

1 Introduction

This application note introduces the porting of MicroPython, the packaging of peripheral functions, and the adaptation of circuit boards, using the example of our work on the i.MX RT1050/1060EVK development board. The code are mainly written in C language, but are presented to the users as Python modules and types. You can either evaluate and use MicroPython on this development board, or use it to port and adapt your new board design. MicroPython's native project management and build environment is based on GCC and Make under Linux. To facilitate the development habits of most MCU embedded engineers, the development environment is also ported to Keil MDK5.

The readers are expected to have basic experience of development with KEIL MDK, knowing what is CMSIS-Pack, the concept of **target** in KEIL and how to switch between them.

2 Hardware platform

2.1 i.MX RT1050/60 crossover process

The [i.MX RT1050/60](#) offered by NXP with single Arm® Cortex®-M7 core can operate at the speed up to 600 MHz. It contains:

- 512 KB on-chip RAM, which can be flexibly configured as core Tightly-Coupled Memory (TCM) or general-purpose RAM
- Additional dedicated 512 KB of OCRAM
- Various interfaces for connecting various external memories
- A wide range of serial communication interfaces, such as USB, Ethernet, SDIO, CAN, UART, I2C, and SPI
- Rich audio and video features, including LCD display, basic 2D graphics, camera interface, SPDIF and I2S audio interface
- Various modules for security, motor control, analog signal processing, and power management

2.2 i.MX RT1050/60 EVK board

[i.MX RT1050 EVKB/1060 EVKB](#) board is a platform designed to show the most commonly used features of the i.MX RT1050 processor. The EVK board offers the below features:

- Memory: 256 Mbit SDRAM, 64 Mbit Quad SPI Flash, 512 Mbit Hyper Flash, TF Card Slot
- Communication interfaces: USB 2.0 OTG connector, USB 2.0 host connector, 10/100 Mbit/s Ethernet connector, CAN bus connector

Contents

1	Introduction.....	1
2	Hardware platform.....	1
2.1	i.MX RT1050/60 crossover process.....	1
2.2	i.MX RT1050/60 EVK board.....	1
3	Micropython.....	2
3.1	Brief introduction to Python Language.....	2
3.2	Brief introduction to Micropython	3
4	Building and running Micropython on i.MX RT1050/1060 EVK.....	3
4.1	Downloading source code.....	3
4.2	Opening project with KEIL and build firmware.....	3
4.3	Preparing the file system.....	5
4.4	Downloading and running.....	5
5	Accessing Micropython file system on PC.....	7
6	Practicing Python development with Micropython.....	7
6.1	Programming with REPL.....	8
6.2	Creating script in TF card and the boot.py and the main.py.....	11
6.3	Accessing Micropython file system with Python code.....	11
7	Basic software resources in Micropython.....	12
7.1	Key folders of Micropython.....	12
7.2	Micropython key built-in features	12
8	Libraries in Micropython.....	13
8.1	Modules and types.....	13
8.2	Micro libraries.....	13
8.3	Libraries related to MCU and board.....	13
9	Check and use the included libraries in your Micropython system.....	14
10	Using the garbage collector wisely	16
11	Get more help.....	17
12	References.....	17
13	Revision history.....	17



- Multimedia interfaces: CMOS sensor connector, LCD connector
- Audio interfaces: 3.5 mm stereo headphone jack, board-mounted microphone, SPDIF connector (not mounted by default)
- Debug interfaces: On-board debug adapter with DAP-Link, JTAG 20-pin connector
- Arduino interface
- User button and LEDs

Figure 1 is a photograph of i.MX RT1050 EVKB.

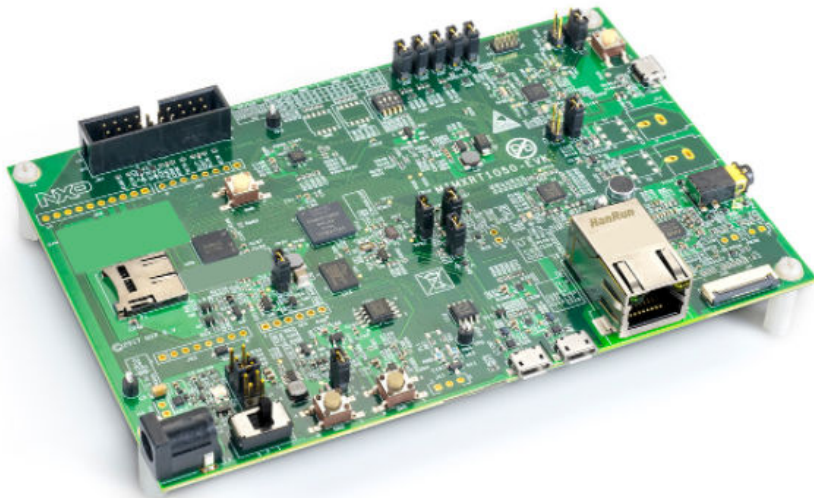


Figure 1. i.MX RT1050 EVKB

3 Micropython

3.1 Brief introduction to Python Language

Python is very friendly to beginner. The learning curve starts with almost zero slope and even teenagers with no computer knowledge can get up to speed. In addition, Python is expressive, with one line of code often superior to multiple lines of C. Python also provides common container data structures, such as linear tables, dictionaries (hash tables), collections, arrays, etc., and related manipulation functions. Variables in Python are objects, and the use of variables is almost always by address reference. Therefore, the contents contained in them can be vividly thought of as **void***, which makes the heterogeneous and nested have natural support and is good at expressing the dynamic data structure that changes during runtime. Of course, the Python runtime environment knows their actual types. On top of that, the arguments and return values of Python functions are very flexible, completely reversing the machine-oriented philosophy of C, making the Python API seem very concise and simple, but also very easy to use and powerful.

Python's support for strings, large integer operations, and especially strings is a leap from the Stone Age to the Information Age compared to C. A common saying among programmers, **Life is short, I use Python**, is to emphasize that Python makes it easy to program efficiently and to get to the point without having to deal with much underlying details of the computer. The current mainstream version of Python is 3.x.

Of course, these benefits of Python are not free, but rather a way to build a programmer's easier life on a CPU's heavier load. This makes using pure Python code on a computer inefficient, and sacrifice real-time capabilities. As a result, it is often Python that provides the API while the underlying work is done with more efficient libraries such as C/C++.

3.2 Brief introduction to Micropython

Don't be misled by the **micro** in the name MicroPython. For a Micro Controller Unit (MCU), when the two **micro**s cancel each other out, it's still a large software project with tens of thousands of lines of source code.

MicroPython is alternate implementation of Python 3.6 that supports all the common Python syntax. MicroPython is also a compact version of Python. It is designed to run on a microcontroller with limited performance, such as a single chip MCU. The minimum code size is less than 256 K and the runtime requires only 16 K of memory. The poor configurations can only run the simplest scripts. For a more fully functional configuration, 512 KB Flash and 64 KB RAM are recommended, or greater. For the i.MX RT series, resources have far richer.

NOTE

In a dynamic language like Python, program code is often called **scripts**.

Micropython tailors most of the standard library to accommodate embedded MCUs, keeping only a few modules such as math and some functions and classes for **sys** module. In addition, many standard modules, such as JSON and RE, have become uJSON and uRE, which begin with **u**, representing the stripped down version of standard libraries for MicroPython development. Currently, Micropython can run on a large number of ARM-based embedded systems, in addition to the PyBoard microcontrollers originally developed. In this article, we introduced the porting of Micropython to NXP i.MX RT1050/1060 and changed the development environment from the official GCC+Make to the Keil MDK 5.

The **micro** in MicroPython makes it easy to think that it's just reducing functionality on Python. In fact, it has also added new features for use on the MCU. For example, on MicroPython, the performance and real-time requirements can still be programmed in C, and the interface bound to Python can be exported, generally achieving a balance of ease of use and serious development. MicroPython itself also supports more efficient and lower-level operations in the MCU with a number of special extensions and modules, including the innovative **native** support and **viper** support that do not use dynamic memory for performance. A set of "Pandora box" like modules, mem8/16/32 modules provide way to directly access the 4GB memory address; Micropython also support inline assembly in Python scripts.

Micropython official website is <http://www.micropython.org/>. Micropython code is open source with the friendly MIT license and the repository is located at <https://github.com/micropython/micropython>.

4 Building and running Micropython on i.MX RT1050/1060 EVK

4.1 Downloading source code

4.1.1 Downloading the specific version

Download the source code (zip) from [micropython-rocky](#) and unzip it. We strongly recommend to download the version with the **an_mpy1050_rev1** tag, as described in this application note.

4.1.2 git clone

Open a command window, navigate to the directory you want to put in, and then execute the command.

```
git clone https://github.com/RockySong/micropython-rocky.git
```

By default, clone the `omv_initial_integrate` branch. To get exactly the same result, check out to `an_mpy_rt1050_60`.

NOTE

When meeting any issue with latest codes, switch to the **an_mpy1050_rev1** tag.

4.2 Opening project with KEIL and build firmware

Based on **an_mpy1050_rev1** version, locate the `|ports|prj_keil_rt1060|mpyrt1060.uvprojx` and open it with KEIL 5.0 or above.

NOTE

This project requires *NXP.MIMXRT1062_DFP.12.1.0* and it is a CMSIS pack.

This project has multiple targets as shown in [Figure 2](#).

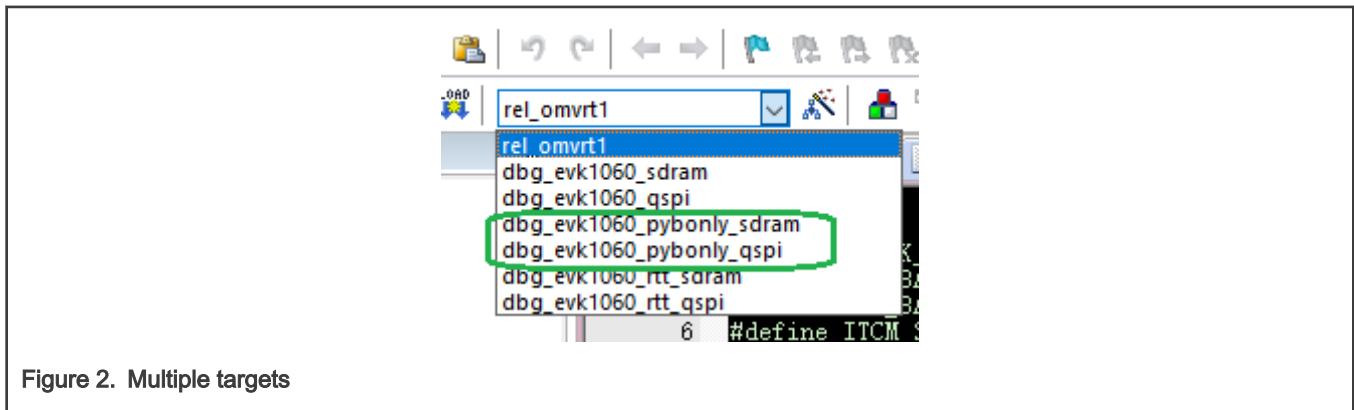



Figure 2. Multiple targets

Select the target with the **pybonly** keyword, which means **contains MicroPython only, not OpenMV main functionality**. There are two: one is debugged in SDRAM and the other is debugged in QSPI Flash. Because debugging in SDRAM is the most convenient, it is generally recommended to choose debugging in SDRAM. Whichever option you choose, click

 or press **F7** to generate the entire project.

If you get error messages shown in [Figure 3](#), it means the specified AC5 compiler version is not found.

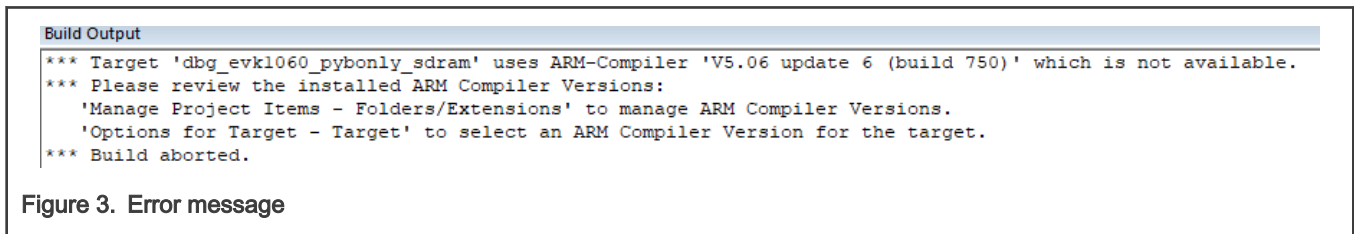



Figure 3. Error message

In this case, click 

or press **Alt+F7**, and the **Options of Targets** dialog appears. Switch to the **Target** tab. In the **Code Generation** frame, select the **Use default compiler 5** in the **ARM Compiler** combo box.

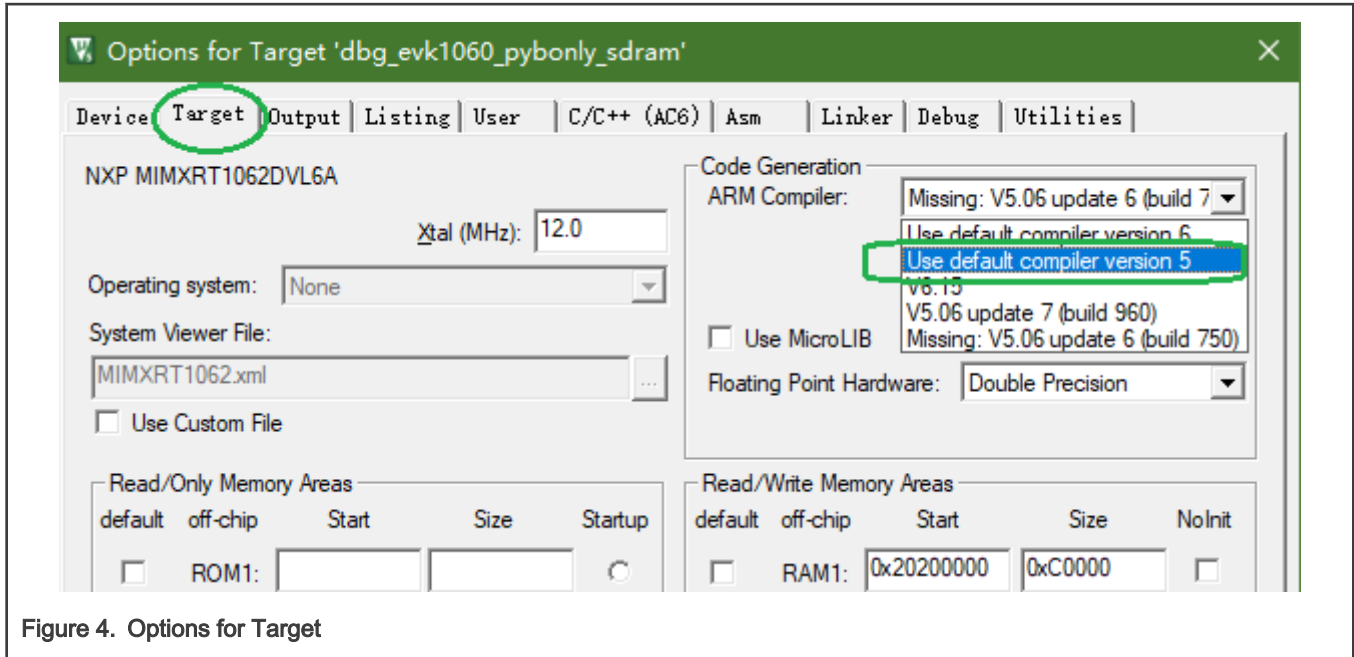


Figure 4. Options for Target

4.3 Preparing the file system

Under MicroPython, the concept of a file system is extremely important, and all Python scripts are stored on the file system. In our port, both the file system on the TF/microSD card and the file system in the program Flash are supported. During startup, the system first checks whether the TF card is inserted, and if so, mounts the TF card into the root file system. If not, then mount a block of approximately 2 MB of space allocated from QSPI Flash as the file system. If it is used for the first time, the system will automatically format this area in the QSPI Flash.

The Flash file system makes it possible to run MicroPython without a TF card. However, its write performance is extremely poor (around 10KB/s) and it does not include wear balance. It is generally recommended to place only files that do not change frequently, such as configuration files, debugged scripts, and so on, into Flash file system. In addition, when writing data to the Flash file system, turn off the interrupt, affecting real-time response. Therefore, it is highly recommended to insert a TF card formatted using FAT/FAT32 on the board.

The below describes the way to access the file system.

4.4 Downloading and running

1. Determine which debugger you are using. i.MX RT1060evk has an on-board CMSIS-DAP compatible debugger, but due to the large firmware generated by this project, the download with CMSIS-DAP is slow. J-Link can also be used. On the 1060EVK, to use J-Link, disconnect J47 and J48 or shortcut them if you use the onboard CMSIS-DAP compatible debugger. Under Keil, J-Link downloads far faster than the on-board CMSIS-DAP. As the firmware has grown several times since the inclusion of OpenMV, it can save lots of time to download it using J-Link.

NOTE

If J-Link is used, download it to SDRAM, click the **Reset** button again before executing the program. Otherwise you may accidentally enter a hard fault.

2. Connect the i.MX RT1060evk as shown in [Figure 5](#).

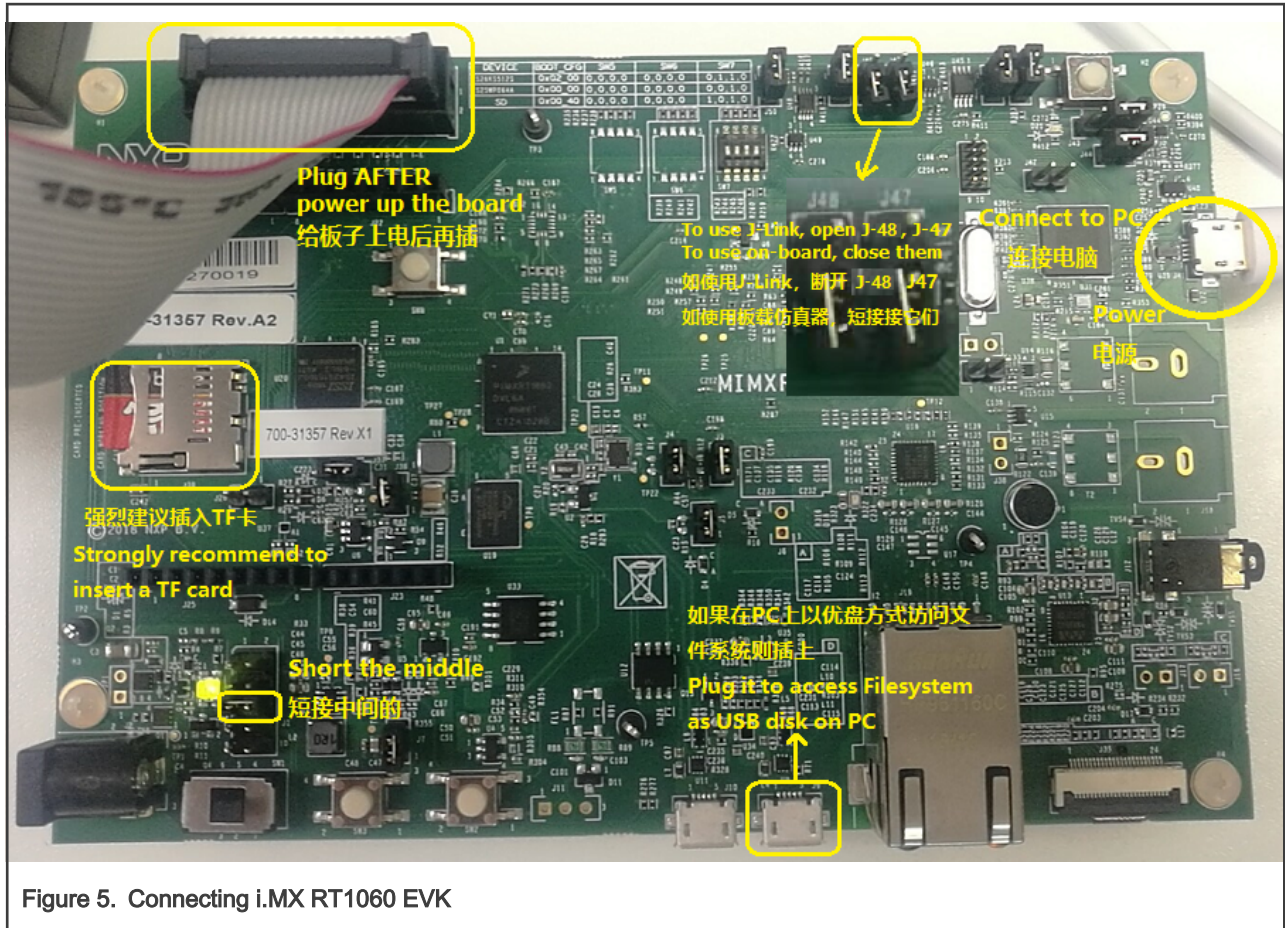



Figure 5. Connecting i.MX RT1060 EVK

3. Open a serial port terminal, such as TeraTM, to connect to the virtual serial port presented by the development board's on-board debugger, with the baud rate set to 115200.
4. If debugging in SDRAM, click  or press **Ctrl-F5** after building. Press **F5** to run at full speed when the program enters the debugging interface after downloading.
5. Switch to the serial terminal, wait for a moment, and a small number of boot messages appear. Enter the >>> prompt and try `print (' Hello MicroPython!)`, as shown in [Figure 6](#).

```
Card inserted.
Executing boot.py
USB device composite demo
Unique ID: e9ba b96f fa5a b66e
loading /cmm_cfg.csv
PYB: sync filesystems
PYB: soft reboot
Executing boot.py                               Set up time to wake up an alarm.
Unique ID: e9ba b96f fa5a b66e
loading /cmm_cfg.csv
MicroPython 180723_milestone-122-g2079cc9-dirty on 2020-05-08; mimxrt1050-evk wi
th i.MX RT105x
Type "help()" for more information.
>>> print('hello micropython!')
hello micropython!
>>>
```

Figure 6. Boot message

Now, MicroPython is ready in this interactive development environment. It is called REPL. Python on the PC can also be used at the command line in a similar way, which is the simplest form of programming in Python.

5 Accessing Micropython file system on PC

The Access to the MicroPython file system from the PC is extremely important because of the lack of a good Python editor on MicroPython. We often write Python code on a PC using our favorite editor. The written code needs to be downloaded to the MicroPython file system. For this, we implemented the USB mass storage device class in the port, and presented the file system in the form of a USB flash drive.

To access the contents of the file system on your PC, connect to the computer on a USB port near the Ethernet port, which will present a removable disk on the computer.

- When no TF card is inserted, the contents of the Flash file system are displayed on the removable disk;
- When the TF card is inserted, the contents of the TF card are displayed on the removable disk.

When accessing the Flash file system, the write speed is only about 10KB/s. When accessing the TF card, the general read and write speed is about 10MB/s.

NOTE

Do NOT put a directory named **flash** (case sensitive) in your TF card. This will cause your Python script still accessing the Flash file system when accessing **/flash**, while seeing the contents of the TF card on your PC in **/flash**.

6 Practicing Python development with Micropython

We've had our first taste of MicroPython programming on the REPL since typing **Hello MicroPython** in [Downloading and running](#). Similar to the experience on the PC, this interactive approach is great for getting started quickly, or trying out a new feature or even a short piece of software. However, it is difficult to write large Python scripts this way. Fortunately, there are many ways to develop, deploy, and execute Python scripts in MicroPython. [Figure 7](#) summarizes three ways:

- Load the script from a file
- Real-time execution from an interactive terminal (as we've seen)
- Cross-compiling and statically link of generated bytecode into firmware

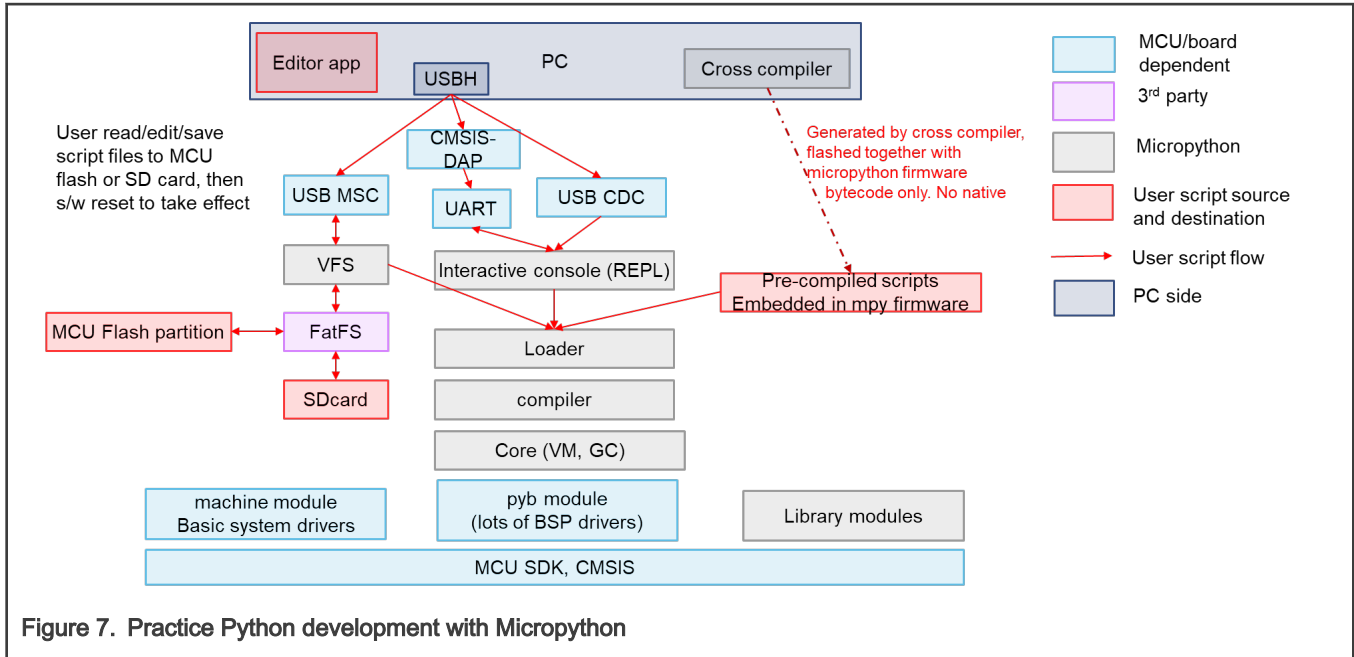


Figure 7. Practice Python development with MicroPython

One path on the left is to load the script from the file system and generate the bytecode by the MicroPython loader and compiler and submit it to the core virtual machine. The file system can be on a TF card or in the program Flash.

Taking the middle two paths, the rest of the steps are similar to those taken from the file system. Although you can either use the MCU's USB virtual serial port or the MCU's UART via the virtual serial port of the onboard debugger, respectively, fetch Python lines or segments from the REPL component.

Executing from the file system and from the REPL is the focus of the rest of this application note.

Here is also a brief introduction to the right side of the road, which is closer to the C language project development habits. We need to compile Python code into bytecode that MicroPython's virtual machine can interpret using MicroPython's cross-compiler on the host PC, a program called **mpy-cross**. The bytecode is encapsulated into a special executable object, serialized into C source code, placed into a MicroPython project, and compiled and linked with other source code.

In this way, the MicroPython compiler can be cropped out of the firmware, saving a small amount of Flash space. However, since this approach sacrifices the convenience of running different code quickly, it is not the purpose of our port, so we won't dig deeper here. Some MicroPython-based variants, such as Micro:bit, uses this approach. It builds your firmware in its server side and lets you download the generated firmware with your browser.

6.1 Programming with REPL

The most straightforward way to get started with MicroPython is through the REPL, which is very similar to typing line by line commands in a command-line interface, such as `print(' Hello MicroPython!')`. However, the REPL is more than a simple command interpreter. It is more like typing a script line by line and executing it immediately after each line is typed. The results of previously executed scripts can be used later.

Besides the line-by-line input and script execution, the below lists important features of the REPL

6.1.1 Breaking a running script

REPL supports breaking a running script with the **Ctrl-C** key, as shown in [Figure 8](#).


```
>>> i = 0
>>> while (True):
...     i += 1
...     Ctrl +C
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt:
>>>
>>>
```

Figure 8. Breaking a running script

As shown in Figure 8, the **KeyboardInterrupt** explains why the script is being broken. The **KeyboardInterrupt** exception has occurred. At implementation time, the exception is raised in the serial port interrupt service routine and detected by the MicroPython VM.

6.1.2 Multi-line script input in REPL

Under REPL, when a line ends with `:`, the REPL does not directly execute the line, but automatically indents it and allows manual input of the script that follows. After that, manually adjust the indentation level based on the program syntax and semantics. When the indentation level is `0`, the REPL assumes that a script has been typed and executes it together.

However, it is not convenient to manually enter multiline scripts directly under the REPL in this way. In addition, if you have a multiline script with indent level 0, you cannot enter multiple lines as a whole. To do this, the REPL provides a **paste mode**.

1. Press **Ctrl-E** on a blank line in the REPL to enter the paste mode.

```
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
===
```

Figure 9. Entering paste mode

2. Copy a pre-edited script, such as :

```
1 def fib_sum(n):
2     s = 2
3     s0 = 1
4     for i in range(2, n):
5         tmp = s0 + s
6         s0 = s
7         s = tmp
8     return s
9 print('Fibonacci to 10 = %d' % fib_sum(10))
```

Figure 10. Editing multiple lines of python script on PC

```
def fib_sum(n):
    s = 2
    s0 = 1
    for i in range(2, n):
        tmp = s0 + s
```

```
s0 = s
s = tmp
return s
print('Fibonacci to 10 = %d' % fib_sum(10))
```

3. Paste it to REPL within the terminal (usually the hot key is the right mouse button), for example:

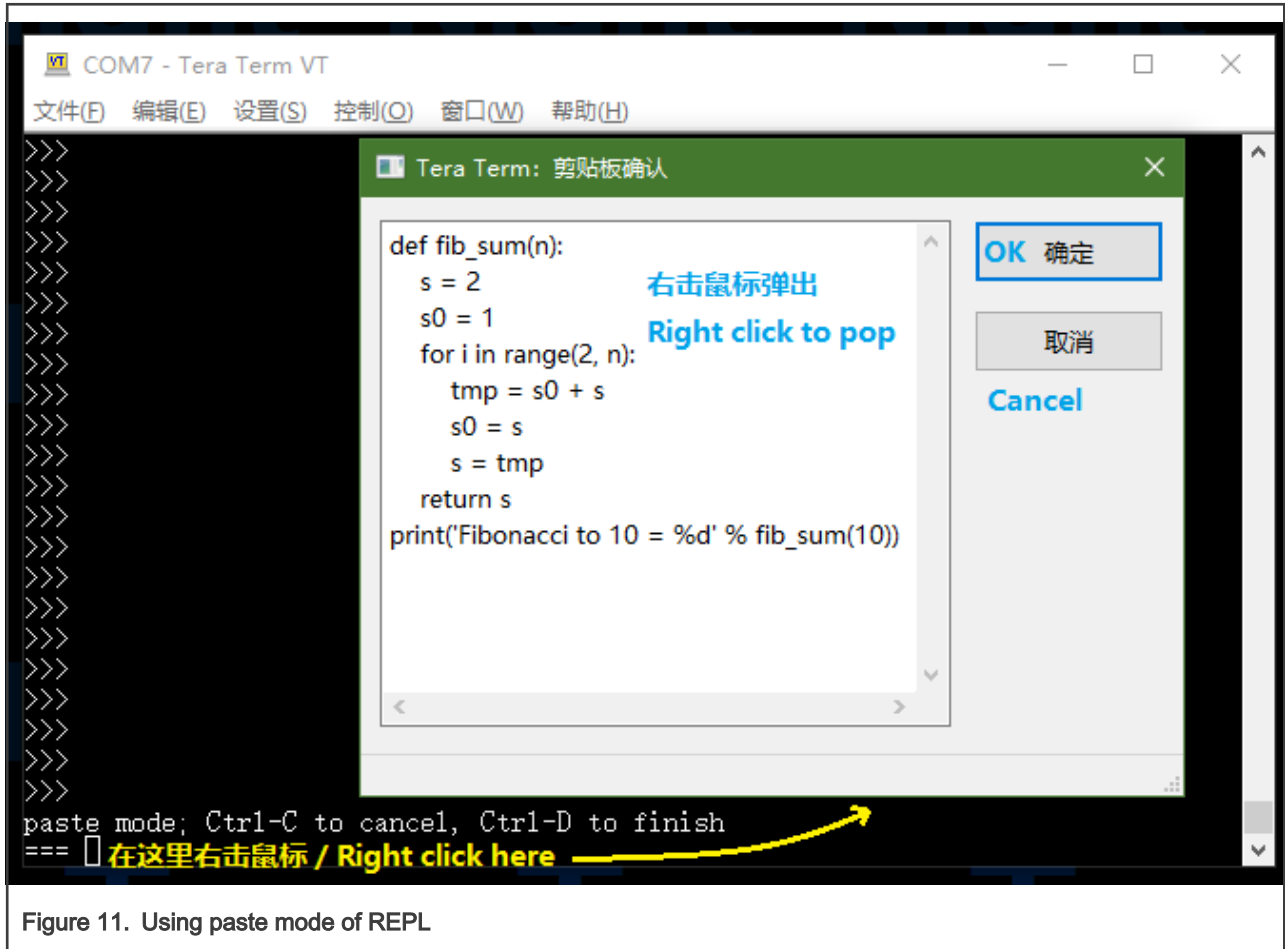
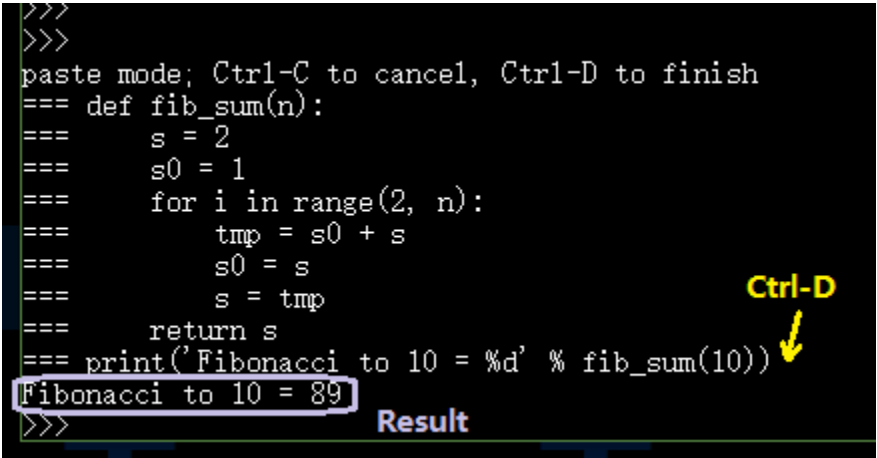


Figure 11. Using paste mode of REPL

4. Click **OK** and the copied script snippet is pasted.
5. Press **Ctrl-D** to finish multi-line script input, and the script will be carried out, for example:



```

>>>
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
=== def fib_sum(n):
===     s = 2
===     s0 = 1
===     for i in range(2, n):
===         tmp = s0 + s
===         s0 = s
===         s = tmp
===     return s
=== print('Fibonacci to 10 = %d' % fib_sum(10))
Fibonacci to 10 = 89
>>>
Result

```

Figure 12. Running pasted script under REPL

6.2 Creating script in TF card and the boot.py and the main.py

While it is convenient and intuitive to use MicroPython directly on an interactive terminal, it is not easy to develop scripts with many lines of code or store them in files for repeated use. In fact, the real MicroPython-based project development is basically organized programs by files, which is the same as the traditional use of C language to develop embedded applications.

C programs start from main, and the entire firmware must be in a specified location and conform to a specific format in order to start successfully. How about in MicroPython?

Under MicroPython, the porting code somehow plays the role of **god**: the startup sequence can actually be designed by the port itself. But by convention, during startup, scan the root directory to find **boot.py** first and then **main.py**, load the startup code, and execute each of them as soon as one is found. Both files are optional and the runtime just silently skip one if it is not found. During our porting, we also followed this startup sequence.

6.2.1 boot.py

Where, `boot.py`, as the name implies, is executed during boot, acting as a boot script. For example, the `boot.py` script can read the state of keypads, jumpers, and dialed switches to read some configurations, such as what device the USB is on. At the time of execution of **boot.py**, most peripherals and software modules are not yet initialized, so `boot.py` can do little. Thus, if you do not have a custom boot process, you generally do not need `boot.py`.

6.2.2 main.py

The real entry point to keep in mind is `main.py`.

Main.py, as the name suggests, is like the `main` function in C, which runs the main business logic of the user program. `main.py` is automatically executed after all necessary initializations in the MicroPython system have been completed.

In addition, as on a PC, you can split the entire program functionality over multiple Python source files and place them in the root directory of a TF card. For other `xxx.py` files, they can be used in `main.py` by **import xxx**. These files can also be drivers or middleware developed in Python. Of course, they must also be executable by MicroPython.

6.3 Accessing Micropython file system with Python code

Access to the MicroPython file system is much the same as on a PC, using either the `OS` module or the `open(path, [mode])` function. In the MicroPython port corresponding to this application note, the contents of the TF card are loaded into the root directory / if there is a TF card. Otherwise the root directory is the virtual file system, and there is a subdirectory called `/flash` that represents the mount point of the QSPI Flash file system.

Whether or not a TF card is inserted, the QSPI Flash file system is mounted as **/flash** and can be browsed on the REPL with the following commands:

```
import os
os.listdir('/flash')
```

To access the contents of a file, use the **open** built-in function in a Python script to return the file object, and then use the read, write, seek, and other methods, just as using Python on a PC.

NOTE

Do not place a folder named **flash** (case sensitive) in the root directory of the TF card. Otherwise it will be obscured by the internal QSPI flash file system.

7 Basic software resources in Micropython

After some first experiences of Micropython, let's dive deeper.

7.1 Key folders of Micropython

It's best to learn something new from the surface to inside. So, let's start with a brief introduction to MicroPython's top-level directory partitioning. The directory structure of the MicroPython source code gives you an intuitive idea of how it functions and is organized. Below we highlight some key directories of Micropython source tree.

- **py**: This is the core of MicroPython and is responsible for the running of Python programs. It includes the compiler, Micropython bytecode generator, bytecode loader, virtual machine, dynamic memory manager with garbage collection support, built-in modules and classes, and more.
- **ports**: Ports of Micropython on multiple hardware/software platforms. Micropython can run on arm, RISC-V, xtensa, x86, either by bare metal on an operating system such as Windows, Linux, Zephyr, FreeRTOS.
- **drivers**: Off-chip peripheral drivers are usually the chips soldered on the same PCB of the main CPU. These drivers can be written in either C or Python.
- **extmod**: There are many non-builtin Micropython modules. Some of them, such as the vfs module, are often used in all ports.
- **lib**: Low level code in C that used by multiple Micropython modules.
- **mpy-cross**: A PC tool to cross compile Micropython source code to bytecode and encapsulate it into files. Bytecode can then be linked into firmware and burned to flash, and directly loaded by Micropython VM, without compiling on device. Micropython comes with the Makefile of mpy-cross for Linux.

7.2 Micropython key built-in features

When you're learning a language, once you understand its basic syntax, you shall get familiar with the key built-in features that programmers typically use the most. Below are some Micropython key built-in features.

- **Built-in functions and exceptions**: Similar to standard C Python, some of most frequently used are:
 - print, len, range, enumerate, open, dir, str, int; type, isinstance, sorted, zip, sum, chr, ord, hex, oct, bin, min, max, sum, map; iter, next; input
- **Heap manager with arbage collection (gc)**
 - Scans the potential memory blocks that are still in use, by (recursively) recognizing pointers to them from the call stack and a statically planned **root pointer** list, while frees other memory blocks.
 - Methods: enable, disable, collect, mem_alloc, mem_free, threshold
- Math library for **real number** and **complex number**
- Other micro libraries: uarray, ubinascii, ucollections, uhashlib, uos, ...

8 Libraries in Micropython

A considerable portion of learning a language is being familiar with its libraries. The more advanced the language, the more important it is to learn the library. C language library is relatively thin, C++ library is much richer. With Python, the language itself is very easy to get started with, and understanding and working with various libraries is even more important. The below is a brief introduction.

8.1 Modules and types

MicroPython libraries generally take one of two forms:

1. modules: Use by import `<module_name>`. Once a module is imported, you can use the methods and types within it. Methods in a module are just like functions and they do not have associated object instances. In some languages, method in modules are called **static methods**.
2. types: Types are included in modules. They act as classes in object-oriented languages, and encapsulate both data and operations as methods. In C, they are the structs and all functions that can work with them. To use a type, instantiate the type and get an object of that type, and then call methods on that object with the popular `object.method()` manner.

8.2 Micro libraries

As a compact implementation of Python 3, MicroPython also comes with commonly used Python libraries, most of which have been condensed into tiny libraries. The micro libraries begin with a `u` and implements only a subset of the functionality of the corresponding CPython modules. [Table 1](#) lists some commonly-used modules.

Table 1. Frequently used compact standard libraries in Micropython

Name	Usage
Built-in functions and exceptions	Use directly without the need of import
math, cmath	math libraries of real number and complex number
gc	Garbage collector
uarray	Array
uos	Micro os module
uio	Input stream and output stream
ustruct	Packing and unpacking of binary data blocks
sys/usys	System specific functions, including stdio
ujson	JSON encoding and decoding
ure	Regular Expression engine
uzlib	Zip uncompressing library

8.3 Libraries related to MCU and board

In MicroPython, the resources on the MCU and the circuit board are typically encapsulated through Python modules and objects created by Python modules. There are two manners of encapsulations:

8.3.1 Peripheral oriented manner

This is also the most straightforward encapsulation manner, as if it were the basic SDK used by Python to drive common peripherals. If you want to use Python modules and objects that are encapsulated in this way, then you need a basic understanding of the corresponding peripheral classes. For example, ADC, I2C, UART, SPI, and so on, their names are usually the same as the peripherals. Most of these functions are packaged in the **pyb** module and **machine** module. It is worth mentioning that, compared with the fundamental library in C (SDK) provided by NXP, these same peripheral-oriented modules only provide a subset of the most commonly used functions, not as complete as the SDK. They contains neither rare features nor much of a direct-to-register API.

Interestingly, however, some subset of functionality is not necessarily included by peripherals. For example, the **scan** method of the I2C class is used to use the I2C peripheral as the host and scan all connected I2C devices on the I2C bus. In general, the registers of a peripheral do not provide such advanced functionality directly. Moreover, in the C language, for such an API, it is not convenient to use a simple type as the return value, and it is likely to require manual allocation of dynamic memory and cleanup. However, under MicroPython, you simply return a list or tuple, with no concern about how long it should live, where the memory is, or how it is freed. This adds a great deal of flexibility to the design of the API under MicroPython.

8.3.2 Use case oriented manner

For libraries written in mind of use case oriented manner, they are more straightforward to specific use cases than peripheral-oriented APIs. For example, the LED type directly provides the basic LED control API but the underlying implementation is automatically using GPIO and PWM (for brightness dimming). They often provide methods that are more convenient for specific application scenarios.

There are some simple features, such as the PIN type, which encapsulates the functionality of GPIO and has the characteristics of both.

8.3.3 Machine module and pyb module

In the MicroPython port described in this document, there are two modules that encapsulate the capabilities of hardware resources on the chip and on the development board:

1. **machine** module: System-level basic operations and information, such as check the main frequency, sleep control, and even **root** level operations such as reset and direct read and write to 4 GB address space. More power, more responsibilities: Improper usage of them can break the normal operation of the whole system.
2. **pyb** module: This module encapsulates the functions of most of the popular MCU peripherals. Some of the most basic and common functions are also included in the machine module, such as the UART type and the `freq()` function.

The machine module and the pyb module first appeared in the PyBoard development board created by the original MicroPython authors. This port also mimics their use to encapsulate common MCU functions in the i.MX RT1050/1060 series processors.

9 Check and use the included libraries in your MicroPython system

The MicroPython system is highly customizable, where the customized libraries can be added during porting. To see modules included in the MicroPython system you are using, type the following command in the REPL

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

For example, a MicroPython system we configured on the i.MX RT1050/1060evk contains the following modules:

```
Type "help()" for more information.
>>> help('modules')
===== modules =====
__main__      math          ucollections  uos
builtins      mcu           uctypes       urandom
cmath         micropython  uerrno        ure
cmm           pyb          uhashlib      uselect
doc           sys          uheapq        ustruct
gc            time         uio           utime
lcd160cr      uarray       ujson         utimeq
lcd160cr_test ubinascii    umachine      uzlib
Plus any modules on the filesystem
>>>
```

Figure 13. Checking the built-in modules in a running Micropython system

A large number of MicroPython mini-libraries begin with u.

To see the methods and types in the module, import the module and use the `dir` built-in function. Under MicroPython, for `<module name>`. Press the **Tab** key, and the REPL will automatically pop the available methods, types, and constants (A special type). For example, [Figure 14](#) shows how to check out what is inside the `pyb` module.

```
>>> import pyb
>>> pyb.
class_      name_      main      stop
flash      LED        ADC        ADCall
Flash      Pin        SD         SDCard
Switch     UART       USB_VCP    bootloader
delay      disable_irq elapsed_micros elapsed_millis
enable_irq fault_debug freq        hard_reset
have_cdc   info       micros     millis
mount      repl_info  repl_uart  rng
rtc        standby   sync       udelay
uniqueID   unique id  usb mode   wfi
>>> pyb.
```

Figure 14. Populating contents of a module with Tab key

The `pyb` module encapsulates common functions of the MCU, which mimics the way MicroPython encapsulates MCU functions on its official PyBoard products. As show in [Figure 14](#), the `pyb` module contains many members. In general, lowercase is the method in the module and uppercase is the type or constant in the module. Use Python's built-in `type()` function to see what's behind each name. For example, `freq` is a function type (class `function`).

```
>>> type(pyb.freq)
<class 'function'>
>>> pyb.freq()
(24, 600, 600, 150)
>>> □
```

Figure 15. Getting the type of a member

After being invoked, it returns the 4-element tuple of the frequencies, in MHz, of four most important system clocks (Oscillator, CPU, AHB, Peripheral).

Similarly, `UART` is a type.

```
>>> type(pyb.UART)
<class 'type'>
>>>
```

Figure 16. UART

UART is the encapsulation type for the basic functions of UART under the **pyb** module. You can create an instance of this type and call its methods, for example:

```
>>> u1 = pyb.UART(1, baudrate=115200)
>>> u1.
__class__      __next__      any           read
readinto      readline      write         init
deinit        readchar     sendbreak     writechar
>>> u1.write('hello')
hello5
>>>
```

sizeof(hello)

Figure 17. Creating an object and invoking its method

As shown in [Figure 17](#), an object instance of type UART is created with **pyb.uart (1, baudrate=115200)** using the LPUART peripheral # 1, with baudrate set to **115200**, and referenced by name **u1**. Input **u1.<TAB key>**, and after that, the REPL automatically lists the methods supported by the UART type. For example, use the **write** method to write a string with UART.

After the **u1.write('hello')** command is typed, **hello5** is printed because strings in MicroPython also record the length of the string at the end, rather than ending in zero as in C. UART1 happens to be the UART used by the REPL on the 1060EVK, so its output can be captured and displayed by the REPL.

10 Using the garbage collector wisely

Once getting a basic knowledge of MicroPython, you need to learn a little more about garbage collection, which is the exact opposite of the memory management philosophy of programming in C - trust the programmers vs. don't trust the programmers!

In C, we have to manage memory ourselves, knowing the difference between static memory, stack memory, and heap memory, and how variables, especially pointer variables, are placed and scoped. In particular, dynamic memory obtained through malloc must be carefully managed for its lifetime and the number of Pointers, which can lead to hidden bugs. This is also the result of C's **trust the programmer** philosophy: while we enjoy the supreme power of memory use, we also have the responsibility of micromanaging memory. One of the most common annoyances is to pair each malloc with a **free**.

In Python, these rights and obligations are taken back. When using any Python objects, don't manually allocate and free their memory. The Python language runtime takes over, and so does in MicroPython. If you use Python just to make a computer system to work for you in a programmable manner, this way is undoubtedly a great liberation: to be a system programmer is no longer the prerequisite for being a Python programmer we can focus on our applications to solve the problem, and need not get bored by the tedious and error prone details of memory management.

In fact, objects in Python are still allocated from the heap memory, and the heap manager also has malloc and free methods. However, the key difference is, the heap manager in Micropython also keeps tracking other clues about which blocks of memory it is managing are still in use at any given moment, and automatically calls free for blocks not in use.

This feature, known in MicroPython as a **Garbage Collector (GC)**, is basically a block-based allocator with fixed size, except that it can allocate multiple contiguous blocks once at a time. The length of a block is a power of 2, typically 16 bytes or more. GC uses a 2-bit bitmap array to track the allocation of individual blocks. When the GC finds that there is no available memory to allocate, it starts the garbage collection operation. The GC scans the script's stack and current registers, as well as a special **root pointer** list, for any 32-bit variables that can be read as Pointers and point to its own allocated memory. Whenever it is found, all chunks of this memory are marked as **non-garbage** it also scans the contents of all **non-garbage** blocks of memory, in 4-byte aligned units, and treats them all as potential Pointers. Whenever any potential pointer is found pointing to a piece of memory that you have allocated, the GC considers that piece of memory to be **non-garbage** too. This operation is performed recursively. This is similar

to filtering COVID-19 infected people and their close contacts, and close contacts of close contacts recursively. Finally, blocks of memory that are unrelated to the **non-garbage** memory are treated as garbage and are disposed of free.

Analyzing the above process, we can see that the recursive process is very uncertain in time consumption. If the GC manages a large amount of memory, blocks are small in granularity, and Python programs use many variables with interlocking references, this recursive screening can consume a lot of time, even more than 100,000 CPU cycles! **During garbage collecting, the script is suspended.** Therefore, for most hard real-time control systems, it is not possible to accept this worst-case response delay. In practice, to make MicroPython-based systems capable of more real-time applications, implement real-time tasks in the underlying C language and manage their memory manually.

Even at the Python level, there are ways to minimize the uncertainty of garbage picking. The GC binds some key functions to a Python module called **gc**. It has an important method called **collect()**. When writing a Python script, if you realize that your script will quickly produce and consume large chunks of memory, that is, quickly creating memory garbage -- you can insert some **gc.collect()** calls appropriately to **clean up** more frequently to avoid spending a lot of time cleaning up after the garbage **piles up**. The cost is the efficiency reduction of garbage collection, that is, reducing uncertainty at the expense of overall performance.

11 Get more help

Besides the built-in `help()` commands mentioned above, MicroPython official website has a very complete documentation system, located at [MicroPython documentation](#). It includes comprehensive information about MicroPython and its related resources, such as an introduction to the MicroPython language itself, libraries, and some of the features of the development boards.

12 References

Following documents may offer further reference.

- [MicroPython](#)
- [MicroPython libraries](#)
- [micropython](#)
- [micropython-rocky](#)

13 Revision history

Revision number	Date	Substantive changes
0	27 April, 2021	Initial release

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, ICODE, JCOP, LIFE, VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, Altivec, CodeWarrior, ColdFire, ColdFire+, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, Tower, TurboLink, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, μ Vision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© NXP B.V. 2021.

All rights reserved.

For more information, please visit: <http://www.nxp.com>

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 27 April, 2021

Document identifier: AN13242

