

RT1021-144Pin-MicroPython 固件接口说明

目录

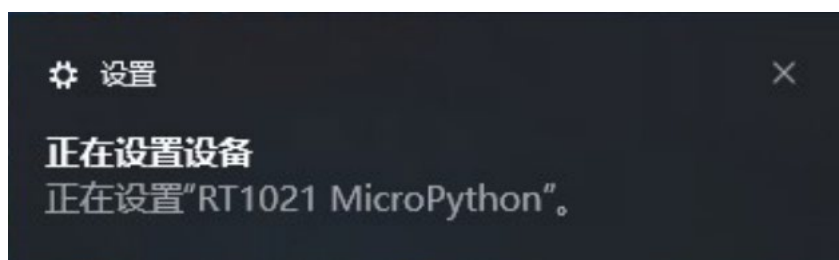
目录.....	1
1. RT1021-144Pin-MicroPython 核心板简要说明	3
1.1. RT1021-144Pin-MicroPython 核心板连接 Thonny 启动 REPL 控制台	3
1.2. 使用 Thonny 保存源码到板载 FLASH 中	5
1.3. RT1021-144Pin-MicroPython 启动顺序说明.....	7
1.4. 如何脱机运行自己的 Python 文件	8
2. MicroPython 固件接口说明	10
2.1. machine 基础类.....	10
2.1.1. Pin 子模块.....	10
2.1.2. ADC 子模块.....	10
2.1.3. UART 子模块.....	11
2.1.4. PWM 子模块.....	12
2.2. NXP 支持的 smartcar 模块.....	14
2.2.1. ADC_Group 子模块.....	14
2.2.2. encoder 子模块.....	15
2.2.3. Ticker 子模块	15
2.3. NXP 支持的 display 模块.....	17
2.4. 逐飞科技支持的 seekfree 模块.....	19
2.4.1. IMU 660/963 RA 子模块	19
2.4.2. KEY_HANDLER 子模块.....	20
2.4.3. MOTOR_CONTROLLER 子模块.....	20

2.4.4. BLDC_CONTROLLER 子模块	21
2.4.5. WIRELESS_UART 子模块	21
2.4.6. TSL1401 子模块	22
2.4.7. DL1B 子模块	23
2.5. 多 Python 源码文件的包含与调用	25
3. 文档版本	26

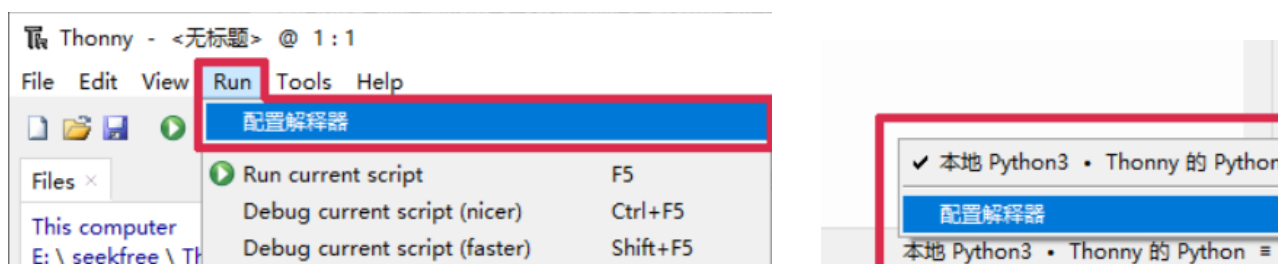
1.RT1021-144Pin-MicroPython 核心板简要说明

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

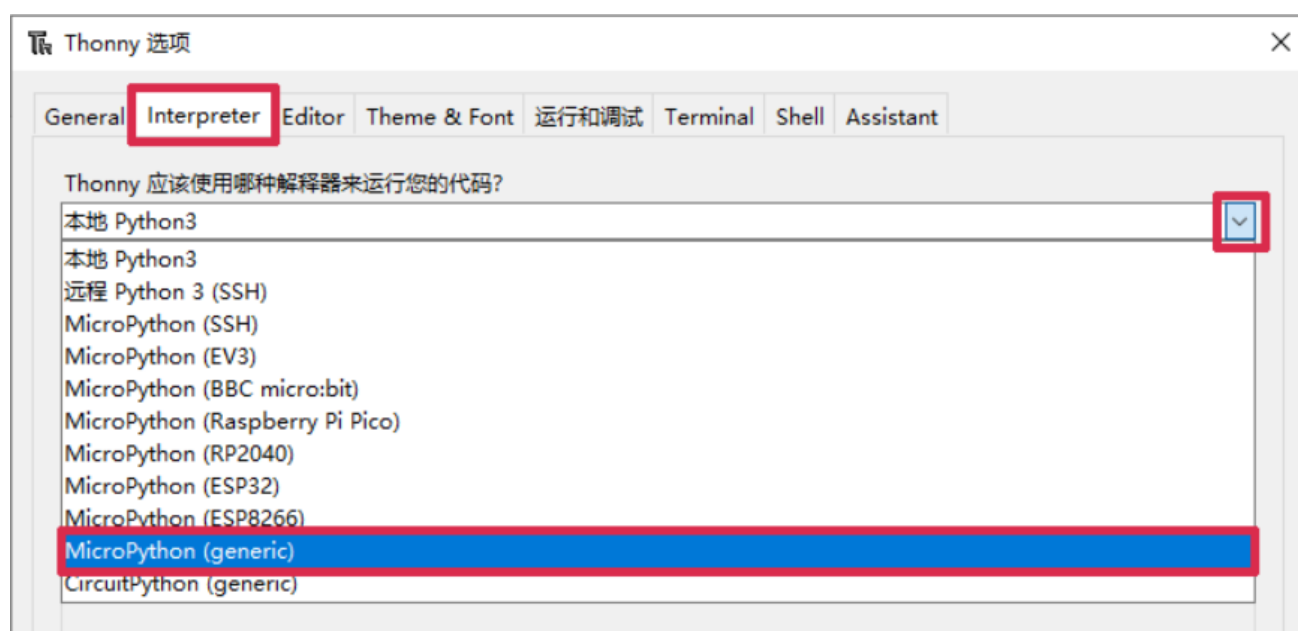
首先将 RT1021 核心板使用 Type-C 线连接 PC，核心板中已默认烧录 MicroPython 固件，使用 USB-CDC 虚拟串口控制，因此在 PC 上会识别到一个串口设备：



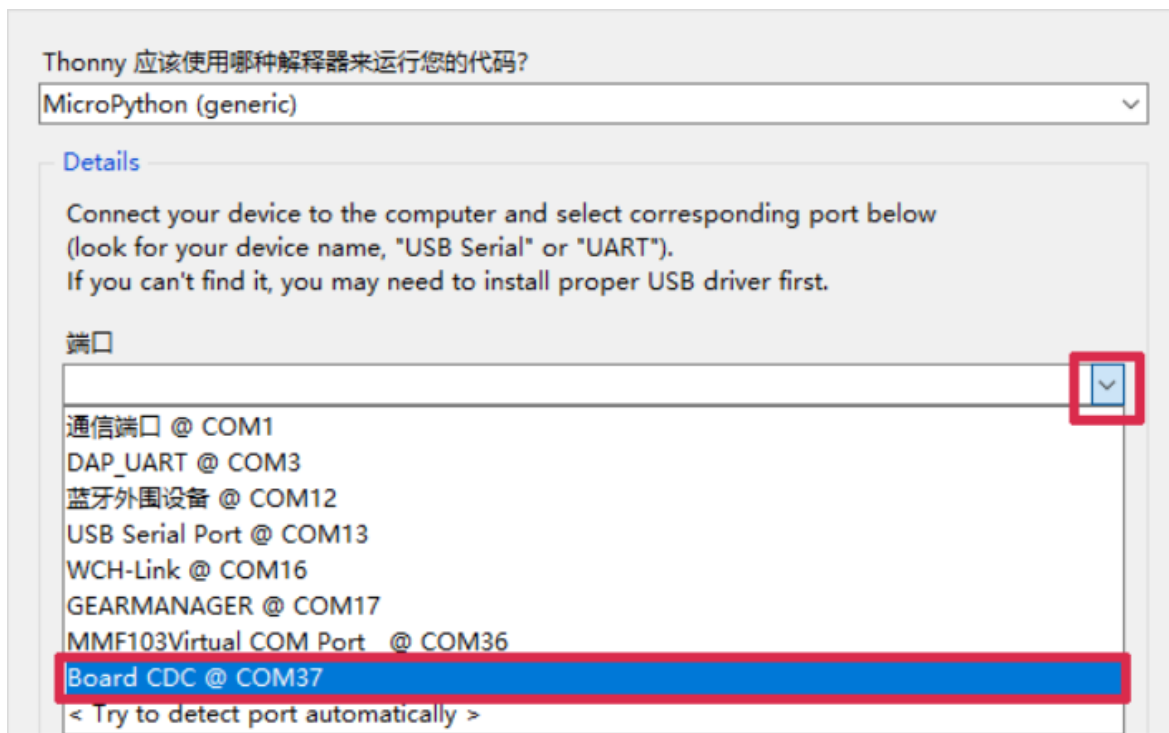
打开 Thonny，打开菜单栏的“运行->解释器配置”选项，或者点击窗口右下角的解释器切换按钮，打开对应的 Thonny 选项界面：



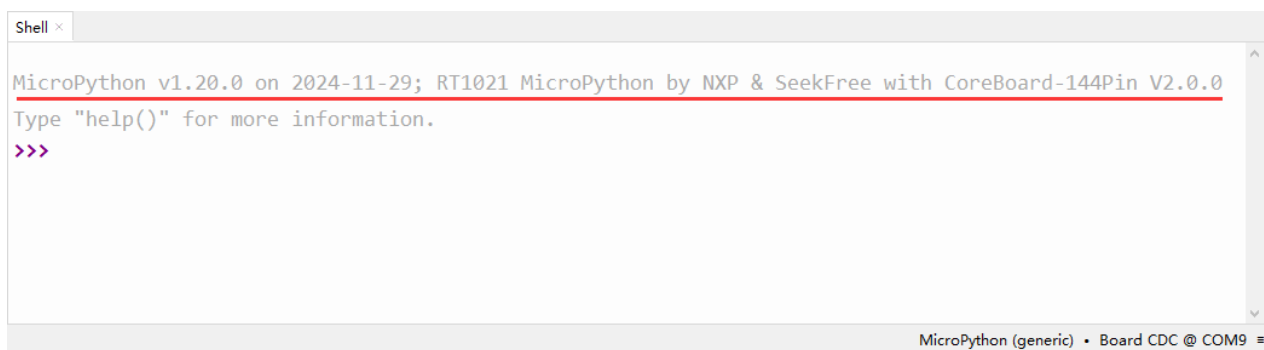
在 Interpreter 选项卡，选择 MicroPython(generic)解释器：



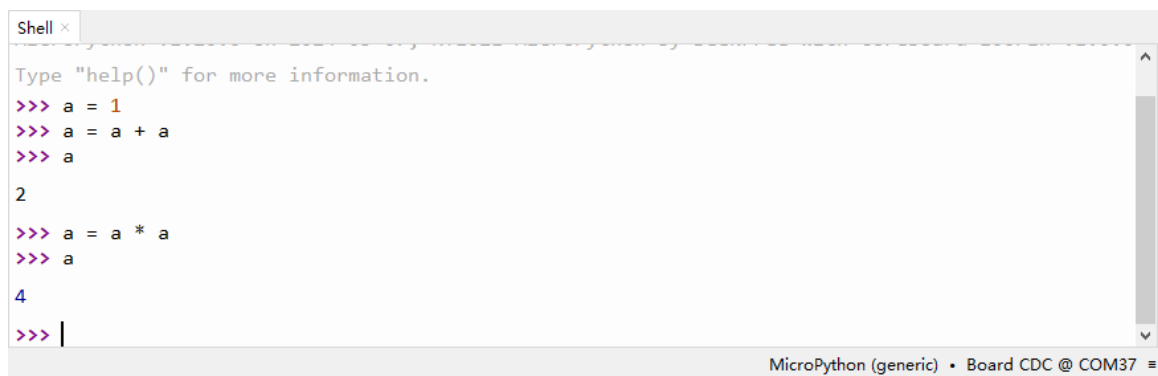
随后并找到核心板对应的 COM 口（此处名称可能会有差异）设置好，并确认保存：



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：



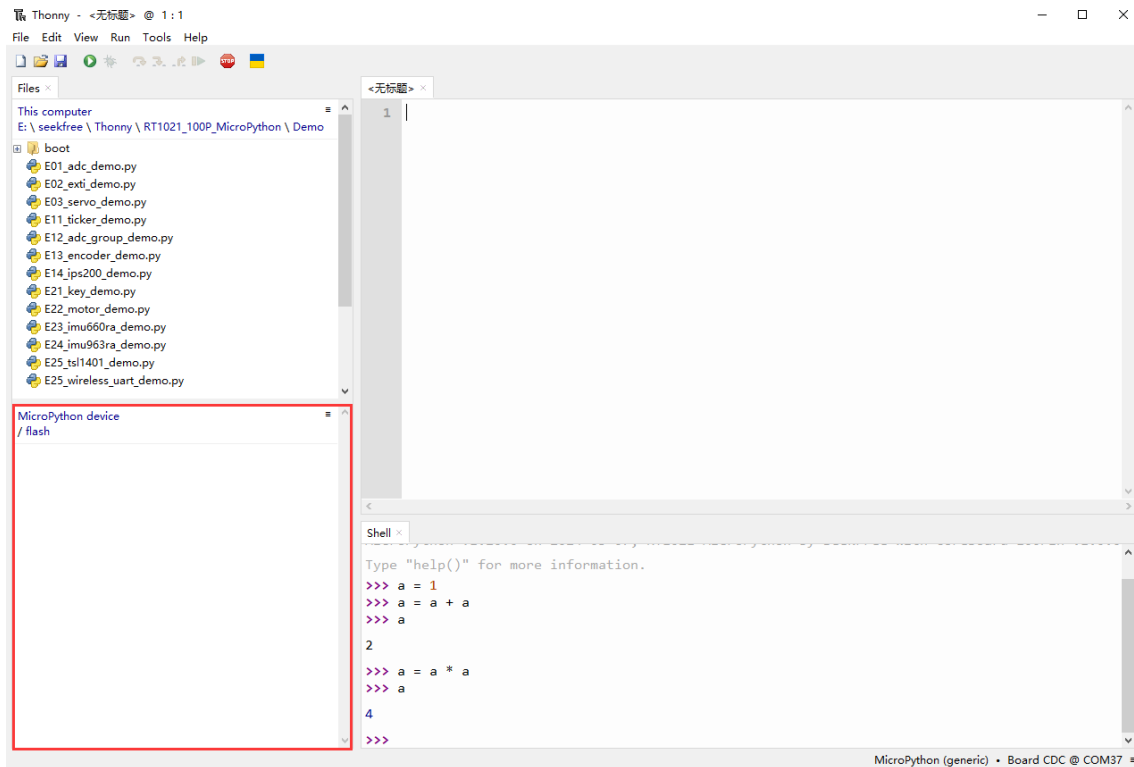
可以简单输入 Python 语句测试：



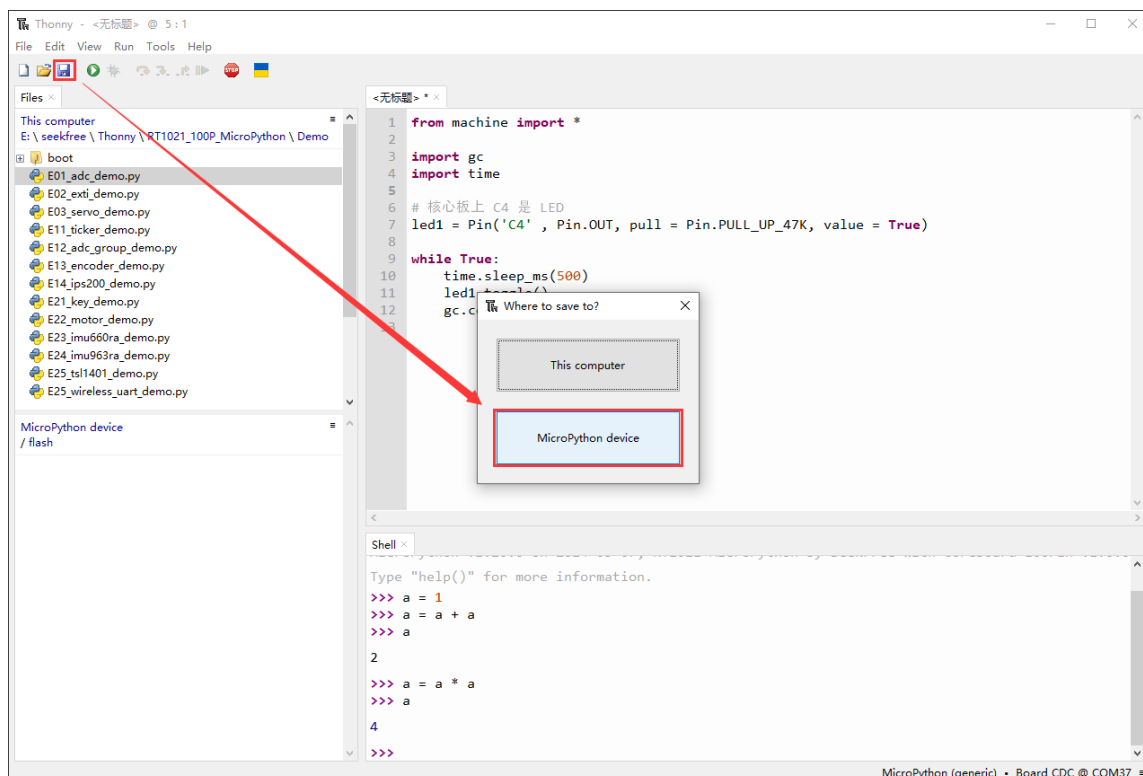
接下来便可以使用 Thonny 进行 Python 的编辑与调试了。

1.2.使用 Thonny 保存源码到板载 FLASH 中

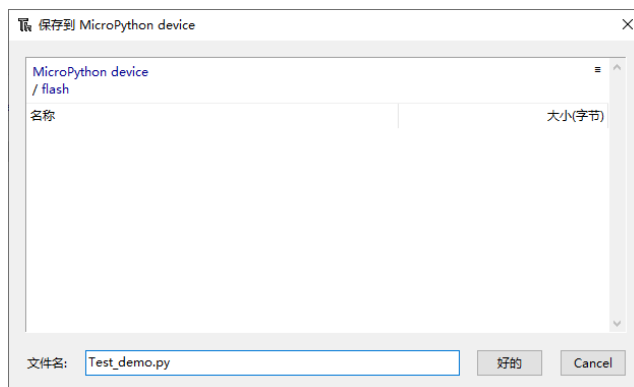
连接到 MicroPython 目标板后，Thonny 会显示出当前设备的 Flash 中的文件：



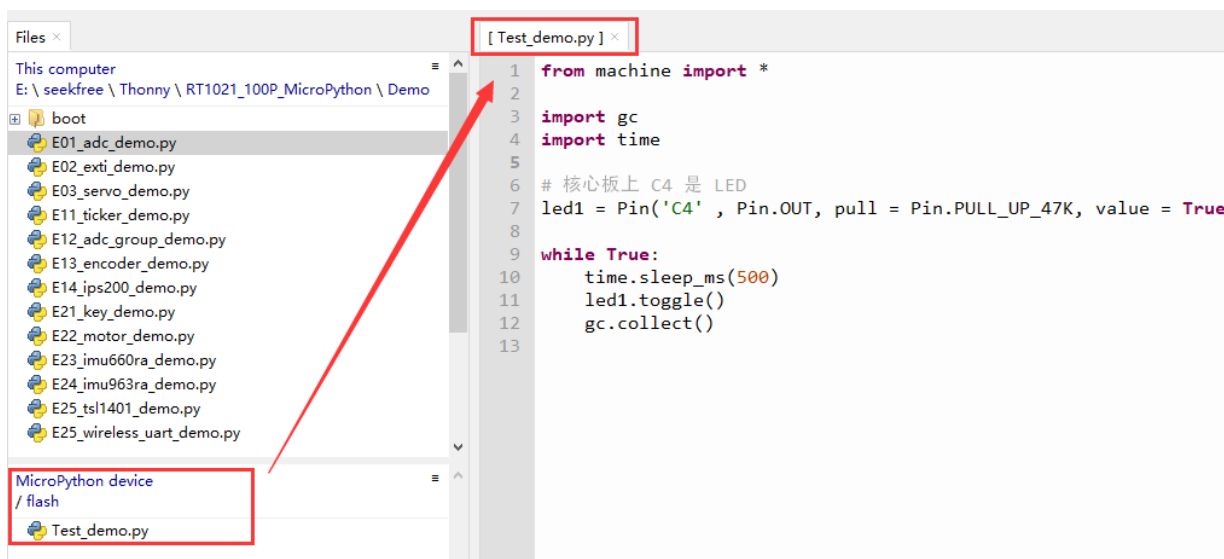
自己新建的 Python 文件可以通过 Thonny 保存到 MicroPython 板的 Flash 中：



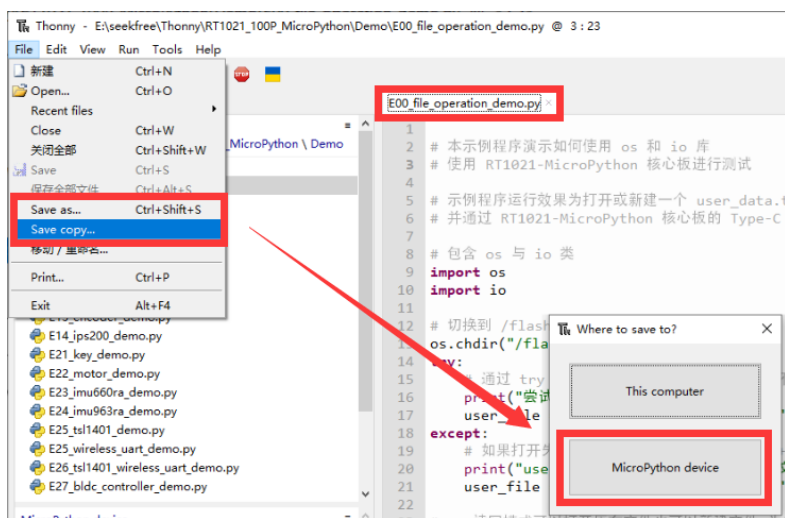
在弹出文件列表窗口命名好文件并确认即可，也可以在这个界面选中文件然后删除



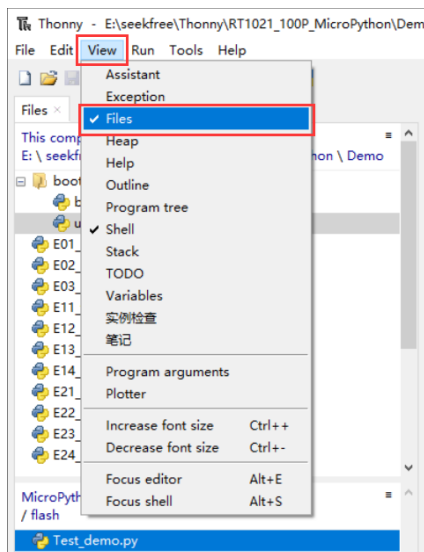
保存完成后, MicroPython device 文件显示栏会显示设备中的文件, 打开设备中的文件时, 文件名会被中括号标识:



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:



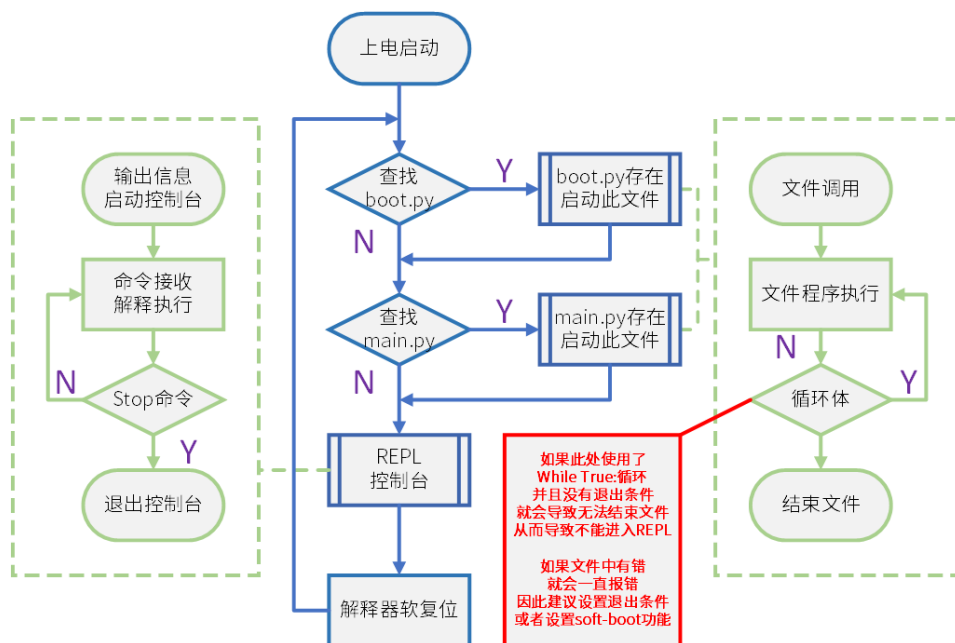
如果文件管理器窗口被不小心关闭了，那么可以通过工具栏“View->Files”勾选重新打开：



1.3.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：

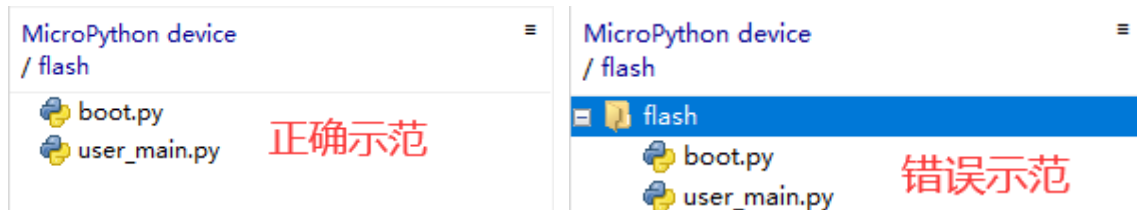


因此 RT1021-144Pin-MicroPython 固件默认入口文件为 `boot.py` 和 `main.py`，如果/flash 中有对应文件就会自动被执行。

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
```

1.4.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
```

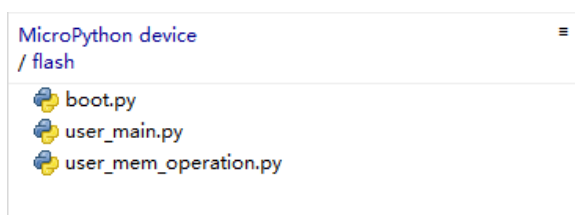
```
import gc,time

# 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)

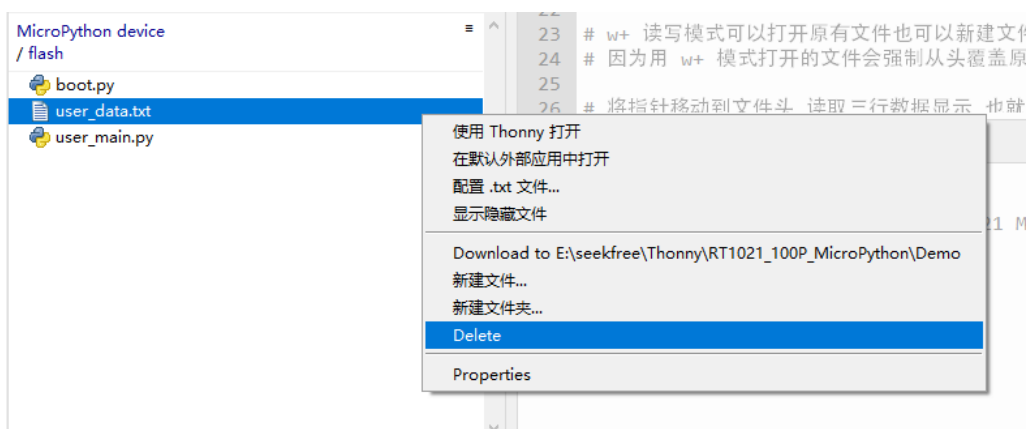
# 如果拨码开关打开 对应引脚拉低 就启动用户文件
if boot_select.value() == 0:
    try:
        os.chdir("/flash")
        execfile("user_main.py")
    except:
        print("File not found.")
```

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。



2. MicroPython 固件接口说明

2.1. 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。

2.1.1. 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
```

2.1.2. 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。

2.1.3.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
```

2.1.4.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_buff))
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)
```

2.1.5.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。

2.1.6.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
```

2.2.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。

2.2.1.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。

2.2.2.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。

2.2.3.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_1 = encoder("C0", "C1")
...
pit1.capture_list(adc1, encoder_1[,...])
# 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_1 = encoder("C0", "C1")
...
pit = ticker(1)
pit1.capture_list(adc1, encoder_1[,...])
pit.callback(time_pit_handler)
pit.start(100)

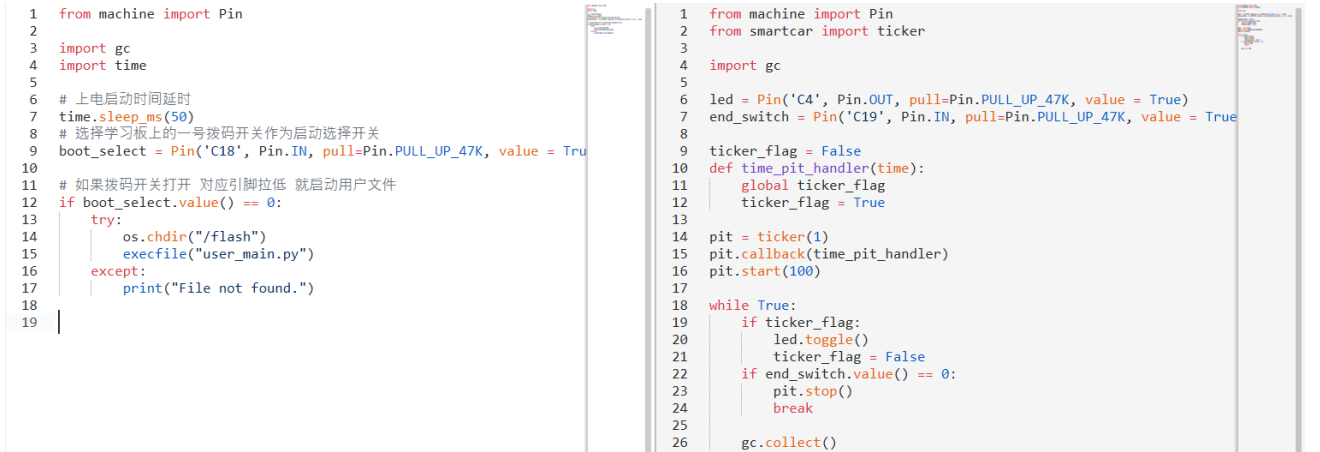
while True:
    if ticker_flag:
```

```

    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：



```

1 from machine import Pin
2
3 import gc
4 import time
5
6 # 上电启动时间延时
7 time.sleep_ms(50)
8 # 选择学习板上的一号拨码开关作为启动选择开关
9 boot_select = Pin('C18', Pin.IN, pull=Pin.PULL_UP_47K, value = True)
10
11 # 如果拨码开关打开 对应引脚拉低 就启动用户文件
12 if boot_select.value() == 0:
13     try:
14         os.chdir("/flash")
15         execfile("user_main.py")
16     except:
17         print("File not found.")
18
19

```

```

1 from machine import Pin
2 from smartcar import ticker
3
4 import gc
5
6 led = Pin('C4', Pin.OUT, pull=Pin.PULL_UP_47K, value = True)
7 end_switch = Pin('C19', Pin.IN, pull=Pin.PULL_UP_47K, value = True)
8
9 ticker_flag = False
10 def time_pit_handler(time):
11     global ticker_flag
12     ticker_flag = True
13
14 pit = ticker(1)
15 pit.callback(time_pit_handler)
16 pit.start(100)
17
18 while True:
19     if ticker_flag:
20         led.toggle()
21         ticker_flag = False
22     if end_switch.value() == 0:
23         pit.stop()
24         break
25
26 gc.collect()

```

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 Ticker，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。

2.3.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=6000000, 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        字符串数据 | 必填参数 字符串数据
# color      显示颜色 | 可选参数 RGB565 格式 显示字符串颜色
# 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)
```

2.4.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）。

2.4.1.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。

2.4.2.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 Ticker 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()
```

2.4.3.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_1 = 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_1.duty(1000)
```

2.4.4.BLDC_CONTROLLER 子模块

支持学习板上的无刷电机调信号接口 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)
```

2.4.5.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]))
```

2.4.6.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,height,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
# height 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)
```

2.4.7.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))
```

2.4.8.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]))
```

2.5.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

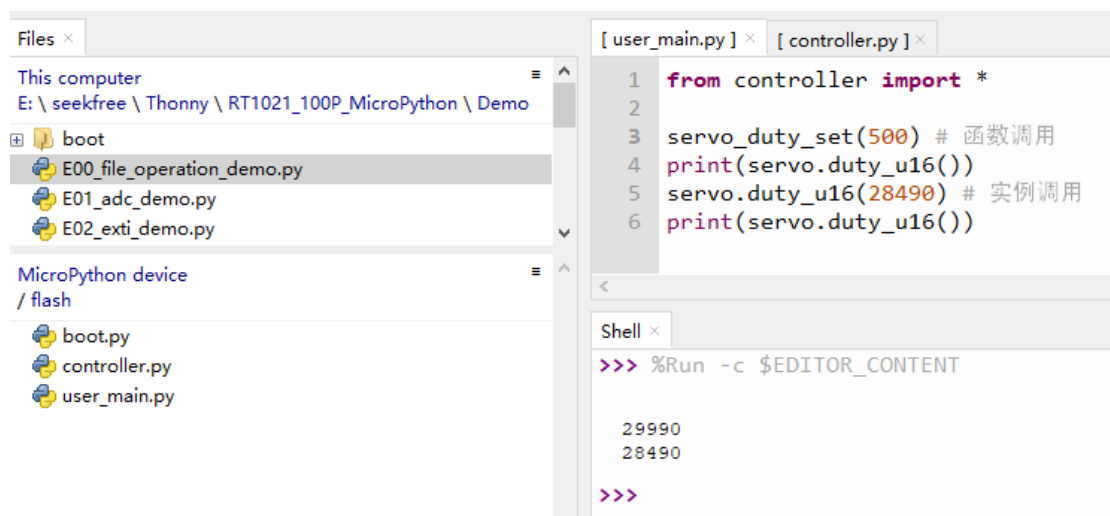
servo_duty_mid      = 29490.75
servo_duty_diff_max = 1092.25
servo = PWM("C20", 300, int(servo_duty_mid))

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。

3.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。