AN4964
Application note

MicroPython scripting language over SPWF04S

Introduction

This document describes how to access MicroPython within SPWF04S modules, classes and their methods, the scripts integrated in the software and a few useful examples for further development.

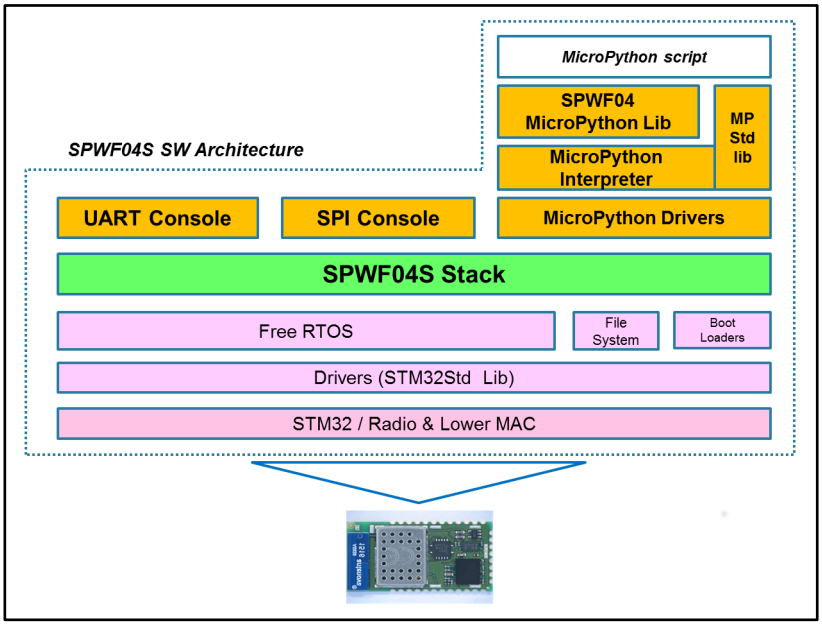
The SPWF04S software integrates a MicroPython engine enabling MicroPython scripts to be executed in the STM32 micro, thus allowing end-user software to be embedded in SPWF04S modules. Therefore, it is possible to create fully standalone Wi-Fi devices with SPWF04S modules without using an external MCU in order to reduce the final device size and cost.

MicroPython on SPWF04S can support the development of fairly complex and powerful applications, providing the necessary libraries for managing Wi-Fi connections, security, data transfer and analysis and SPWF04S hardware interfaces such as UART, SPI, I2C, GPIO, PWM, DAC and ADC available in the module.

Both the standard and custom MicroPython classes exporting the SPWF04S hardware and protocol stack capabilities are available to the user for script development.

For a detailed description of MicroPython concepts and standard classes, you can refer to *Section 6: "Tips and tricks"*.

Figure 1: SPWF04S architecture diagram



Contents

1 MicroPython on the SPWF04 6
1.1 Getting started 6
1.2 Storage volume IDs 6
1.3 Operational modes 6
1.3.1 REPL: interactive console 7
1.3.2 Run time script execution 7
1.3.3 Hard script execution 7
1.4 Flashing LEDs and errors 7
1.5 Configuration 8
1.5.1 Configuration variables 8
1.5.2 GPIOs 8
1.5.3 AT command 8
2 Using REPL 10
2.1 REPL features 10
2.1.1 Auto-indent 10
2.1.2 The underscore variable 11
2.1.3 Paste mode 11
2.1.4 Raw mode 11
2.1.5 Exit command 12
3 MicroPython libraries 13
3.1 Python standard libraries and micro-libraries 13
3.2 MicroPython specific libraries 13
3.3 Specific libraries for the SPWF04S port 13
4 Built-in examples 16
5 MicroPython script examples 17
5.1 Basics 17
5.2 WIND 17
5.3 LED interface 18
5.4 WLAN 18
5.5 Socket 18
5.6 Garbage collector 20
5.7 How to use hardware peripherals 21
5.7.1 UART 21
5.7.2 SPI and PIN 21

5.7.3 I²C ... 22
5.7.4 ADC ... 22
5.7.5 DAC ... 23
6 Tips and tricks ... 24
6.1 RAM allocation ... 24
6.2 Buffers ... 24
6.3 Byte use ... 24
6.4 Arrays ... 24
6.5 Heap management ... 25
6.6 MicroPython extras ... 26
6.6.1 Catching object references ... 26
6.6.2 Controlling garbage collection ... 26
6.6.3 Compilation phase optimization ... 26
6.6.4 String operation ... 26
6.6.5 Object references ... 27
7 Features not supported in MicroPython ... 28
8 References ... 29
9 Revision history ... 30

List of tables

Table 1: Storage volume IDs 6
Table 2: Console-enabled configuration summary 8
Table 3: MicroPython modes 9
Table 4: micropython.mem_info(1) output 25
Table 5: Document revision history 30

List of figures

Figure 1: SPWF04S architecture diagram 1
Figure 2: MicroPython modes 7
Figure 3: DAC waveform on GPIO15 23

1 MicroPython on the SPWF04

1.1 Getting started

In the default configuration, when the module is powered up, the MicroPython engine is disabled.

To manage all the supported modes and resources, a set of configuration variables, AT commands, and GPIOs are provided to handle MicroPython execution modes.

1.2 Storage volume IDs

The SPWF04 module can support different volumes, to be formatted as FAT12.

Table 1: Storage volume IDs

Volume ID	Mode	Brief	Description
0	RW	External Flash	SD card or external SPI Flash (if soldered)
1	RO	Application Flash	A resident FAT12 filesystem. The user can upload a new filesystem using AT+S.FSUPDATE command. The content of this partition is not lost if an Over-The-Air firmware update is performed. It is therefore advisable to host the script MicroPython used as main application. You can protect the content of this partition when you build the binary image. In such a way the content is not accessible through a web server or AT shell commands.
2	RW	RAM	A volatile filesystem. Files stored in the RAM are lost after a reboot.
3	RO	User Flash	A resident FAT12 filesystem. The user can upload a new filesystem using AT+S.FSUPDATE command. The content of this partition is lost if an Over-The-Air firmware update is performed. For this reason, it is not recommended to host the script MicroPython used as main application.

1.3 Operational modes

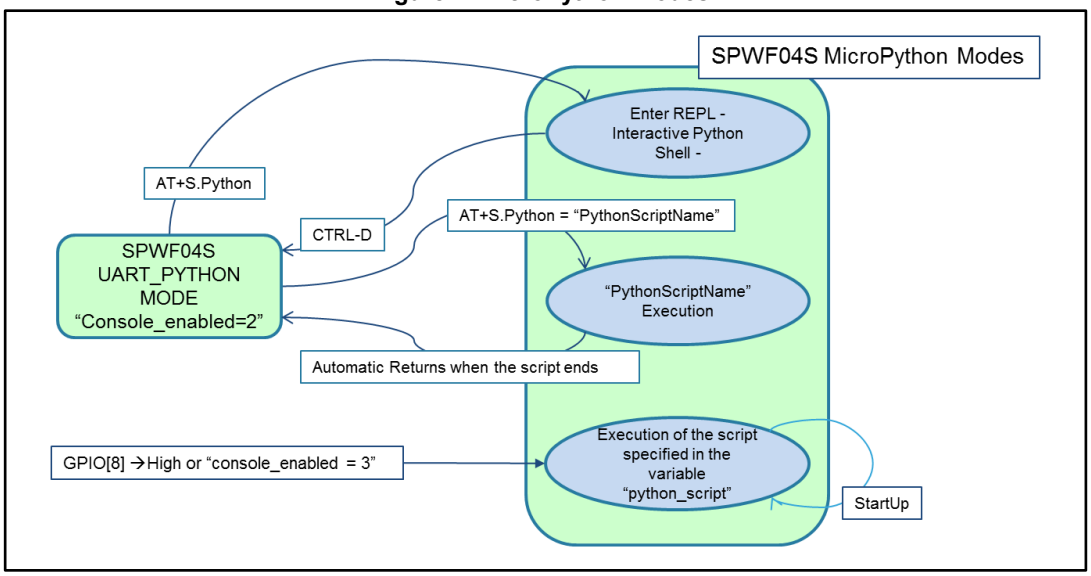
The SPWF04S module firmware contains the MicroPython engine which is not activated by default.

SPWF04S supports the following operational modes:

1. REPL interactive console (only for debug);
2. Run Time script execution mode;
3. Hard script execution.

The user can select the desired working mode through dedicated GPIOs and AT commands.

Figure 2: MicroPython modes



1.3.1 REPL: interactive console

REPL (read evaluation print loop) is the standard name given to the interactive MicroPython prompt enabled on SPWF04S. Typically used for debugging purposes, it is the easiest way to interactively test the code and run commands.

On SPWF04S, the AT+s.python command launched without parameters and the CTRL-D escape character are respectively used to enter and to exit the REPL console (`AT+s.python`).

The code executed in REPL mode is slower than the other mode execution times. It is useful for functional tests but not for benchmarking.

1.3.2 Run time script execution

When the UART console is set on the module, a Python script can be launched through the AT+s.python command using the script specified as its parameter (`AT+s.python=<volume id>:/<script name>`). The script must be hosted on the module local filesystem or SD card (see *Section 1.2: "Storage volume IDs"* for volume ids full list).

1.3.3 Hard script execution

The user can enable this mode by:

- keeping GPIO(8) high at the boot time
- setting the console_enabled environment to 3 (refer to *Table 2: "Console-enabled configuration summary"*).

Therefore, the default script specified in the python_script configuration variable is automatically executed after the boot. This option is suitable for host-less working mode.

1.4 Flashing LEDs and errors

When starting the MicroPython engine:

- in REPL console mode, check whether the CONSOLE_LED (connected to GPIO(14)) is powered on;
- in a script using run time or hard script execution mode, ensure that:

a. the CONSOLE_LED is powered on at starting time;
b. the CONSOLE_LED is off during script execution and ready to be used by the application if required.
c. the CONSOLE_LED is switched and held on when an error occurs during script execution or when the script execution has been stopped.

1.5 Configuration

1.5.1 Configuration variables

The available configuration variables are:

- **console_enabled**: defined to set the console on SPWF04S. To enable the MicroPython engine you can use just two allowed values (refer to *Table 2: "Console-enabled configuration summary"*);
- **console_enabled = 2 (UART_PYTHON mode)**: mainly used for debug, it allows the user to switch between the AT console and the REPL. Therefore, the user can also execute a script (runtime mode script) and go back to AT command at script completion.
- **console_enabled = 3 (PYTHON_ONLY mode)**: allows executing only a preloaded MicroPython script. No UART/SPI communication is allowed to/from external host through an embedded AT/SPI console.
- **python_memsize**: configures the amount of memory to be reserved for MicroPython script/REPL execution. The default value is 32 (in units of 1024 bytes).
- **python_script**: configures the script name executed at boot when console_enabled is set to 3 (PYTHON_ONLY). The default value is 3:/uPython_test.py (a ROM resident example script).

Table 2: Console-enabled configuration summary

Setting	Mode	Notes	Description
Console-enabled = 1	UART_ONLY	Default	Enables the UART console use
Console-enabled = 0	SPI_ONLY	-	Enables the SPI console use
Console_enabled = 2	UART_PYTHON	-	Enables the REPL activation and MicroPython scripts via an AT Command
Console_enabled = 3	PYTHON_ONLY	-	Activates Python script execution at boot

1.5.2 GPIOs

The available GPIOs are:

- **GPIO (8)**: reserved to enter PYTHON_ONLY mode via hardware. If the GPIO (8) level is high at startup, the SPWF04 firmware boots directly in MicroPython mode and runs the default script configured in the python_script configuration variable.
- **GPIO (14) (CONSOLE_LED)**: held high or low according to the MicroPython execution mode and status (refer to *Section 1.4: "Flashing LEDs and errors"*).

1.5.3 AT command

AT+s.PYTHON command is defined to run the REPL or a MicroPython script when the UART AT console is enabled. It requires the console-enabled variable set to 2 to be effective.

The following table shows MicroPython modes characteristics and the main device settings required to enable the desired mode.

Table 3: MicroPython modes

Mode	Device setting	Command	CONSOLE_LED
UART_PYTHON, REPL use	AT+S.SCFG = console_enabled, 2 AT+S.SCFG = python_memsize, 32 GPIO(8): low	`AT+S.PYTHON`; this command returns the string: `AT-S.OK MicroPython v1.6.2-84-g7f202fb on 2016-07-13; SPWF04Sx with STM32F439 >>>` (1)	The CONSOLE_LED is on when REPL is active.
UART_PYTHON, script execution	AT+S.SCFG=console_enabled, 2 AT+S.SCFG = python_memsize, 32 GPIO(8): low	`AT+S.PYTHON=LED.py`; this command returns the string: `AT-S.Launching script:2:LED.py AT-S.OK+WIND:0:Console active` When an error is detected by the MicroPython engine, the execution is stopped and the error is displayed on REPL prompt, before closing it and coming back to AT command shell.	The CONSOLE_LED is on when the script is running.
PYTHON_ONLY, hardware driven	AT+S.SCFG = python_memsize, 32 AT+S.SCFG = python_script, 3:/uPython_test.py GPIO(8): high	At every reboot, the GPIO (8) is checked. If high, the script pointed at by the python_script variable is executed independently of the console_enabled value.	The CONSOLE_LED is on at startup, off when the script starts, TOGGLE when the script finishes
PYTHON_ONLY, variable driven	AT+S.SCFG = console_enabled, 3 AT+S.SCFG = python_memsize, 32 AT+S.SCFG = python_script, 3:/uPython_test.py GPIO(8): low	At every reboot the script pointed at by the python_script variable is executed.	The CONSOLE_LED is on at startup, off when the script starts, TOGGLE when the script finishes.

Notes:

(1)Refer to *Section 3: "MicroPython libraries"* for a list of possible REPL commands.

2 Using REPL

When enabled, the REPL is always available on the SPWF04 AT console serial interface. Through a USB-serial convertor, you can access REPL directly from a PC.

To access the prompt over USB-serial you need to install a terminal emulator on your PC (e.g. TeraTerm, Minicom, Putty).

So you can edit your MicroPython script using the preferred editor and check it by the REPL console.

After launching the command `AT+s.python`, the REPL prompt appears (`>>>`).

2.1 REPL features

2.1.1 Auto-indent

When typing python statements which end in a colon (e.g., if, for and while), the prompt changes to three dots (...) and the cursor is indented by 4 spaces.

When you press return, the next line continues at the same level of indentation for regular statements or an additional level of indentation where appropriate.

If you press the backspace key then it deletes one level of indentation.

If your cursor is all the way back at the beginning, pressing RETURN executes the code that you enter.

Entering a *for* statement, the following string appears (the underscore shows where the cursor winds up):

```
>>> for i in range(3):
... _
```

Then, if you enter an *if* statement, an additional level of indentation is added and you can insert the required commands inside the *if* statement like a *break* command followed by RETURN and BACKSPACE:

```
>>> for i in range(30):
...     if i > 3:
...         break
...
```

Finally type print(i), press RETURN, BACKSPACE and then RETURN again:

```
>>> for i in range(30):
...     if i > 3:
...         break
...     print(i)
...
0
1
2
3
>>>
```

Auto-indent is not applied if the previous two lines are all spaces. This means you can finish a compound statement by pressing RETURN twice, and then pressing it one last time to execute the script.

2.1.2 The underscore variable

When using the REPL, you may perform computations and see the results, which are stored in the variable _ (underscore). So you can use the underscore to save the result in a variable:

```
>>> 1 + 2 + 3 + 4 + 5
15
>>> x = _
>>> x
15
>>>
```

2.1.3 Paste mode

If you want to paste some code into your terminal window, the auto-indent feature could cause some errors. For example, if you try to paste this Python code into the normal REPL:

```
def foo():
    print('This is a test to show paste mode')
    print('Here is a second line')
foo()
```

the result is:

```
>>> def foo():
...         print('This is a test to show paste mode')
...             print('Here is a second line')
...             foo()
...
  File "<stdin>", line 3
IndentationError: unexpected indent
```

Ctrl-E activates paste mode, which essentially turns off the auto-indent feature and changes the prompt from >>> to ===.:

```
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
=== def foo():
===     print('This is a test to show paste mode')
===     print('Here is a second line')
=== foo()
===
This is a test to show paste mode
Here is a second line
>>>
```

Paste mode also allows pasting blank lines. The pasted text is compiled as if it were a file. Pressing **Ctrl-D** exits paste mode and starts compiling.

2.1.4 Raw mode

The raw mode is not a common operating mode as it is intended for programmatic use. It essentially behaves like paste mode with echo turned off.

Raw mode is activated by pressing **Ctrl-A**; after entering the Python code, press **Ctrl-D**, then an *OK* signals that the code is going to be compiled and executed. Any output (or errors) will be returned.

Press **Ctrl-B** to exit the raw mode and return to normal REPL.

2.1.5 Exit command

Press **Ctrl-D** to exit the REPL console.

To exit Python code, launch `raise SystemExit`.

3 MicroPython libraries

The following sections describe standard Python libraries built in MicroPython.

Additional libraries can be downloaded from the micropython-lib repository: *github.com/micropython/micropython-lib*.

3.1 Python standard libraries and micro-libraries

The following standard Python libraries have been modified to provide core functionality for MicroPython modules.

Modules have a u-name and a non-u-name. The non-u-name can be overridden by a file with the same name in your package path. For example, `import json` first searches and load a json.py file or json directory if found, otherwise, it goes back to load the built-in ujson module.

The supported Python standard libraries and micro-libraries are:

- array: numeric data arrays
- builtins: basic functions
- cmath: mathematical functions for complex numbers
- gc: garbage collector
- math: mathematical functions
- select: wait for events on a set of streams
- sys: system specific functions
- ubinascii: binary/ASCII conversions
- ucollections: collection and container types
- uhashlib: hashing algorithm
- uheapq: heap queue algorithm
- uio: input/output streams
- ujson: JSON encoding and decoding
- uos: basic “operating system” services
- ure: regular expressions
- usocket: socket module
- ustruct: pack and unpack primitive data types
- utime: time related functions
- uzlib: zlib decompression
- uctypes: access binary data in a structured way

3.2 MicroPython specific libraries

Specific funcions for MicroPython implementation are available in the following libraries:

- machine: functions related to the board
- micropython: access and control MicroPython internals

3.3 Specific libraries for the SPWF04S port

The following libraries are specific for the SPWF04 modules:

- **Time related functions**:
  - delay (ms): delays for a given number of milliseconds
  - µdelay (µs): delays for a given number of microseconds
  - millis (): returns the number of milliseconds since the last board reset

  - elapsed_millis (start): returns the number of milliseconds which have elapsed since a given start

- **Classes**:
  - ADC: STM32 ADC peripheral:
    - ADC.read (): reads the value on the analog pin and returns it
  - DAC: STM32 DAC peripheral
    - DAC.write (value): direct access to the DAC output
  - I^2C – STM32 I2C3 peripheral:
    - I^2C.deinit (): turns off the I^2C3 bus
    - I^2C.init (mode, *, addr=0x12, baudrate=400000,gencall=False): initializes the I^2C3 bus with given parameters
    - I^2C.is_ready (addr): checks if an I^2C device responds to a given address (valid only in master mode)
    - I^2C.mem_read (data, addr, memaddr, *, timeout=5000,addr_size=8): reads from the memory of an I^2C device
    - I^2C.mem_write (data, addr, memaddr, *, timeout=5000,addr_size=8): writes in the memory of an I^2C device
    - I^2C.recv (recv, addr=0x00, *, timeout=5000): receives data on the bus
    - I^2C.send (recv, addr=0x00, *, timeout=5000): sends data to the bus
    - I^2C.scan () – scans all I^2C addresses from 0x01 to 0x7f and returns a list of those that respond (valid only in master mode)
  - LED (SPWF04S debug LEDs 1 to 3):
    - LED.on () - turns the LED on
    - LED.off () - turns the LED off
    - LED.toggle () - toggles the LED between on and off
  - PIN (SPWF04S GPIOs 0 to 18):
    - Pin.high (): sets the pin to high level
    - Pin.init (mode): initializes the pin
    - Pin.low (): sets the pin to low level
  - SPI (STM32 SPI1 peripheral):
    - SPI.deinit () – turns the SPI1 bus off
    - SPI.init (mode, baudrate=328125, *, prescaler, polarity=1, phase=0, bits=8, firstbit=SPI.MSB, ti=False,crc=None): initializes the SPI1 bus with given parameters
    - SPI.recv (recv, *, timeout=5000): receives data on the bus
    - SPI.send (send, *, timeout=5000): sends data to the bus
    - SPI.send_recv (send, recv=None, *, timeout=5000): sends and receives data on the bus at the same time
  - UART (STM32 UART1 peripheral):
    - UART.init (baudrate, bits=8, parity=None, stop=1, *, timeout=1000, flow=0, timeout_char=0,read_buf_len=64): initializes the UART1 bus with given parameters
    - UART.deinit (): turns the UART1 bus off
    - UART.any (): returns the number of waiting bytes
    - UART.writechar (char): writes a single character (an integer) on the bus
    - UART.read ([nbytes]): reads characters. If nbytes has been defined, then the UART reads at most the bytes specified. If nbytes are available in the buffer, it returns immediately, otherwise it returns when sufficient characters are sent or the timeout elapses
    - UART.readall (): reads as much data as possible; returns after the timeout has elapsed
    - UART.readchar (char): reads a single character on the bus

  - UART.readinto (buf[, nbytes]): reads bytes in the buf. If nbytes has been defined, then the UART reads at most the bytes specified. Otherwise, it reads at most len (buf) bytes
  - UART.readline (): reads a line, ending in a new line character. If such a line exists, return is immediate. If the timeout elapses, all available data is returned regardless of any a new line
  - UART.write (buf): writes the buffer of bytes to the bus
- WIND (SPWF04S asynchronous messages event handler):
  - WIND.callback (function): registers a callback function to be called on received events

4 Built-in examples

The module includes the following pre-loaded Python script in the internal filesystem:

- **WLAN_STA.py**: shows how to configure the device as a station or mini access point using the MicroPython WLAN module.
- **RL_TCP_CL.py**: connects to a remote server using TCP socket client and waits for incoming commands.
- **RL_UDP_CL.py**: connects to a remote server using UDP socket client and waits for incoming commands. The server can then send commands to switch a LED on/off.
- **RL_TCP_SE.py**: instantiates a local TCP server and wait for incoming connections. The client then sends commands to switch a LED on/off.
- **RL_UDP_SE.py**: instantiates a local UDP server and waits for incoming connections. The client then sends commands to switch a LED on/off.
- **RL_TCP_SE_GC_COLLECT.py**: instantiates a local TCP server and waits for incoming connections. The client then sends commands to to switch a LED on/off. When free available memory is lower than 24000 bytes, the script triggers a garbage collection.

5 MicroPython script examples

Some useful MicroPython examples running on SPWF04S can be tested as follows:

1. Copy/paste script content to a PC file **test.py**. Run an offline PC tool to create an **FS.img** file and install it Over-The-Air to SPWF04S through the **AT+S.FSUPDATE** command (SPI command 0x58). This file becomes part of the SPWF04S resident filesystem and can be found in the **1:/test.py** path. At this stage, run the script using mode 2 or 3;
2. Forward script content to the **AT+S.FSC** command (SPI command 0x23). This file becomes part of the SPWF04S volatile filesystem and can be found in the **2:/test.py** path. At this stage, run the script using mode 2;
3. Enter mode 1 and switch to paste mode (CTRL+E). Copy/paste script content to the SPWF04S UART and run the script (CTRL+D).

5.1 Basics

A very simple script containing some *while* and *if* and messages print on the REPL console:

```
a = 0
while True:
        a = a + 1
        if a % 100 == 0:
               print(a)
```

results in:

```
100
200
300
```

5.2 WIND

The WIND built-in module manages asynchronous messages coming from the SPWF04S network events and resident software stack events.

The script:

```
from pyb import WIND
import utime
counter = 0
def cb():
         global counter
         try:
                  if w.code() == 35:
                           print('network scanned without results')
                           counter = counter + 1
                  else:
                           print('Indication '+str(w.code())+' occurred')
         except:
                  pass
         return
w = WIND()
w.callback(cb)
while True:
        if counter == 3:
                 print("3 loops with scan/complete. Suggest: switch in miniAP
mode!")
        utime.sleep(1)
```

results in:

```
INDICATION 21 OCCURRED
NETWORK SCANNED WITHOUT RESULTS
INDICATION 21 OCCURRED
NETWORK SCANNED WITHOUT RESULTS
INDICATION 21 OCCURRED
NETWORK SCANNED WITHOUT RESULTS
3 LOOPS WITH SCAN/COMPLETE. SUGGEST: SWITCH TO MINIAP MODE!
```

5.3 LED interface

This example shows how to use hardcoded LEDs, mainly for debugging purposes. Instead, for typical LED use, choose Pin class.

The script:

```
from pyb import LED
import utime
l = LED(2)
l.on()
while True:
        l.toggle()
        utime.sleep(1)
```

results in toggling CONSOLE_LED, 1 Hz frequency.

5.4 WLAN

Use the WLAN module to configure the device to join a defined network or reconfigure it as a mini access point (miniAP mode) in case the preferred network was not found or association failed.

The script:

```
from network import WLAN
w = WLAN()
MySsid = 'HomeNetwork'
Found = False
print('INIT')
nets = w.scan()
for net in nets:
        if net.ssid == MySsid:
                found = True
                print('<'+MySsid+'> found: trying to connect...')
                w.init(mode=WLAN.STA, ssid=net.ssid)
                break
if w.isconnected() == False and found == False:
         print(MySsid+' not available')
         w.init(mode=WLAN.AP, ssid='ApLost')
print('END of Job')
```

results in:

```
INIT
HOMENETWORK NOT AVAILABLE
END OF JOB
```

5.5 Socket

The following script shows how to use the built-in socket module to open a socket server, listen for incoming messages and, if the command is recognized, execute the required

actions. This is also a good example to understand how to use the WIND module to handle incoming messages and data. The listen command must use "8" as argument value.

```
from pyb import WIND
from usocket import socket
import utime
from pyb import LED
class LightBulb:
          def __init__(self, myLed, start):
                   print('INIT')
                   self.newDataToProcess = 0
                   self.led = myLed
                   self.s = socket(usocket.AF INET, usocket.SOCK STREAM)
                   try:
                             self.s.bind(2121)
                             self.s.listen(8)
                             self.myStart = True
                   except:
                             self.myStart = False
          def processMsg(self):
                  tuple_data = self.s.recvfrom(1024)
                  cmd = tuple data[0].rstrip()
                  if b'on' == cmd:
                            print('Turn on the lamp')
                            self.led.on()
                  elif b'off' == cmd:
                               print('Turn off the lamp')
                               self.led.off()
                  elif b'stop' == cmd:
                               self.myStart = False
                  else:
                               print('Unknown command received')
                  self.newDataToProcess = 0
                  return
          def run(self):
                  try:
                               self.s.send("\n\rrun")
                               pass
                  except:
                               pass
                  if self.newDataToProcess == 1:
                               self.processMsg()
                  return
def cb():
          try:
                  if w.code() == 55:
                           x.newDataToProcess = 1
          except:
                  print("Error on callback execution")
          return
l = LED(1)
l.on()
canContinue = 0
x = LightBulb(l, canContinue)
w = WIND()
w.callback(cb)
while x.myStart:
         x.run()
         utime.sleep(1)
print('END of Job')
```

results in:

```
INIT
TURN ON THE LAMP
TURN OFF THE LAMP
END OF JOB
```

5.6 Garbage collector

The following basic test shows how to use garbage collector in a script and check memory availability during script execution.

```
import gc

def func():
        i=0
        A='HELLO'
        while(i<10):
                A=A*2
                print(gc.mem_free())
                i=i+1

        print(gc.mem_free())
        print('END FUNC')
        return

print('RUN 1')
print(gc.mem free())
func()
print('TRIGGER GC')
gc.collect()
print('RUN 2')
func()
print('TRIGGER GC')
gc.collect()
print(gc.mem_free())
```

This results in:

```
RUN 1
30832
30800
30752
30688
30576
30384
30032
29360
28048
25456
20304
20304
END FUNC
TRIGGER GC
RUN 2
20432
20384
20320
20208
20016
19664
18992
17680
15088
9936
9936
END FUNC
TRIGGER GC
24592
```

5.7 How to use hardware peripherals

5.7.1 UART

The script below shows how to use an STM32 UART1 peripheral. In particular, the UART1 port is used here to interface the SPWF04 module directly with the SPBT2632 Bluetooth module.

The reserved circular buffer used to store received asynchronous message is 1024 bytes.

Handle communication properly to avoid data loss.

Hardware flow control enabled at peripheral initialization is highly recommended.

The full AT command parsing can be directly handled using MicroPython without any extra external host.

```
from pyb import UART
import utime
uart = UART(1,115200)
uart.init(115200,bits=8,stop=1,parity=None)
temp = uart.write('at+ab discovery')
uart.writechar(0x0D)
data = False
while data == False:
         data = uart.any()
         utime.sleep(1)
s = bytearray(256)
temp = uart.readinto(s)
print(s)
```

This results in:

```
bytearray(b'AT-AB InqPending\r\nAT-AB DiscoveryPending 5\r\nAT-AB Device
dc5360908110
"AGRCWL5387"\r\nAT-AB Device 0080e2fedccd "Unknown"\r\nAT-AB Device 402cf464fcf0
"AGRCWL2024"\r\nAT-AB Device 000272cc9405 "Unknown"\r\nAT-AB Device 0080e1abcdef
"jenkins 4.0"\r\n\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0')
```

5.7.2 SPI and PIN

In this example, the SPI built-in module interfaces an LIS3L accelerometer with the SPWF04 module through the SPI(1) interface. The SPI module allows configuring the peripheral as a master and starts reading and writing operations in a very simple way. GPIO3 is used as CS.

In this specific case, once the device interface has been configured, the script reads the “WHO_AM_I” register verifying the return value.

```
from pyb import Pin
from pyb import SPI
cs = Pin.board.GPIO3
cs.init(Pin.OUT)
cs.high()
spi = SPI(1,SPI.MASTER,baudrate=400000,polarity=1,phase=2,bits=8,firstbit=SPI.MSB)
cs.low()
spi.send(b'\x20\xC7')
cs.high()
expected = b’\xff\x3a’
cs.low()
whoami = spi.send_recv(b'\x8F\xFF')
cs.high()
if whoami != expected:
```

```
        print('Cannot detect accelerometer!')
else:
        print('Accelerometer detected!')
```

This results in:

```
Accelerometer detected!
```

5.7.3 I²C

The following example shows how to use I²C built-in class. In particular, the X-NUCLEO-IKS01A1 MEMS inertial and environmental sensor board is used here to interface the SPWF04 module using the I²C(3) peripheral.

It is designed around STMicroelectronics:

- LSM6DS0 3-axis accelerometer
- LIS3MDL 3-axis magnetometer
- HTS221 humidity and temperature sensor
- LPS25HB pressure sensor.

The script initializes the peripheral and performs a scan operation to detect all the connected slaves, printing the I²C address list.

In this example, a non-existing device is also checked to show how to handle exception cases.

```
from pyb import I2C
i2c = I2C(3,I2C.MASTER)
i2c.init(I2C.MASTER,baudrate=20000)
if i2c.is_ready(11) != True:
           print('Cannot detect LCD Display!')
           i2c.scan()
else:
           print('LCD Display detected!')
```

This results in:

```
Cannot detect LCD Display!
[30, 59, 93, 95, 107]
```

5.7.4 ADC

A built-in ADC module facilitates STM32 ADC peripheral management.

```
from pyb import ADC
import utime
a = ADC(1)
while True:
         print(a.read())
         utime.sleep(1)
```

Using a 2 kΩ trimmer on GPIO1 (min. to max. shift, then max. to min. again), this results in:

```
3299
2712
2151
1349
929
0
0
0
217
762
2325
3299
```

```
…
```

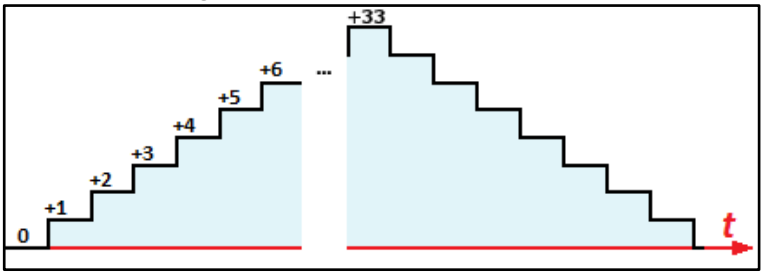
5.7.5 DAC

A simple MicroPython script example shows how to deal with STM32 DAC peripheral, running on SPWF04S Wi-Fi module:

```
from pyb import DAC
import utime
counter = 0
up = True
d = DAC(1)
while True:
        d.write(counter)
        if up == True:
                counter = counter + 100
                if counter == 3300:
                        up = False
        else:
                counter = counter - 100
                if counter == 0:
                        up = True
        utime.sleep_ms(100)
```

This results in the following waveform on GPIO15 (split into 33 steps on each side):

Figure 3: DAC waveform on GPIO15



6 Tips and tricks

6.1 RAM allocation

When an object is created or grows in size (for example when an item is added to a list) the necessary RAM is allocated from a block called heap. This takes a significant amount of time and sometimes triggers a process known as garbage collection which can last several milliseconds.

Consequently the performance of a function or method can be improved if an object is created only once and not allowed to grow in size. This implies that the object persists for the duration of its use; typically, it is instantiated in a class constructor and used in various methods (refer to *Section 6.6.2: "Controlling garbage collection"* for further details).

6.2 Buffers

When accessing devices such as instances of UART, I²C and SPI interfaces, using pre-allocated buffers avoids the creation of needless objects. Consider these two loops:

```
while True:
var = spi.read(100)
# process data
buf = bytearray(100)
while True:
spi.readinto(buf)
# process data in bufis passed
```

The first creates a buffer on each pass, whereas the second re-uses a pre-allocated buffer.

The latter solution is faster and more efficient in terms of memory fragmentation.

6.3 Byte use

It is recommended to use bytes instead of integers as they use a smaller amount of RAM.

On most platforms, an integer consumes four bytes.

Consider the two calls to the function foo():

```
def foo(bar):
for x in bar:
print(x)
foo((1, 2, 0xff))
foo(b'\1\2\xff')
```

In the first call a tuple of integers is created in the RAM; instead, the second call efficiently creates a byte object consuming the minimum amount of RAM. If the module is frozen as bytecode, the bytes object resides in the Flash memory.

6.4 Arrays

Consider the use of the various types of array classes as an alternative to lists. The array module supports several element types with 8-bit elements supported by Python built-in bytes and bytearray classes. These data structures store elements in contiguous memory locations. Therefore, to avoid memory allocation in critical code, these should be pre-allocated and elaborated as arguments or as bound objects.

When elaborating portions of objects such as bytearray instances, Python creates a copy which allocates a size proportional to the size of the portion. This can be minimized using a memoryview object, which is allocated on a heap and is a small, fixed-size object.

```
ba = bytearray(10000) # big array
func(ba[30:2000]) # a copy is passed, ~2K new allocation
mv = memoryview(ba) # small object is allocated
func(mv[30:2000]) # a pointer to memory is passed
```

A memoryview can only be applied to objects supporting the buffer protocol (including arrays but not lists). While the memoryview object is live, it also keeps the original buffer object alive. For instance, in the example above, if you have a 10K buffer and just need bytes 30:2000 from it, it may be better to make a portion, and release the 10K buffer (ready for garbage collection), instead of making a long-living memoryview and keeping 10K blocked for garbage collection.

Nonetheless, memoryview is indispensable for advanced pre-allocated buffer management.

The .readinto() method puts data at the beginning of the buffer and fills the entire buffer.

If you need to put data in the middle of an existing buffer, create a memoryview in the buffer section and pass it to .readinto().

6.5 Heap management

When a running program instantiates an object, the necessary RAM is allocated from a fixed size pool known as the heap. When the object goes out of scope (it becomes inaccessible to the code), the redundant object is called garbage.

A process known as garbage collection (GC) reclaims that memory, returning it to the free heap. This process runs automatically, but it can also be directly invoked by issuing gc.collect().

In a complex or long running program the heap can become fragmented: even if there is a substantial amount of available RAM, there is insufficient contiguous space to allocate a particular object and the program fails with a memory error.

To limit fragmentation problems, it is suggested to instantiate, when possible, large permanent buffers in the early execution phase of your program.

You can monitor the heap status using some available functions in the GC and MicroPython modules:

- gc.collect(): forces a garbage collection;
- micropython.mem_info(): prints a summary of RAM allocation;
- gc.mem_free(): returns the free heap size in bytes;
- gc.mem_alloc(): returns the number of bytes currently allocated;
- micropython.mem_info(1): prints a table of heap utilization (refer to the following table).

Table 4: micropython.mem_info(1) output

Symbol	Meaning
.	free block
h	head block
=	tail block
m	marked head block
T	tuple

Symbol	Meaning
L	list
D	dict
F	float
B	byte code
M	module

6.6 MicroPython extras

6.6.1 Catching object references

When a function or method repeatedly access objects, the performance is improved by caching the object in a local variable.

6.6.2 Controlling garbage collection

When memory allocation is required, MicroPython attempts to locate an adequate size block on the heap.

This operation might fail, usually because the heap is cluttered with objects which are no longer referenced to by code.

If a failure occurs, the garbage collection reclaims the memory used by these redundant objects and the allocation is then tried again (a process which can last several milliseconds).

Doing a periodic garbage collection (`gc.collect()`) before it is actually required is quicker (typically 1ms if done frequently). Furthermore, you can determine the point in the code where this time is used rather than have a longer delay occurring at random points, possibly in a critical section. Finally, performing regular collections can reduce heap fragmentation (as severe fragmentation may lead to non-recoverable allocation failures).

6.6.3 Compilation phase optimization

When a module is imported, MicroPython compiles the code in bytecode which is then executed by the MicroPython virtual machine (VM). The bytecode is stored in the RAM.

The compiler itself requires RAM, but this becomes available only when the compilation stops.

If several modules have already been imported, there could be an insufficient amount of RAM to run the compiler: in this case the import statement creates a memory exception.

If a module instantiates global objects to import, it allocates RAM which is then unavailable for subsequent imports.

To maximize the RAM available to the compiler, the initialization code should be run by the application after all the modules have been imported.

An offline compiler is not yet available for the SPWF04 platform. The use of a precompiled script helps to reduce the amount of RAM required at boot time.

6.6.4 String operation

Consider the following code fragments which aim at producing constant strings:

```
var = "foo" + "bar"
var1 = "foo" "bar"
```

```
var2 = """\
foo\
bar"""
```

All of them return the same output, but the first one creates two string objects at runtime, allocating more RAM for concatenation before producing the third. The others perform the concatenation when compiling, thus reducing fragmentation.

Rather than creating a large string object, create a substring and feed it to the stream before dealing with the next.

The best way to create dynamic strings is by means of the string format method:

```
X=5.3
var="test {:5.4f}\n".format(temp)
print(var)
```

Nevertheless, when a module is compiled, strings which occur multiple times are stored only once (string interning). In MicroPython an interned string is called qstr. In an imported module normally that single instance is located in the RAM.

String comparisons are also performed efficiently using hashing rather than character by character.

6.6.5 Object references

MicroPython passes, returns and (by default) copies objects by reference. A reference occupies a single machine word so these processes are efficient in terms of RAM use and speed.

Where variables whose size is neither a byte nor a machine word are required, there are standard libraries which can efficiently assist in storage and conversion performance.

7 Features not supported in MicroPython

The MicroPython version currently embedded in the SPWF04 module does not support the following features:

- pre-compiled script execution;
- inline assembler for Thumb-2 architecture;
- garbage collector thresholding;
- REPL auto-completion;
- direct access to hardware;
- byte code emitter is the only one supported (no possibility of using Native or Viper code emitter).

8 References

1. MicroPython homepage: *http://micropython.org/*
2. Documentation: *http://docs.micropython.org/*
3. Forum: *http://forum.micropython.org/*
4. Wiki: *http://wiki.micropython.org/*
5. Tutorials: *https://micropython.org/doc/tut-contents*
6. Source code repository: *https://github.com/micropython/micropython*
7. Garbage collector quick tour: *https://segfault.net.nz/posts/2014-06-07-micropython:-a-quick-gc-tour.html*
8. Python 3.X: *https://docs.python.org/3.4/whatsnew/3.0.html#print-is-a-function*
9. Pyboard MicroPython Documentation: *http://docs.micropython.org/en/latest/micropython-pyboard.pdf*

9 Revision history

Table 5: Document revision history

Date	Version	Changes
17-Jan-2017	1	Initial release.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.