

# 勘智K210-MicroPython & OpenMV 使用指南



提升社会运行效率 改善人类生活方式

# 关于本文档

本文档为用户提供基于勘智K210的MicroPython & OpenMV使用指南

## 发布说明

日期	版本	发布说明
2020-12-20	V1.0	初始版本
2021-01-15	V1.0.1	修正内容
2021-03-15	V1.0.2	修正内容、统一排版格式
2021-03-26	V1.1	修正内容

## 免责声明

本文中的信息, 包括参考的 URL 地址, 如有变更, 恕不另行通知。文档 “按现状” 提供, 不负任何担保责任, 包括对适销性、适用于特定用途或非侵权性的任何担保, 和任何提案、规格或样品在他处提到的任何担保。本文档不负任何责任, 包括使用本文档内信息产生的侵犯任何专利权行为的责任。本文档在此未以禁止反言或其他方式授予任何知识产权使用许可, 不管是明示许可还是暗示许可。文中提到的所有商标名称、商标和注册商标均属其各自所有者的财产, 特此声明。

## 版权公告

版权归 © 2021 嘉楠科技所有。保留所有权利。

## 目录

<b>1. 文档概述.....</b>	<b>1</b>
1.1 适用范围.....	1
1.2 应用场景.....	1
1.3 优势.....	2
1.4 支持范围.....	3
<b>2. 开发环境.....</b>	<b>3</b>
2.1 使用注意.....	3
2.2 架构示意.....	4
2.3 环境搭建.....	4
2.4 SDK 结构.....	5
2.5 配置项说明 (menuconfig ) .....	7
<b>3. MicroPython 标准库.....</b>	<b>12</b>
3.1 uarray.....	13
3.2 cmath.....	14
3.3 gc.....	15
3.4 math.....	16
3.5 sys.....	21
3.6 ubinascii.....	23
3.7 ucollections.....	24
3.8 uerrno.....	25
3.9 uhashlib.....	25
3.10 uheapq.....	27
3.11 uio.....	27
3.12 ujson.....	28
3.13 uos.....	29

---

3.14 ure.....	30
3.15 uselect.....	31
3.16 usocket.....	33
3.17 ussl.....	37
3.18 ustruct.....	37
3.19 utime.....	39
3.20 uzlib.....	42
3.21 _thread.....	42
<b>4. MicroPython 特定库.....</b>	<b>42</b>
4.1 machine.....	42
4.2 machine.Pin.....	43
4.3 machine.UART.....	46
4.4 machine.SPI.....	48
4.5 machine.I2C.....	51
4.6 FPIOA.....	54
4.7 I2S.....	57
4.8 camera.....	59
4.9 FFT.....	60
4.10 KPU.....	61
4.11 machine.Timer.....	64
4.12 machine.PWM.....	66
4.13 machine.RTC.....	67
4.14 machine.LCD.....	69
4.15 MicroPython.....	71
4.16 network.....	73
4.17 rtthread.....	75
<b>5. OpenMV.....</b>	<b>76</b>

---

<i>6. OpenMV IDE</i>	77
<i>7. Rt-Thread MicroPython IDE</i>	84
<i>8. OTA</i>	85

# 1. 文档概述

本文档将介绍K210-MicroPython的特性与优势、可以被应用的领域，以及提供的各类库函数。

用户可以通过本文档提供的各模块Demo示例快速上手，可以在不熟悉太多硬件的背景下进行硬件产品原型开发、产品落地。

## 1.1 适用范围

软件/硬件产品经理、软件/硬件工程师、项目经理、硬件入门学习者、  
Python工程师

## 1.2 应用场景

MicroPython 基于K210完整实现了 Python3 的核心功能，可以在产品开发的各个阶段给开发者带来便利。

通过 MicroPython 提供的库和函数，开发者可以快速控制 LED、液晶、舵机、多种传感器、SD卡、UART、I2C 等，实现各种功能，而不用再去研究底层硬件模块的使用方法，翻看寄存器手册。这样不但降低了开发难度，而且减少了重复开发工作，可以加快开发速度，提高开发效率。以前需要较高水平的嵌入式工程师花费数天甚至数周才能完成的功能，现在普通的嵌入式开发者用几个小时就能实现类似的功能。

随着半导体技术的不断发展，芯片的功能、内部的存储器容量和资源不断增加，成本不断降低，可以使用 MicroPython 来进行开发设计的应用领域也会越来越多。

### ● 产品原型验证

众所周知，在开发新产品时，原型设计是一个非常重要的环节，这个环节需要以最快速的方式设计出产品的大致模型，并验证业务流程或者技术点以及市场验证。与传统开发方法相比，使用 MicroPython 对于原型验证非常有用，让原型验证过程变得轻松，加速原型验证过程。

在进行一些物联网功能开发时，网络功能也是 MicroPython 的长处，可以利用现成的众多 MicroPython 网络模块，节省开发时间。而这些功能如果使用 C/C++ 来完成，会耗费几倍的时间。

- 硬件快速测试

嵌入式产品在开发时，一般会分为硬件开发及软件开发。硬件工程师并不一定都擅长软件开发，所以在测试新硬件时，经常需要软件工程师参与。这就导致软件工程师可能会耗费很多时间帮助硬件工程师查找设计或者焊接问题。有了 MicroPython 后，将 MicroPython 固件烧入待测试的新硬件，在检查焊接、连线等问题时，只需使用简单的 Python 命令即可测试。这样，硬件工程师一人即可搞定，再也不用麻烦别人了。

- 创客DIY

MicroPython 无需复杂的设置，不需要安装特别的软件环境和额外的硬件，使用任何文本编辑器就可以进行编程。大部分硬件功能，使用一个命令就能驱动，不用了解硬件底层就能快速开发。这些特性使得 MicroPython 非常适合创客使用来开发一些有创意的项目。

- 教育

MicroPython 使用简单、方便，非常适合于编程入门。在校学生或者业余爱好者都可以通过 MicroPython 快速的开发一些好玩的项目，在开发的过程中学习编程思想，提高自己的动手能力。

## 1.3 优势

Python 是一款容易上手的脚本语言，同时具有强大的功能，语法优雅简单。使用 MicroPython 编程可以降低嵌入式的开发门槛，让更多的人体验嵌入式的乐趣。

通过 MicroPython 实现硬件底层的访问和控制，不需要了解底层寄存器、数据手册、厂家的库函数等，即可轻松控制硬件。

外设与常用功能都有相应的模块，降低开发难度，使开发和移植变得容易和快速。

## 1.4 支持范围

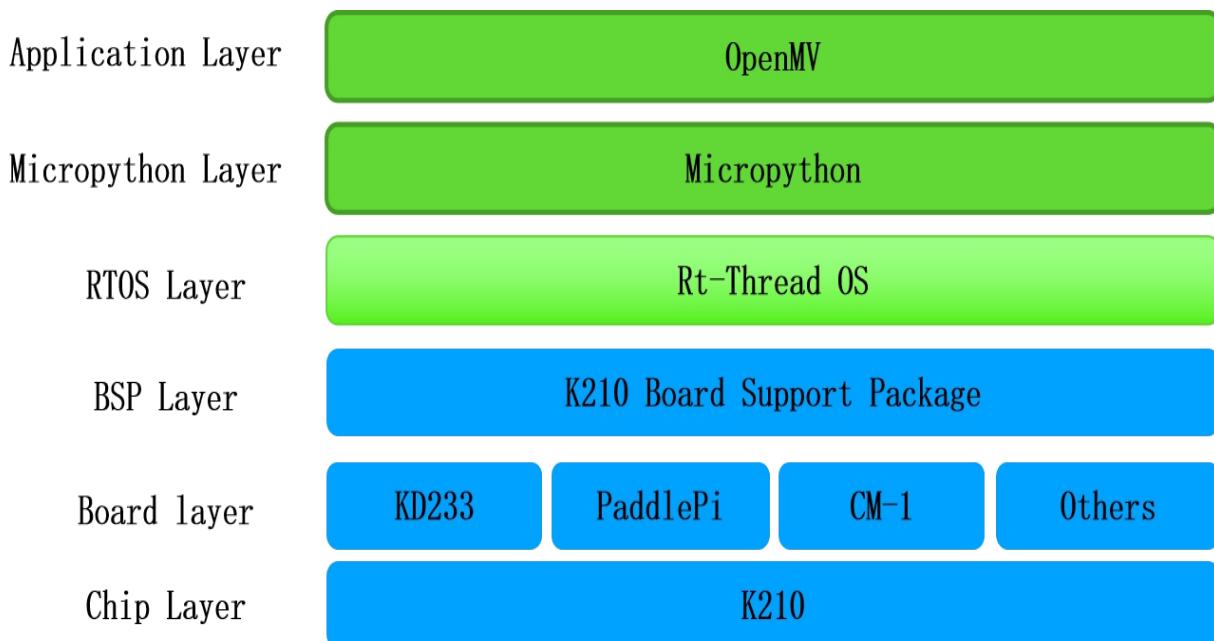
- 实现了 OpenMV 图形算法
- 可用 OpenMV IDE 调试脚本
- 支持 OTA 升级
- 支持 AT(esp8266/w600)、W5500(以太网)、RW007(wifi) 联网
- 支持 SD 卡
- 支持 MicroPython 标准库以及大部分特定库
- 支持 littleVGL 的 mpy 模块
- 支持音频采集/播放

## 2. 开发环境

### 2.1 使用注意

- 支持在 windows/Linux 环境下进行开发环境搭建和 IDE 工具；
- 当没有开启 micropython 或 omv 时，用户可用的 sram 约为 4.5MB；  
当开启 micropython 或 omv 时，用户可用的 sram 约为 2.9MB；
- 操作系统版本：Rt-Thread 4.0.3 【K210-MicroPython 运行在该版本下，与该版本驱动深度集成，不保证与其它 OS 版本的兼容性】；
- 编译器版本 xpack-riscv-none-embed-gcc-8.3.0-1.9；
- 开发工具可选择由 Rt-Thread 提供的 MicroPython IDE，下载地址为：  
[Rt-Thread MicroPython Develop Environment](#)

## 2.2 架构示意



## 2.3 环境搭建

Windows:

- 下载omv-k210(该开发包集成Rt-Thread 4.0.3, 编译器 xpack-riscv-none-embed-gcc-8.3.0-1.9 Windows)  
下载地址: <https://github.com/kendryte/K210-Micropython-OpenMV>
- 安装编译器并将安装路径\xpack-riscv-none-embed-gcc-8.3.0-1.9\bin加入环境变量;
- 安装 Rt-Thread 配置工具 ENV, 它兼具配置与调用编译器的功能。参考: [Rt-Thread 配置工具手册](#)

Ubuntu18.04 (recommend) :

- 下载omv-k210(该开发包集成Rt-Thread 4.0.3, 编译器 xpack-riscv-none-embed-gcc-8.3.0-1.9 Linux)  
下载地址: <https://github.com/kendryte/K210-Micropython-OpenMV>
- 安装编译器, 例如安装到/opt目录下, 则编译器可执行文件位于/opt/xpack-riscv-none-embed-gcc-8.3.0-1.9/bin下

- 参考在Ubuntu平台开发RT-Thread完成环境配置。

## 2.4 SDK 结构

整个 sdk 目录结构：

```
--doc          # 使用说明文档
--src          # 整个 sdk 工程源码
--toolchain    # 编译工具链

src 文件夹下目录结构：
--bsp          # 板级支持包
  --driver      # 外设驱动
  --kendryte-sdk # 芯片厂商支持包
--extmods      # MicroPython 用户扩展模块
  --lvgl        # |lvg| 的 MicroPython 绑定
  --k210        # k210 片上或板载外设的 mpy 模块
  --cameras     # 各种摄像头驱动
  --moddefsmisc.h # 模块声明
  --qstrdefsmisc.h # 字符串声明

--OpenMV
  --omv/boards   # 板子相关的功能配置
  --omv/cameras  # 各种摄像头驱动
  --omv/hal       # 文件系统、外设等底层接口
  --omv/img       # 图像算法
  --omv/py        # 应用接口的 mpy 模块
  --omv/rt_thread_dbg.c # 和 OpenMV IDE 之间的通信线程

--projects      # 用户项目代码，当用户有多个项目时，可统一放在该目录下
  --boot_k210    # bootloader 参考代码
  --app_k210     # 应用项目参考代码

--rtthread      # rtthread 内核源码
--test          # 部分 MicroPython 及 c 测试参考代码

工程项目 boot_k210 文件夹下目录结构：
--boot_k210
  --applications # 应用层代码
  --packages      # 在线软件包，此目录由 rtthread ENV 工具自动生成
```

```
--rtconfig.h          #功能配置文件，由配置工具产生  
--rtconfig.py         #工具链配置脚本
```

工程项目app\_k210文件夹下目录结构：

```
--app_k210  
  --applications      #应用层代码  
  --packages          #在线软件包，此目录由rtthread ENV工具自动生成  
  --rtconfig.h          #功能配置文件，由配置工具产生  
  --rtconfig.py         #工具链配置脚本  
  --moddefs.user.extmods.h #向基于rtthread 移植的MicroPython-v1.13 注册用户模块  
  --qstrdefs.user.extmods.h # 注册字符串
```

## 2.5 配置项说明 (menuconfig )

注：当某个配置项被关闭且保存后，再次开启会恢复默认，需重新设置。

- 开启 AT 模块 (w600)

```
Rt-Thread online packages --->
  IoT - internet of things --->
    [*] AT DEVICE: Rt-Thread AT component porting ... --->
      [*]   WinnerMicro W60X --->
        [ ]   Enable initialize by thread (NEW)
        [*]   Enable sample (NEW)
        (rtthread) WIFI ssid (NEW)           #修改为合适的
        (12345678) WIFI password (NEW)
        (uart1) AT client device name (NEW)  #根据连接设置
        (512)   The maximum length of receive line buffer (NEW)
```

```
Rt-Thread online packages --->
  IoT - internet of things --->
    [*] WIZnet: WIZnet TCP/IP chips SAL framework implement --->
      WIZnet: WIZnet TCP/IP chips SAL framework implement
      WIZnet device type (W5500) --->
      WIZnet device configure --->
        (spi30) SPI device name (NEW)          #需根据实际连线设置
        (10)  Reset PIN number (NEW)          #需根据实际连线设置
        (11)  IRQ PIN number (NEW)            #需根据实际连线设置
    [*]   Enable alloc IP address through DHCP (NEW)
```

- 开启 W5500

```
[*]   Enable Ping utility (NEW)
[ ]   Enable debug log output (NEW)
Version (latest) --->
```

- 开启 RW007

第一步：

```
Hardware Drivers --->
[*] Enable RW007 wifi module
    (spi11) the SPIDEV rw007 driver on (NEW) #spi11意为spi1总线的片选1
    (8) reset PIN (NEW) #根据连接修改
    (7) INT and BUSY status pin (NEW) #根据连接修改
```

第二步：

```
Rt-Thread online packages --->
IoT - internet of things --->
Wi-Fi -->
    -*- rw007: SPI WIFI rw007 driver --->
        version (latest) --->
        example driver port (not use example driver, porting by myself) --->
            (30000000) SPI Max Hz (NEW)
```

- 开启 MicroPython

```
Rt-Thread online packages --->
language packages --->
[*] MicroPython: ... -->
...
User Extended Module ---> #省略其他项
[*] modules define in your project #如无自定义模块可关闭此项

(1500000) Heap size for python run environment #根据需要修改
[ ] Enable micropython to use float instead of double
    # 不要使能该项
    Version (latest) --->
```

- 开启 OpenMV

```
OpenMV --->
[*] Enable OpenMV(Cross-Platform)
Board type (Kendryte KD233) ---> [*]
Hal K210
(230400) Total Framebuffer Size #320*240*3, 最大支持的图像大小, 注意需为尺寸
*3

[ ] Enable LAB LUT           #LAB算法的静态数据表, 关闭优化体积
[ ] Enable YUV LUT
```

- 开启外设驱动

注：选中某项带“---”的配置项后，需进入子菜单作详细配置

```
Hardware Drivers --->
    IO Groups Power Supply Settings ---> #IO BANK电压配置
[*] Enable High Speed UART
    General Purpose UARTs --->
[*] Enable I2C0
    (35) I2C0 SCL pin number      #指定引脚, 编号为芯片引脚名为IOxx后的数字
    (34) I2C0 SDA pin number
[ ] Enable I2C1
[ ] Enable I2C2
[*] Enable SPI1 --->
[*] Enable LCD on SPI0 --->
[ ] Enable I2S0(Play Only)
[ ] Enable I2S1(Record Only)
[ ] Enable I2S2

[ ] Enable PWM ----
[ ] Enable Timer0 ----
[ ] Enable Timer1 ----

[ ] Enable DVP Camera          #配置DVP引脚
[ ] Enable bridge module        #配置和OpenMV IDE之间的通信转换模块
[ ] Enable RW007 wifi module
```

- 开启 littleVGL

注： 在最新版中屏幕尺寸从驱动获取可以不用配置

```
Rt-Thread online packages --->
system packages --->
[*] LittlevGL2RTT: The LittlevGl gui ... --->
    version (latest) ---> #需最新版
    LittlevGL2RTT Options --->
        Color depth (32bit) ---> #当用lvgl mpy bindings时选32bit色
        Garbage Collector (enable GC) ---> #用lvgl mpy bindings时选择
    [ ] LittlevGL2RTT demo example #用lvgl mpy bindings时关闭示例
```

- 开启 littleVGL MPY Bindings

注： 当开启此项功能前如未配置过littleVGL 需对其进行配置

```
extmods --->
[*] Enable MPY extmods
[*] Enable lvgl mpy bindings (NEW) ----
```

- 开启 K210 的特定 `mpy` 扩展模块

注： 其中DVP模块默认关闭

```
extmods --->
[*] Enable K210 extmods (NEW) --->
```

- 配置网络功能

注： 当选择W5500或RW007等网络模块后会默认开启网络相关功能， 但还需少量配置

```
Rt-Thread Components --->
Network --->
  Socket abstraction layer --->
    [*] Enable socket abstraction layer
    [*] Enable BSD socket operated by file ... #要支持select必选
    light weight TCP/IP stack ---> #不用RW007时不用配置此项
      (4096) the stack size of lwIP thread
    ... #其他项略
```

- 开启 SD 卡

第一步： 配置相应SPI驱动

第二步：

```
Rt-Thread Components --->
Device Drivers --->
  -*- Using SPI Bus/Device device drivers
  [*] Using SD/TF card driver with spi
```

第三步：

```
Rt-Thread Components --->
Device virtual file system --->
  [*] Enable elm-chan fatfs
```

- 开启 OTA 下载

```
Rt-Thread online packages --->
IoT - internet of things --->
  [*] ota_downloader: The firmware downloader ... --->
    [*] Enable HTTP/HTTPS OTA
      (http://xxx/xxx/rtthread.rbl) HTTP OTA Download ... (NEW) # 按需修改
    [*] Enable Ymodem OTA
  Version (latest) --->
```

# 3. MicroPython 标准库

## ● 内建函数

• abs()	• hash()	• range()
• all()	• hex()	• repr()
• any()	• id()	• reversed()
• bin()	• input()	• round()
• class bool	• class int	• class set
• class bytearray	• classmethod from_bytes(bytes, byteorder) In MicroPython, byteorder parameter must be positional (this is compatible with CPython).	• setattr()
• class bytes	• to_bytes(size, byteorder) In MicroPython, byteorder parameter must be positional (this is compatible with CPython).	• class slice The slice builtin is the type that slice objects have.
• callable()	• isinstance()	• sorted()
• chr()	• issubclass()	• staticmethod()
• classmethod()	• iter()	• class str
• compile()	• len()	• sum()
• class complex	• class list	• super()
• delattr(obj, name)	• locals()	• class tuple
• class dict	• map()	• type()
• dir()	• max()	• zip()
• divmod()	• class memoryview	
• enumerate()	• min()	
• eval()	• next()	
• exec()	• class object	
• filter()	• oct()	
• class float	• open()	
• class frozenset	• ord()	
• getattr()	• pow()	
• globals()	• print()	
• hasattr()	• property()	

## ● Exceptions

• exception AssertionError	• exception OSError <i>See CPython documentation: OSError. MicroPython doesn't implement errno attribute, instead use the standard way to access exception arguments: exc.args[0].*</i>
• exception AttributeError	• exception RuntimeError
• exception Exception	• exception StopIteration
• exception ImportError	• exception SyntaxError
• exception IndexError	• exception SystemExit <i>See CPython documentation: SystemExit.</i>
• exception KeyboardInterrupt	• exception TypeError <i>See CPython documentation: TypeError.</i>
• exception KeyError	• exception ValueError
• exception MemoryError	• exception ZeroDivisionError
• exception NameError	
• exception NotImplementedError	

## 3.1 uarray

uarray模块定义了一个对象类型，它可以简洁地表示基本值的数组、字符、整数、浮点数。支持代码格式：b, B, h, H, i, I, l, L, q, Q, f, d（最后 2 个需要支持浮点数）。

### ● 构造方法

```
class uarray.array(typecode [, iterable])
```

描述：用给定类型的元素创建数组。数组的初始内容由 `iterable` 提供，如果没有提供，则创建一个空数组。

### ● 入参说明

`typecode`: 数组的类型

`iterable`: 数组初始内容

### ● Demo示例

```
>>> import uarray
>>> a = uarray.array('i', [2, 4, 1, 5])
>>> b = uarray.array('f')
>>> print(a)
array('i', [2, 4, 1, 5])
>>> print(b)
array('f')
```

## ● 方法

`uarray.append(val)`

将一个新元素追加到数组的末尾。

示例：

```
>>> a = uarray.array('f', [3, 6])
>>> print(a)
uarray('f', [3.0,
6.0])
>>> a.append(7.0)
>>> print(a)
uarray('f', [3.0, 6.0, 7.0])
```

`uarray.extend(iterable)`

将一个新的数组追加到数组的末尾，注意追加的数组和原来数组的数据类型要保持一致。

示例：

```
>>> a = uarray.array('i', [1, 2, 3])
>>> b = uarray.array('i', [4, 5])
>>> a.extend(b)
>>> print(a)
uarray('i', [1, 2, 3, 4, 5])
```

更多内容可参考[uarray](#)。

## 3.2 cmath

`cmath`模块提供了一些用于复数运算的方法。这个模块中的方法接受整数、浮点数或复数作为参数，同时还将接受任何有复数或浮点方法的 Python 对象：这些方法分别用于将对象转换成复数或浮点数，然后将该方法应用到转换的结果中。

## ● 方法

`cmath.cos(z)`

返回z的余弦

`cmath.exp(z)`

返回z的指数

`cmath.log(z)`

返回z的对数

`cmath.log10(z)`

返回z的常用对数

`cmath.phase(z)`

返回z的相位，范围是 $(-\pi, +\pi]$ ，以弧度表示

`cmath.polar(z)`

返回z的极坐标

`cmath.rect(r, phi)`

返回模量r和相位phi的复数

`cmath.sin(z)`

返回z的正弦

`cmath.sqrt(z)`

返回z的平方根

### ● 常数

`cmath.e`

自然对数的指数

`cmath.pi`

圆周率。

更多内容可参考[cmath](#)

## 3.3 gc

gc模块提供了垃圾收集器的控制接口

### ● 方法

`gc.enable()`

允许自动回收内存碎片

`gc.disable()`

禁止自动回收，但可以通过 `collect()` 方法进行手动回收内存碎片

`gc.collect()`

运行一次垃圾回收

`gc.mem_alloc()`

返回已分配的内存数量

`gc.mem_free()`

返回剩余的内存数量

更多内容可参考[gc](#)

## 3.4 math

### ● 方法

`math.acos(x)`

传入弧度值，计算 $\cos(x)$ 的反三角方法。

`math.acosh(x)`

返回x的逆双曲余弦

`math.asin(x)`

传入弧度值，计算 $\sin(x)$ 的反三角方法

示例：

```
>>> x = math.asin(0.5)
>>> print(
x)
0.5235988
```

`math.asinh(x)`

返回x的逆双曲正弦

`math.atan(x)`

返回x的逆切线

`math.atan2(y, x)`

返回 $y/x$ 的反正切函数值

`math.atanh(x)`

返回x的反双曲正切

`math.ceil(x)`

向上取整。

示例：

```
>>> x = math.ceil(5.6454)
>>> print(x)
6
```

`math.copysign(x, y)`

返回带有y符号的x

`math.cos(x)`

传入弧度值，计算x的余弦。

示例：计算  $\cos 60^\circ$

```
>>> math.cos(math.radians(60))
0.5
```

math.cosh(x)

返回双曲cosx

math.degrees(x)

弧度转化为角度

示例：

```
>>> x = math.degrees(1.047198)
>>> print(x)
60.00002
```

math.erf(x)

返回x的误差函数

math.erfc(x)

返回x的互补误差函数

math.exp(x)

计算 e 的x次方（幂）

示例：

```
>>> x = math.exp(2)
>>> print(x)
7.389056
```

math.expm1(x)

计算 math.exp(x) - 1

math.fabs(x)

计算x的绝对值

示例：

```
>>> x = math.fabs(-5)
>>> print(x)
5.0
>>> y = math.fabs(5.0)
>>> print(y)
5.0
```

math.floor(x)

对x向下取整

示例：

```
>>> x = math.floor(2.99)
>>> print(x)
2
>>> y = math.floor(-2.34)
>>> print(y)
-3
```

math.fmod(x, y)

取x除以y的模

示例：

```
>>> x = math.fmod(4, 5)
>>> print(x)
4.0
```

math.frexp(x)

将一个浮点数分解为它的尾数和指数。返回值是元组(m, e)，使得 $x == m * 2**e$  完全正确。如果 $x == 0$ ，则函数返回(0.0, 0)，否则返回关系  $0.5 \leq \text{abs}(m) < 1$ 保持不变。

math.gamma(x)

返回伽马方法。示例：

```
>>> x = math.gamma(5.21)
>>> print(x)
33.08715
```

math.isfinite(x)

如果x是有限的，返回True

math.isinf(x)

如果x是无限的，则返回True

math.isnan(x)

如果x不是a-number，则返回True

math.ldexp(x, exp)

返回  $x * (2**exp)$

math.lgamma(x)

返回伽马方法的自然对数。示例：

```
>>> x = math.lgamma(5.21)
>>> print(x)
3.499145
```

`math.log(x)`

计算以 e 为底的x的对数。示例：

```
>>> x = math.log(10)
>>> print(x)
2.302585
```

`math.log10(x)`

计算以 10 为底的x的对数。示例：

```
>>> x = math.log10(10)
>>> print(x)
1.0
```

`math.log2(x)`

计算以 2 为底的x的对数。示例：

```
>>> x = math.log2(8)
>>> print(x)
3.0
```

`math.modf(x)`

返回一个由两个浮点数组成的元组，是x的小数部分和整数部分。两个返回值都有与x相同的符号。

`math.pow(x, y)`

计算 x 的 y 次方（幂）。示例：

```
>>> x = math.pow(2, 3)
>>> print(x)
8.0
```

`math.radians(x)`

角度转化为弧度。示例：

```
>>> x = math.radians(60)
>>> print(x)
1.047198
```

`math.sin(x)`

传入弧度值，计算正弦。示例：计算  $\sin 90^\circ$

```
>>> math.sin(math.radians(90))
1.0
```

`math.sinh(x)`

返回x的双曲正弦值

`math.sqrt(x)`

计算平方根。示例：

```
>>> x = math.sqrt(9)
>>> print(x)
3.0
```

`math.tan(x)`

传入弧度值，计算正切。示例：计算  $\tan 60^\circ$

```
>>> math.tan(math.radians(60))
1.732051
```

`math.tanh(x)`

返回双曲正切x

`math.trunc(x)`

取整。示例：

```
>>> x = math.trunc(5.12)
>>> print(x)
5
>>> y = math.trunc(-6.8)
>>> print(y)
-6
```

## ● 常数

`math.e`

自然对数的底数。示例：

```
>>>import math  
>>>print(math.e)  
2.718282
```

`math.pi` 圆周率。示例：

```
>>> print(math.pi)  
3.141593
```

更多内容可参考[math](#)。

## 3.5 sys

`sys`模块提供系统特有的功能。

### ● 方法

`sys.exit(retval=0)`

终止当前程序给定的退出代码。方法会抛出 `SystemExit` 异常。

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

打印异常与追踪到一个类似文件的对象 `file` (或者缺省 `sys.stdout`)

提示：这是 CPython 中回溯模块的简化版本。不同于 `traceback.print_exception()`，这个方法用异常值代替了异常类型、异常参数和回溯对象。文件参数在对应位置，不支持更多参数。CPython 兼容回溯模块在 `MicroPython-lib`。

## ● 常数

`sys.argv`

当前程序启动时参数的可变列表

`sys.byteorder`

系统字节顺序 (“little” or “big”).

`sys.implementation`

关于当前 Python 实现的信息，对于 MicroPython 来说，有以下属性：

1) 名称 - ‘’MicroPython‘’

2) 版本 - 元组 (主要, 次要, 小)，比如 (1, 9, 3)

`sys.modules`

已加载模块的字典。在一部分移植中，它可能不包含内置模块

`sys.path`

用来搜索导入模块地址的列表

`sys.platform`

返回当前平台的信息。

`sys.stderr`

标准错误流

`sys.stdin`

标准输入流

`sys.stdout`

标准输出流

`sys.version`

符合的 Python 语言版本，如字符串。

`sys.version_info`

本次实现使用的 Python 语言版本，用一个元组的方式表示。示例：

```
>>> import sys
>>> sys.version
'3.4.0'
>>> sys.version_info
```

```
(3, 4, 0)
>>> sys.path
['', '/libs/mpy/']
>>> sys.__name__
'sys'
>>> sys.platform
'Rt-Thread'
>>> sys.byteorder
'little'
```

更多内容可参考[sys](#)。

## 3.6 ubinascii

ubinascii模块包含许多在二进制和各种 ascii 编码的二进制表示之间转换的方法。

### ● 方法

```
ubinascii.hexlify(data[, sep])
```

将字符串转换为十六进制表示的字符串

示例：

```
>>> ubinascii.hexlify('hello Rt-Thread')
b'68656c6c6f2052542d546872656164'
>>> ubinascii.hexlify('summer')
b'73756d6d6572'
```

如果指定了第二个参数sep，它将用于分隔两个十六进制数。示例：

```
>>> ubinascii.hexlify('hello Rt-Thread', " ")
b'68 65 6c 6c 6f 20 52 54 2d 54 68 72 65 61 64'
>>> ubinascii.hexlify('hello Rt-Thread', ", ")
b'68,65,6c,6c,6f,20,52,54,2d,54,68,72,65,61,64'
```

```
ubinascii.unhexlify(data)
```

转换十六进制字符串为二进制字符串，功能和hexlify相反。示例：

```
>>> ubinascii.unhexlify('73756d6d6572')
b'summer'
```

```
ubinascii.a2b_base64(data)
```

Base64 编码的数据转换为二进制表示，返回字节串。

```
ubinascii.b2a_base64(data)
```

编码base64格式的二进制数据，返回字符串。

更多内容可参考[ubinascii](#)。

## 3.7 ucollections

`ucollections`模块实现了专门的容器数据类型，它提供了 Python 的通用内置容器的替代方案，包括了字典、列表、集合和元组。

### ● 方法

```
ucollections.namedtuple(name, fields)
```

这是工厂方法创建一个新的 `namedtuple` 型与一个特定的字段名称和集合。`namedtuple` 是元组允许子类要访问它的字段不仅是数字索引，而且还具有属性使用符号字段名访问语法。字段是字符串序列指定字段名称。为了兼容的实现也可以用空间分隔的字符串命名的字段（但效率较低）。

代码示例：

```
from ucollections import namedtuple

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

```
ucollections.OrderedDict(...)
```

字典类型的子类，会记住并保留键/值的追加顺序。当有序的字典被迭代输出时，键/值会按照他们被添加的顺序返回：

```
from ucollections import OrderedDict

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

输出：

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

更多的内容可参考[ucollections](#)。

## 3.8 uerrno

`uerrno`模块提供了标准的 `errno` 系统符号，每个符号都有对应的整数值。示例：

```
try:  
    uos.mkdir("my_dir")  
except OSError as exc:  
    if exc.args[0] == uerrno.EEXIST:  
        print("Directory already  
exists")
```

`uerrno.errorcode`

将数字错误码映射为带有符号错误码的字符串的字典(见上文)：

```
>>>print(uerrno.errorcode[uerrno.E  
EXIST])  
EEXIST
```

更多内容可参考[uerrno](#)。

## 3.9 uhashlib

`uhashlib`模块实现了二进制数据哈希算法。

### ● 算法功能

`SHA256`

当代的散列算法（`SHA2` 系列），它适用于密码安全的目的。被包含在 `MicroPython` 内核中，除非有特定的代码大小限制，否则推荐任何开发板都支持这个功能。

`SHA1`

上一代的算法，不推荐新的应用使用这种算法，但是 `SHA1` 算法是互联网标准和现有应用程序的一部分，所以针对网络连接便利的开发板会提供这种功能。

`MD5`

一种遗留下来的算法，作为密码使用被认为是不安全的。只有特定的开发板，为了兼容老的应用才会提供这种算法。

### ● 构造方法

```
class uhashlib.sha256([data])
```

创建一个 `SHA256` 哈希对象并提供 `data` 赋值

```
class uhashlib.sha1([data])
```

创建一个 SHA1 哈希对象并提供 data 赋值

```
class uhashlib.md5([data])
```

创建一个 MD5 哈希对象并提供 data 赋值

## ● 方法

```
uhash.update(data)
```

将更多二进制数据放入哈希表中

```
uhash.digest()
```

返回字节对象哈希的所有数据。调用此方法后，将无法将更多数据送入哈希

```
uhash.hexdigest()
```

此方法没有实现，使用`ubinascii.hexlify(hash.digest())` 达到类似效果

更多内容可参考[uhashlib](#)。

## 3.10 uheapq

uheapq模块提供了堆排序相关算法，堆队列是一个列表，它的元素以特定的方式存储。

### ● 方法

`uheapq.heappush(heap, item)`

将对象压入堆中

`uheapq.heappop(heap)`

从`heap`弹出第一个元素并返回。如果是堆时空的会抛出 `IndexError`。

`uheapq.heapify(x)`

将列表`x`转换成堆。

更多内容可参考[uheapq](#)。

## 3.11 uio

uio模块包含流类型（类似文件）对象和帮助方法。

### ● 方法

`uio.open(name, mode='r', kwargs)`

打开一个文件，关联到内建方法 `open()`。所有端口（用于访问文件系统）需要支持模式参数，但支持其他参数不同的端口。

### ● 类

`class uio.FileIO(...)`

这个文件类型用二进制方式打开文件，等于使用`open(name, "rb")`。不应直接使用这个实例。

`class uio.TextIOWrapper(...)`

这个类型以文本方式打开文件，等同于使用`open(name, "rt")`。不应直接使用这个实例。

`class uio.StringIO([string])`

`class uio.BytesIO([string])`

内存文件对象。`StringIO`用于文本模式 I/O（用“t”打开文件），`BytesIO`用于二进制方式（用“b”方式）。文件对象的初始内容可以用字符串参数指定（`StringIO`用普通字符串，`BytesIO`用 bytes 对象）。所有的文件方法，如 `read()`, `write()`, `seek()`, `flush()`, `close()` 都可以用在这些对象上，包括下面方法：

`getvalue()`

获取缓存区内容。

更多内容可参考[uio](#)。

## 3.12 ujson

ujson 模块提供 Python 对象到 JSON (JavaScript Object Notation) 数据格式的转换。

### ● 方法

`ujson.dumps(obj)`

将 dict 类型转换成 str。obj: 要转换的对象

示例：

```
>>> obj = {1:2, 3:4, "a":6}
>>> print(type(obj), obj) #原来为dict类型
<class 'dict'> {3: 4, 1: 2, 'a': 6}
>>> jsObj = ujson.dumps(obj) #将dict类型转换成str
>>> print(type(jsObj), jsObj)
<class 'str'> {3: 4, 1: 2, "a": 6}
```

`ujson.loads(str)`

解析 JSON 字符串并返回对象。如果字符串格式错误将引发 `ValueError` 异常。

示例：

```
>>> obj = {1:2, 3:4, "a":6}
>>> jsDumps = ujson.dumps(obj)
>>> jsLoads = ujson.loads(jsDumps)
>>> print(type(obj), obj)
<class 'dict'> {3: 4, 1: 2, 'a': 6}
```

更多内容可参考[ujson](#)

## 3.13 uos

uos 模块包含了对文件系统的访问操作，是对应 CPython 模块的一个子集。

### ● 方法

`uos.chdir(path)`

更改当前目录

`uos.getcwd()`

获取当前目录。

`uos.listdir([dir])`

没有参数就列出当前目录，否则列出给定目录。

`uos.mkdir(path)`

创建一个目录。

`uos.remove(path)`

删除文件。

`uos.rmdir(path)`

删除目录。

`uos.rename(old_path, new_path)`

重命名文件或者文件夹。

`uos.stat(path)`

获取文件或目录的状态。

`uos.sync()`

同步所有的文件系统。

示例：

```
>>> import uos
>>> uos.                                     # Tab
__name__          uname        chdir        getcwd
listdir         mkdir        remove       rmdir
stat            unlink       mount        umount
>>> uos.mkdir("rtthread")
>>> uos.getcwd()
'/
>>> uos.chdir("rtthread")
>>> uos.getcwd()
'/rtthread'
>>> uos.listdir()
['web_root', 'rtthread', '11']
>>> uos.rmdir("11")
>>> uos.listdir()
['web_root', 'rtthread']
>>>
```

更多内容可参考 [uos](#)

## 3.14 ure

ure 模块用于测试字符串的某个模式，执行正则表达式操作。

匹配字符集 匹配任意字符

'.'

匹配字符集合，支持单个字符和一个范围

'[]'

支持多种匹配元字符

'^' '\$' '?' '\*' '+' '?' '\*' '+?' '{m, n}'

### ● 方法

`ure.compile(regex)`

编译正则表达式，返回 `regex` 对象。

`ure.match(regex, string)`

用 `string` 匹配 `regex`，匹配总是从字符串的开始匹配。

`ure.search(regex, string)`

在 `string` 中搜索 `regex`。不同于匹配，它搜索第一个匹配位置的正则表达式字符串（结果可能会是 0）。

`ure.DEBUG`

标志值，显示表达式的调试信息。

### 正则表达式对象：

编译正则表达式，使用 `ure.compile()` 创建实例。

`regex.match(string)`

`regex.search(string)`

`regex.split(string, max_split=-1)`

### 匹配对象：

匹配对象是 `match()` 和 `search()` 方法的返回值。

`match.group([index])`

只支持数字组

更多内容可参考 [ure](#)

## 3.15 uselect

`uselect` 模块提供了等待数据流的事件功能。

### ● 常数

`uselect.POLLIN`

读取可用数据

`uselect.POLLOUT`

写入更多数据

`uselect.POLLERR`

发生错误

`uselect.POLLHUP`

流结束/连接终止检测

### ● 方法

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

监控对象何时可读或可写，一旦监控的对象状态改变，返回结果（阻塞线程）。这个方法是为了兼容，效率不高，推荐用 `poll` 方法。

-- `rlist`: 等待读就绪的文件描述符数组

-- `wlist`: 等待写就绪的文件描述符数组

-- `xlist`: 等待异常的数组

-- `timeout`: 等待时间（单位：秒）

示例：

```
def selectTest():
    global s
    rs, ws, es = uselect.select([s], [], [])
    #程序会在此等待直到对象s可读
    print(rs)
    for i in rs:
        if i == s:
            print("s can read now")
            data,addr=s.recvfrom(1024)
            print('received:',data,'from',addr)
```

## ● Poll类

`uselect.poll()`

创建 poll 实例。示例：

```
>>>poller = uselect.poll()
>>>print(poller)
<poll>
```

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

注册一个用以监控的对象，并设置被监控对象的监控标志位 `eventmask`。

-- `obj`: 被监控的对象

-- `eventmask`: 被监控的标志

✓ `uselect.POLLIN` — 可读

✓ `uselect.POLLHUP` — 已挂断

✓ `uselect.POLLERR` — 出错

✓ `uselect.POLLOUT` — 可写

`poll.unregister(obj)`

解除监控的对象的注册。

-- `obj`: 注册过的对象示例：

```
>>>READ_ONLY = uselect.POLLIN | uselect.POLLHUP | uselect.POLLERR
>>>READ_WRITE = uselect.POLLOUT | READ_ONLY
>>>poller.register(s, READ_WRITE)
>>>poller.unregister(s)
```

`poll.modify(obj, eventmask)`

修改已注册的对象监控标志。

-- `obj`: 已注册的被监控对象

-- `eventmask`: 修改为的监控标志示例：

```
>>>READ_ONLY = uselect.POLLIN | uselect.POLLHUP | uselect.POLLERR
>>>READ_WRITE = uselect.POLLOUT | READ_ONLY
>>>poller.register(s, READ_WRITE)
>>>poller.modify(s, READ_ONLY)
```

`poll.poll([timeout])`

等待至少一个已注册的对象准备就绪。返回 `(obj, event, ...)` 元组，`event` 元素指定了一个流发生的事件，是上面所描述的 `uselect.POLL*` 常量组合。根据平台和版本的不同，在元组中可能有其他元素，所以不要假定元组的大小是 2。如果超时，则返回空列表。

更多内容可参考[uselect](#)。

## 3.16 usocket

`usocket` 模块提供对 BSD 套接字接口的访问。

### ● 常数

地址族

- ✓ `usocket.AF_INET` =2 — TCP/IP - IPv4
- ✓ `usocket.AF_INET6` =10 — TCP/IP - IPv6

套接字类型

- ✓ `usocket.SOCK_STREAM` =1 — TCP 流
- ✓ `usocket.SOCK_DGRAM` =2 — UDP 数据报
- ✓ `usocket.SOCK_RAW` =3 — 原始套接字
- ✓ `usocket.SO_REUSEADDR` =4 — socket 可重用

IP 协议号

- ✓ `usocket.IPPROTO_TCP` =16
- ✓ `usocket.IPPROTO_UDP` =17

套接字选项级别

- ✓ `usocket.SOL_SOCKET` =4095

## ● 方法

`usocket.socket`

```
usocket.socket(usocket.AF_INET, usocket.SOCK_STREAM, usocket.IPPROTO_TCP)
```

创建新的套接字，使用指定的地址、类型和协议号。

`usocket.getaddrinfo(host, port)`

将主机域名（host）和端口（port）转换为用于创建套接字的 5 元组序列。元组列表的结构如下：

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

示例：

```
>>> info = usocket.getaddrinfo("Rt-Thread.org", 10000)
>>> print(info)
[(2, 1, 0, '', ('118.31.15.152', 10000))]
```

`usocket.close()`

关闭套接字。一旦关闭后，套接字所有的功能都将失效。远端将接收不到任何数据（清理队列数据后）。虽然在垃圾回收时套接字会自动关闭，但还是推荐在必要时用 `close()` 去关闭。

`usocket.bind(address)`

将套接字绑定到地址，套接字不能是已经绑定的。

`usocket.listen([backlog])`

监听套接字，使服务器能够接收连接。

--`backlog`: 接受套接字的最大个数，至少为 0，如果没有指定，则默认一个合理值。

`usocket.accept()`

接收连接请求。注意：只能在绑定地址端口号和监听后调用，返回 `conn` 和 `address`。

--`conn`: 新的套接字对象，可以用来收发消息

--`address`: 连接到服务器的客户端地址

`usocket.connect(address)`

连接服务器

--`address`: 服务器地址和端口号的元组或列表

`usocket.send(bytes)`

发送数据，并返回成功发送的字节数，返回字节数可能比发送的数据长度少。

--`bytes:bytes` 类型数据

`usocket.recv(bufsize)`

接收数据，返回接收到的数据对象。

-- `bufsize`: 指定一次接收的最大数据量

示例：

```
data = conn.recv(1024)
```

`usocket.sendto(bytes, address)`

发送数据，目标由 `address` 决定，常用于 UDP 通信，返回发送的数据大小。

-- `bytes:bytes` 类型数据

-- `address`: 目标地址和端口号的元组

示例：

```
data = sendto("hello Rt-Thread", ("192.168.10.110", 100))
```

`usocket.recvfrom(bufsize)`

接收数据，常用于 UDP 通信，并返回接收到的数据对象和对象的地址。

-- `bufsize`: 指定一次接收的最大数据量

示例：

```
data,addr=fd.recvfrom(1024)
```

`usocket.setsockopt(level, optname, value)`

根据选项值设置套接字。

-- `level`: 套接字选项级别

-- `optname`: 套接字的选项

-- `value`: 可以是一个整数，也可以是一个表示缓冲区的 `bytes` 类对象。

示例：

```
s.setsockopt(usocket.SOL_SOCKET, usocket.SO_REUSEADDR, 1)
```

`usocket.settimeout(value)`

设置超时时间，单位：秒。

示例：

```
s.settimeout(2)  
usocket.setblocking(flag)
```

设置阻塞或非阻塞模式：如果 `flag` 是 `false`，设置非阻塞模式。

`usocket.read([size])`

从套接字中读取大小为 `size` 的字节，返回一个 `bytes` 对象。如果没有给出 `size`，则读取从套接字到 `EOF` 的所有可用数据；因此，在套接字关闭之前，该方法不会返回。这个函数试图读取所请求的尽可能多的数据（没有“短读”）。但是，对于非阻塞套接字，这是不可能的，这样返回的数据就会更少。

`usocket.readinto(buf[, nbytes])`

将字节读入 `buf`。如果指定了 `nbytes`，则最多读取这个字节数。否则，最多读取 `len(buf)` 字节。与 `read()` 一样，这个方法遵循“no short reads”策略。返回值：读取并存储到 `buf` 中的字节数。

`usocket.readline()`

接收一行数据，遇换行符结束，并返回接收数据的对象。

`usocket.write(buf)`

将字节类型数据写入套接字，并返回写入成功的数据大小。

示例：

#### Example for TCP Server

```
>>> import usocket
>>> s = usocket.socket(usocket.AF_INET,usocket.SOCK_STREAM)      # Create STREAM TCP
    socket
>>> s.bind(('192.168.12.32', 6001))                            # Bind to port 6001
>>> s.listen(5)
>>> s.setblocking(True)
>>> sock,addr=s.accept()
>>> sock.recv(10)
b'Rt-Thread\r'
>>> s.close()
TCP Client example
>>> import usocket
>>> s = usocket.socket(usocket.AF_INET,usocket.SOCK_STREAM)
>>> s.connect(("192.168.10.110",6000))
>>> s.send("MicroPython")
11
>>> s.close()
```

#### Example for Connecting to a web server

```
s = usocket.socket()
s.connect(usocket.getaddrinfo('www.MicroPython.org', 80)[0][-1])
```

更多的内容可参考 [usocket](#)。

## 3.17 ussl

这个模块实现了对应CPython模块的一个子集，如下所述。有关更多信息，请参阅原始CPython文档:[ssl](#)。

此模块提供对传输层安全性(以前被广泛称为“安全套接字层”)的访问，以及对网络套接字(客户端和服务器端)的对等身份验证功能。

### ● 方法

```
ussl.wrap_socket
ussl.wrap_socket(sock, server_side=False, key=None, cert=None)
```

建立一个流(通常是SOCK\_STREAM类型的usocket.socket实例)，并返回将底层流包装在SSL上下文中的ssl.SSLSocket实例。返回对象包含常见的流接口方法，如read()、write()等。在MicroPython中，返回的对象不会公开recv()、send()等套接字接口和方法，特别是，服务器端SSL套接字应该从非SSL倾听服务器套接字上的accept()返回的普通套接字创建，部分或全部关键字不能支持，因为它是根据特定MicroPython端口实现的底层模块。

注意:一些ussl模块的实现没有验证服务器证书，这使得建立的SSL连接容易受到攻击。

### ● 异常类型

```
ussl.SSLError
```

不存在该异常，可直接沿用操作系统异常替代。

### ● 常量

```
ussl.CERT_NONE
```

```
ussl.CERT_OPTIONAL
```

```
ussl.CERT_REQUIRED
```

支持cert\_reqs参数的值

更多的内容可参考[ussl](#)

## 3.18 ustruct

ustruct 模块在 Python 值和以 Python 字节对象表示的 C 结构之间执行转换。

-- 支持 size/byte 的前缀: @, <, >, !.

-- 支持的格式代码: b, B, h, H, i, I, L, q, Q, s, P, f, d (最后 2 个需要支持浮点数)。

## ● 方法

`ustruct.calcsize(fmt)`

返回存放某一类型数据 `fmt` 需要的字节数。

`fmt`: 数据类型

`b` — 字节型

`B` — 无符号字节型

`h` — 短整型

`H` — 无符号短整型

`i` — 整型

`I` — 无符号整型

`l` — 整型

`L` — 无符号整型

`q` — 长整型

`Q` — 无符号长整型

`f` — 浮点型

`d` — 双精度浮点型

`P` — 无符号型

示例：

```
>>>  
print(ustruct.calcsize("i")) 4  
>>>  
print(ustruct.calcsize("B")) 1
```

`ustruct.pack(fmt, v1, v2, ...)`

按照格式字符串 `fmt` 打包参数 `v1, v2, ...`。返回值是参数打包后的字节对象。

`fmt`: 同上

示例：

```
>>> ustruct.pack("ii", 3, 2)  
b'\x03\x00\x00\x00\x02\x00\x00\x00'
```

`ustruct.unpack(fmt, data)`

从 `fmt` 中解包数据。返回值是解包后参数的元组。

`data`: 要解压的字节对象

示例：

```
>>> buf = ustruct.pack("bb", 1, 2)  
>>> print(buf)  
b'\x01\x02'  
>>> print(ustruct.unpack("bb", buf))  
(1, 2)
```

```
ustruct.pack_into(fmt, buffer, offset, v1, v2, ...)
```

按照格式字符串 `fmt` 压缩参数 `v1, v2, ...` 到缓冲区 `buffer`, 开始位置是 `offset`。当 `offset` 为负数时, 从缓冲区末尾开始计数。

```
ustruct.unpack_from(fmt, data, offset=0)
```

以 `fmt` 作为规则从 `data` 的 `offset` 位置开始解包数据, 如果 `offset` 是负数就是从缓冲区末尾开始计算。返回值是解包后的参数元组。

```
>>> buf =ustruct.pack("bb", 1, 2)
>>> print(ustruct.unpack("bb", buf))
(1, 2)
>>> print(ustruct.unpack_from("b", buf, 1))
(2,)
```

更多的内容可参考 [ustruct](#)

## 3.19 utime

`utime` 模块提供获取当前时间和日期、测量时间间隔和延迟的功能。

**初始时刻:** Unix 使用 POSIX 系统标准, 从 1970-01-01 00:00:00 UTC 开始。嵌入式程序从 2000-01-01 00:00:00 UTC 开始。

**保持实际日历日期/时间:** 需要一个实时时钟 (RTC)。在底层系统 (包括一些 RTOS 中), RTC 已经包含在其中。设置时间是通过 OS/RTOS 而不是 MicroPython 完成, 查询日期/时间也需要通过系统 API。对于裸板系统时钟依赖于 `machine.RTC()` 对象。设置时间通过 `machine.RTC().datetimedelta(tuple)` 方法, 并通过下面方式维持:

- 后备电池 (可能是选件、扩展板等)。
- 使用网络时间协议 (需要用户设置)。
- 每次上电时手工设置 (大部分只是在硬复位时需要设置, 少部分每次复位都需要设置)。

如果实际时间不是通过系统 / MicroPython RTC 维持, 那么下面方法结果可能不是和预期的相同。

### ● 方法

```
utime.localtime([secs])
```

从初始时间的秒转换为元组: (年, 月, 日, 时, 分, 秒, 星期, yearday)。如果 `secs` 是空或者 `None`, 那么使用当前时间。

- 1) `year` 年份包括世纪 (例如 2014)。
- 2) `month` 范围 1-12
- 3) `day` 范围 1-31
- 4) `hour` 范围 0-23

- 5) minute 范围 0-59
- 6) second 范围 0-59
- 7) weekday 范围 0-6 对应周一到周日
- 8) yearday 范围 1-366

#### `utime.mktime()`

时间的反方法，它的参数是完整 8 参数的元组，返回值一个整数自 2000 年 1 月 1 日以来的秒数。

#### `utime.sleep(seconds)`

休眠指定的时间（秒），Seconds 可以是浮点数。注意有些版本的 MicroPython 不支持浮点数，为了兼容可以使用 `sleep_ms()` 和 `sleep_us()` 方法。

#### `utime.sleep_ms(ms)`

延时指定毫秒，参数不能小于 0。

#### `utime.sleep_us(us)`

延时指定微秒，参数不能小于 0。

#### `utime.ticks_ms()`

返回不断递增的毫秒计数器，在某些值后会重新计数（未指定）。计数值本身无特定意义，只适合用在 `ticks_diff()`。

注：直接在这些值上执行标准数学运算（+，-）或关系运算符（<，>，>，>=）会导致无效结果。执行数学运算然后传递结果作为参数给 `ticks_diff()` 或 `ticks_add()` 也将导致方法产生无效结果。

#### `utime.ticks_us()`

和上面 `ticks_ms()` 类似，只是返回微秒。

#### `utime.ticks_cpu()`

与 `ticks_ms()` 和 `ticks_us()` 类似，具有更高精度（使用 CPU 时钟），并非每个端口都实现此功能。

#### `utime.ticks_add(ticks, delta)`

给定一个数字作为节拍的偏移值 `delta`，这个数字的值是正数或者负数都可以。给定一个 `ticks` 节拍值，本方法允许根据节拍值的模算数定义来计算给定节拍值之前或者之后 `delta` 个节拍的节拍值。`ticks` 参数必须是 `ticks_ms()`，`ticks_us()`，or `ticks_cpu()` 方法的直接返回值。然而，`delta` 可以是一个任意整数或者是数字表达式。`ticks_add` 方法对计算事件/任务的截至时间很有用。（注意：必须使用 `ticksdiff()` 方法来处理最后期限）。

代码示例：

```
## 查找 100ms 之前的节拍值
print(utime.ticks_add(utime.ticks_ms(), -100))

## 计算操作的截止时间然后进行测试
deadline = utime.ticks_add(utime.ticks_ms(), 200)
while utime.ticks_diff(deadline, utime.ticks_ms()) > 0:
    do_a_little_of_something()

## 找出本次移植节拍值的最大值
print(utime.ticks_add(0, -1))
```

utime.ticks\_diff(ticks1, ticks2)

计算两次调用 ticks\_ms(), ticks\_us(), 或 ticks\_cpu() 之间的时间。因为这些方法的计数值可能会回绕，所以不能直接相减，需要使用 ticks\_diff() 方法。“旧” 时间需要在“新” 时间之前，否则结果无法确定。这个方法不要用在计算很长的时间（因为 ticks\*() 方法会回绕，通常周期不是很长）。通常用法是在带超时的轮询事件中调用：

代码示例：

```
## 等待 GPIO 引脚有效，但是最多等待 500 微秒
start = utime.ticks_us()
while pin.value() == 0:
    if utime.ticks_diff(utime.ticks_us(), start) > 500:
        raise TimeoutError
```

utime.time()

返回从开始时间的秒数（整数），假设 RTC 已经按照前面方法设置好。如果 RTC 没有设置，方法将返回参考点开始计算的秒数（对于 RTC 没有后备电池的板子，上电或复位后的情况）。如果你开发便携版的 MicroPython 应用程序，你不要依赖方法来提供超过秒级的精度。如果需要高精度，使用 ticks\_ms() 和 ticks\_us() 方法。如果需要日历时间，使用不带参数的 localtime() 是更好选择。

### 与 CPython 的区别

在 CPython 中，这个方法用浮点数返回从 Unix 开始时间（1970-01-01 00:00 UTC）的秒数，通常是毫秒级的精度。在 MicroPython 中，只有 Unix 版才使用相同开始时间，如果允许浮点精度，将返回亚秒精度。嵌入式硬件通常没有用浮点数表示长时间访问和亚秒精度，所以返回值是整数。一些嵌入式系统硬件不支持 RTC 电池供电方式，所以返回的秒数是从最后上电、或相对某个时间、以及特定硬件时间（如复位）。

示例：

```
>>> import utime
>>> utime.sleep(1)           # sleep for 1 second
>>> utime.sleep_ms(500)      # sleep for 500 milliseconds
>>> utime.sleep_us(10)       # sleep for 10 microseconds
>>> start = utime.ticks_ms() # get value of millisecond counter
>>> delta = utime.ticks_diff(utime.ticks_ms(), start) # compute time difference
>>> delta
6928
>>> print(utime.ticks_add(utime.ticks_ms(), -100))
1140718
>>> print(utime.ticks_add(0, -1))
1073741823
```

更多内容可参考 [utime](#)

## 3.20 uzlib

`uzlib` 模块实现了使用 DEFLATE 算法解压缩二进制数据（常用的 `zlib` 库和 `gzip` 文档）。目前不支持压缩。

### ● 方法

`uzlib.decompress(data)`

返回解压后的 `bytes` 数据。

更多内容可参考 [uzlib](#)

## 3.21 \_thread

`_thread` 模块提供了用于处理多线程的基本方法——多个控制线程共享它们的全局数据空间。为了实现同步，提供了简单的锁（也称为互斥锁或二进制信号量）。

示例：

```
import _thread
import utime
def testThread():
    while True:
        print("Hello from thread")
        utime.sleep(2)

_thread.start_new_thread(testThread, ())
while True:
    pass
```

输出结果（启动新的线程，每隔两秒打印字符）：

```
Hello from thread Hello from thread Hello from thread Hello from thread Hello from thread
```

更多内容可参考 [thread](#)

## 4. MicroPython 特定库

### 4.1 machine

`machine` 模块包含与特定开发板上的硬件相关的特定函数。在这个模块中的大多数功能允许实现直接和不受限制地访问和控制系统上的硬件块（如 CPU，定时器，总线等）。如果使用不当，会导致故障，死机，崩溃，在极端的情况下，硬件会损坏。

需要注意的是，由于不同开发板的硬件资源不同，MicroPython 移植所能控制的硬件也是不一样的。因此对于控制硬件的例程来说，在使用前需要修改相关的配置参数来适配不同的开发板，或者直接运行已经对某一开发板适配好的 MicroPython 示例程序。

#### ● 函数

##### 1) 复位相关函数

`machine.info()`

显示关于系统介绍和内存占用等信息。

`machine.reset()` 注：暂未实现

重启设备，类似于按下复位按钮。

`machine.reset_cause()` 注：暂未实现

获得复位的原因，查看可能的返回值的常量。

##### 2) 中断相关函数

`machine.disable_irq()`

禁用中断请求。返回先前的 IRQ 状态，该状态应该被认为是一个未知的值。这个返回值应该在`disable_irq` 函数被调用之前被传给`enable_irq` 函数来重置中断到初始状态。

`machine.enable_irq(state)`

重新使能中断请求。状态参数应该是从最近一次禁用功能的调用中返回的值。

##### 3) 功耗相关函数

`machine.freq()`

返回 CPU 的运行频率。

`machine.idle()` 注：暂未实现

阻断给 CPU 的时钟信号，在较短或者较长的周期里减少功耗。当中断发生时，外设将继续工作。

`machine.sleep()` 注：暂未实现

停止 CPU 并禁止除了 WLAN 之外的所有外设。系统会从睡眠请求的地方重新恢复工作。

为了确保唤醒一定会发生，应当首先配置中断源。

`machine.deepsleep()` 注：暂未实现

停止 CPU 和所有外设（包括网络接口）。执行从主函数中恢复，就像被复位一样。复位的原因可以检查 `machine.DEEPSLEEP` 参数获得。为了确保唤醒一定会发生，应该首先配置中断源，比如一个引脚的变换或者 RTC 的超时。

## 4.2 machine.Pin

`machine.Pin` 类是 `machine` 模块下面的一个硬件类，用于对引脚的配置和控制，提供对 Pin 设备的操作方法。

Pin 对象用于控制输入/输出引脚（也称为 GPIO）。Pin 对象通常与一个物理引脚相关联，他可以驱动输出电压和读取输入电压。Pin 类中有设置引脚模式（输入/输出）的方法，也有获取和设置数字逻辑（0 或 1）的方法。

一个 Pin 对象是通过一个标识符来构造的，它明确地指定了一个特定的输入输出。标识符的形式和物理引脚的映射是特定于一次移植的。标识符可以是整数，字符串或者是一个带有端口和引脚号码的元组。在 K210-MicroPython 中，引脚标识符是一个由代号和引脚号组成的元组，如 `Pin(("PB15", 31), Pin.OUT_PP)` 中的 ("PB15", 31)。

### ● 构造函数

在 Rt-Thread MicroPython 中 Pin 对象的构造函数如下：

```
class machine.Pin( id, mode = -1, pull = -1, value)
```

-- id : 由用户自定义的引脚名和 Pin 设备引脚号组成，如 ("PB15", 31)，  
“PB15” 为用户自定义的引脚名，31 为 Rt-Thread Pin 设备驱动在本次移植中的引脚号。

-- mode : 指定引脚模式，可以是以下几种：

✓ Pin.IN : 输入模式

✓ Pin.OUT\_PP : 输出模式

✓ Pin.OUT\_OD : 开漏模式

-- pull : 如果指定的引脚连接了上拉下拉电阻，那么可以配置成下面的状态：

None : 没有上拉或者下拉电阻。

✓ Pin.PULL\_UP : 使能上拉电阻。

✓ Pin.PULL\_DOWN : 使能下拉电阻。

-- value : value 的值只对输出模式和开漏输出模式有效，用来设置初始输出值。

## ● 方法

`Pin.init(mode=-1, pull=-1, *, value, drive, alt)`

根据输入的参数重新初始化引脚。只有那些被指定的参数才会被设置，其余引脚的状态将保持不变，详细的参数可以参考上面的构造函数。

`Pin.value([x])`

如果没有给定参数 `x`，这个方法可以获得引脚的值。如果给定参数 `x`，如 0 或 1，那么设置引脚的值为逻辑 0 或逻辑 1。

`Pin.name()`

返回引脚对象在构造时用户自定义的引脚名。

`Pin.irq(handler=None, trigger=(Pin.IRQ_RISING))`

配置在引脚的触发源处于活动状态时调用的中断处理程序。如果引脚模式是，`Pin.IN` 则触发源是引脚上的外部值。如果引脚模式是，`Pin.OUT` 则触发源是引脚的输出缓冲器。否则，如果引脚模式是，`Pin.OPEN_DRAIN` 那么触发源是状态'0'的输出缓冲器和状态'1'的外部引脚值。

参数：

-- `handler` 是一个可选的函数，在中断触发时调用

-- `trigger` 配置可以触发中断的事件。可能的值是：

- ✓ `Pin.IRQ_FALLING` 下降沿中断
- ✓ `Pin.IRQ_RISING` 上升沿中断
- ✓ `Pin.IRQ_RISING_FALLING` 上升沿或下降沿中断
- ✓ `Pin.IRQ_LOW_LEVEL` 低电平中断
- ✓ `Pin.IRQ_HIGH_LEVEL` 高电平中断

## ● 常量

下面的常量用来配置 `Pin` 对象。

1) 选择引脚模式：

- ✓ `Pin.IN`
- ✓ `Pin.OUT_PP`
- ✓ `Pin.OUT_OD`

2) 选择上/下拉模式：

- ✓ `Pin.PULL_UP`
- ✓ `Pin.PULL_DOWN`
- ✓ `None` 使用值 `None` 代表不进行上下

3) 选择中断触发模式:

- ✓ Pin. IRQ\_FALLING
- ✓ Pin. IRQ\_RISING
- ✓ Pin. IRQ\_RISING\_FALLING
- ✓ Pin. IRQ\_LOW\_LEVEL
- ✓ Pin. IRQ\_HIGH\_LEVEL

### ● 示例一

控制引脚输出高低电平信号，并读取按键引脚电平信号。

```
from machine import Pin

PIN_OUT = 31
PIN_IN = 58

p_out = Pin("PB15", PIN_OUT), Pin.OUT_PP)
p_out.value(1)                      # set io high
p_out.value(0)                      # set io low

p_in = Pin("key_0", PIN_IN), Pin.IN, Pin.PULL_UP)
print(p_in.value())                 # get value, 0 or 1
```

### ● 示例二

```
from machine import Pin

PIN_KEY0 = 58      # PD10
key_0 = Pin("key_0", PIN_KEY0), Pin.IN, Pin.PULL_UP)

def func(v):
    print("Hello Rt-Thread!")

key_0.irq(trigger=Pin.IRQ_RISING, handler=func)
```

上升沿信号触发引脚中断后执行中断处理函数。

更多内容可参考 [machine.Pin](#)。

## 4.3 machine.UART

`machine.UART` 类是 `machine` 模块下面的一个硬件类，用于对 UART 的配置和控制，提供对 UART 设备的操作方法。UART 实现了标准的 `uart/usart` 双工串行通信协议，在物理层上，它由两根数据线组成：RX 和 TX。通信单元是一个字符，它可以是 8 或 9 位宽。

### ● 构造函数

在 Rt-Thread MicroPython 中 UART 对象的构造函数如下：

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

在给定总线上构造一个 UART 对象，`id` 取决于特定的移植。初始化参数可以参考下面的 `UART.init` 方法。

使用硬件 UART 在初始化时只需传入 UART 设备的编号即可，如传入 1 表示 `uart1` 设备。初始化方式可参考示例。

### ● 方法

```
UART.init(baudrate = 9600, bits=8, parity=None, stop=1)
```

— `baudrate` : SCK 时钟频率。

— `bits` : 每次发送数据的长度。

— `parity` : 校验方式。

— `stop` : 停止位的长度。

```
UART.deinit()
```

关闭串口总线。

```
UART.read([nbytes])
```

读取字符，如果指定读 `n` 个字节，那么最多读取 `n` 个字节，否则就会读取尽可能多的数据。返回值：一个包含读入数据的字节对象。如果超时则返回 `None`。

```
UART.readinto(buf[, nbytes])
```

读取字符到 `buf` 中，如果指定读 `n` 个字节，那么最多读取 `n` 个字节，否则就读取尽可能多的数据。另外读取数据的长度不超过 `buf` 的长度。返回值：读取并存储到 `buf` 中的字节数。如果超时则返回 `None`。

```
UART.readline()
```

读一行数据，以换行符结尾。返回值：读入的行数，如果超时则返回 `None`。

```
UART.write(buf)
```

将 `buf` 中的数据写入总线。返回值：写入的字节数，如果超时则返回 `None`。

示例：

在构造函数的第一个参数传入 1，系统就会搜索名为 `uart1` 的设备，找到之后使用这个设备来构建UART 对象：

```
from machine import UART
uart = UART(1, 115200)                      # init with given baudrate
uart.init(115200, bits=8, parity=None, stop=1) # init with given parameters
uart.read(10)        # read 10 characters, returns a bytes object
uart.read()         # read all available characters
uart.readline()     # read a line
uart.readinto(buf) # read and store into the given buffer
uart.write('abc')   # write the 3 characters
```

更多内容可参考 [machine.UART](#)。

## 4.4 machine.SPI

`machine.SPI` 类是 `machine` 模块下面的一个硬件类，用于对 SPI 的配置和控制，提供对 SPI 设备的操作方法。

SPI 是一个由主机驱动的同步串行协议。在物理层，总线有三根:SCK、MOSI、MISO。多个设备可以共享同一总线，每个设备都由一个单独的信号 SS 来选中，也称片选信号。主机通过片选信号选定一个设备进行通信。SS 信号的管理应该由用户代码负责。（通过 `machine.Pin`）

### ● 构造函数

在 Rt-Thread MicroPython 中 SPI 对象的构造函数如下：

```
class machine.SPI(id, ...)
```

在给定总线上构造一个 SPI 对象，`id` 取决于特定的移植。

如果想要使用软件 SPI，即使用引脚模拟 SPI 总线，那么初始化的第一个参数需要设置为 -1，可参考软件 SPI 示例。

使用硬件 SPI 在初始化时只需传入 SPI 设备的编号即可，如 '50' 表示 SPI5 总线上的第 0 个设备。初始化方式可参考硬件 SPI 示例。

如果没有额外的参数，SPI 对象会被创建，但是不会被初始化，如果给出额外的参数，那么总线将被初始化，初始化参数可以参考下面的 `SPI.init` 方法。

### ● 方法

```
SPI.init    SPI.init(baudrate=1000000, *, polarity=0, phase=0, bits=8,  
firstbit=SPI.MSB, sck=None, mosi=None, miso=None)
```

用给定的参数初始化 SPI 总线：

- `baudrate` :SCK 时钟频率。
- `polarity` : 极性可以是 0 或 1，是时钟空闲时所处的电平。
- `phase` : 相位可以是 0 或 1，分别在第一个或者第二个时钟边缘采集数据。
- `bits` : 每次传输的数据长度，一般是 8 位。
- `firstbit` : 传输数据从高位开始还是从低位开始，可以是 `SPI.MSB` 或者 `SPI.LSB`。
- `sck` : 用于 `sck` 的 `machine.Pin` 对象。
- `mosi` : 用于 `mosi` 的 `machine.Pin` 对象。
- `miso` : 用于 `miso` 的 `machine.Pin` 对象。

```
SPI.deinit()
```

关闭 SPI 总线。

```
SPI.read(nbytes, write=0x00)
```

读出 `n` 字节的同时不断的写入 `write` 给定的单字节。返回一个存放着读出数据的字节对

象。

`SPI.readinto(buf, write=0x00)`

读出 `n` 字节到 `buf` 的同时不断地写入 `write` 给定的单字节。这个方法返回读入的字节数。

`SPI.write(buf)`

写入 `buf` 中包含的字节。返回 `None`。

`SPI.write_readinto(write_buf, read_buf)`

在读出数据到 `readbuf` 时，从 `writebuf` 中写入数据。缓冲区可以是相同的或不同，但是两个缓冲区必须具有相同的长度。返回 `None`。

## ● 常量

`SPI.MSB`

设置从高位开始传输数据。

`SPI.LSB`

设置从低位开始传输数据。

## ● 示例

软件模拟 SPI

```
>>> from machine import Pin, SPI
>>> clk = Pin("clk", 26), Pin.OUT_PP
>>> mosi = Pin("mosi", 27), Pin.OUT_PP
>>> miso = Pin("miso", 28), Pin.IN
>>> spi = SPI(-1, 500000, polarity = 0, phase = 0, bits = 8, firstbit = 0, sck = clk
    , mosi = mosi, miso = miso)
>>> print(spi)
SoftSPI(baudrate=500000, polarity=0, phase=0, sck=clk, mosi=mosi, miso=miso)
>>> spi.write("hello Rt-Thread!")
>>> spi.read(10)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

硬件 SPI

需要先开启 SPI 设备驱动，查找设备可以在 `msh` 中输入 `list_device` 命令。在构造函数的第一个参数传入 50，系统就会搜索名为 `spi50` 的设备，找到之后使用这个设备来构建 SPI 对象：

```
>>> from machine import SPI
>>> spi = SPI(50)
>>> print(spi)
SPI(device port : spi50)
>>> spi.write(b'\x9f')
>>> spi.read(5)
b'\xff\xff\xff\xff\xff'
>>> buf = bytearray(1)
>>> spi.write_readinto(b"\x9f",buf)
>>> buf
bytearray(b'\xef')
>>> spi.init(100000,0,0,8,1)      # Resetting SPI parameter
```

更多内容可参考 [machine.SPI](#)。

## 4.5 machine.I2C

`machine.I2C` 类是 `machine` 模块下面的一个硬件类，用于对 I2C 的配置和控制，提供对 I2C 设备的操作方法。

I2C 是一种用于设备间通信的两线协议。在物理层上，它由两根线组成：SCL 和 SDA，即时钟和数据线。

I2C 对象被创建到一个特定的总线上，它们可以在创建时被初始化，也可以之后再来初始化。打印 I2C 对象会打印出配置时的信息。

### ● 构造函数

在 Rt-Thread MicroPython 中 I2C 对象的构造函数如下：

```
class machine.I2C(id=-1, scl, sda, freq=400000)
```

使用下面的参数构造并返回一个新的 I2C 对象：

-- `id`：标识特定的 I2C 外设。如果填入 `id = -1`，即选择软件模拟的方式实现 I2C，这时可以使用任意引脚来模拟 I2C 总线，这样在初始化时就必须指定 `scl` 和 `sda`。软件 I2C 的初始化方式可参考软件 I2C 示例。硬件 I2C 的初始化方式可参考硬件 I2C 示例。

-- `scl`：应该是一个 Pin 对象，指定为一个用于 `scl` 的 Pin 对象。

-- `sda`：应该是一个 Pin 对象，指定为一个用于 `sda` 的 Pin 对象。

-- `freq`：应该是为 `scl` 设置的最大频率。

### ● 方法

```
I2C.init(scl, sda, freq=400000)
```

初始化 I2C 总线，参数介绍可以参考构造函数中的参数。

```
I2C.deinit()
```

关闭 I2C 总线。

```
I2C.scan()
```

扫描所有 0x08 和 0x77 之间的 I2C 地址，然后返回一个有响应地址的列表。如果一个设备在总线上收到了他的地址，就会通过拉低 SDA 的方式来响应。

## ● I2C 基础方法

下面的方法实现了基本的 I2C 总线操作，可以组合成任何的 I2C 通信操作，如果需要对总线进行更多的控制，可以使用他们，否则可以使用后面介绍的标准使用方法。

`I2C.start()`

在总线上产生一个启动信号。 (SCL 为高时, SDA 转换为低)

`I2C.stop()`

在总线上产生一个停止信号。 (SCL 为高时, SDA 转换为高)

`I2C.readinto(buf, nack=True)`

从总线上读取字节并将他们存储到 buf 中，读取的字节数是 buf 的长度。在收到最后一个字节以外的所有内容后，将在总线上发送 ACK。在收到最后一个字节之后，如果 NACK 是正确的，那么就会发送一个 NACK，否则将会发送 ACK。

`I2C.write(buf)`

将buf中的数据接入到总线，检查每个字节之后是否收到 ACK，并在收到 NACK 时停止传输剩余的字节。这个函数返回接收到的 ACK 的数量。

## ● I2C 标准总线操作

下面的方法实现了标准 I2C 主设备对一个给定从设备的读写操作。

`I2C.readfrom(addr, nbytes, stop=True)`

从 addr 指定的从设备中读取 n 个字节，如果 stop = True，那么在传输结束时会产生一个停止信号。函数会返回一个存储着读到数据的字节对象。

`I2C.readfrom_into(addr, buf, stop=True)`

从 addr 指定的从设备中读取数据存储到 buf 中，读取的字节数将是 buf 的长度，如果 stop = True，那么在传输结束时会产生一个停止信号。这个方法没有返回值。

`I2C.writeto(addr, buf, stop=True)`

将 buf 中的数据写入到 addr 指定的从设备中，如果在写的过程中收到了 NACK 信号，那么就不会发送剩余的字节。如果 stop = True，那么在传输结束时会产生一个停止信号，即使收到一个 NACK。这个函数返回接收到的 ACK 的数量。

## ● 内存操作

一些 I2C 设备充当一个内存设备，可以读取和写入。在这种情况下，有两个与 I2C 相关的地址，从机地址和内存地址。下面的方法是与这些设备进行通信的便利函数。

`I2C.readfrom_mem(addr, memaddr, nbytes, *, addrsize=8)`

从 addr 指定的从设备中 memaddr 地址开始读取 n 个字节。addrsize 参数指定地址的长度。返回一个存储读取数据的字节对象。

`I2C.readfrom_mem_into(addr, memaddr, buf, *, addrsize=8)`

从 `addr` 指定的从设备中 `memaddr` 地址读取数据到 `buf` 中，，读取的字节数是 `buf` 的长度。这个方法没有返回值。

```
I2C.writeto_mem(addr, memaddr, buf, *, addrsize=8)
```

将 `buf` 里的数据写入 `addr` 指定的从机的 `memaddr` 地址中。这个方法没有返回值。

## ● 示例

### 软件模拟 I2C

```
>>> from machine import Pin, I2C
>>> clk = Pin(("clk", 29), Pin.OUT_OD)      # Select the 29 pin device as the clock
>>> sda = Pin(("sda", 30), Pin.OUT_OD)      # Select the 30 pin device as the data line
>>> i2c = I2C(-1, clk, sda, freq=100000) # create I2C peripheral at frequency of 100 kHz

>>> i2c.scan()                           # scan for slaves, returning a list of 7-bit
addresses
[81]                                         # Decimal representation
>>> i2c.writeto(0x51, b'123') 3        # write 3 bytes to slave with 7-bit address 42
>>> i2c.readfrom(0x51, 4)
42                                           # read 4 bytes from slave with 7-bit address
b'\xf8\xc0\xc0\xc0'
>>> i2c.readfrom_mem(0x51, 0x02, 1)
),                                              # read 1 bytes from memory of slave 0x51(7-bit
b'\x12'                                         # starting at memory-address 8 in the slave
>>> i2c.writeto_mem(0x51, 2, b'\x10') # write 1 byte to memory of slave 42,
                                         # starting at address 2 in the slave
```

### 硬件 I2C

需要先开启 I2C 设备驱动，查找设备可以在 `msh` 中输入 `list_device` 命令。在构造函数的第一个参数传入 0，系统就会搜索名为 `i2c0` 的设备，找到之后使用这个设备来构建 I2C 对象：

```
>>> from machine import Pin, I2C
>>> i2c = I2C(0)                         # create I2C peripheral at frequency of 100kHz
>>> i2c.scan()                           # scan for slaves, returning a list of 7-bit
addresses
[81]                                         # Decimal representation
```

更多内容可参考 [machine.I2C](#)。

## 4.6 FPIOA

### ● 构造函数

```
class k210.FPIOA()
```

### ● 方法

```
bool set_function(int pin, int func, int set_sl, int set_st, int set_io_driving)
```

设置引脚功能

-- pin : 引脚号, 取值范围 0 到 48

-- func : 引脚功能, 取值范围见附 F1

-- set\_sl : (可选参数)

-- set\_st : (可选参数)

-- set\_io\_driving : (可选参数)

```
int get_io_by_function(int func)
```

查看某功能所在引脚

-- func : 取值范围见附 F1

附 F1:

```
JTAG_TCLK JTAG_TDI JTAG_TMS JTAG_TDO
SPI0_D0 SPI0_D1 SPI0_D2 SPI0_D3
SPI0_D4 SPI0_D5 SPI0_D6 SPI0_D7
SPI0_SS0 SPI0_SS1 SPI0_SS2 SPI0_SS3
SPI0_ARB SPI0_SCLK UARTHS_RX UARTHS_TX
CLK_SPI1 CLK_I2C1 GPIOHS0 GPIOHS1 GPIOHS2
GPIOHS3 GPIOHS4 GPIOHS5 GPIOHS6
GPIOHS7 GPIOHS8 GPIOHS9 GPIOHS10
GPIOHS11 GPIOHS12 GPIOHS13 GPIOHS14
GPIOHS15 GPIOHS16 GPIOHS17 GPIOHS18
GPIOHS19 GPIOHS20 GPIOHS21 GPIOHS22
GPIOHS23 GPIOHS24 GPIOHS25 GPIOHS26
GPIOHS27 GPIOHS28 GPIOHS29 GPIOHS30
GPIOHS31 GPIO00 GPIO01 GPIO02
GPIO03 GPIO04 GPIO05 GPIO06
GPIO07 UART1_RX UART1_TX UART2_RX
UART2_TX UART3_RX UART3_TX SPI1_D0
SPI1_D1 SPI1_D2 SPI1_D3 SPI1_D4
SPI1_D5 SPI1_D6 SPI1_D7 SPI1_SS0
SPI1_SS1 SPI1_SS2 SPI1_SS3 SPI1_ARB
SPI1_SCLK SPI_SLAVE_D0 SPI_SLAVE_SS SPI_SLAVE_SCLK
```

```
I2S0_MCLK I2S0_SCLK I2S0_WS I2S0_IN_D0
I2S0_IN_D1 I2S0_IN_D2 I2S0_IN_D3 I2S0_OUT_D0
I2S0_OUT_D1 I2S0_OUT_D2 I2S0_OUT_D3 I2S1_MCLK
I2S1_SCLK I2S1_WS I2S1_IN_D0 I2S1_IN_D1
I2S1_IN_D2 I2S1_IN_D3 I2S1_OUT_D0 I2S1_OUT_D1
I2S1_OUT_D2 I2S1_OUT_D3 I2S2_MCLK I2S2_SCLK
I2S2_WS I2S2_IN_D0 I2S2_IN_D1 I2S2_IN_D2
I2S2_IN_D3 I2S2_OUT_D0 I2S2_OUT_D1 I2S2_OUT_D2
I2S2_OUT_D3 I2C0_SCLK I2C0_SDA I2C1_SCLK
I2C1_SDA I2C2_SCLK I2C2_SDA CMOS_XCLK
CMOS_RST CMOS_PWDN CMOS_VSYNC CMOS_HREF
CMOS_PCLK CMOS_D0 CMOS_D1 CMOS_D2
CMOS_D3 CMOS_D4 CMOS_D5 CMOS_D6
CMOS_D7 SCCB_SCLK SCCB_SDA UART1_CTS
UART1_DSR UART1_DCD UART1_RI UART1_SIR_IN
UART1_DTR UART1 RTS UART1_OUT2 UART1_OUT1
UART1_SIR_OUT UART1_BAUD UART1_RE UART1_DE
UART1_RS485_EN UART2_CTS UART2_DSR UART2_DCD
UART2_RI
UART2_SIR_IN
```

```
UART2_DTR
UART2_RTS
UART2_OUT2
UART2_OUT1
UART2_SIR_OUT
UART2_BAUD
UART2_RE
UART2_DE
UART2_RS485_EN
UART3_CTS
UART3_DSR
UART3_DCD
UART3_RI
UART3_SIR_IN
UART3_DTR
UART3_RTS
UART3_OUT2
UART3_OUT1
UART3_SIR_OUT
UART3_BAUD
UART3_RE
UART3_DE
UART3_RS485_EN
TIMER0_TOGGLE1
TIMER0_TOGGLE2
TIMER0_TOGGLE3
TIMER0_TOGGLE4
TIMER1_TOGGLE1
TIMER1_TOGGLE2
```

```
TIMER1_TOGGLE3
TIMER1_TOGGLE4
TIMER2_TOGGLE1
TIMER2_TOGGLE2
TIMER2_TOGGLE3
TIMER2_TOGGLE4
CLK_SPI2
CLK_I2C2
```

### ● 示例

```
from k210 import FPIOA

fpioa = FPIOA()
fpioa.set_function(1, fpioa.GPIOHS0)
fpioa.get_io_by_function(fpioa.GPIOHS0)
```

## 4.7 I2S

### ● 构造函数

```
class k210.I2S(int bus, int mode)
-- bus : I2S 总线编号, 取值范围 0 到 2
-- mode : 收/发模式, 取值 TRANSMITTER(播放), RECEIVER(录音)
```

### ● 方法

```
void init()
```

初始化

```
void set_param(int sample_rate, int bps, int track_num)
```

设置参数

```
-- sample_rate: 采样率
```

```
-- bps : 采样位数
```

```
-- track_num : 音道数, 取值 1 或 2
```

```
void play(bytarray pcm)
```

播放

```
-- pcm : PCM 格式的音频数据
```

```
bytarray record(void)
```

录音

### ● 示例

实时播放录音

```
from k210 import I2S

i2sp = I2S(0, I2S.TRANSMITTER)
i2sp.init()
i2sp.set_param(8000, bps = 16, track_num = 2)

i2sr = I2S(1, I2S.RECEIVER)
i2sr.init()
i2sr.set_param(8000)

while (True):
    pcm = i2sr.record()
    i2sp.play(pcm)
```

播放三角函数

```
import math
from k210 import I2S

sample_rate = 8000

i2s=I2S(0, I2S.TRANSMITTER)
i2s.init()
i2s.set_param(sample_rate, bps = 16, track_num = 1)

def sin_sound(freq, len, samrate):
    pcm = bytearray()

    step = (2 * math.pi * freq)/samrate

    for i in range(0, len):
        val = 5000 * math.sin(i * step)
        iv = int(val)
        pcm += iv.to_bytes(2, "little", True)

    return pcm
```

## 4.8 camera

### ● 构造函数

```
class k210.camera()
```

### ● 方法

```
void reset()
```

初始化

```
void set_pixformat(int fmt)
```

设置图像格式

-- fmt : 格式, 取值 RGB565

```
void set_framesize(int width, int height)
```

设置图像大小

```
picture snapshot()
```

获取单幅图像

return : 自定义的 picture 对象

### ● 示例

```
from k210 import camera
from k210 import picture

cam = camera()

cam.reset()
cam.set_pixformat(cam.RGB565)
cam.set_framesize(320, 240)

for i in range(0, 1000):
    img = cam.snapshot()
    img.show()

cam.set_pixformat(cam.YUV422)
for i in range(0, 1000):
    img = cam.snapshot()
    img.show()
```

## 4.9 FFT

- 构造函数

```
class k210.FFT()
```

- 方法

```
list<int real, int imag> run(int[] input, int shift, int direction)
```

计算 fft

参数说明：

-- input : 待变换的数据，类型：int 数组

-- shift : 默认 0

-- direction : 取值 DIR\_BACKWARD/DIR\_FORWARD

返回值说明：

-- return : 返回包含实部和虚部的列表，虚/实为 int 类型

- 示例

```
from k210 import FFT
import math

fft = FFT()
```

## 4.10 KPU

### ● 构造函数

```
class k210.KPU()
```

### ● 方法

```
load_kmodel(path, size)
```

加载 kmodel 模型

参数说明：

-- path : 文件系统路径或 flash 地址, 参数类型 string/int

-- size : path 为 flash 地址时, 指定模型大小

```
run(input, dma)
```

运行模型推理

参数说明：

-- input : omv 图片对象或文件路径

-- dma : 指定 dma 通道, 整型, 默认: 5

```
list get_output(index, getlist)
```

获取模型推理结果

参数说明：

-- index : 整型, 0 获取全部结果 (默认); >0, 获取对应结果

-- getlist : Boolean, 是否将结果返回为 list, 默认: FALSE

返回值说明：

-- return : 模型推理结果的 list

```
regionlayer_init(anchor_num, crood_num, landm_num, cls_num, in_w, in_h, obj_thresh, n  
ms_thresh, variances, max_num, anchor)
```

yolo 相关 api, 初始化相关层

参数	描述	取值 (默认)
anchor_num	-	3160
crood_num	-	4
landm_num	-	5
cls_num	-	1
in_w	-	320
in_h	-	240
obj_thresh	-	0.7
nms_thresh	-	0.4
variances	-	[0.1, 0.2]
max_num	-	200
anchor	-	见 prior.h 文件

```
regionlayer()  
YOLO 模型相关 api, 处理 yolo 输出结果  
返回值说明:  
return : 返回检测边界框坐标
```

### ● 示例一：mnist 手写数字识别

mnist 手写数字识别，输出为 10 个数字概率，对输出结果判断，概率值最大的索引即为识别数字。

1) 准备文件

```
model: uint8_mnist_cnn_model.kmodel  
mnist dataset: infer.bin
```

2) 示例代码

```
#demo : Mnist_cnn  
import lcd,sensor,image,k210, gc  
sensor.reset()  
sensor.set_pixformat(sensor.RGB565)  
sensor.set_framesize(sensor.QVGA)  
lcd.init()  
m = k210.KPU()  
m.load_kmodel("/uint8_mnist_cnn_model.kmodel")  
# m.load_kmodel(0xb00000,540*1024) #load from flash  
m.run("/infer.bin")  
out=m.get_output(getlist=True)  
out[0].index(max(out[0]))
```

## ● 示例二： face detection

人脸检测 demo，模块实现对预测边界框的 NMS 处理。输出结果即为人脸检测的有效预测框坐标；

1) 准备文件

model: ulffd\_landmark.kmodel

2) 示例代码

```
#demo : face_landmark
import lcd,sensor,image,k210, gc
sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
lcd.init()
m = k210.KPU()
m.load_kmodel("/ulffd_landmark.kmodel")
m.regionlayer_init(anchor_num=3160,cood_num=4,max_num=200)
m

while(True):
    gc.collect()
    img=sensor.snapshot()
    m.run(img)
    m.get_output(0)
    dect = m.regionlayer()
    print("dect:",dect)
    for l in dect :
        img.draw_rectangle(l[0],l[1],l[2]-l[0],l[3]-l[1])
    lcd.display(img) #请注意缩进，属于while代码内.
```

## 4.11 machine.Timer

`machine.Timer` 类是 `machine` 模块下的一个硬件类，用于 `Timer` 设备的配置和控制，提供对 `Timer` 设备的操作方法。

-- `Timer` (硬件定时器)，是一种用于处理周期性和定时性事件的设备。

-- `Timer` 硬件定时器主要通过内部计数器模块对脉冲信号进行计数，实现周期性设备控制的功能。

-- `Timer` 硬件定时器可以自定义超时时间和超时回调函数，并且提供两种定时器模式：

■ `ONE_SHOT`: 定时器只执行一次设置的回调函数；

■ `PERIOD`: 定时器会周期性执行设置的回调函数；

-- 打印 `Timer` 对象会打印出配置的信息。

### ● 构造函数

在 Rt-Thread MicroPython 中 `Timer` 对象的构造函数如下：

```
class machine.Timer(id)
```

参数说明：

-- `id`: 使用的 `Timer` 设备编号，`id = 1` 表示编号为 1 的 `Timer` 设备，或者表示使用的 `timer` 设备名，如`id = "timer"` 表示设备名为 `timer` 的 `Timer` 设备；该函数主要用于通过设备编号创建 `Timer` 设备对象。

### ● 方法

```
Timer.init(mode = Timer.PERIODIC, period = 0, callback = None)
```

参数说明：

-- `mode`: 设置 `Timer` 定时器模式，可以设置两种模式:`ONE_SHOT` (执行一次)、`PERIOD` (周期性执行)，默认设置的模式为 `PERIOD` 模式；

-- `period`: 设置 `Timer` 定时器定时周期，单位：毫秒 (ms)

-- `callback`: 设置 `Timer` 定义器超时回调函数，默认设置的函数为 `None` 空函数，设置的函数格式如下所示：

```
def callback_test(device):          # 回调函数有且只有一个入参，为创建的 Timer 对象
    print("Timer callback test")
    print(device)                   # 打印 Timer 对象配置信息
```

该函数使用方式如下示例所示：

```
timer.init(wdt.PERIOD, 5000, callback_test)  # 设置定时器模式为周期性执行，超时时间为 5 秒，超时函数为 callback_test
```

### Timer.deinit()

该函数用于停止并关闭 Timer 设备。

#### ● 常量

下面的常量用来配置 Timer 对象。

选择定时器模式:

- Timer.PERIODIC
- Timer.ONE\_SHOT

#### ● 示例

```
>>> from machine import Timer # 从 machine 导入 Timer 类
>>> timer = Timer(15) # 创建 Timer 对象, 当前设备编号
为 11
>>> # 进入粘贴模式
paste mode; Ctrl-C to cancel, Ctrl-D to finish
==> def callback_test(device): # 定义超时回调函数
    print("Timer callback test")
>>> timer.init(timer.PERIODIC, 5000, callback_test) # 初始化 Timer 对象, 设置定时器
模式为循环执行, 超时时间为 5 秒, 超时回调函数 callback_test
>>> Timer callback test # 5 秒超时循环执行回调函数, 打印
日志
>>> Timer callback test
>>> Timer callback test
>>> timer.init(timer.ONE_SHOT, 5000, callback_test) # 设置定时器模式为只执行一次, 超
时时间为 5 秒, 超时回调函数为 callback_test
>>> Timer callback test # 5 秒超时后执行一次回调函数, 打
印日志
>>> timer.deinit() # 停止并关闭 Timer 定时器
```

更多内容可参考 [machine.Timer](#)。

## 4.12 machine.PWM

`machine.PWM` 类是 `machine` 模块下的一个硬件类，用于指定 PWM 设备的配置和控制，提供对 PWM 设备的操作方法。

-- PWM (Pulse Width Modulation, 脉冲宽度调制) 是一种对模拟信号电平进行数字编码的方式；

-- PWM 设备可以通过调节有效电平在一个周期信号中的比例时间来操作设备；

-- PWM 设备两个重要的参数：频率 (`freq`) 和占空比 (`duty`) 【频率：从一个上升沿（下降沿）到下一个上升沿（下降沿）的时间周期，单位为 Hz；占空比：有效电平（通常为电平）在一个周期内的时间比例】；

### ● 构造函数

在 Rt-Thread MicroPython 中 PWM 对象的构造函数如下：

```
class machine.PWM(id, channel, freq, duty)
```

在给定的总线上构建一个 PWM 对象，参数介绍如下：

-- `id`: 使用的 PWM 设备编号，如 `id = 1` 表示编号为 1 的 PWM 设备，或者表示使用的 PWM 设备名，如 `id = "pwm"` 表示设备名为 `pwm` 的 PWM 设备；

-- `channel`: 使用的 PWM 设备通道号，每个 PWM 设备包含多个通道，范围为 [0, 4]；

-- `freq`: 初始化频率，范围 [1, 156250]；

-- `duty`: 初始化占空比数值，范围 [0 255]；

例如：`PWM(1, 4, 100, 100)` 表示当前使用编号为 1 的 PWM 设备的 4 通道，初始化频率为 1000 Hz，初始化占空比的数值为 100。

### ● 方法

```
PWM.init(channel, freq, duty)
```

根据输入的参数初始化 PWM 对象，参数说明同上。

```
PWM.deinit()
```

用于关闭 PWM 对象，对象 `deinit` 之后需要重新 `init` 才能使用。

```
PWM.freq(freq)
```

用于获取或者设置 PWM 对象的频率，频率的范围为 [1, 156250]。如果参数为空，返回当前 PWM 对象的频率；如果参数非空，则使用该参数设置当前 PWM 对象的频率。

```
PWM.duty(duty)
```

用于获取或者设置 PWM 对象的占空比数值，占空比数值的范围为 [0, 255]，例如 `duty = 100`，表示当前设备占空比为  $100/255 = 39.22\%$ 。如果参数为空，返回当前 PWM 对象的占空比数值；如果参数非空，则使用该参数设置当前 PWM 对象的占空比数值。

## ● 示例

```
>>> from machine import PWM      # 从 machine 导入 PWM 类
>>> pwm = PWM(3, 3, 1000, 100)    # 创建 PWM 对象，当前使用编号为 3 的 PWM 设备的 3 通道，初始化的频率为 1000Hz，占空比数值为 100 (占空比为 100/255 = 39.22%)
>>> pwm.freq(2000)              # 设置 PWM 对象频率
>>> pwm.freq()                  # 获取 PWM 对象频率
2000
>>> pwm.duty(200)              # 设置 PWM 对象占空比数值
>>> pwm.duty()                  # 获取 PWM 对象占空比数值
200
>>> pwm.deinit()                # 关闭 PWM 对象
>>> pwm.init(3, 1000, 100)       # 开启并重新配置 PWM 对象
```

更多内容可参考 [machine.PWM](#)。

## 4.13 machine.RTC

`machine.RTC` 类是 `machine` 模块下面的一个硬件类，用于对指定 RTC 设备的配置和控制，提供对 RTC设备的操作方法。

RTC (Real-Time Clock ) 实时时钟可以提供精确的实时时间，它可以用子产生年、月、日、时、分、秒等信息。

### ● 构造函数

在 Rt-Thread MicroPython 中 RTC 对象的构造函数如下：

```
class machine.RTC()
```

所以在给定的总线上构造一个 RTC 对象，无入参对象，使用方式可参考示例。

### ● 方法

```
RTC.init(datetime)
```

根据传入的参数初始化 RTC 设备起始时间。入参 `datetime` 为一个时间元组，格式如下：

```
(year, month, day, wday, hour, minute, second, yday)
```

参数介绍如下所示：

- `year`: 年份;
- `month`: 月份，范围 [1, 12];
- `day`: 日期，范围 [1, 31];
- `wday`: 星期，范围 [0, 6]，0 表示星期一，以此类推;
- `hour`: 小时，范围 [0, 23];
- `minute`: 分钟，范围 [0, 59];
- `second`: 秒，范围 [0, 59];

-- yday: 从当前年份 1 月 1 日开始的天数, 范围 [0, 365], 一般置位 0 未实现。

使用的方式可参考示例。

`RTC.deinit()`

重置 RTC 设备时间到 2015 年 1 月 1 日, 重新运行 RTC 设备。

`RTC.now()`

获取当前时间, 返回值为上述 `datetime` 时间元组格式。

## ● 示例

```
>>> from machine import RTC
>>> rtc = RTC()                      # 创建 RTC 设备对象
>>> rtc.init((2019,6,5,2,10,22,30,0)) # 设置初始化时间
>>> rtc.now()                        # 获取当前时间
(2019, 6, 5, 2, 10, 22, 40, 0)
>>> rtc.deinit()                    # 重置时间到2015年1月1日
>>> rtc.now()                        # 获取当前时间
(2015, 1, 1, 3, 0, 0, 1, 0)
```

更多内容可参考 [machine.RTC](#)。

## 4.14 machine.LCD

`machine.LCD` 类是 `machine` 模块下面的一个硬件类，用于对 LCD 的配置和控制，提供对 LCD 设备的操作方法。

### ● 构造函数

在 Rt-Thread MicroPython 中 LCD 对象的构造函数如下：

```
class machine.LCD()
```

在给定总线上构造一个 LCD 对象，无入参，初始化的对象取决于特定硬件，初始化方式可参考示例。

### ● 方法

```
LCD.light(value)
```

控制是否开启 LCD 背光，入参为 `True` 则打开 LCD 背光，入参为 `False` 则关闭 LCD 背光。

```
LCD.fill(color)
```

根据给定的颜色填充整个屏幕，支持多种颜色，可以传入的参数有：

```
WHITE BLACK BLUE BRED GRED GBLUE RED MAGENTA GREEN CYAN YELLOW BROWN BRRED GRAY  
GRAY175 GRAY151 GRAY240
```

详细的使用方法可参考示例。

```
LCD.pixel(x, y, color)
```

向指定的位置 `(x, y)` 画点，点的颜色为 `color` 指定的颜色，可指定的颜色和上一个功能相同。

注意：`(x, y)` 坐标不要超过实际范围，使用下面的方法对坐标进行操作时同样需要遵循此限制。

```
LCD.text(str, x, y, size)
```

在指定的位置 `(x, y)` 写入字符串，字符串由 `str` 指定，字体的大小由 `size` 指定，`size` 的大小可为 16, 24, 32。

```
LCD.line(x1, y1, x2, y2)
```

在 LCD 上画一条直线，起始地址为 `(x1, y1)`，终点为 `(x2, y2)`。

```
LCD.rectangle(x1, y1, x2, y2)
```

在 LCD 上画一个矩形，左上角的位置为 `(x1, y1)`，右下角的位置为 `(x2, y2)`。

```
LCD.circle(x1, y1, r)
```

在 LCD 上画一个圆形，圆心的位置为 `(x1, y1)`，半径长度为 `r`。

```
LCD.show_bmp(x, y, pathname)
```

在 LCD 指定位置上显示 32-bit bmp 格式的图片信息，注意显示 bmp 图片时，`(x, y)`

坐标是图片的左下角。

### ● 示例

```
from machine import LCD      # 从 machine 导入 LCD 类
lcd = LCD()                  # 创建一个 lcd 对象
lcd.light(False)              # 关闭背光
lcd.light(True)               # 打开背光
lcd.fill(lcd.BLACK)          # 将整个 LCD 填充为黑色#
lcd.fill(lcd.RED)            # 将整个 LCD 填充为红色# 将
lcd.fill(lcd.GRAY)           # 整个 LCD 填充为灰色# 将整
lcd.fill(lcd.WHITE)          # 个 LCD 填充为白色
lcd.pixel(50, 50, lcd.BLUE)  # 将 (50,50) 位置的像素填充为蓝色
lcd.text("hello Rt-Thread", 0, 0, 16)    # 在 (0, 0) 位置以 16 号字打印字符串
lcd.text("hello Rt-Thread", 0, 16, 24)     # 在 (0, 16) 位置以 24 号字打印字符串
lcd.text("hello Rt-Thread", 0, 48, 32)     # 在 (0, 48) 位置以 32 号字打印字符串
lcd.line(0, 50, 239, 50)        # 以起点 (0, 50) , 终点 (239, 50) 画一条线
lcd.line(0, 50, 239, 50)        # 以起点 (0, 50) , 终点 (239, 50) 画一条线
lcd.rectangle(100, 100, 200, 200) # 以左上角为 (100,100) , 右下角 (200,200) 画矩形
lcd.circle(150, 150, 80)        # 以圆心位置 (150,150) , 半径为 80 画圆
lcd.show_bmp(180, 50, "sun.bmp") # 以位置 (180,50) 为图片左下角坐标显示文件系统中的
                                # bmp 图片 "sun.bmp"
```

## 4.15 MicroPython

MicroPython 是访问和控制 MicroPython 内部的构件。

### ● 方法

`MicroPython.const(expr)`

用于表示表达式为常量，这样编译就可以将其优化。该函数应按照下述说明使用：

```
from MicroPython import const

CONST_X = const(123)
CONST_Y = const(2 * CONST_X + 1)
```

以此方式声明的常量仍可作为全局变量从其声明的模块外部访问。另一方面，若某常量以下划线开始，则会被隐藏，则该常量无法作为全局变量访问，且在执行时不会占据任何内存。

`const` 函数是由 MicroPython 解析器直接识别的，并作为 MicroPython 模块的一部分提供，因此可通过以上模式编写在 CPython 和 MicroPython 下运行的脚本。

`MicroPython.opt_level([level])`

若 `level` 给定，则该函数为后续脚本编译设置优化级别，并返回 `None`。否则返回当前的优化级别。优化级别控制以下编译特性：

-- Assertions 断言：在 0 级时，断言语句被启用并编译为字节码；1 级及更高级别的断言不会编译。

-- 内置 `debug` 变量：0 级时，该变量扩展为 `True`；1 级及以上时，该变量扩展为 `False`。

-- 源代码行号：在 0 级、1 级和 2 级时，源代码行号与字节码一起存储，以便异常可以报告它们发生的行号；3 级及更高级别的行号不会存储。默认的优化级别通常是 0 级。

`MicroPython.alloc_emergency_exception_buf(size)`

为紧急异常缓冲区（适宜大小为 100 字节）分配 RAM 的 `size` 字节。当正常的 RAM 分配失败时（如在中断处理程序中），缓冲区用来创建异常，因此在此情况下提供有用的回溯信息。

使用该函数的较好方法为将其置于主脚本的开始，然后紧急异常缓冲区将会为其后所有代码激活。

`MicroPython.mem_info([verbose])`

打印关于当前占用内存的信息。若给定 `verbose` 参数，则打印附加信息。

所打印的信息与实现相关，但当前包含存储栈和堆的使用量。在详细模式下会打印整个堆，并标明指示哪些已占用，哪些尚可用。

`MicroPython.qstr_info([verbose])`

打印关于当前 `interned` 字符串的信息。若给定 `verbose` 参数，则打印附加信息。所打

印的信息与实现相关，但当前包含 `interned` 字符串和 RAM 使用量。在详细模式下会打印所有 `RAM-interned` 字符串的名称。

#### `MicroPython.stack_use()`

返回一个代表使用中的当前堆数量。该数值的绝对值并没有什么用处，但是这一数值应用于计算在不同点上堆使用的不同。

#### `MicroPython.heap_lock()` `MicroPython.heap_unlock()`

锁定或解锁堆。当锁定时，无法进行内存分配，若试图执行任何堆分配，会引发 `MemoryError`。

这些函数可嵌套，`heap_lock()` 可在一行中调用多次，锁定深度将会增加，且 `heap_unlock()` 必须调用同样次数，以使堆重新可用。

如果 REPL 在堆处于锁定状态时变为活动状态，那么它将被强制解锁。

#### `MicroPython.kbd_intr(chr)`

设置会引发 `KeyboardInterrupt` 异常的字符。在脚本执行中，其默认值为 3，与 `Ctrl-C` 相对应。传递-1 给此函数将禁用的捕捉，传递 3 将恢复。

该函数可用于在传入的字符流中捕捉 `ctrl-c`（该字符流常用于 REPL），以防止该流被用于其他目的。

#### `MicroPython.schedule(func, arg)`

将执行的函数调度为“非常快速”。该函数传递 `arg` 值作为其唯一参数。“非常快速”意味着运行时间将尽其所能尽早执行该函数，假定运行尽可能高效，以下条件依然有效：

预定函数不会抢占另一预定函数。

预定函数总在“操作码之间”执行，也就意味着所有基本 Python 操作（例如添加一个列表）都确保为极小的。

给定端口可能会定义“临界区”，预定函数不在该区域中执行。函数可能在临界区内预定，但函数只在退出该区域后才会执行。临界区一个例子即抢占中断处理程序（一个 IRQ）。

该函数可用来从抢占 IRQ 中预定回调。这样的 IRQ 对 IRQ（例如，堆可能被锁定）中运行的代码进行了限制，并预定一个稍后会回调的函数能够接触这些限制。

注意：如果从抢占式 IRQ 中调用 `schedule()`，当不允许内存分配并且要传递给 `schedule()` 的回调是绑定方法时，则直接传递该方法将失败。这是因为创建对绑定方法的引用会导致内存分配。一种解决方案是在类构造函数中创建对该方法的引用，并将该引用传递给 `schedule()`。这里将详细讨论:[ref:reference documentation <isr\\_rules>](#)在“Creation of Python object 创建 Python 对象”章节中。有一个用来存放预定函数的堆栈，若堆栈已满，则 `schedule` 会引发 `RuntimeError`。

## 4.16 network

此模块提供网络驱动程序和路由配置。特定硬件的网络驱动程序在此模块中可用，用于配置硬件网络接口。然后，配置接口提供的网络服务可以通过 `usocket` 模块使用。

### ● 专用的网络类配置

下面具体的类实现了抽象网卡的接口，并提供了一种控制各种网络接口的方法。

```
class WLAN
```

该类为 WiFi 网络处理器提供一个驱动程序。使用示例：

```
import network
# enable station interface and connect to WiFi access point nic
= network.WLAN(network.STA_IF)
nic.active(True)
nic.connect('your-ssid', 'your-password')
# now use sockets as usual
```

### ● 构造函数

在 Rt-Thread MicroPython 中 `WLAN` 对象的构造函数如下：

```
class network.WLAN(interface_id)
```

创建一个 `WLAN` 网络接口对象。支持的接口是 `network.STA_IF` (STA 模式，可以连接到上游的 WiFi 热点上) 和 `network.AP_IF` (AP 模式，允许其他 WiFi 客户端连接到自身的热点)。下面方法的可用性取决于接口的类型。例如，只有 STA 接口可以使用 `WLAN.connect()` 方法连接到 AP 热点上。

### ● 方法

```
WLAN.active([is_active])
```

如果向该方法传入布尔数值，传入 `True` 则使能卡，传入 `False` 则禁止网卡。否则，如果不传入参数，则查询当前网卡的状态。

```
WLAN.connect(ssid, password)
```

使用指定的账号和密码链接指定的无线热点。

```
WLAN.disconnect()
```

从当前链接的无线网络中断开。

```
WLAN.scan()
```

扫描当前可以连接的无线网络。

只能在 STA 模式下进行扫描，使用元组列表的形式返回 WiFi 接入点的相关信息。

```
(ssid, bssid, channel, rssi, authmode, hidden)
```

**WLAN.status([param])**

返回当前无线连接的状态。

当调用该方法时没有附带参数，就会返回值描述当前网络连接的状态。如果还没有从热点连接中获得 IP 地址，此时的状态为 `STATION_IDLE`。如果已经从连接的无线网络中获得 IP 地址，此时的状态为 `STAT_GOT_IP`。

当调用该函数使用的参数为 `rssi` 时，则返回 `rssi` 的值，该函数目前只支持这一个参数。

**WLAN.isconnected()**

在 STA 模式时，如果已经连接到 WiFi 网络，并且获得了 IP 地址，则返回 `True`。如果处在 AP 模式，此时已经与客户端建立连接，则返回 `True`。其他情况下都返回 `False`。

**WLAN.ifconfig([(ip, subnet, gateway, dns)])**

获取或者设置网络接口的参数，IP 地址，子网掩码，网关，DNS 服务器。当调用该方法不附带参数时，该方法会返回一个包含四个元素的元组来描述上面的信息。想要设置上面的值，传入一个包含上述四个元素的元组，例如：

```
nic.ifconfig('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))  
WLAN.config('param')  
WLAN.config(param=value, ...)
```

获取或者设置一般网络接口参数，这些方法允许处理标准的 `ip` 配置之外的其他参数，如 `WLAN.ifconfig()` 函数处理的参数。这些参数包括特定网络和特定硬件的参数。对于参数的设置，应该使用关键字的语法，可以一次性设置多个参数。

当查询参数时，参数名称的引用应该为字符串，且每次只能查询一个参数。

```
# Set WiFi access point name (formally known as ESSID) and WiFi password  
ap.config(essid='My_AP', password="88888888")  
# Query params one by one  
print(ap.config('essid'))  
print(ap.config('channel'))
```

下面是目前支持的参数：

```
-- mac : MAC address (bytes)  
-- essid : WiFi access point name (string)  
-- channel : WiFi channel (integer)  
-- hidden : Whether ESSID is hidden (boolean)  
-- password : Access password (string)
```

### ● 示例一：STA 模式

```
import network
wlan = network.WLAN(network.STA_IF)
wlan.scan()
wlan.connect("rtthread", "0218888888")
wlan.isconnected()
```

### ● 示例二：AP 模式下

```
import network
ap = network.WLAN(network.AP_IF) ap.config(essid="hello_Rt-
Thread", password="88888888") ap.active(True)
ap.config("essid")
```

## 4.17 rtthread

rtthread 模块提供了与 Rt-Thread 操作系统相关的功能，如查看栈使用情况等。

### ● 方法

`rtthread.current_tid()`

返回当前线程的 id。

`rtthread.is_preempt_thread()`

返回是否是可抢占线程。

### ● 示例：

```
>>> import rtthread
>>>
>>> rtthread.is_preempt_thread()          # determine if it's a preemptible thread
True
>>> rtthread.current_tid()              # current thread id
268464956
>>>
```

## 5. OpenMV

- 图像处理API支持

- 1) [参考图像处理](#)
- 2) [参考OpenMV官方](#)

- UI-API支持

- 1) [API接口（支持C语言）](#)

- 示例

```
import lvgl as lv
import lvdrv

lvdrv.init() # 必须先调用

scr = lv.obj()
btn = lv.btn(scr)
btn.align(lv.scr_act(), lv.ALIGN.CENTER, 0, 0)
label = lv.label(btn)
label.set_text("Hello World!")
lv.scr_load(scr)
```

## 6. OpenMV IDE

### ● OpenMV IDE 概述

现在我们来谈谈 OpenMV IDE。OpenMV IDE 是您用来编程 OpenMV Cam 的工具。它具有一个由 QtCreator 支持的功能强大的文本编辑器，一个帧缓冲器查看器，直方图显示器，以及一个用于 OpenMV 调试输出的集成串行终端。

无论如何，IDE 是或多或少可以直接使用。我们在 IDE 中放入了许多东西，在 IDE 中您应该了解几件事情。

#### 1) 文件菜单和示例代码

在 OpenMV IDE 的文件菜单下，您可以使用所有的标准文本编辑器选项。新建，打开，保存，另存为，打印等。无论如何，当你创建一个新的文件或打开一个文件，它会出现在文本编辑器窗格中。请注意，您可以打开多个文件，您可以通过单击显示当前文件名称的组合框进行选择。OpenMV IDE 使用 QtCreator 作为文本编辑器的后端，实际上支持在多个窗口中打开多个文件，具有水平和垂直分割功能。然而，所有这一切都是隐藏的，以便明确哪些脚本将在单击运行按钮时运行，就像 Arduino IDE 一样。如果您想在不同的编辑器中编辑脚本，或者需要一次打开多个文件，只需在您选择的编辑器中打开文件并对其进行处理即可。

注意：一次只能在您的计算机上使用 OpenMV IDE 的一个实例。这样做是为了如果您尝试通过文件资源管理器打开文件，OpenMV IDE 的一个实例将打开该文件。这也使您可以使用拖放功能打开文件。

在文件菜单下，您会看到示例菜单，其中有大量的脚本展示了如何使用 OpenMV Cam 的不同功能。随着我们实现更多的功能，将会出现更多的例子。确保熟悉示例脚本，以便您可以有效地使用 OpenMV。

对于个人脚本存储，OpenMV IDE 会在您的文档文件夹中查找“OpenMV”目录。OpenMV IDE 将在文件 → 文件夹中显示“OpenMV”文件夹的内容。

最后，无论何时在 OpenMV IDE 中创建一个新文件或打开示例脚本，OpenMV IDE 都会创建一个不需要存储在磁盘上的文件副本，就像在 Arduino IDE 中一样。这让您快速测试出创意，而无需创建文件。

## 2) 文本编辑

OpenMV IDE 有一个由 QtCreator 后端支持的专业文本编辑器。您可以对所有打开的文件进行无限撤消和重做，空格可视化（对于 MicroPython 非常重要），字体大小的控制以及 QtCreator 查找和替换等。查找和替换功能正则表达式与查找过程中捕获文本的能力相匹配，并使用捕获的文本进行替换。此外，它也可以在更换时保存大小写。最后，查找和替换功能不仅在当前文件上起作用，而且还可以在文件夹中的所有文件或 OpenMV IDE 中的所有打开的文件上起作用。

除了漂亮的文本编辑环境之外，OpenMV IDE 还提供关于关键字的自动完成支持悬停工具提示。因此，例如在 OpenMV IDE 中用 python 输入 . 后，将会检测到您即将编写函数或方法名称，并会显示一个自动完成文本框。一旦你写了函数/方法的名字，它也将引导你写入参数。最后，如果将鼠标光标悬停在任何关键字上，OpenMV IDE 将在工具提示中显示该关键字的文档。这两个特性使编辑脚本变得愉快！

## 3) 连接到您的 OpenMV Cam

连接到您的 OpenMV 在上一个教程页面 Hardware Setup 有详细介绍。一旦解决来所有涉及编程固件等的启动问题，只需点击 OpenMV IDE 左下角的连接按钮即可连接到您的 OpenMV。

OpenMV IDE 智能地连接并自动过滤所有不是 OpenMV Cam 的串行端口。如果只有一个 OpenMV 连接到您的电脑，它会发现它，并立即连接。如果您有两个 OpenMV，它会询问您连接到哪个串行端口。请注意，OpenMV IDE 将记住您的选择，以便下次如果连接 OpenMV Cam 的串行端口，您想连接的那个串行端口已经被选中。

连接到您的 OpenMV Cam 的串行端口后，OpenMV IDE 将尝试确定与您的 OpenMV Cam 相关的计算机上的 USB 闪存驱动器。OpenMV IDE 会对 USB 闪存驱动器进行一些智能过滤，以便在可能的情况下自动连接到正确的 USB 闪存驱动器。然而，它可能无法自动确定正确的一个，并会要求你帮忙，如果不这样的话。与上面的串口一样，OpenMV IDE 将记住您的选择，因此下次连接时会自动突出显示您以前的选择。

注意：OpenMV IDE 需要将 OpenMV Cam 上的 USB 闪存与您的 OpenMV Cam 的虚拟串行端口相关联，以便它可以解析编译错误消息并轻松打开带有错误的文件。它也需要 Save open script to OpenMV Cam 命令的驱动信息。这就是说，OpenMV IDE 仍然可以在不知道 OpenMV Cam 的 USB 闪存驱动器的情况下运行，并且只会禁用依赖于此知识的功能。

最后，连接到 OpenMV Cam 后，连接按钮将被替换为断开按钮。点击断开连接按钮，断开您的 OpenMV。请注意，断开连接会停止 OpenMV Cam 上正在执行的任何脚本。您也可以直接从电脑上拔下 OpenMV Cam，而无需断开连接，而 OpenMV IDE 将检测到并自动断开您的 OpenMV Cam。如果您的 OpenMV 崩溃，OpenMV IDE 也会检测到这一点，并断开您的 OpenMV。

**注意：如果您的 OpenMV Cam 无响应，可能是在 Windows / Mac / Linux 上的串口驱动程序崩溃。**

在 Windows 上，您将注意到这是一个不可操作的 OpenMV IDE 进程，在任务管理器中没有

任何窗口。如果发生这种情况唯一的办法是重新启动计算机，因为杀不死的进程将阻止任何 OpenMV IDE 的新实例连接到任何 OpenMV 。这个问题的存在是因为自 Windows XP 以来一直存在的未修复 Windows bug 。

在 Mac / Linux 上，同样的问题可能会发生，但更难遇到。基本上，至少在 Windows 上发生的事情是，当 USB 虚拟串行端口驱动程序从 OpenMV Cam 崩溃变为无响应时，永远不会让 OpenMV IDE 的串行线程从内核函数调用中返回，这使得 OpenMV IDE 的串行线程不可用。无论如何，我们做了大量工作来确保它不会发生，但要注意它仍有发生的可能。

#### 4) 运行脚本

完成编辑代码并准备运行脚本后，只需单击 OpenMV IDE 左下角的绿色运行按钮即可。该脚本将被发送到您的 OpenMV 编译成 Python 字节码，并由您的 OpenMV 凸轮执行。

如果您的脚本中有任何错误，您的 OpenMV Cam 将发回编译错误显示在 terminal 中，OpenMV IDE 将自动解析查找错误。当 OpenMV IDE 检测到错误时，它会自动打开错误的文件，并突出显示错误的行，同时显示一个错误消息框。这个功能可以节省大量的时间修复错误。

无论如何，如果你想停止脚本，只需点击停止按钮（在脚本运行时替换运行按钮）。请注意，由于程序运行结束或编译错误，脚本可以自动停止。无论哪种情况，运行按钮都会再次出现。

**注意：**单击运行按钮时，OpenMV IDE 会自动扫描脚本中的导入，并复制 OpenMV Cam 中缺少的任何外部所需脚本。当您单击“运行”按钮时，OpenMV IDE 还将自动更新 OpenMV Cam 上任何过时的外部模块。OpenMV IDE 首先在您的个人“OpenMV”文档文件夹中查找外部模块，然后查找示例文件夹。OpenMV IDE 能够解析单个文件模块和目录模块。

#### 5) FrameBuffer 帧缓存查看器

OpenMV IDE 的特别之处在于集成的帧缓冲区查看器。这让你在处理你的代码的时候可以很容易地看到你的 OpenMV Cam 正在看什么。在 sensor.snapshot() 调用时，帧缓冲区查看器会显示以前 OpenMV Cam 的帧缓冲区中的内容。无论如何，我们稍后再谈。现在，下面是你需要了解的帧缓冲器查看器：

- ✓ OpenMV IDE 顶部右上角的 ``Record`` 按钮记录帧缓存器中的内容。使用它来快速制作您的 OpenMV Cam 所看到的视频。请注意，录制工作以 30 FPS 记录 OpenMV IDE 帧缓冲区中的任何内容。但是，取决于应用程序，帧缓冲区可能会更快或更慢地更新。无论如何，录制完成后，OpenMV IDE 将使用 FFMPEG 将录制代码转换为任何想要共享的文件格式。
- ✓ OpenMV IDE 顶部右上角的 ``Zoom`` 按钮控制帧缓存器的缩放。根据您的需要启用或禁用该功能。
- ✓ OpenMV IDE 顶部右上角的 ``Disable`` 按钮控制你的 OpenMV 是否会发送图像到 OpenMV IDE。基本上，您的 OpenMV 必须不断 JPEG 压缩图像流到 OpenMV IDE 。这会降低性能。所以，如果你想看看你的脚本在 OpenMV Cam 没有连接到你的电脑的情况下运行得有多快，只需点击 ``Disable`` 按钮即可。当帧缓冲区被禁用时，您将无法看到

您的OpenMV Cam 正在查看的内容，但您仍然可以在串行终端中看到来自 OpenMV Cam 的调试输出。

最后，您可以右键单击在帧缓冲区查看器中看到的任何图像，以将该图像保存到磁盘。此外，如果通过单击并拖动来选择帧缓冲区中的区域，则可以将该区域保存到磁盘。请注意，在尝试将帧缓冲区保存到磁盘之前，应该停止脚本。否则，你可能得不到你想要的确切图像。

要取消选择帧缓冲区中的某个区域，只需单击任意位置，而无需拖动即可删除选择。但是，取消选择时可能创建一个像素选择区域，所以尝试单击帧缓冲区中的空白区域。

#### 6) 直方图显示

OpenMV IDE 中的集成直方图显示主要是为了填充帧缓冲区查看器下的空白空间，并为您提供一些视觉效果。然而，对于得到关于房间中照明质量的反馈，确定颜色跟踪设置，以及通常只是给你关于 OpenMV Cam 所查看的图像质量的想法，这也是非常有用的。

您可以在直方图中选择四个不同的颜色空间。RGB，灰度，LAB 和 YUV。只有灰度和 LAB 在以编程方式控制您的 OpenMV 时是有用的。RGB 很直观。因为我们使用 JPEG 进行 JPEG 压缩，所以我们可能会加上 YUV。

无论如何，默认情况下直方图显示整个图像的信息。但是，如果通过单击并拖动帧缓冲区来选择一个区域，那么直方图将只显示该区域中的颜色分布。这种特性使得直方图显示在你需要在 `image.find_blobs()` 和 `image.binary()` 脚本中使用正确的灰度和 LAB 颜色通道设置时是很有用的。

最后，图像分辨率和你在图像上选择的边界框 ROI (x, y, w, h) 将显示在直方图的上方。

#### 7) 串行终端

要显示串行终端，请点击位于 OpenMV IDE 底部的串行终端按钮。串行终端内置在主窗口中，更易于使用。它只是与您的文本编辑窗口分开。

无论如何，由您创建的 OpenMV Cam 中的所有调试文本 `print` 都将显示在串行终端中。除此之外，没有什么可说的了。

请注意，串行终端或多或少会无限缓冲文本。它将在 RAM 中保留最后一百万行的文本。所以，你可以使用它来缓冲大量的调试输出。此外，如果您在 Windows / Linux 按 `ctrl+f` 或 Mac 上等效的快捷键，则可以搜索调试输出。最后，串行终端是足够智能的，如果你想查看以前的调试输出它非常好用，不会自动滚动。如果滚动到文本输出的底部，自动滚动将再次打开。

#### 8) 状态栏

在状态栏上，OpenMV IDE 将显示您的 OpenMV Cam 的固件版本，串行端口，驱动器和 FPS。固件版本标签实际上是一个按钮，如果您的 OpenMV 的固件已过时，您可以单击以更新您的 OpenMV。串行端口标签只显示您的 OpenMV 的串行端口，没有别的。Drive 标签是另一个按钮，您可以点击更改链接到 OpenMV Cam 的驱动器。最后，FPS 标签显示 OpenMV IDE 正在从您的 OpenMV Cam 中获取的 fps。

注意：OpenMV IDE 显示的 FPS 可能与 OpenMV Cam 的 FPS 不匹配。OpenMV IDE 上的 FPS 标签是从您的 OpenMV 获得的 FPS。但是，您的 OpenMV Cam 实际上可能比 OpenMV IDE 运行得更快，OpenMV IDE 只是从 OpenMV Cam 中抽取一些帧，而不是全部。无论如何，OpenMV IDE 的 FPS 永远不会比 OpenMV Cam 的 FPS 更快，但速度可能会更慢。

#### 9) 工具

您可以在 OpenMV IDE 的“工具”菜单下找到适用于 OpenMV Cam 的有用工具。特别是，Save open script to your OpenMV 和 CamReset OpenMV Cam 工具使用 OpenMV 开发应用程序时很有用。

- ✓ 使用OpenMV IDE 的 ``Configure OpenMV Cam Settings file`` 将允许你修改存储在您的OpenMV 中的 ``.ini`` 文件，您OpenMV 将在启动时针对特定硬件配置读取。
- ✓ ``Save open script to your OpenMV Cam`` 该命令将当前正在查看的任何脚本保存到您的OpenMV。此外，它还可以自动清除脚本中的空白和注释，从而占用更少的空间。一旦您认为您的程序已准备好在没有OpenMV IDE 的情况下进行部署，请使用此命令。请注意，该命令会将您的脚本保存在您的OpenMV Cam 的USB 闪存驱动器上。  
``main.py`` 是您的OpenMV Cam 将在完成启动后运行的脚本。
- ✓ ``Reset OpenMV Cam`` 该命令重置，然后断开您的OpenMV。如果您运行在OpenMV 上创建文件的脚本，您将需要使用此选项，因为在OpenMV 的USB 闪存驱动器安装之后，Windows / Mac / Linux 不会显示在您的OpenMV 上以编程方式创建的任何文件。

接下来，在 Tools 菜单下，您可以调用 boot-loader 来重新编程您的 OpenMV Cam。启动加载程序只能在您的 OpenMV Cam 的 OpenMV IDE 断开连接时调用。你可以给它一个二进制 .bin 文件或 .dfu 文件来重新编程你的 OpenMV 凸轮。引导加载程序功能仅适用于计划更改默认 OpenMV Cam 固件的高级用户。

#### 10) Open Terminal

打开终端功能允许您创建新的串行终端，使用 OpenMV IDE 远程调试未连接到计算机的 OpenMV Cam。开放终端功能也可以用来编程任何 MicroPython 开发板。

您可以使用“打开终端”功能打开连接到串行端口，tcp 端口或 udp 端口的终端。请注意，串行端口可以是用于无线连接的蓝牙端口。

#### 11) 机器视觉

机器视觉子菜单包含许多用于 OpenMV 的机器视觉工具。例如，您可以使用颜色阈值编辑器来获取最佳的颜色跟踪 `image.find_blobs()` 阈值。我们会定期提供新的机器视觉工具，让您的生活更轻松。

#### 12) 视频工具

如果您需要压缩由 OpenMV Cam 生成的 .gif 文件或将 .jpeg 或 ImageWriter 的 .bin 视频文件转换为 .mp4，则可以使用转换视频文件操作来执行此操作。或者，如果您只想播放这些视频，则也可以使用播放视频文件操作执行此操作。

请注意，在播放视频之前，请先将视频文件从 OpenMV Cam 的闪存驱动器复制到电脑上，因为OpenMV Cam 上的磁盘 I / O 通过 USB 是缓慢的。最后，FFMPEG 用于提供转换和视频播放支持，可用于任何您喜欢的非 OpenMV Cam 活动。FFMPEG可以转换/播放大量的文件格式。

要将视频文件转换为一组图片，请选择视频文件作为源，并使目标为“% 07d.bmp” / “% 07d.jpg” / 等文件名。

FFMPEG 理解 ``printf()`` 就像带有图像文件格式扩展名的格式语句，意味着它应该将视频文件分解为目标格式的静止图像。

- ✓ 要将一系列静止图像转换为视频，请将源文件名设置为“% 7d.bmp” / “% 7d.jpg” / 等。其中目录中的所有图像都有一个类似的名称，如 ``1.bmp``，``2.bmp`` 等。
- ✓ FFMPEG 理解 ``printf()`` 就像带有图像文件格式扩展名的格式语句一样，意味着应该将这些图像文件一起加入视频中。
- ✓ 要将 `ImageWriter` 文件转换为任何其他视频格式，请选择文件作为源和目标，使其成为您想要的任何文件格式。
- ✓ 要将任何格式的视频文件转换为 `ImageWriter` 文件，请选择要转换为源的视频文件，并将目标设置为 ``.bin`` 文件。
- ✓ 然后，FFMPEG 将视频分解为 JPG，OpenMV IDE 将使用 `ImageWriter` 文件格式将这些 JPG 转换为 RAW 灰度或 RGB565 帧保存到 ``.bin`` 文件。
- ✓ 为了优化你的OpenMV Cam 为 web 保存的 ``.gif`` 文件，将源文件设置为 ``.gif``，将目标文件设置为另一个 ``.gif``。
- ✓ 要将OpenMV Cam 保存的MJPEG 文件转换为另一种格式，请将MJPEG 文件设置为源，将另一种格式（如 ``.mp4``）设置为目标。

### 13) Option 选项

在工具菜单下（或关于 Mac 菜单），您可以访问 OpenMV IDE 的选项对话框。你可以配置编辑器的字体，大小，缩放，tab，自动清理空白，列边距，等等。

更多内容参考 [OpenMV 中文官网](#)。

### ● Bridge

由于 K210 芯片没有 USB 接口，连接 OpenMV IDE 需要使用额外的转接桥芯片，SDK 中提供的是基于STM32F411 的固件。

#### 1) 硬件连接(STM32 Pin函数功能示意)

```
PB12 - SPI_CS  
PB13 - SPI_SCK  
PB14 - SPI_MISO  
PB15 - SPI_MOSI  
PA8  - SPI_INT  
PA9  - SPI_READY_PIN  
PA10 - DATA_READY_PIN
```

#### 2) 编译

使用Rt-Thread ENV进入目录 `src/bridge/stm32f411`，执行下列指令更新软件包并编

```
pkgs --update  
scons
```

## 7. Rt-Thread MicroPython IDE

### ● 概述

Rt-Thread 为广大开发者提供了 VSCode 最好用的 MicroPython 插件来帮助大家使用 MicroPython 来开发应用程序。该插件为 MicroPython 开发提供了功能强大的开发环境，主要特性如下：

- ✓ 便捷的开发板连接方式（串口、网络、USB）
- ✓ 支持基于 MicroPython 的代码智能补全与语法检查
- ✓ 支持 MicroPython REPL 交互环境
- ✓ 提供丰富的代码示例与 demo 程序
- ✓ 提供工程同步功能
- ✓ 支持下载单个文件或文件夹至开发板
- ✓ 支持在内存中快速运行代码文件功能
- ✓ 支持运行代码片段功能
- ✓ 支持多款主流 MicroPython 开发板
- ✓ 将支持 Windows、Ubuntu、Mac 操作系统

可通过查看如下文档进一步了解并使用 Rt-Thread MicroPython IDE：

[Rt-Thread MicroPython Develop Environment](#)

## 8. OTA

### ● Bootloader

boot 的烧录使用嘉楠官方提供的工具 kflash 烧录

1) kflash 安装

-pip 安装

```
sudo pip3 install kflash
```

-源码下载

```
wget https://raw.githubusercontent.com/kendryte/kflash.py/master/kflash.py
```

2) 烧录

根据不同的硬件平台具有不同的烧录命令如下所示：

```
# Linux or macOS
# Using pip
kflash -B dan boot.bin
kflash -B dan -t boot.bin # Open a Serial Terminal After Finish
# Using source code
python3 kflash.py -B dan boot.bin
python3 kflash.py -B dan -t boot.bin # Open a Serial Terminal After Finish

# Windows CMD or PowerShell
# Using pip
kflash -B dan boot.bin
kflash -B dan -t boot.bin # Open a Serial Terminal After Finish
kflash -B dan -n -t boot.bin # Open a Serial Terminal After Finish, do not use ANSI
    colors
# Using source code
python kflash.py -B dan boot.bin
python kflash.py -B dan -t boot.bin # Open a Serial Terminal After Finish
python kflash.py -B dan -n -t boot.bin # Open a Serial Terminal After Finish, do not
    use ANSI colors

# Windows Subsystem for Linux
# Using pip
sudo kflash -B dan -p /dev/ttyS13 boot.bin # ttyS13 Stands for the COM13 in Device
    Manager
sudo kflash -B dan -p /dev/ttyS13 -t boot.bin # Open a Serial Terminal After Finish
# Using source code
sudo python3 kflash.py -B dan -p /dev/ttyS13 boot.bin # ttyS13 Stands for the COM13
    in Device Manager
sudo python3 kflash.py -B dan -p /dev/ttyS13 -t boot.bin # Open a Serial Terminal
    After Finish
```

当 bootloader 烧录完成并打开调试串口终端可以看见以下输出：

```
heap: [0x80587d99 - 0x80600000]
initialize rti_board_start: 0 done
initialize io_config_init: 2 done
initialize rt_hw_pin_init: 0 done

\ | /
- RT -      Thread Operating System
/ | \    4.0.3 build Dec 28 2020
2006 - 2020 Copyright by rt-thread team
do components initialization.
initialize rti_board_end: 0 done
initialize dfs_init: 0 done
initialize ulog_init: 0 done
initialize ulog_console_backend_init: 0 done
initialize k210_hwtimer_init: 0 done
initialize libc_system_init: 0 done
initialize cplusplus_system_init: 0 done
initialize load_application[D/FAL] (fal_flash_init: 63) Flash device |
    spi_nor | addr: 0x00000000 | len: 0x01000000 | blk_size: 0
    x00001000 | initialized finish.
[I/FAL] ===== FAL partition table =====
[I/FAL] | name      | flash_dev | offset   | length   |
[I/FAL] -----
[I/FAL] | bl        | spi_nor   | 0x00000000 | 0x00100000 |
[I/FAL] | app       | spi_nor   | 0x00100000 | 0x00300000 |
[I/FAL] | download  | spi_nor   | 0x00400000 | 0x00200000 |
[I/FAL] | fs        | spi_nor   | 0x00600000 | 0x00200000 |
[I/FAL] =====
[I/FAL] RT-Thread Flash Abstraction Layer (V0.5.0) initialize success.
[I/OTA] RT-Thread OTA package(V0.2.3) initialize success.
```

```
[E/OTA] (get_fw_hdr: 150) Get firmware header occur CRC32(calc.crc:
    ffffffc933e758 != hdr.info_crc32: ffffffaaab66c3) error on 'download'
    partition!
[142] I/OTA: check upgrade...
[E/OTA] (get_fw_hdr: 150) Get firmware header occur CRC32(calc.crc:
    ffffffc933e758 != hdr.info_crc32: ffffffaaab66c3) error on 'download'
    partition!
[E/OTA] (rt_ota_check_upgrade: 464) Get OTA download partition firmware header
    failed!
[168] I/OTA: No firmware upgrade!
[E/OTA] (get_fw_hdr: 150) Get firmware header occur CRC32(calc.crc: 7e61bad8 != hdr
    .info_crc32: 773d95aa) error on 'app' partition!
[185] E/OTA: App verify failed! Need to recovery factory firmware.
: 0 done
initialize finsh_system_init: 0 done
msh />world

msh />
```

## ● 定制 OTA 固件

### 1) 添加下载器功能

本小节介绍如何将下载器功能添加到 app 固件中。

添加该功能需要使用 env 工具，本次下载的软件包在 iot 类别中，需要按照如下步骤操作：

--下载 ota\_downloader 软件包，选中Ymodem 功能。

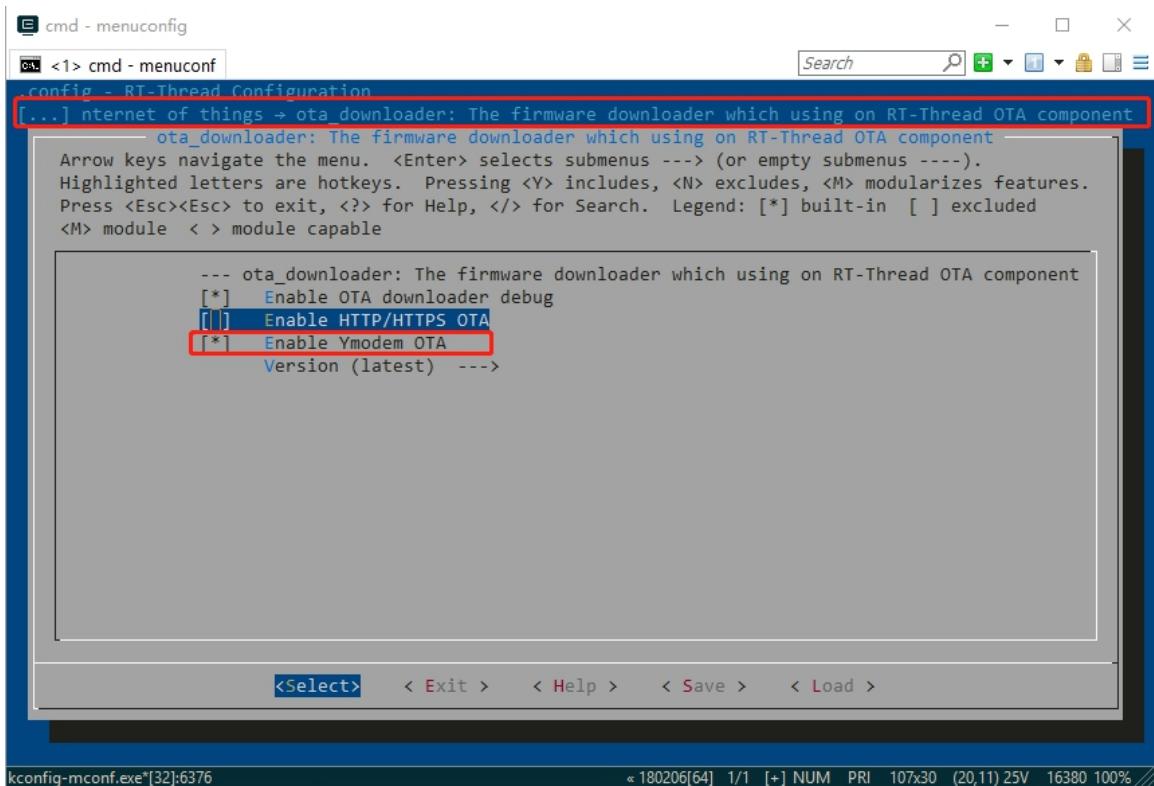


图 7.1: ota\_downloader 软件包配置

--根据需要选配 HTTP 更新。

--确保 fal 驱动已正确开启，SDK 中 fal 的相关配置与 bootloader 使用同一套头文件编译，所以不要随意修改fal\_cfg.h以免出现分区异常

--重新编译固件即可。

### 2) 打包发布

OTA 固件是在普通固件的基础上进行了二次打包，打包之后支持加密，压缩等功能，打包方法如下：

--双击打开tools\ota\_packager\rt\_ota\_packaging\_tool.exe程序，使用OTA 固件打包工具将编译出的rtthread.bin文件打包成可被升级的rtthread.rbl文件，如下图所示：



图 7.2: 打包固件

固件打包器提供三种固件压缩方式：fastlz、quicklz 和 gzip，一种固件加密方式 AES256。可以根据实际需求选择合适的加密压缩方式。

SDK 提供的 bootloader 对应的加密密钥为：

加密密钥：

qnbGbxjnxtwaPqjuntaZyQvZepywimdg

IV:

usguJdwWUugadheA

### 3) bootloader 下通过 ymodem 更新固件

使用 Ymodem 协议升级固件时，推荐使用 Xshell 终端。

在 msh 命令行中输入 ymodem\_ota 命令后，点击鼠标右键，然后在菜单栏找到用 YMDEMOD 发送选项发送文件，如下图所示：

--选择 Ymodem 方式发送升级固件：

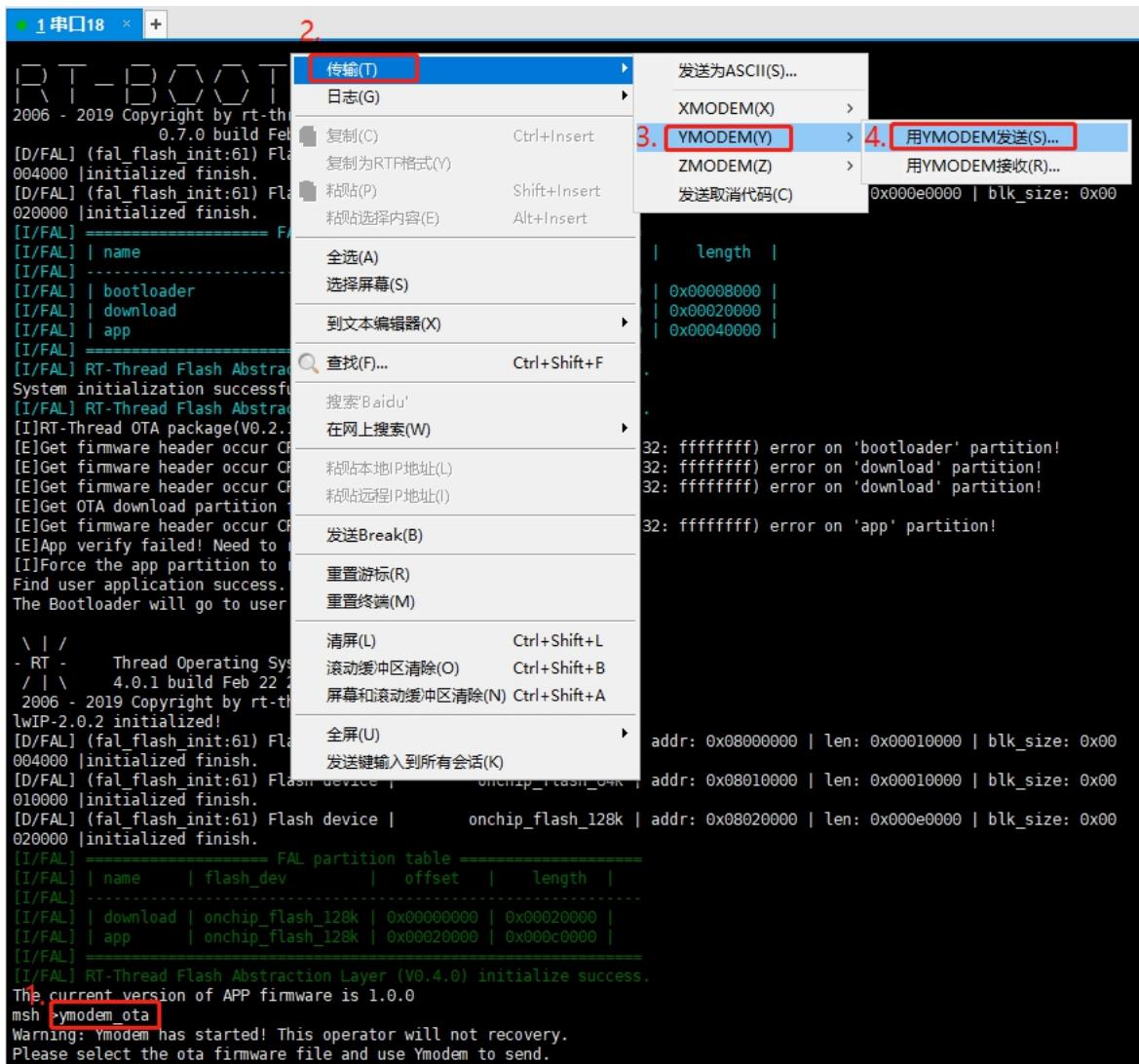


图 7.3: Ymodem 发送固件

--选中之前 OTA 固件打包工具生成的rtthread.rbl文件。

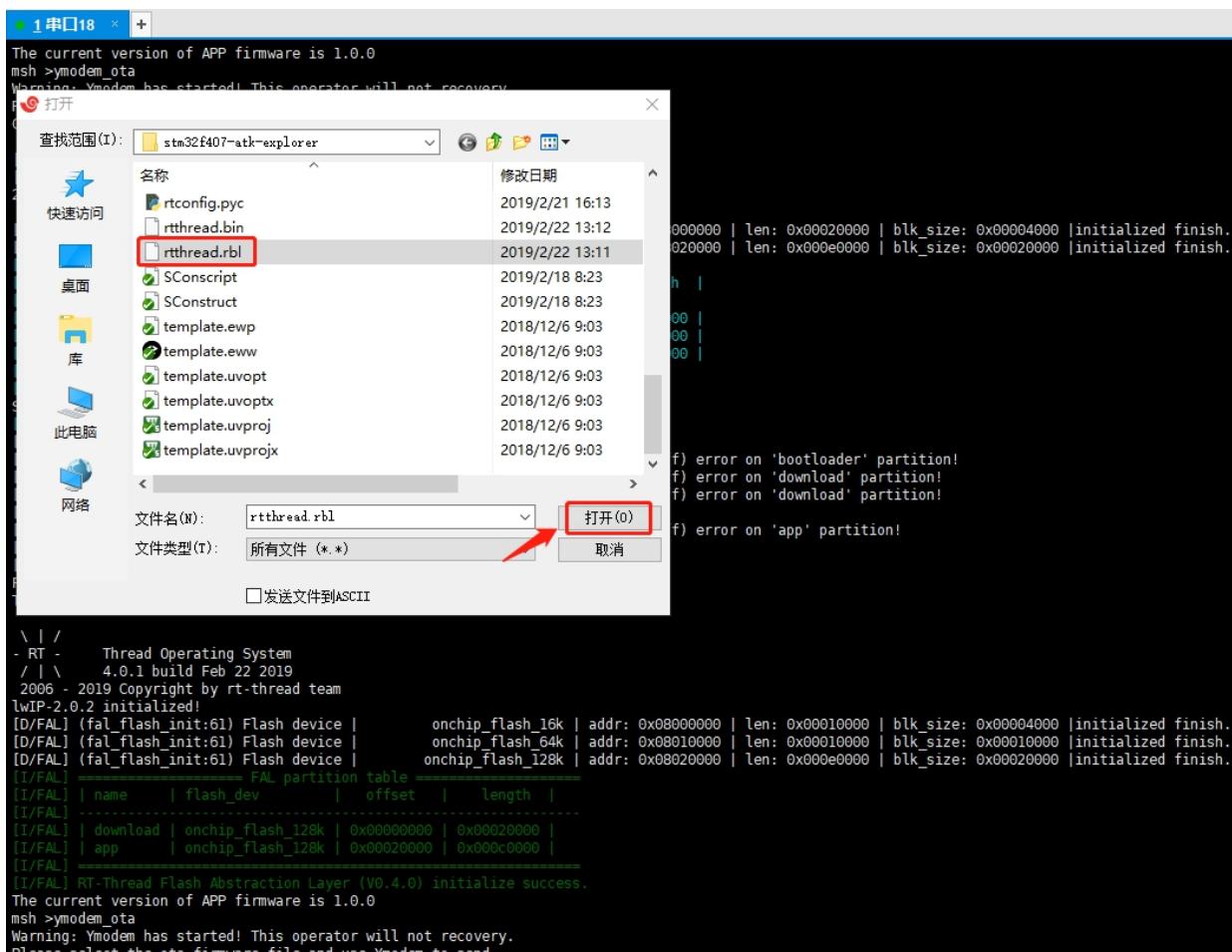


图 7.4: Ymodem 选择文件

--接下来升级固件就会通过 Ymodem 的方式被下载到 download 分区。固件被下载到 download 分区后，系统会自动重启，执行 OTA 升级。

#### 4) 在 app 下 OTA 升级固件

在 app 下 Ymodem 方式升级固件与 bootloader 相同，HTTP 升级固件命令为：

```
http_ota [url]
```

#### 5) 自定义的 OTA 下载器

OTA 下载器工作原理为将 rbl 文件内容写入 download 分区。

--初始化 FAL 组件

--获取download分区句柄

--擦除download分区

--写入 rbl 文件数据

## 接口示例

### 初始化 FAL 组件

```
/**  
 * FAL (Flash Abstraction Layer) initialization.  
 * It will initialize all flash device and all flash partition.  
 *  
 * @return >= 0: partitions total number  
 */  
int fal_init(void);
```

### 获取分区句柄

```
/**  
 * find the partition by name  
 *  
 * @param name partition name  
 *  
 * @return != NULL: partition  
 *          NULL: not found  
 */  
const struct fal_partition *fal_partition_find(const char *name);
```

### 擦除分区

```
/**  
 * erase partition data  
 *  
 * @param part partition  
 * @param addr relative address for partition  
 * @param size erase size  
 *  
 * @return >= 0: successful erased data size  
 *         -1: error  
 */  
int fal_partition_erase(const struct fal_partition *part, uint32_t addr, size_t size  
 );  
  
/**  
 * erase partition all data  
 *  
 * @param part partition  
 *  
 * @return >= 0: successful erased data size  
 *         -1: error
```

```
/*
int fal_partition_erase_all(const struct fal_partition *part);
```

## 写入分区

```
/**
 * write data to partition
 *
 * @param part partition
 * @param addr relative address for partition
 * @param buf write buffer
 * @param size write size
 *
 * @return >= 0: successful write data size
 *         -1: error
 */
int fal_partition_write(const struct fal_partition *part, uint32_t addr, const
                       uint8_t *buf, size_t size);
```

## ymodem OTA 下载器示例

```
/*
 * Copyright (c) 2006-2018, RT-Thread Development Team
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Change Logs:
 * Date           Author    Notes
 * 2018-01-30     armink   the first version
 * 2018-08-27     Murphy    update log
 */

#include <rtthread.h>
#include <stdio.h>
#include <stdbool.h>
#include <finsh.h>
#include <fal.h>
#include <ymodem.h>

#define DBG_ENABLE
#define DBG_SECTION_NAME          "ymodem"
#ifndef OTA_DOWNLOADER_DEBUG
#define DBG_LEVEL                DBG_LOG
#else
#define DBG_LEVEL                DBG_INFO
#endif
#define DBG_COLOR
```

```
#include <rtdbg.h>

#ifndef PKG_USING_YMODEM_OTA

#define DEFAULT_DOWNLOAD_PART "download"

static char* recv_partition = DEFAULT_DOWNLOAD_PART;
static size_t update_file_total_size, update_file_cur_size;
static const struct fal_partition * dl_part = RT_NULL;

static enum rym_code ymodem_on_begin(struct rym_ctx *ctx, rt_uint8_t *buf, rt_size_t
len)
{
    char *file_name, *file_size;

    /* calculate and store file size */
    file_name = (char *)&buf[0];
    file_size = (char *)&buf[rt_strlen(file_name) + 1];
    update_file_total_size = atol(file_size);
    rt_kprintf("Ymodem file_size:%d\n", update_file_total_size);

    update_file_cur_size = 0;

    /* Get download partition information and erase download partition data */
    if ((dl_part = fal_partition_find(recv_partition)) == RT_NULL)
    {
        LOG_E("Firmware download failed! Partition (%s) find error!", recv_partition
);
        return RYM_CODE_CAN;
    }

    if (update_file_total_size > dl_part->len)
    {
        LOG_E("Firmware is too large! File size (%d), '%s' partition size (%d)",
            update_file_total_size, recv_partition, dl_part->len);
        return RYM_CODE_CAN;
    }

    LOG_I("Start erase. Size (%d)", update_file_total_size);

    /* erase DL section */
    if (fal_partition_erase(dl_part, 0, update_file_total_size) < 0)
    {
        LOG_E("Firmware download failed! Partition (%s) erase error!", dl_part->name
);
        return RYM_CODE_CAN;
    }

    return RYM_CODE_ACK;
}
```

```
}
```

```
static enum rym_code ymodem_on_data(struct rym_ctx *ctx, rt_uint8_t *buf, rt_size_t len)
{
    /* write data of application to DL partition */
    if (fal_partition_write(dl_part, update_file_cur_size, buf, len) < 0)
    {
        LOG_E("Firmware download failed! Partition (%s) write data error!", dl_part->name);
        return RYM_CODE_CAN;
    }

    update_file_cur_size += len;

    return RYM_CODE_ACK;
}

void ymodem_ota(uint8_t argc, char **argv)
{
    struct rym_ctx rctx;

    if (argc < 2)
    {
        recv_partition = DEFAULT_DOWNLOAD_PART;
        rt_kprintf("Default save firmware on download partition.\n");
    }
    else
    {
        const char *operator = argv[1];
        if (!strcmp(operator, "-p")) {
            if (argc < 3) {
                rt_kprintf("Usage: ymodem_ota -p <partiton name>.\n");
                return;
            } else {
                /* change default partition to save firmware */
                recv_partition = argv[2];
            }
        }else{
            rt_kprintf("Usage: ymodem_ota -p <partiton name>.\n");
            return;
        }
    }
}

rt_kprintf("Warning: Ymodem has started! This operator will not recovery.\n");
rt_kprintf("Please select the ota firmware file and use Ymodem to send.\n");

if (!rym_recv_on_device(&rctx, rt_console_get_device(), RT_DEVICE_OFLAG_RDWR |
    RT_DEVICE_FLAG_INT_RX,
```

```
        ymodem_on_begin, ymodem_on_data, NULL,
        RT_TICK_PER_SECOND))
{
    rt_kprintf("Download firmware to flash success.\n");
    rt_kprintf("System now will restart...\r\n");

    /* wait some time for terminal response finish */
    rt_thread_delay(rt_tick_from_millisecond(200));

    /* Reset the device, Start new firmware */
    extern void rt_hw_cpu_reset(void);
    rt_hw_cpu_reset();
    /* wait some time for terminal response finish */
    rt_thread_delay(rt_tick_from_millisecond(200));
}

else
{
    /* wait some time for terminal response finish */
    rt_thread_delay(RT_TICK_PER_SECOND);
    rt_kprintf("Update firmware fail.\n");
}

return;
}
/***
 * msh />ymodem_ota
*/
MSH_CMD_EXPORT(ymodem_ota, Use Y-MODEM to download the firmware);

#endif /* PKG_USING_YMODEM_OTA */
```