

RT1021-100Pin-MicroPython 固件接口说明

目录

目录	1
1. 前言	3
2. RT1021-MicroPython 核心板连接与脱机启动	4
2.1. 开源 MicroPython 开发工具	4
2.2. RT1021-MicroPython 核心板连接电脑	6
2.3. Thonny 连接 RT1021-MicroPython 核心板	7
2.4. RT1021-MicroPython 核心板启动顺序说明	9
2.5. 使用 Thonny 进行文件操作	11
2.6. 脱机运行示范	13
2.7. 注意事项与常见问题	13
3. RT1021-MicroPython 核心板固件说明	15
3.1. 底层接口 machine 库	15
3.1.1. Pin 子模块	15
3.1.2. ADC 子模块	15
3.1.3. PWM 子模块	16
3.1.4. UART 子模块	16
3.1.5. SPI 子模块	17
3.1.6. IIC 子模块	18

3.2. NXP 的 smartcar 库	19
3.2.1. ticker 子模块.....	19
3.2.2. ADC_Group 子模块.....	19
3.2.3. encoder 子模块	20
3.3. NXP 的 display 库.....	21
3.4. 逐飞科技的 seekfree 库.....	23
3.4.1. MOTOR_CONTROLLER 子模块.....	23
3.4.2. BLDC_CONTROLLER 子模块	24
3.4.3. KEY_HANDLER 子模块	24
3.4.4. IMU660/963RX 子模块.....	25
3.4.5. DL1X 子模块.....	25
3.4.6. TSL1401 子模块	26
3.4.7. 传感器子模块与 ticker 的 caputer_list 关联采集说明	27
3.4.8. WIRELESS_UART 子模块.....	29
4. 多文件调用与类定义	30
4.1. 多文件加载.....	30
4.2. 多文件变量作用域.....	31
4.3. 类的应用	32
5. 文档版本.....	34

1.前言

本文档的主要作用是帮助各位快速入门 RT1021-MicroPython 开发,主要内容分三个部分。

第一个部分: 环境与硬件连接说明, 内容为环境安装与介绍, 硬件检查与连接, RT1021-MicroPython 核心板硬件启动流程以及注意事项;

第二个部分: 固件接口说明, 内容为核心板固件自带的接口库说明, 用于配合例程学习如何使用对应的接口与传感器驱动;

第三个部分: 关于多文件的加载与引用问题, 以及 Class 的使用示例等。

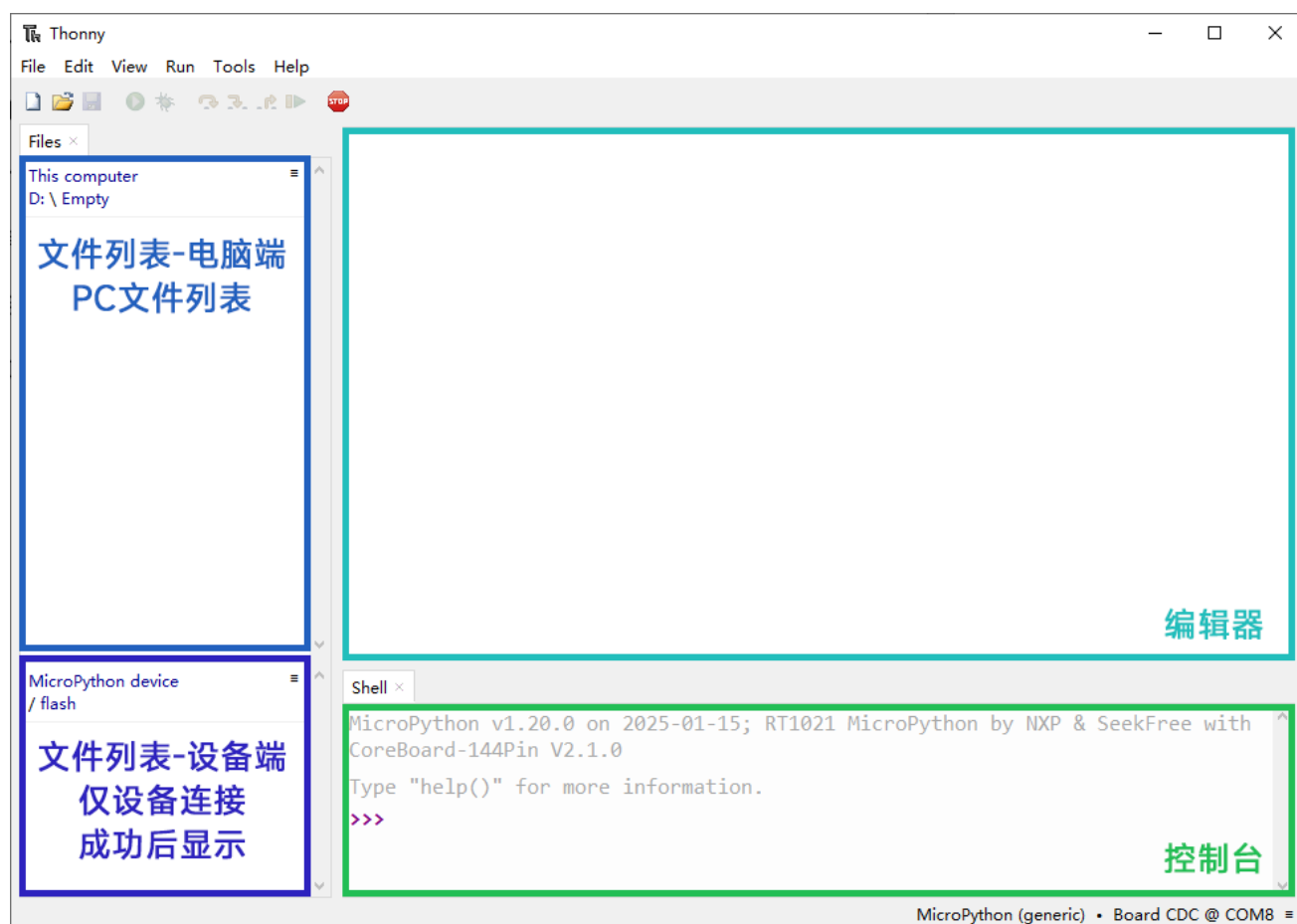
文档并不能直接详尽介绍如何使用 RT1021-MicroPython 核心板完成整个开发流程, 只能作为入门指导和使用手册, 如有良好的修改意见也欢迎各位提出建议。

2.RT1021-MicroPython 核心板连接与脱机启动

2.1.开源 MicroPython 开发工具

进行 MicroPython 开发需要一个代码编辑器、支持 MicroPython 文件协议的文件管理器和一个 REPL 控制台。如果各位有自己熟悉的环境，可以跳过这一个小节。

可以使用开源软件——Thonny，它自带文件系统支持和基于串口连接的 REPL 控制台：

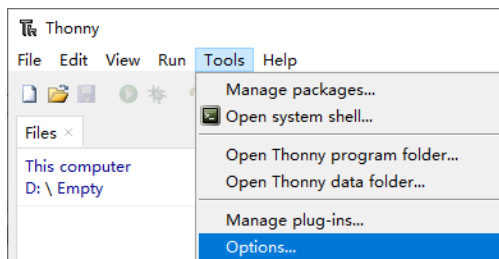


Thonny 软件下载链接：<https://github.com/thonny/thonny/releases>

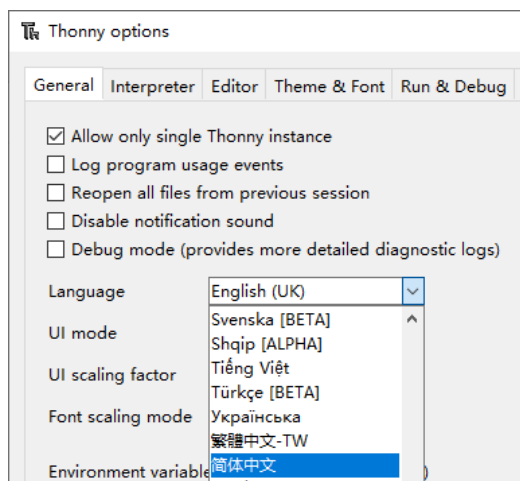
需要使用到一些“上网技巧”访问 GitHub，建议至少使用 V4.1.0 及以上版本的 Thonny，否则与本说明书中使用的版本的界面、功能可能会有较大的差异。

Thonny 源码开源链接：<https://github.com/thonny/thonny>

可以通过工具栏的 Tools->Options...打开设置：

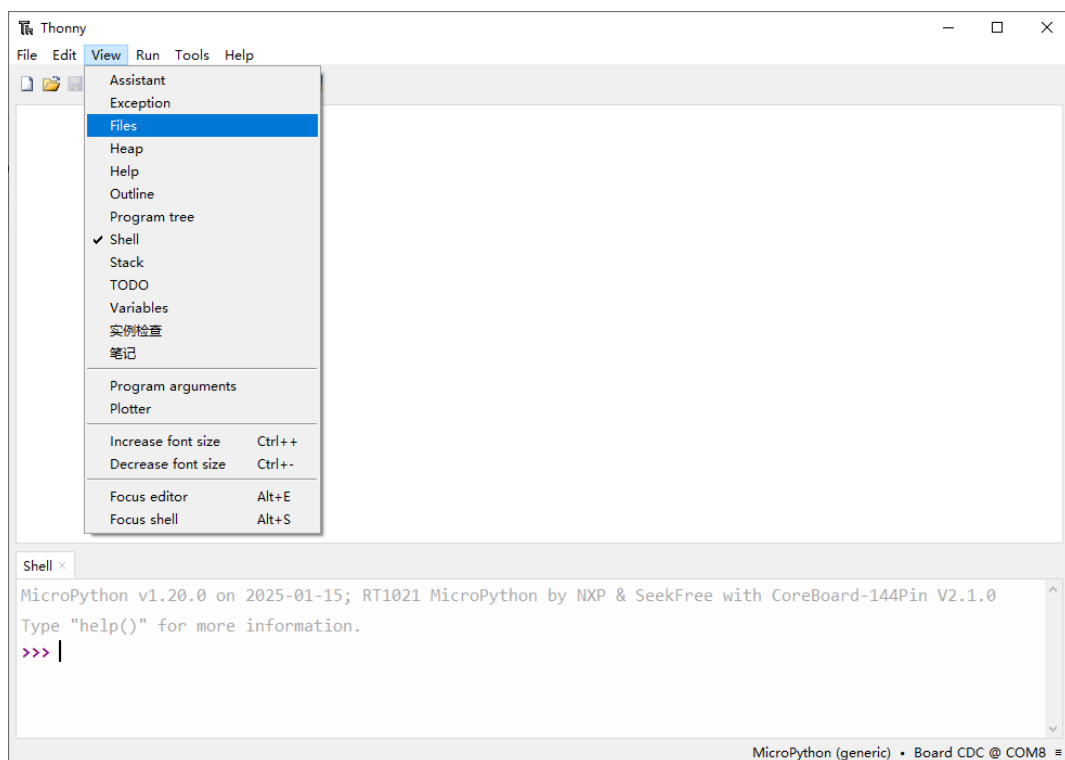


然后在 General 选项卡下的 Language 下拉框选择简体中文，将软件设置为中文模式：



设置完成后需要重启软件生效。

如果界面没有显示文件管理器，可以通过工具栏 View->Files 打开文件管理视图。



2.2.RT1021-MicroPython 核心板连接电脑

使用前，请务必先检查核心板是否正常，**确保没有水滴、金属丝落在核心板上导致短路！**

并且尽量做好绝缘防护（可以给核心板上表面裸露金属部分贴绝缘胶带进行保护）。如果是秋冬季，**请尽量做好防静电措施！避免对板子放电导致静电击穿损坏！**

任何电路板都不可以直接放置在金属桌面、笔记本电脑的金属面板上！

任何电路板都不可以直接放置在金属桌面、笔记本电脑的金属面板上！

任何电路板都不可以直接放置在金属桌面、笔记本电脑的金属面板上！

确保以上注意事项都符合，就可以使用 Type-C 数据线（**确保使用的是数据线，而不是单纯的充电线！**）将 RT1021-MicroPython 核心板连接到 PC 的 USB 口：

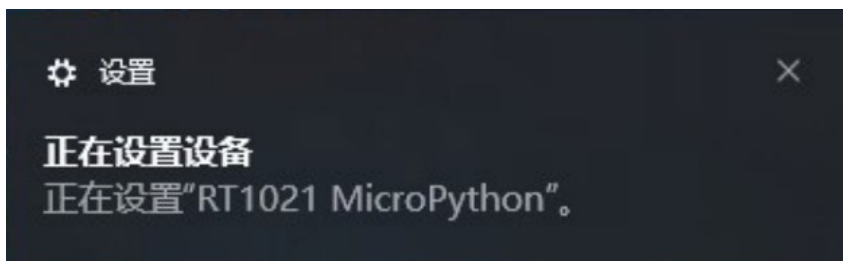


部分笔记本电脑可能没有足够多的 USB 接口而使用 USB 拓展坞，需要注意**有的拓展坞由于版本较老、芯片落后而导致无法正常枚举设备**，如果遇到这类问题，请自行检查并确保 USB 拓展坞能够正常工作。

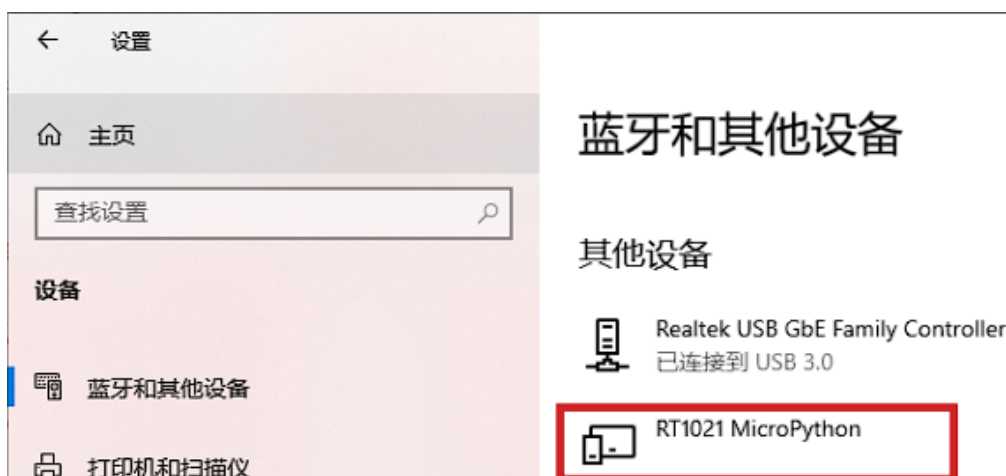
当连接正常时，系统会有 USB 设备插入的提示音，同时在设备管理器中会新增一个串口设备，在说明书使用的测试 PC 上它的 COM 号是“COM8”，在不同电脑上其编号会有差异：



过一段时间，设备会完成枚举并识别具体信息，可能会在右下角弹出设备提示：



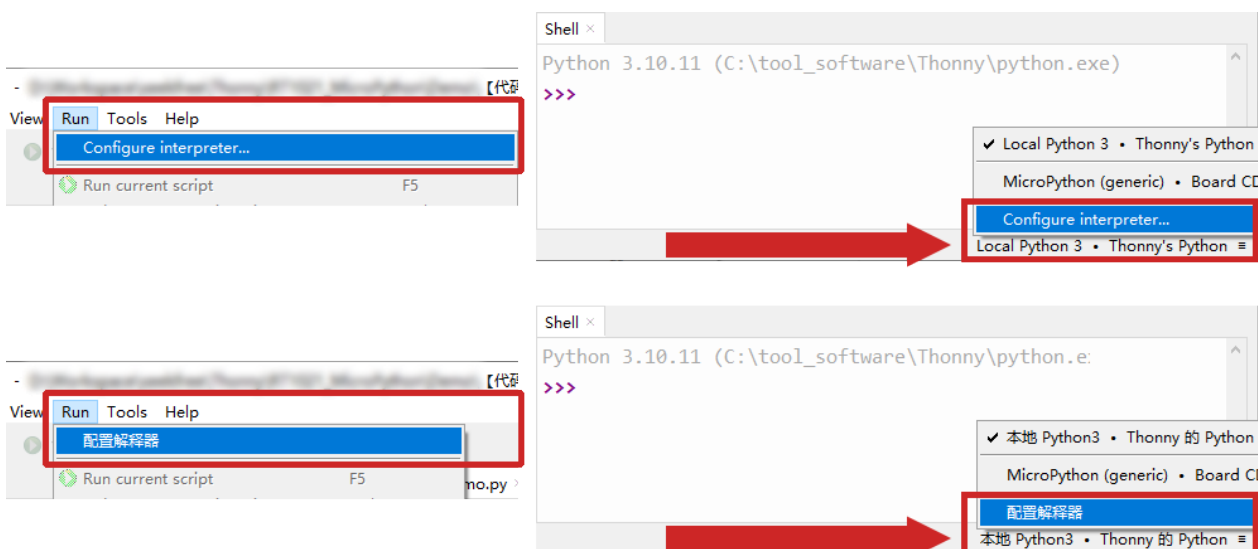
或者可以打开系统设置，找到蓝牙与其他设备，可以在其他设备下找到它：



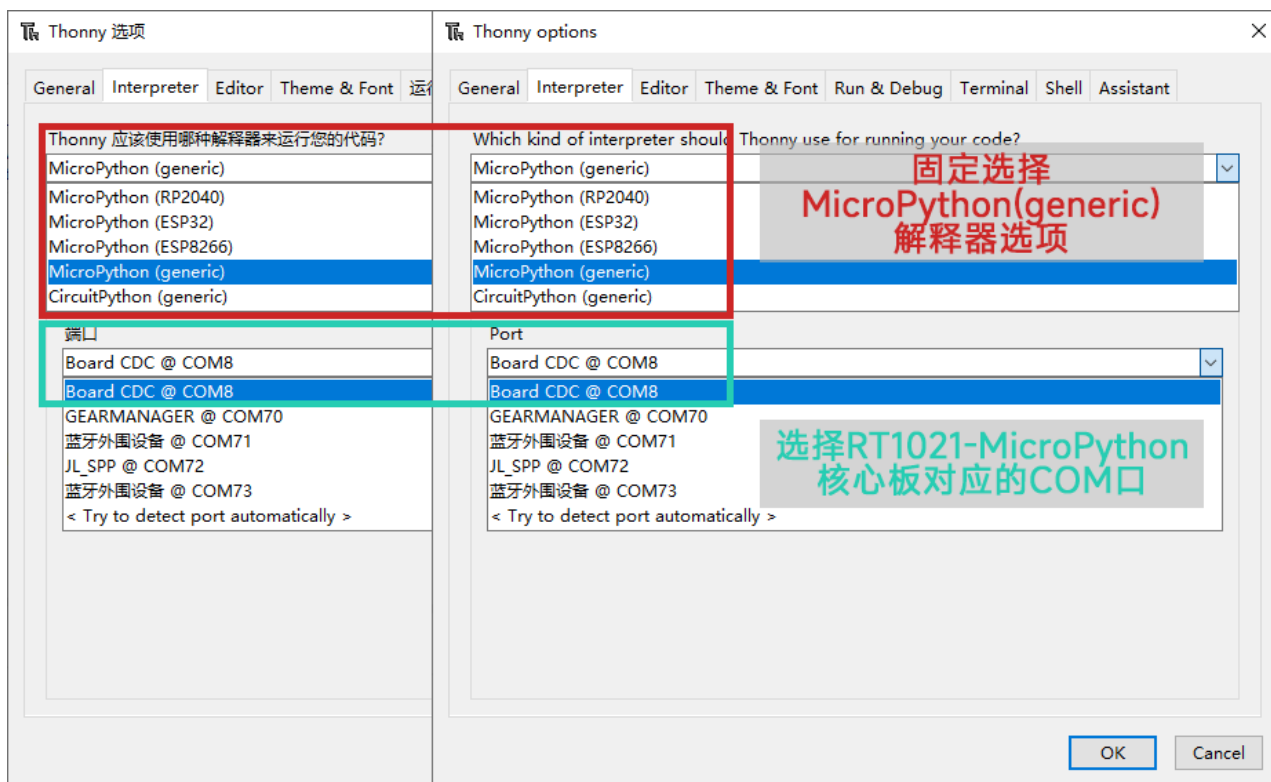
能够识别到 COM 号并显示在其他设备中，就证明设备连接正常。

2.3.Thonny 连接 RT1021-MicroPython 核心板

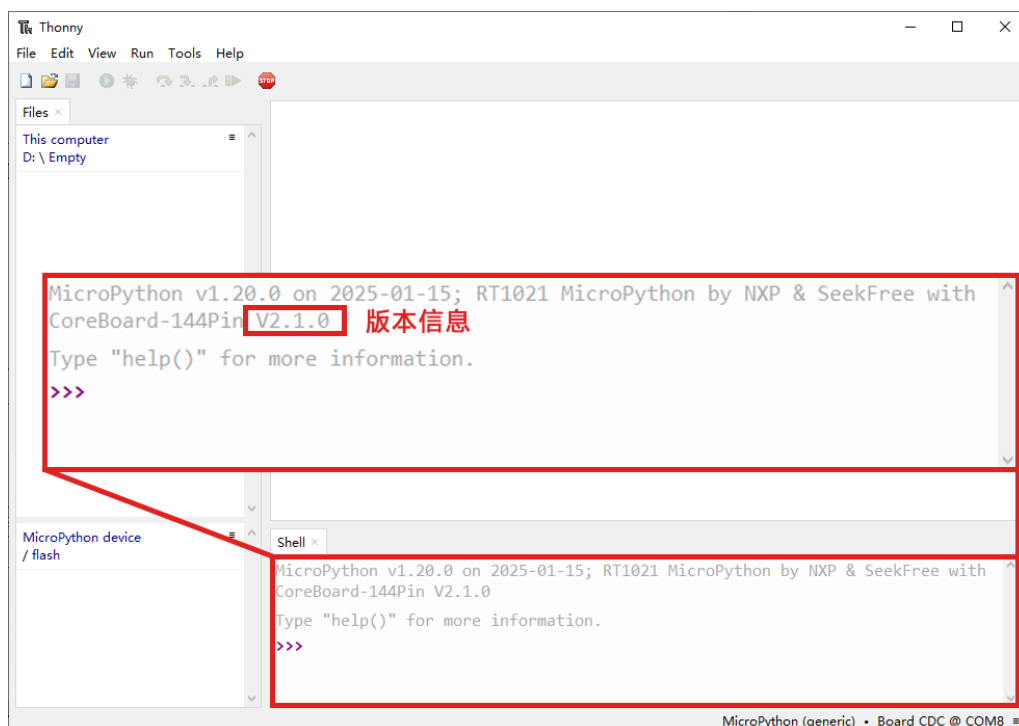
打开 Thonny，打开菜单栏的“Run->Configure Interpreter”（英文模式）/“运行->解释器配置”（中文模式）选项，或者点击窗口右下角的解释器切换按键：



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



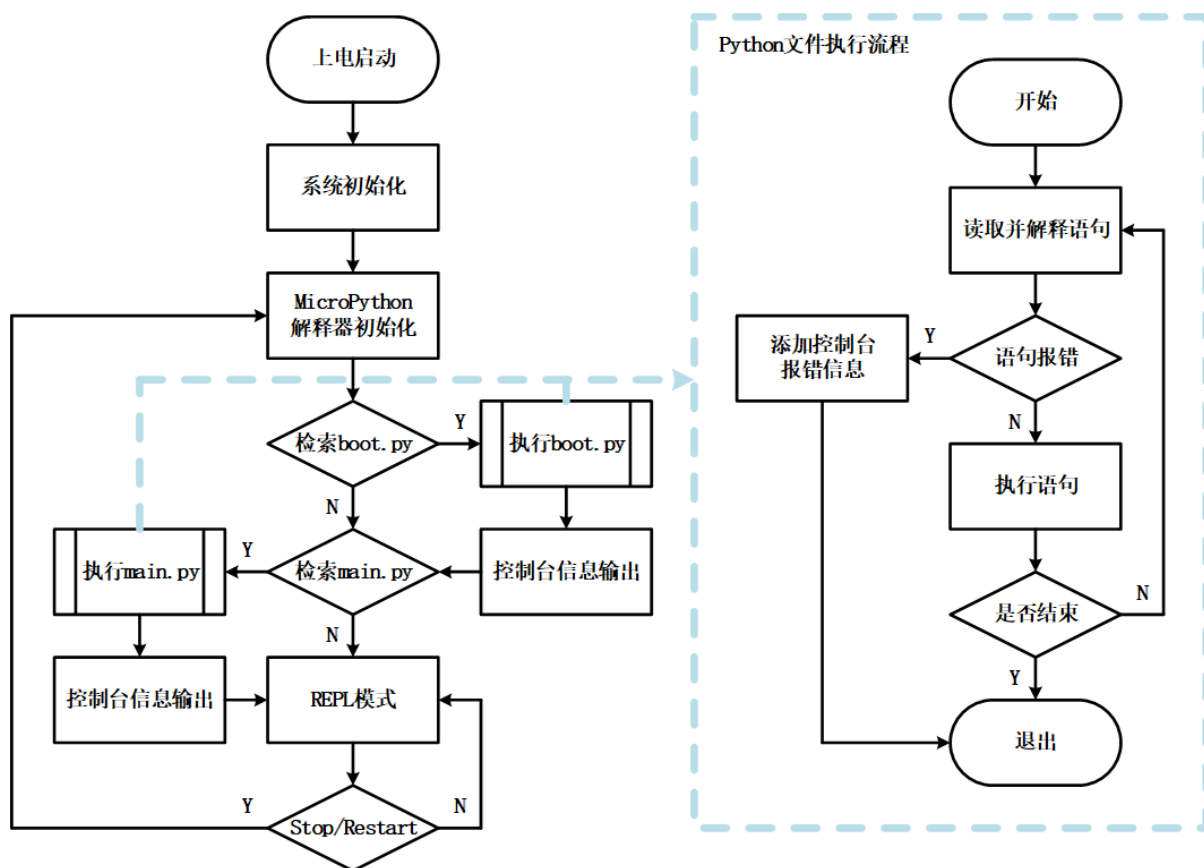
在 RT1021-MicroPython 核心板单独连接电脑，不连接任何传感器、不插在学习板或者自己主板上时，并且 RT1021-MicroPython 核心板中并未保存任何 Python 文件情况下，Thonny 将通过对应的 COM 口连接到 RT1021-MicroPython 核心板并在控制台输出固件信息：



2.4.RT1021-MicroPython 核心板启动顺序说明

由于核心板没有提供额外的 SD 卡插槽，也不支持 USB 连接作为 U 盘进行文件读写，而是使用核心板自带的 Flash 芯片进行文件存储。因此它只能通过 MicroPython 的文件管理协议来进行文件系统操作，这使得它必须依赖 Thonny 这类工具才能进行文件系统操作。

由于 MicroPython 设备会自动检索文件系统中是否存在 boot.py 与 main.py 文件，如果存在这两个文件，它就会依照顺序打开并执行。

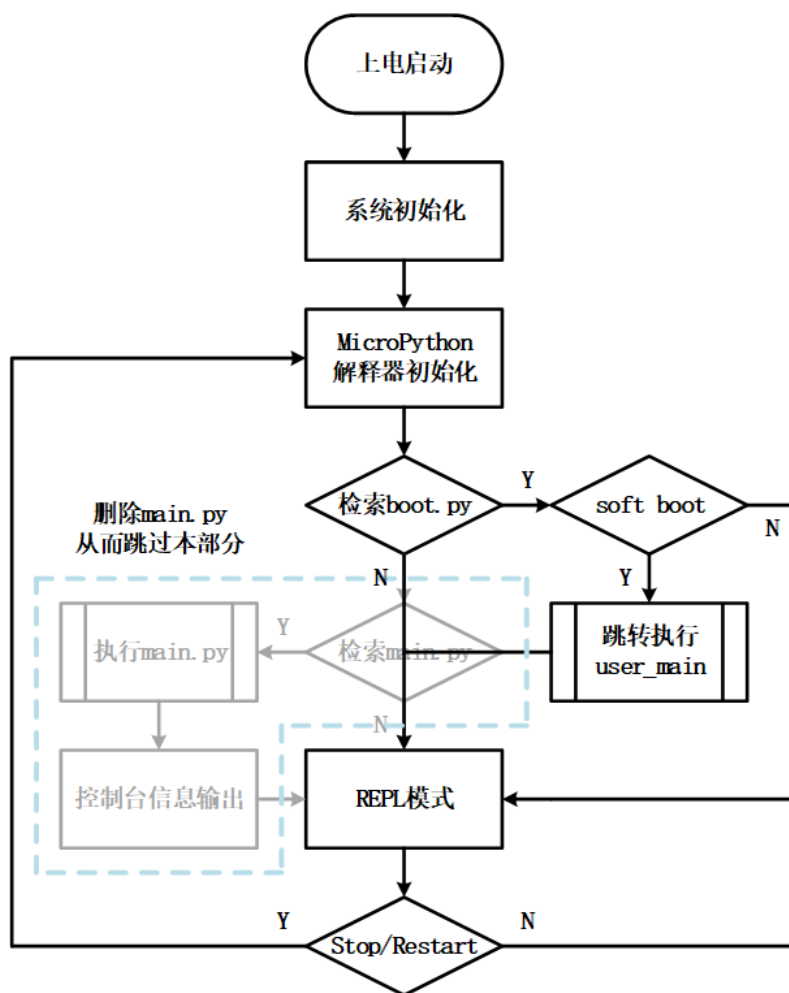


这会导致一个问题，**如果 boot/main.py 文件中存在无法中断的循环或中断，并且没有设置退出条件，就会导致无法退出该文件，也就无法进入 REPL 控制台，从而导致无法通过 Thonny 连接。当无法通过 Thonny 连接时，就无法对其文件系统进行操作，这就形成了死锁，最终核心板无法再操作。**为了避免这种情况，需要进行 soft boot 操作，**建议使用 boot.py 进行软启动操作，并删除 main.py。**通过核心板 IO 引脚来进行 soft boot 的选择启动：

```
from machine import *
import time

time.sleep_ms(50) # 上电启动时间延时
boot_select = Pin('D8', Pin.IN, pull=Pin.PULL_UP_47K) # 选择学习板上的一号拨码开关作为启动选择开关
# 如果拨码开关打开 对应引脚拉低 就启动用户文件
# 如果拨码开关关闭 对应引脚拉高 就跳过用户文件 直接进入 REPL 模式
if boot_select.value() == 0:
    try:
        os.chdir("/flash")
        execfile("user_main.py")
    except:
        print("File not found.")
```

此时核心板的启动流程为：



这样保证复位后一定会通过 **boot.py** 进行 **soft boot** 启动，就算用户文件有无法中断的循环或中断，也可以通过 **boot.py** 选择跳过执行用户文件，直接进入下一步。因为文件系统中检索不到 **main.py** 就会直接进入 **REPL** 模式，就可以完成 Thonny 连接并对文件进行操作了。

2.5.使用 Thonny 进行文件操作

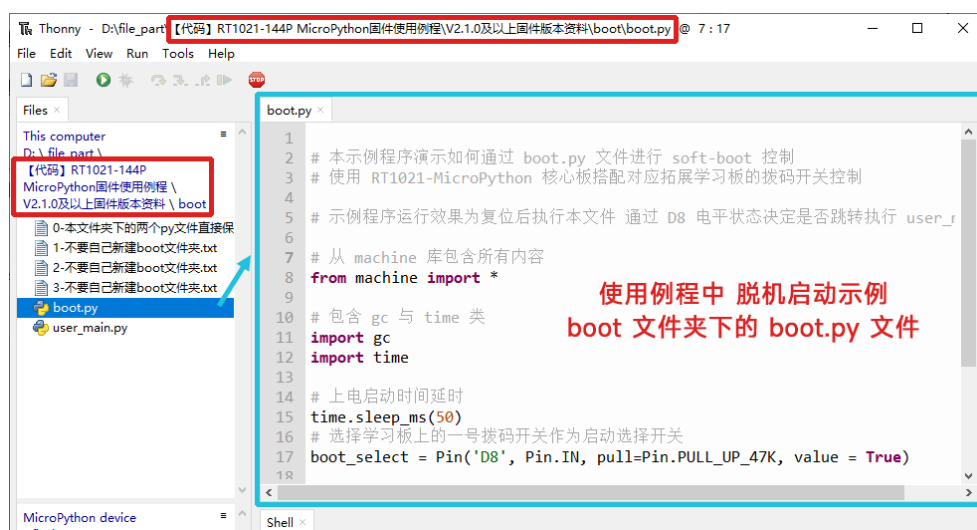
进行文件系统操作时，请务必确认：

使用学习板或主板保证供电稳定，否则掉电导致文件系统异常无法启动。

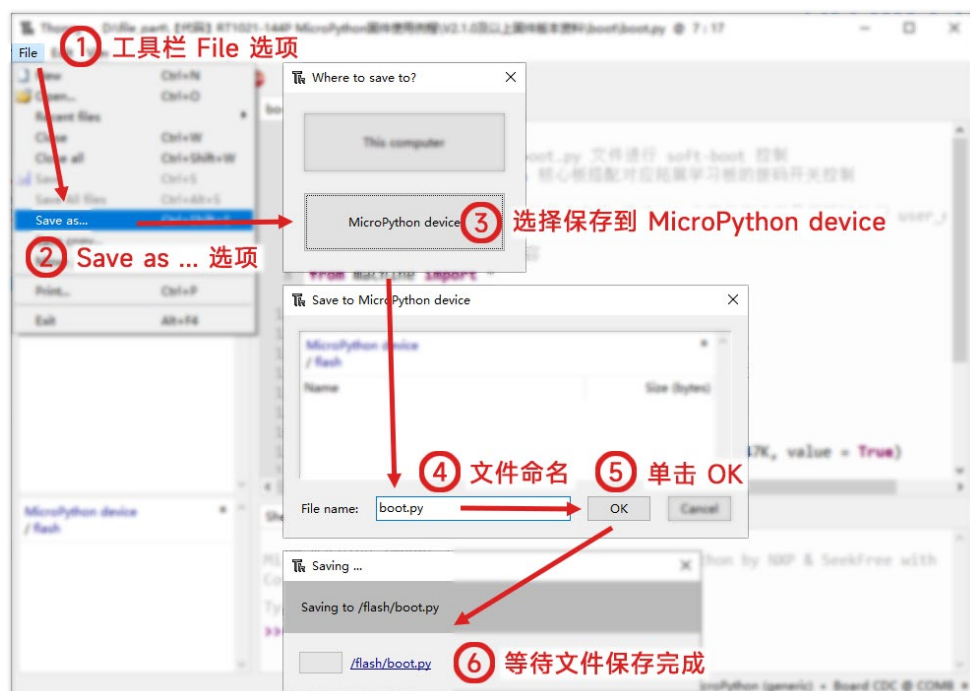
使用学习板或主板保证供电稳定，否则掉电导致文件系统异常无法启动。

使用学习板或主板保证供电稳定，否则掉电导致文件系统异常无法启动。

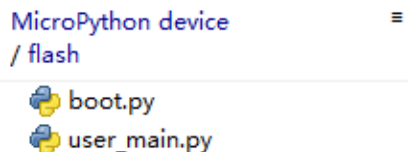
通过 Thonny 连接板子，再打开资料附带的 boot/boot.py:



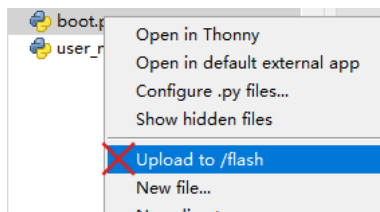
选择工具栏 File->save as...在弹窗中选择 MicroPython device 将 boot 文件保存：



用同样的方式将 user_main.py 保存到 RT1021-MicroPython 核心板：



需要注意的是，**禁止使用 Upload 操作**，因为 Upload to /flash 操作支持并不完善，可能会导致文件系统操作超时，此时关闭窗口可能会导致文件错误！

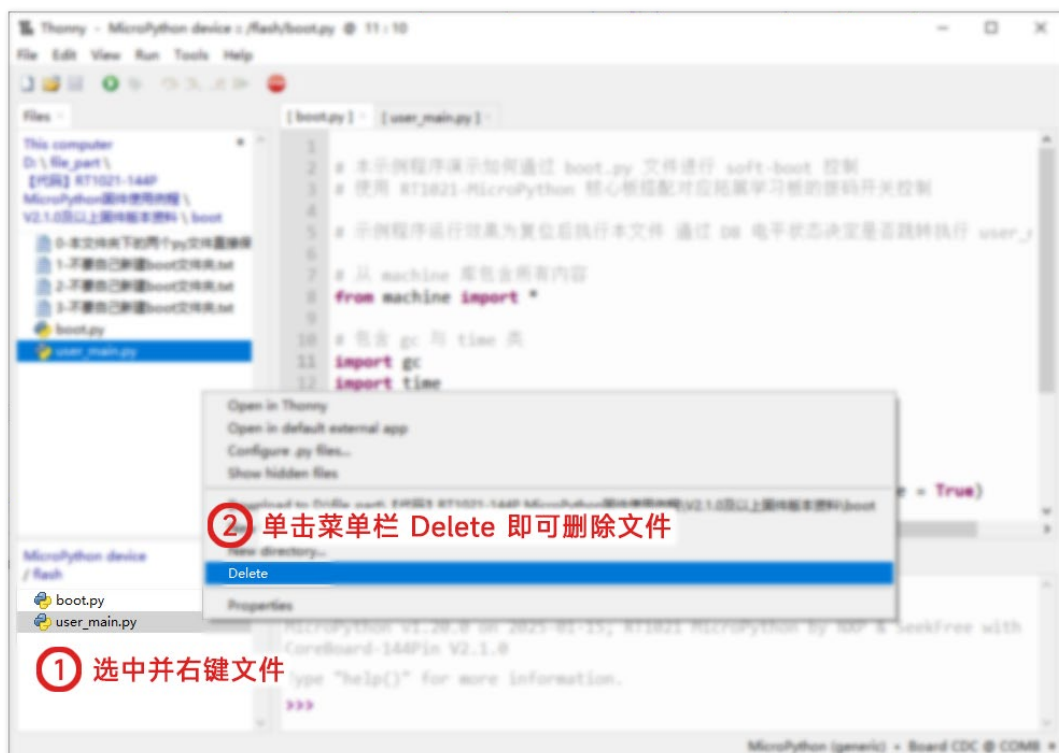


禁止使用Upload

修改文件后直接使用 Ctrl + S 或者单击工具栏的保存按钮即可：



删除文件只需要右键对应的文件，在弹出的菜单中选择对应的选项即可：



2.6.脱机运行示范

使用 boot 示例中的 boot.py 与 user_main.py 文件，将文件保存到核心板中后，**将 D8 对应的拨码开关拨到 ON**，此时可以用万用表检测到 D8 引脚对应电为低，双击打开 MicroPython device/flash/boot.py 文件，并通过 Thonny 运行，会发现它可以自动运行 user_main.py 了：



随后按下核心板复位按键，会发现板子会自动运行 user_main.py 开始闪烁 LED。

2.7.注意事项与常见问题

请务必在使用学习板或主板供电的情况下，再进行文件保存、修改、删除的操作！

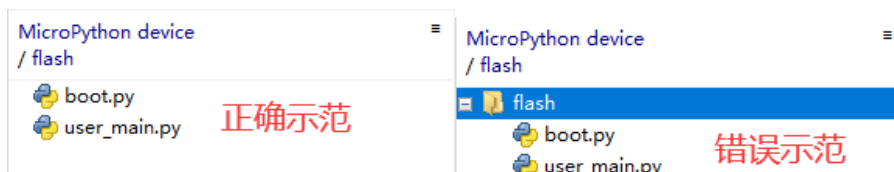
请务必在使用学习板或主板供电的情况下，再进行文件保存、修改、删除的操作！

请务必在使用学习板或主板供电的情况下，再进行文件保存、修改、删除的操作！

这是为了保证在文件操作过程中不会因为 Type-C 松动导致断电，从而使得 Flash 芯片操作异常，导致固件覆写损坏或者文件系统异常，最终使得核心板无法正常启动、使用。

问题一：如果新建了 boot 文件，但它不会脱机运行。

那么检查位置是否正确，名称是否正确无误，拨码开关是否电平正确。



确保正确后检查对应的启动拨码开关是否电平正常。

问题二：代码运行显示“**MemoryError: memory allocation failed, allocating **** bytes.**”

需要检查代码中是否存在冗杂的变量、对象申请，导致内存泄露无法运行。一般常见于 Python 代码有问题，无法在脱机运行后再次连接 Thonny 运行。需要自己优化代码解决，通常来说，可以将全局变量通过 class 封装，通过传递对象的方式使用，避免出现多文件作用域问题导致的内存泄露。

问题三：确保 soft boot 正常工作，但 user_main 依旧无法脱机运行，使用 Thonny 手动运行又没有发现问题，可以正常执行程序。

那么尝试拨码开关跳过执行，然后复位连接 Thonny，再运行 boot 文件，此时可能会看到报错信息，这类情况一般是代码写的有问题，类或变量调用不正确导致的，只会在第一次运行报错，而第二次运行就正常。

3.RT1021-MicroPython 核心板固件说明

固件总共分成三个部分：NXP 实现的 MicroPython 底层接口 machine 库、NXP 为智能车高实时性应用编写的 smartcar 和 display 库、逐飞科技为智能车应用添加的传感器库。

3.1.底层接口 machine 库

3.1.1.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 Pin.x, x = {IN, OUT, OPEN_DRAIN}
# pull   上拉下拉    | 可选参数 Pin.x, x = {PULL_UP, PULL_UP_47K, PULL_UP_22K, PULL_DOWN, PULL_HOLD}
# value  初始电平    | 可选参数 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    | 可选参数 关键字参数 Pin.x, x = {PIN_DRIVE_OFF, PIN_DRIVE_0, ..., PIN_DRIVE_6}

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.x, x = {IRQ_RISING, IRQ_FALLING}
# hard    应用模式    | 可选参数 可用值为 False True
```

3.1.2.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(pin)
# 构造接口 是标准 MicroPython 的 machine.ADC 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
from machine import ADC
adc = ADC('B22')

# 其余接口:
ADC.read_u16() # 读取当前端口的 ADC 转换值 数据返回范围是 [0, 65535]
```

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。

3.1.3.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 可以传入字符串或者元组 元组引脚必须是硬件互补通道
# freq 工作频率 | 必要参数
# duty_u16 初始脉宽 | 必要参数 关键字输入 范围 [1, 65535]
from machine import PWM
pwm1 = PWM("D4", 13000, duty_u16 = 1) # 传入引脚名称字符串 初始化对应引脚
pwm2 = PWM(("D6", "D7"), 13000, duty_u16 = 1) # 传入 A/B 相引脚字符串元组 则会把这一对引脚初始化为互补通道

# 其余接口:
PWM.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
PWM.freq([value]) # 传入 value 则更新频率设置 否则仅反馈当前频率设置
```

这个接口通常用来控制舵机，或者单个的 PWM 引脚，因为它只能使用单个通道，或者把一对硬件互补引脚初始化成互补输出通道。

也不能单独初始化同个子模块下的 A、B 通道无法单独控制占空比，如果单独初始化同个子模块的 A、B 通道，他们占空比不能独立控制。

3.1.4.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, 7] 总共 8 个 UART 模块
#
#           |   HW-UART |   Logical |   TX |   RX |
#           |   LPUART1 |   id = 0 |   B6 |   B7 |
#           |   LPUART2 |   id = 1 |   C22 |   C23 |
#           |   LPUART3 |   id = 2 |   C6 |   C7 |
#           |   LPUART4 |   id = 3 |   B26 |   B27 |
#           |   LPUART5 |   id = 4 |   B10 |   B11 |
#           |   LPUART6 |   id = 5 |   D20 |   D21 |
#           |   LPUART7 |   id = 6 |   D2 |   D3 |
#           |   LPUART8 |   id = 7 |   D22 |   D23 |

from machine import UART
uart1 = UART(2)

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 |   可选参数 关键字输入 默认 9600
#   bits      数据位数 |   可选参数 默认 8 bits 数据位
#   parity     校验位数 |   可选参数 默认 None {None, 0, 1} 无校验, 偶校验, 奇校验
#   stop       停止位数 |   可选参数 默认 1 bit 停止位

UART.init(9600)

# 其余接口:
buf = UART.read(n)      # 读取 n 字节到 buf
UART.readinto(buf)      # 读取数据字节到 buf
UART.write(buf)         # 将 buf 内容通过 UART 发送
UART.any()              # 判断 UART 是否有数据可读取
```

3.1.5.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, 2] 总共 3 个 SPI 模块
#
#           |   HW-UART |   Logical |   SCK |   MOSI |   MISO |   CS0 |
#           |   LPSPI1  |   id = 0 |   B10 |   B12 |   B13 |   B11 |
#           |   LPSPI3  |   id = 1 |   B28 |   B30 |   B31 |   B29 |
#           |   LPSPI4  |   id = 2 |   D0 |   D2 |   D3 |   D1 |

from machine import SPI
spi = SPI(1)

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 电平极性 |   可选参数 默认 0 {0 - 时钟空闲时低电平, 1 - 时钟空闲时高电平}
#   phase     时钟相位 |   可选参数 默认 0 {0 - 第一个时钟沿采样数据, 1 - 第二个时钟沿采样数据}
```

```
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
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
spi.write_readinto(tx_buff, rx_buff) # 输出 tx_buff 长度数据 同时读取数据到 rx_buff 这两个缓冲区必须一样长
```

3.1.6. 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
#           |   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   |
#   freq     传输速率 |   可选参数 默认 400000
from machine import I2C
iic = I2C(1, freq = ...)
```

# 其余接口:	
addr_list = I2C.scan()	# 扫描 IIC 是否有设备从 0x08 到 0x77 输出一个响应的地址列表
rx_byte = I2C.readfrom(addr_list[0], 1, True)	# 读取一个字节数据 True 发送停止信号 False 无停止信号
I2C.readfrom_into(addr_list[0], rx_buff, True)	# 读取 rx_buff 长度数据 True 发送停止信号 False 无停止信号
I2C.writeto(addr_list[0], tx_buff, True)	# 输出 tx_buff 长度数据 True 发送停止信号 False 无停止信号
I2C.writevto(addr_list[0], vectors, True)	# 输出 vectors 矩阵数据 True 发送停止信号 False 无停止信号

```
# 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
```

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。

3.2.NXP 的 smartcar 库

3.2.1.ticker 子模块

由于标准 machine 中的 Timer 类是基于软定时器实现，无法保证回调的实时性，因此 NXP 编写了 ticker 子模块，用于提供一个周期中断以保证可以有一个实时性较强的模块。

```
ticker(id)
# 构造接口 用于构建一个 ticker 对象
# id 通道索引 | 必要参数 RT1021 对应 [0,3] 总共四个通道可选 通道为顺序触发 没有优先级 没有中断嵌套
from smartcar import ticker
pit = ticker(1)
# 其余接口:
def ticker_callback (ticker):      # 定义一个回调函数 传入的参数是 ticker 对象本身
    ...                             # 回调处理代码略
ticker.callback(ticker_callback)    # 设置回调函数 ticker 必须设置一个回调后才能工作
ticker.start(ms)                    # ticker 以 ms 周期运行触发
ticker.stop()                       # ticker 停止
ticker.ticks()                      # 返回当前 ticker 的触发次数
```

但是 Python 本质上是解释型语言，它无法保证强实时性，这意味着无法保证数据采样的周期，这可能会导致一些控制上的问题，例如：编码器采样周期短，而 Python 代码无法保证强实时性，就会导致采样的编码器数据上下浮动。因此 NXP 提供了 caputer_list 用于进行底层自动采集，这样就可以保证用户的数据采集始终可以保持稳定的触发采集周期。

```
ticker.capture_list(obj1, obj2, ....) # 将输入对象绑定到底层自动捕获 最多支持 8 个对象 绑定顺序就是采集顺序
# 支持的对象类型为 smartcar 下的 ADC_Group / encoder
# seekfree 下的 DL1X / IMU660RX / IMU9363RX / KEY_HANDLER / TSL1401
```

需要注意的底层采集也是需要消耗时间的！例如，尽管 ADC_Group、encoder 模块的采集并不会过多消耗时间，但它也会占用一定的 ticker 模块的中断时间，而多个 TSL1401 的采集则会占用较长的采集时间。**需要自行考虑使用合适的 ticker 周期！**保险起见不要低于 5ms。

3.2.2.ADC_Group 子模块

通过 Python 代码调用 ADC 对象进行信号转换时需要通过解释器解释、对象查找、接口

调用、底层转换、数据转换等步骤，这使得需要转换多个 ADC 数据时会浪费很多时间。

因此 NXP 提供了 ADC_Group 子模块，使得用户可以将同一个 ADC 模块下的引脚编入一个组中，通过组序列转换的方式一次性将多个通道数据完成转换，可以提高工作效率。

```
ADC_Group(id)
# 构造接口 用于构建一个 ADC_Group 对象
# id      模块索引 | 必要参数 RT1021 对应有 [1,2] 总共两个模块可选
from smartca import ADC_Group
adc = ADC_Group(1)

ADC_Group.init(id, period = ADC_Group.PMODE3, average = ADC_Group.AVG16)
# ADC_Group 参数设置 参数说明
# id      模块索引 | 必要参数 RT1021 对应有 [1,2] 总共两个模块可选
# period  采样周期 | 可选参数 关键字输入 默认 ADC_Group.x, x = {PMODE0, PMODE1, PMODE2, PMODE3}
# average 均值选项 | 可选参数 关键字输入 默认 ADC_Group.x, x = {AVG1, AVG4, AVG8, AVG16, AVG32}

# 其余接口:
ADC_Group.addch(pin_name)      # 输入引脚字符串名称 向组中添加对应名称的通道
ADC_Group.capture()            # 触发一次 ADC_Group 的转换
ADC_Group.get()                # 将 ADC_Group 转换结果更新到数据缓冲区 并返回为一个列表
ADC_Group.read()               # 触发一次转换 并将转换结果更新到数据缓冲区 返回为一个列表
```

不过需要注意的是，**对应的 ADC_Group 子模块只能添加同个 ADC 模块下的通道**，也就是说，ADC_Group1 子模块只能添加 ADC1 的通道，ADC_Group2 子模块只能添加 ADC2 通道，两个组并不能相互混用。

3.2.3.encoder 子模块

由于标准 machine 中并没有 enc 类，所以无法进行编码器采集。因此 NXP 提供了 encoder 子模块用于进行编码器采集，支持正交编码器和带方向的增量式编码器。

```
encoder(PhaseA, PhaseB, invert = False)
# 构造接口 用于构建一个 encoder 对象
# PhaseA  引脚名称 | 必要参数 引脚名称字符串 编码器 A 相或 PLUS 引脚
# PhaseB  引脚名称 | 必要参数 引脚名称字符串 编码器 B 相或 DIR 引脚
# invert   模块索引 | 可选参数 是否反向 可以通过这个参数调整编码器旋转方向数据极性
from smartca import encoder
enc1 = encoder("D13", "D14")
enc2 = encoder("D15", "D16", True)
```

```
# 其余接口:
encoder.capture()      # 触发一次 encoder 的采集请求
encoder.get()          # 将 encoder 转换结果更新到数据缓冲区
encoder.read()         # 触发一次转换 并将转换结果更新到数据缓冲区
```

3.3.NXP 的 display 库

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 | 必要参数 关键字输入 选择屏幕所用的 SPI 接口索引
# BAUDRATE 通信速率 | 必要参数 关键字输入 SPI 的通信速率 最高 60MHz
# DC_PIN 命令引脚 | 必要参数 关键字输入 一个 Pin 实例
# RST_PIN 复位引脚 | 必要参数 关键字输入 一个 Pin 实例
# LCD_TYPE 屏幕类型 | 必要参数 关键字输入 目前仅支持 LCD_Drv.LCD200_TYPE

from machine import *
from display import *

# 定义片选引脚 拉高拉低一次 CS 片选确保屏幕通信时序正常
cs = Pin('C5', Pin.OUT, pull=Pin.PULL_UP_47K, value=1)
cs.high()
cs.low()

# 定义控制引脚 需要注意的是 blk 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
rst = Pin('B9', Pin.OUT, pull=Pin.PULL_UP_47K, value=1)
dc = Pin('B8', Pin.OUT, pull=Pin.PULL_UP_47K, value=1)
blk = Pin('C4', 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
drv = LCD_Drv(SPI_INDEX=1, BAUDRATE=6000000, DC_PIN=dc, RST_PIN=rst, LCD_TYPE=LCD_Drv.LCD200_TYPE)
lcd = LCD(drv) # 新建 LCD 实例

LCD.color(pcolor, bgcolor) # 修改 LCD 的前景色与背景色
# pcolor 前景色 | 必要参数 RGB565 格式
# bgcolor 背景色 | 必要参数 RGB565 格式
lcd.color(0xFFFF, 0x0000)

LCD.mode(dir) # 修改 LCD 的显示方向
# dir 显示方向 | 必要参数 [0:竖屏, 1:横屏, 2:竖屏 180 旋转, 3:横屏 180 旋转]
lcd.mode(2)

LCD.clear([color]) # 清屏显示 不输入参数按照背景颜色清屏 输入参数则将按照参数清屏并更新背景颜色
# color 清屏颜色 | 非必要参数 RGB565 格式 输入参数则更新背景色并清屏
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 | 必要参数
# y      起点纵轴 | 必要参数
# str    字符数据 | 必要参数 字符串数据
# color  显示颜色 | 可选参数 RGB565 格式 显示字符颜色
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 | 必要参数
# y      起点纵轴 | 必要参数
# x      终点横轴 | 必要参数
# y      终点纵轴 | 必要参数
# 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
lcd.line(0,84,200,16 + 84,color=0xFFFF,thick=1)
lcd.line(200,84,0,16 + 84,color=0x3616,thick=3)
```

特别提示：不论你需要显示什么数据，对于 Python 来说，它们都是字符串！全部都使用

LCD.str(x, y, str, color)接口进行显示！**

跟 print 函数一样，通过字符串格式化将数据整合为一个字符串对象即可！

跟 print 函数一样，通过字符串格式化将数据整合为一个字符串对象即可！

跟 print 函数一样，通过字符串格式化将数据整合为一个字符串对象即可！

3.4.逐飞科技的 seekfree 库

由于 Python 是解释型语言，使用它来进行底层接口通信实现传感器驱动会非常的浪费性能，并且耗时会很长。因此基于 NXP 的 smartcar 库下 sensor 框架，逐飞科技提供了几个模块方便使用与调试（所用到的引脚暂时不可随意修改，使用固定的引脚）。

3.4.1.MOTOR_CONTROLLER 子模块

用于电机驱动，因为原生的 PWM 模块只能将 RT1021 的同个 PWM 子模块下的 A/B 相初始化为互补通道，而无法独立控制。但电机驱动通常需要两个独立控制的 PWM 通道，或者一个 PWM 通道一个 IO 信号。所以我们提供了这个电机驱动控制子模块，便于电机驱动控制。

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]）。

驱动方式	PWM+DIR 组合驱动 (DRV8701)	PWM*2 组合驱动 (HIP4082)
引脚组合	MOTOR_CONTROLLER.PWM_C24_DIR_C26	MOTOR_CONTROLLER.PWM_C24_PWM_C26
	MOTOR_CONTROLLER.PWM_C25_DIR_C27	MOTOR_CONTROLLER.PWM_C25_PWM_C27

可以用于直接驱动 DRV8701 单/双驱或者 HIP4082 单/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 | 必要参数 [PWM_C24_DIR_C26, PWM_C25_DIR_C27,
#            |          PWM_C24_PWM_C26, PWM_C25_PWM_C27]
# freq       信号频率 | 必要参数 PWM 信号的频率 范围是 [1 - 100000]
# duty       占空比值 | 可选参数 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     反向设置 | 可选参数 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

from seekfree import MOTOR_CONTROLLER
motor = MOTOR_CONTROLLER(MOTOR_CONTROLLER.PWM_C24_DIR_C26, 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

# 其余接口：
MOTOR_CONTROLLER.help()      # 可以直接通过类调用 也可以通过对象调用 输出模块的使用帮助信息
MOTOR_CONTROLLER.info()     # 通过对象调用 输出当前对象的自身信息
    
```


3.4.2.BLDC_CONTROLLER 子模块

用于直接驱动负压风扇无刷电调，支持学习板上的无刷电机电调信号接口 C26/C27。不过需要注意的是。

```
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 | 必要参数 可选参数为 [PWM_C26, PWM_C27]
# freq       信号频率 | 可选参数 关键字参数 PWM 频率 范围 50-300 默认 50
# highlevel_us 高电平值 | 可选参数 关键字参数 初始的高电平时长 范围 [1000-2000] 默认 1000
from seekfree import BLDC_CONTROLLER
bldc1 = BLDC_CONTROLLER(BLDC_CONTROLLER.PWM_C26, freq=300, highlevel_us = 1000)

BLDC_CONTROLLER.highlevel_us([highlevel_us])    # 更新或获取高电平时间值
# highlevel_us 高电平值 | 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]

# 其余接口：
BLDC_CONTROLLER.help()        # 可以直接通过类调用 也可以通过对象调用 输出模块的使用帮助信息
BLDC_CONTROLLER.info()        # 通过对象调用 输出当前对象的自身信息
```

3.4.3.KEY_HANDLER 子模块

四个按键（D20/ D21/ D22/ D23）的驱动，带消抖、长短按检测。默认为短按松发（按下后松开触发），默认消抖时长为 100ms，长按检测 500ms 生效。

```
KEY_HANDLER(period)          # 构造接口 学习主板上的按键驱动
# period  扫描周期 | 必要参数 按键的扫描周期 一般配合填写 Tickter 的运行周期
from seekfree import KEY_HANDLER
key = KEY_HANDLER(10)

KEY_HANDLER.capture()         # 执行一次按键状态扫描
KEY_HANDLER.get()             # 输出当前四个按键状态
print("key: " + str(key.get()))
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

# 其余接口：
KEY_HANDLER.help()           # 可以直接通过类调用 也可以通过对象调用 输出模块的使用帮助信息
KEY_HANDLER.info()           # 通过对象调用 输出当前对象的自身信息
```

默认按键连接为释放高电平，按下低电平。

3.4.4.IMU660/963RX 子模块

用于驱动 IMU660/963RX (X=[A, B, ...]) 系列的六轴或九轴姿态传感器模块，使用固定引脚 (SCK-B10 / MOSI-B12 / MISO-B13 / CS-B11)，推荐挂载在 Ticker 下自动采集。

```
IMUxxxRX([capture_div]) # 构造接口 支持使用主板上的 IMU 接口连接 IMU660RA / IMU660RB 或者 IMU963RA 模块
# capture_div 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
from seekfree import IMUxxxRX
imu = IMUxxxRX ()

IMUxxxRX.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
IMUxxxRX.get()     # 输出当前采集缓存的 IMU 数据
IMUxxxRX.read()    # 立即进行一次 capture 并输出缓存数据

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/B 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

# 其余接口:
IMUxxxRX.help()    # 可以直接通过类调用 也可以通过对象调用 输出模块的使用帮助信息
IMUxxxRX.info()    # 通过对象调用 输出当前对象的自身信息
```

可以通过 IMUxxxRX.help()查看当前版本的固件到底支持哪些模块型号。

3.4.5.DL1X 子模块

用于驱动 DL1X (X=[A, B, ...]) 系列 ToF 红外激光测距模块，使用固定引脚 (SCL-C22 / SDA-C23 / XS-B4)，推荐挂载在 Ticker 自动采集，但需要注意采集频率不能高于模块频率。

```
DL1X([capture_div]) # 构造接口 支持使用主板上的 ToF 接口连接 DL1A 1.2M@30Hz / DL1B 1.4M@100Hz 模块
# capture_div 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
from seekfree import DL1X
tof = DL1X ()

DL1X.capture() # 执行一次 DL1X 数据采集触发 达到触发数时执行采集并将数据缓存
DL1X.get()     # 输出当前采集缓存的 DL1X 数据
DL1X.read()    # 立即进行一次 capture 并输出缓存数据

# 其余接口:
DL1X.help()    # 可以直接通过类调用 也可以通过对象调用 输出模块的使用帮助信息
DL1X.info()    # 通过对象调用 输出当前对象的自身信息
```

3.4.6.TSL1401 子模块

用于驱动 TSL1401 红孩儿线阵 CCD 模块，固定使用 B29/B31 两个模拟输入引脚，占用 ADC2 模块，以及 B3/C8 作为控制引脚。推荐挂载在 Ticker 下自动采集，可以调整采集分频控制曝光时间，不过建议保持默认分频，选择合适的 ticker 周期作为曝光周期。

```
TSL1401([capture_div])          # 构造接口 学习主板上的线阵 CCD 接口
#   capture_div   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
from seekfree import TSL1401
ccd = TSL1401()

TSL1401.set_resolution(resolution) # 设置 CCD 的转换精度
#   resolution   接口索引   |   必要参数 TSL1401.x , x = {RES_8BIT, RES_12BIT}
ccd.set_resolution(TSL1401.RES_12BIT)

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
TSL1401.get(index)                # 输出当前采集缓存的 TSL1401 数据
TSL1401.read(index)               # 立即进行一次 capture 并输出缓存数据
#   index         接口索引   |   必要参数 范围 [0, 3] 对应学习板上 CCD1/2/3/4 接口

# 其余接口：
TSL1401.help()                    # 可以直接通过类调用 也可以通过对象调用 输出模块的使用帮助信息
TSL1401.info()                    # 通过对象调用 输出当前对象的自身信息
```

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：

```
# 通过 wave 接口显示数据波形 (x,y,width,height,data,data_max)
#   x             横轴坐标   |   必要参数 起始显示 X 坐标
#   y             纵轴坐标   |   必要参数 起始显示 Y 坐标
#   width         显示宽度   |   必要参数 等同于数据个数
#   height        显示高度   |   必要参数 实际显示高度 因为数据可能比屏幕高度值大
#   data          波形数据   |   必要参数 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   |   可选参数 关键字参数 TSL1401.RES_8BIT - [0, 255] TSL1401.RES_12BIT - [0, 4095]
LCD.wave(0, 0, 128, 64, ccd_data1, max = 255)

# 例：
...                               # 代码略
ccd_data1 = ccd.get(0)
lcd.wave(0, 0, 128, 64, ccd_data1, max = 4095)
...                               # 代码略
```

3.4.7. 传感器子模块与 ticker 的 caputer_list 关联采集说明

在前面《3.2.3.ticker 子模块》中提到过“NXP 提供了 caputer_list 用于进行底层自动采集”。

在实际应用时，需要考虑各模块的采集周期、模块采集耗时，还要考虑自己的回调代码需要消耗多少时间。这里给出一个参考：

首先使用 4 个 CCD 模块，一个 IMU 模块，一个 ToF 模块，并且使用 KEY_HANDLER 子模块，再加上两个轮子各自一个 encoder 模块，总共六个需要挂载自动采集的模块。

然后检查这几个模块中更新频率最低的，是 ToF 模块，如果使用 DL1B 则 100Hz 更新频率，10ms 的周期，如果使用 DL1A 模块则 30Hz 更新频率，要至少 33ms 的周期。这里就考虑使用 10ms 作为 ticker 的触发周期。使用 DL1B 时正好匹配频率周期，使用 DL1A 时进行 4 分频采集即可，等效 25Hz 频率 40ms 周期。

这样 TSL1401、IMU 可以直接不分频挂载，一来是 IMU 的输出频率高于 100Hz（陀螺仪数据输出频率 200+Hz），二来 CCD 的曝光时间差不多正好，也兼顾图像刷新帧率，亮度不够可以补光。

```
from machine import *
from smartcar import *
from seekfree import *

ccd = TSL1401()           # 实例化模块对象 采集不分频 采集周期为 10ms (ticker 周期)
imu = IMU963RA()          # 实例化模块对象 采集不分频 采集周期为 10ms (ticker 周期)
tof = DL1B()              # 实例化模块对象 采集不分频 采集周期为 10ms (ticker 周期)

key = KEY_HANDLER(10)     # 这里填入 10ms (ticker 周期) 的周期值

encoder_1 = encoder("D15", "D16", True) # 实例化编码器对象 反向
encoder_2 = encoder("D13", "D14")       # 实例化编码器对象

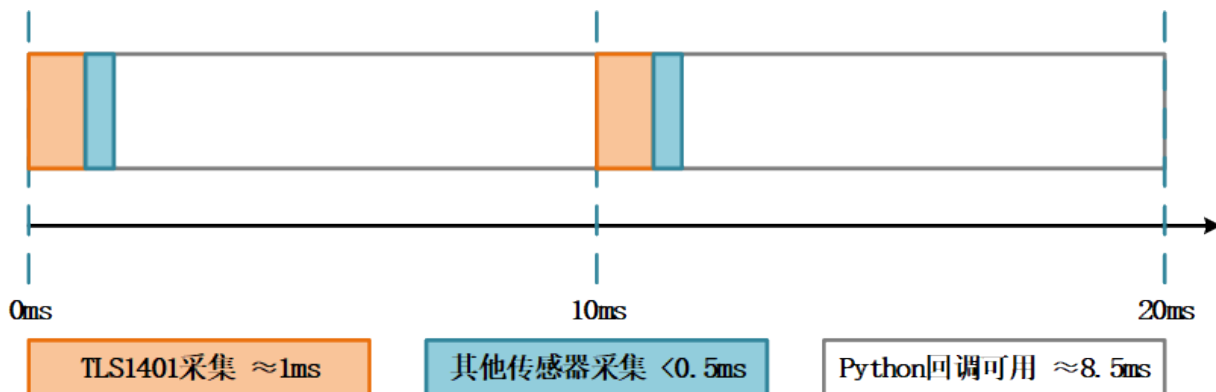
ticker_flag = False
def time_pit_handler (time):           # 定义一个回调函数 需要一个参数 这个参数就是 ticker 实例自身
    global ticker_flag                 # 需要注意的是这里得使用 global 修饰全局属性
    ticker_flag = True                 # 否则它会新建一个局部变量
    ...                                # 回调处理代码略
```

```

pit = ticker(1)                                # 实例化 ticker 模块
pit.callback(time_pit_handler)                 # 绑定回调函数
# 将各模块挂载到 ticker 的 capture_list 中 绑定顺序就是采集顺序
pit.capture_list(ccd, imu, tof, key, encoder_1, encoder_2)
pit.start(10)                                  # 以 10ms 周期启动 ticker 模块

```

那么此时 ticker 底层触发与采集状态大致情况为 (此处时间仅为参考并非真实采样时间):



如果使用分频去调节曝光或者采集, 就可能会导致 Python 的回调函数可用时间不确定,

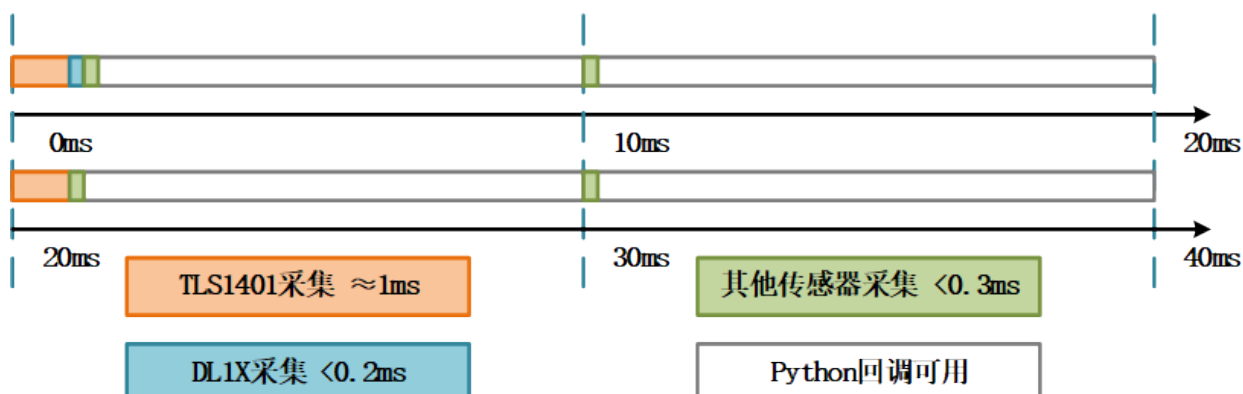
例如如下的代码设置, 修改了 TSL1401 和 DL1B 的采集分频:

```

... # 代码略
ccd = TSL1401(2) # 实例化模块对象 采集 2 分频 采集周期为 2 * 10ms (ticker 周期)
imu = IMU963RA() # 实例化模块对象 采集不分频 采集周期为 10ms (ticker 周期)
tof = DL1B(4) # 实例化模块对象 采集 4 分频 采集周期为 4 * 10ms (ticker 周期)
... # 代码略

```

那么此时 ticker 底层触发与采集状态大致情况为如下四个周期的循环 (此处时间仅为参考并非真实采样时间):



这样就会导致回调可用时间不固定, 不过也并不会过多的影响采集实时性和周期一致性,

需要自行进行规划考虑。

3.4.8.WIRELESS_UART 子模块

用于驱动逐飞科技的无线转串口模块，使用固定引脚：UART3 的 C6-TX/C7-RX 以及一个 D24-RTS，用于对接逐飞助手上位机，方便进行调试。

如果使用 V2.4 版本以上无线串口模块，可以任意设置波特率，模块会自动识别并匹配，如果使用 V2.4 以下版本，需要自行通过上位机设置波特率后程序设置对应的波特率才能通信。

```
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
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 | 必要参数
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 | 至少一个数据 最多可以填八个数据 数据类型支持浮点数
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,
# | CCD1_2_BUFFER_INDEX, CCD3_4_BUFFER_INDEX]
# | 分别代表 仅显示 CCD1 图像、仅显示 CCD2 图像、选择两个 CCD 图像一起显示
wireless.send_ccd_image(WIRELESS_UART.CCD1_BUFFER_INDEX)

WIRELESS_UART.data_analysis() # 逐飞助手调参数据解析 会返回八个标志位的列表 标识各通道是否有数据更新
WIRELESS_UART.get_data()     # 逐飞助手调参数据获取 会返回八个数据的列表
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]))
```

4.多文件调用与类定义

虽然使用 Python 编写程序可以用较少的代码量完成任务，但渡过起步阶段后引入更多的算法、决策必然会导致代码量增大，此时就有分文件管理的需求了。

4.1.多文件加载

RT1021 允许多文件加载，当通过 import 包含其他文件时，它是将文件加载到内存作为一个对象来使用，因此使用 from name import * 的方式引用文件时，如果不同文件内有同样名称的函数，就会被重载覆盖掉：

```
# func.py 文件
def func1(x):
    return (x+1)

# func1.py 文件
def func1(x):
    return (x+2)
```

使用 REPL 测试如下：

```
Shell x
MPY: soft reboot
MicroPython v1.20.0 on 2025-02-11; RT1021 MicroPython by NXP & SeekFree with CoreBoard-144Pin V2.1.0
Type "help()" for more information.
>>> from func import *
>>> func1(1)
2
>>> from func1 import *
>>> func1(1)
3
>>> |
```

因此如果多个文件中包含同样的 class、function 的话，尽量使用 import name 的方式然后通过对对象调用：

```
Shell x
MPY: soft reboot
MicroPython v1.20.0 on 2025-02-11; RT1021 MicroPython by NXP & SeekFree with CoreBoard-144Pin V2.1.0
Type "help()" for more information.
>>> import func
>>> import func1
>>> func.func1(1)
2
>>> func1.func1(1)
3
>>> |
```

4.2.多文件变量作用域

基于这个文件加载的特性，在多文件使用全局变量时，建议通过 `import name`，使用 `name.value` 的方式访问全局变量，否则容易导致因作用域不清晰而产生的数据错误。例如我们使用如下两个文件测试变量作用域：

func.py 文件：

```
data = 10

def func (x):
    global data
    return(x + data)
```

user_main.py 文件：

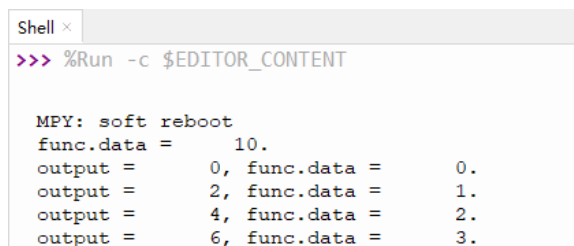
```
from machine import *
import func
import gc, time

num = 0
led = Pin('C4' , Pin.OUT, pull = Pin.PULL_UP_47K, value = True)

def test(x):
    func.data = x
    return func.func(x)

while True:
    time.sleep_ms(500)
    led.toggle()
    print("output = {:>6d}, func.data = {:>6d}.".format(test(num), func.data))
    num = num + 1
    gc.collect()
```

运行 user_main 文件就会看到如下效果，这里就由于是通过对象访问，因此传递的变量就是正确的值，输出值是 func.py 文件中 data 变量的两倍（传入值赋值给了 data）：



```
Shell x
>>> %Run -c $EDITOR_CONTENT

MPY: soft reboot
func.data =      10.
output =      0, func.data =      0.
output =      2, func.data =      1.
output =      4, func.data =      2.
output =      6, func.data =      3.
```


而如果使用 `from name import *` 的方式引用文件，可能会导致 `user_main` 文件中的函数无法调用到 `func` 文件中的变量，使用 `global` 时就会额外自行新建一个变量：

`user_main.py` 文件：

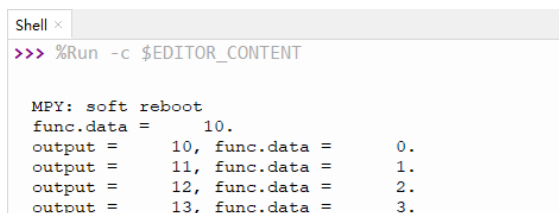
```
from machine import *
from func import *
import gc, time

num = 0
led = Pin('C4', Pin.OUT, pull = Pin.PULL_UP_47K, value = True)
print("func.data = {:>6d}.".format(data)) # 此时输出的是 func.py 文件中 data 变量的值

def test(x):
    global data # 设想它要引用 func.py 文件中 data 变量
    data = x # 但实际它是新建了一个变量替代掉了 data 这个标签
    return func(x) # 在这之后的代码 data 标签将不再访问到 func.py 文件中 data 变量

while True:
    time.sleep_ms(500)
    led.toggle()
    print("output = {:>6d}, func.data = {:>6d}.".format(test(num), data))
    num = num + 1
    gc.collect()
```

而此时 `user_main` 中会额外新建 `global` 的 `data` 变量覆盖了“`data`”这个标签，导致计算结果并非我们设想的那样（`func` 文件中的 `data` 一直为 10 没有改变，改变的是 `user_main` 中的 `data` 标签的变量值），并且我们也无法再访问到 `func` 文件中的 `data` 变量。



```
Shell x
>>> %Run -c $EDITOR_CONTENT

MPY: soft reboot
func.data =      10.
output =      10, func.data =      0.
output =      11, func.data =      1.
output =      12, func.data =      2.
output =      13, func.data =      3.
```

4.3.类的应用

除了《4.2.多文件变量作用域》的变量跨文件调用方式，还可以使用自拟定类的方式，通过传递对象的方式来传递数据。例如电机控制、专项控制需要用到 PID 算法，此时可以将 PID 算法定义一个类，编写成单独的文件 `pid_func.py`：

```
class pid_class:
    def __init__(self, kp = 0, ki = 0, kd = 0, ei = 0, ei_max = 0, output_max = 0):
        self.kp = kp
    ...
    # 代码略

    def pid_standard_integral (self, error):
        # 位置式 PID 算法
        self.e1 = self.e0
    ...
    # 代码略

    def pid_standard_incremental (self, error):
        # 增量式 PID 算法
        self.e1 = self.e0
    ...
    # 代码略
```

这样直接通过 from name import * 的方式引用文件直接获得类定义，跨文件也可以调用：

func.py 文件：

```
from pid_func import *

def func (x):
    print(x.kp)
```

func1.py 文件：

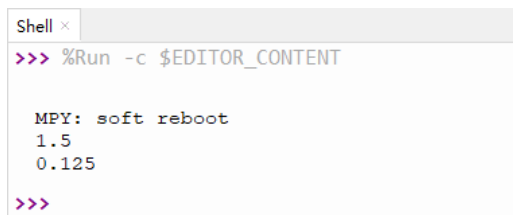
```
def func1 (x):
    print(x.ki)
```

user_main.py 文件：

```
from func import *
from func1 import *

gyro_pid = pid_class(kp = 1.5, ki = 0.125)
func(gyro_pid)
func1(gyro_pid)
```

运行得到结果：



```
Shell x
>>> %Run -c $EDITOR_CONTENT

MPY: soft reboot
1.5
0.125
>>>
```

虽然在 func1.py 文件中并未引用 pid_func 文件，但是由于它们都实际在 user_main.py 文件引用，而 user_main.py 通过 func.py 获得了 pid_func 的类定义，所以它依旧可以运行。也就是实际上 user_main.py 将各文件引用过来后在同一个作用域（user_main）生效。

5.文档版本

版本号	日期	作者	内容变更
V1.0	2024/3/8	云猫	初始版本。
V1.1	2024/3/9	云猫	修改 Thonny 配图与说明。
V1.2	2024/3/11	云猫	增加多文件调用说明
V1.3	2024/3/12	云猫	根据反馈修改描述，并增加 Ticker 启动处理的描述
V1.4	2024/3/15	云猫	新增版本描述，新增固件启动说明并配图
V1.5	2024/4/9	云猫	新增 DL1B 支持描述 修改原有描述 新增脱机运行详述
V1.6	2024/12/9	云猫	修改部分描述，修复部分错误
V2.0	2025/2/12	云猫	统一 V2.1.0 固件版本新说明书初版。