MICROPYTHON

INDEX:

1. Introduction

MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library and is optimised to run on microcontrollers and in constrained environments

MicroPython strives to be as compatible as possible with normal Python (known as CPython) so that if you know Python you already know MicroPython. On the other hand, the more you learn about MicroPython the better you become at Python.

In addition to implementing a selection of core Python libraries, MicroPython includes modules such as "machine" for accessing low-level hardware.

2. Micropython libraries

MicroPython is a full Python compiler and runtime that runs on the bare-metal. You get an interactive prompt (the REPL) to execute commands immediately, along with the ability to run and import scripts from the built-in filesystem. The REPL has history, tab completion, auto-indent and paste mode for a great user experience.

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

```
MicroPython v1.23.0-preview.379.gcfd5a8ea3 on 2024-05-23;
NUCLEO-F401RE with STM32F401xE

Type "help()" for more information.
>>> help('modules')
__main__          builtins        json            select
_asyncio          cmath           machine         socket
_onewire          collections     math            stm
array             deflate         micropython     struct
asyncio/__init__  dht             network         sys
asyncio/core      errno           onewire         time
asyncio/event     framebuf        os              uasyncio
asyncio/funcs     gc              platform        uctypes
asyncio/lock      hashlib         pyb             vfs
asyncio/stream    heapq           random
```

binascii	io	re
Plus any modules on the filesystem		

3. 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.

3.1 array – arrays of numeric data

This module implements a subset of the corresponding CPython module

3.1.1 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.

methods	usage
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(, [])", where is the type code letter for the array and 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.

3.1.2 Example code-1

```
import array

# Create an array with typecode 'i' (signed integer) and initial elements
a = array.array('i', [1, 2, 3, 4, 5])
print("Initial array:", a)

# Append a new element to the array
a.append(6)
print("Array after append:", a)

# Extend the array with another array (not a list)
a.extend(array.array('i', [7, 8, 9]))
print("Array after extend:", a)

# Indexed read (__getitem__)
print("Element at index 2:", a[2])
print("Elements from index 2 to 5:", a[2:6])

# Indexed write (__setitem__)
a[3] = 10
print("Array after setting index 3 to 10:", a)
a[2:4] = array.array('i', [11, 12])
print("Array after setting slice 2:4 to [11, 12]:", a)

# Get the length of the array (__len__)
print("Length of the array:", len(a))

# Concatenate two arrays (__add__)
```

```
b = array.array('i', [13, 14, 15])
c = a + b
print("Array after concatenation:", c)

# In-place concatenation (__iadd__)
a += array.array('i', [16, 17])
print("Array after in-place concatenation:", a)

# String representation (__repr__)
print("String representation of the array:", repr(a))
```

Output:

Initial array: array('i', [1, 2, 3, 4, 5])

Array after append: array('i', [1, 2, 3, 4, 5, 6])

Array after extend: array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9])

Element at index 2: 3

Elements from index 2 to 5: array('i', [3, 4, 5, 6])

Array after setting index 3 to 10: array('i', [1, 2, 3, 10, 5, 6, 7, 8, 9])

Array after setting slice 2:4 to [11, 12]: array('i', [1, 2, 11, 12, 5, 6, 7, 8, 9])

Length of the array: 9

Array after concatenation: array('i', [1, 2, 11, 12, 5, 6, 7, 8, 9, 13, 14, 15])

Array after in-place concatenation: array('i', [1, 2, 11, 12, 5, 6, 7, 8, 9, 16, 17])

String representation of the array: array('i', [1, 2, 11, 12, 5, 6, 7, 8, 9, 16, 17])

3.2 asyncio — asynchronous I/O schedule

The asyncio module in Python provides a framework for writing single-threaded concurrent code using coroutines, making it particularly well-suited for I/O-bound and high-level structured network code. It allows for the scheduling of asynchronous tasks and cooperative multitasking.

3.2.1 uasyncio

it is a MicroPython module that provides support for asynchronous programming, allowing you to run multiple tasks concurrently without blocking the execution of other tasks. Here's an explanation of some of its key methods:

1. **create_task(coro):**

   - This method creates a new task from the given coroutine coro.
   - It schedules the task to run asynchronously.
   - Returns the corresponding Task object.

2. **sleep(t):**

   - This coroutine function suspends the execution of the current task for t seconds (which can be a float).
   - It allows other tasks to run concurrently during the sleep period.
   - After the sleep duration, the task resumes execution.

3. **gather(*awaitables, return_exceptions=False):**

   - This coroutine function runs all the given awaitables concurrently.
   - Any awaitables that are not tasks are promoted to tasks internally.
   - Returns a list of return values of all the awaitables.
   - The gather function in uasyncio (and asyncio in Python) is used to run multiple coroutines concurrently and wait for all of them to complete. Here's why we use gather:

4. **run(coro) :**

   - In uasyncio, the run() function is used to start the event loop and keep it running until the program terminates or until the loop is explicitly stopped

5. **cancel():**

   - The cancel() method in uasyncio is used to cancel a running task. When you call cancel() on a task, it raises a CancelledError inside the corresponding coroutine, causing it to exit early

3.2.2 Difference between uasyncio and asyncio

The primary difference between uasyncio and asyncio lies in the platforms they support:

1. **uasyncio**:
   - **Platform**: MicroPython
   - **Use**: Designed specifically for constrained environments like microcontrollers running MicroPython. It's a lightweight version of asyncio tailored for resource-constrained devices.
2. **asyncio**:
   - **Platform**: CPython (standard Python implementation)
   - **Use**: Standard asynchronous I/O library for Python. It's designed for general-purpose asynchronous programming on standard computing platforms.

3.2.3 Example code-2

```
import uasyncio

# Task 1: Define a coroutine
```

```python
async def coroutine_task():
    print("Task 1 is running...")
    await uasyncio.sleep(1)
    print("Task 1 completed")

# Task 2: Define another coroutine
async def another_coroutine_task():
    print("Task 2 is running...")
    await uasyncio.sleep(2)
    print("Task 2 completed")

# Define a function to run the event loop
async def main():
    # Create tasks
    task1 = uasyncio.create_task(coroutine_task())
    task2 = uasyncio.create_task(another_coroutine_task())

    # Run tasks concurrently
    await uasyncio.gather(task1, task2)

# Run the event loop continuously
while True:
    uasyncio.run(main())
```

Output:

Task 1 is running...

Task 2 is running...

Task 1 completed

Task 2 completed

Task 1 is running...

Task 2 is running...

Task 1 completed

Task 2 completed

Task 1 is running...

Task 2 is running...

Task 1 completed

Task 2 completed

Task 1 is running...

Task 2 is running...

Aborted

3.2.4 Example code-3

```python
import uasyncio


# Task 1: Fetch data
async def fetch_data(task):
    while True:
        print(f"current task :{task}")
        print("task1 is running..")
        await uasyncio.sleep(2)


# Task 2: Print numbers
async def print_numbers(task):
    i = 0
    while True:
        print(f"current task :{task}")
        print("task2 is running..")
        print(f"Number: {i}")
        i += 1
        await uasyncio.sleep(2)


# Task 3: Toggle an LED
async def toggle_led(task):
    # Replace this with actual code to toggle LED on a regular Python 
environment
    from machine import Pin
    led = Pin('PA5', Pin.OUT)
    while True:
        print(f"current task :{task}")
        print("task3 is running...")
        led.value(not led.value())
        print("Task toggle_led toggled LED")
        await uasyncio.sleep(2)




async def main():
    # Create and start tasks
    task1 = uasyncio.create_task(fetch_data(uasyncio.current_task()))
    task2 = uasyncio.create_task(print_numbers(uasyncio.current_task()))
    task3 = uasyncio.create_task(toggle_led(uasyncio.current_task()))

    # Wait for some time
    await uasyncio.sleep(5)

    # Cancel task2
    task1.cancel()
    print("task1 is cancelled")

    # Run tasks indefinitely
    await uasyncio.gather(task1, task2, task3, return_exceptions=True)  # Use 
return_exceptions to suppress cancellation exceptions



# Run the main coroutine
uasyncio.run(main())
```

Output

current task :<Task>

task1 is running..

current task :<Task>

task2 is running..

Number: 0

current task :<Task>

task3 is running...

Task toggle_led toggled LED

current task :<Task>

task1 is running..

current task :<Task>

task2 is running..

Number: 1

current task :<Task>

task3 is running...

Task toggle_led toggled LED

current task :<Task>

task1 is running..

current task :<Task>

task2 is running..

Number: 2

current task :<Task>

task3 is running...

Task toggle_led toggled LED

task1 is cancelled

current task :<Task>

task2 is running..

Number: 3

current task :<Task>

task3 is running...

Task toggle_led toggled LED

current task :<Task>

task2 is running..

Number: 4

current task :<Task>

task3 is running...

Task toggle_led toggled LED

current task :<Task>

task2 is running..

Number: 5

current task :<Task>

task3 is running...

Task toggle_led toggled LED

current task :<Task>

task2 is running..

Number: 6

current task :<Task>

task3 is running...

Task toggle_led toggled LED

3.3 class Event

It allows synchronization between coroutines by signaling events between them. Here's an overview of the uasyncio.Event class and its methods:

3.3.1 uasyncio.Event Class

Constructor:

- `uasyncio.Event()`: Creates a new event object. The event is initially cleared (unset).

method	usage
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

3.3.2 Example code-4

```
import uasyncio as asyncio
from machine import Pin  # Assuming Pin is imported correctly for your
hardware setup


async def toggle_led(event):
    print("Waiting for the event to be set...")
    await event.wait()
    print("Event is set. Resuming execution.")
    led = Pin('PA5', Pin.OUT)  # Adjust Pin and setup as per your hardware
    try:
        while True:
            if event.is_set():
                led.value(not led.value())  # Toggle LED
                print("LED toggled.")
            await asyncio.sleep(1)
    except asyncio.CancelledError:
        print("Task cancelled and event cleared")
        event.clear()
        print("the event after cleared returns : ",event.is_set())

async def cancel_task(task, delay):
    await asyncio.sleep(delay)
    task.cancel()


async def main():
    # Create a new event
    event = asyncio.Event()

    # Start the LED toggle task
    task2 = asyncio.create_task(toggle_led(event))

    # Start the cancellation task
    asyncio.create_task(cancel_task(task2, 5))  # Cancel after 5 seconds

    # Wait for some time
    await asyncio.sleep(2)

    # Set the event
```

```
    print("Setting the event...")
    event.set()

    # Wait for the toggle_led task to finish
    await task2



# Run the main coroutine
asyncio.run(main())
```

Output:

Waiting for the event to be set...

Setting the event...

Event is set. Resuming execution.

LED toggled.

LED toggled.

LED toggled.

Task cancelled and event cleared

the event after cleared returns : False

3.4 class ThreadSafeFlag

In MicroPython's `uasyncio` module, the `ThreadSafeFlag` class provides a synchronization mechanism similar to asyncio's `asyncio.Event`, but it's designed to work in scenarios where code outside the asyncio loop, such as other threads, interrupts, or scheduler callbacks, needs to interact with asyncio tasks.

Here's an explanation of the `uasyncio.ThreadSafeFlag` class and its methods:

3.4.1 uasyncio.ThreadSafeFlag Class

Constructor:

- `uasyncio.ThreadSafeFlag()`: Creates a new `ThreadSafeFlag` object. The flag is initially in the cleared state.

method	usage
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

3.4.2 Example code-5

```
import uasyncio as asyncio


async def waiter(flag):
    print("Waiting for the flag to be set...")
    await flag.wait()
    print("Flag is set. Resuming execution.")

async def main():
    # Create a new ThreadSafeFlag
    flag = asyncio.ThreadSafeFlag()

    # Start the waiter task
    task1 = asyncio.create_task(waiter(flag))

    # Wait for some time
    await asyncio.sleep(2)

    # Set the flag
    print("Setting the flag...")
    flag.set()

    # Wait for the waiter task to finish
    await task1

# Run the main coroutine
asyncio.run(main())
```

Output:

Waiting for the flag to be set...

Setting the flag...

Flag is set. Resuming execution.

3.5 class Lock

3.5.1 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.

method	usage
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

3.5.2 Example code-6

```
import asyncio

# Define a shared resource
shared_resource = 0

# Define a coroutine to increment the shared resource
async def increment(lock):
    global shared_resource
    print("Trying to acquire the lock to increment...")
    await lock.acquire()
    print("Lock acquired to increment.")
    shared_resource += 1
    await asyncio.sleep(1)  # Simulate some work
    print("Shared resource incremented to:", shared_resource)
    lock.release()
    print("Lock released after increment.")

# Define a coroutine to decrement the shared resource
async def decrement(lock):
    global shared_resource
    print("Trying to acquire the lock to decrement...")
    await lock.acquire()
    print("Lock acquired to decrement.")
    shared_resource -= 1
    await asyncio.sleep(1)  # Simulate some work
    print("Shared resource decremented to:", shared_resource)
    lock.release()
    print("Lock released after decrement.")

async def main():
    # Create a lock
```

```
    lock = asyncio.Lock()

    # Run the coroutines concurrently
    await asyncio.gather(
        increment(lock),
        decrement(lock)
    )

# Run the main coroutine
asyncio.run(main())
```

Output:

Trying to acquire the lock to increment...

Lock acquired to increment.

Trying to acquire the lock to decrement...

Shared resource incremented to: 1

Lock released after increment.

Lock acquired to decrement.

Shared resource decremented to: 0

Lock released after decrement.

3.5.3 Example code-7

Same above example code but using **async with lock and lock.locked()**

The async with lock: statement automatically releases the lock when the associated block exits, so there's no need to manually call lock.release() within the coroutine.

```
import asyncio

# Define a shared resource
shared_resource = 0

# Define a coroutine to increment the shared resource
async def increment(lock):
    global shared_resource
    print("Trying to increment the shared resource...")
    async with lock:
        print("Lock acquired to increment.")
        if lock.locked():
            shared_resource += 1
            await asyncio.sleep(1)  # Simulate some work
            print("Shared resource incremented to:", shared_resource)
    print("Lock released after increment.")

# Define a coroutine to decrement the shared resource
async def decrement(lock):
    global shared_resource
```

```
    print("Trying to decrement the shared resource...")
    async with lock:
        print("Lock acquired to decrement.")
        if lock.locked():
            shared_resource -= 1
            await asyncio.sleep(1)  # Simulate some work
            print("Shared resource decremented to:", shared_resource)
    print("Lock released after decrement.")

async def main():
    # Create a lock
    lock = asyncio.Lock()

    # Run the coroutines concurrently
    await asyncio.gather(
        increment(lock),
        decrement(lock)
    )

# Run the main coroutine
asyncio.run(main())
```

Output:

Trying to increment the shared resource...

Lock acquired to increment.

Trying to decrement the shared resource...

Shared resource incremented to: 1

Lock released after increment.

Lock acquired to decrement.

Shared resource decremented to: 0

Lock released after decrement.

3.6 Event Loop

3.6.1 asyncio.get_event_loop()

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

3.6.2 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.

3.6.3 class asyncio.Loop

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

method	usage
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 loo
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'. The context dictionary typically contains the following keys: 1. **message**: A string message describing the error. 2. **exception**: The actual exception object that was raised. 3. **future**: The future or task that raised the exception.

3.6.4 Example code-7

```
import asyncio

# Define a coroutine that raises an exception
async def buggy_coroutine():
    print("Running buggy coroutine...")
    # This line will raise a ZeroDivisionError
    result = 1 / 0

# Define a coroutine that raises another exception
```

```
async def non_integer():
    print("Running non-integer coroutine...")
    # This line will raise a TypeError
    result = 'a' / 1

# Define an exception handler
def exception_handler(loop, context):
    # The context contains the exception and other information
    print("Exception occurred:", context['message'])
    print("Exception type:", type(context['exception']))
    print("Exception:", context['exception'])
    print("Future:", context['future'])
    # Stop the event loop
    loop.stop()

async def main():
    # Get the current event loop
    loop = asyncio.get_event_loop()

    # Set the exception handler
    loop.set_exception_handler(exception_handler)

    # Create tasks for the buggy coroutines
    task1 = loop.create_task(buggy_coroutine())
    task2 = loop.create_task(non_integer())

    try:
        # Run the tasks until they complete
        await asyncio.gather(task1, task2)
    except Exception as e:
        print("Caught exception:", e)

    # Call the default exception handler manually
    loop.call_exception_handler({
        "message": "Manually triggered exception",
        "exception": Exception("Manual exception"),
        "future": None
    })

    # Stop the event loop (if not already stopped)
    loop.stop()

    # Close the event loop
    loop.close()

# Run the main coroutine
asyncio.run(main())
```

Output:

Running buggy coroutine...

Running non-integer coroutine...

Caught exception: divide by zero

Exception occurred: Manually triggered exception

Exception type: <class 'Exception'>

Exception: Manual exception

Future: None

3.6.5 Example code-8

```
import asyncio

# Define a coroutine that runs indefinitely
async def infinite_task():
    while True:
        print("Infinite task is running...")
        await asyncio.sleep(1)

# Define a coroutine that completes after a delay
async def finite_task():
    print("Finite task is starting...")
    await asyncio.sleep(3)
    print("Finite task is completed.")

async def stop_loop_after(loop, delay):
    await asyncio.sleep(delay)
    print(f"Stopping loop after {delay} seconds")
    loop.stop()

# Main function to demonstrate the two methods
def main():
    loop = asyncio.get_event_loop()

    # Create the infinite task
    loop.create_task(infinite_task())

    # Create the finite task and run until complete
    loop.run_until_complete(finite_task())

    # Schedule the event loop to stop after 5 seconds
    loop.create_task(stop_loop_after(loop, 5))

    # Run the event loop forever
    print("Running the event loop forever...")
    try:
        loop.run_forever()
    finally:
        print("Closing the event loop.")
        loop.close()

# Run the main function
main()
```

Output:

Infinite task is running...

Finite task is starting...

Infinite task is running...

Infinite task is running...

Finite task is completed.

Running the event loop forever...

Infinite task is running...

Infinite task is running...

Infinite task is running...

Infinite task is running...

Infinite task is running...

Stopping loop after 5 seconds

Closing the event loop.

4. Difference between coroutine and tasks

In the context of asynchronous programming, a coroutine and a task serve different purposes:

1. **Coroutine**:
   - A coroutine is a special type of function that can suspend its execution at certain points to allow other code to run before it resumes.
   - It is defined using the `async def` syntax in Python.
   - Coroutines are executed within an event loop and are often used to perform non-blocking I/O operations.
   - They are defined to be asynchronous and typically return awaitable objects, such as `await asyncio.sleep()` or other coroutines.
2. **Task**:
   - A task, in the context of asyncio, is a higher-level abstraction built on top of coroutines.
   - It represents the execution of a coroutine within an event loop.
   - Tasks are created using the `asyncio.create_task()` function or `loop.create_task()` method.
   - They allow you to concurrently execute multiple coroutines and manage their execution states.
   - Tasks are awaitable objects, which means you can `await` them to wait for their completion or gather them using `asyncio.gather()`.

In summary, a coroutine is the asynchronous function itself, while a task represents the execution of that coroutine within the asyncio event loop. Tasks are used to manage the execution of coroutines and coordinate their completion

5. 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.

A note of callbacks used by functions and class methods of machine module: all these callbacks should be considered as executing in an interrupt context.

5.1 Memory access

The module exposes three objects used for raw memory access

5.1.1 machine.mem8

Read/write 8 bits of memory.

5.1.2 machine.mem16

Read/write 16 bits of memory.

5.1.3 machine.mem32

Read/write 32 bits of memory

5.1.4 Example code-9

```
import machine

# Write a value to a 32-bit memory address
address = 0x1000
value = 0xABCD1234
machine.mem32[address] = value

# Read the value from the same address
read_value = machine.mem32[address]

print("Value at address {}: {}".format(hex(address), hex(read_value)))
```

Output:

Value at address 0x1000: -0x800b9d0

5.2 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)
```

5.2.1 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.

5.2.2 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.

5.2.3 Example code – 10

```
import machine


# Function to perform a time-critical operation
def critical_operation():
    # Disable interrupts and save the current state
    print("disabling the interrupts..")
    irq_state = machine.disable_irq()

    try:
        # Perform a small amount of time-critical work here
        # For example, updating a shared resource safely
        # Note: Keep this section as short as possible
        shared_resource = 42  # Example of critical operation
        shared_resource += 1
        print("Critical operation performed: shared_resource =",
shared_resource)

    finally:
        # Re-enable interrupts, restoring the previous state
        print("enabling the interrupts..and restoring the previous state")
        machine.enable_irq(irq_state)


# Main code execution
print("Starting main code execution")

# Perform the critical operation
critical_operation()

print("Main code execution continues")
```

Output:

Starting main code execution

disabling the interrupts..

Critical operation performed: shared_resource = 43

enabling the interrupts..and restoring the previous state

Main code execution continues

5.3 Power related functions

5.3.1 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.

5.3.2 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 (on many ports this includes system timer interrupt occurring at regular intervals on the order of millisecond).

5.3.3 machine.lightsleep([time_ms])

5.3.4 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 deep-sleep wake from other resets.

5.4 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.

5.4.1 class machine.Pin

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.

5.4.2 Pin class Methods

5.4.2.1 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.

5.4.2.2 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.

5.4.2.3 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.

5.4.2.4 Pin.on()

Set pin to “1” output level.

5.4.2.5 Pin.off()

Set pin to “0” output level.

The following methods are not part of the core Pin API and only implemented on certain ports.

5.4.2.6 Pin.low()

Set pin to “0” output level. Availability: nrf, rp2, stm32 ports.

5.4.2.7 Pin.high()

Set pin to “1” output level. Availability: nrf, rp2, stm32 ports.

5.4.2.8 Pin.mode([mode])

Get or set the pin mode. See the constructor documentation for details of the mode argument. Availability: cc3200, stm32 ports.

5.4.2.9 Pin.pull([pull])

Get or set the pin pull state. See the constructor documentation for details of the pull argument. Availability: cc3200, stm32 ports.

5.4.2.10 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.

5.4.3 Example code-11

```
import machine

# Initialize a pin as an output pin with an initial value of 0
led_pin = machine.Pin('PA5', machine.Pin.OUT)
```

```python
# Initialize a pin as an input pin with a pull-up resistor
button_pin = machine.Pin('PC13', mode=machine.Pin.IN,
pull=machine.Pin.PULL_UP)



# Function to demonstrate reading and writing pin values
def pin_operations():
    # Set the LED pin high
    led_pin.value(1)
    print("LED pin value set to: " ,led_pin.value())

    # Set the LED pin low using the on/off methods
    led_pin.off()
    print("LED pin value set to: ",led_pin.value())
    led_pin.on()
    print("LED pin value set to: " ,led_pin.value())

    # Read the button pin value
    button_value = button_pin.value()
    print("Button pin value is: " ,button_value)




# Function to demonstrate disabling and enabling interrupts
def critical_section():
    # Disable interrupts
    irq_state = machine.disable_irq()

    try:
        # Perform time-critical operations here
        print("Performing time-critical operations")
        # Example: Toggle the LED pin
        led_pin.value(not led_pin.value())
    finally:
        # Re-enable interrupts
        machine.enable_irq(irq_state)


# Main code execution
print("Starting main code execution")

# Perform pin operations
pin_operations()

#Execute a critical section with interrupts disabled
critical_section()

print("Main code execution continues")
```

Output:

Starting main code execution

LED pin value set to: 1

LED pin value set to: 0

LED pin value set to: 1

Button pin value is: 1

Performing time-critical operations

Main code execution continues

5.4.4 Example code-12

```
import machine
import time
# Define a function to be called when the interrupt occurs
def button_pressed(b):
    print("Button ",b," pressed!")

# Initialize the button pin as an input with pull-down resistor
button = machine.Pin('PC13', machine.Pin.IN, machine.Pin.PULL_DOWN)
uart = machine.UART(2, baudrate=115200)

# Attach an interrupt to the button pin
button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_pressed)

# Main loop
while True:
    uart.write('hello\n')  # Send the message "hello"
    time.sleep(1)  # Wait for 1 second
    if uart.any():  # Check if there is any incoming data
        msg = uart.read()  # Read the received data
        print(msg)
```

Output:

hello

hello

5.4.5 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

Pin.PULL_HOLD

Selects whether there is a pull up/down resistor. Use the value None for no pull.
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

5.5 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

5.5.1 class machine.UART(id, ...)

- Construct a UART object of the given id

5.5.2 UART class methods

5.5.2.1 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.

5.5.2.2 UART.deinit()

- Turn off the UART bus

5.5.2.3 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.

5.5.2.4 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

5.5.2.5 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

5.5.2.6 UART.write(buf)

- Write the buffer of bytes to the bus. Return value: number of bytes written or None on timeout.

5.5.2.7 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.

5.5.3 Example code – 13

```
import machine
import time

# Initialize UART (use UART2 which is available on the Nucleo-F401RE)
uart = machine.UART(2, baudrate=115200)

# Main loop to send "hello" over UART
while True:
    uart.write('hello\n')  # Send the message "hello"
    time.sleep(1)  # Wait for 1 second
    if uart.any():  # Check if there is any incoming data
        msg = uart.read()  # Read the received data
        print(msg)
```

5.5.4 Example code – 14 :Interrupt with UART

```
import machine
import time
# Define a function to be called when the interrupt occurs
def button_pressed(b):
    print("Button ",b," pressed!")

# Initialize the button pin as an input with pull-down resistor
button = machine.Pin('PC13', machine.Pin.IN, machine.Pin.PULL_DOWN)
uart = machine.UART(2, baudrate=115200)

# Attach an interrupt to the button pin
button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_pressed)

# Main loop
while True:
    uart.write('hello\n')  # Send the message "hello"
    time.sleep(1)  # Wait for 1 second
    if uart.any():  # Check if there is any incoming data
        msg = uart.read()  # Read the received data
        print(msg)
```

6. micropython – access and control MicroPython internals

6.1 micropython functions

6.1.1 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 the entire heap indicating which blocks are used and which are free

6.1.2 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.

6.1.3 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).
- 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.
- 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().
- There is a finite queue to hold the scheduled functions and schedule() will raise a RuntimeError if the queue is full.

6.2 Example code-15

```
import micropython

# Display memory info
micropython.mem_info()

"""
Output:
stack: 476 out of 15360
GC: total: 61248, used: 1504, free: 59744
 No. of 1-blocks: 15, 2-blocks: 6, max blk sz: 40, max free sz: 3722



The output you're seeing from micropython.mem_info() provides information
about memory usage and garbage collection (GC) statistics in a MicroPython
environment. Let's break down each section of the output:

Memory Information Breakdown:
Stack Usage:

stack: 476 out of 15360: This indicates the stack usage of your program.
476 is the amount of stack space currently in use.
15360 is the total stack space available.
This shows how much of the stack is currently allocated and used by your
program.
Garbage Collection (GC) Statistics:

GC: total: 61248, used: 1504, free: 59744:
total: Total heap memory available for allocation.
used: Amount of heap memory currently in use.
free: Remaining free heap memory available for allocation.
These values collectively describe the current heap memory usage in your
MicroPython environment.
Block Information:

No. of 1-blocks: 15, 2-blocks: 6:
Indicates the number of blocks of memory currently allocated in the heap.
1-blocks are single blocks of memory.
2-blocks are blocks of memory that are larger and span multiple blocks.
max blk sz: 40, max free sz: 3722:
max blk sz: Maximum size of a single block of memory allocated.
max free sz: Maximum size of a contiguous free block of memory available."""
```

6.3 Example code -16

```
import micropython

# Print stack usage
print("Stack usage:", micropython.stack_use())


"""
```

```
(venv) PS C:\Users\vlab\PycharmProjects\MicroPython_codes\micropython_module>
ampy --port COM7 run .\stack_usage.py
Stack usage: 484
"""
```

6.3 Example code -17

```
import micropython
import time

def task1(arg):
    print(f"Task 1 executed withh argument: {arg}")

def task2(arg):
    print(f"Task 2 executed with argument: {arg}")

# Schedule task1 to run asynchronously
micropython.schedule(task1,"Hello")

# Schedule task2
micropython.schedule(task2,123)

# Main program loop
while True:
    print("Main program running...")
    time.sleep(1)

"""
(venv) PS C:\Users\vlab\PycharmProjects\MicroPython_codes\micropython_module>
ampy --port COM7 run .\micropython_schedule.py
Main program running...
Task 1 executed withh argument: Hello
Task 2 executed with argument: 123
Main program running...
Main program running...
Main program running...
"""
```

7. pyb — functions related to the board

The pyb module contains specific functions related to the board.

7.1 Time related functions

7.1.1 pyb.delay(ms)

- Delay for the given number of milliseconds.

7.1.2 pyb.udelay(us)

- Delay for the given number of microseconds

7.2 Reset related functions

7.2.1 pyb.hard_reset()

- Resets the pyboard in a manner similar to pushing the external RESET button.

7.2.2 pyb.bootloader()

- Activate the bootloader without BOOT* pins

7.3 Interrupt related functions

7.3.1 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.

7.3.2 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

7.4 Power related functions

7.4.1 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

7.4.2 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.

7.4.3 pyb.stop()

- Put the pyboard in a "sleeping" state.
- This reduces power consumption to less than 500 uA.

7.4.4 pyb.standby()

- Put the pyboard into a "deep sleep" state.
- This reduces power consumption to less than 50 uA.

7.4.5 Example code - 18

```
import pyb
import time
# Initialize an LED to indicate the state
led = pyb.LED(1)

# Function to toggle the LED on interrupt
def handle_interrupt(line):
    print("Interrupt occurred on line:", line)
    led.toggle()

# Configure an external interrupt on pin C13
pin = pyb.Pin('C13', pyb.Pin.IN, pyb.Pin.PULL_DOWN)
extint = pyb.ExtInt(pin, pyb.ExtInt.IRQ_RISING, pyb.Pin.PULL_DOWN, lambda
line: handle_interrupt(line))

# Main loop
while True:
    print("Going to sleep (WFI)...")
    pyb.wfi()  # Enter low-power state until an interrupt occurs
    time.sleep(1)

"""
Going to sleep (WFI)...
Going to sleep (WFI)...
Going to sleep (WFI)...
Going to sleep (WFI)...
Going to sleep (WFI)...
Interrupt occurred on line: 13
Going to sleep (WFI)...
Interrupt occurred on line: 13
Going to sleep (WFI)...
Interrupt occurred on line: 13
Going to sleep (WFI)...
Interrupt occurred on line: 13
Interrupt occurred on line: 13
Going to sleep (WFI)...
Interrupt occurred on line: 13
Interrupt occurred on line: 13
Going to sleep (WFI)...
Interrupt occurred on line: 13
Going to sleep (WFI)...
Interrupt occurred on line: 13
"""
```

7.5 Miscellaneous functions

7.5.1 pyb.unique_id()

- Returns a string of 12 bytes (96 bits), which is the unique ID of the MCU

7.6 class ADC – analog to digital conversion

7.6.1 class pyb.ADC(pin)

- Create an ADC object associated with the given pin. This allows you to then read analog values on that pin.

7.6.2 ADC.read()

- Read the value on the analog pin and return it. The returned value will be between 0 and 4095

7.6.3 Example code – 19 : Internal temperature sensor

```
from pyb import ADC
import time

# Initialize the ADC on the internal temperature sensor channel (typically
ADC channel 16)
temp_sensor = ADC(16)

# Function to convert raw ADC value to temperature
def raw_to_temperature(raw_value):
    # STM32 internal temperature sensor calibration values
    V25 = 0.76  # Voltage at 25 degrees Celsius (in Volts)
    Avg_Slope = 2.5  # Average slope (in mV/degree Celsius)
    V_ref = 3.3  # Reference voltage (in Volts)

    # Convert the raw ADC value to a voltage
    voltage = (raw_value / 4095) * V_ref

    # Calculate temperature in Celsius
    temperature = (voltage - V25) / (Avg_Slope / 1000) + 25

    return temperature

# Continuously read and print the temperature
while True:
    raw_value = temp_sensor.read()
    temperature = raw_to_temperature(raw_value)
    print("Temperature: {:.2f}C".format(temperature))
    time.sleep(1)  # Delay for 1 second between readings

"""
(venv) PS C:\Users\vlab\PycharmProjects\MicroPython_codes> ampy --port COM7
run .\ADC_internal_Temp_Sensor.py
Temperature: 24.33C
Temperature: 24.33C
```

```
Temperature: 24.00C
Temperature: 23.68C
Temperature: 23.68C
Temperature: 23.68C
Temperature: 24.65C
"""
```

7.6.4 Example code – 20: ADC multiple channel reading

```
import pyb

# Initialize ADC for channels PA1 and PA4
adc1 = pyb.ADC(pyb.Pin('PA1'))  # Initialize ADC for PA1 connected to 3.3V
pin
adc4 = pyb.ADC(pyb.Pin('PA4'))  # Initialize ADC for PA4 connected to GND pin

def read_adc_values():
    value1 = adc1.read()  # Read value from ADC channel PA1
    value4 = adc4.read()  # Read value from ADC channel PA4
    return value1, value4

def main():
    while True:
        adc_values = read_adc_values()
        print("ADC Channel PA1:", adc_values[0])
        print("ADC Channel PA4:", adc_values[1])
        pyb.delay(1000)  # Wait for 1 second

if __name__ == "__main__":
    main()

"""
(venv) PS C:\Users\vlab\PycharmProjects\MicroPython_codes> ampy --port COM7
run .\ADC_Multi_channel.py
ADC Channel PA1: 4095
ADC Channel PA4: 3
ADC Channel PA1: 4095
ADC Channel PA4: 2
ADC Channel PA1: 4095
ADC Channel PA4: 4
ADC Channel PA1: 4095
ADC Channel PA4: 2
ADC Channel PA1: 4095
ADC Channel PA4: 2
ADC Channel PA1: 4095
ADC Channel PA4: 2
"""
```

7.6.5 Example code – 21 : reading VREFINT(reference voltage of ADC)

```
from pyb import ADC
import time

# Initialize ADC for internal reference voltage (channel 17)
vref = ADC(17)  # Channel 17 is typically the internal reference voltage
```

```
while True:
    # Read the raw ADC value
    raw_value = vref.read()

    # Convert raw_value to voltage using the appropriate formula
    # Assuming VREF+ is 3.3V and ADC is 12-bit (0-4095 range)
    voltage = (raw_value / 4095.0) * 3.3

    # Print the voltage value
    print("VREF Voltage: {:.3f} V".format(voltage))

    # Delay for a short period
    time.sleep(1)

"""
(venv) PS C:\Users\vlab\PycharmProjects\MicroPython_codes> ampy --port COM7
run .\ADC_REFINT.py
VREF Voltage: 1.203 V
VREF Voltage: 1.202 V
VREF Voltage: 1.202 V
"""
```

7.6.6 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
- 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
- read_vref() is evaluated by measuring the internal voltage reference and backscale it using factory calibration value of the internal voltage reference

7.6.6.1 Example code 22 : ADCALL

```
from pyb import ADCAll
import time

# Initialize ADCAll object
adc = ADCAll(12, 0x70000)

# Example function to read and print ADCAll values
def read_adc_values():
    core_vbat = adc.read_core_vbat()
    vref = adc.read_vref()
    core_vref = adc.read_core_vref()
    temp = adc.read_core_temp()

    print("Core VBAT:", core_vbat)
    print("VREF:", vref)
    print("Core VREF:", core_vref)
    print("Internal Temp Sensor:",temp)

# Example usage in a loop
while True:
```

```
    read_adc_values()
    time.sleep_ms(1000)  # Delay between readings (1000 milliseconds)

# Optionally, you can deinitialize the ADCAll object when done
adc.deinit()
```

7.7 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,

7.7.1 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.

7.7.2 ExtInt Methods

7.7.2.1 ExtInt.disable()

- Disable the interrupt associated with the ExtInt object.

7.7.2.2 ExtInt.enable()

- Enable a disabled interrupt.

7.7.2.3 ExtInt.line()

- Return the line number that the pin is mapped to.

7.7.3 Constants

7.7.3.1 ExtInt.IRQ_FALLING

- interrupt on a falling edge

7.7.3.2 ExtInt.IRQ_RISING

- interrupt on a rising edge

7.7.3.3 ExtInt.IRQ_RISING_FALLING

- interrupt on a rising or falling edge

7.7.4 example code – 23

```
from pyb import Pin, ExtInt
import time

def callback(line):
    print("Interrupt triggered on line:", line)

# Configure external interrupt on pin PA0
ext_int = ExtInt(Pin('PC13'), ExtInt.IRQ_RISING, Pin.PULL_NONE, callback)

while True:
    print("Waiting for interrupt...")
    time.sleep(1)

"""
(venv) PS C:\Users\vlab\PycharmProjects\MicroPython_codes> ampy --port COM7
run .\External_interrupts.py
Waiting for interrupt...
Waiting for interrupt...
Waiting for interrupt...
Waiting for interrupt...
Interrupt triggered on line: 13
Waiting for interrupt...
Waiting for interrupt...
Interrupt triggered on line: 13
Waiting for interrupt...
Interrupt triggered on line: 13
Waiting for interrupt...
Interrupt triggered on line: 13
Waiting for interrupt...
Interrupt triggered on line: 13
Waiting for interrupt...
Interrupt triggered on line: 13
Waiting for interrupt...

Aborted!
"""
```

7.8 class LED – LED object

The LED object controls an individual LED (Light Emitting Diode).

7.8.1 class pyb.LED(id)

Create an LED object associated with the given LED:

- id is the LED number, 1-4.

7.8.2 LED Methods

7.8.1 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.

7.8.2.2 LED.off()

- Turn the LED off.

7.8.2.3 LED.on()

- Turn the LED on, to maximum intensity.

7.8.2.4 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.

7.8.3 Example code – 24

```
led = pyb.LED(1)
while True:
    led.toggle()
    pyb.delay(1000)
```

7.9 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.

7.9.1 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

7.9.2 Pin Methods

7.9.2.1 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.

7.9.2.2 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

7.9.3 Constants

7.9.3.1 Pin.ALT

initialise the pin to alternate-function mode for input or output

7.9.3.2 Pin.AF_OD

initialise the pin to alternate-function mode with an open-drain drive

7.9.3.3 Pin.AF_PP

initialise the pin to alternate-function mode with a push-pull drive

7.9.3.4 Pin.ANALOG

initialise the pin to analog mode Pin.IN initialise the pin to input mode

7.9.3.5 Pin.OUT_OD

initialise the pin to output mode with an open-drain drive

7.9.3.6 Pin.OUT_PP

initialise the pin to output mode with a push-pull drive

7.9.3.7 Pin.PULL_DOWN

enable the pull-down resistor on the pin

7.9.3.8 Pin.PULL_NONE

don't enable any pull up or down resistors on the pin

7.9.3.9 Pin.PULL_UP

enable the pull-up resistor on the pin

7.10 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.

7.10.1 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.

7.10.2 Timer Methods

7.10.2.1 Timer.init(*, freq, prescaler, period, mode=Timer.UP, div=1, callback=None, deadtime=0, brk=Timer.BRK_OFF)

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

• **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.

7.10.2.2 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.

7.10.2.3 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.

7.10.2.4 Timer.counter([value])

- Get or set the timer counter.

7.10.2.5 Timer.freq([value])

- Get or set the frequency for the timer (changes prescaler and period if set).

7.10.2.6 Timer.period([value])

- Get or set the period of the timer.

7.10.2.7 Timer.prescaler([value])

- Get or set the prescaler for the timer.

7.10.2.8 Timer.source_freq()

- Get the frequency of the source of the timer

7.11 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.

7.11.1 Methods

7.11.1.1 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.

7.11.1.2 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%

7.11.1.3 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%.

7.11.2 Example code – 25

```
tim =pyb.Timer(1)
print(tim)
tim.init(freq=10)
print(tim)
print(tim.source_freq())

#timer counter
print(tim.counter())
```

7.11.3 example code – 26

```
import pyb

def f():
    pyb.LED(1).toggle()

# Initialize Timer 1 with a frequency of 20 Hz
tim1 = pyb.Timer(1, freq = 20)

# Set the callback function for Timer 1

tim1.callback(f)
print(pyb.freq())
```

7.11.4 example code – 27 : PWM

```
from pyb import Pin, Timer

# Define the pin connected to the LED
led_pin = Pin('PA5')  # Change this to the appropriate pin for your setup

# Create a Timer object
tim = Timer(2, freq=1000)  # Timer 2, with a frequency of 1000 Hz
```

```
# Configure the Timer channel for PWM
ch = tim.channel(1, Timer.PWM, pin=led_pin)

# Set the duty cycle to achieve 50% brightness (50% duty cycle)
ch.pulse_width_percent(1)
```

7.12 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

7.12.1 class pyb.UART(bus, ...)

- Construct a UART object on the given bus

7.12.2 UART Methods

7.12.2.1 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).

7.12.2.2 UART.deinit()

- Turn off the UART bus.

7.12.2.3 UART.any()

- Returns the number of bytes waiting (may be 0).

7.12.2.4 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.

7.12.2.5 UART.readchar()

- Receive a single character on the bus

7.12.2.6 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.

7.12.2.7 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.

7.12.2.8 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.

7.13 class Switch – switch object

- A Switch object is used to control a push-button switch.

7.13.1 class pyb.Switch

- Create and return a switch object

7.13.2 Switch Methods

7.13.2.1 Switch.__call__()

- Call switch object directly to get its state: True if pressed down, False otherwise.

7.13.2.2 Switch.value()

- Get the switch state. Returns True if pressed down, otherwise False.

7.13.2.3 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

7.13.3 Example code – 28

```
import pyb
sw = pyb.Switch()
while True:
    #print(sw.value())
    print(sw())
    pyb.delay(1000)
```

7.13.4 Example code – 29: press switch to glow LED

```python
sw = pyb.Switch()

#method1
while True:
    sw.callback(lambda:pyb.LED(1).toggle())


#method 2
def led():
    pyb.LED(1).toggle()

while True:
    sw.callback(led)
```

8. stm — functionality specific to STM32 MCUs

8.1 Memory access

The module exposes three objects used for raw memory access.

8.1.1 stm.mem8

- Read/write 8 bits of memory.

8.1.2 stm.mem16

- Read/write 16 bits of memory.

8.1.3 stm.mem32

- Read/write 32 bits of memory.

8.2 Example code -30

```python
import pyb
import stm

# Get the unique MCU ID
mcu_id = pyb.unique_id()

# Access the flash size register (STM32F401 specific)
flash_size_kb = stm.mem16[0x1FFF7A22]

# Print the MCU ID and flash size
print("MCU ID:", mcu_id)
print("Flash Size: {} KB".format(flash_size_kb))

#MCU ID: b'U\x00D\x00\x03P2R720 '
#Flash Size: 512 KB
```