MicroPython is a lean and efficient implementation of the [Python 3](http://www.python.org/) 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.

**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

**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

**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. |

**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])

**Differenece between co-routine 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

**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.

**uasyncio** 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.

1. **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.

1. **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

**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.

Example code-2

import uasyncio  
  
# Task 1: Define a coroutine  
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

**Example code-3**

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

**class Event**

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

**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 |

**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

**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:

**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 |

**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.

**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.

|  |  |
| --- | --- |
| **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 |

**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.

**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.

**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

|  |  |
| --- | --- |
| **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. |

**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

**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.

**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.

**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

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

**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.

**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

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

**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 deep-sleep wake from other resets.

**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.

**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.

**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.

**🡪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.

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: nrf, rp2, stm32 ports.

**🡪Pin.high()** Set pin to “1” output level. Availability: nrf, rp2, stm32 ports.

**🡪Pin.mode([mode ])** Get or set the pin mode. See the constructor documentation for details of the mode argument. Availability: cc3200, stm32 ports.

**🡪Pin.pull([pull])** Get or set the pin pull state. See the constructor documentation for details of the pull argument. Availability: cc3200, stm32 ports.

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)  
  
# 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

**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.

**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

**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