	0/0	•
10	PA10	A2/D2	10	18	0/2	2/2	1/0	0/2
11	PA11	A3/D3	11	19	0/3	2/3	1/1	0/3
8	PA08	A4/D4	•	16	0/0	2/0	0/0	1/2
9	PA09	A5/D5	9	17	0/1	2/1	0/1	1/3
40	PB08	A6/D6	8	2	•	4/0	4/0	•
41	PB09	A7/D7	9	3	•	4/1	4/1	•
7	PA07	A8/D8	7	7	•	0/3	1/1	•
5	PA05	A9/D9	5	5	•	0/1	0/1	•
6	PA06	A10/D10	6	6	•	0/2	1/0	•
28	PA28	D12 *	8	•	•	•	•	•
17	PA17	D13 *	1	•	1/1	3/1	2/1	0/7
16	PA16	D14 *	0	•	1/0	3/0	2/0	0/6
15	PA15	D15 *	15	•	2/3	4/3	3/1	0/5
14	PA14	D16 *	14	•	2/2	4/2	3/0	0/4
13	PA13	D17 *	13	•	2/1	4/1	2/0	0/7
12	PA12	D18 *	12	•	2/0	4/0	2/0	0/6
19	PA19	D19 *	3	•	1/3	3/3	3/1	0/3
20	PA20	D20 *	4	•	5/2	3/2	7/0	0/4
21	PA21	D21 *	5	•	5/3	3/3	7/1	0/7
42	PB10	D22 *	10	•	•	4/2	5/0	0/4

continues on next page

Table 5 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
43	PB11	D23 *	11	•	•	4/3	5/1	0/5
55	PB23	D24 *	7	•	•	5/3	7/1	•
23	PA23	D25 *	7	•	3/1	5/1	4/1	0/5
22	PA22	D26 *	6	•	3/0	5/0	4/0	0/4
18	PA18	D27 *	2	•	1/2	3/2	3/0	0/2
17	PA17	SCL1 *	1	•	1/1	3/1	2/1	0/7
16	PA16	SDA1 *	0	•	1/0	3/0	2/0	0/6
27	PA27	RGB_LED *	15	•	•	•	•	•
54	PB22	BUTTON *	6	•	•	5/2	7/0	•
18	PA18	RX_LED	2	•	1/2	3/2	3/0	0/2
41	PB09	RX	9	3	•	4/1	4/1	•
40	PB08	TX	8	2	•	4/0	4/0	•
8	PA08	SDA	•	16	0/0	2/0	0/0	1/2
9	PA09	SCL	9	17	0/1	2/1	0/1	1/3
6	PA06	MOSI	6	6	•	0/2	1/0	•
5	PA05	MISO	5	5	•	0/1	0/1	•
7	PA07	SCK	7	7	•	0/3	1/1	•
30	PA30	SWCLK	10	•	•	1/2	1/0	•
31	PA31	SWDIO	11	•	•	1/3	1/1	•
19	PA19	TX_LED	3	•	1/3	3/3	3/1	0/3
24	PA24	USB_DM	12	•	3/2	5/2	5/0	1/2

continues on next page

Table 5 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
25	PA25	USB_DP	13	•	3/3	5/3	5/1	1/3
17	PA17	USER_LEI	1	•	1/1	3/1	2/1	0/7
0	PA00	•	0	•	•	1/0	2/0	•
1	PA01	•	1	•	•	1/1	2/1	•
3	PA03	•	3	1	•	•	•	•
34	PB02	•	2	10	•	5/0	6/0	•
35	PB03	•	3	11	•	5/1	6/1	•

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M0 Express *Adafruit ItsyBitsy M0 Express pin assignment table*.

Pins marked with an asterisk (*) are available only on the XIAO SAMD21 Plus.

The default devices at the board are:

- UART 4 at pins PB08/PB09, labelled D6/D7 (also A6/A7)
- I2C 2 at pins PA08/PA09, labelled D4/D5 (also A4/A5)
- SPI 0 at pins PA06/PA05/PA07, labelled D10, D9 and D8 (also A10, A9 and A8)
- DAC output on pin PA02, labelled A0 (also D0)

16.3.6 Adafruit Feather M0 Express pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
2	PA02	A0	2	0	•	•	•	•
40	PB08	A1	8	2	•	4/0	4/0	•
41	PB09	A2	9	3	•	4/1	4/1	•
4	PA04	A3	4	4	•	0/0	0/0	•
5	PA05	A4	5	5	•	0/1	0/1	•
34	PB02	A5	2	10	•	5/0	6/0	•

continues on next page

Table 6 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
11	PA11	D0	11	19	0/3	2/3	1/1	0/3
10	PA10	D1	10	18	0/2	2/2	1/0	0/2
14	PA14	D2	14	•	2/2	4/2	3/0	0/4
9	PA09	D3	9	17	0/1	2/1	0/1	1/3
8	PA08	D4	•	16	0/0	2/0	0/0	1/2
15	PA15	D5	15	•	2/3	4/3	3/1	0/5
20	PA20	D6	4	•	5/2	3/2	7/0	0/4
21	PA21	D7	5	•	5/3	3/3	7/1	0/7
7	PA07	D9	7	7	•	0/3	1/1	•
55	PB23	RX	7	•	•	5/3	7/1	•
54	PB22	TX	6	•	•	5/2	7/0	•
18	PA18	D10	2	•	1/2	3/2	3/0	0/2
16	PA16	D11	0	•	1/0	3/0	2/0	0/6
19	PA19	D12	3	•	1/3	3/3	3/1	0/3
17	PA17	D13	1	•	1/1	3/1	2/1	0/7
13	PA13	FLASH_CS	13	•	2/1	4/1	2/0	0/7
35	PB03	LED_RX	3	11	•	5/1	6/1	•
27	PA27	LED_TX	15	•	•	•	•	•
12	PA12	MISO	12	•	2/0	4/0	2/0	0/6
42	PB10	MOSI	10	•	•	4/2	5/0	0/4
6	PA06	NEOPIXEL	6	6	•	0/2	1/0	•
43	PB11	SCK	11	•	•	4/3	5/1	0/5
23	PA23	SCL	7	•	3/1	5/1	4/1	0/5

continues on next page

Table 6 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
22	PA22	SDA	6	•	3/0	5/0	4/0	0/4
11	PA11	RX	11	19	0/3	2/3	1/1	0/3
10	PA10	TX	10	18	0/2	2/2	1/0	0/2
30	PA30	SWCLK	10	•	•	1/2	1/0	•
31	PA31	SWDIO	11	•	•	1/3	1/1	•
24	PA24	USB_DM	12	•	3/2	5/2	5/0	1/2
25	PA25	USB_DP	13	•	3/3	5/3	5/1	1/3
0	PA00	•	0	•	•	1/0	2/0	•
1	PA01	•	1	•	•	1/1	2/1	•
3	PA03	•	3	1	•	•	•	•
28	PA28	•	8	•	•	•	•	•

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M0 Express *Adafruit ItsyBitsy M0 Express pin assignment table*.

The default devices at the board are:

- UART 2 at pins PA11/PA10, labelled RX/TX
- I2C 3 at pins PA22/PA23, labelled SDA/SCL
- SPI 4 at pins PB10/PA12/PB11, labelled MOSI, MISO and SCK
- DAC output on pin PA02, labelled A0

16.3.7 Adafruit Trinket M0 pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
8	PA08	D0	•	16	0/0	2/0	0/0	1/2
2	PA02	D1	2	0	•	•	•	•
9	PA09	D2	9	17	0/1	2/1	0/1	1/3
7	PA07	D3	7	7	•	0/3	1/1	•

continues on next page

Table 7 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
6	PA06	D4	6	6	•	0/2	1/0	•
1	PA01	DOT-STAR_CLK	1	•	•	1/1	2/1	•
0	PA00	DOT-STAR_DAT	0	•	•	1/0	2/0	•
7	PA07	RX	7	7	•	0/3	1/1	•
6	PA06	TX	6	6	•	0/2	1/0	•
8	PA08	SDA	•	16	0/0	2/0	0/0	1/2
9	PA09	SCL	9	17	0/1	2/1	0/1	1/3
6	PA06	MOSI	6	6	•	0/2	1/0	•
9	PA09	MISO	9	17	0/1	2/1	0/1	1/3
7	PA07	SCK	7	7	•	0/3	1/1	•
10	PA10	LED	10	18	0/2	2/2	1/0	0/2
30	PA30	SWCLK	10	•	•	1/2	1/0	•
31	PA31	SWDIO	11	•	•	1/3	1/1	•
24	PA24	USB_DM	12	•	3/2	5/2	5/0	1/2
25	PA25	USB_DP	13	•	3/3	5/3	5/1	1/3
3	PA03	•	3	1	•	•	•	•
4	PA04	•	4	4	•	0/0	0/0	•
5	PA05	•	5	5	•	0/1	0/1	•
11	PA11	•	11	19	0/3	2/3	1/1	0/3
14	PA14	•	14	•	2/2	4/2	3/0	0/4
15	PA15	•	15	•	2/3	4/3	3/1	0/5
16	PA16	•	0	•	1/0	3/0	2/0	0/6
17	PA17	•	1	•	1/1	3/1	2/1	0/7

continues on next page

Table 7 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
18	PA18	•	2	•	1/2	3/2	3/0	0/2
19	PA19	•	3	•	1/3	3/3	3/1	0/3
22	PA22	•	6	•	3/0	5/0	4/0	0/4
23	PA23	•	7	•	3/1	5/1	4/1	0/5
27	PA27	•	15	•	•	•	•	•
28	PA28	•	8	•	•	•	•	•

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M0 Express *Adafruit ItsyBitsy M0 Express pin assignment table*.

The default devices at the board are:

- UART 0 at pins PA07/PA06, labelled D3/D4
- I2C 2 at pins PA08/PA09, labelled D0/D2
- SPI 0 at pins PA06/PA09/PA08, labelled D4, D2 and D0
- DAC output on pin PA02, labelled D1

16.3.8 Adafruit QT PY pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
2	PA02	A0	2	0	•	•	•	•
3	PA03	A1	3	1	•	•	•	•
4	PA04	A2	4	4	•	0/0	0/0	•
5	PA05	A3	5	5	•	0/1	0/1	•
7	PA07	RX	7	7	•	0/3	1/1	•
6	PA06	TX	6	6	•	0/2	1/0	•
8	PA08	FLASH_CS	•	16	0/0	2/0	0/0	1/2
19	PA19	FLASH_MI	3	•	1/3	3/3	3/1	0/3
22	PA22	FLASH_MO	6	•	3/0	5/0	4/0	0/4
23	PA23	FLASH_SC	7	•	3/1	5/1	4/1	0/5
9	PA09	MISO	9	17	0/1	2/1	0/1	1/3
10	PA10	MOSI	10	18	0/2	2/2	1/0	0/2
18	PA18	NEOPIX	2	•	1/2	3/2	3/0	0/2
15	PA15	NEO_PWR	15	•	2/3	4/3	3/1	0/5
11	PA11	SCK	11	19	0/3	2/3	1/1	0/3
17	PA17	SCL	1	•	1/1	3/1	2/1	0/7
16	PA16	SDA	0	•	1/0	3/0	2/0	0/6
30	PA30	SWCLK	10	•	•	1/2	1/0	•
31	PA31	SWDIO	11	•	•	1/3	1/1	•
24	PA24	USB_DM	12	•	3/2	5/2	5/0	1/2
25	PA25	USB_DP	13	•	3/3	5/3	5/1	1/3

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M0 Express *Adafruit ItsyBitsy M0 Express pin assignment table*.

The default devices at the board are:

- UART 0 at pins PA07/PA06, labelled RX/TX
- I2C 1 at pins PA16/PA17, labelled SDA/SCL
- SPI 0 at pins PA09/PA10/PA11, labelled MISO, MOSI and SCK
- DAC output on pin PA02, labelled A0

16.3.9 Adafruit NeoKey Trinkey pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
15	PA15	NEOPIXEI	15	•	2/3	4/3	3/1	0/5
30	PA30	SWCLK	10	•	•	1/2	1/0	•
31	PA31	SWDIO	11	•	•	1/3	1/1	•
18	PA18	SWITCH	2	•	1/2	3/2	3/0	0/2
7	PA07	TOUCH	7	7	•	0/3	1/1	•
24	PA24	USB_DM	12	•	3/2	5/2	5/0	1/2
25	PA25	USB_DP	13	•	3/3	5/3	5/1	1/3

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M0 Express *Adafruit ItsyBitsy M0 Express pin assignment table*.

The board does not provide access to UART, I2C, SPI or DAC.

16.3.10 SparkFun Redboard Turbo assignment table

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
2	PA02	A0	2	0	•	•	•	•
40	PB08	A1	8	2	•	4/0	4/0	•
41	PB09	A2	9	3	•	4/1	4/1	•
4	PA04	A3	4	4	•	0/0	0/0	•
5	PA05	A4	5	5	•	0/1	0/1	•

continues on next page

Table 8 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
34	PB02	A5	2	10	•	5/0	6/0	•
11	PA11	D0	11	19	0/3	2/3	1/1	0/3
10	PA10	D1	10	18	0/2	2/2	1/0	0/2
14	PA14	D2	14	•	2/2	4/2	3/0	0/4
9	PA09	D3	9	17	0/1	2/1	0/1	1/3
8	PA08	D4	•	16	0/0	2/0	0/0	1/2
15	PA15	D5	15	•	2/3	4/3	3/1	0/5
20	PA20	D6	4	•	5/2	3/2	7/0	0/4
21	PA21	D7	5	•	5/3	3/3	7/1	0/7
6	PA06	D8	6	6	•	0/2	1/0	•
7	PA07	D9	7	7	•	0/3	1/1	•
11	PA11	RX	11	19	0/3	2/3	1/1	0/3
10	PA10	TX	10	18	0/2	2/2	1/0	0/2
3	PA03	AREF	3	1	•	•	•	•
18	PA18	D10	2	•	1/2	3/2	3/0	0/2
16	PA16	D11	0	•	1/0	3/0	2/0	0/6
19	PA19	D12	3	•	1/3	3/3	3/1	0/3
17	PA17	D13	1	•	1/1	3/1	2/1	0/7
13	PA13	FLASH_CS	13	•	2/1	4/1	2/0	0/7
35	PB03	FLASH_MI	3	11	•	5/1	6/1	•
54	PB22	FLASH_MO	6	•	•	5/2	7/0	•
55	PB23	FLASH_SC	7	•	•	5/3	7/1	•
31	PA31	LED_RX	11	•	•	1/3	1/1	•
27	PA27	LED_TX	15	•	•	•	•	•

continues on next page

Table 8 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
12	PA12	MISO	12	•	2/0	4/0	2/0	0/6
42	PB10	MOSI	10	•	•	4/2	5/0	0/4
30	PA30	NEOPIXEI	10	•	•	1/2	1/0	•
43	PB11	SCK	11	•	•	4/3	5/1	0/5
23	PA23	SCL	7	•	3/1	5/1	4/1	0/5
22	PA22	SDA	6	•	3/0	5/0	4/0	0/4
30	PA30	SWCLK	10	•	•	1/2	1/0	•
31	PA31	SWDIO	11	•	•	1/3	1/1	•
24	PA24	USB_DM	12	•	3/2	5/2	5/0	1/2
25	PA25	USB_DP	13	•	3/3	5/3	5/1	1/3
0	PA00		0	•	•	1/0	2/0	•
1	PA01		1	•	•	1/1	2/1	•
28	PA28		8	•	•	•	•	•

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M0 Express *Adafruit ItsyBitsy M0 Express pin assignment table*.

The default devices at the board are:

- UART 0 at pins PA11/PA10, labelled RX/TX
- I2C 3 at pins PA22/PA23, labelled SDA/SCL
- SPI 4 at pins PB10/PA12/PB11, labelled MISO, MOSI and SCK
- DAC output on pin PA02, labelled A0

16.3.11 SparkFun SAMD21 Dev Breakout assignment table

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
2	PA02	A0	2	0	•	•	•	•
40	PB08	A1	8	2	•	4/0	4/0	•
41	PB09	A2	9	3	•	4/1	4/1	•
4	PA04	A3	4	4	•	0/0	0/0	•
5	PA05	A4	5	5	•	0/1	0/1	•
34	PB02	A5	2	10	•	5/0	6/0	•
11	PA11	D0	11	19	0/3	2/3	1/1	0/3
10	PA10	D1	10	18	0/2	2/2	1/0	0/2
14	PA14	D2	14	•	2/2	4/2	3/0	0/4
9	PA09	D3	9	17	0/1	2/1	0/1	1/3
8	PA08	D4	•	16	0/0	2/0	0/0	1/2
15	PA15	D5	15	•	2/3	4/3	3/1	0/5
20	PA20	D6	4	•	5/2	3/2	7/0	0/4
21	PA21	D7	5	•	5/3	3/3	7/1	0/7
6	PA06	D8	6	6	•	0/2	1/0	•
7	PA07	D9	7	7	•	0/3	1/1	•
11	PA11	RX	11	19	0/3	2/3	1/1	0/3
10	PA10	TX	10	18	0/2	2/2	1/0	0/2
3	PA03	AREF	3	1	•	•	•	•
18	PA18	D10	2	•	1/2	3/2	3/0	0/2
16	PA16	D11	0	•	1/0	3/0	2/0	0/6
19	PA19	D12	3	•	1/3	3/3	3/1	0/3
17	PA17	D13	1	•	1/1	3/1	2/1	0/7

continues on next page

Table 9 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
54	PB22	D30	6	•	•	5/2	7/0	•
55	PB23	D31	7	•	•	5/3	7/1	•
13	PA13	D38	13	•	2/1	4/1	2/0	0/7
35	PB03	LED_RX	3	11	•	5/1	6/1	•
27	PA27	LED_TX	15	•	•	•	•	•
12	PA12	MISO	12	•	2/0	4/0	2/0	0/6
42	PB10	MOSI	10	•	•	4/2	5/0	0/4
43	PB11	SCK	11	•	•	4/3	5/1	0/5
23	PA23	SCL	7	•	3/1	5/1	4/1	0/5
22	PA22	SDA	6	•	3/0	5/0	4/0	0/4
30	PA30	SWCLK	10	•	•	1/2	1/0	•
31	PA31	SWDIO	11	•	•	1/3	1/1	•
24	PA24	USB_DM	12	•	3/2	5/2	5/0	1/2
25	PA25	USB_DP	13	•	3/3	5/3	5/1	1/3
0	PA00		0	•	•	1/0	2/0	•
1	PA01		1	•	•	1/1	2/1	•
28	PA28		8	•	•	•	•	•

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M0 Express *Adafruit ItsyBitsy M0 Express pin assignment table*.

The default devices at the board are:

- UART 0 at pins PA11/PA10, labelled RX/TX
- I2C 3 at pins PA22/PA23, labelled SDA/SCL
- SPI 4 at pins PB10/PA12/PB11, labelled MISO, MOSI and SCK
- DAC output on pin PA02, labelled A0

16.3.12 SAMD21 Xplained PRO pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
32	PB00	EXT1_PIN	0	8	•	5/2	7/0	•
33	PB01	EXT1_PIN	1	9	•	5/3	7/1	•
38	PB06	EXT1_PIN	6	14	•	•	•	•
39	PB07	EXT1_PIN	7	15	•	•	•	•
34	PB02	EXT1_PIN	2	10	•	5/0	6/0	•
35	PB03	EXT1_PIN	3	11	•	5/1	6/1	•
36	PB04	EXT1_PIN	4	12	•	•	•	•
37	PB05	EXT1_PIN	5	13	•	•	•	•
8	PA08	EXT1_PIN	•	16	0/0	2/0	0/0	1/2
9	PA09	EXT1_PIN	9	17	0/1	2/1	0/1	1/3
41	PB09	EXT1_PIN	9	3	•	4/1	4/1	•
40	PB08	EXT1_PIN	8	2	•	4/0	4/0	•
5	PA05	EXT1_PIN	5	5	•	0/1	0/1	•
6	PA06	EXT1_PIN	6	6	•	0/2	1/0	•
4	PA04	EXT1_PIN	4	4	•	0/0	0/0	•
7	PA07	EXT1_PIN	7	7	•	0/3	1/1	•
10	PA10	EXT2_PIN	10	18	0/2	2/2	1/0	0/2
11	PA11	EXT2_PIN	11	19	0/3	2/3	1/1	0/3
20	PA20	EXT2_PIN	4	•	5/2	3/2	7/0	0/4
21	PA21	EXT2_PIN	5	•	5/3	3/3	7/1	0/7
44	PB12	EXT2_PIN	12	•	4/0	•	4/0	0/6
45	PB13	EXT2_PIN	13	•	4/1	•	4/1	0/7

continues on next page

Table 10 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
46	PB14	EXT2_PIN	14	•	4/2	•	5/0	•
47	PB15	EXT2_PIN	15	•	4/3	•	5/1	•
43	PB11	EXT2_PIN	11	•	•	4/3	5/1	0/5
42	PB10	EXT2_PIN	10	•	•	4/2	5/0	0/4
17	PA17	EXT2_PIN	1	•	1/1	3/1	2/1	0/7
18	PA18	EXT2_PIN	2	•	1/2	3/2	3/0	0/2
16	PA16	EXT2_PIN	0	•	1/0	3/0	2/0	0/6
19	PA19	EXT2_PIN	3	•	1/3	3/3	3/1	0/3
2	PA02	EXT3_PIN	2	0	•	•	•	•
3	PA03	EXT3_PIN	3	1	•	•	•	•
15	PA15	EXT3_PIN	15	•	2/3	4/3	3/1	0/5
12	PA12	EXT3_PIN	12	•	2/0	4/0	2/0	0/6
13	PA13	EXT3_PIN	13	•	2/1	4/1	2/0	0/7
28	PA28	EXT3_PIN	8	•	•	•	•	•
27	PA27	EXT3_PIN	15	•	•	•	•	•
49	PB17	EXT3_PIN	1	•	5/1	•	6/1	0/5
54	PB22	EXT3_PIN	6	•	•	5/2	7/0	•
48	PB16	EXT3_PIN	9	•	5/0	•	6/0	0/4
55	PB23	EXT3_PIN	7	•	•	5/3	7/1	•
62	PB30	LED	14	•	•	5/0	0/0	1/2
30	PA30	SWCLK	10	•	•	1/2	1/0	•

continues on next page

Table 10 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
31	PA31	SWDIO	11	•	•	1/3	1/1	•
24	PA24	USB_DM	12	•	3/2	5/2	5/0	1/2
25	PA25	USB_DP	13	•	3/3	5/3	5/1	1/3
0	PA00	•	0	•	•	1/0	2/0	•
1	PA01	•	1	•	•	1/1	2/1	•
14	PA14	•	14	•	2/2	4/2	3/0	0/4
22	PA22	•	6	•	3/0	5/0	4/0	0/4
23	PA23	•	7	•	3/1	5/1	4/1	0/5
63	PB31	•	15	•	•	5/1	0/1	1/3

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M0 Express *Adafruit ItsyBitsy M0 Express pin assignment table*.

There are no pins labelled for default devices on this board. DAC output is on pin PA02, labelled EXT3_PIN3

16.3.13 Minisam M4 pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
16	PA16	D0	0	•	•	1/0	3/1	2/0	1/0	0/4
17	PA17	D1	1	•	•	1/1	3/0	2/1	1/1	0/5
19	PA19	D3	3	•	•	1/3	3/3	3/1	1/3	0/7
20	PA20	D4	4	•	•	5/2	3/2	7/0	1/4	0/0
21	PA21	D5	5	•	•	5/3	3/3	7/1	1/5	0/1
2	PA02	A0_D9	2	0	•	•	•	•	•	•
40	PB08	A1_D10	8	2	0	•	4/0	4/0	•	•

continues on next page

Table 11 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
41	PB09	A2_D11	9	3	1	•	4/1	4/1	•	•
4	PA04	A3_D12	4	4	•	•	0/0	0/0	•	•
5	PA05	A4_D13	5	5	•	•	0/1	0/1	•	•
6	PA06	A5	6	6	•	•	0/2	1/0	•	•
7	PA07	A6_D2	7	7	•	•	0/3	1/1	•	•
3	PA03	AREF	3	10	•	•	•	•	•	•
0	PA00	BUT-TON	0	•	•	•	1/0	2/0	•	•
34	PB02	DOT-STAR_C	2	14	•	•	5/0	6/0	2/2	•
35	PB03	DOT-STAR_D	9	15	•	•	5/1	6/1	•	•
15	PA15	LED	15	•	•	2/3	4/3	3/1	2/1	1/3
16	PA16	RX	0	•	•	1/0	3/1	2/0	1/0	0/4
17	PA17	TX	1	•	•	1/1	3/0	2/1	1/1	0/5
55	PB23	MOSI	7	•	•	1/3	5/3	7/1	•	•
54	PB22	MISO	22	•	•	1/2	5/2	7/0	•	•
43	PB11	QSPI_CS	12	•	•	•	4/3	5/1	0/5	1/1
8	PA08	QSPI_D(	•	8	2	0/0	2/1	0/0	0/0	1/4
9	PA09	QSPI_D1	9	9	3	0/1	2/0	0/1	0/1	1/5
10	PA10	QSPI_D2	10	10	•	0/2	2/2	1/0	0/2	1/6
11	PA11	QSPI_D3	11	11	•	0/3	2/3	1/1	0/3	1/7
42	PB10	QSPI_SC	10	•	•	•	4/2	5/0	0/4	1/0
1	PA01	SCK	1	•	•	•	1/1	2/1	•	•

continues on next page

Table 11 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
13	PA13	SCL	13	•	•	2/1	4/0	2/1	0/7	1/3
12	PA12	SDA	12	•	•	2/0	4/1	2/0	0/6	1/2
30	PA30	SWCLK	14	•	•	7/2	1/2	6/0	2/0	•
31	PA31	SWDIO	15	•	•	7/3	1/3	6/1	2/1	•
24	PA24	USB_DM	8	•	•	3/2	5/2	5/0	2/2	•
25	PA25	USB_DP	9	•	•	3/3	5/3	5/1	•	•
14	PA14	•	14	•	•	2/2	4/2	3/0	2/0	1/2
18	PA18	•	2	•	•	1/2	3/2	3/0	1/2	0/6
22	PA22	•	6	•	•	3/0	5/1	4/0	1/6	0/2
23	PA23	•	7	•	•	3/1	5/0	4/1	1/7	0/3
27	PA27	•	11	•	•	•	•	•	•	•

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M4 Express *Adafruit ItsyBitsy M4 Express pin assignment table*.

The default devices at the board are:

- UART 1 at pins PA16/PA17, labelled D0/D1
- I2C 2 at pins PA12/PA13, labelled SDA/SCL
- SPI 1 at pins PB22/PB23/PA01, labelled MOSI, MISO and SCK
- DAC output on pins PA02 and PA05, labelled A0_D9 and A4_D13

16.3.14 Seeed WIO Terminal pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
33	PB01	CS	1	13	•	•	5/3	7/1	•	•
59	PB27	RX	13	•	•	2/1	4/0	•	1/3	•

continues on next page

Table 12 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
58	PB26	TX	12	•	•	2/0	4/1	•	1/2	•
79	PC15	3V3_EN.	15	•	•	7/3	6/3	•	0/5	1/1
78	PC14	5V_ENA	14	•	•	7/2	6/2	•	0/4	1/0
40	PB08	A0_D0	8	2	0	•	4/0	4/0	•	•
41	PB09	A1_D1	9	3	1	•	4/1	4/1	•	•
7	PA07	A2_D2	7	7	•	•	0/3	1/1	•	•
36	PB04	A3_D3	4	•	6	•	•	•	•	•
37	PB05	A4_D4	5	•	7	•	•	•	•	•
38	PB06	A5_D5	6	•	8	•	•	•	•	•
4	PA04	A6_D6	4	4	•	•	0/0	0/0	•	•
39	PB07	A7_D7	7	•	9	•	•	•	•	•
6	PA06	A8_D8	6	6	•	•	0/2	1/0	•	•
90	PC26	BUT-TON_1	10	•	•	•	•	•	•	•
91	PC27	BUT-TON_2	11	•	•	1/0	•	•	•	•
92	PC28	BUT-TON_3	12	•	•	1/1	•	•	•	•
107	PD11	BUZZER	6	•	•	7/3	6/3	•	0/4	•
47	PB15	GP-CLK0	15	•	•	4/3	•	5/1	4/1	0/3
44	PB12	GP-CLK1	12	•	•	4/0	•	4/0	3/0	0/0
45	PB13	GP-CLK2	13	•	•	4/1	•	4/1	3/1	0/1
48	PB16	I2C_BCI	0	•	•	5/0	•	6/0	3/0	0/4

continues on next page

Table 12 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
20	PA20	I2S_LRC	4	•	•	5/2	3/2	7/0	1/4	0/0
21	PA21	I2S_SDI	5	•	•	5/3	3/3	7/1	1/5	0/1
22	PA22	I2S_SDC	6	•	•	3/0	5/1	4/0	1/6	0/2
50	PB18	LCD_MI	2	•	•	5/2	7/2	•	1/0	•
51	PB19	LCD_MC	3	•	•	5/3	7/3	•	1/1	•
52	PB20	LCD_SC	4	•	•	3/0	7/1	•	1/2	•
53	PB21	LCD_CS	5	•	•	3/1	7/0	•	1/3	•
70	PC06	LCD_D/C	6	•	•	6/2	•	•	•	•
71	PC07	LCD_RE	9	•	•	6/3	•	•	•	•
74	PC10	LCD_XL	10	•	•	6/2	7/2	•	0/0	1/4
76	PC12	LCD_XR	12	•	•	7/0	6/1	•	0/2	1/6
77	PC13	LCD_YL	13	•	•	7/1	6/0	•	0/3	1/7
75	PC11	LCD_YU	11	•	•	6/3	7/3	•	0/1	1/5
15	PA15	LED_BL	15	•	•	2/3	4/3	3/1	2/1	1/3
69	PC05	LED_LC	5	•	•	6/1	•	•	•	•
94	PC30	MIC	14	•	12	•	•	•	•	•
32	PB00	MISO	9	12	•	•	5/2	7/0	•	•
34	PB02	MOSI	2	14	•	•	5/0	6/0	2/2	•
35	PB03	SCK	9	15	•	•	5/1	6/1	•	•
12	PA12	SCL0	12	•	•	2/0	4/1	2/0	0/6	1/2

continues on next page

Table 12 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
13	PA13	SDA0	13	•	•	2/1	4/0	2/1	0/7	1/3
16	PA16	SCL1	0	•	•	1/0	3/1	2/0	1/0	0/4
17	PA17	SDA1	1	•	•	1/1	3/0	2/1	1/1	0/5
117	PD21	SD_DET	11	•	•	1/3	3/3	•	1/1	•
83	PC19	SD_CS	3	•	•	6/3	0/3	•	0/3	•
82	PC18	SD_MIS(	2	•	•	6/2	0/2	•	0/2	•
43	PB11	QSPI_CS	12	•	•	•	4/3	5/1	0/5	1/1
8	PA08	QSPI_D(	•	8	2	0/0	2/1	0/0	0/0	1/4
9	PA09	QSPI_D1	9	9	3	0/1	2/0	0/1	0/1	1/5
10	PA10	QSPI_D2	10	10	•	0/2	2/2	1/0	0/2	1/6
11	PA11	QSPI_D3	11	11	•	0/3	2/3	1/1	0/3	1/7
42	PB10	QSPI_SC	10	•	•	•	4/2	5/0	0/4	1/0
80	PC16	SD_MOS	0	•	•	6/0	0/1	•	0/0	•
81	PC17	SD_SCK	1	•	•	6/1	0/0	•	0/1	•
30	PA30	SWCLK	14	•	•	7/2	1/2	6/0	2/0	•
31	PA31	SWDIO	15	•	•	7/3	1/3	6/1	2/1	•
108	PD12	SWITCH	7	•	•	•	•	•	0/5	•
116	PD20	SWITCH	10	•	•	1/2	3/2	•	1/0	•
104	PD08	SWITCH	3	•	•	7/0	6/1	•	0/1	•
105	PD09	SWITCH	4	•	•	7/1	6/0	•	0/2	•
106	PD10	SWITCH	5	•	•	7/2	6/2	•	0/3	•

continues on next page

Table 12 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
24	PA24	USB_DM	8	•	•	3/2	5/2	5/0	2/2	•
25	PA25	USB_DP	9	•	•	3/3	5/3	5/1	•	•
0	PA00	•	0	•	•	•	1/0	2/0	•	•
1	PA01	•	1	•	•	•	1/1	2/1	•	•
2	PA02	•	2	0	•	•	•	•	•	•
3	PA03	•	3	10	•	•	•	•	•	•
5	PA05	•	5	5	•	•	0/1	0/1	•	•
14	PA14	•	14	•	•	2/2	4/2	3/0	2/0	1/2
18	PA18	•	2	•	•	1/2	3/2	3/0	1/2	0/6
19	PA19	•	3	•	•	1/3	3/3	3/1	1/3	0/7
23	PA23	•	7	•	•	3/1	5/0	4/1	1/7	0/3
27	PA27	•	11	•	•	•	•	•	•	•
46	PB14	•	14	•	•	4/2	•	5/0	4/0	0/2
49	PB17	•	1	•	•	5/1	•	6/1	3/1	0/5
54	PB22	•	22	•	•	1/2	5/2	7/0	•	•
55	PB23	•	7	•	•	1/3	5/3	7/1	•	•
56	PB24	•	8	•	•	0/0	2/1	•	•	•
57	PB25	•	9	•	•	0/1	2/0	•	•	•
60	PB28	•	14	•	•	2/2	4/2	•	1/4	•
61	PB29	•	15	•	•	2/3	4/3	•	1/5	•

continues on next page

Table 12 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
62	PB30	•	14	•	•	7/0	5/1	0/0	4/0	0/6
63	PB31	•	15	•	•	7/1	5/0	0/1	4/1	0/7
64	PC00	•	0	•	10	•	•	•	•	•
65	PC01	•	1	•	11	•	•	•	•	•
66	PC02	•	2	•	4	•	•	•	•	•
67	PC03	•	3	•	5	•	•	•	•	•
68	PC04	•	4	•	•	6/0	•	•	0/0	•
84	PC20	•	4	•	•	•	•	•	0/4	•
85	PC21	•	5	•	•	•	•	•	0/5	•
86	PC22	•	6	•	•	1/0	3/1	•	0/5	•
87	PC23	•	7	•	•	1/1	3/0	•	0/7	•
88	PC24	•	8	•	•	0/2	2/2	•	•	•
89	PC25	•	9	•	•	0/3	2/3	•	•	•
95	PC31	•	15	•	13	•	•	•	•	•
96	PD00	•	0	•	14	•	•	•	•	•
97	PD01	•	1	•	15	•	•	•	•	•

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M4 Express *Adafruit ItsyBitsy M4 Express pin assignment table*.

Default pin assignments: - UART 2 at pins PB27 and PB26, labelled RX and TX - I2C 4 at pins PA12 and PA13, labelled SCL0 and SDA0 - I2C 3 at pins PA16 and PA17, labelled SCL1 and SDA1 - SPI 5 at pins PB00, PB02 and PB03, labelle MISO, MOSI and SCK

There seems to be no default pin assignment for this board.

16.3.15 SparkFun SAMD51 Thing Plus pin assignment table

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
2	PA02	A0	2	0	•	•	•	•	•	•
40	PB08	A1	8	2	0	•	4/0	4/0	•	•
41	PB09	A2	9	3	1	•	4/1	4/1	•	•
4	PA04	A3	4	4	•	•	0/0	0/0	•	•
5	PA05	A4	5	5	•	•	0/1	0/1	•	•
34	PB02	A5	2	14	•	•	5/0	6/0	2/2	•
13	PA13	D0	13	•	•	2/1	4/0	2/1	0/7	1/3
12	PA12	D1	12	•	•	2/0	4/1	2/0	0/6	1/2
6	PA06	D4	6	6	•	•	0/2	1/0	•	•
15	PA15	D5	15	•	•	2/3	4/3	3/1	2/1	1/3
20	PA20	D6	4	•	•	5/2	3/2	7/0	1/4	0/0
21	PA21	D7	5	•	•	5/3	3/3	7/1	1/5	0/1
7	PA07	D9	7	7	•	•	0/3	1/1	•	•
18	PA18	D10	2	•	•	1/2	3/2	3/0	1/2	0/6
16	PA16	D11	0	•	•	1/0	3/1	2/0	1/0	0/4
19	PA19	D12	3	•	•	1/3	3/3	3/1	1/3	0/7
17	PA17	D13	1	•	•	1/1	3/0	2/1	1/1	0/5
10	PA10	FLASH_	10	10	•	0/2	2/2	1/0	0/2	1/6
11	PA11	FLASH_	11	11	•	0/3	2/3	1/1	0/3	1/7
8	PA08	FLASH_	•	8	2	0/0	2/1	0/0	0/0	1/4

continues on next page

Table 13 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
9	PA09	FLASH_	9	9	3	0/1	2/0	0/1	0/1	1/5
13	PA13	RX	13	•	•	2/1	4/0	2/1	0/7	1/3
12	PA12	TX	12	•	•	2/0	4/1	2/0	0/6	1/2
43	PB11	MISO	12	•	•	•	4/3	5/1	0/5	1/1
44	PB12	MOSI	12	•	•	4/0	•	4/0	3/0	0/0
55	PB23	RXD	7	•	•	1/3	5/3	7/1	•	•
35	PB03	RXLED	9	15	•	•	5/1	6/1	•	•
45	PB13	SCK	13	•	•	4/1	•	4/1	3/1	0/1
23	PA23	SCL	7	•	•	3/1	5/0	4/1	1/7	0/3
22	PA22	SDA	6	•	•	3/0	5/1	4/0	1/6	0/2
30	PA30	SWCLK	14	•	•	7/2	1/2	6/0	2/0	•
31	PA31	SWDIO	15	•	•	7/3	1/3	6/1	2/1	•
54	PB22	TXD	22	•	•	1/2	5/2	7/0	•	•
27	PA27	TXLED	11	•	•	•	•	•	•	•
24	PA24	USB_DM	8	•	•	3/2	5/2	5/0	2/2	•
25	PA25	USB_DP	9	•	•	3/3	5/3	5/1	•	•
0	PA00	•	0	•	•	•	1/0	2/0	•	•
1	PA01	•	1	•	•	•	1/1	2/1	•	•
3	PA03	•	3	10	•	•	•	•	•	•
14	PA14	•	14	•	•	2/2	4/2	3/0	2/0	1/2
32	PB00	•	9	12	•	•	5/2	7/0	•	•

continues on next page

Table 13 – continued from previous page

Pin	GPIO	Pin name	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
33	PB01	•	1	13	•	•	5/3	7/1	•	•
36	PB04	•	4	•	6	•	•	•	•	•
37	PB05	•	5	•	7	•	•	•	•	•
38	PB06	•	6	•	8	•	•	•	•	•
39	PB07	•	7	•	9	•	•	•	•	•
42	PB10	•	10	•	•	•	4/2	5/0	0/4	1/0
46	PB14	•	14	•	•	4/2	•	5/0	4/0	0/2
47	PB15	•	15	•	•	4/3	•	5/1	4/1	0/3
48	PB16	•	0	•	•	5/0	•	6/0	3/0	0/4
49	PB17	•	1	•	•	5/1	•	6/1	3/1	0/5
62	PB30	•	14	•	•	7/0	5/1	0/0	4/0	0/6
63	PB31	•	15	•	•	7/1	5/0	0/1	4/1	0/7

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M4 Express *Adafruit ItsyBitsy M4 Express pin assignment table*.

The default devices at the board are:

- UART 2 at pins PA13/PA12, labelled RXD/TXD
- I2C 3 at pins PA22/PA23, labelled SDA/SCL
- SPI 4 at pins PB12/PB11/PB13, labelled MOSI, MISO and SCK
- DAC output on pins PA02 and PA05, labelled A0 and A4

16.3.16 Generic SAMD21x18 pin assignment table

Pin	GPIO	Name/Pac	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
0	PA00	EGJ	0	•	•	1/0	2/0	•
1	PA01	EGJ	1	•	•	1/1	2/1	•
2	PA02	EGJ	2	0	•	•	•	•
3	PA03	EGJ	3	1	•	•	•	•
4	PA04	EGJ	4	4	•	0/0	0/0	•
5	PA05	EGJ	5	5	•	0/1	0/1	•
6	PA06	EGJ	6	6	•	0/2	1/0	•
7	PA07	EGJ	7	7	•	0/3	1/1	•
8	PA08	EGJ	•	16	0/0	2/0	0/0	1/2
9	PA09	EGJ	9	17	0/1	2/1	0/1	1/3
10	PA10	EGJ	10	18	0/2	2/2	1/0	0/2
11	PA11	EGJ	11	19	0/3	2/3	1/1	0/3
12	PA12	GJ	12	•	2/0	4/0	2/0	0/6
13	PA13	GJ	13	•	2/1	4/1	2/0	0/7
14	PA14	EGJ	14	•	2/2	4/2	3/0	0/4
15	PA15	EGJ	15	•	2/3	4/3	3/1	0/5
16	PA16	EGJ	0	•	1/0	3/0	2/0	0/6
17	PA17	EGJ	1	•	1/1	3/1	2/1	0/7
18	PA18	EGJ	2	•	1/2	3/2	3/0	0/2
19	PA19	EGJ	3	•	1/3	3/3	3/1	0/3
20	PA20	GJ	4	•	5/2	3/2	7/0	0/4
21	PA21	GJ	5	•	5/3	3/3	7/1	0/7

continues on next page

Table 14 – continued from previous page

Pin	GPIO	Name/Pac	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
22	PA22	EGJ	6	•	3/0	5/0	4/0	0/4
23	PA23	EGJ	7	•	3/1	5/1	4/1	0/5
24	PA24	USB_DM	12	•	3/2	5/2	5/0	1/2
25	PA25	USB_DP	13	•	3/3	5/3	5/1	1/3
27	PA27	EGJ	15	•	•	•	•	•
28	PA28	EGJ	8	•	•	•	•	•
30	PA30	SWCLK	10	•	•	1/2	1/0	•
31	PA31	SWDIO	11	•	•	1/3	1/1	•
32	PB00	J	0	8	•	5/2	7/0	•
33	PB01	J	1	9	•	5/3	7/1	•
34	PB02	GJ	2	10	•	5/0	6/0	•
35	PB03	GJ	3	11	•	5/1	6/1	•
36	PB04	J	4	12	•	•	•	•
37	PB05	J	5	13	•	•	•	•
38	PB06	J	6	14	•	•	•	•
39	PB07	J	7	15	•	•	•	•
40	PB08	GJ	8	2	•	4/0	4/0	•
41	PB09	GJ	9	3	•	4/1	4/1	•
42	PB10	GJ	10	•	•	4/2	5/0	0/4
43	PB11	GJ	11	•	•	4/3	5/1	0/5
44	PB12	J	12	•	4/0	•	4/0	0/6

continues on next page

Table 14 – continued from previous page

Pin	GPIO	Name/Pac	IRQ	ADC	Serial	Serial	TCC/TC	TCC/TC
45	PB13	J	13	•	4/1	•	4/1	0/7
46	PB14	J	14	•	4/2	•	5/0	•
47	PB15	J	15	•	4/3	•	5/1	•
48	PB16	J	0	•	5/0	•	6/0	0/4
49	PB17	J	1	•	5/1	•	6/1	0/5
54	PB22	GJ	6	•	•	5/2	7/0	•
55	PB23	GJ	7	•	•	5/3	7/1	•
62	PB30	J	14	•	•	5/0	0/0	1/2
63	PB31	J	15	•	•	5/1	0/1	1/3

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M0 Express *Adafruit ItsyBitsy M0 Express pin assignment table*.

The Package column indicates the package letter providing this pin. An entry EGJ tells for instance, that the pin is available for SAMD21E18, SAMD21G18 and SAMD21J18.

16.3.17 Generic SAMD51x19 and SAM51x20 pin assignment table

For the definition of the table columns see the explanation at the table for Adafruit ItsyBitsy M4 Express *Adafruit ItsyBitsy M4 Express pin assignment table*.

Pin	GPIO	Name/P	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
8	PA08	QSPI_D(	•	8	2	0/0	2/1	0/0	0/0	1/4
9	PA09	QSPI_D1	9	9	3	0/1	2/0	0/1	0/1	1/5
10	PA10	QSPI_D2	10	10	•	0/2	2/2	1/0	0/2	1/6
11	PA11	QSPI_D3	11	11	•	0/3	2/3	1/1	0/3	1/7
42	PB10	QSPI_SC	10	•	•	•	4/2	5/0	0/4	1/0
23	PA23	USB_SO	7	•	•	3/1	5/0	4/1	1/7	0/3

continues on next page

Table 15 – continued from previous page

Pin	GPIO	Name/P	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
24	PA24	USB_DM	8	•	•	3/2	5/2	5/0	2/2	•
25	PA25	USB_DP	9	•	•	3/3	5/3	5/1	•	•
0	PA00	GJP	0	•	•	•	1/0	2/0	•	•
1	PA01	GJP	1	•	•	•	1/1	2/1	•	•
2	PA02	GJP	2	0	•	•	•	•	•	•
3	PA03	GJP	3	10	•	•	•	•	•	•
4	PA04	GJP	4	4	•	•	0/0	0/0	•	•
5	PA05	GJP	5	5	•	•	0/1	0/1	•	•
6	PA06	GJP	6	6	•	•	0/2	1/0	•	•
7	PA07	GJP	7	7	•	•	0/3	1/1	•	•
12	PA12	GJP	12	•	•	2/0	4/1	2/0	0/6	1/2
13	PA13	GJP	13	•	•	2/1	4/0	2/1	0/7	1/3
14	PA14	GJP	14	•	•	2/2	4/2	3/0	2/0	1/2
15	PA15	GJP	15	•	•	2/3	4/3	3/1	2/1	1/3
16	PA16	GJP	0	•	•	1/0	3/1	2/0	1/0	0/4
17	PA17	GJP	1	•	•	1/1	3/0	2/1	1/1	0/5
18	PA18	GJP	2	•	•	1/2	3/2	3/0	1/2	0/6
19	PA19	GJP	3	•	•	1/3	3/3	3/1	1/3	0/7
20	PA20	GJP	4	•	•	5/2	3/2	7/0	1/4	0/0
21	PA21	GJP	5	•	•	5/3	3/3	7/1	1/5	0/1
22	PA22	GJP	6	•	•	3/0	5/1	4/0	1/6	0/2

continues on next page

Table 15 – continued from previous page

Pin	GPIO	Name/P	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
27	PA27	GJP	11	•	•	•	•	•	•	•
30	PA30	SWCLK	14	•	•	7/2	1/2	6/0	2/0	•
31	PA31	SWDIO	15	•	•	7/3	1/3	6/1	2/1	•
32	PB00	JP	0	12	•	•	5/2	7/0	•	•
33	PB01	JP	1	13	•	•	5/3	7/1	•	•
34	PB02	GJP	2	14	•	•	5/0	6/0	2/2	•
35	PB03	GJP	3	15	•	•	5/1	6/1	•	•
36	PB04	JP	4	•	6	•	•	•	•	•
37	PB05	JP	5	•	7	•	•	•	•	•
38	PB06	JP	6	•	8	•	•	•	•	•
39	PB07	JP	7	•	9	•	•	•	•	•
40	PB08	GJP	8	2	0	•	4/0	4/0	•	•
41	PB09	GJP	9	3	1	•	4/1	4/1	•	•
44	PB12	JP	12	•	•	4/0	•	4/0	3/0	0/0
45	PB13	JP	13	•	•	4/1	•	4/1	3/1	0/1
46	PB14	JP	14	•	•	4/2	•	5/0	4/0	0/2
47	PB15	JP	15	•	•	4/3	•	5/1	4/1	0/3
48	PB16	JP	0	•	•	5/0	•	6/0	3/0	0/4
49	PB17	JP	1	•	•	5/1	•	6/1	3/1	0/5
50	PB18	P	2	•	•	5/2	7/2	•	1/0	•
51	PB19	P	3	•	•	5/3	7/3	•	1/1	•

continues on next page

Table 15 – continued from previous page

Pin	GPIO	Name/P	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
52	PB20	P	4	•	•	3/0	7/1	•	1/2	•
53	PB21	P	5	•	•	3/1	7/0	•	1/3	•
54	PB22	GJP	6	•	•	1/2	5/2	7/0	•	•
55	PB23	GJP	7	•	•	1/3	5/3	7/1	•	•
56	PB24	P	8	•	•	0/0	2/1	•	•	•
57	PB25	P	9	•	•	0/1	2/0	•	•	•
58	PB26	P	12	•	•	2/0	4/1	•	1/2	•
59	PB27	P	13	•	•	2/1	4/0	•	1/3	•
60	PB28	P	14	•	•	2/2	4/2	•	1/4	•
61	PB29	P	15	•	•	2/3	4/3	•	1/5	•
62	PB30	JP	14	•	•	7/0	5/1	0/0	4/0	0/6
63	PB31	JP	15	•	•	7/1	5/0	0/1	4/1	0/7
64	PC00	P	0	•	10	•	•	•	•	•
65	PC01	P	1	•	11	•	•	•	•	•
66	PC02	P	2	•	4	•	•	•	•	•
67	PC03	P	3	•	5	•	•	•	•	•
68	PC04	P	4	•	•	6/0	•	•	0/0	•
69	PC05	P	5	•	•	6/1	•	•	•	•
70	PC06	P	6	•	•	6/2	•	•	•	•
71	PC07	P	9	•	•	6/3	•	•	•	•
74	PC10	P	10	•	•	6/2	7/2	•	0/0	1/4

continues on next page

Table 15 – continued from previous page

Pin	GPIO	Name/P	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
75	PC11	P	11	•	•	6/3	7/3	•	0/1	1/5
76	PC12	P	12	•	•	7/0	6/1	•	0/2	1/6
77	PC13	P	13	•	•	7/1	6/0	•	0/3	1/7
78	PC14	P	14	•	•	7/2	6/2	•	0/4	1/0
79	PC15	P	15	•	•	7/3	6/3	•	0/5	1/1
80	PC16	P	0	•	•	6/0	0/1	•	0/0	•
81	PC17	P	1	•	•	6/1	0/0	•	0/1	•
82	PC18	P	2	•	•	6/2	0/2	•	0/2	•
83	PC19	P	3	•	•	6/3	0/3	•	0/3	•
84	PC20	P	4	•	•	•	•	•	0/4	•
85	PC21	P	5	•	•	•	•	•	0/5	•
86	PC22	P	6	•	•	1/0	3/1	•	0/5	•
87	PC23	P	7	•	•	1/1	3/0	•	0/7	•
88	PC24	P	8	•	•	0/2	2/2	•	•	•
89	PC25	P	9	•	•	0/3	2/3	•	•	•
90	PC26	P	10	•	•	•	•	•	•	•
91	PC27	P	11	•	•	1/0	•	•	•	•
92	PC28	P	12	•	•	1/1	•	•	•	•
94	PC30	P	14	•	12	•	•	•	•	•
95	PC31	P	15	•	13	•	•	•	•	•
96	PD00	P	0	•	14	•	•	•	•	•

continues on next page

Table 15 – continued from previous page

Pin	GPIO	Name/P	IRQ	ADC	ADC	Serial	Serial	TC	PWM	PWM
97	PD01	P	1	.	15	.	.	.	.	.
104	PD08	P	3	.	.	7/0	6/1	.	0/1	.
105	PD09	P	4	.	.	7/1	6/0	.	0/2	.
106	PD10	P	5	.	.	7/2	6/2	.	0/3	.
107	PD11	P	6	.	.	7/3	6/3	.	0/4	.
108	PD12	P	7	.	.	.	.	.	0/5	.
116	PD20	P	10	.	.	1/2	3/2	.	1/0	.
117	PD21	P	11	.	.	1/3	3/3	.	1/1	.

The Package column indicates the package letter providing this pin. An entry GJP tells for instance, that the pin is available for SAMD51G19, SAMD51J19/-J20 and SAMD51P19/-P20.

16.3.18 Scripts for creating the pin assignment tables

The tables shown above were created with small a Python script running on the target board:

```
from samd import pininfo
from machine import Pin
import os

def print_item(e, txt):
    print(txt, end=": ")
    if e == 255:
        print(" - ", end="")
    else:
        print("%d/%d" % (e >> 4, e & 0x0f), end="")

def print_pininfo(pin_id, name, info):

    print("%3d" % pin_id, end=" ")
    print("%4s %12s" % (info[0], name), end="")
    print(" IRQ:%2s" % (info[1] if info[1] != 255 else "-"), end="")
    print(" ADC0:%2s" % (info[2] if info[2] != 255 else "-"), end="")
    if len(info) == 7:
        print_item(info[3], " Serial1")
        print_item(info[4], " Serial2")
        print_item(info[5], " PWM1" if (info[5] >> 4) < 3 else "   TC")
        print_item(info[6], " PWM2")
    else:
        print(" ADC1:%2s" % (info[3] if info[3] != 255 else "-"), end="")
```

(continues on next page)

(continued from previous page)

```
        print_item(info[4], " Serial1")
        print_item(info[5], " Serial2")
        print_item(info[6], " TC")
        print_item(info[7], " PWM1")
        print_item(info[8], " PWM2")
    print()

def tblkey(i):
    name = i[1]
    if name != "":
        if len(name) < 3:
            return " " + name
        else:
            return name
    else:
        return "zzzzzzz%03d" % i[0]

def pinnum(p):
    return (ord(p[1]) - ord("A")) * 32 + int(p[2:])

def table(num = 127, sort=True):
    pintbl = []
    pinlist = []
    for name in Pin.board.__dict__.keys():
        p = Pin(name)
        pi = pininfo(p)
        pintbl.append((pinnum(pi[0]), name, pi))
        pinlist.append(p)
    for pc in Pin.cpu.__dict__.keys():
        p = Pin(pc)
        pi = pininfo(p)
        if not p in pinlist:
           pintbl.append((pinnum(pi[0]), "", pi))
    if sort:
        pintbl.sort(key=tblkey)
    for item in pintbl:
        print_pininfo(item[0], item[1], item[2])

table()
```

16.4 Installing MicroPython

See the corresponding section of tutorial: *Getting started with MicroPython on the SAMD*. It also includes a troubleshooting subsection.

16.5 General board control

The MicroPython REPL is on the USB port, configured in VCP mode. Tab-completion is useful to find out what methods an object has. Paste mode (Ctrl-E) is useful to paste a large slab of Python code into the REPL.

The *machine* module:

```
import machine

machine.freq()             # get the current frequency of the CPU
machine.freq(96_000_000)   # set the CPU frequency to 96 MHz
```

The range accepted by the function call is 1_000_000 to 200_000_000 (1 MHz to 200 MHz) for SAMD51 and 1_000_000 to 54_000_000 (1 MHz to 54 MHz) for SAMD21. The safe range for SAMD51 according to the data sheet is up to 120 MHz, for the SAMD21 up to 48Mhz. Frequencies below 48Mhz are set by dividing 48Mhz by an integer, limiting the number of discrete frequencies to 24Mhz, 16Mhz, 12MHz, and so on. At frequencies below 8 MHz USB will be disabled. Changing the frequency below 48 MHz impacts the baud rates of UART, I2C and SPI. These have to be set again after changing the CPU frequency. The ms and µs timers are not affected by the frequency change.

16.6 Delay and timing

Use the *time* module:

```
import time

time.sleep(1)           # sleep for 1 second
time.sleep_ms(500)      # sleep for 500 milliseconds
time.sleep_us(10)       # sleep for 10 microseconds
start = time.ticks_ms() # get millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start) # compute time difference
```

Note that *time.sleep_us()* delays by busy waiting. During that time, other tasks are not scheduled.

16.7 Clock and time

Two groups of functions are provided for time information. All boards have the datetime(), mktime() and time() functions. Boards with a 32kHz crystal also provide an RTC() module. The epoch start date is 1.1.2000.

Use the *time* module:

```
import time

date_time = time.localtime()        # Show the actual date/time information
date_time = time.localtime(seconds) # decode the date/time form the seconds value
seconds = time.mktime(date_time_tuple) # Convert seconds to a datetime tuple
second = time.time()                # Return the actual system time.
```

The format of the date_time tuple follows the standard. The µs value of the date_time tuple is ignored. On boards without the RTC module, time.localtime(seconds) sets the system time. Use of the Use the `RTC` module:

```
from machine import RTC
rtc = RTC()
```

(continues on next page)

(continued from previous page)

```
date_time = rtc.datetime()       # return the actual date & time.
rtc.datetime(date_time_tuple)    # Set date & time, ignoring weekday
date_time = rtc.now()            # Return date & time in Unix order.
rtc.calibration(value)           # Set a calibration factor
```

The weekday value set will be ignored and calculated in the returned tuple from the actual date. rtc.now() is only provided at SAMD51 boards. The value used in the rtc.calibration() call has a range from -127 - 127. It defines roughly a ppm quantity, by which the clock can run faster or slower.

16.8 Timers

The SAMD21/SAMD51 uses software timers. Use the *machine.Timer* class:

```
from machine import Timer

tim0 = Timer()
tim0.init(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(0))

tim1 = Timer()
tim1.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(1))
```

The period is in milliseconds.

16.9 Pins and GPIO

Use the *machine.Pin* class:

```
from machine import Pin

p0 = Pin('D0', Pin.OUT) # create output pin on GPIO0
p0.on()                 # set pin to "on" (high) level
p0.off()                # set pin to "off" (low) level
p0.value(1)             # set pin to on/high

p2 = Pin('D2', Pin.IN)  # create input pin on GPIO2
print(p2.value())       # get value, 0 or 1

p4 = Pin('D4', Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor
p7 = Pin('PA07', Pin.OUT, value=1) # set pin high on creation
```

Pins can be denoted by a string or a number. The string is either the pin label of the respective board, like “D0” or “SDA”, or in the form “Pxnn”, where x is A,B,C or D, and nn a two digit number in the range 0-31. Examples: “PA03”, PD31”.

Pin numbers are the MCU port numbers in the range:

```
PA0..PA31:  0..31
PB0..PB31: 32..63
PC0..PC31: 64..95
PD0..PD31: 96..127
```

Note: On Adafruit Feather and ItsyBity boards, pin D5 is connected to an external gate output and can therefore only be used as input.

16.10 UART (serial bus)

See *machine.UART*.

```
# Use UART 3 on a ItsyBitsy M4 board
from machine import UART

uart3 = UART(3, tx=Pin('D1'), rx=Pin('D0'), baudrate=115200)
uart3.write('hello')  # write 5 bytes
uart3.read(5)         # read up to 5 bytes

uart = UART()         # Use the default values for id, rx and tx.
uart = UART(baudrate=9600) # Use the default UART and set the baudrate
```

The SAMD21/SAMD51 MCUs have up to eight hardware so called SERCOM devices, which can be used as UART, SPI or I2C device, but not every MCU variant and board exposes all TX and RX pins for users. For the assignment of Pins to devices and UART signals, refer to the *SAMD pinout*. If the id, rx or tx pins are not specified, the default values are used. The first positional argument (if given) is assumed to be the UART id. If the baudrate is changed and the UART id is omitted, it must be set using the baudrate keyword.

16.11 PWM (pulse width modulation)

Up to five timer device of the SAMD21/SAMD51 MCUs are used for creating PWM signals.

The PWM functions are provided by the *machine.PWM* class. It supports all basic methods listed for that class.

```
# Samples for Adafruit ItsyBitsy M4 Express

from machine import Pin, PWM

# create PWM object from a pin and set the frequency and duty cycle
pwm = PWM(Pin('D7'), freq=2000, duty_u16=32768)
pwm.freq()              # get the current frequency
pwm.freq(1000)          # set/change the frequency
pwm.duty_u16()          # get the current duty cycle, range 0-65535
pwm.duty_u16(200)       # set the duty cycle, range 0-65535
pwm.deinit()            # turn off PWM on the pin

pwm                     # show the PWM objects properties
```

16.11.1 PWM Constructor

class PWM(*dest*, ***, *freq*, *duty_u16*, *duty_ns*, *invert*, *device*)

Construct and return a new PWM object using the following parameters:

- *dest* is the Pin object on which the PWM is output.

PWM objects are provided by TCC timer module. The TCC timer modules have up to six channels and eight outputs. All channels of a module run at the same frequency, but allow for different duty cycles. Outputs are assigned to channels in modulo-n fashion, where n is the number of channels. Outputs of a channel have the same frequency and duty rate, but may have different polarity. So if for instance a module has four channels, output 0 and 4, 1 and 5, 2 and 6, 3, and 7 share the same frequency and duty rate.

Only one of *duty_u16* and *duty_ns* should be specified at a time.

Keyword arguments:

- *freq* should be an integer which sets the frequency in Hz for the PWM cycle. The valid frequency range is 1 Hz to 24 MHz.
- *duty_u16* sets the duty cycle as a ratio `duty_u16 / 65535`.
- *duty_ns* sets the pulse width in nanoseconds. The limitation for X channels apply as well.
- *invert*=True|False. Setting a bit inverts the respective output.
- *device*=n Use TCC module n if available. At some pins two TCC modules could be used. If not device is mentioned, the software tries to use a module which is not yet used for a PWM signal. But if pins shall have the same frequency and/or duty cycle to be changed synchronously, they must be driven by the same TCC module.

16.11.2 PWM Methods

The methods are identical to the generic *machine.PWM* class, with additional keyword arguments to the init() method, matchings those of the constructor.

16.11.3 PWM Pin Assignment

Pins are specified in the same way as for the Pin class. For the assignment of Pins to PWM signals, refer to the *SAMD pinout*.

16.12 ADC (analog to digital conversion)

On the SAMD21/SAMD51 ADC functionality is available on Pins labelled 'Ann'.

Use the *machine.ADC* class:

```
from machine import ADC

adc0 = ADC(Pin('A0'))             # create ADC object on ADC pin, average=16
adc0.read_u16()                   # read value, 0-65535 across voltage range 0.0v - 3.
↪3v
adc1 = ADC(Pin('A1'), average=1) # create ADC object on ADC pin, average=1
```

The resolution of the ADC is 12 bit with 12 bit accuracy, irrespective of the value returned by read_u16(). If you need a higher resolution or better accuracy, use an external ADC.

16.12.1 ADC Constructor

class ADC(*dest*, *, *average=16*, *bits=12*, *vref=ADC.AREF*, *callback=None*)

On the SAMD21/SAMD51 ADC functionality is available on Pins labelled 'Ann'.

Use the *machine.ADC* class:

```
from machine import ADC

adc0 = ADC(Pin("A0"))             # create ADC object on ADC pin, average=16
adc0.read_u16()                   # read value, 0-65536 across voltage range 0.0v - 3.
↪3v
adc1 = ADC(Pin("A1"), average=1) # create ADC object on ADC pin, average=1
```

The resolution of the ADC is set by the bits keyword option. The default is 12. Suitable values are 8, 10 and 12. If you need a higher resolution or better accuracy, use an external ADC. The default value of average is 16. Averaging is used to reduce the noise. With a value of 16 the LSB noise is about 1 digit. When averaging is enabled, the resolution is forced to 12 bits. The vref=n option sets the reference voltage for the ADC. The default setting is for 3.3V. Other values are:

Symbol	Value	SAMD21	SAMD51
INT_VREF	0	1.0V voltage reference	1V internal bandgap reference
VDDA	1	1/1.48 Analogue voltage supply	Analogue voltage supply
VDDA2	2	1/2 Analogue voltage supply	1/2 Analogue voltage supply
AREF	3	External reference A (PA03)	External reference A (PA03)
AREFB	4	External reference B (PA04)	External reference B (PA04)
AREFC	5		External reference C (PA06)

The callback keyword option is used for timed ADC sampling. The callback is executed when all data has been sampled.

16.12.2 ADC Methods

read_u16()

Read a single ADC value as unsigned 16 bit quantity. The voltage range is defined by the vref option of the constructor, the resolutions by the bits option.

read_timed(*data*, *freq*)

Read ADC values into the data buffer at a supplied frequency. The buffer must be preallocated. Values are stored as 16 bit quantities in the binary range given by the bits option. If bits=12, the value range is 0-4095. The voltage range is defined by the vref option. The sampling frequency range depends on the bits and average setting. At bits=8 and average=1, the largest rate is >1 MHz for SAMD51 and 350kHz for SAMD21. the lowest sampling rate is 1 Hz. The call to the method returns immediately, The data transfer is done by DMA in the background, controlled by a hardware timer. If in the constructor a callback was defined, it will be called after all data has been read. Alternatively, the method busy() can be used to tell, if the capture has finished.

Example for a call to adc.read_timed() and a callback:

```
from machine import ADC
from array import array

def finished(adc_o):
    print("Sampling finished on ADC", adc_o)

# create ADC object on ADC pin A0, average=1
adc = ADC(Pin("A0"), average=1, callback=finished)
buffer = array("H", bytearray(512))  # create an array for 256 ADC values
adc.read_timed(buffer, 10000)        # read 256 12 bit values at a frequency of
                                     # 10 kHz and call finished() when done.
```

busy()

busy() returns `True` while the data acquisition using read_timed() is ongoing, `False` otherwise.

Deinitialize an ADC object and release the resources used by it, especially the ADC channel and the timer used for read_timed().

16.13 DAC (digital to analogue conversion)

16.13.1 DAC Constructor

class DAC(*id*, *, *vref=3*, *callback=None*)

The DAC class provides a fast digital to analogue conversion. Usage example:

```
from machine import DAC

dac0 = DAC(0)             # create DAC object on DAC pin A0
dac0.write(1023)          # write value, 0-4095 across voltage range 0.0V - 3.3V
dac1 = DAC(1)             # create DAC object on DAC pin A1
dac1.write(2000)          # write value, 0-4095 across voltage range 0.0V - 3.3V
```

The resolution of the DAC is 12 bit for SAMD51 and 10 bit for SAMD21. SAMD21 devices have 1 DAC channel at GPIO PA02, accepting only 0 as id. SAMD51 devices have 2 DAC channels at GPIO PA02 and PA05 with values 0 and 1 for the id. The vref arguments defines the output voltage range, the callback option is used for dac_timed(). Suitable values for vref are:

Symbol	Value	SAMD21	SAMD51
INT_VREF	0	Internal voltage reference	Internal bandgap reference (~1V)
VDDA	1	Analogue voltage supply	Analogue voltage supply
AREF	2	External reference	Unbuffered external reference
AREFB	3		Buffered external reference

16.13.2 DAC Methods

write(*value*)

Write a single value to the selected DAC output. The value range is 0-1023 for SAMD21 and 0-4095 for SAMD51. The voltage range depends on the vref setting.

write_timed(*data*, *freq*[, *count=1*])

The call to dac_timed() allows to output a series of analogue values at a given rate. data must be a buffer with 16 bit values in the range of the DAC (10 bit of 12 bit). freq may have a range of 1Hz to ~200kHz for SAMD21 and 1 Hz to ~500kHz for SAMD51. The optional argument count specifies, how often data output will be repeated. The range is 1 - (2**32 - 1). If count == 0, the data output will be repeated until stopped by a call to deinit(). If the data has been output count times, a callback will be called, if given.

Example:

```
from machine import DAC
from array import array

data = array("H", [i for i in range(0, 4096, 256)]) # create a step sequence

def done(dac_o):
    print("Sequence done at", dac_o)

dac = DAC(0, callback=done)
dac.write_timed(data, 1000, 10) # output data 10 times at a rate of 1000 values/s
                                # and call done() when finished.
```

The data transfer is done by DMA and not affected by python code execution. It is possible to restart dac.write_timed() in the callback function with changed parameters.

busy()

Tell, whether a write_timed() activity is ongoing. It returns `True` if yes, `False` otherwise.

deinit()

Deinitialize the DAC and release the resources used by it, especially the DMA channel and the Timer. On most SAMD21 boards, there is just one timer available for dac.write_timed() and adc.read_timed(). So they cannot run both at the same time, and releasing the timer may be important. The DAC driver consumes a substantial amount of current. deinit() will reduce that as well. After calling deinit(), the DAC objects cannot be used any more and must be recreated.

16.14 Software SPI bus

Software SPI (using bit-banging) works on all pins, and is accessed via the *machine.SoftSPI* class.

```
from machine import Pin, SoftSPI

# construct a SoftSPI bus on the given pins
# polarity is the idle state of SCK
# phase=0 means sample on the first edge of SCK, phase=1 means the second
spi = SoftSPI(baudrate=100000, polarity=1, phase=0, sck=Pin('D7'), mosi=Pin('D9'),␣
↪miso=Pin('D10'))

spi.init(baudrate=200000) # set the baud rate

spi.read(10)            # read 10 bytes on MISO
spi.read(10, 0xff)      # read 10 bytes while outputting 0xff on MOSI

buf = bytearray(50)     # create a buffer
spi.readinto(buf)       # read into the given buffer (reads 50 bytes in this case)
spi.readinto(buf, 0xff) # read into the given buffer and output 0xff on MOSI

spi.write(b'12345')     # write 5 bytes on MOSI

buf = bytearray(4)      # create a buffer
spi.write_readinto(b'1234', buf) # write to MOSI and read from MISO into the buffer
spi.write_readinto(buf, buf) # write buf to MOSI and read MISO back into buf
```

The highest supported baud rate is 500000.

16.15 Hardware SPI bus

The SAMD21/SAMD51 MCUs have up to eight hardware so called SERCOM devices, which can be used as UART, SPI or I2C device, but not every MCU variant and board exposes all signal pins for users. Hardware SPI is accessed via the *machine.SPI* class and has the same methods as software SPI above:

```
from machine import SPI

spi = SPI(1, sck=Pin("SCK"), mosi=Pin("MOSI"), miso=Pin("MISO"), baudrate=10000000)
spi.write('Hello World')
```

For the assignment of Pins to SPI devices and signals, refer to *SAMD pinout*. If the id, miso, mosi or sck pins are not specified, the default values are used. So it is possible to create the SPI object as:

```
from machine import SPI
spi = SPI()  # Use the default device and default baudrate
spi = SPI(baudrate=12_000_000)  # Use the default device and change the baudrate
```

If the MISO signal shall be omitted, it must be defined as miso=None. The first positional argument (if given) is assumed to be the SPI id. If the baudrate is changed while the SPI id is omitted, it must be set using the baudrate keyword.

Note: Even if the highest reliable baud rate at the moment is about 24 Mhz, setting a baud rate will not always result in exactly that frequency, especially at high baud rates.

16.16 Software I2C bus

Software I2C (using bit-banging) works on all output-capable pins, and is accessed via the *machine.SoftI2C* class:

```
from machine import Pin, SoftI2C

i2c = SoftI2C(scl=Pin('D10'), sda=Pin('D11'), freq=100000)

i2c.scan()              # scan for devices

i2c.readfrom(0x3a, 4)   # read 4 bytes from device with address 0x3a
i2c.writeto(0x3a, '12') # write '12' to device with address 0x3a

buf = bytearray(10)     # create a buffer with 10 bytes
i2c.writeto(0x3a, buf)  # write the given buffer to the slave
```

The highest supported freq is 400000.

16.17 Hardware I2C bus

The SAMD21/SAMD51 MCUs have up to eight hardware so called SERCOM devices, which can be used as UART, SPI or I2C device, but not every MCU variant and board exposes all signal pins for users. For the assignment of Pins to devices and I2C signals, refer to *SAMD pinout*. If the id, scl or sda pins are not specified, the default values are used.

Hardware I2C is accessed via the *machine.I2C* class and has the same methods as software SPI above:

```
from machine import I2C
```

(continues on next page)

(continued from previous page)

```
i2c = I2C(2, scl=Pin("SCL"), sda=Pin("SDA"), freq=400_000)
i2c.writeto(0x76, b"Hello World")

i2c2 = I2C()   # Use the default values for id, scl and sda.
i2c2 = I2C(freq=100_000) # Use the default device and set freq.
```

The first positional argument (if given) is assumed to be the I2C id. If the freq is changed and the I2C id is omitted, it must be set using the freq keyword.

16.18 OneWire driver

The OneWire driver is implemented in software and works on all pins:

```
from machine import Pin
import onewire

ow = onewire.OneWire(Pin('D12')) # create a OneWire bus on GPIO12
ow.scan()                        # return a list of devices on the bus
ow.reset()                       # reset the bus
ow.readbyte()                    # read a byte
ow.writebyte(0x12)               # write a byte on the bus
ow.write('123')                  # write bytes on the bus
ow.select_rom(b'12345678')       # select a specific device by its ROM code
```

There is a specific driver for DS18S20 and DS18B20 devices:

```
import time, ds18x20
ds = ds18x20.DS18X20(ow)
roms = ds.scan()
ds.convert_temp()
time.sleep_ms(750)
for rom in roms:
    print(ds.read_temp(rom))
```

Be sure to put a 4.7k pull-up resistor on the data line. Note that the `convert_temp()` method must be called each time you want to sample the temperature.

16.19 DHT driver

The DHT driver is implemented in software and works on all pins:

```
import dht
import machine

d = dht.DHT11(machine.Pin('D4'))
d.measure()
d.temperature() # eg. 23 (°C)
d.humidity()    # eg. 41 (% RH)

d = dht.DHT22(machine.Pin('D4'))
d.measure()
```

(continues on next page)

(continued from previous page)

```
d.temperature() # eg. 23.6 (°C)
d.humidity()    # eg. 41.3 (% RH)
```

Be sure to have a 4.7k pull-up resistor on the data line. Some DHT modules may already have one.

16.20 Driving an APA102 LED

The APA102 on some Adafruit boards can be controlled using SoftSPI:

```
from machine import SoftSPI, Pin
# create the SPI object. miso can be any unused pin.
spi=SoftSPI(sck=Pin('D25'), mosi=Pin('D26'), miso=Pin('D14'))

# define a little function that writes the data with
# preamble and postfix
def write(red, green, blue):
    spi.write(b"\x00\x00\x00\x00\xff")
    spi.write(bytearray((blue, green, red)))
    spi.write(b"\xff\xff\xff")

# set the LED to red
write(128, 0, 0)
```

Since SoftSPI does not allow miso to be undefined, miso has to be assigned to an otherwise unused pin.

16.21 Driving a Neopixel LED

The built-in machine.bitstream() method supports driving Neopixel LEDs in combination with the Neopixel driver from the MicroPython driver library:

```
import neopixel
import machine

# 1 LED connected to Pin D8 on Adafruit Feather boards
p = machine.Pin('D8', machine.Pin.OUT)
n = neopixel.NeoPixel(p, 1)

# set the led to red.
n[0] = (128, 0, 0)

# Update the LED.
n.write()
```

machine.bitstream() is set up for a SAMD21 clock frequency of 48MHz and a SAMD51 clock frequency of 120 MHz. At other clock frequencies, the timing will not fit.

16.22 Transferring files

Files can be transferred to the SAMD21/SAMD51 devices for instance with the *mpremote* tool. See the MicroPython forum for community-supported alternatives to transfer files to an SAMD21/SAMD51 board, like rshell or Thonny.

PYTHON MODULE INDEX

_

_thread, 61

a

aioespnow, 259
array, 2
asyncio, 3

b

binascii, 8
bluetooth, 61
btree, 73

c

cmath, 13
collections, 14
cryptolib, 75

d

deflate, 76

e

errno, 15
esp, 237
esp32, 239
espnow, 248

f

framebuf, 78

g

gc, 16
gzip, 18

h

hashlib, 19
heapq, 20

i

io, 21

j

json, 25

l

lcd160cr, 226

m

machine, 81
marshal, 25
math, 26
micropython, 142
mimxrt, 263

n

neopixel, 146
network, 147

o

openamp, 164
os, 28

p

platform, 31
pyb, 181

r

random, 32
re, 33
rp2, 264

s

select, 36
socket, 38
ssl, 43
stm, 224
string.templatelib, 46
struct, 50
sys, 52

t

time, 55

U

uctypes, 166

V

vfs, 171

W

weakref, 58
wipy, 232

Z

zephyr, 278
zlib, 59
zsensor, 281