

2024

RT1021-144Pin-MicroPython 固件接口说明

# 目录

[目录 1](#_Toc183854841)

[1. RT1021-144Pin-MicroPython核心板简要说明 3](#_Toc183854842)

[1.1. RT1021-144Pin-MicroPython核心板连接Thonny启动REPL控制台 3](#_Toc183854843)

[1.2. 使用Thonny保存源码到板载FLASH中 5](#_Toc183854844)

[1.3. RT1021-144Pin-MicroPython启动顺序说明 7](#_Toc183854845)

[1.4. 如何脱机运行自己的Python文件 8](#_Toc183854846)

[2. MicroPython固件接口说明 10](#_Toc183854847)

[2.1. machine基础类 10](#_Toc183854848)

[2.1.1. Pin子模块 10](#_Toc183854849)

[2.1.2. ADC子模块 10](#_Toc183854850)

[2.1.3. UART子模块 11](#_Toc183854851)

[2.1.4. PWM子模块 12](#_Toc183854852)

[2.2. NXP支持的smartcar模块 12](#_Toc183854853)

[2.2.1. ADC\_Group子模块 12](#_Toc183854854)

[2.2.2. encoder子模块 13](#_Toc183854855)

[2.2.3. Ticker子模块 14](#_Toc183854856)

[2.3. NXP支持的display模块 16](#_Toc183854857)

[2.4. 逐飞科技支持的seekfree模块 17](#_Toc183854858)

[2.4.1. IMU 660/963 RA子模块 17](#_Toc183854859)

[2.4.2. KEY\_HANDLER子模块 18](#_Toc183854860)

[2.4.3. MOTOR\_CONTROLLER子模块 19](#_Toc183854861)

[2.4.4. BLDC\_CONTROLLER子模块 19](#_Toc183854862)

[2.4.5. WIRELESS\_UART子模块 19](#_Toc183854863)

[2.4.6. TSL1401子模块 20](#_Toc183854864)

[2.4.7. DL1B子模块 22](#_Toc183854865)

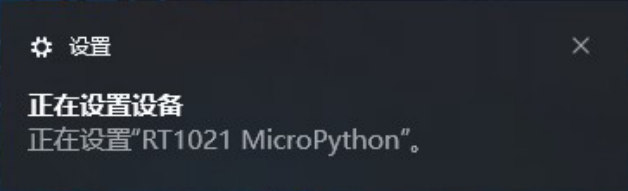
[2.5. 多Python源码文件的包含与调用 22](#_Toc183854866)

[3. 文档版本 24](#_Toc183854867)

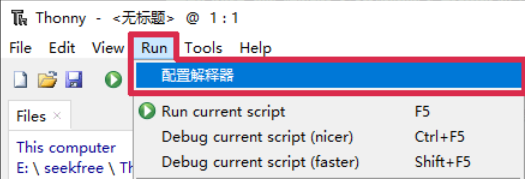
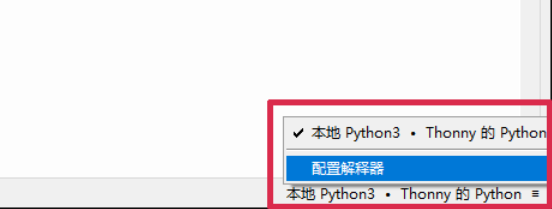
# RT1021-144Pin-MicroPython核心板简要说明

## RT1021-144Pin-MicroPython核心板连接Thonny启动REPL控制台

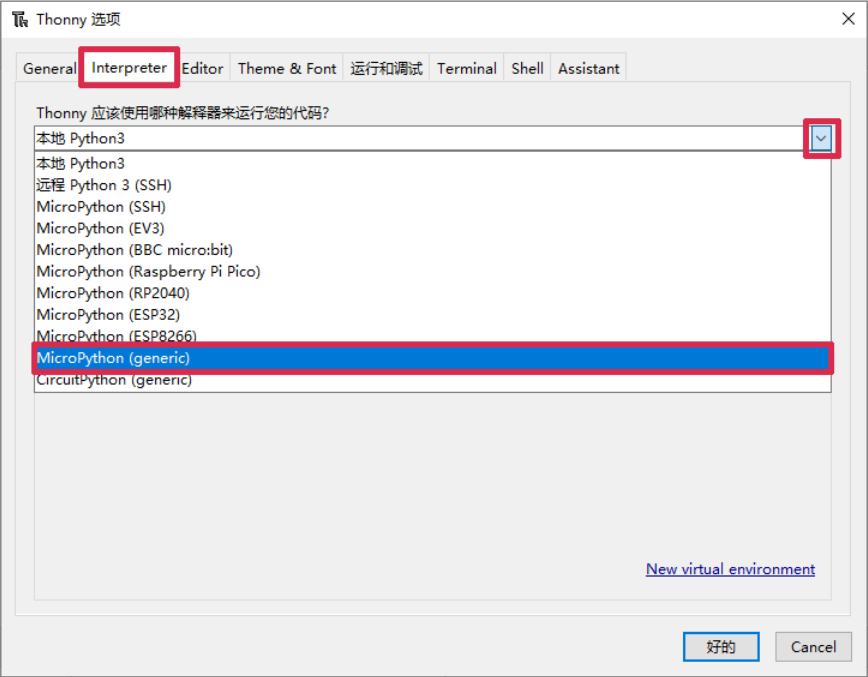
首先将RT1021核心板使用Type-C线连接PC，核心板中已默认烧录MicroPython固件，使用USB-CDC虚拟串口控制，因此在PC上会识别到一个串口设备：



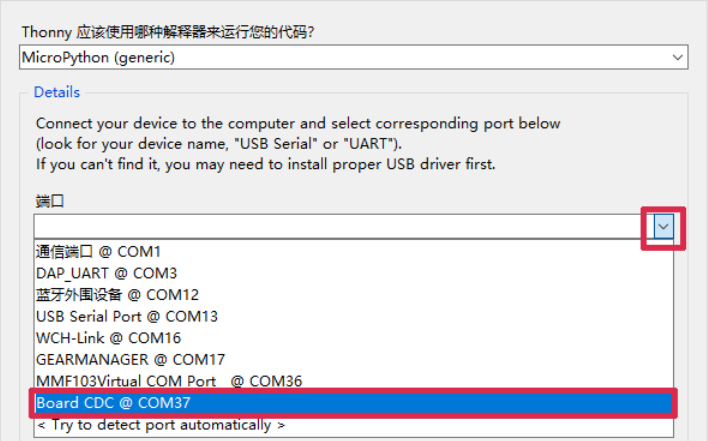
打开Thonny，打开菜单栏的 “运行->解释器配置” 选项，或者点击窗口右下角的解释器切换按键，打开对应的Thonny选项界面：

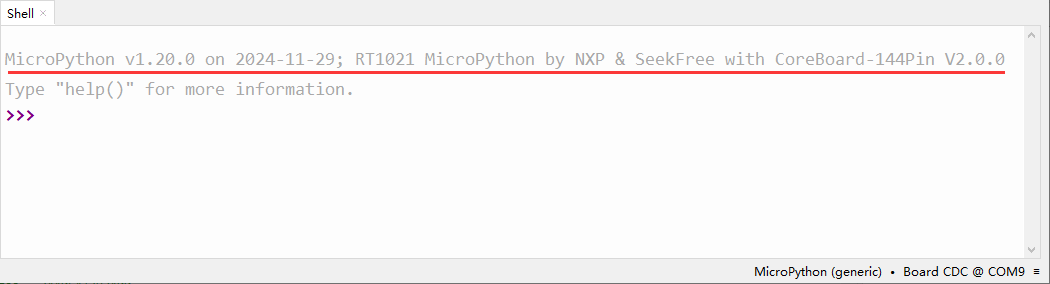
在Interpreter选项卡，选择MicroPython(generic)解释器：



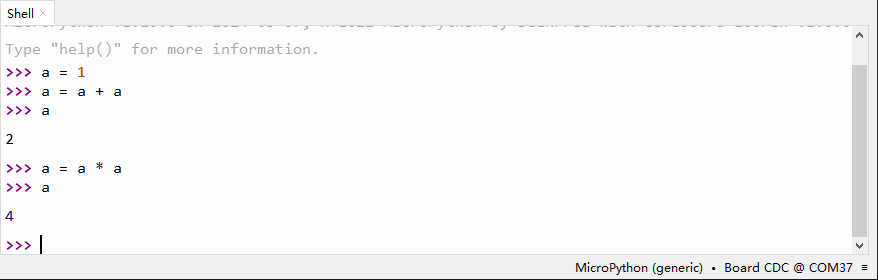
随后并找到核心板对应的COM口（此处名称可能会有差异）设置好，并确认保存：



正确连接并选择正确的COM口连接后，Thonny的Shell会输出如下信息：



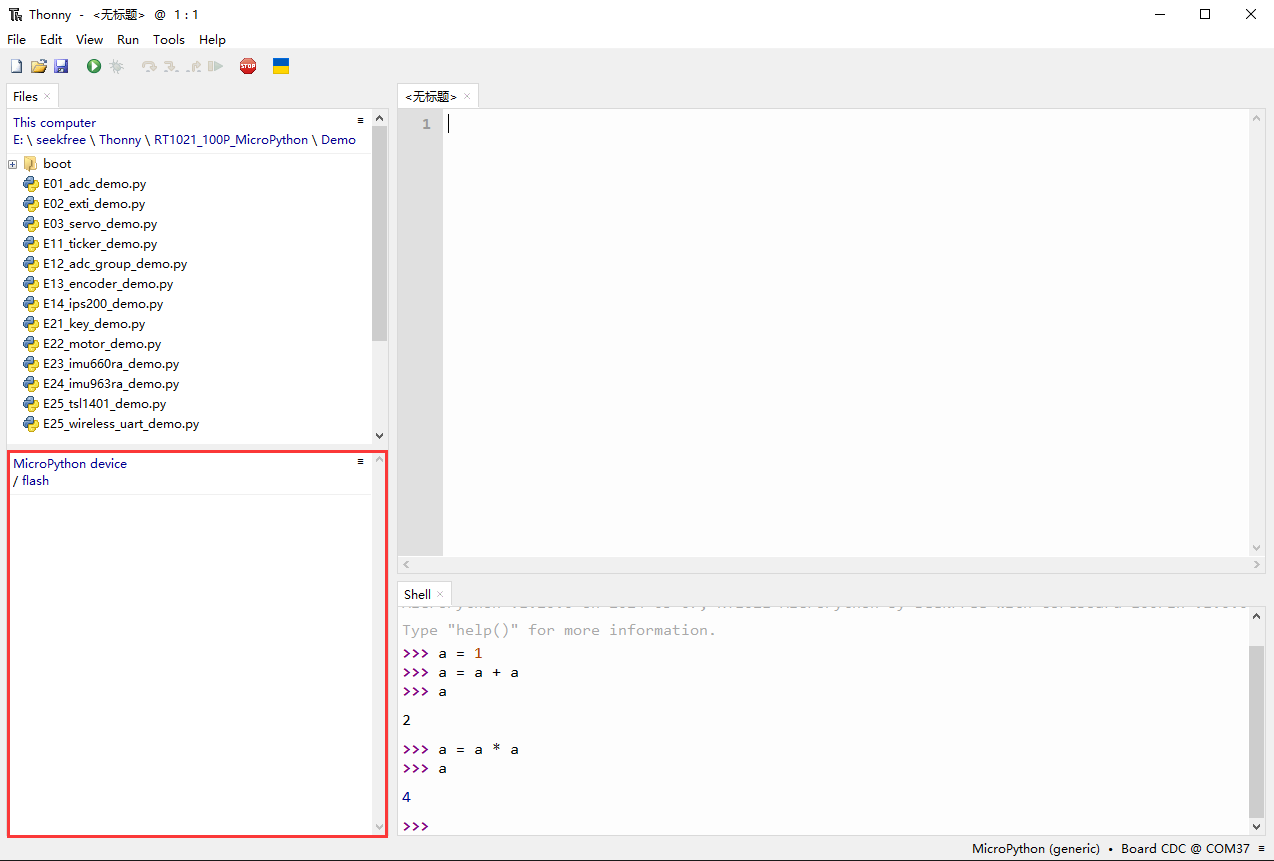
可以简单输入Python语句测试：



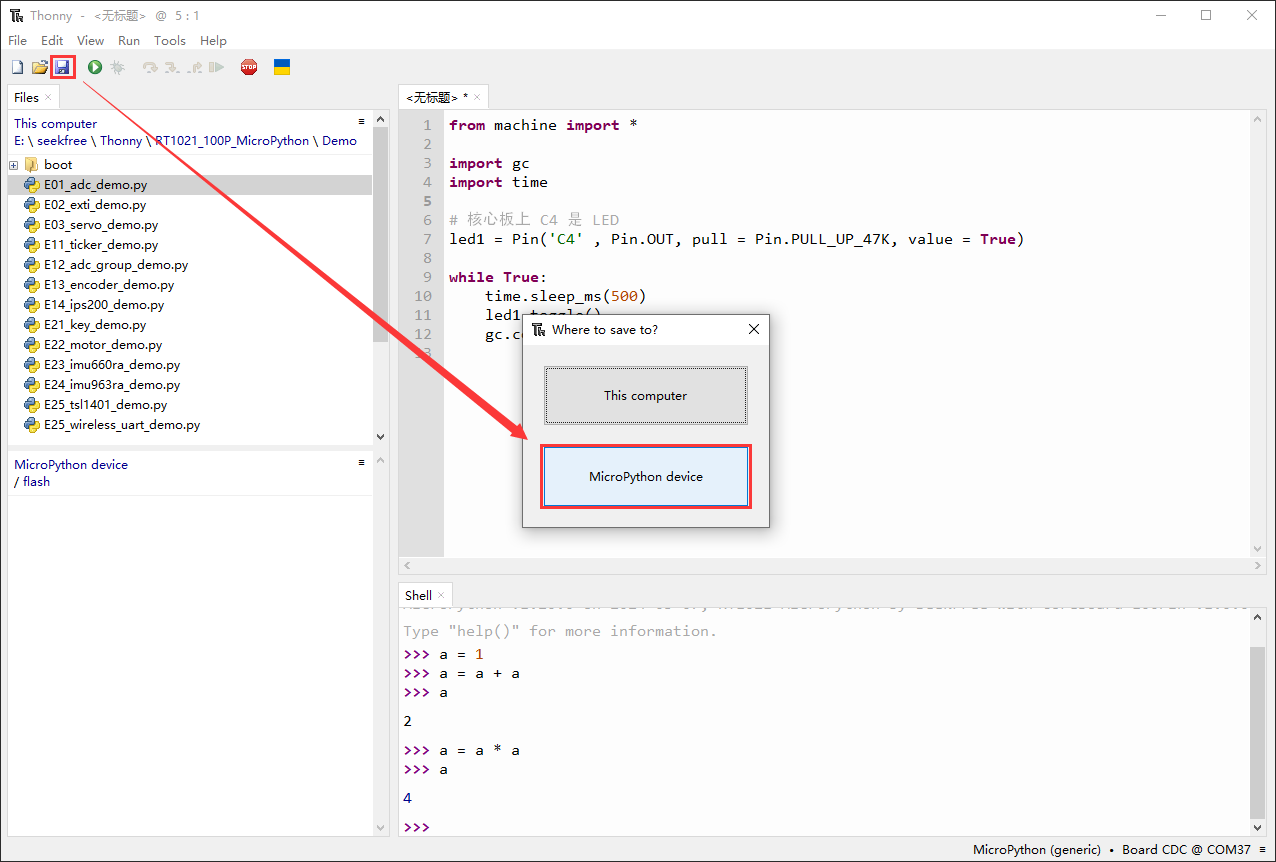
接下来便可以使用Thonny进行Python的编辑与调试了。

## 使用Thonny保存源码到板载FLASH中

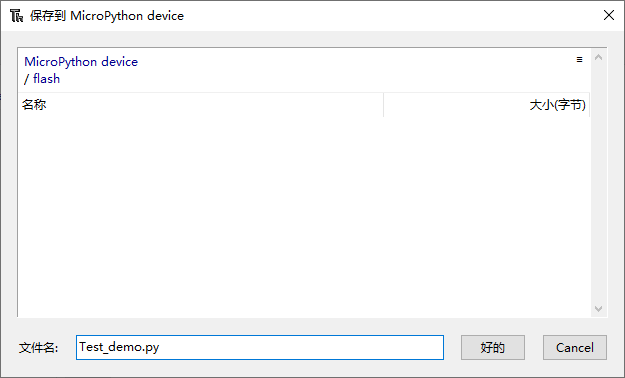
连接到MicroPython目标板后，Thonny会显示出当前设备的Flash中的文件：



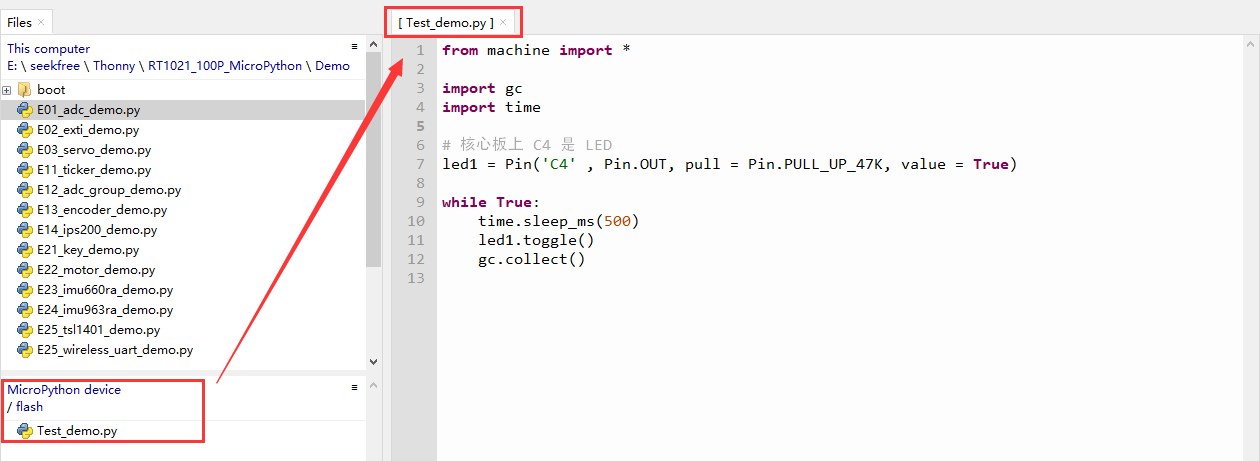
自己新建的Python文件可以通过Thonny保存到MicroPython板的Flash中：



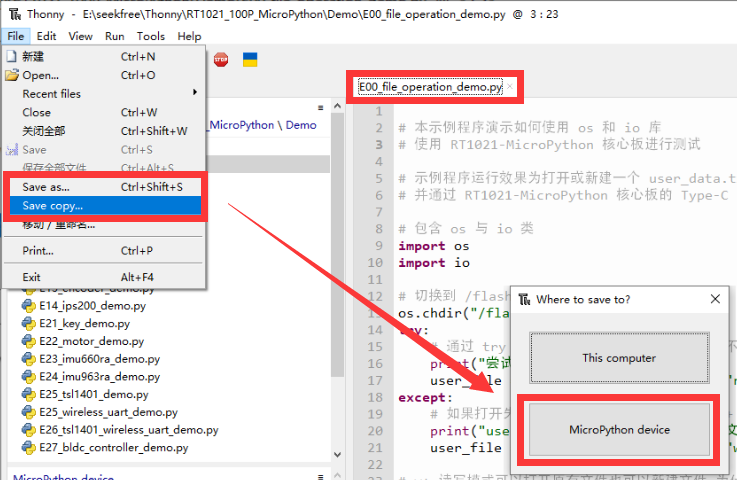
在弹出文件列表窗口命名好文件并确认即可，也可以在这个界面选中文件然后删除



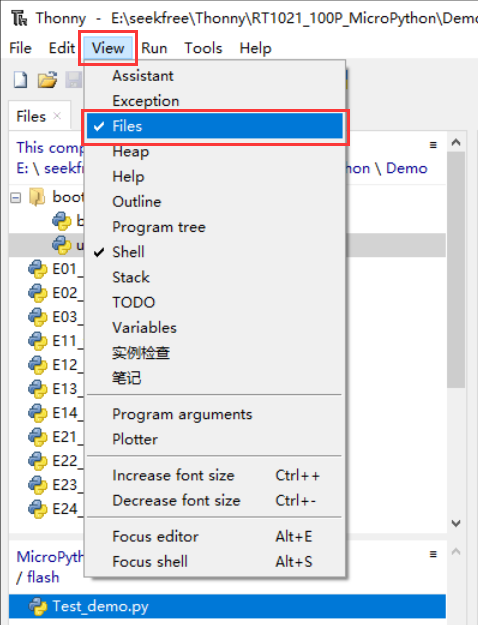
保存完成后，MicroPython device文件显示栏会显示设备中的文件，打开设备中的文件时，文件名会被中括号标识：



如果想要将本机文件复制或者保存到MicroPython板的flash中，则可以将文件另存为，然后选择保存到MicroPython板的Flash中：



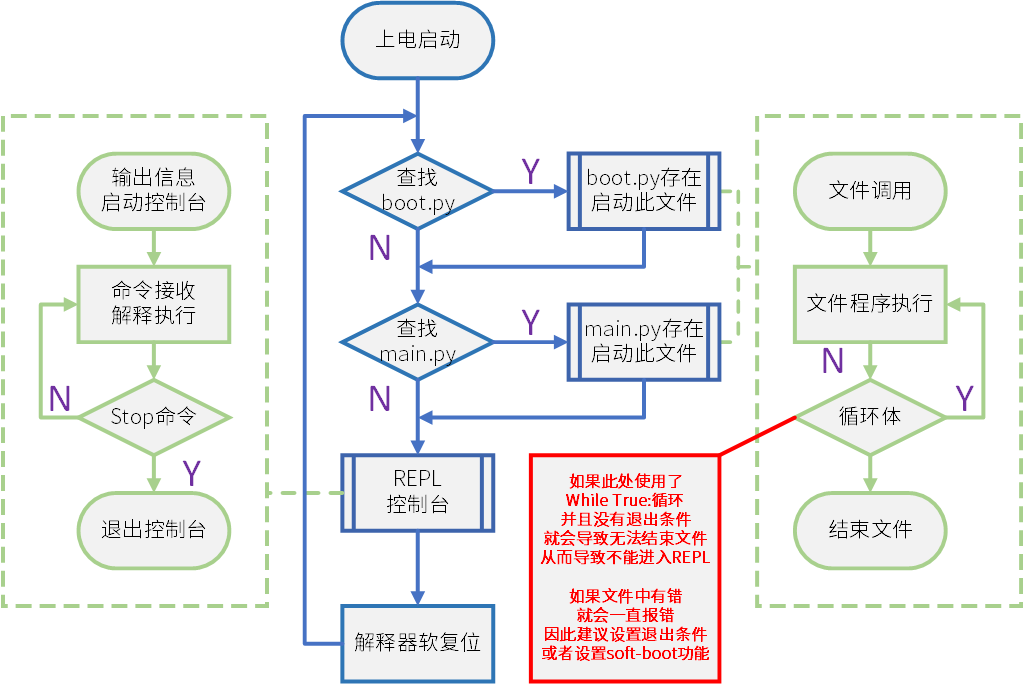
如果文件管理器窗口被不小心关闭了，那么可以通过工具栏 “View->Files” 勾选重新打开：



## RT1021-144Pin-MicroPython启动顺序说明

核心板默认烧录了MicroPython的固件，该固件是基于RT1021DAG5x-LQFP144芯片进行资源规划并开发的。固件中使用了Flash中一块区域作为Python文件管理系统，固定路径为/flash，用户可以将自己的文件保存在Flash中。

MicroPython的固件设置了默认的Python文件启动流程，上电**默认查找/flash路径下是否存在boot.py和main.py文件，如果存在对应文件就会直接执行对应文件**，固件启动流程：

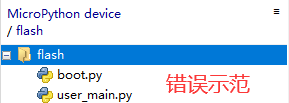


**因此RT1021-144Pin-MicroPython固件默认入口文件为boot.py和main.py，**如果/flash中有对应文件就会自动被执行。

**不要自己新建文件夹目录例如/flash/boot/boot.py，固件不会检索子目录！！！**

**不要自己新建文件夹目录例如/flash/boot/boot.py，固件不会检索子目录！！！**

**不要自己新建文件夹目录例如/flash/boot/boot.py，固件不会检索子目录！！！**



**需要注意尽量在boot.py和main.py文件中保留按键退出或者定时退出的功能，否则可能导致程序在文件死循环，无法进入REPL控制台。**

# 选择学习板上的二号拨码开关作为退出选择开关

end\_switch **=** Pin**(**'C19'**,** Pin**.**IN**,** pull**=**Pin**.**PULL\_UP\_47K**,** value **=** **True)**

end\_state **=** end\_switch**.**value**()**

**while** **True:**

**...**

# 如果拨码开关打开 对应引脚拉低 就退出循环

# 这么做是为了防止写错代码导致异常 有一个退出的手段

**if** end\_switch**.**value**()** **!=** end\_state**:**

**print(**"Ticker stop."**)**

**break**

## 如何脱机运行自己的Python文件

由于RT1021-144Pin-MicroPython固定启动boot.py与main.py，那么可以通过这两个文件来跳转执行自己的文件，例如通过boot.py使用拨码开关实现的soft-boot跳转执行user\_main.py：

# 本示例程序演示如何通过 boot.py 文件进行 soft-boot 控制

# 使用 RT1021-144Pin-MicroPython 核心板搭配对应拓展学习板的拨码开关控制

# 示例程序运行效果为复位后执行本文件 通过 C18 电平状态跳转执行 user\_main.py 或进入 main.py

# 从 machine 库包含所有内容

**from** machine **import** **\***

# 包含 gc 与 time 类

**import** gc,time

# 上电启动时间延时

time**.**sleep\_ms**(**50**)**

# 选择学习板上的一号拨码开关作为启动选择开关

boot\_select **=** Pin**(**'C18'**,** Pin**.**IN**,** pull**=**Pin**.**PULL\_UP\_47K**,** value **=** **True)**

# 如果拨码开关打开 对应引脚拉低 就启动用户文件

**if** boot\_select**.**value**()** **==** 0**:**

**try:**

os**.**chdir**(**"/flash"**)**

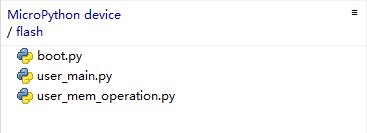
execfile**(**"user\_main.py"**)**

**except:**

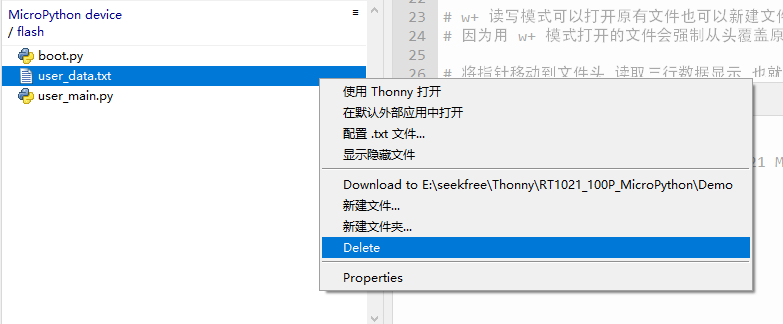
**print(**"File not found."**)**

这样就可以通过拨码开关来选择上电后自动通过boot.py直接启动user\_main.py文件执行，还是跳过执行直接进入REPL控制台。

**当通过boot.py实现软BootLoader时，建议main.py文件保持为空（即新建main.py文件但什么都不写或删除main.py文件保证/flash目录下无该文件），这样可以确保可以通过boot开关选择直接进入REPL**。（推荐保持/flash中只有boot.py没有main.py，例如：）



这样就可以避免文件写了错误的代码导致无法正常连接的问题，直接对储存在Flash的文件进行修改、删除。



# MicroPython固件接口说明

## machine基础类

本部分接口是MicroPython原生machine的实现，只有与原MicroPython的machine小部分的差异。

### Pin子模块

GPIO与外部中断接口，参数与原生略有差异：

Pin**(**pin**,** mode**,** **[,** pull **=** Pin**.**PULL\_UP\_47K**,** value **=** 1**,** drive **=** Pin**.**DRIVE\_OFF**])**

# 构造接口 是标准 MicroPython 的 machine.Pin 模块 参数说明

# pin 引脚名称 | 必要参数 引脚名称 本固件以核心板上引脚编号为准

# mode 引脚模式 | 必要参数 对应引脚工作状态 可用值为 Pin.[IN, OUT, OPEN\_DRAIN]

# pull 上拉下拉 | 可选参数 可用值为 Pin.[PULL\_UP, PULL\_UP\_47K, PULL\_UP\_22K, PULL\_DOWN, PULL\_HOLD]

# value 初始电平 | 可选关键字参数 可以设置为 0,1 对应低电平与高电平

# drive 内阻模式 | 可选关键字参数 可用值为 Pin.[PIN\_DRIVE\_OFF, PIN\_DRIVE\_[0-6]]

# 例：

**from** machine **import** Pin

rst **=** Pin**(**'B8' **,** Pin**.**OUT**,** pull**=**Pin**.**PULL\_UP\_47K**,** value**=**1**)**

Pin**.**on**()** # 端口电平置位

Pin**.**off**()** # 端口电平复位

Pin**.**low**()** # 端口电平输出低电平

Pin**.**high**()** # 端口电平输出高电平

Pin**.**toggle**()** # 端口电平翻转

Pin**.**value**(**x**)** # 传入参数 x 则将端口电平设置为对应 bool 值

# 不传入参数则只返回端口电平 bool 值

Pin**.**irq**(**handler**,** trigger**,** hard**)** # 参数说明

# handler 回调函数 | 必要参数 触发后对应的回调函数 python 函数

# trigger 触发模式 | 必要参数 可用值为 Pin.[IRQ\_RISING, IRQ\_FALLING]

# hard 应用模式 | 可选参数 可用值为 False True

### ADC子模块

ADC模块，与原生的MicroPython的ADC模块基本一致，不过传入的是Pin类的对象，或者引脚的名称编号，这里以核心板上标注的引脚编号为准。

ADC**(id)** # 构造接口 是标准 MicroPython 的 machine.ADC 模块 参数说明

# id 引脚名称 | 必要参数 引脚名称 本固件以核心板上引脚编号为准

# 例：

**from** machine **import** ADC

adc **=** ADC**(**'B22'**)**

ADC**.**read\_u16**()** # 读取当前端口的 ADC 转换值

需要注意的是，引脚可能同时支持ADC1和ADC2模块的通道输入，调用ADC的构造函数时，会自动适配到ADC1模块上。

### UART子模块

兼容原生MicroPython的UART的模块。

UART**(id)** # 构造接口 标准 MicroPython 的 machine.UART 模块 参数说明

# id 串口编号 | 必要参数 本固件支持 0 - 6 总共 7 个 UART 模块

**from** machine **import** UART

uart1 **=** UART**(**2**)**

# HW-UART | Logical | TX | RX |

# ---------------------------------

# LPUART1 | 0 | DEBUG USB |

# LPUART1 | 0 | B6 | B7 |

# LPUART2 | 1 | C22 | C23 |

# LPUART3 | 2 | C6 | C7 |

# LPUART4 | 3 | D0 | D1 |

# LPUART5 | 4 | B10 | B11 |

# LPUART6 | 5 | D20 | D21 |

# LPUART7 | 6 | D17 | D18 |

# LPUART8 | 7 | D22 | D23 |

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

# 串口参数设置 参数说明

# baudrate串口速率 |默认 9600

# bits 数据位数 |默认 8 bits 数据位

# parity 校验位数 |默认 无校验

# stop 停止位数 |默认 1 bit 停止位

**from** machine **import** UART

uart1 **=** UART**(**2**)**

uart1**.**init**(**9600**)**

# 其余接口：

buf **=** uart1**.**read**(**n**)** # 读取 n字节到 buf

uart1**.**readinto**(**buf**)** # 读取数据节到 buf

uart1**.**write**(**buf**)** # 将 buf 内容通过 UART 发送

uart1**.any()** # 判断 UART 是否有数据可读取

### SPI子模块

SPI模块，基本兼容MicroPython的SPI模块：

*SPI***(id)** # 构造接口 标准 MicroPython 的 machine.SPI 模块 参数说明

# id 串口编号 | 必要参数 本固件支持 0 - 3 总共 4 个 SPI 模块

**from** machine **import** SPI

spi **=** SPI**(**1**)**

# HW-UART | Logical | SCK | MOSI | MISO | CS0 |

# LPSPI1 | id = 0 | B10 | B12 | B13 | B11 |

# LPSPI2 | id = 1 | C10 | C12 | C13 | C11 |

# LPSPI3 | id = 2 | B28 | B30 | B31 | B29 |

# LPSPI4 | id = 3 | B18 | B20 | B21 | B19 |

SPI**.***init***(**baudrate **=** 1000000**,** polarity **=** 0**,** phase **=** 0**)**

# 串口参数设置 参数说明

# baudrate 传输速率 | 默认 1000000 1Mbps

# polarity 数据位数 | 默认 0 空闲时钟线电平 可以是 0 或 1

# phase 校验位数 | 默认 0 在第一个或第二个时钟沿上采样数据 可以是 0 或 1

**from** machine **import** SPI

spi **=** SPI**(**1**)**

spi**.***init***(**baudrate **=** 1000000**,** polarity **=** 0**,** phase **=** 0**)**

# 其余接口：

rx\_byte **=** spi**.***read***(**1**)** # 读取一个字节数据 默认输出 0x00

tx\_buff **=** **bytearray(**b'1234'**)**

rx\_buff **=** **bytearray(len(**tx\_buf**))**

spi**.***readinto***(**rx\_buff**)** # 读取 rx\_buff 长度数据 默认输出 0x00

spi**.***write***(**tx\_buff**)** # 输出 tx\_buff 长度数据

# 输出 tx\_buff 长度数据 同时读取数据到

# tx\_buff rx\_buff 这两个缓冲区必须一样长

spi**.***write\_readinto***(**tx\_buff**,** rx\_buff**)**

### IIC子模块

IIC模块，基本兼容MicroPython的IIC模块：

*I2C***(id)** # 构造接口 标准 MicroPython 的 machine.I2C 模块 参数说明

# id 串口编号 | 必要参数 本固件支持 0 - 3 总共 4 个 I2C 模块

**from** machine **import** I2C

iic **=** I2C**(**1**,** freq **=** **...)**

# HW-I2C | Logical | SCL | SDA |

# LPI2C1 | id = 0 | B30 | B31 |

# LPI2C2 | id = 1 | C19 | C18 |

# LPI2C3 | id = 2 | B8 | B9 |

# LPI2C4 | id = 3 | D22 | D23 |

**from** machine **import** SPI

# freq | 传输速率 | 默认 400000

iic **=** I2C**(**3**,** freq **=** 100000**)**

# 其余接口：

# 需要注意的是

# 任意的总线错误都会导致程序报错

# 任意的总线错误都会导致程序报错

# 任意的总线错误都会导致程序报错

# 包括 NACK 、 起始停止异常等

# 读取一个字节数据 True 发送停止信号 False 不发生停止信号

rx\_byte **=** iic**.***readfrom***(**device\_list**[**0**],** 1**,** **True)**

# 读取 rx\_buff 长度数据 True 发送停止信号 False 不发生停止信号

iic**.***readfrom\_into***(**device\_list**[**0**],** rx\_buff**,** **True)**

# 输出 tx\_buff 长度数据 True 发送停止信号 False 不发生停止信号

iic**.***writeto***(**device\_list**[**0**],** tx\_buff**,** **True)**

# 输出 vectors 矩阵数据 True 发送停止信号 False 不发生停止信号

iic**.***writevto***(**device\_list**[**0**],** vectors**,** **True)**

不推荐使用I2C接口，因为它非常容易报错，一旦操作有误、通信线路受干扰，它就会直接报错停止运行。由于NXP固件中仅实现了硬件的IIC部分，软件IIC暂未支持，如果使用基本I2C操作接口进行自拟定的操作，也无法避免通信错误，并且效率极低。

### PWM子模块

PWM模块，基本兼容MicroPython的PWM模块：

PWM**(**pin**,** freq**,** duty\_u16**[,** kw\_opts**])**

# 构造接口 参数说明

# pin 引脚名称 | 必要参数 对应核心板上有 PWM 功能的引脚

# freq 工作频率 | 必要参数

# duty\_u16初始脉宽 | 必要参数 1 - 65535

# 例：

**from** machine **import** PWM

pwm1 **=** PWM**(**"D4"**,** 13000**,** duty\_u16 **=** 1**)**

pwm1**.**duty\_u16**([**value**])** # 传入 value 则更新占空比设置 否则仅反馈当前占空比设置

pwm1**.**freq**([**value**])** # 传入 value 则更新频率设置 否则仅反馈当前频率设置

## NXP支持的smartcar模块

本部分由NXP官方编写支持，主要为方便传感器使用并提高运行效率。

### ADC\_Group子模块

为方便进行多通道的ADC采集，简化使用步骤提高运行效率而实现的一个接口类。

ADC\_Group**(id)**

# 构造接口 参数说明

# id 索引编号 | 必要参数 RT1021 共两个 ADC 模块 因此此处索引号范围为 1,2

# 例：

**from** smartcar **import** ADC\_Group

adc1 **=** ADC\_Group**(**1**)**

构造函数会通过参数绑定到对应的芯片ADC模块，这里推荐使用对应模块的引脚绑定。

ADC\_Group**.**addch**(**pin**)** # 添加 ADC 通道 参数说明

# pin 引脚名称 | 必要参数 对应引脚 可选1021的ADC功能引脚

# 例：

adc1**.**addch**(**"B14"**)**

adc1**.**addch**(**"B14"**)**

ADC\_Group**.**capture**()** # 进行一次序列 ADC 通道转换

# 例：

adc1**.**capture**()**

ADC\_Group**.**get**()** # 将所有通道数据输出为一个元组

# 例：

adc\_data **=** adc1**.**get**()**

ad\_l **=** adc\_data**[**0**]**

ad\_r **=** adc\_data**[**1**]**

**print(**"adc ={:>6d}, {:>6d}\r\n"**.format(**ad\_l**,** ad\_r**))**

ADC\_Group**.**read**()** # 立即进行一次序列 ADC 通道转换 将所有通道数据输出为一个元组

# 例：

adc\_data **=** adc1**.**read**()**

ad\_l **=** adc\_data**[**0**]**

ad\_r **=** adc\_data**[**1**]**

**在使用ticker模块时，可以使用ticker的采样列表来关联ADC\_Group进行自动数据转换（详见Ticker章节）**，仅需要get()读取，不再使用capture()进行转换采集或者read()转换输出。

### encoder子模块

专门实现的编码器接口，用于采集编码器数据。

encoder**(**pinA**,** pinB**[,** invert **=** ADC\_Group**.**AVGx**])** # 构造接口 参数说明

# pinA 引脚名称 | 必要参数 对应的编码器 A 相输入或 PLUS 引脚

# pinB 引脚名称 | 必要参数 对应的编码器 B 相输入或 DIR 引脚

# invert 反向控制 | 可选关键字参数 bool 开启的话输出方向反向

# 例：

**from** smartcar **import** encoder

encoder\_l **=** encoder**(**"C0"**,** "C1"**)**

encoder\_r **=** encoder**(**"C0"**,** "C1"**,** **True)**

encoder**.**capture**()** # 将编码器当前计数缓存并输出 并清空计数开始下一次采集

# 例：

encl\_data **=** encoder\_l**.**capture**()**

encr\_data **=** encoder\_r**.**capture**()**

encoder**.**get**()** # 输出当前采集缓存的编码器采集计数

# 例：

encl\_data **=** encoder\_l**.**get**()**

encr\_data **=** encoder\_r**.**get**()**

encoder**.**read**()** # 将编码器当前计数输出

# 例：

encl\_data **=** encoder\_l**.**read**()**

encr\_data **=** encoder\_r**.**read**()**

**在使用ticker模块时，可以使用ticker的采样列表来关联encoder进行自动数据转换（详见Ticker章节）**，仅需要get()读取，不再使用capture()进行采集或者read()采集输出。

### Ticker子模块

用于实现实时的硬件定时器中断，并自动执行传感器的capture采集。

ticker**(id)** # 构造接口 参数说明

# id 模块编号 | 必要参数 0-3 对应 4 个硬件 PIT 通道

# 例：

**from** smartcar **import** ticker

pit1 **=** ticker**(**0**)**

ticker**.**capture\_list**(**obj1**,** obj2**,** **....)** # 传感器 capture 关联 参数说明

# objx 模块对象 | 必要参数 最少一个 最多八个

# 例：

adc1 **=** ADC\_Group**(**1**)**

encoder\_l **=** encoder**(**"C0"**,** "C1"**)**

**...**

pit1**.**capture\_list**(**adc1**,** encoder\_l**[,...])**

# ADC\_Group\_x 与 encoder\_x 为 smartcar 的接口类

# IMU660RA, IMU963RA, KEY\_HANDLER, TSL1401 为 seekfree 的接口类

ticker**.**callback**(**soft**[,** hard**])** # 回调函数绑定 参数说明

# soft 软件回调 | 必要参数 软回调 绑定 Python 函数 周期触发解释器调度

# hard 硬件回调 | 可选参数 硬回调 绑定 Python 函数 周期底层定时器执行

# 例：

**def** dis **(**pit\_x**):**

**print(**"tick"**)**

pit1**.**callback**(**dis**)**

ticker**.**start**(**period**)** # 启动定时器 需要注意 必须先关联传感器或者添加了回调才能启动

# period 触发周期 | 必要参数 毫秒单位

# 例：

pit1**.**start**(**500**)**

# 其余接口：

ticker**.**stop**()** # 停止定时器

ticker**.**ticks**()** # 返回定时器执行的次数

推荐的代码处理方式是：在代码中添加手动停止Ticker代码，启用了Ticker模块时可以通过按键或者其他方式停止Ticker，然后再使用Thonny的Stop按钮，Thonny就可以正常复位Python解释器并重新连接到REPL控制台：

**from** machine **import** Pin

**from** smartcar **import** **\***

end\_switch **=** Pin**(**'C19'**,** Pin**.**IN**,** pull**=**Pin**.**PULL\_UP\_47K**,** value **=** **True)**

ticker\_flag **=** **False**

**def** time\_pit\_handler**(**time**):**

**global** ticker\_flag # 全局修饰 代表使用的是全局 ticker\_flag

ticker\_flag **=** **True** # 对变量赋值

adc1 **=** ADC\_Group**(**1**)**

encoder\_l **=** encoder**(**"C0"**,** "C1"**)**

**...**

pit **=** ticker**(**1**)**

pit1**.**capture\_list**(**adc1**,** encoder\_l**[,...])**

pit**.**callback**(**time\_pit\_handler**)**

pit**.**start**(**100**)**

**while** **True:**

**if** ticker\_flag**:**

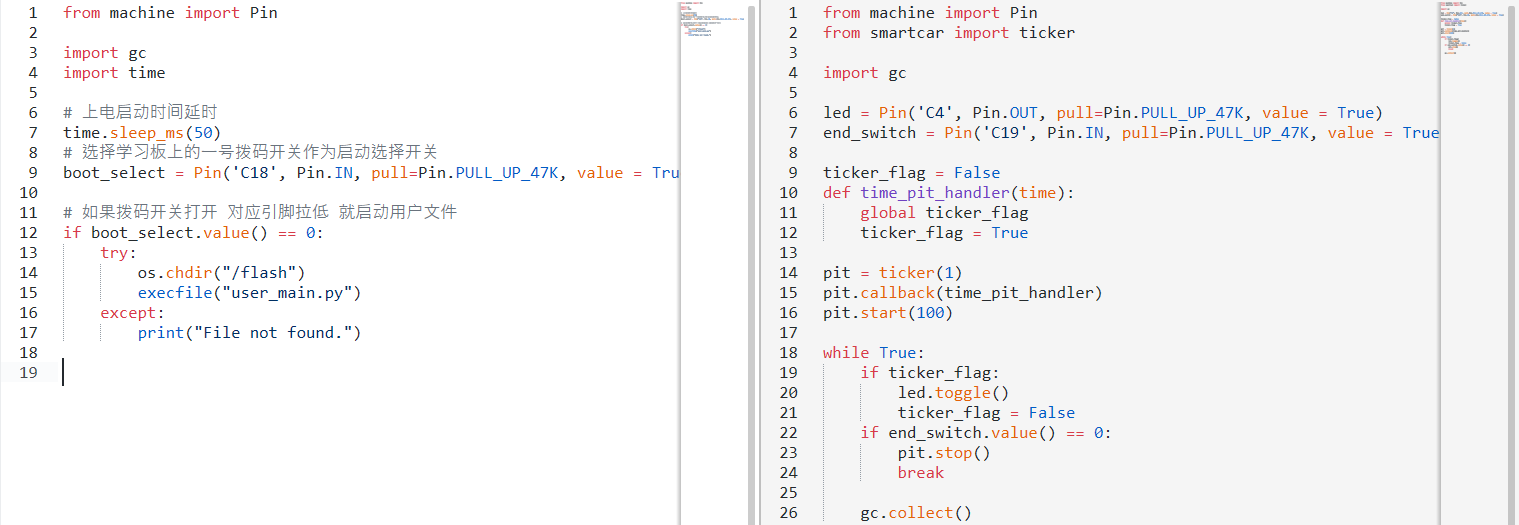
ticker\_flag **=** **False**

**if** end\_switch**.**value**()** **==** 0**:** # 判断拨码开关电平状态 低电平有效

pit**.**stop**()** # 停止 TIcker 模块

**break** # 退出循环

例如上述代码使用一个拨码开关实现了手动停止Ticker并退出循环。一般建议boot.py中使用拨码开关实现选择启动，然后不使用main.py，这样就可以在调试时，使用拨码开关跳过自动运行，在Thonny中打开对应代码文件运行来进行调试：



例如上图左侧是boot.py文件的软BootLoader，右侧是user\_main.py源码，在调试时保持boot\_select开关为高电平，随后通过end\_switch开关控制停止Tikcer，就可以使用Thonny的停止命令直接重启到REPL控制台，随后就可以再手动运行user\_main.py文件调试。

## NXP支持的display模块

由于实际车模调试需要脱机操作，那么添加一个屏幕用于显示调试参数变成了一个迫切的需求，因此NXP提供了一个display模块兼容逐飞科技的IPS200-SPI串口屏幕用于显示。

LCD\_Drv**(**SPI\_INDEX**,** BAUDRATE**,** DC\_PIN**,** RST\_PIN**,** LCD\_TYPE**)** # 构造接口 学习主板上的屏幕接口

# SPI\_INDEX 接口索引 | 必填关键字参数 选择屏幕所用的 SPI 接口索引

# BAUDRATE 通信速率 | 必填关键字参数 SPI 的通信速率 最高 60MHz

# DC\_PIN 命令引脚 | 必填关键字参数 一个 Pin 实例

# RST\_PIN 复位引脚 | 必填关键字参数 一个 Pin 实例

# LCD\_TYPE 屏幕类型 | 必填关键字参数 目前仅支持 LCD\_Drv.LCD200\_TYPE

# 例：

**from** machine **import** **\***

**from** display **import** **\***

# 定义片选引脚

cs **=** Pin**(**'B29'**,** Pin**.**OUT**,** pull**=**Pin**.**PULL\_UP\_47K**,** value**=**1**)**

# 拉高拉低一次 CS 片选确保屏幕通信时序正常

cs.high**()**

cs.low**()**

# 定义控制引脚

rst **=** Pin**(**'B31'**,** Pin**.**OUT**,** pull**=**Pin**.**PULL\_UP\_47K**,** value**=**1**)**

dc **=** Pin**(**'B5' **,** Pin**.**OUT**,** pull**=**Pin**.**PULL\_UP\_47K**,** value**=**1**)**

# 需要注意的是背光引脚需要自己控制 可以调节亮度 不过一般给高电平即可

blk **=** Pin**(**'C21'**,** Pin**.**OUT**,** pull**=**Pin**.**PULL\_UP\_47K**,** value**=**1**)**

# 新建 LCD 驱动实例 这里的索引范围与 SPI 示例一致 当前仅支持 IPS200 目前绑定 SPI\_INDEX = 1

drv **=** LCD\_Drv**(**SPI\_INDEX**=**2**,** BAUDRATE**=**60000000**,** DC\_PIN**=**dc**,** RST\_PIN**=**rst**,** LCD\_TYPE**=**LCD\_Drv**.**LCD200\_TYPE**)**

# 新建 LCD 实例

lcd **=** LCD**(**drv**)**

LCD**.**color**(**pcolor**,** bgcolor**)** # 修改 LCD 的前景色与背景色

# pcolor 前景色 | 必填参数 RGB565 格式

# bgcolor 背景色 | 必填参数 RGB565 格式

# 例：

lcd**.**color**(**0xFFFF**,** 0x0000**)**

LCD**.**mode**(dir)** # 修改 LCD 的显示方向

# dir 显示方向 | 必填参数 [0:竖屏,1:横屏,2:竖屏180旋转,3:横屏180旋转]

# 例：

lcd**.**mode**(**2**)**

LCD**.**clear**(**color**)** # 清屏显示

# color 清屏颜色 | 必填参数 RGB565 格式

# 例：

lcd**.**clear**(**0x0000**)**

LCD**.**str12**(**x**,** y**,** **str[,** color**])** # 显示字符串

LCD**.**str16**(**x**,** y**,** **str[,** color**])** # 显示字符串

LCD**.**str24**(**x**,** y**,** **str[,** color**])** # 显示字符串

LCD**.**str32**(**x**,** y**,** **str[,** color**])** # 显示字符串

# x 起点横轴 | 必填参数

# y 起点纵轴 | 必填参数

# str 字符数据 | 必填参数 字符串数据

# color 显示颜色 | 可选参数 RGB565 格式 显示字符颜色

# 例：

lcd**.**str12**(**0**,** 0**,**"15={:b},{:d},{:o},{:#x}."**.format(**15**,**15**,**15**,**15**),**0xF800**)**

lcd**.**str16**(**0**,**12**,**"1.234={:>.2f}."**.format(**1.234**),**0x07E0**)**

lcd**.**str24**(**0**,**28**,**"123={:<6d}."**.format(**123**),**0x001F**)**

lcd**.**str32**(**0**,**52**,**"123={:>6d}."**.format(**123**),**0xFFFF**)**

LCD**.**line**(**x1**,** y1**,** x2**,** y2**[,** color**,** thick**])** # 清屏显示

# x 起点横轴 | 必填参数

# y 起点纵轴 | 必填参数

# x 终点横轴 | 必填参数

# y 终点纵轴 | 必填参数

# color 显示颜色 | 可选参数 RGB565 格式 线条颜色

# thick 线条粗细 | 可选参数 默认为 1 数值越大越粗

# 例：

lcd**.**line**(**0**,**84**,**200**,**16 **+** 84**,**color**=**0xFFFF**,**thick**=**1**)**

lcd**.**line**(**200**,**84**,**0**,**16 **+** 84**,**color**=**0x3616**,**thick**=**3**)**

## 逐飞科技支持的seekfree模块

由于在Python语言层面实现一个传感器的驱动比较麻烦，并且效率会降低很多，不利于获取实时数据，因此基于smartcar的sensor框架编写了几个模块方便使用与调试（但所用到的引脚暂时不可随意修改，使用固定的引脚）。

### IMU 660/963 RA子模块

可以使用学习板板的IMU接口（SCK-C10/MOSI-C12/MISO-C13/CS-C11）连接IMU660RA或者IMU963RA姿态传感器直接使用。

IMUxxxRA**()** # 构造接口 支持使用主板上的 IMU 接口连接 IMU660RA 或者 IMU963RA 模块

# period 采集分频 | 非必要参数 默认为 1 也就是每次都采集 代表多少次触发进行一次采集

# 例：

**from** seekfree **import** IMUxxxRA

imu **=** IMUxxxRA **()**

IMUxxxRA**.**capture**()** # 执行一次 IMU 数据采集触发 达到触发数时执行采集并将数据缓存

# 例：

imu**.**capture**()**

IMUxxxRA**.**get**()** # 输出当前采集缓存的 IMU 数据

# 例：

imu\_data **=** imu**.**get**()**

**print(**"acc = {:>6d}, {:>6d}, {:>6d}."**.format(**imu660ra\_data**[**0**],** imu660ra\_data**[**1**],** imu660ra\_data**[**2**]))**

**print(**"gyro = {:>6d}, {:>6d}, {:>6d}."**.format(**imu660ra\_data**[**3**],** imu660ra\_data**[**4**],** imu660ra\_data**[**5**]))**

# IMU660RA 数据为 6 个 int 类型的数据 acc\_x/y/z gyro\_x/y/z

# IMU963RA 数据为 9 个 int 类型的数据 acc\_x/y/z gyro\_x/y/z mag\_x/y/z

IMUxxxRA**.**read**()** # 立即进行一次 capture 并输出缓存数据

# 例：

imu\_data **=** imu**.**read**()**

**print(**"acc = {:>6d}, {:>6d}, {:>6d}."**.format(**imu660ra\_data**[**0**],** imu660ra\_data**[**1**],** imu660ra\_data**[**2**]))**

**print(**"gyro = {:>6d}, {:>6d}, {:>6d}."**.format(**imu660ra\_data**[**3**],** imu660ra\_data**[**4**],** imu660ra\_data**[**5**]))**

# IMU660RA 数据为 6 个 int 类型的数据 acc\_x/y/z gyro\_x/y/z

# IMU963RA 数据为 9 个 int 类型的数据 acc\_x/y/z gyro\_x/y/z mag\_x/y/z

**在使用ticker模块时，可以使用ticker的采样列表来关联IMU进行自动数据转换（详见Ticker章节）**，仅需要get()读取，不再使用capture()进行采集或者read()采集输出。

### KEY\_HANDLER子模块

四个按键（C8/C9/C14/C15）的驱动，支持短按松发（按下后松开触发）和长按检测。

KEY\_HANDLER**(**period**)** # 构造接口 学习主板上的按键驱动

# period 扫描周期 | 必填参数 按键的扫描周期 一般配合填写 Tickter 的运行周期

# 例：

**from** seekfree **import** KEY\_HANDLER

key **=** KEY\_HANDLER**(**10**)**

KEY\_HANDLER**.**capture**()** # 执行一次按键状态扫描

# 例：

key**.**capture**()**

KEY\_HANDLER**.**get**()** # 输出当前四个按键状态

# 例：

key\_data **=** key**.**get**()**

**print(**"key = {:>6d}, {:>6d}, {:>6d}, {:>6d}."**.format(**key\_data**[**0**],** key\_data**[**1**],** key\_data**[**2**],** key\_data**[**3**]))**

KEY\_HANDLER**.**clear**([**index**])** # 清除按键状态 长按会锁定长按状态不被清除

# index 按键序号 | 可选参数 1 - 4 清除对应按键的触发状态

# 例：

key**.**clear**()**

### MOTOR\_CONTROLLER子模块

支持学习板上的电机驱动信号接口，引脚固定（[C28/C29/C30/C31]、[D4/D5/D6/D7]），可用组合为PWM\_C30\_DIR\_C31、PWM\_C28\_DIR\_C29、PWM\_D4\_DIR\_D5、PWM\_D6\_DIR\_D7、PWM\_C30\_PWM\_C31、PWM\_C28\_PWM\_C29、PWM\_D4\_PWM\_D5、PWM\_D6\_PWM\_D7，可以用于直接驱动DRV8701双驱或者HIP4082双驱。

MOTOR\_CONTROLLER**(**index**,** freq**,[**duty**,** invert**])** # 构造接口 学习主板上的电机驱动信号接口

# index 电机索引 | 必填参数 [PWM\_C30\_DIR\_C31,PWM\_C28\_DIR\_C29,PWM\_D4\_DIR\_D5,PWM\_D6\_DIR\_D7,

# PWM\_C30\_PWM\_C31,PWM\_C28\_PWM\_C29,PWM\_D4\_PWM\_D5,PWM\_D6\_PWM\_D7]

# freq 信号频率 | 必填参数 PWM 信号的频率 范围是 [1 - 100000]

# duty 初始占空比 | 可选关键字参数 默认为 0 范围 ±10000 正数正转 负数反转 正转反转方向取决于 invert

# invert 扫描周期 | 可选关键字参数 是否反向 默认为 0 可以通过这个参数调整电机方向极性

# 例：

**from** seekfree **import** MOTOR\_CONTROLLER

motor\_l **=** MOTOR\_CONTROLLER**(**MOTOR\_CONTROLLER**.**PWM\_C30\_DIR\_C31**,** 13000**,** duty **=** 0**,** invert **=** **True)**

MOTOR\_CONTROLLER**.**duty**([**duty**])** # 更新或获取占空比值

# duty 占空比 | 可选参数 填数值就设置新的占空比 否则返回当前占空比 范围是 ±10000

# 例：

motor\_l**.**duty**(**1000**)**

### BLDC\_CONTROLLER子模块

支持学习板上的无刷电机电调信号接口C25/C26，可以用于直接驱动负压风扇无刷电调。

BLDC\_CONTROLLER**(**index**,[**freq**,** highlevel\_us**])** # 构造接口 学习主板上的电调接口

# index 接口索引 | 必填参数 可选参数为 [PWM\_C25, PWM\_C27]

# freq 信号频率 | 可选关键字参数 PWM 频率 范围 50-300 默认 50

# highlevel\_us 高电平时长 | 可选关键字参数 初始的高电平时长 范围 1000-2000 默认 1000

# 例：

**from** seekfree **import** BLDC\_CONTROLLER

bldc1 **=** BLDC\_CONTROLLER**(**BLDC\_CONTROLLER**.**PWM\_C25**,** freq**=**300**,** highlevel\_us **=** 1000**)**

BLDC\_CONTROLLER**.**highlevel\_us**([**highlevel\_us**])** # 更新或获取占空比值

# highlevel\_us 高电平时长 | 可选参数 填数值就设置新的高电平时长 否则返回当前高电平时长 范围是 [1000-2000]

# 例：

bldc2**.**highlevel\_us**(**1000**)**

### WIRELESS\_UART子模块

支持学习板上的串口无线模块接口，固定使用UART3的C6-TX/C7-RX以及一个C5-RTS流控引脚，直接适配V2.4版本以上无线串口模块，用于对接逐飞助手上位机，方便进行调试。

WIRELESS\_UART**([**baudrate**])** # 构造接口 学习主板上的串口无线模块接口

# baudrate 波特率 | 可选参数 默认 460800

# 例：

**from** seekfree **import** WIRELESS\_UART

wireless **=** WIRELESS\_UART**(**460800**)**

WIRELESS\_UART**.**send\_str**(str)** # 发送字符串

# str 字符串数据 | 必填参数

# 例：

wireless**.**send\_str**(**"Hello World.\r\n"**)**

wireless**.**send\_str**(**"hall\_count ={:>6d}, hall\_state ={:>6d}"**.format(**hall\_count**,** x**.**value**()))**

WIRELESS\_UART**.**send\_oscilloscope**(**d1**,[**d2**,** d3**,** d4**,** d5**,** d6**,** d7**,** d8**])** # 逐飞助手虚拟示波器数据上传

# dx 波形数据 | 至少有一个数据 最多可以填八个数据

# 例：

wireless**.**send\_oscilloscope**(**

data\_wave**[**0**],**data\_wave**[**1**],**data\_wave**[**2**],**data\_wave**[**3**],**

data\_wave**[**4**],**data\_wave**[**5**],**data\_wave**[**6**],**data\_wave**[**7**])**

WIRELESS\_UART**.**send\_ccd\_image**(**index**)** # 逐飞助手 CCD 显示数据上传

# index 接口编号 | 参数为 [CCD1\_BUFFER\_INDEX,CCD2\_BUFFER\_INDEX,

# CCD3\_BUFFER\_INDEX,CCD4\_BUFFER\_INDEX,ALL\_CCD\_BUFFER\_INDEX]

# | 分别代表 仅显示 CCD1 图像 、 仅显示 CCD2 图像 、 两个 CCD 图像一起显示

例：

wireless**.**send\_ccd\_image**(**WIRELESS\_UART**.**CCD1\_BUFFER\_INDEX**)**

WIRELESS\_UART**.**data\_analysis**()** # 逐飞助手调参数据解析 会返回八个数据的列表

WIRELESS\_UART**.**get\_data**()** # 逐飞助手调参数据获取 会返回八个数据的列表

# 例：

**while** **True:**

data\_flag **=** wireless**.**data\_analysis**()**

**for** i **in** **range(**0**,**8**):**

# 判断哪个通道有数据更新

**if** **(**data\_flag**[**i**]):**

# 数据更新到缓冲

data\_wave**[**i**]** **=** wireless**.**get\_data**(**i**)**

# 将更新的通道数据输出到 Thonny 的控制台

**print(**"Data[{:<6}] updata : {:<.3f}.\r\n"**.format(**i**,**data\_wave**[**i**]))**

### TSL1401子模块

支持学习板上的线阵CCD接口，默认CCD1/2/3/4都接入，可以用于直接驱动红孩儿线阵CCD，固定使用B22/B23/B24/B25四个模拟输入引脚，占用ADC2模块，以及B13/B16作为控制引脚。如果只接入1个CCD模块，可以通过索引号获取对应的CCD数据。

TSL1401**([**period**])** # 构造接口 学习主板上的线阵 CCD 接口

# period 采集分频 | 非必要参数 默认为 1 也就是每次都采集 代表多少次触发进行一次采集

# 例：

**from** seekfree **import** TSL1401

**from** smartcar **import** ticker

ccd **=** TSL1401**(**10**)**

pit1 **=** ticker**(**1**)**

pit1**.**capture\_list**(**ccd**)**

TSL1401**.**capture**()** # 执行一次 CCD 数据采集触发 达到触发数时执行采集并将数据缓存

# 例：

ccd**.**capture**()**

数据以元组方式获取，可以调用屏幕的显示接口显示在屏幕上，或无线串口的发送接口上传到逐飞助手显示。默认为8bit的数据精度：

TSL1401**.**get**(**index**)** # 将对应 CCD 通道数据输出为一个元组

# index 接口索引 | 必填参数 选择 0-CCD1 1-CCD2 接口

例：

ccd\_data1 **=** ccd**.**get**(**0**)**

# 通过 wave 接口显示数据波形 (x,y,width,high,data,data\_max)

# x 横轴坐标 | 必填参数 起始显示 X 坐标

# y 纵轴坐标 | 必填参数 起始显示 Y 坐标

# width 显示宽度 | 必填参数 等同于数据个数

# high 显示高度 | 必填参数 实际显示高度 因为数据可能比屏幕高度值大

# data 波形数据 | 必填参数 数据对象 这里基本仅适配 TSL1401 的 get 接口返回的数据对象

# max 最大数值 | 可选关键字参数 数据最大值 TSL1401 的数据范围 默认 255

lcd**.**wave**(**0**,** 0**,** 128**,** 64**,** ccd\_data1**,** **max** **=** 255**)**

TSL1401**.**read**(**index**)** # 立即进行一次 capture 并输出缓存数据

# index 接口索引 | 必填参数 选择 0-CCD1 1-CCD2 接口

# 例：

ccd\_data1 **=** ccd**.**read**(**0**)**

lcd**.**wave**(**0**,** 0**,** 128**,** 64**,** ccd\_data1**)**

可以通过新的接口修改CCD的转换精度：

TSL1401**.**set\_resolution**(**resolution**)** # 设置 CCD 的转换精度

# resolution 接口索引 | 必填参数 [RES\_8BIT, RES\_12BIT]

例：

ccd**.**set\_resolution**(**TSL1401.RES\_12BIT**)**

### DL1B子模块

DL1B子模块，支持学习板上的ToF模块接口，固定使用SCL-D0/SDA-D1/XS-D2/三个引脚，推荐挂载在Ticker下自动采集。

DL1B **()** # 构造接口 支持使用主板上的 ToF 接口连接 DL1B 模块

# period 采集分频 | 非必要参数 默认为 1 也就是每次都采集 代表多少次触发进行一次采集

# 例：

**from** seekfree **import** DL1B

tof **=** DL1B**()**

DL1B**.**capture**()** # 执行一次 DL1B 数据采集触发 达到触发数时执行采集并将数据缓存

# 例：

tof**.**capture**()**

DL1B**.**get**()** # 输出当前采集缓存的 DL1B数据

# 例：

tof\_data **=** tof**.**get**()**

**print(**"distance = {:>6d}."**.format(**tof\_data**))**

DL1B**.**read**()** # 立即进行一次 capture 并输出缓存数据

# 例：

tof\_data **=** tof**.**read**()**

**print(**"distance = {:>6d}."**.format(**tof\_data**))**

### WIFI\_SPI子模块

WIFI\_SPI子模块，支持学习板上的WIFI\_SPI模块接口，固定使用SCK-B18/MOSI-B20/MISO-B21/CS-B19/INT-B3/RST-B4引脚，与无线模块使用基本相同。

*WIFI\_SPI***(wifi\_ssid, pass\_word, connect\_type, ip\_addr, connect\_port)**

# 构造接口 学习主板上的WIFI\_SPI模块接口

# wifi\_ssid WiFi 名称 字符串

# pass\_word WiFi 密码 字符串

# connect\_type 连接类型 WIFI\_SPI.TCP\_CONNECT / WIFI\_SPI.UDP\_CONNECT

# ip\_addr 目标连接地址 字符串

# connect\_port 目标连接端口 字符串

# 例：

**from** seekfree **import** WIFI\_SPI

wifi **=** WIFI\_SPI**(**"WIFI\_NAME"**,** "WIFI\_PASSWORD"**,** WIFI\_SPI**.***TCP\_CONNECT***,** "192.168.1.13"**,** "8086"**)**

WIFI\_SPI**.***send\_str***(str)** # 发送字符串

# str 字符串数据 | 必填参数

# 例：

wifi**.***send\_str***(**"Hello World.\r\n"**)**

wifi**.***send\_str***(**"hall\_count ={:>6d}, hall\_state ={:>6d}"**.format(**hall\_count**,** x**.***value***()))**

WIFI\_SPI**.***send\_oscilloscope***(**d1**,[**d2**,** d3**,** d4**,** d5**,** d6**,** d7**,** d8**])** # 逐飞助手虚拟示波器数据上传

# dx 波形数据 | 至少有一个数据 最多可以填八个数据

# 例：

wifi**.***send\_oscilloscope***(**

data\_wave**[**0**],**data\_wave**[**1**],**data\_wave**[**2**],**data\_wave**[**3**],**

data\_wave**[**4**],**data\_wave**[**5**],**data\_wave**[**6**],**data\_wave**[**7**])**

WIFI\_SPI**.***send\_ccd\_image***(**index**)** # 逐飞助手 CCD 显示数据上传

# index 接口编号 | 参数为 [CCD1\_BUFFER\_INDEX,CCD2\_BUFFER\_INDEX,

# CCD3\_BUFFER\_INDEX,CCD4\_BUFFER\_INDEX,ALL\_CCD\_BUFFER\_INDEX]

# | 分别代表 仅显示 CCD1 图像 、 仅显示 CCD2 图像 、 两个 CCD 图像一起显示

wifi**.***send\_ccd\_image***(**WIFI\_SPI**.***CCD1\_BUFFER\_INDEX***)**

WIFI\_SPI**.***data\_analysis***()** # 逐飞助手调参数据解析 会返回八个数据的列表

WIFI\_SPI**.***get\_data***()** # 逐飞助手调参数据获取 会返回八个数据的列表

# 例：

**while** **True:**

data\_flag **=** wifi**.***data\_analysis***()**

**for** i **in** **range(**0**,**8**):**

# 判断哪个通道有数据更新

**if** **(**data\_flag**[**i**]):**

# 数据更新到缓冲

data\_wave**[**i**]** **=** wifi**.***get\_data***(**i**)**

# 将更新的通道数据输出到 Thonny 的控制台

**print(**"Data[{:<6}] updata : {:<.3f}.\r\n"**.format(**i**,**data\_wave**[**i**]))**

## 多Python源码文件的包含与调用

假设以boot.py启动user\_main.py运行用户程序，但是用户自己的控制源码是单独封装为另一个文件controller.py：

**from** machine **import** **\***

**import** gc

servo\_duty\_mid **=** 29490.75

servo\_duty\_diff\_max **=** 1092.25

servo **=** PWM**(**"C20"**,** 300**,** **int(**servo\_duty\_mid**))**

**def** servo\_duty\_set **(**duty**):**

duty **=** **(**duty**)** **if(**servo\_duty\_diff\_max **>** duty**)** **else** **(**servo\_duty\_diff\_max**)**

duty **=** **(**duty**)** **if(**duty **>** -servo\_duty\_diff\_max**)** **else** **(**-servo\_duty\_diff\_max**)**

pwm\_servo**.**duty\_u16**(int(**servo\_duty\_mid **+** duty**))**

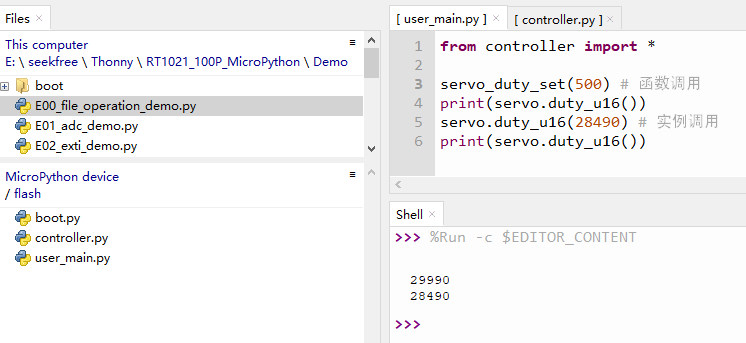
那么可以通过如下代码在user\_main.py中调用controller.py中的接口：

**from** controller **import** **\***

servo\_duty\_set**(**500**)** # 函数调用

servo**.**duty\_u16**(**500**)** # 实例调用

是否能直接调用controller.py中的servo对象？servo.duty\_u16(500)就是直接使用的servo对象进行操作。



其他的方式是否能够使用？例如Class封装等等，为什么不直接试一试呢？实践出真知。

# 文档版本

|  |  |  |  |
| --- | --- | --- | --- |
| 版本号 | 日期 | 作者 | 内容变更 |
| V1.0 | 2024/12/09 | TQC | 初始版本。 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |