machine-硬件接口控制模块

版本：1.0

日期：2026-02-09

状态：受控/临时文件

文件保密级别：（请勾选 ■ ）		
绝密 □	保密 □	公开 ■

文件管控表

文件变更记录			
修订日期	最新版本	修订内容	编制
2026-02-09	1	首次发行	Wells Li

上海移远通信技术股份有限公司（以下简称“移远通信”）始终以为客户提供最及时、最全面的服务为宗旨。如需任何帮助，请随时联系我司上海总部，联系方式如下：

上海移远通信技术股份有限公司
上海市闵行区田林路 1016 号科技绿洲 3 期（B 区）5 号楼　　邮编：200233
电话：+86 21 5108 6236　　邮箱：info@quectel.com

或联系我司当地办事处，详情请登录：http://www.quectel.com/cn/support/sales.htm。

如需技术支持或反馈我司技术文档中的问题，请随时登录网址：
http://www.quectel.com/cn/support/technical.htm 或发送邮件至：support@quectel.com。

前言

移远通信提供该文档内容以支持客户的产品设计。客户须按照文档中提供的规范、参数来设计产品。同时，您理解并同意，移远通信提供的参考设计仅作为示例。您同意在设计您目标产品时使用您独立的分析、评估和判断。在使用本文档所指导的任何硬软件或服务之前，请仔细阅读本声明。您在此承认并同意，尽管移远通信采取了商业范围内的合理努力来提供尽可能好的体验，但本文档和其所涉及服务是在“可用”基础上提供给您的。移远通信可在未事先通知的情况下，自行决定随时增加、修改或重述本文档。

使用和披露限制

许可协议

除非移远通信特别授权，否则我司所提供硬软件、材料和文档的接收方须对接收的内容保密，不得将其用于除本项目的实施与开展以外的任何其他目的。

版权声明

移远通信产品和本协议项下的第三方产品可能包含受移远通信或第三方材料、硬软件和文档版权保护的相关资料。除非事先得到书面同意，否则您不得获取、使用、向第三方披露我司所提供的文档和信息，或对此类受版权保护的资料进行复制、转载、抄袭、出版、展示、翻译、分发、合并、修改，或创造其衍生作品。移远通信或第三方对受版权保护的资料拥有专有权，不授予或转让任何专利、版权、商标或服务商标权的许可。为避免歧义，除了正常的非独家、免版税的产品使用许可，任何形式的购买都不可被视为授予许可。对于任何违反保密义务、未经授权使用或以其他非法形式恶意使用所述文档和信息的违法侵权行为，移远通信有权追究法律责任。

商标

除另行规定，本文档中的任何内容均不授予在广告、宣传或其他方面使用移远通信或第三方的任何商标、商号及名称，或其缩略语，或其仿冒品的权利。

第三方权利

您理解本文档可能涉及一个或多个属于第三方的硬软件和文档（“第三方材料”）。您对此类第三方材料的使用应受本文档的所有限制和义务约束。

移远通信针对第三方材料不做任何明示或暗示的保证或陈述，包括但不限于任何暗示或法定的适销性或特定用途的适用性、平静受益权、系统集成、信息准确性以及与许可技术或被许可人使用许可技术相关的不侵犯任何第三方知识产权的保证。本协议中的任何内容都不构成移远通信对任何移远通信产品或任何其他硬软件、设备、工具、信息或产品的开发、增强、修改、分销、营销、销售、提供销售或以其他方式维持生产的陈述或保证。此外，移远通信免除因交易过程、使用或贸易而产生的任何和所有保证。

隐私声明

为实现移远通信产品功能，特定设备数据将会上传至移远通信或第三方服务器（包括运营商、芯片供应商或您指定的服务器）。移远通信严格遵守相关法律法规，仅为实现产品功能之目的或在适用法律允许的情况下保留、使用、披露或以其他方式处理相关数据。当您与第三方进行数据交互前，请自行了解其隐私保护和数据安全政策。

免责声明

1) 移远通信不承担任何因未能遵守有关操作或设计规范而造成损害的责任。
2) 移远通信不承担因本文档中的任何因不准确、遗漏、或使用本文档中的信息而产生的任何责任。
3) 移远通信尽力确保开发中功能的完整性、准确性、及时性，但不排除上述功能错误或遗漏的可能。除非另有协议规定，否则移远通信对开发中功能的使用不做任何暗示或法定的保证。在适用法律允许的最大范围内，移远通信不对任何因使用开发中功能而遭受的损害承担责任，无论此类损害是否可以预见。
4) 移远通信对第三方网站及第三方资源的信息、内容、广告、商业报价、产品、服务和材料的可访问性、安全性、准确性、可用性、合法性和完整性不承担任何法律责任。

目录

1 模块概述

machine 模块提供了对硬件外设的直接控制和访问功能，包括 GPIO、SPI、I2C、UART 等接口。

2 Pin(GPIO 控制)

2.1. Pin 构造函数

2.1.1. 接口概述

创建一个 Pin 对象，用于控制 GPIO 引脚。

2.1.2. 函数原型

pin = machine.Pin(id, mode, pull= Pin.PULL_NONE, value=None)

2.1.3. 参数描述

参数名	是 否必填	数据类型	默认值	说明
id	是	string	-	引脚名称, ex'PA0'
mode	是	int	-	引脚模式： Pin.IN, Pin.OUT, Pin.OPEN_DRAIN
pull	否	int	Pin.PULL_NONE	上拉/下拉： Pin.PULL_UP,Pin.PULL_DOWN, Pin.PULL_NONE
value	否	int	None	初始输出值（仅 OUT 模式有效），只能通过关键字 value 传入

2.1.4. 返回值描述

返回 Pin 对象。

2.1.5. 示例

```
import machine

# 创建输出引脚
led = machine.Pin('PB7', machine.Pin.OUT, value=1)

# 创建输入引脚
button = machine.Pin('PC13', machine.Pin.IN, machine.Pin.PULL_UP)
```

2.2. value

2.2.1. 接口概述

获取或设置引脚的逻辑电平。

2.2.2. 函数原型

```
# 获取当前值
val = pin.value()

# 设置新值
pin.value(new_value)
```

2.2.3. 参数描述

参数名	是 否必填	数据类型	默认值	说明
new_value	否	int	-	要设置的电平值

2.2.4. 返回值描述

读取时返回当前电平值（0 或 1）。

2.2.5. 示例

```
led = Pin('PB7', Pin.OUT, Pin.PULL_NONE, value=0)
time.sleep(1)
print(led.value())
time.sleep(1)
led.value(1)
```

```
print(led.value())
```

2.3. on

2.3.1. 接口概述

设置引脚为高电平。

2.3.2. 函数原型

```
pin.on()
```

2.3.3. 参数描述

无

2.3.4. 返回值描述

无返回值。

2.3.5. 示例

```
led = Pin('PB7', Pin.OUT, Pin.PULL_NONE, value=0)
led.on()
print(led.value())
```

2.4. off

2.4.1. 接口概述

设置引脚为低电平。

2.4.2. 函数原型

```
pin.off()
```

2.4.3. 参数描述

无

2.4.4. 返回值描述

无返回值。

2.4.5. 示例

```
led = Pin('PB7', Pin.OUT, Pin.PULL_NONE, value=1)
```

```
led.off()
print(led.value())
```

2.5. high

2.5.1. 接口概述

设置引脚为高电平。

2.5.2. 函数原型

```
pin.high()
```

2.5.3. 参数描述

无

2.5.4. 返回值描述

无返回值。

2.5.5. 示例

```
led = Pin('PB7', Pin.OUT, Pin.PULL_NONE, value=0)
led.high()
print(led.value())
```

2.6. low

2.6.1. 接口概述

设置引脚为低电平。

2.6.2. 函数原型

```
pin.low()
```

2.6.3. 参数描述

无。

2.6.4. 返回值描述

无返回值。

2.6.5. 示例

```
led = Pin('PB7', Pin.OUT, Pin.PULL_NONE, value=1)
led.low()
print(led.value())
```

2.7. irq()

2.7.1. 接口概述

配置引脚中断处理。

2.7.2. 函数原型

```
pin.irq(handler=None, trigger=Pin.IRQ_RISING|Pin.IRQ_FALLING, hard=False)
```

2.7.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
handler	否	function	None	中断处理函数
trigger	否	int	IRQ_RISING|IRQ_FALLING	触发条件
hard	否	bool	False	True=硬件中断，False=软件中断,只支持软中断

2.7.4. 返回值描述

无返回值。

2.7.5. 示例

```
button = machine.Pin('SW', Pin.IN, Pin.PULL_DOWN)

last_interrupt_time = 0
DEBOUNCE_MS = 200   # 200ms 防抖动时间窗口

def button_handler(pin):
    global last_interrupt_time

    current_time = time.ticks_ms()

    # 关键：如果距离上次中断时间小于 200ms，则直接放弃
    if time.ticks_diff(current_time, last_interrupt_time) < DEBOUNCE_MS:
```

```
            return  # 直接返回，不处理本次中断
        last_interrupt_time = current_time

        # 原有的消抖逻辑
        time.sleep_ms(20)

        # 延时后再次确认电平状态
        if pin.value() == 1:  # 确认是稳定的按下状态
            print("press button！")

button.irq(trigger=Pin.IRQ_RISING, handler=button_handler)
print("start testing")
while True:
    time.sleep(1)
```

3 SPI(串行外设接口)

3.1. SPI 构造函数

3.1.1. 接口概述

创建 SPI 主设备对象。

3.1.2. 函数原型

```
spi = machine.SPI(id, baudrate=500000, polarity=0, phase=0, bits=8, firstbit=SPI.MSB)
```

3.1.3. 参数描述

参数名	是 否必填	数据类型	默认值	说明
id	是	int	-	SPI 总线 ID
baudrate	否	int	500000	通信波特率（Hz）
polarity	否	int	0	时钟极性
phase	否	int	0	时钟相位
bits	否	int	8	数据位宽
firstbit	否	int	SPI.MSB	传输顺序

3.1.4. 参数详细说明

3.1.4.1. 通信模式配置

SPI 有 4 种工作模式，由极性（polarity）和相位（phase）组合决定：

模式	polarity	phase	描述
0	0	0	时钟空闲低电平，在第一个边沿采样
1	0	1	时钟空闲低电平，在第二个边沿采样
2	1	0	时钟空闲高电平，在第一个边沿采样
3	1	1	时钟空闲高电平，在第二个边沿采样

3.1.4.2. 传输顺序

```
# 高位先传（默认）
spi = machine.SPI(0, firstbit=machine.SPI.MSB)

# 低位先传
spi = machine.SPI(0, firstbit=machine.SPI.LSB)
```

3.1.4.3. 引脚配置说明

```
# 以下写法会抛出异常
spi = machine.SPI(0, sck=machine.Pin('PA5'))  # 错误：不支持

# 正确用法：使用默认引脚
spi = machine.SPI(1)
```

3.1.5. 返回值描述

返回 SPI 对象。

3.1.6. 示例

```
import machine

# 创建 SPI1 对象，默认配置
spi1 = machine.SPI(0)

# 创建 SPI2 对象，自定义波特率
spi2 = machine.SPI(1, baudrate=1000000)

# 创建 SPI3 对象，完整配置（模式 3）
spi3 = machine.SPI(1,
```

```
                baudrate=2000000,
                polarity=1,
                phase=1,
                bits=8,
                firstbit=machine.SPI.MSB)
```

3.2. write

3.2.1. 接口概述

向 SPI 总线写入数据。

3.2.2. 函数原型

```
spi.write(data)
```

3.2.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
data	是	bytes/bytearray	-	要发送的数据

3.2.4. 返回值描述

无返回值。

3.2.5. 示例

```
import machine

spi = machine.SPI(0)

# 发送字节数据
spi.write(b'\x01\x02\x03')

# 发送字节数组
data = bytearray([0xAA, 0xBB, 0xCC])
spi.write(data)

# 发送单个字节
spi.write(b'\xFF')
```

3.3. read

3.3.1. 接口概述

从 SPI 总线读取数据。

3.3.2. 函数原型

```
data = spi.read(nbytes)
data = spi.read(nbytes, write=0x00))
```

3.3.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
nbytes	是	int	-	要发送的数据
write	否	int	0	发送的字节值（0-255）

3.3.4. 返回值描述

返回 bytes 类型的数据。

3.3.5. 示例

```
import machine

spi = machine.SPI(1)

# 读取 10 个字节（发送 0x00）
data = spi.read(10)
print(f"读取 {len(data)} 字节: {data.hex()}")

# 读取 5 个字节（发送 0xFF）
data = spi.read(5, 0xFF)
print(f"读取 {len(data)} 字节: {data.hex()}")

# 读取设备 ID（发送 0x9F 命令）
device_id = spi.read(3, 0x9F)
print(f"设备 ID: {device_id.hex()}")
```

4 UART

4.1. UART 构造函数

4.1.1. 接口概述

创建并配置 UART 串口对象。

4.1.2. 函数原型

```
uart = machine.UART(id, baudrate, bits=8, parity=None, stop=1,
                    timeout=0, timeout_char=0, flow=0, rxbuf=-1)
```

4.1.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
id	是	int	-	UART 端口标识符
baudrate	是	int	10	波特率（bits/sec）
bits	否	int	8	数据位：8, 9
parity	否	int/None	None	校验位：None, 0(偶), 1(奇)
stop	否	int	1	停止位：1 或 2
timeout	否	int	0	接收超时（毫秒）
timeout_char	否	int	0	字符间超时（毫秒）
flow	否	int	0	当前不支持流控
rxbuf	否	int	-1	接收缓冲区大小

4.1.4. 返回值描述

返回 UART 对象。

4.1.5. 示例

```
import machine
uart2 = machine.UART(2, 9600)
```

4.2. any

4.2.1. 接口概述

检查接收缓冲区中可用的字符数量。

4.2.2. 函数原型

count = uart.any()

4.2.3. 参数描述

无。

4.2.4. 返回值描述

返回可读取的字符数量（int）。

4.2.5. 示例

```
import machine
import time
uart2 = machine.UART(2, 9600, timeout=1000)
while True:
    count = uart2.any()
    if count:
        data = uart2.readchar()
        if data:
            print(f"receive: {data}")
```

4.3. read

4.3.1. 接口概述

从 UART 读取数据。

4.3.2. 函数原型

data = uart.read(nbytes)

4.3.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
nbytes	否	int	-1	要读取的字节数,-1 表示读取所有数据

4.3.4. 返回值描述

返回 bytes 或 None。

4.3.5. 示例

```
import machine
```

```
uart2 = machine.UART(2, 115200)
data1 = uart2.read()
data2 =uart2.read(10)
```

4.4. readinto

4.4.1. 接口概述

从 UART 读取数据。

4.4.2. 函数原型

```
count = uart.readinto(buf)
```

4.4.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
buf	是	bytearray	-	目标缓冲区

4.4.4. 返回值描述

返回读取的字节数。

4.4.5. 示例

```
import machine
uart2 = machine.UART(2, 115200)
buf=bytearray(64)
count = uart2.readinto(buf)
```

4.5. readchar

4.5.1. 接口概述

读取单个字符。

4.5.2. 函数原型

```
char = uart.readchar()
```

4.5.3. 参数描述

无。

4.5.4. 返回值描述

返回读取的字符（int），超时返回 -1。

4.5.5. 示例

```
import machine
uart2 = machine.UART(2, 115200)
char = uart2.readchar()
```

4.6. write

4.6.1. 接口概述

向 UART 写入数据。

4.6.2. 函数原型

```
uart.write(data)
```

4.6.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
data	是	bytes/bytearray	-1	要发送的数据

4.6.4. 返回值描述

返回写入的字节数。

4.6.5. 示例

```
import machine
uart2 = machine.UART(2, 115200)
uart2.write("Hello")
```

4.7. writechar

4.7.1. 接口概述

向 UART 写入数据。

4.7.2. 函数原型

```
uart.writechar(char)
```

4.7.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
char	是	int	-1	要发送的字符

4.7.4. 返回值描述

无返回值。

4.7.5. 示例

```
import machine
uart2 = machine.UART(2, 115200)
uart.writechar(65) #A
```

4.8. irq

4.8.1. 接口概述

配置 UART 中断。

4.8.2. 函数原型

```
uart.irq(handler, trigger)
```

4.8.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
handler	是	function	-	中断处理函数
trigger	是	int	-	中断触发条件

4.8.4. 中断触发条件

```
# 接收中断
uart.irq(handler=rx_handler, trigger=UART.IRQ_RX)

# 空闲中断
uart.irq(handler=idle_handler, trigger=UART.IRQ_RXIDLE)

# 组合中断
uart.irq(handler=combo_handler, trigger=UART.IRQ_RX | UART.IRQ_RXIDLE)
```

4.8.5. 返回值描述

返回 IRQ 对象。

4.8.6. 示例

```
import machine
total = 0
def uart_rx_handler(uart_obj):
    global total
    """UART 接收中断处理函数"""
    count = uart_obj.any()
    if count:
        data = uart_obj.read(count)
        print(f"receive: {data}")
        total += count
        print(f"total {total}")

# 配置 UART
uart = machine.UART(2, 9600, rxbuf=256)

# 配置接收中断
uart.irq(handler=uart_rx_handler, trigger=machine.UART.IRQ_RX)


# 在主循环中处理（中断会自动触发）
while True:
    machine.idle()  # 进入低功耗模式，等待中断
```

UART

UART 构造函数

接口概述

创建并配置 UART 串口对象。

函数原型

```
uart = machine.UART(id, baudrate, bits=8, parity=None, stop=1,
                    timeout=0, timeout_char=0, flow=0, rxbuf=-1)
```

参数描述

参数名	是否必填	数据类型	默认值	说明
id	是	int	-	UART 端口标识符
baudrate	是	int	10	波特率（bits/sec）
bits	否	int	8	数据位：8, 9
parity	否	int/None	None	校验位：None, 0(偶), 1(奇)

stop	否	int	1	停止位：1 或 2
timeout	否	int	0	接收超时（毫秒）
timeout_char	否	int	0	字符间超时（毫秒）
flow	否	int	0	当前不支持流控
rxbuf	否	int	-1	接收缓冲区大小

返回值描述

返回 UART 对象。

示例

```
import machine
uart2 = machine.UART(2, 9600)
```

any

接口概述

检查接收缓冲区中可用的字符数量。

函数原型

```
count = uart.any()
```

参数描述

无。

返回值描述

返回可读取的字符数量（int）。

示例

```
import machine
import time
uart2 = machine.UART(2, 9600, timeout=1000)
while True:
    count = uart2.any()
    if count:
        data = uart2.readchar()
        if data:
            print(f"receive: {data}")
```

read

接口概述

从 UART 读取数据。

函数原型

```
data = uart.read(nbytes)
```

参数描述

参数名	是否必填	数据类型	默认值	说明
nbytes	否	int	-1	要读取的字节数,-1 表示读取所有数据

返回值描述

返回 bytes 或 None。

示例

```
import machine
uart2 = machine.UART(2, 115200)
data1 = uart2.read()
data2 =uart2.read(10)
```

readinto

接口概述

从 UART 读取数据。

函数原型

```
count = uart.readinto(buf)
```

参数描述

参数名	是否必填	数据类型	默认值	说明
buf	是	bytearray	-	目标缓冲区

返回值描述

返回读取的字节数。

示例

```
import machine
uart2 = machine.UART(2, 115200)
buf=bytearray(64)
count = uart2.readinto(buf)
```

readchar

接口概述

读取单个字符。

函数原型

```
char = uart.readchar()
```

参数描述

无。

返回值描述

返回读取的字符（int），超时返回 -1。

示例

```
import machine
uart2 = machine.UART(2, 115200)
char = uart2.readchar()
```

write

接口概述

向 UART 写入数据。

函数原型

```
uart.write(data)
```

参数描述

参数名	是否必	数据类型	默认值	说明

	填			
data	是	bytes/bytearray	-1	要发送的数据

返回值描述

返回写入的字节数。

示例

```
import machine
uart2 = machine.UART(2, 115200)
uart2.write("Hello")
```

writechar

接口概述

向 UART 写入数据。

函数原型

uart.writechar(char)

参数描述

参数名	是否必填	数据类型	默认值	说明
char	是	int	-1	要发送的字符

返回值描述

无返回值。

示例

```
import machine
uart2 = machine.UART(2, 115200)
uart.writechar(65) #A
```

irq

接口概述

配置 UART 中断。

函数原型

uart.irq(handler, trigger)

参数描述

参数名	是否必填	数据类型	默认值	说明
handler	是	function	-	中断处理函数
trigger	是	int	-	中断触发条件

中断触发条件

```
# 接收中断
uart.irq(handler=rx_handler, trigger=UART.IRQ_RX)

# 空闲中断
uart.irq(handler=idle_handler, trigger=UART.IRQ_RXIDLE)
```

```
# 组合中断
uart.irq(handler=combo_handler, trigger=UART.IRQ_RX | UART.IRQ_RXIDLE)
```

返回值描述

返回 IRQ 对象。

示例

```
import machine
total = 0
def uart_rx_handler(uart_obj):
    global total
    """UART 接收中断处理函数"""
    count = uart_obj.any()
    if count:
        data = uart_obj.read(count)
        print(f"receive: {data}")
        total += count
        print(f"total {total}")

# 配置 UART
uart = machine.UART(2, 9600, rxbuf=256)

# 配置接收中断
uart.irq(handler=uart_rx_handler, trigger=machine.UART.IRQ_RX)


# 在主循环中处理（中断会自动触发）
while True:
    machine.idle()  # 进入低功耗模式，等待中断
```

5 I2C

5.1. I2C 构造函数

5.1.1. 接口概述

创建 I2C 主设备对象，用于 I2C 总线通信。

5.1.2. 函数原型

i2c = machine.I2C(id, freq=400000, timeout=50000)

5.1.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
id	是	int	-	I2C 总线 ID
freq	否	int	400000	通信频率 (Hz)，默认 400kHz
timeout	否	int	50000	超时时间 (微秒)，默认 50ms

5.1.4. 返回值描述

返回 I2C 对象。

5.1.5. 示例

```
import machine
# 1. 初始化 I2C 总线
i2c = machine.I2C(1, freq=400000)  # 使用 I2C1
```

5.2. scan

5.2.1. 接口概述

扫描 I2C 总线上连接的设备地址。

5.2.2. 函数原型

addr_list = i2c.scan()

5.2.3. 参数描述

无。

5.2.4. 返回值描述

返回包含所有发现设备地址的列表 (list)。

5.2.5. 示例

```
import machine

# 创建 I2C 对象
i2c = machine.I2C(1, freq=100000)
```

```
# 扫描 I2C 总线上的设备
devices = i2c.scan()

if devices:
    print(f"发现 {len(devices)} 个设备：")
    for addr in devices:
        print(f"  设备地址: 0x{addr:02X} (十进制: {addr})")
else:
    print("未发现任何设备")
```

5.3. readfrom

5.3.1. 接口概述

从指定 I2C 设备地址读取数据。

5.3.2. 函数原型

```
data = i2c.readfrom(addr, nbytes)
```

5.3.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
addr	是	int	-	设备地址
nbytes	是	int	-	要读取的字节数

5.3.4. 返回值描述

返回读取的字节数据。

5.3.5. 示例

```
#以 AHT20 温湿度传感器为例
import machine
import time

# 初始化 I2C
i2c = machine.I2C(1, freq=100000)

print("I2C 初始化完成")

# AHT20 的固定地址
AHT20_ADDR = 0x38
```

```
# 1. 发送初始化命令
print("\n1. 发送初始化命令...")
i2c.writeto(AHT20_ADDR, b'\xBE\x08\x00')  # 初始化命令
time.sleep_ms(10)

# 2. 发送触发测量命令
print("2. 发送触发测量命令...")
i2c.writeto(AHT20_ADDR, b'\xAC\x33\x00')  # 触发测量
time.sleep_ms(80)  # 等待测量完成

# 3. 读取 6 字节数据
print("3. 读取数据...")
data = i2c.readfrom(AHT20_ADDR, 6)
print(f"   原始数据: {data.hex(' ')}")

# 4. 解析温湿度
# 数据格式: [状态, 湿度(20bit), 温度(20bit)]
humidity_raw = ((data[1] << 12) | (data[2] << 4) | (data[3] >> 4))
temperature_raw = (((data[3] & 0x0F) << 16) | (data[4] << 8) | data[5])

# 转换公式
humidity = (humidity_raw * 100) / 0x100000   # 0-100%
temperature = ((temperature_raw * 200.0) / 0x100000) - 50   # -50~+150°C

print(f"\n 温度: {temperature:.1f}° C")
print(f"湿度: {humidity:.1f}%")
```

5.4. writeto

5.4.1. 接口概述

向指定 I2C 设备地址写入数据。

5.4.2. 函数原型

```
count = i2c.writeto(addr, data)
```

5.4.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
addr	是	int	-	设备地址
data	是	bytes	-	要写入的数据

5.4.4. 返回值描述

返回成功写入的字节数。

5.4.5. 示例

#参考 readfrom 示例

5.5. readfrom_into

5.5.1. 接口概述

从指定 I2C 设备地址读取数据。

5.5.2. 函数原型

i2c.readfrom(addr, buffer)

5.5.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
addr	是	int	-	设备地址
buffer	是	bytearray	-	目标缓冲区

5.5.4. 返回值描述

返回读取的字节数。

5.5.5. 示例

```
#以 AHT20 传感器为例
import machine
import time

# 初始化 I2C
i2c = machine.I2C(1, freq=100000)

print("I2C 初始化完成")

# AHT20 的固定地址
AHT20_ADDR = 0x38

# 1. 发送初始化命令
print("\n1. 发送初始化命令...")
count = i2c.writeto(AHT20_ADDR, b'\xBE\x08\x00')  # 初始化命令
```

```
time.sleep_ms(10)

# 2. 发送触发测量命令
print("2. 发送触发测量命令...")
i2c.writeto(AHT20_ADDR, b'\xAC\x33\x00')  # 触发测量
time.sleep_ms(80)  # 等待测量完成

# 3. 读取 6 字节数据 - 使用 readfrom_into
print("3. 读取数据...")
data_buf = bytearray(6)  # 创建缓冲区
i2c.readfrom_into(AHT20_ADDR, data_buf)  # 读取到缓冲区
print(f"  原始数据: {data_buf.hex(' ')}")

# 4. 解析温湿度
# 数据格式: [状态, 湿度(20bit), 温度(20bit)]
humidity_raw = ((data_buf[1] << 12) | (data_buf[2] << 4) | (data_buf[3] >> 4))
temperature_raw = (((data_buf[3] & 0x0F) << 16) | (data_buf[4] << 8) | data_buf[5])

# 转换公式
humidity = (humidity_raw * 100) / 0x100000  # 0-100%
temperature = ((temperature_raw * 200.0) / 0x100000) - 50  # -50~+150°C

print(f"\n 温度: {temperature:.1f}° C")
print(f"湿度: {humidity:.1f}%")
```

5.6. readfrom_mem

5.6.1. 接口概述

从 I2C 设备的指定内存地址读取数据。

5.6.2. 函数原型

```
data = i2c.readfrom_mem(addr, memaddr, nbytes)
```

5.6.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
addr	是	int	-	设备地址
memaddr	是	int	-	内存地址
nbytes	是	int	-	要读取的字节数

5.6.4. 返回值描述

返回读取的字节数据。

5.6.5. 示例

```
import machine
i2c = machine.I2C(1)
# 从设备 0x50 的地址 0x00 读取 16 个字节
data = i2c.readfrom_mem(0x50, 0x00, 16)
```

5.7. writeto_mem

5.7.1. 接口概述

向 I2C 设备的指定内存地址写入数据。

5.7.2. 函数原型

```
i2c.writeto_mem(addr, memaddr, data)
```

5.7.3. 参数描述

参数名	是否必填	数据类型	默认值	说明
addr	是	int	-	设备地址
memaddr	是	int	-	内存地址
data	是	bytes	-	要写入的数据

5.7.4. 返回值描述

无返回值。

5.7.5. 示例

```
import machine

i2c = machine.I2C(1)

# 向设备 0x50 的地址 0x10 写入配置数据
config_data = b'\x01\x02\x03\x04'
i2c.writeto_mem(0x50, 0x10, config_data)
```