LPC55S69 IoT Kit 专属 Micropython 模组和库函数简介（Ver 0.5）

Table of Contents

第1章 概览

Micropython 是一个开源项目，目标是在微控制器和小型嵌入式系统上实现 Python3.x 的语法，和部分标准的 Python 库，可在资源受限的系统中运行。

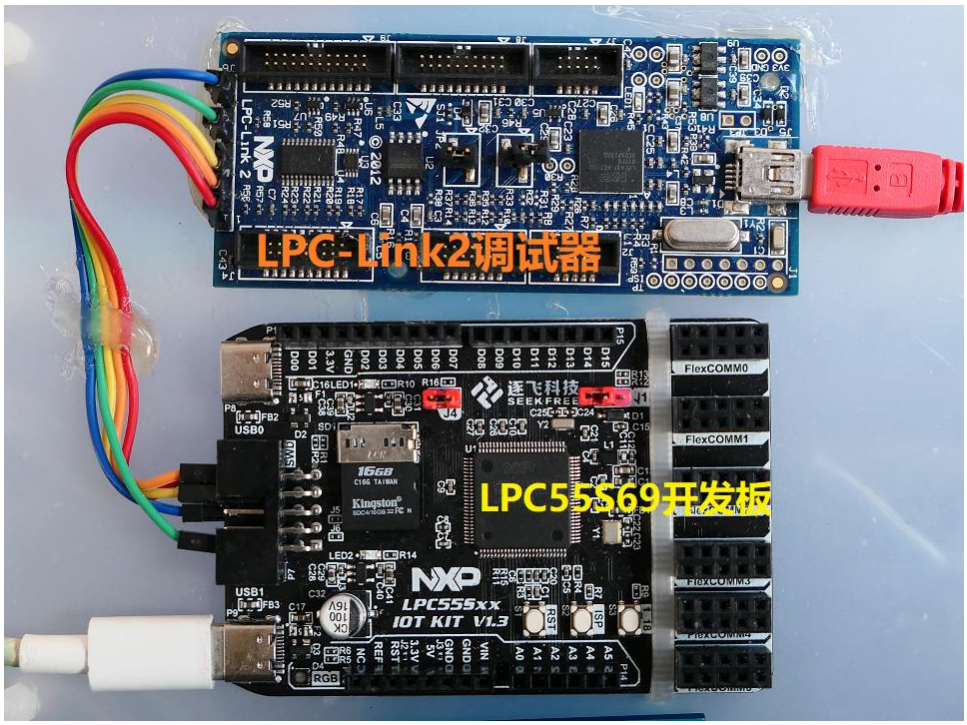
Micropython 实现了完整的 Python3.4 语法(包含 exceptions, with, yield from 等，还实现了 Python3.5 的 async/await 关键字)。提供了以下核心的数据类型：str(包含基本的 Unicode 支持), bytes, bytearray, tuple, list, dict, set, frozenset, array.array, colletions.namedtuple, 类和实例。内置模块包括 sys，time 和 struct 等。某些移植项目支持_thread 模块（多线程）。请注意，仅针对数据类型和模块实现了 Python 3 功能的一个子集。

MicroPython 可以执行文本形式的脚本或预编译字节码，文本脚本或预编译码都是在设备上的文件系统中，或“冻结”在 MicroPython 的可执行文件中。

Micropython 官方发布的代码和文档，可在 micropython.org 阅读和下载。

1.1. Micropython 在 LPC55S69 上的移植

Micropython 在 LPC55S69 上的移植工作是在逐飞科技制作的 IoT Kit 主板上进行开发的。如下图。

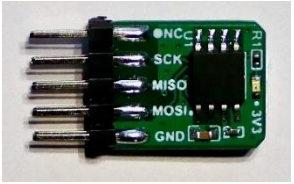


LPC55S69 适合 Micropython 的主要特点是，她有足够大的存储器：640KB 的 Flash 和 320K SRAM。尤其在移植初期，对 Micropython 不太了解的情况下，可以不必考虑存储限制，想测试什么内容、想怎么实现功能，可以放心地去做，而暂时不必考虑太多的优化和裁剪，待开发成熟后，可以经过适当的裁剪后，轻松地迁移到 LPC5500 系列中存储容量较小的型号。

1.2. Micropython 的基本运行环境

LPC55S69 上实现的 Micropython，在 USB 高速接口上，实现了两个 USB-MSC(大容量存储设备)，和两个 USB-CDC 串口设备。

两个 USB-MSC 分别对应开发板上的 SD 卡接口和一个扩展的 SPI Flash 模块，在代码里这个 SPI Flash 固定使用 FlexCOMM3 接口(未来可以根据需要配置使用其它接口，或去除它)。



开发时，使用了右图所示的 SPI Flash 扩展模块，在集成到终端应用中时，应把它直接焊接在 PCB 板上，这样它上面的文件系统和文件都可以方便地预先设置好。

在系统上电启动时，会先检查 SPI Flash 是否存在，如果存在则按照 Micropython 的设定，先执行其中根目录的 boot.py 和 main.py。

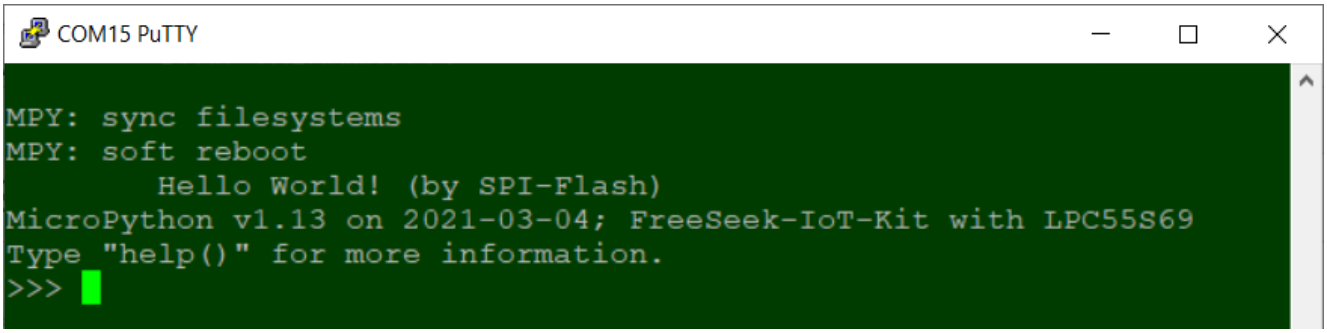
当 SPI Flash 不存在时，则检查是否有 SD 卡，如果有，则执行根目录下的 boot.py 和 main.py

随后，不管是否执行过 SPI Flash 或 SD 卡中的启动文件，将在第一个 USB-CDC 连接的 USB 串口窗口显示提示版本信息和 micropython 的提示符。

这个接口将是 Micropython 的控制终端 REPL。（在其它 micropython 的系统中，通常 REPL 是通过 UART 接口实现的。）

注：两个 CDC 接口中哪一个是 REPL 对应的端口，要看 Windows 加载驱动时先加载哪个，比较快的判断方法是自行试一下，哪个能出现“>>>”提示符。这个测试只需在此开发板第一次连接到你的 PC 时才需要，以后这个端口的编号就不会变了。

注意：由于系统上电后，往往用户来不及打开串口窗口，而看不到 micropython 的版本与提示符，但如果一切正常，在连接并打开串口窗口后，应能看到提示符，此时键入 Ctrl-D 则进行热启动，将能看到启动时显示的版本信息。下图为热启动后显示的内容：



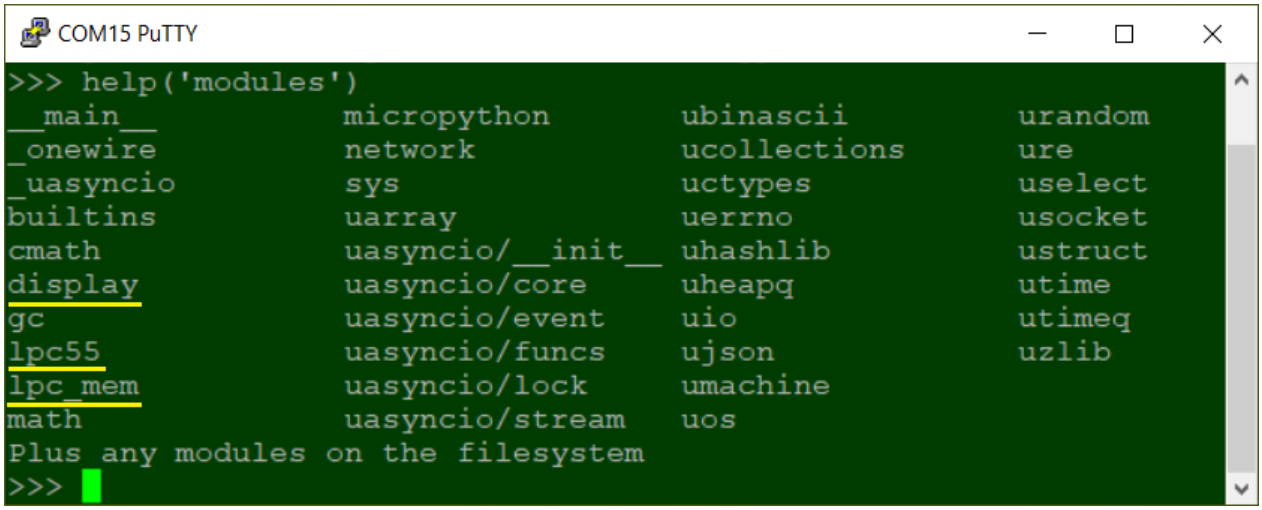
中间那行“Hello World”是 main.py 输出的信息，随后就是 micropython 的版本信息和提示符。

USB-HS 接口上的第二个 USB-CDC 串口设备，则可以由用户自由使用，例如输出一些调试信息等。后面相应章节会介绍如何使用它。

在 LPC55S69 芯片上还有一个 USB-FS 接口，目前的实现代码中没有用到这个接口，以后可以根据需要修改代码，把它配置为任意 USB 设备。

1.3. 内置 Python 模块一览

在 REPL 控制终端输入 help(‘modules’)将显示当前系统中预置的模块，如下图所示：



其中大部分都是标准 micropython 的模块，本文不赘述。有以下几个不同的模块，将会在本文后续章节介绍用法：

- lpc55　　这里包含了所有 LPC5500 片内外设对应的类
- display　　这里包含了所有支持的显示屏对应的类
- lpc_mem　　用于访问 LPC5500 片内存储区

其中的 lpc55 模块是移植的主要部分，这个模块对应标准 micropython 的 pyb 模块。

1.4. 转换 axf 文件并烧录到开发板上

首先执行以下命令把 axf 文件转换为 bin 文件

```
arm-none-eabi-objcopy -O ihex lpc55s69_iot_mpy.axf LPC55S69_mpy.hex
```

然后按住开发板上的 ISP 按钮的同时，插上 USB-HS 线至 PC(插上 USB 线后，可以放开 ISP 按钮)，然后执行以下命令进行烧录：

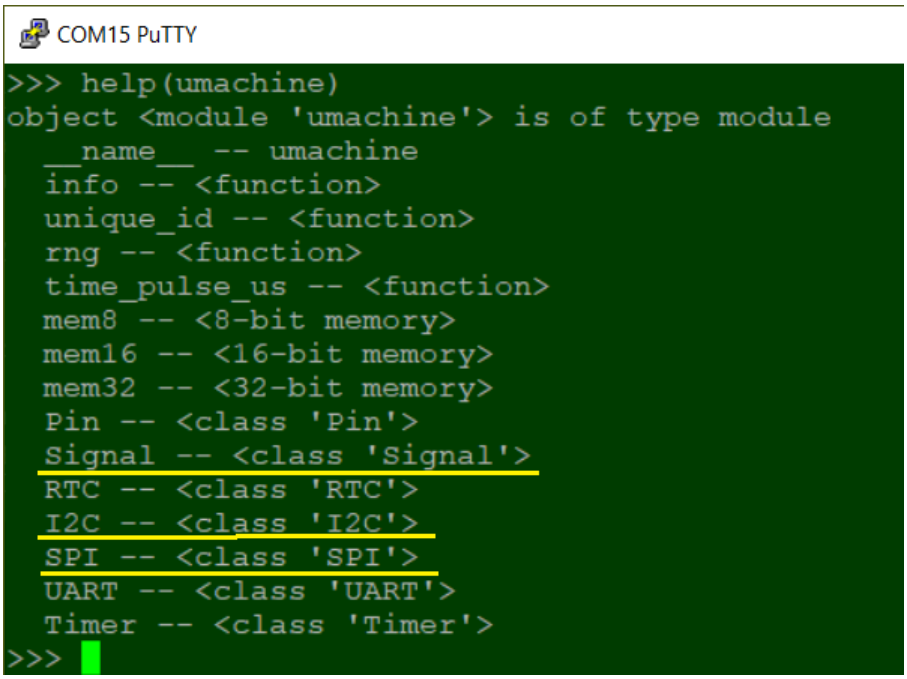
```
blhost -u 0x1fc9,0x21 -- flash-image LPC55S69_mpy.hex erase
```

第2章 内存访问模块——lpc_mem

该模块对应标准 micropython 的 machine 模块，但做了大幅简化，只保留了与存储器(包括寄存器)访问相关的部分，用户可以通过 umachine 模块访问原有的其它功能。

注意：umachine 模块中有几个与片内硬件模块相关的类，还未来得及测试，暂时不要使用。

Pin, UART 和 rng 与 lpc55 模块中相同名字的类的用法相同。



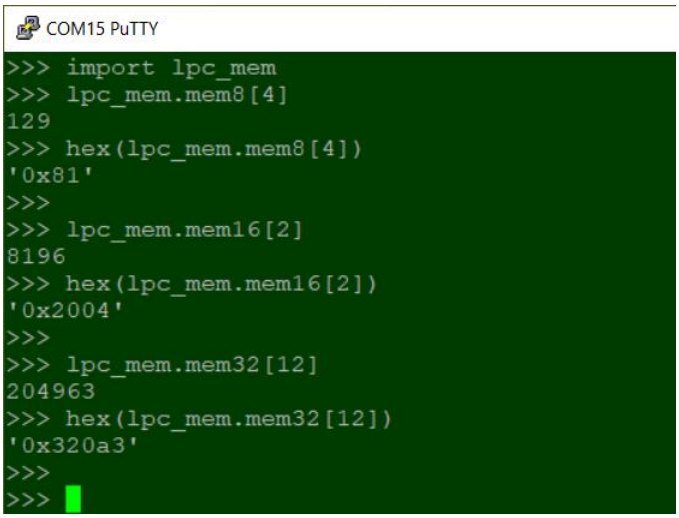
2.1. 内存访问

内存访问是以数组下标的方式操作的，按返回数值的位数，分别有三个数组：mem8，mem16 和 mem32。

- ▲ 数组 mem8 的下标可以是任意地址值。
- ▲ 数组 mem16 的下标必须是 2 字节对齐的地址值。
- ▲ 数组 mem32 的下标必须是 4 字节对齐的地址值。

它们分别返回给定地址的 8 位、16 位或 32 位的数值。

例如下面右图是地址 0x0000 0000 开始的 MCU 向量表的部分数据，左图是操作的结果：



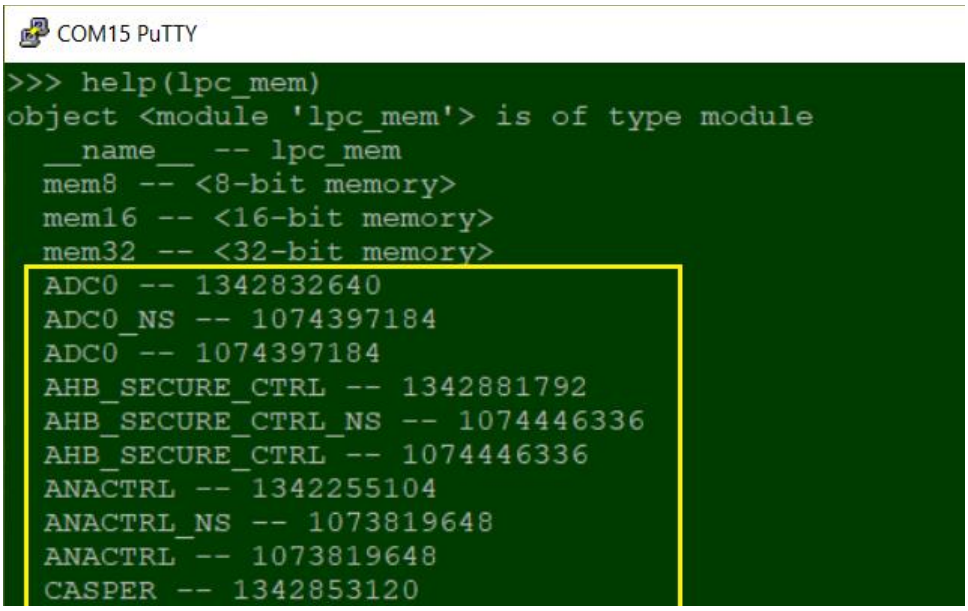
从上至下操作的结果的说明如下：

- 129： 字节地址 0x0000 0004 的十进制数值
- 0x81： 字节地址 0x0000 0004 的十六进制数值
- 8196： 半字地址 0x0000 0002 的十进制数值
- 0x2004： 半字地址 0x0000 0002 的十六进制数值
- 204963： 字地址 0x0000 000C 的十进制数值
- 0x320a3： 字地址 0x0000 000C 的十六进制数值

2.2. 访问寄存器

所有内部寄存器组的基地址和寄存器组中寄存器的偏移地址，都可以以 lpc_mem 常数的方式使用。

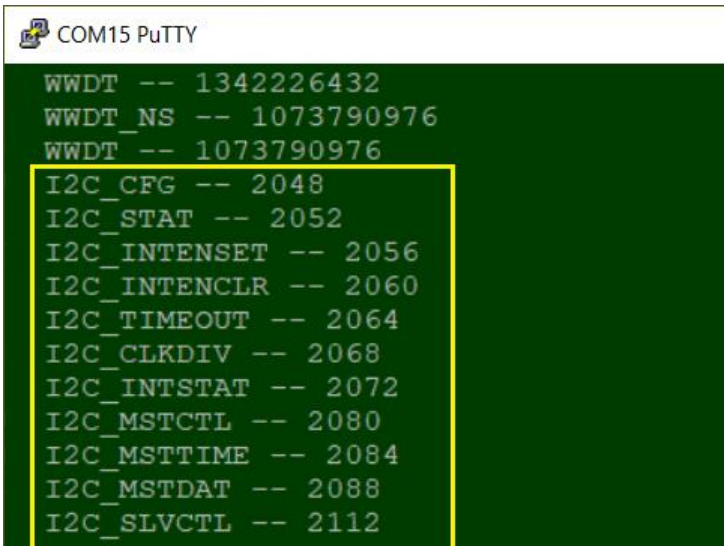
2.2.1 寄存器组基地址



如上图，每个模块的寄存器基地址常数都以模块的名字命名，带后缀_NS 的常数是以非加密方式访问该模块的地址。（目前的固件没有使能芯片的加密功能，所以访问任意模块时，可以用带_NS 后缀或不带这个后缀）

2.2.2 寄存器组内偏移地址

每个寄存器组内的寄存器偏移地址常数，是以模块的名字作为前缀，再加上寄存器的名字构成，如下图：



2.2.3 寄存器访问示例

在逐飞这个 LPC55S69 开发板上，有一个三色 LED，信号线连接如下：

1. 红色 LED ←→ PIO1-12
2. 绿色 LED ←→ PIO0-27
3. 蓝色 LED ←→ PIO1-22

首先初始化各个对应引脚为上拉输出，并熄灭 LED：

```
r = lpc55.Pin('PIO1_12',Pin.OUT, pull=Pin.PULL_UP)
g = lpc55.Pin('PIO0_27',Pin.OUT, pull=Pin.PULL_UP)
b = lpc55.Pin('PIO1_22',Pin.OUT, pull=Pin.PULL_UP)
r.value(1)
g.value(1)
b.value(1)
```

可以用 Pin 类的函数点亮某个 LED：

▲ r.value(0)：在 r 引脚上输出 0，点亮红色 LED

也可以直接操作该引脚对应的寄存器，实现相同目的：

```
import lpc_mem
lpc_mem.mem8[lpc_mem.GPIO_NS + lpc_mem.GPIO_B0 + 32+12]=0  # PIO1_12=0, Turn On LED
lpc_mem.mem8[lpc_mem.GPIO_NS + lpc_mem.GPIO_B0 + 32+12]=1  # PIO1_12=1, Turn off LED
```

2.3. 谨慎使用 lpc_mem

内存访问模块 lpc_mem 提供了最大的灵活性，让用户可以使用 Python 脚本直接操作底层寄存器。

使用 lpc_mem 要十分小心，需要仔细阅读用户手册，搞清楚每个寄存器的用法。

还要避免直接寄存器操作影响到 micropython 类的操作访问，例如当你用 Python 脚本操作 SPI 模块时，如果冒然改变了 SPI 外设的寄存器内容，有可能会出现不可预料的后果。

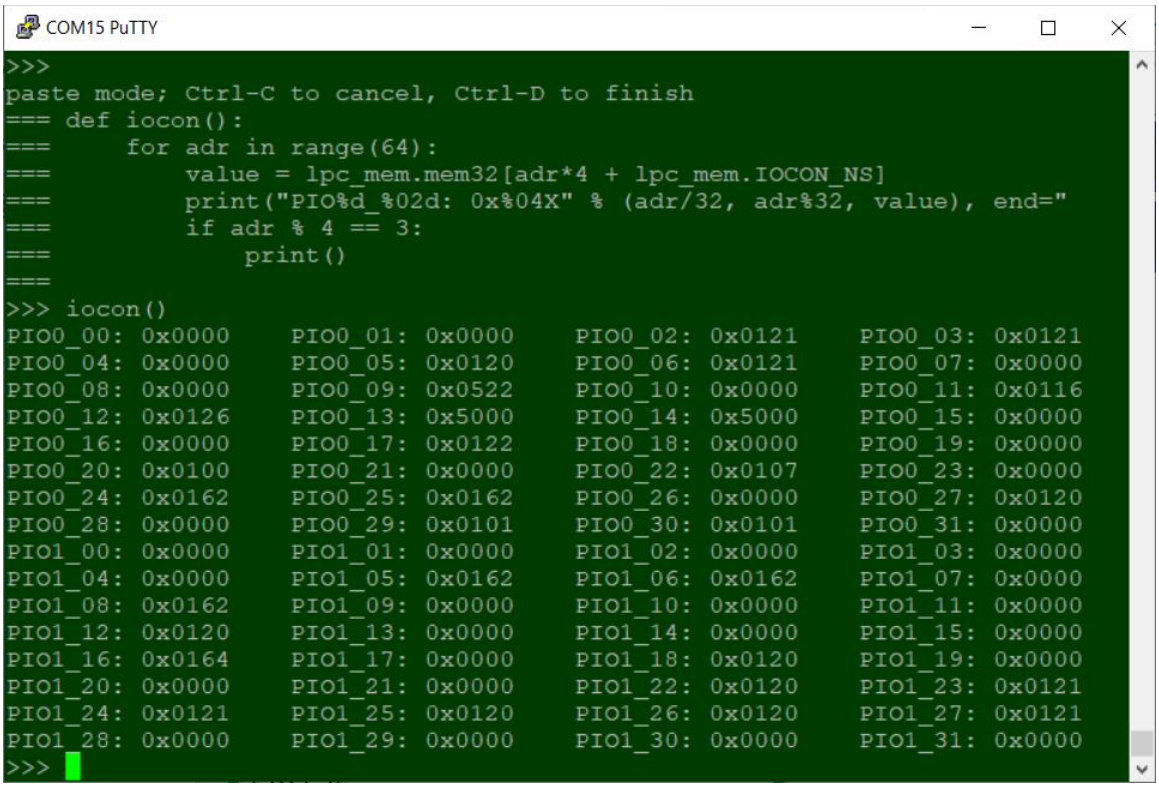
如果你对 LPC5500 的底层非常了解，可以用 lpc_mem 检查一些底层寄存器的内容，方便功能调试，尤其时修改调试 micropython 固件时是很有用的。

例如以下这个自定义函数脚本，可以用来显示所有 IOCON 的寄存器，依此查看所有引脚的配置情况：

代码片段1. **查看所有 IOCON 寄存器**

```
import lpc_mem
def iocon():
    for adr in range(64):
        value = lpc_mem.mem32[adr*4 + lpc_mem.IOCON_NS]
        print("PIO%d_%02d: 0x%04X" % (adr/32, adr%32, value), end="    ")
        if adr % 4 == 3:
            print()
```

实际操作的效果如下图：

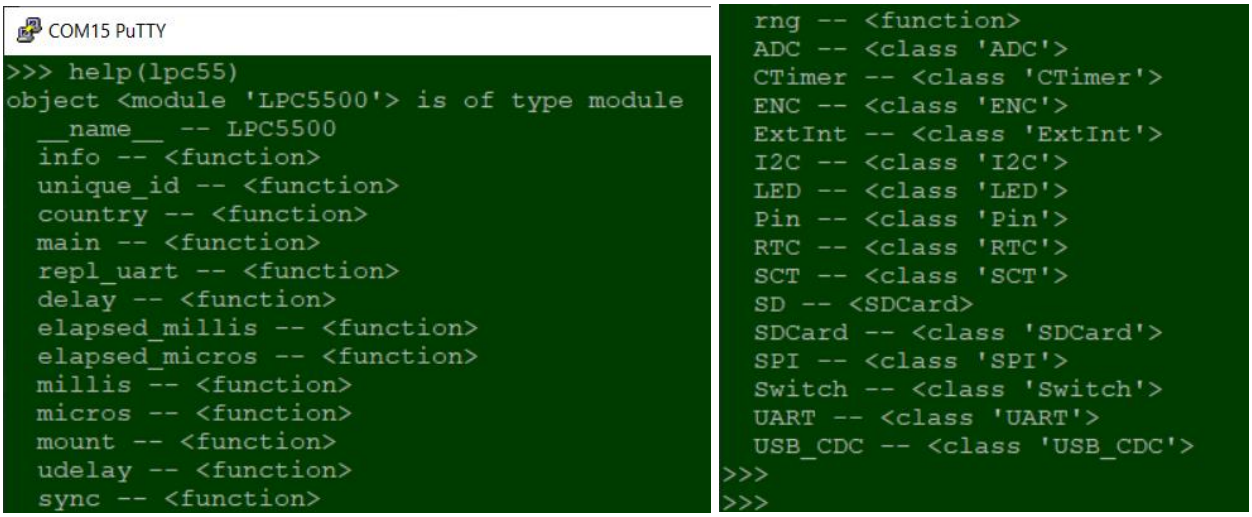

COM15 PuTTY
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
=== def iocon():
===   for adr in range(64):
===   value = lpc_mem.mem32[adr*4 + lpc_mem.IOCON_NS]
===   print("PIO%d_%02d: 0x%04X" % (adr/32, adr%32, value), end="
===   if adr % 4 == 3:
===   print()
===
>>> iocon()
PIO0_00: 0x0000 PIO0_01: 0x0000 PIO0_02: 0x0121 PIO0_03: 0x0121
PIO0_04: 0x0000 PIO0_05: 0x0120 PIO0_06: 0x0121 PIO0_07: 0x0000
PIO0_08: 0x0000 PIO0_09: 0x0522 PIO0_10: 0x0000 PIO0_11: 0x0116
PIO0_12: 0x0126 PIO0_13: 0x5000 PIO0_14: 0x5000 PIO0_15: 0x0000
PIO0_16: 0x0000 PIO0_17: 0x0122 PIO0_18: 0x0000 PIO0_19: 0x0000
PIO0_20: 0x0100 PIO0_21: 0x0000 PIO0_22: 0x0107 PIO0_23: 0x0000
PIO0_24: 0x0162 PIO0_25: 0x0162 PIO0_26: 0x0000 PIO0_27: 0x0120
PIO0_28: 0x0000 PIO0_29: 0x0101 PIO0_30: 0x0101 PIO0_31: 0x0000
PIO1_00: 0x0000 PIO1_01: 0x0000 PIO1_02: 0x0000 PIO1_03: 0x0000
PIO1_04: 0x0000 PIO1_05: 0x0162 PIO1_06: 0x0162 PIO1_07: 0x0000
PIO1_08: 0x0162 PIO1_09: 0x0000 PIO1_10: 0x0000 PIO1_11: 0x0000
PIO1_12: 0x0120 PIO1_13: 0x0000 PIO1_14: 0x0000 PIO1_15: 0x0000
PIO1_16: 0x0164 PIO1_17: 0x0000 PIO1_18: 0x0120 PIO1_19: 0x0000
PIO1_20: 0x0000 PIO1_21: 0x0000 PIO1_22: 0x0120 PIO1_23: 0x0121
PIO1_24: 0x0121 PIO1_25: 0x0120 PIO1_26: 0x0120 PIO1_27: 0x0121
PIO1_28: 0x0000 PIO1_29: 0x0000 PIO1_30: 0x0000 PIO1_31: 0x0000
>>>

第3章 外设控制模块——lpc55（通用部分）

外设控制模块集合了众多操作片上外设的类和函数，这是 micropython 与标准 Python 的主要区别之一，不同的 MCU 上的 micropython 实现都会有类似的外设控制模块，例如在 pyboard 板上的外设控制模块的名称是 pyb。

lpc55 中的类和函数，其语法和使用方法，与 pyb 类大体相同，但为了适配 LPC5500 的功能和内部定义，移植 micropython 时在不少地方做了调整，本章将介绍与其它 Micropython 移植基本相同的模块类和函数，后续章节将分别介绍那些 LPC5500 上特有的模块。

lpc55 模块中有一些函数是与标准 micropython 相同的函数，这些函数与硬件无关，因此它们的语法和功能也和其它 micropython 的实现相同，请读者直接参考 docs.micropython.org 中 pyb 模块的相关描述。



lpc55 模块还有一些与芯片与板上硬件相关的函数和类，下面将分别介绍。

3.1. 硬件无关的函数

本节简要地列出 lpc55 中与 pyb 中语法与功能相同的函数。

那些没有列在这里，但出现在 pyb 文档的函数，是暂时没有实现的功能，例如与功耗控制相关的部分等。

3.1.1 info()： 输出一些 micropython 内部信息

本函数打印输出一些固件的内部信息，主要是为固件开发者参考，因此在 lpc55 中大大简化了输出的内容。

下图是输出的示例：

```
>>> lpc55.info()
ID=8096e2c2:c5dab75d:a1cefcbc
System Clock=150000000
_sstack=2003c000
_estack=20040000
_heap_start=20016b94
_heap_end=2003ab94
qstr:
  n_pool=3
  n_qstr=64
  n_str_data_bytes=598
  n_total_bytes=1206
GC:
  144064 total
  68720 : 75344
  1=2674 2=563 m=64
LFS free: 3149824 bytes
>>>
```

3.1.2 unique_id()：读出芯片唯一 ID

lpc55.unique_id()

该函数返回一个 16 字节的 bytearray()，包含芯片的唯一 ID。

```
>>>
>>> lpc55.unique_id()
b'\x80\x96\xe2\xc2\xc5\xda\xb7]\xa1\xce\xfc\xbce.\xf6\x05'
>>> id=lpc55.unique_id()
>>> len(id)
16
>>>
```

3.1.3 country(code)：设置国家代码

该函数设置一个内部变量的值，但目前只看到与网络相关的部分用到这个内部变量，不太清楚怎么用。

lpc55.country(code)

设置国家代码，code 为 2 个 ASCII 字符。

使用示例

```
lpc55.country('CN')      # 设置国家代码为中国
lpc55.country('US')      # 设置国家代码为美国
```

3.1.4 main(filename)：设置主脚本

lpc55.main(filename)

输入一个脚本文件名字，系统启动时，在 boot.py 之后将执行这个脚本文件。

如果未用该函数指定执行脚本，则 boot.py 之后默认执行 main.py 脚本，如果 main.py 不存在也不会报错。

只有在 boot.py 中调用该函数才有意义。

3.1.5 repl_uart(uart)：指定作为 REPL 的端口

lpc55.repl_uart(uart)

函数的输入参数是一个由 lpc55.UART 创建的实体，用于指定 REPL 的端口。

3.1.6 delay(ms)：延迟若干毫秒

lpc55.delay(ms)

接受的输入是一个整数，函数通过内部 systick 延迟 ms 毫秒，然后返回。

注：由于 micropython 语句的执行开销，函数的延迟并不准确，只能作为一个参考。

3.1.7 udelay(us)：延迟若干微秒

lpc55.delay(us)

接受的输入是一个整数，函数通过内部 systick 延迟 us 微秒，然后返回。

注：由于 micropython 语句的执行开销，函数的延迟并不准确，只能作为一个参考。

3.1.8 millis()：返回自开机后的时间

lpc55.millis()

返回自开机后的时间，单位为毫秒。

3.1.9 micros()：返回自开机后的时间

lpc55.micros()

返回自开机后的时间，单位为微秒。

3.1.10 elapsed_millis()：返回当前时间与给定时间的差

lpc55. elapsed_millis(start)

给定一个开始时间，返回当前时间与给定的时间差值，单位为毫秒。

通常该函数与 millis()一起使用，可以用来测量某段代码执行的时间。

3.1.11 elapsed_micros()：返回当前时间与给定时间的差

lpc55. elapsed_micros(start)

给定一个开始时间，返回当前时间与给定的时间差值，单位为微秒。

通常该函数与 micros()一起使用，可以用来测量某段代码执行的时间。

下图演示了以上 4 个函数的用法。

```
>>>
>>> start = lpc55.micros()
>>> lpc55.delay(50)
>>> lpc55.elapsed_micros(start)
54986
>>>
>>> start = lpc55.millis()
>>> lpc55.delay(1000)
>>> lpc55.elapsed_millis(start)
1006
>>>
>>>
```

3.1.12 mount()：挂载一个存储设备

lpc55. mount(device, mountpoint, *, readonly=False, mkfs=False)

该函数的语法与功能与 pyb.mount()一样，请参考 docs.micropython.org

例如将挂载 SD 卡至根目录/sd：lpc55.mount(lpc55.SD, '/sd')，下图为执行演示：

```
>>> import uos
>>> uos.listdir('/')
['flash']
>>> lpc55.mount(lpc55.SD, '/sd')
>>> uos.listdir('/')
['flash', 'sd']
>>> uos.listdir('/sd')
['LOST.DIR', '.android_secure', 'DCIM', 'quicknote', 'notegallery', 'Android', '
Download', 'tencent', '360', 'SKIPSD', 'DevIcon.fil', 'DevLogo.fil', 'Playlists'
, 'Albums', '.udstate', 'Photo', '.dir_com.qihoo.appstore', 'Movies', '\x0f\x0f\
x0f\x0f\x08\x01~1', 'System Volume Information', 'aia_doc', 'boot.py', 'mpy', '_
main.py']
>>>
```

3.1.13 sync()：同步文件系统

lpc55. sync()

同步所有文件系统。

由于各种内部缓存的缘故，有时候文件系统的物理介质里内容与内部缓存不一致，这个函数将同步缓存与介质的内容。

一般在写入文件系统后，尤其是在关机断电之前，强烈建议使用这个函数进行文件系统同步。

3.2. 芯片内部外设相关类概述

每个芯片内部的外设模块，都有一个 Python 类封装了对它们的操作函数。本节将逐一介绍已经实现的、与其它 Micropython 移植相同功能的类和函数，以及它们的使用方法。

下表列出了 lpc55 模块中所有外设相关类：

类(函数)	简介	章节索引
rng	这是一个函数，每次调用返回一个随机数	
ADC	模拟数字转换	
CTimer	LPC5500 特有模块，用于操作片内的任一 CTimer	下一章
ENC	LPC5500 特有模块，用于读取旋转编码器的状态	
ExtInt	外部引脚中断控制	
I2C	I2C 模块控制	
LED	板上 LED 控制	
Pin	GPIO 引脚控制	
RTC	RTC 外设控制	
SCT	LPC5500 特有模块，用于操作片上 SCT 定时器	下一章
SD	这是一个将被淘汰的命名，功能同 SDCard。	
SDCard	用于操作板上 SD 卡	
SPI	SPI 模块控制	
Switch	用于读取板上的按键	
UART	UART 模块控制	
USB_CDC	用于读写 USB CDC 通道	

3.2.1 rng()：读取一个 30 位的随机数

lpc55. rng()

返回一个由硬件产生的 30 位随机数。

右图是一个例子。

```
>>> for x in range(4):
...     print(lpc55.rng())
...
756476364
428345726
116638269
502504659
>>>
```

3.3. ADC 模数转换类及函数

芯片内部有一个 16 位、采样率为 1Msps 的 ADC 模块，芯片外部有 10 个 ADC 引脚；还有一些内部通道用于测量内部电压，例如片内温度传感器。

3.3.1 ADC 创建函数

class lpc55.ADC(pin, init=False)

创建一个与指定引脚绑定的 ADC 实体，随后可以通过这个实体读取对应引脚的模拟值。

- pin 可以是一个 ADC 引脚的名字，如果给出的引脚不是 ADC 引脚，则抛出 ValueError 异常。
- pin 也可以是一个整数编号，对应 ADC 内部通道。
  - 按照 LPC55S69 用户手册，共有 3 个内部通道，分别为：
    12 = VDD3V3_ADC
    13 = Bias_vref_1v from aux_bias module
    26 = Temperature sensor
  - 注：内部通道部分未经测试，暂时不要使用
- init 表示是否需要强制初始化 ADC 模块，默认为 False。在第一次上电时，固件内部会自动初始化，一般不需人工干预。

3.3.2 ADC.deinit()函数

ADC.deinit()

关闭 ADC 模块，以节省耗电。

注：该函数将关闭 ADC 模块，所有的 ADC 引脚都不能再进行转换了，须通过创建函数再次初始化才能继续使用。

3.3.3 ADC.read()：读取当前转换数值

ADC.read()

读取 ADC 引脚上的电压值并返回一个整数，返回值的范围是 0~65535。

注：由于各种误差，强烈建议舍弃返回值的最低 3~4 位，即将返回值右移 3~4 位再用。

```
a4 = lpc55.ADC('PIO1_0')
value = a4.read() >> 4
```

3.3.4 ADC.read_timed()：按定时器设定频率读取

ADC.read_timed(buf, timer)

按照 timer 的速率，读出一组数据并存放在 buf 中。

该函数设置好周期转换后将立即返回，需设置一个回调函数以获知转换的结束。

- buf 可以是一个 bytearray 或 array.array，数组中的每个单元应该可以容纳 16 位或以上位数的整数，如果 buf 的每个单元只有 8 位(bytearray 的情况)，则只取结果的高 8 位。
- timer 是一个 CTimer 类的实体，定时器周期性地触发 ADC 转换并读取结果至 buf。
- timer 也可以是一个整数，给定采样频率，micropython 固件将自动使用 CTimer4 的通道 3 作为 ADC 周期采样的触发源。

注 1：根据 LPC55S69 的用户手册，只能使用 CTimer1~4 的通道 3 触发 ADC 转换，应合理配置 CTimer。

注 2：ADC 的触发源还可以是 SCT 的一些输出端，和其它一些信号，目前的固件暂不支持这些触发源，未来可以根据需要添加相应的配置方式。

3.3.5 ADC.callback()：设置成组转换结束时的回调函数

ADC.callback(func)

- func 是一个回调函数，read_timed()函数设置的成组转换结束后，固件将调用这个回调函数，用户可以在这个回调函数处理读取的数据，或设置结束标志。
- 如果没有用这个函数设置回调函数，或 func 参数空缺或等于 None，则 ADC.read_timed()函数再转换完指定次数后才返回。

下面是一个完整的使用 read_timed()进行周期性 ADC 转换的例子。

回调函数 cb(x)带了一个参数，这是一个整数，表示转换的次数。

```
from lpc55 import ADC
from lpc55 import CTimer
import uarray

a4 = ADC('PIO1_0')   # Pin on Arduino port

# Configure a CTimer to have 100kHz wave
t4 = CTimer(4, 149)     # Config CTimer4, Prescaler = 150. Get 1MHz
t4_3 = t4.channel(3)    # Channel 3
t4_3.action(CTimer.COUNTER_RESET | CTimer.MATCH_TOGGLE)   # On match to reset counter
 and toggle output
t4_3.match(5)               # 1MHz / 10 = 100,000Hz

buf = uarray.array('H', (0,)*100)      # setup a 100 elements (unsigned short) array
```

```
# Callback function to set finish flag
def cb(x):
    global start, finish
    print('Conversion spend: %dus' % lpc55.elapsed_micros(start))
    finish = True

a4.callback(cb)     # setup the callback funtion

finish = False
t4.enable()         # Start the CTimer4 running
start = lpc55.micros()
a4.read_timed(buf, t4)
while not finish:       # Wait for end of conversion
    pass
t4.enable(False)        # Stop CTimer4
```

注：关于回调函数(中断函数)的用法，尤其是硬中断与软中断的区别，请查看 micropython 文档：
http://docs.micropython.org/en/latest/reference/isr_rules.html

3.3.6 ADC.read_timed_multi()：多通道成组转换

ADC.read_timed_multi((adcx, adcy, ...), (bufx, bufy, ...), timer, func)

这是一个静态函数(static method)，用于从多个 ADC 通道进行转换。

该函数按 timer 实体设置的速率，分别从多个 ADC 通道读取转换值，并存入各自对应的缓冲区。

- 第一个参数是一个 tuple，(adcx, adcy, ...)，按顺序给出转换通道的引脚名字
- 第二个参数也是一个 tuple，(bufx, bufy, ...)， 按照上述引脚名字的顺序给出相应的存储数组，所有的数组长度和单元类型必须相同。数组的设置规则与 *3.3.4* 的 buf 参数相同。
- timer 是一个 CTimer 类的实体，与 *3.3.4* 的 timer 参数意义相同。
- func 是一个回调函数，使用方法与 *3.3.5* 相同

该函数返回一个整数，表示转换的次数。

下面是一个使用 read_timed_multi()的例子：

```
from lpc55 import ADC
from lpc55 import CTimer
import uarray

# Configure a CTimer to have 100Hz wave
t4 = CTimer(4, 249)   # Prescaler = 250. Get 600kHz
t4_3 = t4.channel(3)    # Channel 3
t4_3.action(CTimer.COUNTER_RESET | CTimer.MATCH_TOGGLE)   # On match to reset counter and toggle output
t4_3.match(3000)   # 600,000Hz / 200Hz
t4.enable()

bufa = uarray.array('H', (0,)*100)      # setup a 100 elements (unsigned short) array
bufb = uarray.array('H', (0,)*100)      # setup a 100 elements (unsigned short) array

num = ADC.read_timed_multi(('PIO1_0', 'PIO1_9'), (bufa, bufb), t4)
```

注：每次调用 read_timed_multi()，固件都会自动初始化 ADC 硬件模块。

3.4. CTimer 定时器类及函数

CTimer 是 LPC5500 系列的一个标准定时器，在 LPC55S69 中有 5 个 CTimer，编号分别为 0~4。

CTimer 的基本工作原理是一个递增计数器，输入时钟经预分频后作为计数时钟。

每个 CTimer 有四个通道，每个通道都有一个匹配(比较)寄存器和一个捕获寄存器，目前的固件暂时不支持配置捕获寄存器的操作，以下描述都是关于匹配寄存器的功能。

每个匹配寄存器的数值会实时地与递增计数器比较，使用者可以配置当比较匹配时执行以下动作：

- 复位递增计数器为零（即重新开始计数），或停止计数
- 在对应的外部引脚上执行设置、清除或 Toggle 操作

可以同时配置以上对计数器的操作和对引脚的操作。

每个匹配寄存器还有一个相伴的影子寄存器，当计数器开始计数后，用户不能直接写入匹配寄存器而改变设置，必须通过影子寄存器而间接地修改匹配寄存器。每次复位计数器时，影子寄存器的内容会传送到对应的匹配寄存器中。

以下逐一结束 CTimer 类和各函数的用法。

3.4.1 CTimer 创建函数

class lpc55.CTimer(id, ….)

创建一个 CTimer 实体。

- id 是对应 CTimer 的编号，可以是 0~4 之间的整数。
- 如果有其它参数，则直接执行 init()函数的初始化。

3.4.2 CTimer.init()初始化函数

CTimer.init(prescale)

配置预分频系数，并初始化 CTimer。

- prescale 为预分频系数值。
- CTimer 的输入时钟为 150MHz，计数器的计数时钟频率是 150MHz/(prescale+1)

3.4.3 CTimer.deinit()

CTimer.deinit()

关闭 CTimer 定时器。

不用 CTimer 定时器时，关闭它可以节省相应的功耗，并释放资源供其它功能使用。

3.4.4 CTimer.counter()读出计数器

CTimer.counter()

读出当前计数器的计数值。

3.4.5 CTimer.enable()使能计数器

CTimer.enable(en=True)

使能计数器开始计数

- en 为一个布尔值，缺省时默认为 True。
- en=True 表示清零计数器并启动计数。
- en=False 表示停止计数器的计数。

3.4.6 CTimer.mode()设置定时器工作模式

CTimer.mode(mode, freq)

设置 CTimer 的工作模式。

- mode 指定工作模式，目前实现了三种模式：MODE_NORMAL，MODE_PWM 和 MODE_TRIGGER。
- mode=MODE_NORMAL 表示用户可以自由设置 CTimer 的工作方式，见下面说明。
- mode=MODE_PWM 用于配置 PWM 输出，见下面说明。
- mode=MODE_TRIGGER 用于配置 CTimer 作为另一个模块的触发源，例如 ADC.read_timed()。
- freq 仅用于在 MODE_TRIGGER 模式下给出触发频率，其它模式时该参数缺省。由于分频系数只能是整数，不一定能准确得到用户要求的触发频率，因此配置触发模式时，该函数返回一个计算得到的最接近频率值。

3.4.7 CTimer.MODE_TRIGGER 触发模式的用法

该模式是一个特殊配置，CTimer 的所有四个通道都配置为相同的频率，并产生 50%占空比的 PWM 输出，可以将任一通道对应的引脚配置为输出，并得到%50 的 PWM 波形。

该模式适合于与 ADC 模块配合使用，例如 3.3.5 节例子中 CTimer 部分的配置，可以改为如下方式：

```
# Configure a CTimer to have 100kHz wave
t4 = CTimer(4, 149)     # Config CTimer4, Prescaler = 150. Get 1MHz
t4.mode(CTimer.MODE_TRIGGER, 100000)    # 100,000Hz PWM
```

3.4.8 CTimer.channel()获取 CTimerChannel 类实体

CTimer.channel(id)

这个函数返回一个 CTimerChannel 类的实体，用于对相应通道的配置。

- id 的取值范围是 0~3

CTimerChannel 类的函数用于配置相应的通道，根据 CTimer.mode()选取的模式不同，在某些模式下不能使用一些函数，对应关系如下表(Y 表示可以使用相应函数)：

函数 / 模式	Normal	PWM	Trigger
match()	Y		
action()	Y		
shadow()	Y		Y
output()	Y	Y	Y
callback()	Y	Y	Y
pwm_freq()		Y	
pwm_duty()		Y	

3.4.9 CTimerChannel.match()：设置/读取匹配寄存器

CTimerChannel.match(value)

设置/读取匹配寄存器。

- value 为写入匹配寄存器的整数值，缺省时该函数返回当前的寄存器值。

3.4.10 CTimerChannel.action()设置发生匹配时的动作

CTimerChannel.action(act, pwm=False)

该函数指定当匹配成功时，定时器需要执行什么操作。

- act 可以取值 CTimer.COUNTER_RESET 或 CTimer.COUNTER_STOP，表示复位(清零)计数器或停止计数器。
- act 也可以取以下四个值之一：

匹配时对外部引脚的操作	说明
CTimer.MATCH_NONE	不改变外部引脚的电平
CTimer.MATCH_CLEAR	清除外部引脚为‘0’
CTimer.MATCH_SET	设置外部引脚为‘1’
CTimer.MATCH_TOGGLE	翻转外部引脚的状态

- act 可以是 COUNTER_xxx 和 MATCH_xxx 的逻辑或。
- pwm=True 时，表示设置这个通道为 PWM 模式。缺省值为 False。

注：通道配置为 PWM 模式时，表示在这个通道匹配时，自动地复位计数器。详见 LPC55S60 用户手册。

3.4.11 CTimerChannel.output()设置输出引脚

CTimerChannel.output(pin)

配置该通道匹配时的输出引脚。

- pin 应该是 lpc55.Pin()的实体，该函数将重新配置对应的引脚。
- 如果 pin 不是该通道的匹配输出引脚，则抛出 ValueError 异常。
- 如果 pin=None，则曾经配置的引脚将恢复之前 Pin()的配置模式。
- 如果 pin 缺省，则返回曾经配置的引脚名字，或 None。

为方便读者，下表列出 LPC55S69 中 CTimer 的各个通道对应的输出引脚，详细信息请参考用户手册。

左边表格是按通道排序，右边表格是按引脚排序。

引脚	CTimer	通道
PIO0_0	CTimer0	0
PIO0_30	CTimer0	0
PIO0_3	CTimer0	1
PIO0_31	CTimer0	1
PIO0_19	CTimer0	2
PIO1_31	CTimer0	2
PIO1_2	CTimer0	3
PIO1_27	CTimer0	3
PIO0_18	CTimer1	0
PIO1_10	CTimer1	0
PIO0_20	CTimer1	1
PIO1_12	CTimer1	1
PIO0_23	CTimer1	2
PIO1_14	CTimer1	2
PIO1_16	CTimer1	3

引脚	CTimer	通道
PIO0_10	CTimer2	0
PIO1_5	CTimer2	0
PIO1_4	CTimer2	1
PIO1_6	CTimer2	1
PIO0_11	CTimer2	2
PIO1_7	CTimer2	2
PIO0_29	CTimer2	3
PIO1_22	CTimer2	3
PIO0_5	CTimer3	0
PIO1_19	CTimer3	1
PIO0_27	CTimer3	2
PIO1_21	CTimer3	2
PIO0_21	CTimer3	3
PIO0_23	CTimer3	3
PIO0_6	CTimer4	0

引脚	CTimer	通道
PIO0_0	CTimer0	0
PIO0_3	CTimer0	1
PIO0_5	CTimer3	0
PIO0_6	CTimer4	0
PIO0_10	CTimer2	0
PIO0_11	CTimer2	2
PIO0_18	CTimer1	0
PIO0_19	CTimer1	2
PIO0_20	CTimer1	1
PIO0_21	CTimer3	3
PIO0_23	CTimer1	2
PIO0_23	CTimer3	3
PIO0_27	CTimer3	2
PIO0_29	CTimer2	3
PIO0_30	CTimer0	0
PIO0_31	CTimer0	1

引脚	CTimer	通道
PIO1_2	CTimer0	3
PIO1_4	CTimer2	1
PIO1_5	CTimer2	0
PIO1_6	CTimer2	1
PIO1_7	CTimer2	2
PIO1_10	CTimer1	0
PIO1_12	CTimer1	1
PIO1_14	CTimer1	2
PIO1_16	CTimer1	3
PIO1_19	CTimer3	1
PIO1_21	CTimer3	2
PIO1_22	CTimer2	3
PIO1_27	CTimer0	3
PIO1_31	CTimer0	2

3.4.12 CTimerChannel.callback()设置回调函数

CTimerChannel.output(func, hard=False)

当发生匹配时，可以产生中断。该函数设置该通道匹配时的中断回调函数。

- func 给出了回调函数，如果 func 不是 None，则使能该通道的匹配中断，否则关闭中断。
- hard=True 表示硬中断，即回调函数属于内部中断处理程序的一部分，在回调函数中不能使用所有动态分配内存的操作。
- hard=False 时则没有上述限制。这是缺省值。

注：关于回调函数(中断函数)的用法，尤其是硬中断与软中断的区别，请查看 micropython 文档：http://docs.micropython.org/en/latest/reference/isr_rules.html

3.4.13 CTimerChannel.pwm_freq()设置/读取 PWM 频率

CTimerChannel.pwm_freq(freq)

只能用在 mode=MODE_PWM 的定时器，见 3.4.7。

本函数设置相应的通道控制 PWM 的频率，配合 pwm_duty()函数控制 PWM 占空比。

- freq 为需要获得的 PWM 频率。由于内部都是整数分频，实际的 PWM 频率会与设置的参数有偏差，本函数会返回实际计算得出的频率
- 当参数缺省时，本函数返回已经设置的，实际计算得出的 PWM 频率。

3.4.14 CTimerChannel.pwm_duty()设置/读取 PWM 占空比

CTimerChannel.pwm_duty(duty)

只能用在 mode=MODE_PWM 的定时器，见 3.4.7。

本函数需与 pwm_freq()配合使用，用于设置/读取相应通道的 PWM 占空比。

- duty 为需要获得的 PWM 占空比。由于内部都是整数分频，实际的 PWM 占空比会与设置的参数有偏差，本函数会返回实际计算得出的占空比。
- 当参数缺省时，本函数返回已经设置的，实际计算得出的占空比。

注：本函数的 duty 参数和返回整数，都是占空比的 10 倍数值。例如占空比为 50%，则以 500 表示；占空比为 90%，则以 900 表示。

注：设置的占空比表示的是输出信号为低时的百分比，例如数值 100 表示信号的 10%为'低'，90%为'高'。

以下是一个 PWM 输出的例子，在引脚 PIO1_31 上连接了一个 LED 灯，低电平灯亮，高电平灯灭。本例用 PWM 占空比控制灯的明暗：

```
from lpc55 import CTimer
t0 = lpc55.CTimer(0, 149)    # 1MHz
t0.mode(CTimer.MODE_PWM)

t0_3 = t0.channel(3)
t0_3.pwm_freq(1000)          # Select channel 3 control PWM frequency

t0_2 = t0.channel(2)
t0_2.pwm_duty(100)           # Select channel 2 as PWM output
red = Pin('PIO1_31')
t0_2.output(red)             # Select PIO1_31 as output pin

t0.enable()                  # Start running
```

可以随时使用 pwm_duty()调节灯的明暗：

```
t0_2.pwm_duty(900)    # 更明亮
t0_2.pwm_duty(100)    # 更暗
```

也可以随时使用 pwm_freq()灯的闪烁频率：

```
t0_3.pwm_freq(10)     # 10Hz 频率，人眼可见闪烁
```

注：pwm_freq()和 pwm_duty()需要配对使用，先调用 pwm_freq()再调用 pwm_duty()；用 pwm_freq()修改 PWM 频率后，需要再次用 pwm_duty()刷新占空比，否则内部计算结果可能出现错误。

3.5. ENC 旋转编码器类和函数

在 LPC5500 系列中，芯片里没有硬件的旋转编码器模块，因此在 micropython 的实现中使用软件加引脚中断或定时器实现读取旋转编码器状态的功能。

注：本方案是软件方案，速度较慢，只适合与读出手动的旋钮，而不适合测量电机转速的旋转编码器。

注：本方案实现了旋转编码器的去抖动，可以过滤小于 270us 的脉冲信号。

3.5.1 ENC 创建函数

class lpc55.ENC(pinA, pinB[, tim])

创建一个与指定引脚绑定的 ENC 实体。

- pinA 和 pinB 分别是两个引脚的 Pin()实体，对应 A-B 相连接的引脚。要求这两个引脚配置为输入即可。
- tim 是一个 CTimer()的实体，用 CTimer()创建这个实体后，不必对它做任何操作，这个定时器完全由 ENC 控制使用，不得再在其它地方用到它。
- 当 tim 参数缺失时，固件内部使用引脚中断机制处理旋转编码器，如果提供了 tim 参数，则改用定时器处理旋转编码器。
- 使用哪一种方法(引脚中断或定时器)，就看使用者的整个系统中有哪种资源可以用。LPC55S69 中有 8 个引脚中断源和 5 个 CTimer 可用，使用者可以任意选择。

3.5.2 ENC.run()启动工作

ENC.run()

启动对输入信号的采样处理。

执行该函数后，可以用随后的 read()函数读出旋转编码器的状态。

3.5.3 ENC.read()读出转换数值

ENC.read()

读出旋转编码器的转换数值。返回的转换数值按 48ms 为单位表示。

这个函数返回一个 list，其中的每一个分量都是一个具有 3 个整数的 tuple：

1. 第一个整数表示自 run()之后的时间，按每 48ms 增加。
2. 第二个整数表示在本 48ms 周期内，检测到多少顺时针的状态变换。
3. 第二个整数表示在本 48ms 周期内，检测到多少逆时针的状态变换。

使用者可以根据这 3 个数据，计算出速度和旋转的格数。

上述第二、三个整数表示的状态变换是指 A-B 相上电平变换的次数，或检测到的边沿次数。

根据测试目前看到两种旋转编码器，一种是转动一格时状态变换了 2 次，另一种是转动一格状态变换了 4 次。读者可以自行测试一下自己的编码器，以确定这个数值。

使用者需要定期读出转换数值，如果不关心速度，只需将所有顺时针状态数目相加，除以 2 或 4 即得知正转的格数，反转的格数计算方式相同。

注：如果某次读取后，经相加计算后的结果为奇数，最好把除以 2 或 4 之后的余数，合并到下一次读取后的计算中，否则在转动速度较慢时，可能会漏算 1-2 格的旋转。

3.5.4 ENC.callback()设置回调函数

前面已经提到，使用者需要定期读出转换数值，根据内部缓存区的大小，调用 run()之后，如果超过一定时间不读取数据(大约是 256*48ms=12.3 秒)，则记录的数据会溢出丢失。

本函数让使用者可以设置一个回调函数，在这个回调函数中及时读出数据，避免数据溢出丢失。

ENC.callback(func)

设置回调函数。

- func 给出了回调函数。

回调函数有一个参数，表示缓冲区里数据组的数量。目前固件设定内部缓存区大小为存放 256 组数据，当缓存区半满时会调用回调函数，所以通常回调函数的参数是 128。由于固件会经常修改，以后这个数值有可能会变化。

以下是使用 callback()的一个完整例子：

```
ENC1 = lpc55.Pin('P15_3', Pin.IN)
ENC2 = lpc55.Pin('P15_2', Pin.IN)

enc = lpc55.ENC(ENC1, ENC2)
def result(size):
    steps = enc.read()
    for dat in steps:
        print('time={:>5}, pos={:>2}, neg={:>2}'.format(dat[0], dat[1], dat[2]))

enc.callback(result)
enc.run()
```

注：即使是旋转编码器没有动作，内部缓冲区里也会一组一组地记录数据，而 read()函数只返回那些正转或反转不为 0 的数据组，不返回 2 个数值均为 0 的数据而只从内部缓冲区清除这个数据组。因此在这个例子中，回调函数 result()中的 read()操作，可能会返回一个空的 list，代表这段期间编码器没有变化。

3.6. ExtInt 外部引脚中断类和函数

ExtInt 是利用 LPC5500 的 PINT 模块，用于处理外部引脚中断的操作。

3.6.1 ExtInt 创建函数

class lpc55.ExtInt(pin, mode, pull, callback)

返回一个 ExtInt 实体。

- pin 是一个 lpc55.Pin 的实体或一个引脚的名字。
- mode 是描述要求在哪个外部电平的边沿产生外部中断，取值可以是 IRQ_RISING、IRQ_FALLING 或 IRQ_RISING_FALLING，分别表示上升沿、下降沿或双边沿。
- pull 表示如何配置这个引脚的上下拉，取值可以是 lpc55.Pin.PULL_NONE、lpc55.Pin.PULL_UP 或 lpc55.Pin.PULL_DOWN，分别表示无上下拉、上拉或下拉。
- callback 是发生指定的外部中断后的回调函数。

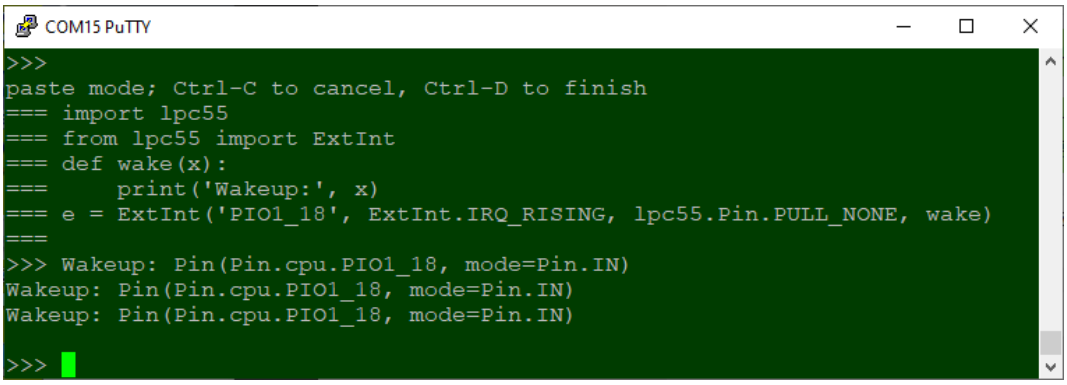
注：这个回调函数是软中断。关于回调函数(中断函数)的用法，尤其是硬中断与软中断的区别，请查看 micropython 文档：http://docs.micropython.org/en/latest/reference/isr_rules.html

回调函数自带一个参数，这个参数是产生这个中断的引脚 Pin 实体。如果用户有多个引脚配置了外部中断，可以使用一个相同的回调函数，通过回调函数的参数而得知是哪个引脚的中断。

下面是一个例子，这个例子里的引脚'PIO1_18'是开发板上的一个按钮，当按下按钮再释放时会产生一个上升沿电平，从而触发配置的外部中断：

```
import lpc55
from lpc55 import ExtInt
def wake(x):
    print('Wakeup:', x)
e = ExtInt('PIO1_18', ExtInt.IRQ_RISING, lpc55.Pin.PULL_NONE, wake)
```

以下是执行这段代码后，按压按钮后的显示：



这里的回调函数 wake()里就是一句话，打印"Wakeup"和这个引脚的信息。

3.6.2 ExtInt.line()查看外部中断的编号

ExtInt.line()

在 LPC55S69 中有 8 个外部引脚中断线路，任何一个引脚都可以通过 8 个线路之一产生中断，但最多只有 8 个引脚可以产生中断。

这个函数返回指定引脚与外部中断线路对应的编号。

3.6.3 ExtInt.deline()释放外部中断线路

ExtInt.deline(line)

这个函数用于把与指定引脚的外部引脚中断线路断开。

由于芯片中只有 8 个外部引脚中断线路，该函数可以释放相应的资源供其它引脚使用。

- line 是引脚对应的外部中断编号，可以调用 ExtInt.line()查看对应的编号。

注：当需要更新回调函数时，也需要先断开引脚对应的中断线路，再重新调用 ExtInt 创建函数设置新的回调函数，固件将重新分配中断线路。

3.6.4 ExtInt.disable()和 ExtInt.enable()屏蔽和开启中断相应

ExtInt.disable()

ExtInt.enable()

这两个函数用于关闭和开启对应的中断。

3.6.5 ExtInt.swint()软件触发中断

ExtInt.swint()

通过软件直接触发对应的中断。

这个函数通常用于调试中断处理函数(回调函数)。

3.7. I2C 类和函数

lpc55.I2C 类直接控制芯片的硬件 I2C 模块，可以获得最大的性能。

3.7.1 I2C 创建函数

class lpc55.I2C(bus, ...)

返回对应一个 I2C 硬件模块的实体。

- bus 是一个 0~7 的整数，表示 I2C 硬件模块的编号。
- 如果没有其它参数，则不初始化硬件。如果还有其它参数，则与 I2C.init()参数相同，并继续执行初始化。

目前固件中按照逐飞开发板的硬件配置，而配置了各个模块的连接，如下表：

模块	SCL	SDA	开发板上接口脚位
I2C0	PIO0_30	PIO0_29	P1_6, P1_4
I2C1	PIO1_11	PIO1_10	P2_6, P2_4
I2C2	PIO1_25	PIO1_24	P3_6, P3_4
I2C3	PIO0_2	PIO0_3	P4_6, P4_4
I2C4	PIO1_20	PIO1_21	P5_6, P5_4
I2C5	PIO1_15	PIO1_14	P12_1, P12_2
I2C6	PIO0_22	PIO1_13	P6_6, P6_4
I2C7	PIO1_30	PIO1_29	P15_7, P15_8

特别注意：在 LPC5500 中，芯片内部 I2C、SPI、UART 和 I2S 都是共用相同的硬件，当某个功能占据了某个硬件模块的编号，其它功能则不能再用这个编号。例如使用了 I2C1 之后，就不能再使用 SPI1 了，否则创建函数会抛出异常。

3.7.2 I2C.init()和 I2C.deinit()

I2C.init(mode, *, addr=18, baudrate=400000)

I2C.deinit()

初始化和关闭 I2C 模块。

- mode 是 I2C.MASTER 或 I2C.SLAVE。目前没有测试 Slave 模式。
- addr 是一个 7 位的本机地址，只在 Slave 模式才有意义。
- baudrate 是 SCL 时钟的速率，只在 Master 模式才有意义。

前面的创建函数，可以带入 init()的参数，在创建的同时执行初始化。

3.7.3 I2C.is_ready()

这个函数的功能暂没有实现。

3.7.4 I2C.scan()

这个函数的功能暂没有实现。

3.7.5 I2C.send()发送数据

I2C.send(data, addr)

发送数据。

- data 可以是一个整数，也可以是一个缓冲区。如果是整数，则发送整数对应的一个字节。如果是缓冲区，例如一个 bytearray 实体，则发送对应的一串字节。

- addr 是一个整数，表示 Slave 设备的地址。

本函数暂不支持 Slave 模式。

3.7.6 I2C.recv()接收数据

I2C.recv(data, addr)

接收数据

- data 可以是一个整数，表示需要接收的字节数目，此时本函数将返回一个 bytearray 缓冲区，包含 data 长度的数据。
- data 也可以是一个缓冲区, 例如一个 bytearray 实体，用于接收一串字节，给出的缓冲区长度就是要接收的数据长度。
- addr 是一个整数，表示 Slave 设备的地址。

本函数暂不支持 Slave 模式。

3.7.7 I2C.mem_write()写存储器

I2C.mem_write(data, addr, memaddr, *, addr_size=8)

向存储器写入数据。

- data 和 addr 的意义与 I2C.send()相同，见 3.7.5。
- memaddr 为 I2C 设备中的存储器地址。
- addr_size 用于指定上述存储器地址的位数，可能的取值为 8、16、24、32。默认为 8 位长度。

本函数用于组织成存储器的 I2C 设备，例如 I2C 的 EEPROM。

3.7.8 I2C.mem_read()读存储器

I2C.mem_read(data, addr, memaddr, *, addr_size=8)

从存储器读出数据。

- data 和 addr 的意义与 I2C.recv ()相同，见 3.7.6。
- memaddr 和 addr_size 与 I2C.mem_write()的参数意义相同。

本函数用于组织成存储器的 I2C 设备，例如 I2C 的 EEPROM。

3.8. LED 类和函数

这是 micropython 的一个基本类，封装了用于控制板载 LED 的基本函数。

3.8.1 LED 创建函数

class lpc55.LED(id)

返回对应一个 LED 的实体。

- id 是 LED 的编号。目前配置 LED 对应为开发板上的三色 LED。
  - id=1 对应红色 LED
  - id=2 对应绿色 LED
  - id=3 对应蓝色 LED

3.8.2 on()、off()和 toggle()点亮、熄灭和翻转

LED.on()

LED.off()

LED.toggle()

三个函数分别用于点亮、熄灭和翻转 LED 的显示状态。

3.8.3 intensity()设置明暗度

LED.intensity(value)

设置 LED 的明暗度。

- value 为明暗系数，取值范围是 0~255；255 表示最亮，0 表示最暗(熄灭)。
- 如果 value 的数值超出 0~255 的范围，当 value>0 时 LED 全亮，当 value<0 时 LED 熄灭。
- 如果未提供参数，函数则返回已经设置的明暗系数数值。

3.8.4 breath()配置呼吸灯

LED.breath(length=2000, dense=100)

配置 LED 以呼吸灯(渐明渐暗交替)方式显示。

- length 给出每个周期的时间长度，单位是 ms。默认值是 2000ms。
- dense 表示每个周期中最亮时的明暗度，默认值是 100。

以下是一个简单的例子：

```
from lpc55 import LED

red=LED(1)
red.breath()    # Use default setting

red.breath(3000) # Set cycle time as 3s
```

3.9. Pin 引脚控制类和函数

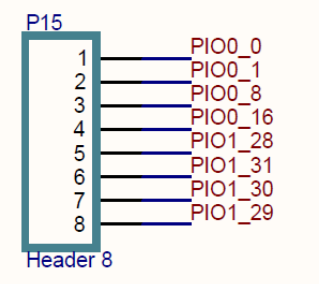
可以通过这个类直接控制通用 I/O 引脚的状态，并查看引脚的各种配置信息。

3.9.1 Pin 创建函数

class lpc55.Pin(name,...)

返回对应引脚的实体。

- name 是引脚的名称，这个名称可以是 CPU 自身的名称，例如 PIO1_12 是连接到红色 LED 的引脚，可以用这个名称创建对应红色 LED 的引脚实体。
- name 也可以是板上插口的名称，需要对照开发板的线路图找出对应的引脚。例如板上 Arduino 接口 P15 的连线图如右图，这个插口上从上至下每一个信号的名称则分别是 P15_1、P15_2、…、P15_8。
- 完整的引脚名称列表，详见 3.9.7 节。
- 如果引脚名称后面还有其它参数，则与 init()相同用于初始化引脚。



3.9.2 Pin.init()引脚初始化

Pin.init(mode, pull=Pin.PULL_NONE, /, value, func=0)

初始化引脚

- mode 描述引脚的类型，可以有如下数值：
  - Pin.IN：配置引脚为输入
  - Pin.OUT：配置引脚为输出
  - Pin.OD：配置引脚为开漏输出

- Pin.FUNC：配置引脚为功能引脚(详见 LPC55S69 用户手册)
- Pin.ANALOG：配置引脚为模拟引脚。一般用不到这个选项，ADC 功能已经封装在 ADC 类中。

➢ pull 描述引脚内部的上下拉配置，可以有如下数值：

- PULL_UP：内部上拉
- PULL_DOWN：内部下拉
- PULL_REPEATER：Repeater 模式(详见 LPC55S69 用户手册)
- PULL_NONE：无内部上下拉。当调用时不带 pull 参数时，此为默认值。

➢ value 为引脚的初始电平值。对于输出引脚才有意义。此为关键字参数。

➢ func 为引脚的功能选项。此为关键字参数，当 mode=Pin.FUNC 时需要这个参数。一般用户不必提供这个参数，常用功能都已经封装在相应的类中，例如 I2C、UART 等。

注：一般用户用不到 mode=Pin.ANALOG 和 mode=Pin.FUNC 的选项，这些选项只是在特殊情况并非常熟悉固件内部运行的情况下才会用到，建议不要随便尝试，以免影响其它部分运行。

3.9.3 Pin.value()的设置引脚电平函数

Pin.value([value])

Pin.on()

Pin.off()

Pin.low()

Pin.high()

除了 value()外，这些函数都是用于改变输出引脚的输出电平。

➢ Pin.value()函数如果有参数，则功能是设置输出引脚的电平；如果没有参数则是读出引脚当前的状态，输出引脚或输入引脚都是用这个方式读。

➢ Pin.on()和 Pin.high()都是设置输出引脚为高。对输入引脚无效。

➢ Pin.off()和 Pin.low()都是设置输出引脚为低。对输入引脚无效。

3.9.4 Pin.irq()设置引脚中断

Pin.irq(/ handler, edge, hard=False)

配置引脚的外部引脚中断功能

➢ handler 是一个回调函数，当这个引脚上有 edge 指定的边沿变化时，这个回调函数被调用。

➢ handler=None 时表示取消该引脚的外部引脚中断功能。

➢ edge 用于指定引脚上产生中断的边沿：

- Pin.IRQ_RISING：上述沿产生中断
- Pin.IRQ_FALLING：下降沿产生中断
- 如果希望两个边沿都产生中断，则可以使 Pin.IRQ_RISING | Pin.IRQ_FALLING。
- edge 参数缺省为双边沿

➢ hard 是一个布尔值，True 表示产生一个硬中断，False 表示产生一个软中断。关于硬中断与软中断的区别，请查看 micropython 文档：http://docs.micropython.org/en/latest/reference/isr_rules.html

调用该函数不带任何参数时，表示取消该引脚的外部引脚中断功能。

注：这个函数的功能可以使用 ExtInt 类实现(见 3.6 节)，唯一区别是如果需要产生硬中断，可以使用这个函数，但要注意硬中断的限制条件。

3.9.5 Pin.name()等引脚配置信息

以下这些函数用于读取引脚实体的各项配置信息。

Pin.name()

返回该引脚对应的 CPU 名称(详见 3.9.7 节)。

Pin.names()

返回一个两个分量的 tuple，第一个分量是该引脚对应的 CPU 名称，第二个是该引脚对应的板上名称。

Pin.port()

返回该引脚对应 GPIO 端口号。

Pin.pin()

返回该引脚对应 GPIO 端口中的引脚号。

Pin.mode()

返回该引脚配置为哪种模式，见 Pin.init()的 mode 参数。

Pin.pull()

返回该引脚配置的上下拉信息，见 Pin.init()的 pull 参数。

Pin.func()

返回该引脚配置的功能信息，见 Pin.init()的 func 参数。

3.9.6 Pin.func_list()列出引脚所有可能的功能

Pin.func_list()

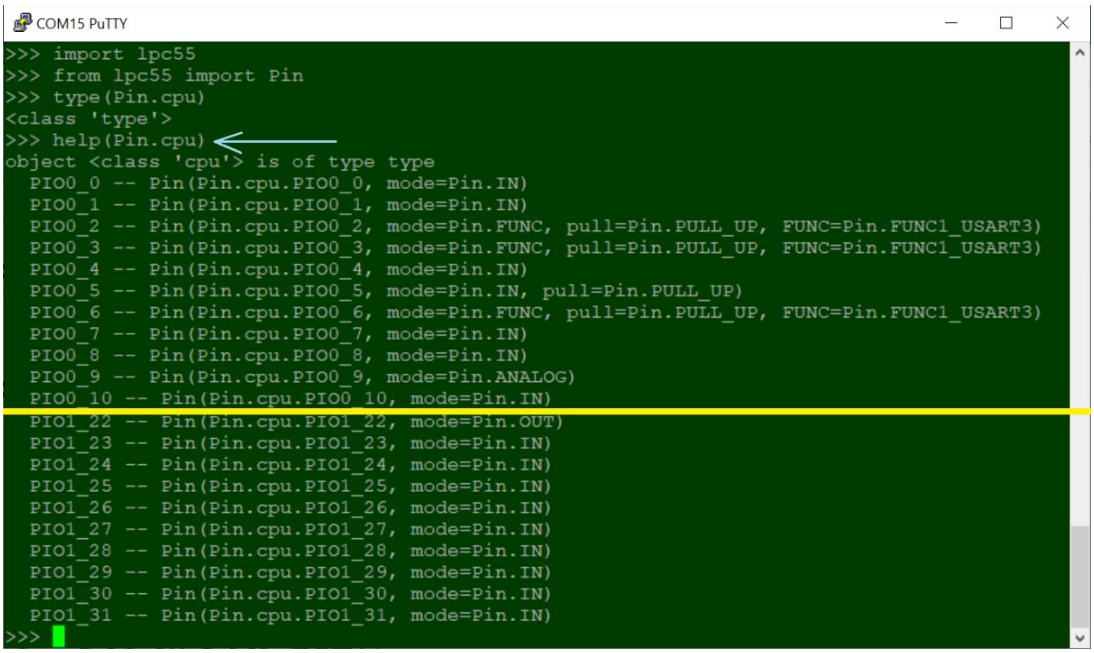
返回一个 tuple，包含所有该引脚可能的功能。

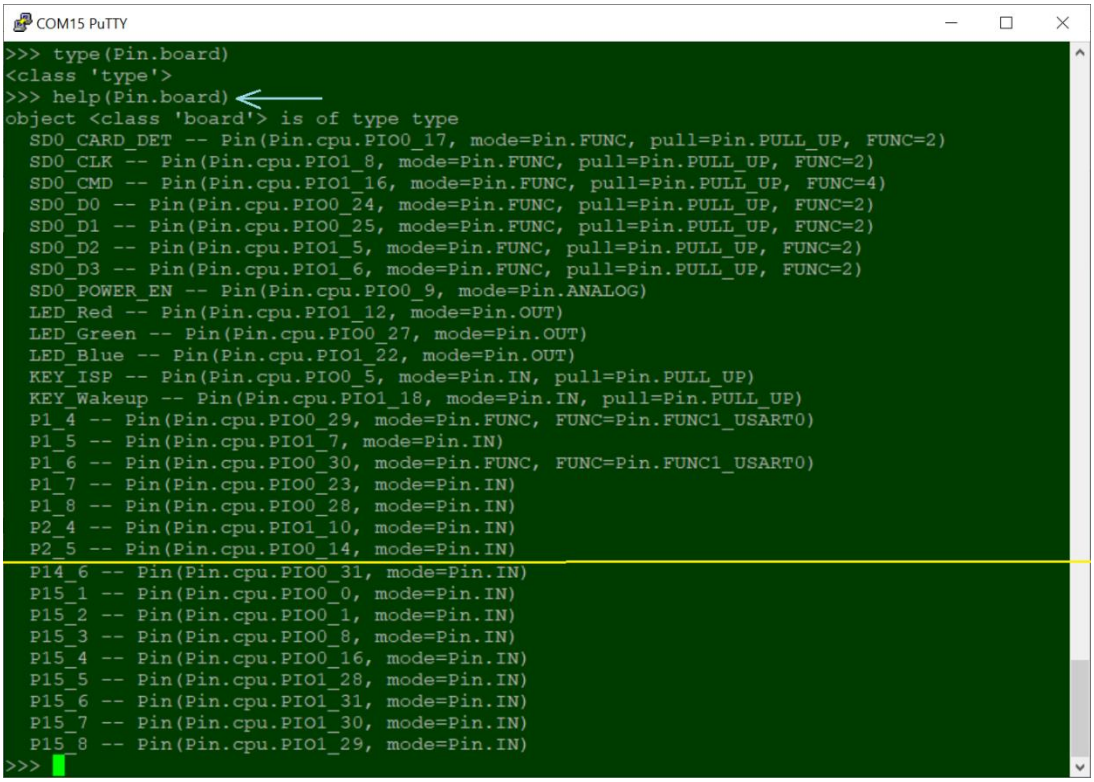
tuple 的每个分量，都是 PinFunc 类的实体，详见以下说明（）。

3.9.7 Pin.cpu 和 Pin.board

这是两个特殊类，这两个类没有操作函数，只有一些常数，分别表示所有 CPU 引脚的名称和开发板接口上引脚的名称。

一般可以用 help 函数查看所有可能的引脚和名称，下面是 Pin.cpu 和 Pin.board 列出的引脚（显示很长，黄线表示省略部分）：





从这两图中还可以看出，每个引脚当前的配置情况；在第二张图里，能够清楚地看到板上插口的编号与芯片的引脚之间的对应关系。

例如 3.9.1 节提到板上的红色 LED 连接的引脚的 CPU 名称是 PIO1_12，从上图可以知道这个引脚的名称也可以是 LED_Red，读者可以自行测试一下。

当任何地方需要引脚名称时，可以自行照此查看正确的名称。

3.9.8 PinFunc 类和函数的用法

PinFunc 类的实体只能从上述的 Pin.func_list()（见 3.9.6 节）得到，它没有创建函数，只有以下几个函数可以查看相应的属性：

PinFunc.index()

返回对应功能的编号。这个编号可用于 init()中的 func 参数（见 3.9.2 节）。

PinFunc.name()

返回引用上述编号时使用的常数名称，参见下面的例子。

PinFunc.reg()

返回对应功能的硬件模块，在芯片中寄存器组的基地址，可以用于 lpc_mem 访问，见 2.2 节。

例如，下面的操作截图，就是用 LED_Red 这个名称，查看该引脚所有功能的操作：

```
>>> red= Pin('LED_Red',Pin.OUT,value=0)
>>> funcs = red.func_list()
>>> funcs
[Pin.FUNC2_USART6, Pin.FUNC2_SPI6, Pin.FUNC2_I2S6, Pin.FUNC3_CT1, Pin.FUNC5_HSPI]
>>> f2 = funcs[2]
>>> f2.index()
2
>>> f2.name()
'FUNC2_I2S6'
>>> hex(f2.reg())
'0x40097000'
>>> help(f2)
object Pin.FUNC2_I2S6 is of type PinFunc
  index -- <function>
  name -- <function>
  reg -- <function>
>>>
```

3.10. RTC 类和函数

这是用于读写片上的内置实时时钟 RTC。

3.10.1 RTC 创建函数

class lpc55.RTC()

返回 RTC 实体。

3.10.2 RTC.datetime()设置/读取日期时间

RTC.datetime([param])

初始化日期和时间，或读出日期和时间。

- param 为一个具有 8 个分量的 tuple，每个分量都是一个整数，分别表示年、月、日、周日、时、分、秒、子秒。其中的取值是：
  - 年：公元纪年的年份，例如今年是 2021 年，则用 2021 表示。
  - 月：取值 1~12。
  - 日：取值 1~31。
  - 周日：用 1~7 分别表示星期一、二、…、六和星期天。固件忽略此值，芯片内部自行计算。
  - 时：取值 0~23。
  - 分、秒：取值 0~59。
  - 子秒：任意值。LPC55S69 不支持子秒，所以固件内部忽略这个数值。
- 当不带参数调用这个函数时，返回一个 tuple，内容和含义与上述设置时相同。

3.10.3 RTC.wakeup()设置唤醒闹钟

RTC.wakeup(second, callback)

设置一个倒计时闹钟，当设置的时间到时则调用回调函数。

- second 为倒计时的时间，单位为秒。
- callback 为回调函数，当调用 wakeup()之后 second 秒之后，这个回调函数会被调用。回调函数自带一个整数参数，这是一个内部中断线路的编号，用户可以忽略。

如果不带参数调用这个函数时，则关闭已经设置的倒计时闹钟。

闹钟的设置是一次性的，不会充分，即设置的时间到时后，闹钟会自动关闭。

右图是一个使用示例：

```
>>> Ctrl-E
paste mode; Ctrl-C to cancel, Ctrl-D to finish
=== import lpc55
=== from lpc55 import RTC
===
=== rtc=RTC()
=== clock = (2021,3,11,0,10,5,20,0)
=== rtc.datetime(clock)      # Initialize rtc
===
=== def awake(x):
===     print('Awaking @', rtc.datetime())
=== rtc.datetime()           # show current time
=== print()                  #
=== rtc.wakeup(5, awake)     # alarm in 5 second
===
(2021, 3, 11, 4, 10, 5, 20, 0)

>>> Awaking @ (2021, 3, 11, 4, 10, 5, 25, 0)
rtc
RTC: 2021-03-11, 10:05:31
```

3.11. SDCard 的使用

在 lpc55 模块中 SDCard 类用于在底层访问操作 SD 卡。

SD 是这个类的实体，可以直接用于挂载。不使用 lpc55.SD 时，也可以自行创建 SDCard 实体：

```
sd = lpc55.SDCard()
```

强烈建议使用文件系统的方式访问 SD 卡，而不要用本节的读写函数，在不熟悉 SD 卡的文件系统组成的情况下直接操作 SD 卡扇区，很可能会破坏上面的文件系统，导致数据丢失或不能使用。

注：一旦 SD 卡的文件系统被破坏而不能使用后，只能将这张卡放到 PC 上重新进行格式化，然后才能在本系统下继续使用。如果不慎破坏了 SD 卡的启动数据块，这张卡有可能不能再被 PC 识别，卡只能作废或找更高级的工程师恢复。

3.11.1 挂载 SD 卡

一般情况下，如果卡槽中有 SD 卡，则在板子上电时，会自动挂载。

当 SD 卡没有挂载在文件系统时，可以用以下命令挂载到根目录下：

```
>>>
>>> import lpc55
>>> import uos
>>> uos.listdir('/')                    # Check root directory
['flash']  ——— 根目录只有flash，没有SD卡
>>> lpc55.mount(lpc55.SD, '/sd')        # Mount SD to /sd
>>> uos.listdir('/')                    # Check root directory again
['flash', 'sd']  ——— SD卡已经挂载到根目录
>>> uos.chdir('/sd')                    # Change current directory to /sd
>>> uos.listdir()                       # List files under current directory
['LOST.DIR', '.android_secure', 'DCIM', 'quicknote', 'notegallery', 'Android', 'Download'
, 'tencent', '360', 'SKIPSD', 'DevIcon.fil', 'DevLogo.fil', 'Playlists', 'Albums', '.udst
ate', 'Photo', '.dir_com.qihoo.appstore', 'Movies', '\x0f\x0f\x0f\x0f\x08\x01~1', 'System
 Volume Information', 'aia_doc', 'boot.py', 'mpy', '_main.py']  SD卡根目录下的文件或目录
>>>
```

以上操作截图显示了挂载 SD 卡之前和之后，对文件系统根目录列表的情况。

3.11.2 SDCard 创建函数

class lpc55.SDCard()

返回 SDCard 实体，用于挂载或 SD 卡的底层操作。

3.11.3 SDCard.present()查看卡槽是否有卡

SDCard.present()

返回一个布尔值，表示卡槽中是否有 SD 卡存在。

注：目前系统不支持热插拔。

3.11.4 SDCard.info()查看 SD 卡信息

SDCard.info()

返回一个 tuple，包含三个整数，分别是(总存储容量，每个数据块大小，卡属性标志)

总存储容量和数据块大小的单位是字节。

卡属性标志表示一些卡的特性，仅供内部使用。

例如以下返回值表示每个块大小是 512 字节，共有 15997075456 字节，经计算知道这是个 16GB 的卡。

```
>>> lpc55.SD.info()
(15997075456, 512, 14)
```

3.11.5 SDCard.readblocks()/writeblocks()读写数据块

SDCard.readblocks(blk_num, buffer)

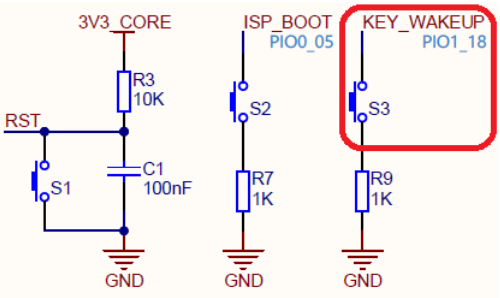
SDCard.writeblocks(blk_num, buffer)

这两个函数分别读写若干个数据块。

- blk_num 是一个整数，表示从数据块地址。
- buffer 是一个以数据块大小(通常为 512 字节)为单位的缓冲区，一般为 bytearray。
- 函数返回一个布尔值，表示读写操作是否成功。

3.12. Switch 类和函数

这个类对应开发板上的 wakeup 按钮，右图是开发板原理图的局部。



3.12.1 Switch 创建函数

class lpc55.Switch()

返回 wakeup 按钮的实体

3.12.2 Switch.value()读取按钮状态

Switch.value()

返回布尔数值，表示 wakeup 按钮是否被按下。

3.12.3 Swtich.callback()设置回调函数

Switch.callback(func)

设置或取消回调函数。

- func 是按钮按下后要调用的回调函数。如果 func=None 表示取消回调函数。

这是一个简单的演示截图：

```
>>> sw.value()
False
>>> sw.value()
True
>>> sw.callback(lambda: print('Hello'))
>>>
>>> Hello
Hello
```

3.13. SPI 类和函数

lpc55.SPI 类直接控制芯片的硬件 SPI 模块，可以获得最大的性能。

3.13.1 SPI 创建函数

class lpc55.SPI(bus, ...)

返回对应一个 SPI 硬件模块的实体。

- bus 是一个 0~8 的整数，表示 SPI 硬件模块的编号。
- 如果没有其它参数，则不初始化硬件。如果还有其它参数，则与 SPI.init()参数相同，并继续执行初始化。

目前固件中按照逐飞开发板的硬件配置，而配置了各个模块的连接，如下表：

模块	MOSI	MISO	SCK	SSEL	SSEL0	开发板上接口脚位
SPI0	PIO0_29	PIO0_30	PIO0_28	PIO1_7	PIO0_23	P1_4, P1_6, P1_8, P1_5, P1_7
SPI1	PIO1_10	PIO1_11	PIO0_7	PIO0_14	PIO0_13	P2_4, P2_6, P2_8, P2_5, P2_7
SPI2	PIO1_24	PIO1_25	PIO1_23	PIO1_27	PIO1_26	P3_4, P3_6, P3_8, P3_5, P3_7
SPI3	PIO0_3	PIO0_2	PIO0_6	PIO0_21	PIO0_20	P4_4, P4_6, P4_8, P4_5, P4_7
SPI4	PIO1_21	PIO1_20	PIO0_4	PIO0_19	PIO0_18	P5_4, P5_6, P5_8, P5_5, P5_7
SPI5	没有固定配置，需自行配置。					
SPI6	PIO1_13	PIO0_22	PIO0_10	PIO1_17	PIO0_15	P6_4, P6_6, P6_8, P6_5, P6_7
SPI7	没有固定配置，需自行配置。					
SPI8	PIO0_26	PIO1_3	PIO1_2	PIO1_1		P12_7, P12_6, P12_5, P12_8

特别注意：在 LPC5500 中，芯片内部 I2C、SPI、UART 和 I2S 都是共用相同的硬件，当某个功能占据了某个硬件模块的编号，其它功能则不能再用这个编号。例如使用了 I2C1 之后，就不能再使用 SPI1 了，否则创建函数会抛出异常。

3.13.2 SPI.init()和 SPI.deinit()

SPI.init(mode, baudrate=500000, *, polarity=1, phase=0, bits=8, ssel=4, firstbit=SPI.MSB)

SPI.deinit()

初始化和关闭 SPI 模块。

- mode 是 SPI.MASTER 或 SPI.SLAVE。目前没有测试 Slave 模式。
- baudrate 是传输速率，只在 Master 模式才有意义。
- polarity 和 phase 共同描述 SPI 的采样模式。
- bits 指示传输的每个帧的位数，有效值是 4~16.
- ssel 表示 SPI 传输时，自动设置哪条片选信号线。
  - 0 表示自动设置 SSEL0

- 1 表示自动设置 SSEL1
- 2 或 3 表示自动设置 SSEL2 或 SSEL3。不是每个硬件的 SPI 模块都有 SSEL2 或 3，SPI 函数内部不会自动初始化对应的引脚，需要用户自行用 Pin()来配置这个功能。
- 4 表示不自动设置任何 SSELx 片选信号。

➢ firstbit 表示每个帧的位顺序。

- SPI.MSB 表示高位在前。
- SPI.LSB 表示低位在前。

前面的创建函数，可以带入 init()的参数，在创建的同时执行初始化。

3.13.3 SPI.send()发送数据

SPI.send(data, *, timeout=5000)

发送数据。

➢ data 是要发送的数据。data 可以是一个整数，表示要发送一个字节；或是一个包含发送数据的缓冲区。

➢ timeout 是发送超时等待时间，单位是 ms。

注：一般情况下不需使用 timeout 参数。

3.13.4 SPI.recv()接收数据

SPI.recv(data, *, timeout=5000)

接收数据。

➢ data 可以是一个整数，表示要接收的字节数目，函数将返回一个包含收到数据的缓冲区。

➢ data 也可以是要接收数据的缓冲区，接收长度就是缓冲区的长度。

➢ timeout 是发送超时等待时间，单位是 ms。

注：一般情况下不需使用 timeout 参数。

3.13.5 SPI.send_recv()

SPI.send_recv(send, recv=None, *, timeout=5000)

发送的同时接收数据。

➢ send 的含义与 send()函数的 data 参数具有相同含义。

➢ recv 的含义与 recv()函数的 data 参数具有相同含义。

➢ recv 可以使用与 send 相同的缓冲区。

➢ 如果 recv=None，表示需要返回一个新的缓冲区。

➢ timeout 是发送超时等待时间，单位是 ms。

注：一般情况下不需使用 timeout 参数。

3.13.6 SPI.config()配置帧位数和 SSEL 线

该函数用于在 init()之后需要临时变更帧位数或 SSEL 线的情形，可以用于一些特殊场合。例如模拟某些协议时需要随时调整帧位数。

SPI.config(*, bits=None, ssel=None)

➢ bits 为帧位数，取值范围是 4~16

- ssel 是需要自动设置的 SSEL 信号，取值范围是 0~4，详见 SPI.init()的说明。需要临时更改 SSEL 信号的一个例子是，把 SSEL 接到 LED 或 LCD 模块的 D/C 信号。

3.13.7 其它函数

SPI 类还有几个函数：SPI.read()、SPI.readinto()、SPI.write()和 SPI.write_readinto()，关于这几个函数的用法，请参考 micropython.org 的文档，关于 machine.SPI 部分。

3.13.8 SPI 相关引脚的配置

在创建一个 SPI 实体时，给定了硬件模块的编号之后，在 SPI.init()中 3.13.23.13.1 节的表格里对应的引脚，除了 SSEL0 以外的引脚都会被初始化为相应的 SPI 功能。

如果不会用到某(几)个引脚，而想作为其它功能使用时，可以在 SPI.init()之后，使用 lpc55.Pin()指定该引脚作为其它功能。例如用 SPI 驱动一个只写的设备时，用不到 MISO 引脚，这个引脚就可以用来做其它功能。

如果希望使用指定硬件模块的其它引脚，而不是 3.13.1 节表格中默认的引脚，也可以在 SPI.init()之后用 lpc55.Pin()将哪些默认的引脚配置为其它功能，而把其它相应的引脚配置为 SPI 功能。例如 SPI0 的 MOSI 默认是 PIO0_29，但希望使用 PIO0_24 或 PIO1_4（详见 LPC55S69 用户手册），则可以使用这个方法重新配置。重新配置好引脚后，其它的函数操作方式不变。

3.14. UART 类和功能

lpc55.UART 类直接控制芯片的硬件 UART 模块，可以获得最大的性能。

3.14.1 UART 创建函数

class lpc55.UART(bus, ...)

返回对应一个 UART 硬件模块的实体。

- bus 是一个 0~7 的整数，表示 UART 硬件模块的编号。
- 如果没有其它参数，则不初始化硬件。有其它参数时则与 UART.init()参数相同，并继续执行初始化。

目前固件中按照逐飞开发板的硬件配置，而配置了各个模块的连接，如下表：

模块	TX	RX	开发板上接口脚位
UART0	PIO0_30	PIO0_29	P1_4, P1_6
UART1	PIO1_11	PIO1_10	P2_4, P2_6
UART2	PIO1_25	PIO1_24	P3_4, P3_6
UART3	PIO0_2	PIO0_3	P4_4, P4_6
UART4	PIO1_20	PIO1_21	P5_4, P5_6
UART5	没有固定配置，需自行配置。		
UART6	PIO0_22	PIO1_13	P6_4, P6_6
UART7	PIO1_30	PIO1_29	P7_4, P7_2

特别注意：在 LPC5500 中，芯片内部 I2C、SPI、UART 和 I2S 都是共用相同的硬件，当某个功能占据了某个硬件模块的编号，其它功能则不能再用这个编号。例如使用了 I2C1 之后，就不能再使用 SPI1 了，也不能使用 UART1 了，否则创建函数会抛出异常。

3.14.2 UART.init()和 UART.deinit()

UART.init(baudrate=9600, bits=8, parity=None, stop=1, *, flow=0, timeout=0, timeout_char=0, read_buf=-1)

UART.deinit()

初始化和关闭 UART 模块。

- baudrate 为波特率。
- bits 为每个字节的位数，取值 7 或 8。
- parity 表示奇偶校验。None 表示无奇偶校验，任一奇数整数表示奇校验，任一偶数整数表示偶校验。
- stop 表示停止位。整数 1 表示一个停止位，其它任意整数表示 2 个停止位。
- flow 表示硬件流控方式。目前暂未实现。
- timeout 表示在接收时，没有收到数据时，需要等多长时间返回。单位为 ms。
- timeout_char 表示在接收每个字节时，没有收到数据时，需要等多长时间。单位为 ms。
- read_buf 表示设置指定长度的接收缓存。
  - 默认是 64 字节的缓存。
  - 这是一个整数。0 表示没有缓存。

3.14.3 UART.any()检测是否有数据到达

UART.any()

返回一个布尔量，表示内部接收缓存是否有数据。

3.14.4 UART.read()读出数据

UART.read([nbytes])

读出数据。

- nbytes 表示要读出多少字节。如果数据未到达，则按照 init()的 timeout 和 timeout_char 给定的时间等待。
  - 如果 nbytes=-1 或不带参数，则读出所有内部缓存中的数据。如果没有内部缓存，则读出尽可能多的数据，直至超时。
  - 如果 nbytes 为任意正整数，则读出指定数量的数据。
  - 数据返回在一个 bytearray 缓冲区。

3.14.5 UART.readline()读出一行字符

UART.readline(max_size)

读出一行数据直至回车字符。

- max_size 限制读出长度。如果 max_size=-1 或缺省，则没有长度限制。

3.14.6 UART.readinto()读出数据至缓冲区

UART.readinto(buffer, max_size)

读至指定的缓冲区。

- buffer 为要接收数据的缓冲区，一般为 bytearray。
- max_size 指定最大数据长度。如果有参数 max_size 而且小于 buffer 的长度，则只读出数据数目为 max_size；否则按 buffer 长度读出数据。

3.14.7 UART.readchar()读出一个字符

UART.readchar()

以整数返回读出的字符，如果超时则返回-1。

3.14.8 UART.write()写数据

UART.write(buf)

写数据至外部设备。

- buf 为数据缓冲区，包含要写的数据。

该函数返回成功写的数据长度。返回-1 表示超时。

3.14.9 UART.writechar()写一个字符

UART.writechar(chr)

写一个字符至外部设备。

- chr 是要写的字符，为一个整数。

3.14.10 UART.sendbreak()发出断开字符

UART.sendbreak()

发送断开(break)字符。

3.14.11 UART.irq()设置中断回调函数

UART.irq(handler, trigger, hard)

按照给出的触发条件，设置中断回调。

- handler 为回调函数。
- trigger 为触发条件。目前只支持 UART.RX_ANY，即当收到任何字符时，产生中断。
- hard 是一个布尔值，表示是否为硬中断(True)，或软中断(False)。关于硬软中断的说明，见 3.9.4 节。

3.15. USB_CDC 虚拟串口类和函数

在固件里配置了两个虚拟串口，即 USB-CDC 设备，用于与 PC 之间的通讯。

LPC55S69 的 micropython 中 USB_CDC 类和函数，与标准文档一致，在 micropython.org 的文档里的名字是 USB_VCP（USB virtual comm port）。

3.15.1 USB_CDC 创建函数

lpc55.USB_CDC(id)

创建 USB_CDC 设备的实体。

- id=0 表示第一个 USB_CDC 设备。这是固件中 REPL 使用的端口。
- id=1 表示第二个 USB_CDC 设备，用户可以用它实现与 PC 端应用程序的数据交换。

3.15.2 USB_CDC.init()初始化

USB_CDC.init(*, flow=- 1)

配置 USB CDC 端口。

- flow 用于配置 USB CDC 的 RTS/CTS 功能，RTS 用于控制读操作，CTS 用于控制写操作。

注：此功能暂未实现。

3.15.3 USB_CDC.setinterrupt()设置中断字符

USB_CDC.setinterrupt(chr)

设置用于中断 Python 代码运行的字符，这个默认的中断字符是 Ctrl-C(整数 3)

- chr 为中断代码运行的字符。
- 当设置中断字符为-1 时，将关闭代码中断功能，这可以方便通过 CDC 端口传输原字节(raw data)。

3.15.4 USB_CDC.isconnected()检测连接状态

USB_CDC.isconnected()

如果 CDC 设备已经连接，则返回 True。

3.15.5 USB_CDC.any()检测是否接收到数据

USB_CDC.any()

返回一个布尔量，表示内部接收缓存是否有数据。

3.15.6 USB_CDC.close()

这个函数本身不做任何事情，只为把 USB_CDC 实体作为文件操作而存在。

3.15.7 USB_CDC.read()读数据

USB_CDC.read([nbytes])

读出最多 nbytes 字节的数据并通过一个 bytes 实体返回。

如果函数不带参数，则读出所有内部缓存的数据。

如果没有数据可读，函数直接返回 None，而不会阻塞代码运行。

3.15.8 USB_CDC.readinto()读出数据

USB_CDC.readinto(buffer, max_size)

读至指定的缓冲区。

- buffer 为要接收数据的缓冲区，一般为 bytearray。
- max_size 指定最大数据长度。如果有参数 max_size 而且小于 buffer 的长度，则只读出数据数目为 max_size；否则按 buffer 长度读出数据。

返回读出字符的数目，如果没有数据可读则返回 None。

3.15.9 USB_CDC.readline()读一行数据

USB_CDC.readline(max_size)

读出一行数据直至回车字符。

- max_size 限制读出长度。如果 max_size=-1 或缺省，则没有长度限制。

返回一个 bytes 实体，包含读出的数据，也包含最后的回车字符。如果没有数据则返回 None。

3.15.10 USB_CDC.readlines()读出多行数据

USB_CDC.readlines()

读出尽可能多的数据，并按行返回在一个 bytes 实体的 list 中。

3.15.11 USB_CDC.write()写缓冲区数据

USB_CDC.write(buf)

将 buf 中的数据发送到 PC 端。

返回送出的字符数目。

3.15.12 USB_CDC.recv()接收数据

USB_CDC.recv(*data*, *, *timeout=5000*)

从 USB 接收数据。

- data 可以是一个整数，表示需要接收的字符数目。收到的数据将通过一个 bytes 缓冲区返回。
- data 也可以是一个缓冲区，用于接收数据。返回值是一个整数，表示接收到的字符数目。

- timeout 是等待接收超时的数值，单位是 ms。

3.15.13 USB_CDC.send()发送数据

USB_CDC.send(*data, *, timeout=5000*)

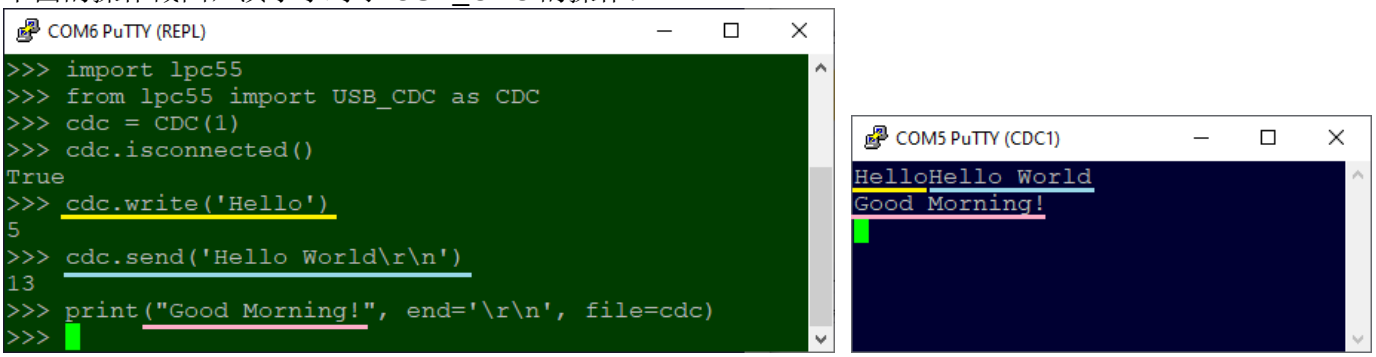
向 USB 发送数据。

- data 可以是一个整数，表示要发送的单个字符。
- data 也可以是一个缓冲区，包含发送的数据。返回值是一个整数，表示送出的字符数目。
- timeout 是等待发送超时的数值，单位是 ms。

3.15.14 USB_CDC 作为文件操作

USB_CDC 设备可以当作文件一样操作，例如可以用 print()向 PC 打印信息，用户也可以通过这个通道与 PC 端交换数据，尤其是把收集到的数据传到上位机处理显示等。

下面的操作截图，演示了对于 USB_CDC 的操作：



左图是 REPL 的窗口，右图是 CDC1 的窗口。图中用颜色下划线标出了执行的脚本与输出的对应关系。

3.16. 其它功能类

可能的功能类有：I2S、功耗管理、看门狗等。

目前这些还没有实现。

读者有什么常用的其它功能，希望添加到 micropython 内部，可以反馈给我。

第4章 SCT 及其用法

SCT 的全称是状态可编程定时器(State Configurable Timer)。SCT 是 LPC 系列 MCU 中一个功能十分强大的定时器模块，在很多 LPC 的 MCU 中都有它的身影，不同 MCU 中的 SCT 用法都一样，所不同的只是通道数目的差别。

SCT 的主要功能是维持一个内部的状态机，状态机的状态可以依据外部输入信号的变化，和内部计数器的变化而改变，用户可以指定在预设的状态下做出各种动作，例如改变 IO 端口的输出、产生中断、产生 DMA 请求或控制计数器的启停运行。

使用状态机的一大好处是，能够长时间地安排预定好的动作序列，在减少软件干预的情况下，尽可能自动地完成对输出信号的编程。

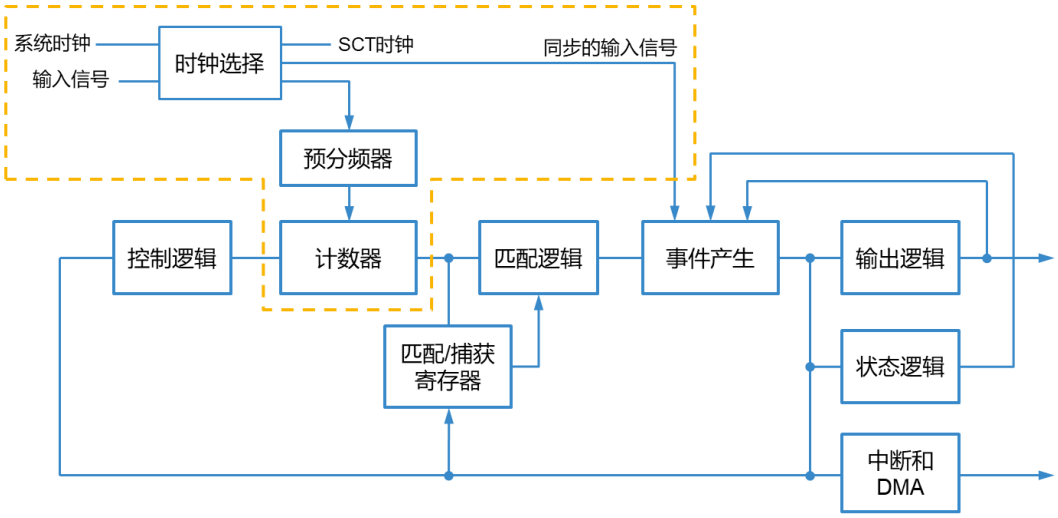
SCT 定时器又叫 PWM 定时器，顾名思义，它的主要功能是产生各种 PWM 信号。

SCT 类及其所属类都是集成在 lpc55 模块中的，这部分内容比较多也相对独立，所以单独作为一章来介绍。

本章假定读者已经熟悉 SCT 模块和它的用法，这部分内容不在此赘述。

4.1. SCT 模块的基本内容

这是 SCT 的基本框图：



基本上，框图中的橙色虚线部分，由 SCT 类进行操作控制。

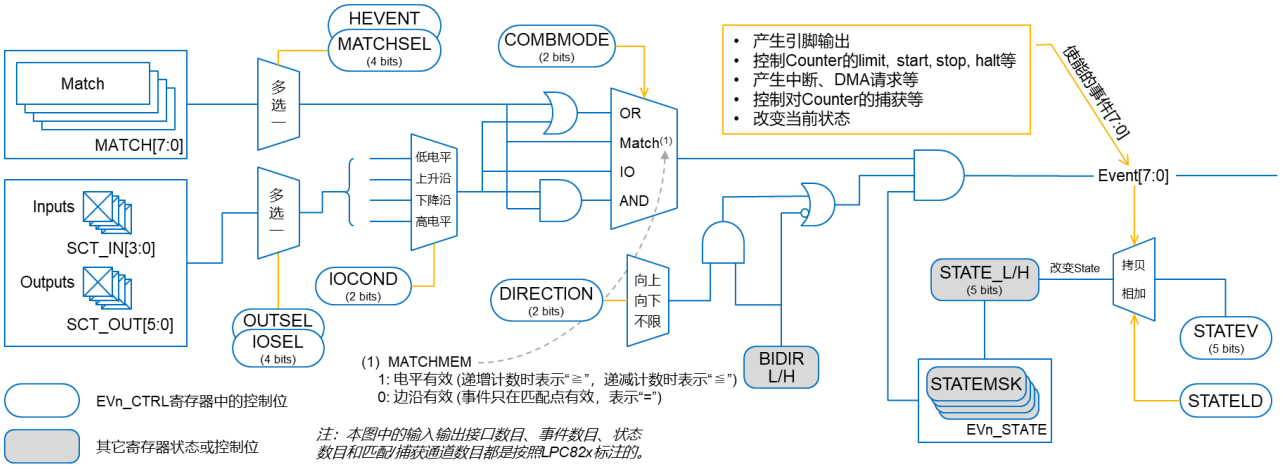
SCT 类是基础类，封装了对 SCT 的整体操作。下表简要总结了 SCT 类的内容：

函数	简介
lpc55.SCT()	创建函数
SCT.Run() SCT.Run2() SCT.Pause() SCT.Pause2() SCT.Halt() SCT.Halt2()	控制 SCT 计数器
SCT.Callback()	设置回调函数
SCT.Event() SCT.IO() SCT.Reg()	工厂函数，获取 SctIO、SctEvent 和 SctReg 类实体
SCT.Validate()	检测冲突(暂未实现)

属性	简介
Counter	计数器
Prescaler	预分频系数
State	状态机状态
Evflag	事件标志

常数	简介
MODE_DUAL	双区模式
MODE_UNIFY	复合模式
REG_MATCH	匹配寄存器
REG_CAPTURE	捕获寄存器
PIN_INPUT	输入引脚
PIN_OUTPUT	输出引脚

上面框图橙色虚线之外的部分，可以用下面的框图示意，按照 LPC55S69 的配置，共有 16 路相同的通道，图中只是一个通道的示意。



这个框图里的内容，由 SctEvent 类配置，这个类的实体由工厂函数 SCT.Event()获得。

SctEvent 类用于描述驱动状态机的事件，以及依赖于该事件的动作(操作)。该类封装了以下的函数和常数：

函数	简介
SctEvent.Action() SctEvent.AddAction() SctEvent.DelAction()	配置事件的动作
SctEvent.EnableState() SctEvent.NextState() SctEvent.EnableDir()	设置使能事件的状态和方向
SctEvent.Deinit()	释放资源
SctEvent.Callback()	设置回调函数

常数	简介
SctEvent.COMB_OR SctEvent.COMB_AND	描述事件的组合方式
SctEvent.COUNTING_UP SctEvent.COUNTING_DOWN	描述使能事件的方向
SctEvent.OP_NONE SctEvent.OP_LIMIT SctEvent.OP_START SctEvent.OP_STOP SctEvent.OP_HALT SctEvent.OP_INT	描述事件的动作

对每个通道中 IO 信号的控制，封装在 SctIO 类中，这个类的实体由工厂函数 SCT.IO()获得。

SctIO 类用于描述事件的 IO 输入源，以及这些输入源的基本属性。该类封装了以下的函数、属性和常数：

函数	简介
SctIO.Value()	获取引脚值

常数	简介
SctIO.CONFLICT_IGNORE SctIO.CONFLICT_SET SctIO.CONFLICT_CLR SctIO.CONFLICT_TOGGLE	描述冲突解决的方式

属性	简介
SctIO.Pin	设置/读取引脚值
SctIO.Rising SctIO.Falling SctIO.High SctIO.Low	输出引脚的 Event 条件 或作为时钟边沿
SctIO.OnConflict	设置解决冲突的方式
SctIO.Set_Pin SctIO.Clr_Pin	设置对应事件下对引脚的操作
SctIO.By_Dir	设置双向操作模式

对每个通道中匹配/捕获寄存器描述，封装在 SctReg 类中，这个类的实体由工厂函数 SCT.SctReg()获得。该类封装了以下的函数、属性和常数：

函数	简介
SctReg.Deinit()	释放资源

属性	简介
SctReg.Value SctReg.Reload	设置/读取寄存器值
SctReg.RegID	查看寄存器编号
SctReg.Type	查看寄存器类型
SctReg.Capture	描述捕获动作
SctReg.Capctrl	读取 CAPCTRL 寄存器

以下将分别介绍这些类的用法。

4.2. SCT 类

SCT 类是基础类，封装了对 SCT 的整体操作。

4.2.1 SCT 创建函数

SCT 的创建函数有四种形式，分别对应四种不同的时钟模式。

一. 系统时钟模式 (System clock mode)

这个模式下，系统时钟驱动整个 SCT 模块，包括计数器和计数器的预分频器。

class lpc55.SCT(mode)

返回 SCT 实体，并配置 SCT 为系统时钟模式。

- mode=SCT.MODE_DUAL 是把 SCT 分为两个 16 位半区。函数返回一个 tuple，包含 2 个 SCT 实体，第一个是低半区的 SCT 实体，第二个是高半区的 SCT 实体。通常使用者不必区分是高半区还是低半区，两者的使用方法完全一样。
- mode=SCT.MODE_UNIFY 是把 SCT 作为一个 32 位的整体。函数只返回一个 SCT 实体。

二. 系统时钟采样模式(Sampled system clock mode)

这个模式下系统时钟驱动 SCT 模块，但计数器和它的预分频器则由输入引脚使能。

class lpc55.SCT(mode, sample=io.Rising 或 io.Falling)

返回 SCT 实体，并配置 SCT 为**系统时钟采样模式**。

- mode 的取值与系统时钟模式相同。
- sample 的指示使用哪个引脚的哪个边沿使能计数器的时钟。
  - sample=io.Rising，其中 io 是静态函数 SCT.IO()返回的一个 SctIO 实体，表示使用对应引脚的上升沿使能计数器的时钟。
  - sample=io.Falling，表示使用对应引脚的下降沿使能计数器的时钟；同上，io 是一个 SctIO 实体。

三. SCT 输入时钟模式 (SCT Input clock mode)

这个模式下，输入边沿采样、计数器和计数器预分频器由输入的时钟驱动，但输出信号与系统时钟同步。

class lpc55.SCT(mode, freq=xxxx)

返回 SCT 实体，并配置 SCT 为**输入时钟模式。**

- mode 的取值与系统时钟模式相同。
- freq 是一个整数时，表示使用 SCTCLKDIV 提供的时钟，freq 表示为计数器时钟的频率。
- freq 也可以指定某个输入引脚的边沿作为时钟：
  - freq=io.Rising，表示使用对应引脚的上升沿作为计数器时钟。
  - freq=io.Falling，表示使用对应引脚的下降沿作为计数器时钟。
  - 同上，这里的 io 是一个 SctIO 实体。

四. 异步模式 (Asynchronous mode)

这个模式下，整个 SCT 都是由外部输入的时钟驱动。

class lpc55.SCT(mode, clock=io.Rising 或 io.Falling)

返回 SCT 实体，并配置 SCT 为**异步时钟模式。**

- mode 的取值与系统时钟模式相同。

- clock 指定某个输入引脚的边沿作为时钟，意义同上。

4.2.2 SCT.Run()和 SCT.Run2()开始运行

配置好所有部分后，即可启动 SCT 的运行。

SCT.Run(state=None)

SCT.Run2(stateL=None, stateH=None)

启动 SCT 运行的函数有两种形式，第一种形式用于 UNIFY 模式下的 SCT，第二种形式用于双区模式。

- state 指定从特定的状态开始执行。如果没有这个参数或为 None，则从当前状态开始执行。
- stateL 和 stateH 分别指定两个半区从哪个状态开始执行。如果某个参数缺失或为 None，则哪个半区从当前状态开始执行。

4.2.3 SCT.Pause()和 SCT.Pause2()暂停运行

SCT.Pause()

SCT.Pause2()

这两个函数分别用于在 UNIFY 或双区模式下暂停 SCT 运行，随后可以用前述的 Run()或 Run2()继续运行。

4.2.4 SCT.Halt()和 SCT.Halt2()终止运行

SCT.Halt()

SCT.Halt2()

这两个函数分别用于在 UNIFY 或双区模式下终止 SCT 运行。

4.2.5 SCT.Callback()设置回调函数（已剔除）

SCT.callback(func)

设置事件的回调函数。

- func 为回调函数。该回调函数带一个整数参数，表示中断线路编号，用户可以忽略这个参数。

注：这个函数可能已经删除了。

4.2.6 SCT 实体的属性字段

SCT 实体拥有一些属性字段，用于读取或设置某些参数。

以下属性字段都是整数类型，可以用于 UNIFY 模式或双区模式的任一一个 SCT 实体。

- Prescaler 预分频系数

这是 SCT 时钟的预分频系数， SCT 使用经这个系数分频的时钟计数，默认分频系数为 1；预分频系数的最大值是 256，即最大分频 1/256。

该字段可以取值或赋值。

例如：factor = sct.Prescaler + 1，表示取预分频系数加 1 后赋值给 factor。

例如：sct.Prescaler = 100，表示设置预分频系数为 100。

- State 状态机状态

该字段用于改变状态机的状态，或查看状态机的状态，可以取值或赋值。

- Counter 计数器数值

读取该字段可以得到计数器的数值。

设置 Counter=0 表示计数器向上计数。

设置 Counter=1 表示计数器为向上再向下计数。

- Evflag 事件中断标志位

可以在中断回调函数中，读出的数值按位图，标示哪个事件发生中断。

4.3. SctIO 类

SctIO 类封装了对输入输出引脚的描述与控制，SCT 操作中涉及到外部引脚的部分，都由 SctIO 中的常数和属性字段进行描述。

在 LPC55S69 中，SCT 模块有 7 个输入引脚和 10 个输出引脚，每个输入输出引脚都可以被映射到多个外部引脚上，它们的对应映射关系如下。

SCT 对应的输出引脚：

外部引脚	SCT 输出引脚	开发板上的接口
PIO0_2	SCT_OUT0	P4_6
PIO0_3	SCT_OUT1	P4_4
PIO0_10	SCT_OUT2	P6_8
PIO0_15	SCT_OUT2	P6_7
PIO0_17	SCT_OUT0	SD0_CARD_DET_N
PIO0_18	SCT_OUT1	P5_7
PIO0_19	SCT_OUT2	P5_5
PIO0_22	SCT_OUT3	P6_6
PIO0_23	SCT_OUT4	P1_7
PIO0_26	SCT_OUT5	P12_7
PIO0_27	SCT_OUT6	LED_GREEN
PIO0_28	SCT_OUT7	P1_8
PIO0_29	SCT_OUT8	P1_4
PIO0_30	SCT_OUT9	P1_6
PIO0_31	SCT_OUT3	P14_6
PIO1_3	SCT_OUT4	P12_6
PIO1_4	SCT_OUT0	P12_9
PIO1_8	SCT_OUT1	P14_3
PIO1_9	SCT_OUT2	P14_4
PIO1_10	SCT_OUT3	P2_4
PIO1_17	SCT_OUT4	P6_5
PIO1_18	SCT_OUT5	KEY_WAKEUP
PIO1_19	SCT_OUT7	P12_10
PIO1_23	SCT_OUT0	P3_8
PIO1_24	SCT_OUT1	P3_4
PIO1_25	SCT_OUT2	P3_6
PIO1_26	SCT_OUT3	P3_7
PIO1_31	SCT_OUT6	P15_6

SCT 对应的输入引脚：

外部引脚	SCT 输入引脚	开发板上的接口
PIO0_0	SCT_GPI0	P15_1
PIO0_1	SCT_GPI1	P15_2
PIO0_2	SCT_GPI2	P4_6
PIO0_3	SCT_GPI3	P4_4
PIO0_4	SCT_GPI4	P5_8
PIO0_5	SCT_GPI5	ISP_BOOT 按键
PIO0_6	SCT_GPI6	P4_8
PIO0_12	SCT_GPI7	SWDIO
PIO0_13	SCT_GPI0	P2_7
PIO0_14	SCT_GPI1	P2_5
PIO0_17	SCT_GPI7	SD0_CARD_DET_N
PIO0_20	SCT_GPI2	P4_7
PIO0_21	SCT_GPI3	P4_5
PIO0_24	SCT_GPI0	SD0_D0
PIO0_25	SCT_GPI1	SD0_D1
PIO1_0	SCT_GPI4	P14_5
PIO1_1	SCT_GPI5	P12_8
PIO1_2	SCT_GPI6	P12_5
PIO1_5	SCT_GPI0	SD0_D2
PIO1_6	SCT_GPI3	SD0_D3
PIO1_7	SCT_GPI4	P1_5
PIO1_19	SCT_GPI7	P12_10
PIO1_22	SCT_GPI5	LED_BLUE
PIO1_29	SCT_GPI6	P15_8
PIO1_30	SCT_GPI7	P15_7

上表中可以看出少部分信号已经被板上的其它资源占用，但大部分信号都引到了扩展接口上，读者可以自行根据需要连接自己的设备。

4.3.1 SctIO 的工厂函数 SCT.IO()

所有的输入输出引脚都隶属于 SCT 模块，因此封装 IO 操作的 SctIO 类没有单独的创建函数，而是通过一个工厂函数 SCT.IO()而创建 SctIO 实体，这个实体将天然地与 SCT 实体建立了内在的联系。

class SCT.IO(type, pin1, pin2, …)

class SCT.IO(type, pinlist)

获取 SctIO 实体的工厂函数有以上两种形式。

- type 指出引脚是输入还是输出，由常数 SCT.PIN_INPUT 或 SCT.PIN_OUTPUT 指定。
- pin1, pin2, …分别是引脚名称，例如'PIO0_1'、'PIO0_2'或'LED_Green'等(参见上节的表格)。
- pinlist 是一个 tuple 或 list，其中每个分量是一个引脚的名称。

一般情况都是一个 SctIO 实体对应一个引脚，即调用这个工厂函数时，只给出一个引脚名称。但有些应用希望能够把多个引脚组织为一个字段处理，则可以使用多个引脚调用这个函数，这样一个 SctIO 实体可以代表这个字段了。

注：由于没有合适的测试案例，还未验证多引脚 SctIO 实体的情形，如有读者需要这个功能，请给予反馈。

4.3.2 SctIO.Value()读取引脚状态

用于读取对应的引脚状态。

SctIO.Value([pin_num])

- pin_num 是一个整数，指示需要读出第几个引脚的值，引脚的编号与工厂函数 SCT.IO()中引脚名称的顺序一致。
- 如果不带参数，这个函数返回一个整数，这个整数的每个二进制位与引脚的编号对应，即 bit0 是 pin1 的状态，bit1 是 pin2 的状态，…。

4.3.3 SctIO 的属性字段

SctIO 有一些属性字段，用于描述外部引脚是在事件中的作用，下表列出用法简介，随后章节将详细介绍。

属性	简介
SctIO.Pin	设置/读取引脚值
SctIO.Rising SctIO.Falling SctIO.High SctIO.Low	输出引脚的 Event 条件 或作为时钟边沿 用于 SctEvent 实体中有关外部引脚的定义，或配置 SCT 时钟时使用。
SctIO.OnConflict	设置在遇到多个输出配置产生冲突时的解决方式
SctIO.Set_Pin SctIO.Clr_Pin	在 SctEvent 中设置对引脚的操作。
SctIO.By_Dir	如果计数器是双向计数时，配置对外部引脚的操作规则。

4.4. SctReg 类

SctReg 类用于控制每个通道对应的寄存器。

4.4.1 SctReg 的工厂函数 SCT.Reg()

SctReg 的实体是通过调用 SCT.Reg()创建。

class SCT.Reg(type[, match])

- type=REG_MATCH 表示返回的 SctReg 是匹配寄存器。
- type=REG_CAPTURE 表示返回的 SctReg 是捕获寄存器。
- match 是一个整数，只在 type=REG_MATCH 时可以有这个参数，表示参数这个寄存器为给定的数值。

随后对这个 SctReg 实体的操作，会依据创建时的类型有所不同。

4.4.2 SctReg 的属性字段

SctReg 有一些属性字段，用于获取或配置一些数值，如下表：

字段	读写权限	说明

SctReg.Value	匹配寄存器时可读可写	用于读出或写入 match 寄存器
	捕获寄存器时只读	用于读出 capture 寄存器
SctReg.Reload	只可用于匹配寄存器读写	用于读出或写入 reload 寄存器
SctReg.RegID	匹配寄存器和捕获寄存器均为只读	用于读取寄存器 ID 编号
SctReg.Type	匹配寄存器和捕获寄存器均为只读	查看寄存器类型，返回 SCT.REG_MATCH 或 SCT.REG_CAPTURE
SctReg.Capture	捕获寄存器读	专用于配置事件的动作时使用，见 SctEvent

用法示例：

```
import lpc55
from lpc55 import SCT

sct = SCT(SCT.MODE_UNIFY, 1000000)    # Got 1MHz clock
sct.Prescaler = 100 # Got 10kHz

reg = sct.Reg(SCT.REG_MATCH, 1234)  # Get a match register
print(reg.Value)                    # Read its value
reg.Reload=2345                     # Set reload register

crg = sct.Reg(SCT.REG_CAPTURE)      # Get a capture register
id = crg.RegID                      # Get its ID
```

4.4.3 SctReg.Deinit()函数

SctReg.Deinit()

这个函数用于释放资源，随后原来被占据的寄存器又可以被分配给后续的请求。

4.5. SctEvent 类

我们知道状态机是由若干状态构成，当满足某种条件时会发生状态变化，在 SCT 中对于能够触发状态变化的条件被称为“事件”，SctEvent 类封装描述/配置事件的函数和属性。

在 LPC55S69 中最多可以有 16 个事件，每个事件可以来源于以下任意一种条件：

1. 匹配寄存器与计数器内容的比较
2. 某个 IO 引脚的指定状态
3. 前 2 个条件的逻辑“或”
4. 前 2 个条件的逻辑“与”

在某个状态 S 下，如果发生了指定的事件(条件)，状态机将发生状态变化，状态 S 将被成为该事件的使能状态。在发生状态变化的同时，这个事件还可以产生一系列的其它操作，这些操作包括：

1. 让计数器复位并重新从 0 开始计数；或让计数器改变计数方向，由递增变为递减或递减变递增。本文将以“限制”代称这个操作
2. 启动、暂停或终止计数器的计数
3. 在输出引脚输出‘0’或输出‘1’
4. 将当前计数器内容拷贝至捕获寄存器，这个操作命名为“捕获”操作。
5. 产生一个中断或 DMA 请求

SctEvent 类的函数都是围绕配置上述操作而设计的。

4.5.1 SctEvent 的工厂函数 SCT.Evt()

所有与事件相关的描述和操作，都是通过 SctEvent 实体实现的，这个实体是通过调用 SCT. Evt()创建。

这个工厂函数有两种形式：

class SCT. Evt(source)

class SCT.Evt(source1, source2, comb)

参见前面对事件源的说明，第一种形式用于单一的事件源，第二种形式用于对两个事件源进行逻辑操作。

- source, source1, source2 是一个 SctIO 实体或 SctReg 实体。

如果是 SctReg 实体表示事件源是匹配寄存器，直接使用 SCT.Reg()返回的实体即可。

如果是 SctIO 实体表示事件源是外部引脚，需要使用 SctIO 实体的条件属性作为参数，条件属性定义如下：

- SctIO.Rising：表示使用引脚的上升沿作为条件
- SctIO.Falling：表示使用引脚的下降沿作为条件
- SctIO.High：表示使用引脚的高电平作为条件
- SctIO.Low：表示使用引脚的低电平作为条件

当使用这个工厂函数的第二种形式时，source1 和 source2 必须分别是 SctReg 实体和 SctIO 实体的条件属性，两者不能为同一类型的实体。

- comb 描述了 source1 和 source2 之间的逻辑关系，可以用以下两个常量之一：
  - SctEvent.COMB_OR：表示用 source1 和 source2 的逻辑或作为事件的条件。
  - SctEvent.COMB_AND：表示用 source1 和 source2 的逻辑与作为事件的条件。

这个工厂函数返回一个 SctEvent 的实体，事件的条件源由上述参数确定。

4.5.2 SctEvent.Action()设置事件驱动的动作

一个事件发生后，除了导致状态机的状态变化，还能够同时执行一些操作，以下函数定义所要执行的操作。

SctEvent.Action(p1, p2, …)

SctEvent.AddAction(p1, p2, …)

SctEvent.DelAction(p1, p2, …)

- p1，p2，…分别定义执行的各种操作，每一个参数定义一个操作。这些参数可以是以下任意一个：
  - 用于对计数器的各种操作：

SCT.SctEvent.OP_LIMIT	限制计数器
SCT.SctEvent.OP_START	启动计数器
SCT.SctEvent.OP_STOP	暂停计数器
SCT.SctEvent.OP_HALT	终止计数器
SCT.SctEvent.OP_NONE	不影响计数器

  - 用于对外部引脚的输出：

SCT.SctIO.Set_Pin	对应引脚置“1”
SCT.SctIO.Clr_Pin	对应引脚置“0”

  - 捕获当前计数器数值：

SCT.SctReg.Capture	捕获计数器数值

  - 产生中断：

SCT.SctReg.OP_INT	产生一个事件中断

以上各项操作配置可以作为参数，以任意顺序出现。

SctEvent 内部有一个隐藏的操作列表，Action()函数用于建立这个表，AddAction()和 DelAction()分别用于从列表中添加或删除操作。需要用户自己记住这个表的内容，以便进行自行增减。

4.5.3 SctEvent.Callback()设置事件中断的回调函数

在以上 Action()函数中，如果配置了产生事件中断(SCT.SctReg.OP_INT)，则需要指定回调函数。

SctEvent.Callback(func[, count])

- func 是回调函数。
- count 为可选项，这个整数表示需要 count 次中断后才调用回调函数，允许用户更灵活地安排程序流程，例如在输出 PWM 波形时，精确控制输出的脉冲个数。

调用 Callback 的 SctEvent 实体，将作为参数传递给回调函数，允许多个事件共用一个回调函数。

4.5.4 SctEvent.EnableState()配置使能本事件的状态

本函数配置这个事件在哪个状态下使能并驱动配置的操作。

SctEvent.EnableState(state, …)

- state 给出了这个事件的使能状态。
- state 可以有多个，每个需用逗号隔开。
- 如果有多个 state，还可以提供一个 tuple 和 list 设置。
- 不带参数调用这个函数时，返回一个 tuple 包含所有已经设置的值。
- LPC55S69 最多有 16 个状态，因此 state 参数的取值应该在整数 0~15 之间。对于其它同系列芯片，取值范围要依 SCT 状态数目确定。

4.5.5 SctEvent.NextState()配置转换至的下一个状态

本函数用于设置产生这个事件后，需要转换到的下一个状态

SctEvent.NextState(state)

- state 给出了这个事件的条件成立后，需要转换至的下一个状态。
- state 参数的取值应该在整数 0~15 之间。
- 如果 state 是整数 256+x(x=0~15)，则表示下一个状态是当前状态+x 的值。
- 如果 state 是负整数-1~-15，则表示下一个状态是当前状态+这个负数的值。
- 参数不符合上述要求的数值时，函数抛出 ValueError 异常。
- 不带参数调用这个函数时，返回已经设置的值。

4.5.6 SctEvent.EnableDir()配置使能本事件时计数器的方向

SCT 使能一个事件时，还需要依据计数器的计数方向，本函数配置这个选项。

SctEvent.EnableDir(dir)

- dir 表示使能事件时计数器的计数方向：
  - dir=SCT.SctEvent.COUNTING_UP 表示只在计数器递增计数时才使能事件
  - dir=SCT.SctEvent.COUNTING_DOWN 表示只在计数器递减计数时才使能事件
  - dir=0 表示计数器的计数方向，不影响事件的使能。

4.5.7 SctEvent.Deinit()释放资源

SctEvent.Deinit()

释放这个事件实体占用的资源。

强烈建议在不使用这个事件实体后，调用这个函数释放资源，尤其是在 REPL 交互操作下。

4.5.8 SctEvent 的属性字段

以下这些属性字段用于配置或获取一些参数：

字段	读写权限	说明
SctEvent.EvID	只读	本事件的编号。
SctEvent.MatchReg	只读	本事件对应的 SctReg 实体，见 4.5.1。
SctEvent.MatchIO	只读	本事件对应的 SctIO 实体，见 4.5.1。
SctEvent.comb	只读	本事件同时包含 SctReg 和 SctIO 时的逻辑关系，见 4.5.1。

4.6. SCT 应用例程

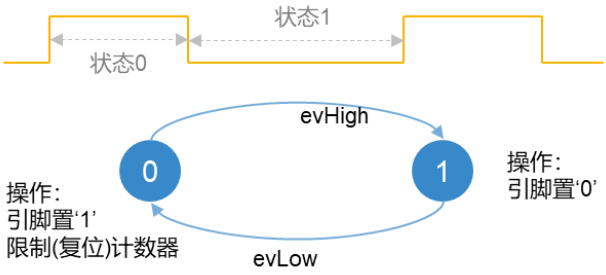
4.6.1 单路 PWM 输出

先看一个简单的例子，使用 SCT 输出一路 PWM 波形。

右图是一个 PWM 波形和产生这个波形的状态机。

这里把输出为“高”的阶段定义为状态 0，输出为“低”的阶段定义为状态 1。

我们定义一个事件，称为 evHigh，产生这个事件的条件是当高电平持续的时间达到要求的占空比时，此时执行将引脚置“0”的操作，并转换至状态 0。



接下来再定义一个 evLow 事件，这个事件的条件是当波形完成一个周期时，这时执行的操作是将引脚置“1”并进入下一个周期，同时转换至状态 1。

这里计数器的作用就是，使用合适的驱动时钟给出时间基准， 如右图当计数器的计数值达到定义的数值 Duty 时，表示要进行状态转换，当计数值达到数值 Freq 时要再次进行状态转换。

下面给出实现的代码。以下代码是产生一个周期为 1Hz 的 PWM 信号，占空比为 30%，这个信号输出到连接一个绿色 LED 的引脚上。

```
import lpc55
from lpc55 import SCT
Green = SCT.IO(SCT.PIN_OUTPUT, 'P12_6')

sct = SCT(SCT.MODE_UNIFY, 10000000)     # Got 10MHz clock
sct.Prescaler = 200                     # Got 50kHz
Hz_Tick = 50000                         # Number of clocks of 1Hz cycle

Duty = 30       # Preset duty
R_Duty = sct.Reg(SCT.REG_MATCH, Duty * Hz_Tick // 100)
R_Freq = sct.Reg(SCT.REG_MATCH, Hz_Tick)        # Got 1Hz

evHigh = sct.Evt(R_Duty)
evHigh.Action(Green.Clr_Pin)    # Clear the pin
evHigh.EnableState(0)
evHigh.NextState(1)             # Move to next state

evLow = sct.Evt(R_Freq)
evLow.Action(Green.Set_Pin, SCT.SctEvent.OP_LIMIT)
evLow.EnableState(1)
evLow.NextState(0)              # Back to state 0
sct.Run(0)
```

Green 是 SctIO 实体，对应外部 LED 信号。

R_Duty 和 R_Freq 是两个 SctReg 的实体，用于 evHigh 和 evLow 事件中。

最后的 sct.RUN(0)表示从状态 0 启动 SCT 运行。

启动运行后，SCT 将自动输出 PWM 波形，而不再需要任何代码干预，不占用 CPU 资源。如果希望改变占空比，可以通过 R_Duty 的 Reload 属性实现。

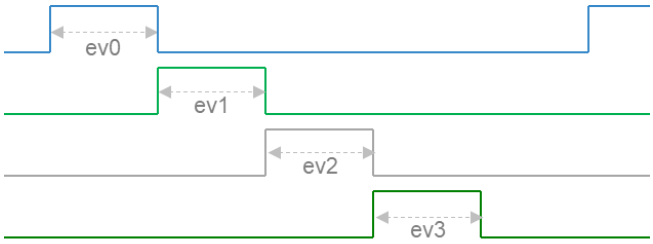
例如上述代码设置的占空比是 30%，设置 R_Duty.Reload=20 * Hz_Tick // 100 将改变占空比为 20%。

4.6.2 四路脉冲输出(走马灯)

这个例子输出四路宽度相同的脉冲，波形如右图，展示的效果就是常说的走马灯。

本例中只使用了一个匹配寄存器 R_Stage，并定义了四个状态 0~3，分别代表每路输出的高电平阶段，状态转换的事件也有四个 ev0~ev3。

匹配寄存器与计数器的比较成功，作为每个事件的使能条件，不同的事件将驱动不同的输出信号，同时每个事件都会限制(复位)计数器重新开始计数。




```
import lpc55
from lpc55 import SCT
Blue  = SCT.IO(SCT.PIN_OUTPUT, 'P12_6')
Green = SCT.IO(SCT.PIN_OUTPUT, 'P12_7')
White = SCT.IO(SCT.PIN_OUTPUT, 'P12_9')
Leaf  = SCT.IO(SCT.PIN_OUTPUT, 'P12_10')

sct = SCT(SCT.MODE_UNIFY, 10000000)     # Got 10MHz clock
sct.Prescaler = 200                     # Got 50kHz
Hz_Tick = 50000                         # Number of clocks of 1Hz cycle

R_Stage = sct.Reg(SCT.REG_MATCH, Hz_Tick)

ev0 = sct.Evt(R_Stage)
ev0.Action(Blue.Clr_Pin, Green.Set_Pin, SCT.SctEvent.OP_LIMIT)
ev0.EnableState(0)
ev0.NextState(0x101)        # The next state is current state +1

ev1 = sct.Evt(R_Stage)
ev1.Action(Green.Clr_Pin, White.Set_Pin, SCT.SctEvent.OP_LIMIT)
ev1.EnableState(1)
ev0.NextState(0x101)        # The next state is current state +1

ev2 = sct.Evt(R_Stage)
ev2.Action(White.Clr_Pin, Leaf.Set_Pin, SCT.SctEvent.OP_LIMIT)
ev2.EnableState(2)
ev0.NextState(0x101)        # The next state is current state +1

ev3 = sct.Evt(R_Stage)
ev3.Action(Leaf.Clr_Pin, Blue.Set_Pin, SCT.SctEvent.OP_LIMIT)
ev3.EnableState(3)
ev3.NextState(0)

Blue.Pin = 1    # Output high on first line at beginning
sct.Run(0)
```

和前面例子一样，可通过改变 R_Stage.Reload 改变脉冲的宽度，从走马灯的角度看就是改变走马灯速度。

4.6.3 四路 PWM 输出

这个例子与第一个例子输出相同波形，但分别输出频率相同的四路 PWM，每路输出的占空比可任意调整。

为了说明状态机的用法，第一个例子使用了 2 个状态实现 PWM 的任意占空比输出。本例给出一个不需状态机状态变化的方法，实现 4 路 PWM 的任意占空比输出。使用这个方法，可以很容易地扩充到所有 10 个输出引脚，得到最多 10 路 PWM 占空比可调的输出。

```
import lpc55
from lpc55 import SCT
# 分别定义四个外部输出引脚
Blue  = SCT.IO(SCT.PIN_OUTPUT, 'P12_6')
Green = SCT.IO(SCT.PIN_OUTPUT, 'P12_7')
White = SCT.IO(SCT.PIN_OUTPUT, 'P12_9')
Gre2n = SCT.IO(SCT.PIN_OUTPUT, 'P12_10')

sct = SCT(SCT.MODE_UNIFY)           # Fixed 150MHz
sct.Prescaler = 250                 # Got 600kHz
Length = 600                        # Tick to count 1ms

Duty0 = 10       # Preset duty
Duty1 = 20       # Preset duty
Duty2 = 40       # Preset duty
Duty3 = 60       # Preset duty

def initPWM(line, duty):
    Rev = sct.Reg(SCT.REG_MATCH, duty * Length // 100)
    ev = sct.Evt(Rev)
    ev.Action(line.Clr_Pin)
    ev.EnableState(0)
    ev.NextState(0)
    Events.append(ev)

Events=[]     # 用于记录调用创建的 SctEvent 实体
initPWM(Blue, Duty0)
initPWM(Green, Duty1)
initPWM(White, Duty2)
initPWM(Gre2n, Duty3)

R_EvReset = sct.Reg(SCT.REG_MATCH, Length)
evReset = sct.Evt(R_EvReset)
evReset.Action(SCT.SctEvent.OP_LIMIT, Blue.Set_Pin, Green.Set_Pin, White.Set_Pin, Gre
2n.Set_Pin)
evReset.EnableState(0)
evReset.NextState(0)

sct.Run(0)
```

上述代码中，对四路占空比控制的初始化部分都是一样的，由一个自定义函数 initPWM()实现。

在这个初始化函数中创建的 SctEvent 实体，集中保存在 Events 队列中，随后可以用来作其它操作，例如以下语句可以用来随时修改对应输出的占空比：

```
Events[0].MatchReg.Reload = 150   # 将 Blue 输出占空比设置为 25%:  (25 * Length // 100)
Events[2].MatchReg.Reload = 30    # 将 White 输出占空比设置为 5%:  (5 * Length // 100)
```

4.6.4 通过中断实现两个 LED 呼吸灯

所谓呼吸灯就是通过输出的 PWM 波，占空比从小变大再从大变小，不断循环往复的过程。本例程通过 SCT 的上下半区，分别输出两路不同频率的 PWM 波形，并通过回调函数的中断，改变占空比。以下是完整代码。

```
import lpc55
from lpc55 import SCT

Blue  = SCT.IO(SCT.PIN_OUTPUT, 'P12_6')
White = SCT.IO(SCT.PIN_OUTPUT, 'P12_9')
sctB, sctW = SCT(SCT.MODE_DUAL)     # Get 2 SCT objects at 150MHz

CTR_blk = {}        # A dictionary to retrieve control block

def duty_cb(ev):        # Callback function
    global CTR_blk
    duty = CTR_blk[ev.EvID]     # Get back the control block
    value = duty.update()
    ev.MatchReg.Reload = value  # Change the ticks

def initPWM(side, led, freq, duty):
    global CTR_blk
    side.Prescaler = 250    # Got 600kHz
    Freq_Ticks = 600000 // freq
    R_Freq = side.Reg(SCT.REG_MATCH, Freq_Ticks)
    R_Led = side.Reg(SCT.REG_MATCH, duty * Freq_Ticks // 100)

    evFreq = side.Evt(R_Freq)       # The event to control the frequency
    evFreq.Action(SCT.SctEvent.OP_LIMIT, led.Set_Pin)
    evFreq.EnableState(1)
    evFreq.NextState(0)

    evLed = side.Evt(R_Led)         # The event to control the duty
    evLed.Action(led.Clr_Pin, SCT.SctEvent.OP_INT)
    evLed.EnableState(0)
    evLed.NextState(1)

    CTR_blk[evLed.EvID] = Duty_Blk(duty, duty, Freq_Ticks)  # Keep the control block
    evLed.Callback(duty_cb, 50)     # Set up the callback

initPWM(sctB, Blue, 2000, 40) # Initialize Blue channel as 2000Hz and max-duty=40
initPWM(sctW, White, 500, 20) # Initialize White channel as 500Hz and max-duty=20

sctB.Run2(0, 0)
```

在这个例子里，使用系统时钟模式，并区分上下半区，因此在创建 SCT 实体时使用参数 MODE_DUAL 并使用变量 sctB 和 sctW 分别接收一个半区。

由于上下半区功能相同，只是代入的参数不一样，上述代码中定义了一个函数 initPWM()用于分别初始化两个半区。两个半区的 PWM 频率不一样，最大的占空比也不一样。

在 evLed 事件的执行动作中，比前面例子增加了 OP_INT 参数，允许产生中断，在回调函数中将更新 evLed 对应的匹配寄存器的比较值。回调函数的参数 ev 是对应 SctEvent 实体，可以用它的属性来访问相应的 SctIO 或 SctReg。

在例子中定义了一个新的类 Duty_Blk，负责管理占空比的递增和递减，这个类定义如下：

```
class Duty_Blk():       # A class to adjust duty cycles
    def __init__(self, duty, maxv, ticks):
        self.Duty = duty
        self.Max = maxv
        self.Ticks = ticks
        self.Up = True
    def update(self):
        if self.Up:
            self.Duty += 1
            if self.Duty >= self.Max:
                self.Duty = self.Max - 1
                self.Up = False
        else:
            self.Duty -= 1
            if self.Duty <= 0:
                self.Duty = 1
                self.Up = True
        return self.Duty * self.Ticks // 100 # The new duty cycle
```

这个例子展示了事件中断的用法，和双区的用法。参考前面一个例子，见 4.6.3，读者可以自行改为不使用状态机，用中断实现多路 PWM 输出的占空比调节。

4.6.5 无需 CPU 干预的自主 LED 呼吸灯

这个例子展示了使用冲突解决机制，实现对输出引脚的翻转。

这个例子巧妙地利用两个半区的时间差，不断翻转同一个引脚输出，实现呼吸灯效果。改变两个半区的时间差值，可以改变呼吸灯循环的频率。启动 SCT 之后，这个例子完全不用 CPU 干预，自动循环运行。

```
import lpc55
from lpc55 import SCT
BASE_VALUE = const(15000)
Green  = SCT.IO(SCT.PIN_OUTPUT, 'P12_7')

sctL, sctH = SCT(SCT.MODE_DUAL)    # System Clock mode. Got 150MHz clock
sctL.Prescaler = 100 # Got 1.5MHz
sctH.Prescaler = 100 # Got 1.5MHz

Green.OnConflict = Green.CONFLICT_TOGGLE        # Toggle output under confliction

base_reg = sctL.Reg(SCT.REG_MATCH, BASE_VALUE)  # Base frequency
base_ev = sctL.Evt(base_reg)
base_ev.Action(Green.Set_Pin, Green.Clr_Pin, base_ev.OP_LIMIT) # Toggle output
base_ev.EnableState(0)
base_ev.NextState(0)

led_reg = sctH.Reg(SCT.REG_MATCH, BASE_VALUE - 200) # Frequency is different!
led_ev = sctH.Evt(led_reg)
led_ev.Action(Green.Set_Pin, Green.Clr_Pin, led_ev.OP_LIMIT)   # Toggle output
led_ev.EnableState(0)
led_ev.NextState(0)

sctL.Run2(0, 0)
```

增加以下代码可以同时驱动两个外部引脚，此方法可以扩展到驱动多个引脚。

但请注意，此方法所有引脚的频率都是一样的；能够改变的是把某个引脚的相位翻转 180 度，这是通过启动 SCT 运行前，给输出引脚设置不同的初值实现的，将以下代码添加到启动 SCT 语句之前即可。

```
Gre2n = SCT.IO(SCT.PIN_OUTPUT, 'P12_10')    # Define another green LED
Gre2n.OnConflict = Gre2n.CONFLICT_TOGGLE
base_ev.AddAction(Gre2n.Set_Pin, Gre2n.Clr_Pin) # Add new actions
led_ev.AddAction(Gre2n.Set_Pin, Gre2n.Clr_Pin)  # Add new actions
Gre2n.Pin=0       # Initial state is off
Green.Pin=1       # Initial state is on, which is different than Gre2n
sctL.Run2(0, 0)
```

第5章 display 显示类及其用法

显示屏是最常见的一种扩展外设，display 是一个模块，包含操控不同类型屏的 Micropython 类，下表列出了目前实现了的三种屏：

Micropython 类名称	支持的显示屏
OLED096	SSD1306 驱动的 0.96 寸 OLED 屏，分辨率为 128x64
LCD154	ST7789 驱动的 1.54 寸 LCD 彩色屏，分辨率为 240x240
LCD350	ILI9486 驱动的 3.5 寸 LCD 彩色屏，分辨率为 480x320

注：由于时间匆忙，不是所有功能(组合)都进行过完整测试，使用中有什么问题，请及时反馈。

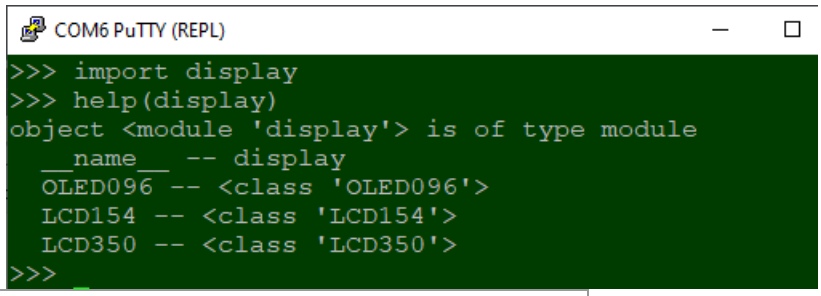
在 Micropython 类的实现中，通过模块化设计，有效地分隔了底层驱动与上层功能的代码，因此可以比较方便地添加其它类型或其它驱动芯片的显示屏，如有这方面的需求，请与我联系。

如无特别说明，本章主要以 LCD154 类作为讲解。

5.1. display 模块一览

display 模块中包含 3 个类，分别对应三种显示屏，见右图和以上列表。

当使用某个显示屏时，首先要用 display 模块里的基础类创建一个 Display 类的实体，Display 类包含以下一些函数：



Display 类的函数	功能简介
orientation()	配置屏幕的显示方向。
getframe()	获取操控屏幕的基础缓冲区。
framebuf()	创建一个帧缓冲区，用于直接在屏幕上显示或与其它帧缓冲区进行图层叠加显示。
scope()	创建一个特殊的帧缓冲区，用于以示波器的方式显示波形。
clear()	清除屏幕。
cmd()	发送命令，直接控制驱动芯片。

5.1.1 framebuf 类

以上的 getframe、framebuf 和 scope 都是工厂函数，均返回一个 framebuf 类的实体。

framebuf 类包含以下一些操作函数：

framebuf 类的函数	功能简介
clear	清除屏幕。
show	将缓冲区的内容显示到屏幕上。
showwith	将缓冲区的内容与另一个帧缓冲区的内容叠加后显示到屏幕上，实现图层的效果。
color	设置显示时所用的作图颜色和背景颜色。
move(), shift()	移动本缓冲区的位置。
point(), line(), circle()	在缓冲区中画点、线或圆。
string()	在缓冲区中显示字串
monoicon()	拷贝一个二值图标到缓冲区中，可用于显示汉字。
loadbmp()	从文件系统中加载一个 bmp 图像。

5.1.2 scope 类

scope 类是 framebuf 的衍生类，除了继承的所有 framebuf 的函数外，还有另外 2 个函数：

scope 类的函数	功能简介
channel()	返回一个 scope_channel 实体的工厂函数。
refresh()	拷贝所有内容到屏幕上。

5.1.3 scope_channel 类

每一个 scope_channel 类实体，将对应屏幕上的一个示波器显示波形，它有以下函数：

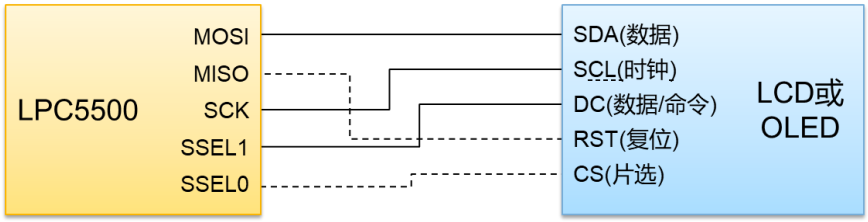
scope_channel 类的函数	功能简介
init()	初始化
clear()	清除缓冲区
color()	设置曲线颜色和背景色
scale()	设置比例系数
value(), wave()	输入要显示波形中各个点的数值

以下各小节将分别详细介绍这些类和函数的用法。

5.2. 显示屏的硬件

介绍 micropython 类之前，先说明一下显示屏信号线与 LPC55xx 的连接。

本章涉及到的所有显示屏，都要求按以下方式连接到 LPC55xx 的任一 SPI 接口上：



其中显示屏的 SDA、SCL 和 DC 必须按照图中的方式连接。尤其是 DC 信号线，需要连接到 SPI 的 SSEL1 上，这样可以发挥 SPI 接口的最大效率。

另外两个信号线，建议按照图中的方式连接，也可以连接到芯片的任意 IO 引脚上。

注：在软件内部，不对 CS 信号做任何操作，用户需自行控制这个信号，或在硬件上把它常拉为低电平。

5.2.1 显示屏的坐标系

每个显示屏都是以左上角作为坐标原点，即(0, 0)。

屏幕的水平方向用 x 坐标标示，始终是从左至右增长。

屏幕的垂直方式用 y 坐标标示，始终是从上至下增长。

因此以 LCD350 为例，屏幕的右下角坐标为(479, 319)。

5.3. Display 类的操控

要使用任意一款显示屏，都需要有一个 Display 实体与之对应，作为最基本的操控对象。

5.3.1 Display 类的创建函数

每一种显示屏都有一个 Display 的创建函数，调用方式也都一样：

class OLED096(spi_obj, pin_rst[, pin_dc])

class LCD154(spi_obj, pin_rst[, pin_dc])

class LCD350(spi_obj, pin_rst[, pin_dc])

三个创建函数的参数都一样，意义也都一样。

- spi_obj 是一个 lpc55.SPI 实体，对应显示屏所处的 SPI 接口，传入这个创建函数时，应该初始化 SPI 为 MASTER 模式，和其它各项参数，包括时钟频率、极性、相位、帧位数等。
- pin_rst 是一个 lpc55.Pin 实体，对应显示屏的复位(RST)引脚，应先初始化为输出引脚。
- pin_dc 也是一个 lpc55.Pin 实体，对应显示屏的 D/C 引脚，该引脚用于区分传输到显示屏的是命令还是数据，应先初始化为输出引脚。

注：见上一节硬件连接，要求 DC 引脚必须接到 SSEL1，因此 pin_dc 参数的功能目前没有实现，这是留待以后扩充功能时用。

下面是一个例子，先创建一个 Display 实体，再在屏幕上显示 Hello World!：

```
from display import LCD154

lcd_spi = lpc55.SPI(6, SPI.MASTER, bits=8, baudrate=25000000, polarity=1, phase=1)
lcd_rst=Pin('P6_6', Pin.OUT, pull=Pin.PULL_UP, value=1)
lcd_cs =Pin('P6_7', Pin.OUT, pull=Pin.PULL_UP, value=0)
lcd = LCD154(lcd_spi, lcd_rst)

lcd.clear(BROWN)        # 清除屏幕为棕色

fb0 = lcd.getframe()    # 获取基础缓冲区
fb0.string(10,100,"Hello World!", color=WHITE, font=24)
```

例子中的其它部分，随后陆续介绍。

5.3.2 Display.orientation()设定显示方向

Display.orientation(orient)

设置屏幕的显示方向。

- orient 是一个整数，表示显示方向。取值范围是 0~3，表示的方向见下图：

orient 取值：	0	1	2	3
示意图				

这里每一个方向的参数，从左至右，依次旋转 90 度。

设置好显示方向后，屏幕的长宽可能会互换，用户需在提供坐标和长宽参数时留意。

5.3.3 Display.framebuf()获取帧缓冲区

所有显示到屏幕上的内容，都必须通过帧缓冲区(framebuf 类)进行操作。

class Display.framebuf(x, y, width, height, /, bits, lut)

这是个工厂函数，返回一个 framebuf 实体，所有显示到屏幕的内容，都是通过这个实体进行操控的。

- x, y 是该帧缓冲区相对于屏幕左上角的坐标。
- width, height 是该屏幕的长度和宽度，这两个数值会随显示方向的改变而改变，见 5.3.2。
- bits 表示这个帧缓冲区使用的颜色位数。单色屏(OLED096)没有这个参数。

对于 LCD154 和 LCD350 这样的彩屏，bits 可用的数值为 1~4 或 16。

如果没有提供这个参数，则默认值是 16 位颜色。

- lut 是颜色对照表。对于彩色屏，当 bits 为 1~4 时，需提供这个颜色对照表。

lut 是一个内容为整数的 tuple 或 list，帧缓冲区内存里对应每个像素点存放的是这个数组的下标，因此这个数组至少需要有 2^{bits} 个整数，每个整数取值范围为 0~0xFFFF；例如 bits=3 时，需要提供一个具有 8 个整数的数组，分别包含 8 种颜色。

参数 bits 和 lut 的设置，是为了大幅节省内存空间。

较少的颜色位，非常适合显示字串、几何图形和波形曲线等，这些元素也是 MCU 显示的主要内容。真正用到 16 位彩色图像的地方还是少数，也不太适合 LPC5500 这样的 MCU。

特别注意：当颜色位数为 1~4 时，所有对这个 framebuf 操作中需要提供颜色参数时，需使用这个颜色对照表的下标，而不是真实的 16 位颜色值。

5.3.4 Display.getframe()获取基础帧缓冲区

每个 Display 实体，都必须首先有一个基础帧缓冲区实体，相当于一个在所有图层最底下的一层。

所有的显示屏操作，都必须调用工厂函数 getframe()获得基础帧缓冲区实体。

class Display.getframe()

该函数不带任何参数，默认覆盖整个显示区域，相当于是上述 framebuf()的特别版。

对于 OLED096，由于这个屏幕的特殊性，该函数返回的帧缓冲区，在固件内部真实地对应一个内存区域，用于存放需要显示的像素点，因此在这个帧缓冲区上的操作实际是在内存中进行，需要调用 show()才能显示到屏幕上。

对于 LCD154 和 LCD350，该函数返回的是一个虚拟的帧缓冲区，在固件内部并没有对应的内存区域，因此所有在这个帧缓冲区上的操作都会直接反映到屏幕上。

5.3.5 Display.scope()获取特殊帧缓冲区

为了显示曲线波形，制作一个小型示波器，特意设计了一个特殊的 scope 类，它是 framebuf 的衍生类，除了所有 framebuf 的函数外，还增加了专为显示曲线波形的函数，方便显示波形。

这个工厂函数用于获得 scope 类的实体。

class Display.scope(x, y, width, height, /, bits, lut)

该函数的参数与 Display.framebuf 完全相同，意义也完全一样，返回 scope 类实体。

注：每个 scope 类最多能处理 7 路波形，强烈建议选取参数 bits 使得 2^{bits} 大于要处理的波形数目。同时由于内存容量的限制，不能选择 16 位颜色，需要配置 bits=1~4。更多细节见 5.5 节关于 scope 类的介绍。

5.3.6 Display.clear()清除屏幕

这个函数会直接对屏幕操作而不经过帧缓冲区。

Display.clear(*, color)

清除屏幕上所有内容。

- 如果是彩色屏幕，color 表示把屏幕填充成指定的颜色，这个参数缺省时，表示填充黑色。
- 如果是 OLED096 单色屏幕，color 是一个填充模板，缺省是为 0x00，即填充黑色。

对于 OLED096，color 填充模板是一个字节，每个二进制位代表屏幕上一个像素点，'0'表示不亮，'1'表示亮。

每个字节对应纵向的 8 个像素点(字节中第 0 位对应编号小的行)，按顺序排列有如下对应关系：

字节顺序	意义
第 1 个字节	代表了屏幕上第 0~7 行的第 0 列的 8 个点
第 2 个字节	代表了屏幕上第 0~7 行的第 1 列的 8 个点
…	…
第 129 字节	代表了屏幕上第 8~15 行的第 0 列的 8 个点
…	…
第 1024 字节	代表了屏幕上第 56~63 行的第 127 列的 8 个点

假设 color=0x55，则调用 clear()后，屏幕的所有偶数行全亮，所有奇数行全不亮，显示间隔线条。

调用 clear(0xFF)则点亮所有像素，相当于屏幕全白。

5.3.7 Display.cmd()直接发送显示芯片命令

如果用户属性屏幕的驱动芯片，希望实现一些特殊功能或调整一些特点参数，或进行一些测试，可以使用这个函数，直接向显示驱动芯片发命令。例如可以用这个函数把显示屏设置为省电状态。

Display.cmd(cmd_data)

- cmd_data 是一个字符缓冲区，一般为 bytearray 类型。缓冲区的长度不限，但仅包含要输出的命令和所有附属参数。

注意，每次调用这个函数，只能有一个命令及其附属函数。

5.4. framebuf 类

要输出到屏幕上的各种图形、字体、波形等，都是通过调用 framebuf 的函数完成的。

用户可以在相同的区域，或部分重叠的区域配置帧缓冲区，然后按照一定的前后顺序把里面的内容映射到屏幕上，相当于引入了一组简单的图层操作，可以实现简单的动画效果。

下面逐个介绍 framebuf 的函数。

5.4.1 framebuf.color()设置画笔与背景颜色

改函数用于设置所有绘画操作(例如画点、线等)的画笔颜色，和设置背景的颜色。

framebuf.color(background, foreground)

- background 为背景颜色。
- foreground 为画笔颜色。
- 这两个参数的取值，对应该帧缓冲区的颜色值，有三种情形：
  - 对于 OLED096，取值是 0 或非 0。0 表示黑色，这是 background 默认值；非 0 表示白色，这是 forground 默认值。
  - 对于彩色屏且是 16 位颜色，可以取值任何 16 位整数范围内的数值。background 默认值是 0x0，一般为黑色。foreground 默认值是 0xFFFF，一般为白色。
  - 对于彩色屏且是 1~4 位颜色，取值范围依次为 0~1、0~3、0~7 和 0~15，参数是 lut 数组的下标。background 默认值是 0x0，foreground 默认值是可取值中的最大数值。

用户可以在任何时候更改画笔和背景，显示不同颜色的字体、画出不同颜色的线条和波形等。

5.4.2 framebuf.clear()清除帧缓冲区

framebuf.clear(*, color, refresh=False)

使用给定颜色填充帧缓冲区表示的整个区域。

- color 表示要填充的颜色(或颜色对照表下标)，如果缺省，则用 color()中设置的背景颜色，见 5.4.1。
- refresh 表示是否需要把这个帧缓冲区的内容刷新到屏幕上，默认是不刷新，即只修改内存里的内容，而不是改变屏幕的显示。

注：如果是虚拟帧缓冲区，refresh 参数无效，始终是刷新屏幕。

5.4.3 framebuf.show()显示帧缓冲区

对于虚拟帧缓冲区，这个函数不执行任何操作，直接返回，相当于 NOP。

framebuf.show(*, mode=0, sticky=False)

将帧缓冲区的内容显示到屏幕上。

- mode 指示如何将帧缓冲区里的内容显示到屏幕上。这个参数的取值和意义如下表：

mode 参数取值	OLED096 上的操作	彩屏(LCD154 和 LCD350)上的操作
0：NORMAL	将帧缓冲区里的内容，直接拷贝到屏幕上。	
1：TRANSPARENT	将帧缓冲区里的内容，非背景色像素点直接拷贝到屏幕上，背景色部分不拷贝。 背景色由 framebuf.color()设置，见 5.4.1。	
2：REVERSE	将帧缓冲区里的所有位取反，再拷贝到屏幕上。即帧缓冲区里黑色像素点，映射到屏幕时变为白色，反之亦然。	无意义，不识别。
3：REVERSE + TRANSPARENT	将帧缓冲区里的所有位取反后，再执行 mode=1 的操作。	无意义，不识别。

- sticky 表示是否把上述操作的结果保存在基础帧缓冲区，此参数只对 OLED096 有意义，所有彩屏则忽略此参数。此参数的默认值为不保存，即不改变基础帧缓冲区的内容。

当不带参数调用此函数时，使用缺省值 mode=0 和 sticky=False。

5.4.4 framebuf.showwith()与另一个帧缓冲区叠加后显示

经常会遇到需要实现图层的效果，例如动画效果就是在既有背景图案上叠加另一个图案，然后再把叠加的图案去掉恢复既有的背景图案，反复执行这样的操作而完成。另一个例子是在一个画面上浮现一个菜单，菜单操作完成后再隐去。

这个函数就是为了方便用户实现上述操作的。

framebuf.showwith(buddy, /, mode=0, sticky=False)

将帧缓冲区自身的内容与另一个帧缓冲区的内容叠加，再显示到屏幕上。

- buddy 是另一个帧缓冲区 framebuf 的实体。

特别注意，虚拟帧缓冲区不能作为 buddy 参数。

- mode 指示如何合并两个帧缓冲区的内容，再显示到屏幕上。这个参数的意义与 framebuf.show()的 mode 参数的意义非常接近，如下表：

mode 参数取值	OLED096 上的操作	彩屏(LCD154 和 LCD350)上的操作
0：NORMAL	将帧缓冲区里的内容，直接拷贝到屏幕上。	
1：TRANSPARENT	将帧缓冲区里的内容，非背景色像素点直接拷贝到屏幕上，背景色部分使用 buddy 在相同坐标下的像素点。 背景色由 framebuf.color()设置，见 5.4.1。	
2：REVERSE	将帧缓冲区里的所有位取反，再拷贝到屏幕上。即帧缓冲区里黑色像素点，映射到屏幕时变为白色，反之亦然。	无意义，不识别。
3：REVERSE + TRANSPARENT	将帧缓冲区里的所有位取反后，再执行 mode=1 的操作。	无意义，不识别。

- sticky 表示是否把上述操作的结果保存在 buddy 帧缓冲区。

对于彩屏来说，这个修改 buddy 的操作在不同颜色位数的情况下，有些局限性，见下表：

buddy 帧缓冲区的颜色位数 源帧缓冲区颜色位数	1	2	3	4	16
1	可以(1)	可以(1)	可以(1)	可以(1)	可以(1)
2	可以(1)	可以(1)	可以(1)	可以(1)	可以(1)
3	可以(1)	可以(1)	可以(1)	可以(1)	可以(1)
4	可以(1)	可以(1)	可以(1)	可以(1)	可以(1)
16	不可以	不可以	不可以	不可以	可以

(1) 如果两者颜色对照表在下标相同的部分内容不同，则合并后在 buddy 中与源帧缓冲区对应的像素颜色，会改变为 buddy 中的颜色，但在源帧缓冲区里不变。如果源帧缓冲区的颜色位数多于 buddy 的颜色位数，则多出的颜色信息将丢失。

注：上述所有操作只针对两个帧缓冲区中坐标相同的部分，非重叠部分不会显示也不会被改变。

实际上不难看出，前述的 framebuf.show()函数是这个函数的一个特例，基础帧缓冲区就是这里的 buddy。

5.4.5 framebuf.move()和 framebuf.shift()移动帧缓冲区位置

framebuf.move(x, y)

framebuf.shift(x, y)

这两个函数的功能是移动帧缓冲区的原点坐标。即把帧缓冲区表示图像进行平移。

- 第一种调用方式中，(x, y)参数是以屏幕原点为参照的绝对坐标，两个参数都必须是正整数。
- 第二种调用方式中，(x, y)参数给出的是偏移量，在原有的位置上下左右移动给定的偏移量，因此任何一个参数都可以是负数。

调用二者之一移动帧缓冲区后，执行 framebuf.shou()或 framebuf.showwith()之前，屏幕显示内容不变。

注：如果帧缓冲区的任何部分移出了物理屏幕范围，固件不做检查，使用者要确定不会产生这种情况。

5.4.6 framebuf.point()在帧缓冲区上画点

framebuf.point(x, y, /, color)

在帧缓冲区的内存中指定位置画一个点。

- x, y 是画点的坐标。这个坐标是相对于帧缓冲区的原点，不是相对于物理屏幕的原点。
- color 是点的颜色，这个数值是颜色对照表的下标而不是 16 位的颜色值。缺省值是 frame.color()设定的画笔颜色。

5.4.7 framebuf.line()在帧缓冲区上画直线

framebuf.line(x0, y0, x1, y1, /, color, size, style)

在帧缓冲区的内存中指定位置画一条直线。

- x0, y0 为直线的起点。
- x1, y1 为直线的终点。

以上两个坐标都是相对于帧缓冲区的原点，不是相对于物理屏幕的原点。

- color 是直线的颜色。这个数值是颜色对照表的下标而不是 16 位的颜色值。缺省值是 frame.color()设定的画笔颜色。
- size 是直线的宽度。
- style 表示是否画间断线(虚线)。0 表示画直线；非 0 表示画虚线，数值表示实线与间隔部分的长度。例如数值 4 表示画 4 个点，空余 4 个点的位置后再画 4 个点，直到直线的终点。

5.4.8 framebuf.circle()在帧缓冲区上画圆

framebuf.circle(x0, y0, radius, /, color, size, style)

在帧缓冲区的内存中指定位置画一个圆圈。

- x0, y0 为圆圈的圆心坐标。这个坐标是相对于帧缓冲区的原点。
- radius 是圆圈的直径。
- color 是圆圈的颜色。这个数值是颜色对照表的下标。缺省值是 frame.color()设定的画笔颜色。

- size 是画笔的宽度。
- style=0 表示画一个空心圆；任何非 0 整数都表示画一个实心圆。

5.4.9 framebuf.string()向帧缓冲区写字串

framebuf.string(x, y, str, /, font, color, mode)

在帧缓冲区的内存中输出一个字串。

- x, y 是整个字串所占矩形的左上角的坐标。这个坐标是相对于帧缓冲区的原点。
- str 是字串本身，长度不限，但超出帧缓冲区范围的部分不能显示，本函数不能实现自动换行。

字串中的每个字符必须是可显示的 ASCII 字符，即每个字符的十六进制值为 0x20~0x7E 之间，超出这个范围的字符将显示“▩”。

- font 表示字体的大小，固件内置了四种大小的字库，分别是(字库大小以’高’x’宽’表示)：
  - font=12 字库大小为 12x6。
  - font=16 字库大小为 16x8。
  - font=24 字库大小为 24x12。
  - font=32 字库大小为 32x16。
- color 是字体的颜色。这个数值是颜色对照表的下标。缺省值是 frame.color()设定的画笔颜色。
- mode 表示如何将字体存入帧缓冲区。它的取值和意义与前述 show()的 mode 参数基本相同，如下表：

mode 参数取值	OLED096 上的操作	彩屏(LCD154 和 LCD350)上的操作
0：NORMAL	将字库中 bit-1 转换为指定颜色，bit-0 转换为背景色，拷贝到帧缓冲区指定位置。 背景色由 framebuf.color()设置，见 5.4.1。	
1：TRANSPARENT	将字库中 bit-1 转换为指定颜色拷贝到帧缓冲区里，bit-0 部分不拷贝。	
2：REVERSE	将字库的所有位取反，再拷贝到帧缓冲区。	无意义，不识别。
3：REVERSE + TRANSPARENT	将帧缓冲区里的所有位取反后，再执行 mode=1 的操作。	无意义，不识别。

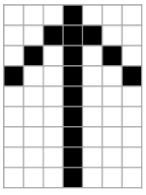
注：如果要显示的字串所占区域超出帧缓冲区的范围，则超出部分不会保存，将被丢弃。例如，当要输出的字串非常接近帧缓冲区的底边，则可能只能保留字符的上半部分。

5.4.10 framebuf.monoicon()向帧缓冲区贴图标

framebuf.monoicon(x, y, width, height, icon, /, mode, convert)

在帧缓冲区的内存中输出一个给定的点阵图标。

- x, y 是要放置图标的位置坐标。这个坐标是相对于帧缓冲区的原点。
- width, height 是要放置图标的宽度和高度，单位是像素点个数。
- icon 是存放图标的缓冲区。



该函数要求图标只能是单色的，即每个像素点只能是一个 bit 位。

基本的图标点阵按从左至右、从上至下的顺序排列，每个字节代表 8 个像素点，如果每行点数不是 8 的倍数，则最后凑足 8 位占据一个字节。因此一个图标的总字节数为：

```
height * (width + 7) / 8。
```

例如左图是一个箭头图标，大小为 7x9，icon 缓冲区中应有 9 个字节，内容如下：

```
0x08, 0x1C, 0x2A, 0x49, 0x80, 0x80, 0x80, 0x80, 0x80
```

Python 表示法是：`b’\x08\x1C\x2A\x49\x80\x80\x80\x80\x80’`

- mode 表示如何将图标存入帧缓冲区。它的取值和意义与前述 framebuf.string()相同。

- convert 给出了图标点阵的排列方式：
  - convert=0 表示按照上述默认的方式排列，即从左至右、从上至下地排列。
  - convert=1 表示按照 convert=0 的方式排列，但内部需要按 OLED 需要的方式转置成纵向排列，即每 8 行像素点为一排，一列 8 个点构成一个字节，每排字节数=width，共(height + 7)/8 排。例如上面这个箭头点阵经过转置后，内容与排列如下（共 14 字节）：

    ```
    0x08, 0x04, 0x02, 0xFF, 0x02, 0x04, 0x08,
    0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00
    ```

  - convert=2 表示按照 convert=0 的方式排列，但每个字节是从高位向低位排列，即每 8 个点中，最左边的点在字节的最高位，最右边的点在字节的最低位，因此上述箭头图标的例子要这样表示：

    ```
    0x10, 0x38, 0x54, 0x92, 0x10, 0x10, 0x10, 0x10, 0x10
    ```

  - convert=3 表示需要同时实现 convert=1 和 convert=2 的变换。

这个函数非常适合显示汉字，或二值的 logo、icon 等。

5.4.11 framebuf.loadbmp()显示图片

framebuf.loadbmp(file)

在屏幕上显示一副图片。

- file 是 open()函数返回的图片文件实体。

目前这个函数只支持 LCD154 屏幕，图片必须是 24-bit Bitmap 类型的文件，且只能是 240x240 点的图片。

以后会扩展这个函数，支持其它屏幕和其它尺寸的图片。

5.5. scope 类

scope 类是 framebuf 的衍生类，用于在屏幕上的某个区域或全部显示动态波形，类似示波器的显示效果。

作为是 framebuf 的衍生类，scopy 类支持上节介绍的所有 framebuf 功能，另外增加了以下一些函数。

scopy 类的实体由 Display.scope()工厂函数获得，见 5.3.5 节。

5.5.1 scope.channel()获取操作波形通道的实体

在 scope 类型下，要显示的波形按通道进行操控，目前一个 scope 实体可以最多支持 7 个通道。

scope.channel(id)

这是一个工厂函数，创建并返回一个 scope_channel 类的实体，随后用这个实体操作波形数据。

- id 是通道的编号，是一个 0~6 的整数。

用户需按编号从小到大的顺序创建 scope_channel 实体，不能跳跃，否则结果不确定。

5.5.2 scope.refresh()刷新所有通道至屏幕

scope.refresh()

处理所有通道的数据，并绘成曲线一起显示到屏幕上。

5.5.3 scope 类的用法

scope 是一个 framebuf，用户可以在上面绘制网格、图标、文字等，在执行 refresh()时，固件会把要显示的波形叠加在这些事先绘制好的图形之上并显示到屏幕上，此时并不改变内存中的已有图形，当波形移动变换时，用户也不会看得波形下面的图形内容有任何变化。

在 5.3.5 节创建 scope 类的实体时，建议颜色位数 bits 使得 2^{bits} 大于要处理的波形数目，这是因为能够用不同的颜色显示每个通道的波形曲线。如果要在显示区域绘制网格、文字等内容时，希望使用与波形不同的颜色，则创建 scope 实体时，要使得 2^{bits} 能允许更多的颜色。颜色下标 0 是背景色(底色)，不能用于任何波形和绘制的文字和图形。

例如需要显示 3 路波形，则选择 bits 的最小值是 2，如果希望用波形以外的颜色绘制图形，则需要选择 bits=3 或 bits=4。同理，如果要显示 4 路波形，至少要选择 bits=3，另有 3 种颜色可用于绘制图形。

5.6. scope_channel 类

scope_channel 类的实体用于接收波形数据并绘制波形曲线。

波形的一大特点是，每个时间点只有一个数值，表现为波形曲线上每个点的 y 坐标，每个点的 x 坐标就是采样这个数值的时间，这里我们假设采样的时间间隔是等长的，每两个点之间的 x 坐标相差数值 1。

在 LPC55xx 的 Micropython 中，用户提供的波形 y 坐标值，较大值会显示在屏幕上方，较小值会显示在屏幕下方，符合一般人的视觉

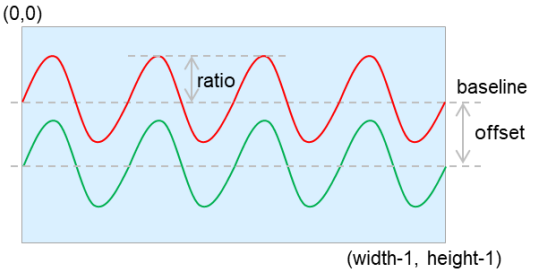
5.6.1 scope_channel.init()初始化

scope_channel.init(baseline, color, /, offset, ratio)

- baseline 用于设置波形显示时的垂直位置，称为基线。基线的位置就是当输入的 y 坐标为 0 时，在 scope 给定的区域中，垂直的位置。

例如要显示一个正弦波，波形的过零点需要落在这个参数设置的基线上。

- color 是这个通道的波形显示的颜色下标。
- offset 是一个整数，作为波形 y 坐标偏移量。
- ratio 是一个浮点数，作为波形 y 坐标的比例变换系数。



显示坐标 = y0 * ratio + offset；其中 y0 是用户提供的 y 坐标值，经过这个变换后，如果结果>0 则显示在 baseline 之上，如果结果<0 则显示在 baseline 之下。

右边的图中的红色正弦波形是在 offset=0 时的显示效果，绿色波形是 offset 为负数时的显示效果。图中的淡蓝色区域是创建 scope 实体时指定的区域，这个区域可以与物理屏幕的显示区域重叠，也可以是物理屏幕的一部分。

5.6.2 scope_channel.value()输入波形数据

scope_channel.value(data)

这个函数用于逐个输入要显示的数据，每次输入一个数据。

- data 是一个浮点数，表示要显示波形的 y 坐标。

这个输入的数据经过前述的变换之后，将存储在 scope 实体内部，当用户调用 scope.refresh()时，会显示到屏幕上。

每次调用这个函数输入一个数据时，它始终占据显示区域最右边的位置，之前输入的数据就会在内存中向左移动一格，当内存中的数据数量超出了显示区域的宽度，最左边的数据自动丢失。

如果每输入一个数据值，就调用一次 scope.refresh()，用户就会在屏幕上看到一个波形自右向左移动，移动的速度取决于输入数据的速度，当然显示的速度受芯片的计算速度和刷屏速度的限制。

5.6.3 scope_channel.wave()成组输入波形数据

scope_channel.wave(array)

这个函数用于一次性输入多个数据。

- array 是一个数组，可以是 tuple 或 list 等。

用户可以用这个函数显示一个静止波形，或一次输入多个数据，加快波形移动的速度。

5.6.4 scope_channel.clear()清除数据

scope_channel.clear()

这个函数用于清除已经输入至该通道的所有数据。相当于所有数据值变为 0。

5.6.5 scope_channel.color()设置波形颜色

scope_channel.color(color)

修改 init()设置的颜色。

- color 是这个通道的波形显示的颜色下标。

5.6.6 scope_channel.scale()设置波形的变换系数

scope_channel.scale(offset, ratio)

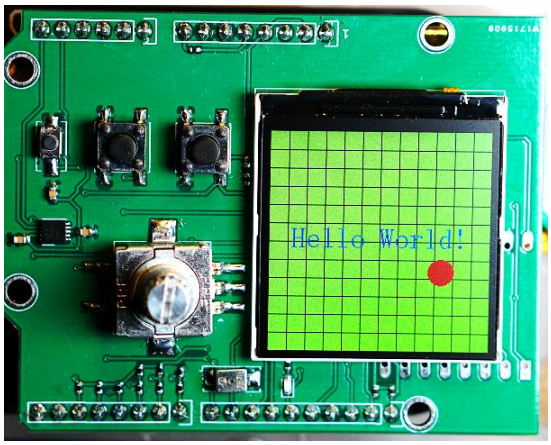
修改 init()设置的变换系数。

- offset, ratio 是这个通道波形的变换系数，意义与 init()函数中的描述一致。

5.7. framebuf 使用例程

5.7.1 碰壁的圆球

这个例程使用 LCD154 屏幕，有一个红色实心圆圈在屏幕上移动，一旦碰到边缘则按入射角等于反射角的方式反弹，并继续移动；在屏幕上铺满某个颜色，并画有方格线，屏幕中央显示“Hello World!”，如右图所示。



这个例程展示了，如何使用 framebuf 的功能，实现圆球的移动以及如何在圆球移动过程中不留痕迹，即背景图案保持不变。

实际运行就会看到，由于图形刷新速度足够快，人眼看不出有刷屏的痕迹，并且圆球移动的速度也很快。这是固件内部高效的算法所带来的优势，如果仅使用 Python 脚本直接操控 SPI 通道进行屏幕内容刷新，是达不到这样的效果的。

在演示代码的开始部分，是一些常数定义和模块的导入：

```
WHITE = const(0xFFFF)
BLACK = const(0x0000)
BLUE  = const(0x001F)
BRED  = const(0XF81F)
GRED  = const(0XFFE0)
RED   = const(0xF800)
MAGENTA=const(0xF81F)
GREEN = const(0x07E0)
CYAN  = const(0x7FFF)
YELLOW= const(0xFFE0)
BROWN = const(0XBC40)
BRRED = const(0XFC07)
GRAY  = const(0X8430)

BALL_SIZE = 32          # 圆球区域大小
SCREEN_WIDTH = 240      # 屏幕宽度
SCREEN_HEIGHT = 240     # 屏幕高度
X_BORDER = SCREEN_WIDTH - BALL_SIZE     # 圆球活动边界
Y_BORDER = SCREEN_HEIGHT - BALL_SIZE    # 圆球活动边界
```

```
import math
import machine
import lpc55
from lpc55 import SPI
from lpc55 import Pin
from display import LCD154
```

接下来是代码的主体，定义了一个 Panel 类：

```
class Panel():
    def __init__(self, id, rst, cs=None):
        lcd_spi = lpc55.SPI(id, SPI.MASTER, baudrate=50000000, polarity=1, phase=1)
        lcd_rst=Pin(rst, Pin.OUT, pull=Pin.PULL_UP, value=1)
        if cs:
            lcd_cs =Pin(cs, Pin.OUT, pull=Pin.PULL_UP, value=0)
        self.lcd = LCD154(lcd_spi, lcd_rst)    # 创建一个 Display 实体
        if not cs:
            self.lcd.orientation(2)
        self.fb0 = self.lcd.getframe()         # 获得基础 framebuf
        self.track = path(X_BORDER, Y_BORDER)  # 获得一个路径类的实体

    def setup(self, color_table):   # 设置要显示的内容和帧缓冲区
        self.desk = self.lcd.framebuf(0, 0, 240, 240, bits=2, lut=color_table)
        self.desk.clear(2)          # 用颜色 2 清除整个帧缓冲区
        self.desk.color(0,1)        # 设置画笔为颜色 1，背景为颜色 0
        for x in range(20,240,20):  # 用画笔画出方格的竖线
            self.desk.line(x, 0, x, SCREEN_HEIGHT)
        for y in range(20,240,20):  # 用画笔画出方格的横线
            self.desk.line(0, y, SCREEN_WIDTH, y)
        # 颜色 3 显示字串，选用字库大小为 32
        self.desk.string(24, 100, "Hello World!", color=3, font=32, mode=1)
        self.desk.show()            # 将上述内容显示到屏幕上

        # 创建另一个帧缓冲区，大小只够容纳圆球，只有 2 种颜色
        self.ball = self.lcd.framebuf(0, 0, 32, 32, bits=1, lut=(BLACK, RED))
        self.ball.clear()           # 清除这个帧缓冲区
        self.ball.circle(15,15,14,color=1,style=1)    # 用颜色 1(RED)画实心圆

    def update(self):               # 移动圆球，并显示
        pos = self.track.next()     # 计算圆球的下一个位置
        if pos:
            self.ball.move(pos[0], pos[1])        # 将包含圆球的帧缓冲区移到新的位置
            self.ball.showwith(self.desk, mode=1) # 透过画有图案的帧缓冲区，显示圆球
############################
# 创建一个 Panel 实体，使用 SPI8，复位引脚是 P12_9
lcd3 = Panel(8, 'P12_9')
lcd3.setup( (WHITE, BLACK, GREEN, BLUE) )         # 设置颜色对照表
```

定义 Panel 类的目的是，可以方便地用这个类处理多个 LCD 屏。

这些脚本代码都很直接，不用过多解释。其中的 path 类用于计算圆球移动的路径，将附在后面。

这里最重要的部分是画实心圆的帧缓冲区实体，它的位置是(0, 0)，但随后会重新计算，所以在哪里都行。这个帧缓冲区的大小定义为 32x32，但圆圈的直径比这个区域要小，而且圆心处于这个区域的中心，大约是

右图这个样子。这样设计的目的是，当移动这个帧缓冲区时，恰好可以把移动后留下的痕迹擦除，而不必另外用语句擦除。

接下来的 update()函数里，调用 path 类的函数获得新的位置，然后把含有圆球的帧缓冲区移到这个位置，再用 TRANSPARENT 方式与含有方格和文字的帧缓冲区叠加，显示到屏幕上。反复这个过程就实现了圆球的无痕移动，由于只有这个小的区域需要刷新屏幕，因此所消耗的时间很少，超过人眼所能感觉的程度，看起来非常流畅。

下面是封装圆球移动轨迹计算的 path 类代码，在初始化时利用 lpc55.rng()随机产生一个初始位置和初始的移动角度，随后就是每次调用 next()给出一个新的位置，这里不做详细解释，请读者自行理解。

```
class path():
    # Param: width & height of the moving area
    def __init__(self, width, height):
        def rand():     # mpy version
            self.x0 = lpc55.rng() % width       # starting point
            self.y0 = lpc55.rng() % height      # starting point
            self.a0 = 15 + lpc55.rng() % 60     # starting angle
            self.a0 = self.a0 + 90 * lpc55.rng() % 4
            self.a0 = self.a0 * (2 * math.pi) / 360     # Conver to angle
        def rand3():    # Version for Python3, for testing on PC
            self.x0 = random.randrange(width)   # starting point
            self.y0 = random.randrange(height)  # starting point
            self.a0 = 30 + random.randrange(30)   # starting angle
            self.a0 = self.a0 + 90 * random.randrange(4)
            self.a0 = self.a0 * (2 * math.pi) / 360
        self.width = width
        self.height = height
        rand()
        self.quadrant(self.a0)
        self.prev = [self.x0, self.y0]

    def quadrant(self, angle):
        self.tan = math.tan(self.a0)
        self.step = 1
        if angle < (math.pi / 2):
            self.delta = (1, 1)
            self.pathx = self.tan <= 1
        elif angle < math.pi:
            self.delta = (-1, 1)
            self.pathx = -self.tan <= 1
        elif angle < math.pi * 1.5:
            self.delta = (-1, -1)
            self.pathx = self.tan <= 1
        else:
            self.delta = (1, -1)
            self.pathx = -self.tan <= 1

    def next(self):
        if self.pathx:
            x1 = self.x0 + self.step * self.delta[0]
            y1 = self.y0 + self.step * self.delta[0] * self.tan
        else:
            x1 = self.x0 + self.step * self.delta[1] / self.tan
            y1 = self.y0 + self.step * self.delta[1]
```

```
        if y1 < 0 or y1 >= self.height:
            angle = math.pi * 2 - self.a0
        elif x1 < 0 or x1 >= self.width:
            angle = math.pi * 3 - self.a0
            if angle > math.pi*2:
                angle -= math.pi*2
        else:
            self.prev[0] = int(x1)
            self.prev[1] = int(y1)
            self.step += 1
            return self.prev
        self.a0 = angle
        self.x0 = self.prev[0]
        self.y0 = self.prev[1]
        self.quadrant(angle)
        return None    #self.next()
```

5.7.2 示波器显示

这个例程展示如何显示动态波形。

同样使用 LCD154 屏幕，在没有波形时屏幕的桌面是如右图这样。

代码的开头，仍然像上一个例子那样是一些常数定义和模块导入：

```
SCREEN_WIDTH = const(240)   # 屏幕宽度
SCREEN_HEIGHT = const(240)  # 屏幕高度
SCOPE_HEIGHT = const(200)       # 波形显示区域高度
SCOPE_WIDTH = SCREEN_WIDTH-10   # 波形显示区域宽度
SCOPE_GRID = 30                 # 波形显示区域内格子宽度
WAVE_NUM = const(5)         # 波形数目
WAVE_HEIGHT = SCOPE_HEIGHT//WAVE_NUM  # 每路波形高度

import math
import lpc55
from lpc55 import SPI
from lpc55 import Pin
from display import LCD154
```

右边图片显示的中间一大部分，就是波形的显示区域，它的宽度(SCOPE_HEIGHT)比物理屏幕略窄，波形区域的高度是由常数 SCOPE_HEIGHT 定义。屏幕上下各留了一个窄条，上面显示一些提示信息，下面则显示格子的刻度值。

在波形显示区域中间还有一行文字，用户可以在这个区域绘制任意图案或文字，甚至是放置桌面图片，随后的波形曲线，会浮在所有这些图案之上。

接下来是绘制和安排整个显示区域的脚本代码：

```
class Panel():
    def __init__(self, id, rst, cs=None):
        lcd_spi = lpc55.SPI(id, SPI.MASTER, baudrate=50000000, polarity=1, phase=1)
        lcd_rst=Pin(rst, Pin.OUT, pull=Pin.PULL_UP, value=1)
        if cs:
            lcd_cs =Pin(cs, Pin.OUT, pull=Pin.PULL_UP, value=0)
        self.lcd = LCD154(lcd_spi, lcd_rst)
        if not cs:
            self.lcd.orientation(2)          # 这个 LCD 是上下颠倒的，要设置正确显示方向
        self.fb0 = self.lcd.getframe()       # Get the basic framebuf
```

```
        self.fb0.clear(BROWN)
        self.fb0.string(18,0,"Oscilloscope DEMO",color=BLACK,font=24,mode=1) #Banner

    def setup(self, color_table):
        BANNER_HEIGHT = 24
        Scope_x = (SCREEN_WIDTH-SCOPE_WIDTH)//2
        self.ss = self.lcd.scope(Scope_x, BANNER_HEIGHT, SCOPE_WIDTH, SCOPE_HEIGHT,
                                 bits=3, lut=color_table)  # 创建波形显示帧缓冲区
        self.ss.clear(0)
        self.ss.color(5,2)

        y_low_banner = BANNER_HEIGHT + SCOPE_HEIGHT
        self.fb0.line(0, y_low_banner, SCREEN_WIDTH, y_low_banner,
                      size=16, color=0xE79C)            # 绘制下面的刻度条
        for x in range(SCOPE_GRID, SCOPE_WIDTH, SCOPE_GRID):
            self.ss.line(x, 0, x, SCOPE_HEIGHT, color=6)  # 画纵向刻度线条
            marker_x = Scope_x + x - len(str(x)) * 4
            self.fb0.string(marker_x, y_low_banner, str(x),
                        font=16, color=0x1C, mode=1)      # 显示刻度字体
        for y in range(SCOPE_GRID, SCOPE_HEIGHT, SCOPE_GRID):
            self.ss.line(0, y, SCOPE_WIDTH, y, color=6)   # 画横向刻度线条

        self.ss.string(48,10,"Hello World!", font=24)     # 字体居上
        self.ss.show()       # 显示整个帧缓冲区至屏幕

        # Create channels
        WAVE_COLOR = (RED, GREEN, BLUE, YELLOW, MAGENTA)
        self.channel = [0,] * WAVE_NUM
        for ch in range(WAVE_NUM):      # 创建 WAVE_NUM 个通道并初始化
            self.channel[ch] = self.ss.channel(ch + 1)
            self.channel[ch].init(WAVE_HEIGHT//2 + ch * WAVE_HEIGHT,
                      WAVE_COLOR[ch], offset=0, ratio=WAVE_HEIGHT/2-1) # 通道初始化

lcd1 = Panel(4, 'P5_6', 'P5_7')
# lcd2 = Panel(8, 'P12_9')
lcd1.setup((BLACK, WHITE, RED, GREEN, BLUE, YELLOW, 0x300, MAGENTA))
# lcd2.setup((BLACK, WHITE, RED, GREEN, BLUE, YELLOW, 0x300, MAGENTA))
```

在创建 Panel 类的实体的初始化部分(__init__())，首先通过基础(虚拟的)帧缓冲区，用棕色填充整个屏幕并显示顶端的文字“Oscilloscope DEMO”。

随后在 setup()函数中创建波形显示帧缓冲区用于随后容纳波形，然后就是画刻度条、刻度格子和刻度标尺，并在波形区域上部显示“Hello World!”。

在 setup()最后，创建若干个波形通道，用于接收波形数据。

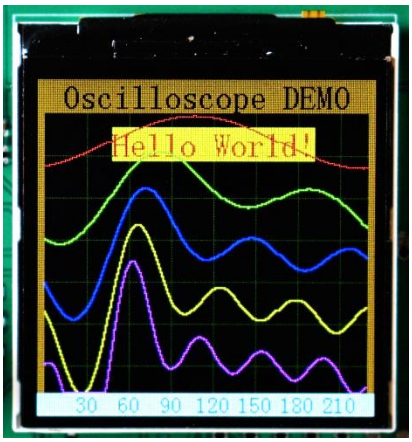
如果想同时使用两个 LCD 屏，可以在所有上述代码的 lcd1 位置，添加另一个屏幕 lcd2 的脚本，只要芯片内的内存容量够，就可以同时驱动两个屏。同样以下代码中，也要在适当位置添加传送数据至 lcd2 的语句。

下面的脚本代码，将循环产生一个正弦波形数据，和一组正弦波形叠加的数据，这些数据将分组地输入至各个通道，并逐步刷新到屏幕上。

```
Samples = 720
STEP = const(4)
wave = [0,]*STEP
```

```
while True:
    for x in range(0, Samples, STEP):
        delta = math.pi/120
        for i in range(x, x+STEP):           # 生成第一个通道的一组数据
            wave[i-x] = math.sin(i*delta)
        for ch in range(WAVE_NUM):
            lcd1.channel[ch].wave(wave)     # 输出一个通道的数据
            delta += math.pi/90
            for i in range(x, x+STEP):      # 计算下一个通道的数据
                wave[i-x] = wave[i-x] + math.sin(i*delta)
        lcd1.ss.refresh()                   # 刷新 scope 帧缓冲区中所有通道至屏幕
```

对于每个通道，上述代码的每个循环中，调用函数 scope_channel.wave()一次性传入 STEP 个数据。当所有通道的数据都已经传送就绪，最后调用 scope.refresh()把所有内存里的波形数据，刷新到屏幕上。



由于 scope.refresh()花费时间较长，如果每传送一个数据就刷新一次，则可以看到波形移动的速度较慢，当数据采样的速率比较慢时，可以采用每个数据都刷新的策略。

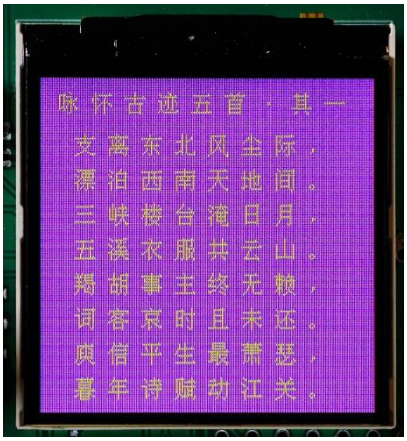
代码中常数 STEP 定义了刷新之前需要传送的数据个数，读者可以根据自己的数据采集速度，和希望的显示效果自行调整。

本例程使用了 scope_channel.wave()一次性传入多个数据，也可以多次调用 scope_channel.value()逐个传送数据。

右图是显示的效果。注意紫色波形向下超出显示区域的部分被削掉了。

5.7.3 提取汉字点阵字库并显示

首先先看，右图是这个例子的颜色效果。



大家都知道，汉字显示最关键的是要有点阵字库，一般字库都会占据不小的存储空间。例如一个 16x16 的点阵，每个字需要占据 32 个字节，国标一级字库有 3755 个汉字，二级字库有 3008 个汉字，这 6763 个汉字的字库就至少需要 216KB 的存储容量(实际有 267K 多)，所以一般 MCU 应用中要用到汉字显示的地方，都是只把用到的汉字字库单独提取出来，存放在 Flash 中，不能显示太多不同的字，同时也不方便随时更换不同的字。

在 Micropython 中，集成了文件系统，而且在我们的开发板上也配置了 SD(TF)卡槽，可以把完整的汉字点阵字库存放在 SD 卡上，想要什么字、什么字体、什么大小，直接从文件系统里调用读取即可，这大大方便了编写应用，也扩展了应用的适用范围。

先来看关于显示的部分，后面再看怎么提取字库。下面这段脚本是 LCD 的配置部分，基本和前例相同：

```
class Panel():
    def __init__(self, id, rst, cs=None):
        lcd_spi = lpc55.SPI(id, SPI.MASTER, baudrate=50000000, polarity=1, phase=1)
        lcd_rst=Pin(rst, Pin.OUT, pull=Pin.PULL_UP, value=1)
        if cs:
            lcd_cs =Pin(cs, Pin.OUT, pull=Pin.PULL_UP, value=0)
        self.lcd = LCD154(lcd_spi, lcd_rst)
        if not cs:
            self.lcd.orientation(2)
        self.fb0 = self.lcd.getframe()     # Get the basic framebuf
        self.desk = self.lcd.framebuf(0, 0, 240, 240,
                        bits=2, lut=(BLACK, MAGENTA, YELLOW, BLUE)) # 创建桌面帧缓冲区
```

```
        self.desk.clear(1)        # 用紫色(MAGENTA)充填桌面
        self.desk.color(0,2)      # 设置画笔颜色为黄色(YELLOW)
```

接下来是显示的主要部分：

```
def show_CN(fbuf, str, posx=0, posy=0):  # 显示一个汉字字符串
    w=0
    for utf8 in str:
        try:
            gb = hz.Utf8ToGB(utf8.encode()) # 将每个字符的 UTF8 编码转换为对应的 GB 编码
            ft = hz.Get_font(gb)            # 从文件系统中按照 GB 编码读取字库
        except ValueError:
            ft = b'\x55\x55\xaa\xaa' * 8    # 如果转换失败或找不到字库，则显示▒
        fbuf.monoicon(w*24+posx, posy, 16, 16,
                      ft, mode=1, convert=2) # 字库当成图标按透明方式显示
        w=w+1                               # 修改显示的坐标系数

hz=Hanzi()   # 创建一个处理汉字编码转换和提取字库的 Hanzi 类实体

lcd1 = Panel(8, 'P12_9')     # 创建操控显示的实体
show_CN(lcd1.desk, '咏怀古迹五首·其一', 12, 10)   # 逐行输出要显示的汉字串
show_CN(lcd1.desk, '支离东北风尘际，', 24, 40)
show_CN(lcd1.desk, '漂泊西南天地间。', 24, 65)
show_CN(lcd1.desk, '三峡楼台淹日月，', 24, 90)
show_CN(lcd1.desk, '五溪衣服共云山。', 24, 115)
show_CN(lcd1.desk, '羯胡事主终无赖，', 24, 140)
show_CN(lcd1.desk, '词客哀时且未还。', 24, 165)
show_CN(lcd1.desk, '庾信平生最萧瑟，', 24, 190)
show_CN(lcd1.desk, '暮年诗赋动江关。', 24, 215)

lcd1.desk.show()             # 把所有内存中的图案显示到屏幕上
```

这里核心的是 framebuf.monoicon()，把字库当做一个图标映射到帧缓冲区的内存中。用自定义的 show_CN 函数封装输出整个汉字串的操作，调用时给出坐标、给出 UTF8 字串即可。

注意：国标的汉字点阵中每个字节都是高位显示在左，低位显示在右，因此需要设置 convert=2 进行翻转。

其中的 Hanzi 是一个 Python 类，负责处理汉字编码转换和字库提取。因为在 Windows 上得到的汉字都是 UTF8 编码，而国标汉字库都是按照 GB 编码方式排列，必须进行两者之间的转换才行。

以下关于 UTF8->GB 的编码转换代码是从开源的 C 代码转换到 Python 代码，读者可以看代码中的注释，这里不做解释。

这里用到了两个文件，一个是 UTF16 至 GB 编码转换的对照表，另一个是 16x16 的国标汉字点阵库，这两个文件放在 SD 卡中，运行这段代码时需确认当前目录是/sd。

```
"""
Unicode 符号范围     |        UTF-8 编码方式
(十六进制)           |              （二进制）
----------------------+---------------------------------------------
0000 0000-0000 007F | 0xxxxxxx
0000 0080-0000 07FF | 110xxxxx 10xxxxxx
0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
"""
```

```
import ustruct

class Hanzi():
    def __init__(self):
        # 这是 utf16->gb2312 查找表，其中 utf16 字符集升序排序，方便使用二分法由 utf16 查找到对应的 gb2312 字符集
        self.f_utf16 = open('mpy/fontmap/utf16.bin','rb')
        self.len_bin = self.f_utf16.seek(-1,2)
        # 这是点阵字库，按照 GB2312 排列
        self.f_hzk = open('mpy/hzk16', 'rb')

    ########################
    # * @brief utf8 编码转 unicode 字符集(usc4)，最大支持 4 字节 utf8 编码，(4 字节以上在中日韩文中为生僻字)
    # * @param *utf8 utf8 变长编码字节集 1~4 个字节
    # * @param *unicode utf8 编码转 unicode 字符集结果，最大 4 个字节，返回的字节序与 utf8 编码序一致
    # * @return length 0：utf8 解码异常，others：本次 utf8 编码长度
    #
    # Return No. of byte of the uft8 code, and the unicode value
    def UTF8ToUnicode(self, utf8):
        LUT_SIZE = 3
        length_lut = (2,3,4)
        range_lut = (0xE0, 0xF0, 0xF8)
        mask_lut = (0x1F, 0x0F, 0x07)
        #
        b = utf8[0]
        unicode = int()
        # utf8 编码兼容 ASCII 编码,使用 0xxxxxx 表示 00~7F
        if utf8[0] < 0x80:
            unicode = b
            return 1, unicode
        #
        # utf8 不兼容 ISO8859-1 ASCII 拓展字符集
        # 同时最大支持编码 6 个字节即 1111110X
        if b < 0xC0 or b > 0xFD:
            unicode = 0
            return 0, unicode
        #
        for i in range(LUT_SIZE):
            if b < range_lut[i]:
                unicode = b & mask_lut[i]
                length = length_lut[i]
                break
        else:
            # 超过四字节的 utf8 编码不进行解析
            unicode = 0
            return 0, unicode
        #
        # 取后续字节数据
        for i in range(1, length):
            b = utf8[i]
            # 多字节 utf8 编码后续字节范围 10xxxxxx~10111111
```

```
            if b < 0x80 or b > 0xBF:
                break
            unicode <<= 6
            # 00111111
            unicode |= (b & 0x3F)
        # 长度校验
        return 0 if i+1 < length else length, unicode

    ########################
    #  * @brief 4 字节 unicode(usc4)字符集转 utf16 编码
    #  * @param unicode unicode 字符值
    #  * @param *utf16 utf16 编码结果
    #  * @return utf16 长度，(2 字节)单位
    def UnicodeToUTF16(self, unicode):
        # Unicode 范围 U+000~U+FFFF
        # utf16 编码方式：2 Byte 存储，编码后等于 Unicode 值
        utf16 = [0, 0]
        if unicode <= 0xFFFF:
            utf16[0] = unicode & 0xFFFF
            return 1, utf16
        elif unicode <= 0xEFFFF:
            utf16[0] = 0xD800 + (unicode >> 10) - 0x40  # 高 10 位
            utf16[1] = 0xDC00 + (unicode &0x03FF)       # 低 10 位
            return 2, utf16
        return 0, utf16

    def Utf8ToGB(self, utf8):
        len, unicode = self.UTF8ToUnicode(utf8)
        if len == 0:
            return unicode
        len, utf16 = self.UnicodeToUTF16(unicode)
        ####
        start = 0
        end = self.len_bin
        code = utf16[0]
        while start < end:
            mid = int(start + (end - start) / 2)
            mid &= 0xFFFFFC  # align to 4 bytes
            _ = self.f_utf16.seek(mid)
            u16 = ustruct.unpack('HH', self.f_utf16.read(4))
            if code == u16[0]:
                break
            if code > u16[0]:
                if start == mid:
                    raise ValueError('Unknown code: ', utf8)
                start = mid
            else:
                if end == mid:
                    raise ValueError('Unknown code: ', utf8)
                end = mid
        return ((u16[1] >> 8) | (u16[1] << 8)) & 0xFFFF
```

以下的 Hanzi.Get_font()函数，按照给定的 GB 编码，计算出所需汉字在字库中的位置，读出并返回 32 字节的数据，随后将作为图标进行显示。

```
    def Get_font(self, code):
        if isinstance(code, bytes):
            code = self.Utf8ToGB(code)
        offset = 94 * (((code >> 8) & 0xff) - 0xa0 - 1) + ((code & 0xff) - 0xa0 -1)
        self.f_hzk.seek(offset * 32)
        font = self.f_hzk.read(32)
        return font
```