

# 用 户 手 册

## KMBOX

Keyboard & Mouse Box

20250707 V2.0

# 一、Kmbox简介

KMBOX 是一款专为硬件开发与应用设计的高性能 USB 主从设备控制器。它采用纯硬件架构，能够以硬件级别精准模拟和控制键盘、鼠标等 USB 设备，无需依赖任何额外驱动程序或 DLL 注入，确保了设备的兼容性与稳定性。

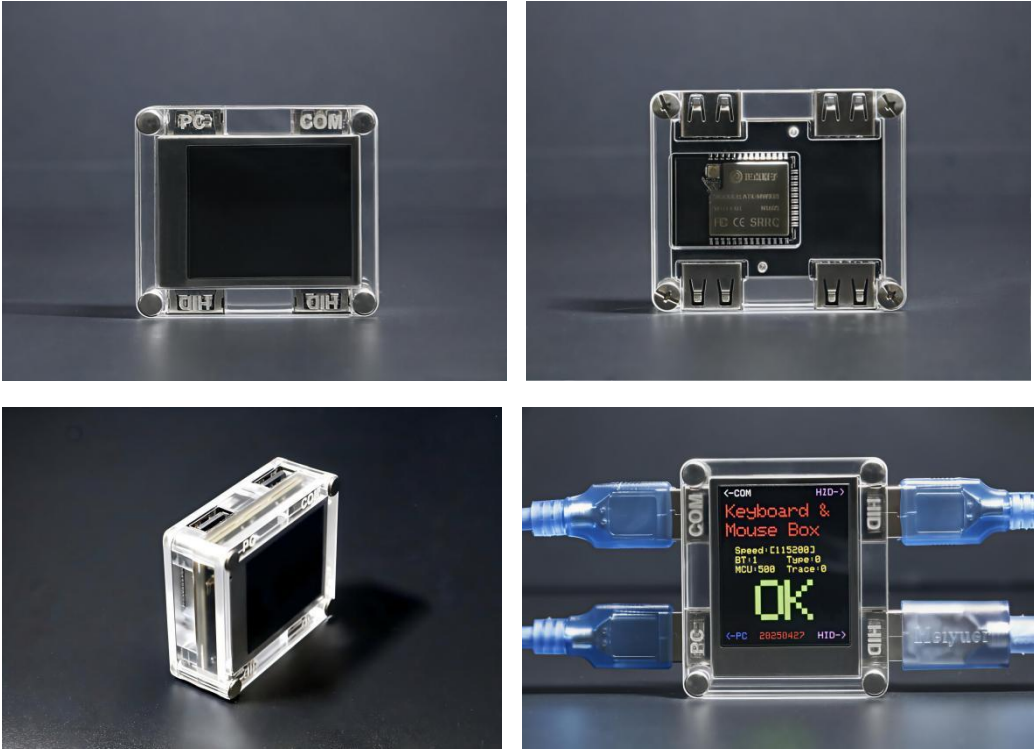
KMBOX 的核心优势在于其强大的可编程性。它搭载独立 CPU，支持脚本运行，完全脱离 PC 系统，可为普通键盘、鼠标（非宏键鼠）赋予高级可编程功能，实现复杂的宏操作。通过简洁的 Python 调用，用户能够轻松实现对设备的深度定制，将普通外设瞬间升级为功能强大的智能设备。

KMBOX 的应用范围广泛，不仅可用于快速开发纯硬件物理外挂，还能助力用户高效完成各种复杂任务。其硬件设计经过严格优化，确保运行安全、高效，且具备出色的抗检测能力，为用户提供可靠保障。

此外，KMBOX 还是一款理想的编程与硬件学习工具。它以简单易用的 Python 语言为入口，帮助用户快速入门编程，深入理解硬件逻辑与上层应用框架。无论是初学者还是资深爱好者，都能通过 KMBOX 在实践中不断提升技能，实现从底层硬件到上层应用的全面掌握。

KMBOX还是DIY爱好者的得力助手，无论是探索硬件底层奥秘，还是开发创新应用，它都能陪伴你一路成长，助你轻松实现各种创意与目标，开启硬件开发与应用的全新旅程。

## kmbox 实物图



## 二、kmbox特性介绍

### 1、核心配置

项目	参数
CPU	ESP32-S3, Xtensa® 32-bit LX6 双核处理器，运算能力最高可达 600 DMIPS 448 KByte
内存	512KB SRAM + 8KB RTC 快速 SRAM + 8KB RTC 慢速 SRAM
存储	16MB（128Mbit）SPI Flash（支持文件系统）
接口	USB(PC)受控设备，USB(COM)串口，USB(HID)×2 设备端
通信	蓝牙 5.0（Bluetooth LE），WiFi 802.11 b/g/n
功耗	3.3V/95mA-97mA（典型），500mA（最小供电需求）

### 2、USB(PC)端口特性（连接受控 PC）

USB(PC)端口连接PC后，Kmbox会自动枚举成一个标准的键盘和鼠标。kmbox模拟出来的是超高回报率的键鼠。下面是 kmbox模拟的键盘与普通键盘关键参数对比：

#### KMBOX模拟键盘、普通键盘性能对比

参数	Kmbox 键盘	常规键盘
回报率	1ms (1000 次/秒)	10ms (100 次/秒)
无冲按键	10 键	6 键
端点长度	64Byte	8Byte

1、回报率就是键盘被主机轮询的时间间隔。

这个间隔越短说明报告次数越多，键盘的硬件运行速度越快。常规键盘是 10ms, 每 秒 100 次报告。kmbox 是 1ms，每秒 1000 次报告。十倍于常规键盘。

2、报告格式就是一包数据能发送的无冲按键数量。

常规键盘采用 1+6 模式，支持 6 键无冲。而 KMBOX 支持 1+10 式，实现 10 键无冲。（PS：理论上可以做到108键无冲，但考虑到人类只有10个手指，10键无冲已完全满足日常使用需求）。

3、端点长度就是数据硬件缓冲区的大小。

他决定了键盘一次传输最多能传多少字节。普通键盘是 8byte。Kmbox 的缓冲是常规键盘的 8 倍。可支持做更多的扩展。

举个简单例子，假设键盘输入“1234567890”这十个数字。

常规键盘 : 第一包 123456-->10ms 后主机接收-->第二包 7890-->10ms 后主机接收-->结束

Kmbox 键盘 : 第一包 1234567890-->1ms 后主机接收-->结束

可以看到，同样的操作，普通键盘需要 20ms，kmbox 只需要 1ms。

**KMBOX模拟鼠标、普通鼠标性能对比**

参数	Kmbox 鼠标	常规鼠标
回报率	1ms (1000 次/秒)	10ms (100 次/秒)
按键数	8 键（可定制）	3 键
X 坐标范围 Y 坐标范围	-32767 ~ 32767（可任意定制）	-127 ~ 127
滚轮范围	-127 ~ 127（可任意定制）	-127 ~ 127
报告长度	6 字节（可任意定制）	4 字节

**1、 回报率：**

kmbox回报率是普通鼠标的10倍。因此反应更灵敏迅速，延迟更低。

**2、 按键数：**

kmbox除了常规的左中右键外还默认多出5个侧键。没有侧键的鼠标接入 kmbox 后就等于有 5 个侧键。（PS：按键数可任意定制）

**3、 XY坐标范围：**

kmbox的XY坐标范围是普通鼠标的256倍

举个例子：

鼠标从屏幕右下角（1920x1080）移动到屏幕左上角（0,0）。普通鼠标一次能移动的 X 最大值是127。那么移动 1920 个单位需要  $1920/127=15.118$ 。取整就是 16 次。一次报告的时间间隔是 10ms。那么 16 次需要  $16*10ms=160ms$ 。kmbox 的 XY 坐标范围是正负 32767。所以移动 1920 只需要 1 包数据就行。也就是 1ms 的时间就够了。

### 3. USB(COM) 功能（连接控制 PC）

- 1) 作为通信接口，连接控制计算机对受控计算机进行控制，控制计算机通过串口向 KMBOX 发送指令，从而控制受控计算机的键盘鼠标操作。
- 2) 设备连接和数据传输，可以连接到其他设备，如传感器、控制器等，实现数据的传输和交互。
- 3) 固件升级，使用升级工具对 KMBOX 的固件进行更新。
- 4) 在开发和调试过程中，通过串口向KMBOX发送指令、配置参数或接收调试信息。

### 4. USB(HID) 功能（连接键盘鼠标）

两个USB(HID)端口用于连接被接管USB设备。目前kmbox默认支持接管外设为USB键盘和鼠标。也就是kmbox能通过USB(HID)端口实时获取键盘鼠标的所有数据，并且在中间把这些底层的数据做处理，再将处理后的数据通过USB(PC)端口（或蓝牙）传给PC。让普通键盘鼠标也拥有可编程宏功能，以此实现任意改键、任意重组、任意屏蔽按键、任意逻辑判断、键盘连点、鼠标连点等等。具体详见kmbox教程。

### 5. kmbox 软件特性

- 1) kmbox 内置 python 释义器。支持Python 编程语言。
- 2) Kmbox 内置键盘鼠标控制模块（km），只需要简单的 API 调用就能实现强大的键盘宏鼠标宏等功能。
- 3) Kmbox 支持条件判断，变量，循环，多线程等 python 语法。
- 4) Kmbox 支持与上位机通信。可以通过串口与kmbox 通信。例如上位机找图找色等。
- 5) Kmbox 目前兼容所有的 window 和 Linux 主机，免驱插上就能用。
- 6) Kmbox 支持文件系统管理，可以存放多个脚本在板子上，随意调用。
- 7) Kmbox 支持固件更新。不需要其他工具，直接通过串口升级固件。

### 6. Kmbox 典型应用

Kmbox 的核心是随意控制键盘鼠标数据。所以只要是键盘鼠标能实现的 kmbox 就能实现。具体请看以下例子

#### 1、 智能辅助键盘宏

- (1) 按键连点宏（点我看视频效果）
- (2) [自动喊话宏（点我看视频效果）](#)
- (3) 键盘改键宏（点我看视频效果）
- (4) .....

## 2、 智能辅助鼠标宏

- (1) [吃鸡压枪宏（点我看视频效果）](#)
- (2) [左键连点宏（点我看视频效果）](#)
- (3) [鼠标绘图宏（点我看视频效果）](#)
- (4) [鼠标精确控制宏（点我看视频效果）](#)
- (5) .....

## 3、 DIY 扩展应用

- (1) [USB 有线键盘鼠标蓝牙化（点我看视频效果）](#)
- (2) [改 USB 游戏手柄蓝牙化玩王者荣耀（点我看视频效果）](#)
- (3) [无线鼠标控制小车](#)
- (4) WIFI 探针（链接）
- (5) 天气预报（链接）
- (6) 物联网开关控制（链接）
- (7) .....

更多功能，欢迎用户自己探索。

# 7. Kmbox 特点

## 1、 安全

对于电脑来说 kmbox 就是一个干干净净的标准键盘和鼠标。标准的 HID 设备，不需要任何驱动，电脑自动识别，不是软件宏、驱动宏可以比拟，从源头上杜绝了封号盗号风险。

## 2、 高效

独立双核 CPU，240Mhz 的超高主频。强劲的数据处理能力和高效的脚本执行速度。不占用 PC 主机 CPU 时间。其性能相当于2.4G主频双核 CPU 拿 10%的性能专门处理键盘和鼠标。

## 3、 简单

Kmbox 使用灵活的 python3 脚本。内置强大的 km 模块。支持条件判断、变量、循环、多线程。能通过简单的 API 调用轻松写硬件外挂。Python 是公认看看就会的语言。

## 4、 实用

KMBOX是一款能够将普通键盘和鼠标功能极大扩展的设备。它赋予键盘和鼠标强大的自定义能力，包括键位修改、自动点击、宏编程等功能。无论是鼠标连点、键盘连点，还是复杂的鼠标宏和键盘宏，KMBOX都能轻松实现。它支持高级游戏操作，如“吃鸡”游戏中的压枪和自动刷怪，以及一键触发多个技能等。此外，KMBOX还能将普通键盘和鼠标转化为蓝牙设备，实现与电脑、手机等设备的无线连接。通过KMBOX，用户可以完全控制所有键盘和鼠标的的数据，极大地提升工作效率和游戏体验。

5、高速

kmbox 可以直接 10 倍提升键盘鼠标的硬件性能。常规键鼠 10ms 报告间隔，每秒 100 次。Kmbox 是 1ms 报告间隔，每秒 1000 次。接上 kmbox 后能让你的键鼠设备硬件超频，时刻保持 10 倍的速度领先。

6、强大

KMBOX搭载16MB的存储空间，这一充足的容量使其能够轻松存储复杂的逻辑代码和配置文件，同时支持高效的文件系统管理。这使得KMBOX成为学习USB技术、Python编程以及软硬件开发的完美工具。其强大的扩展能力允许你外接各种外设模块，实现无限的功能扩展，满足你的DIY需求。

三、Kmbox基础教程

本章介绍 kmbox 的一些基本使用方法。

1. 开箱

配件清单如下：

物料名称	数量	备注
Kmbox设备	1	
USB公对公数据线	2	长度 1.5 米



## 2. 连接电脑

首先将 USB 延长线一端接kmbox 的 USB1 端口。另一端连接电脑的 USB 端口。稍等片刻。你会在设备管理器里看到这些：

Kmbox 连接到 PC 后会默认出现 4 个硬件设备。如上图所示：

### 1、串口（重要）

Kmbox 与主机的通信是通过串口完成。如果没有出现串口，请安装串口驱动。串口驱动是用来调试代码，下载脚本的。如果想玩编程和调试就一定得需要用到。当安装串口驱动后，可以使用任何串口调试软件登录板卡。

驱动下载：<https://github.com/SDRRADIO001/Keyboard-Mouse-Box>

### 2、一个键盘（自动生成）

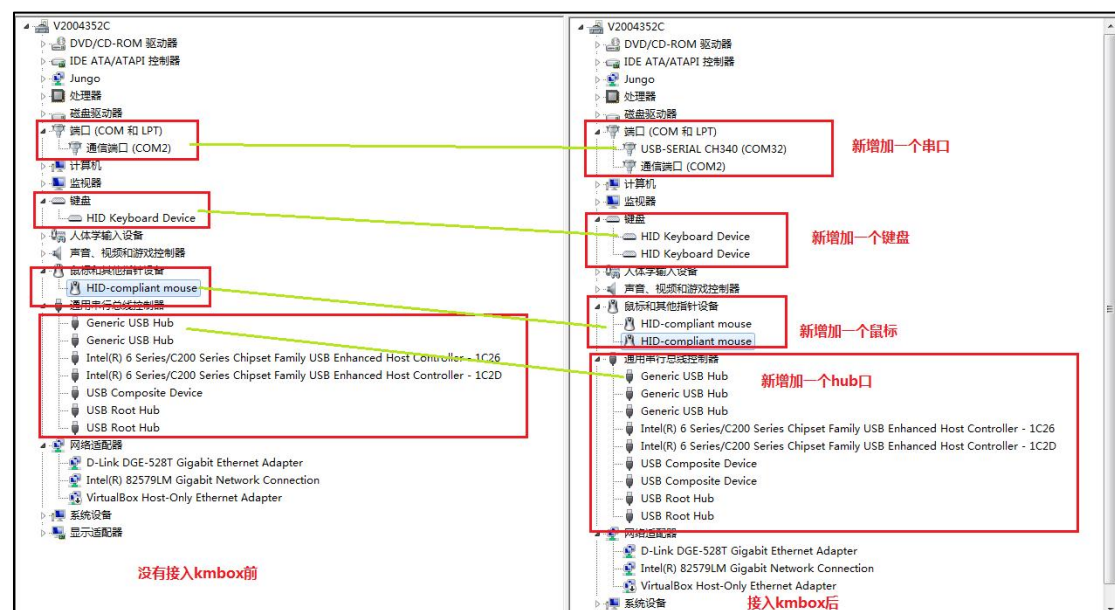
此键盘是 Kmbox 自动模拟生成的。这是一个真实的物理设备。

### 3、一个鼠标（自动生成）

此鼠标也是 kmbox 自动枚举生成的鼠标。真实的物理设备。

### 4、一个 HUB 集线器

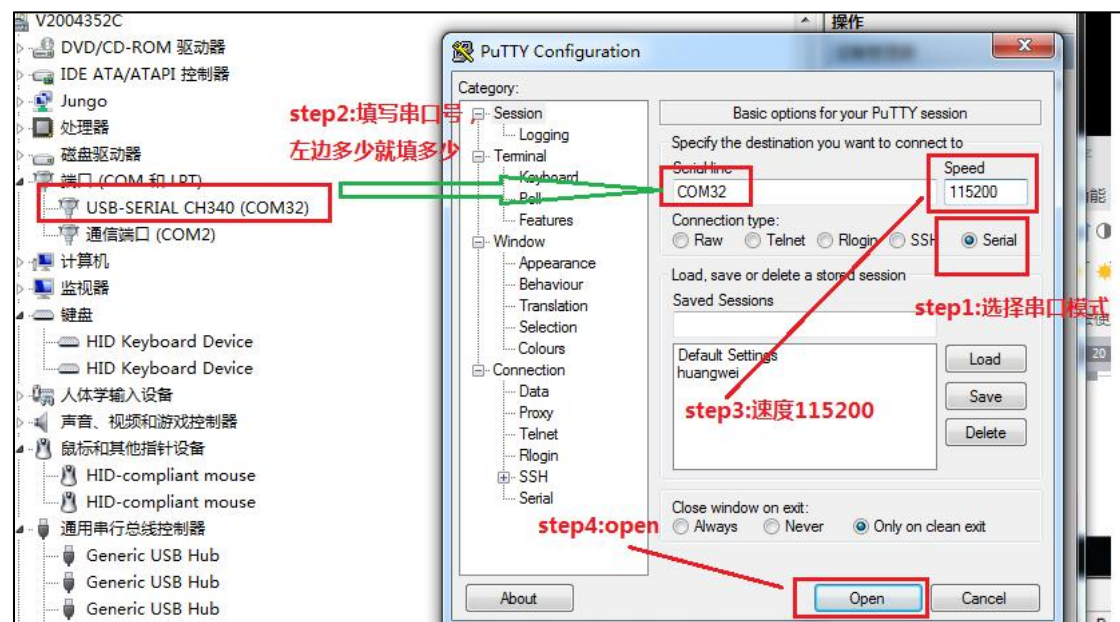
Hub 集线器是kmbox 内部使用，用于键盘鼠标的数据通道和板卡调试传输升级。





### 3. 登录 kmbox

任意串口工具均可登录 kmbox。常见的串口工具有 SecureCRT, upyloader, 串口调试助手等。这里我们以putty为例, 介绍一下简单开发流程。其他工具类似。双击 Putty。按照下图步骤填写, 然后点击open



点击Open 后你可以输入回车, 会出现“>>>”的符号就是与板卡通信成功。你可以输入” help() ”然后回车查看板卡内部的帮助信息。如下图所示:

```
>>>
>>>
>>>
>>> km.help()
*****
欢迎使用km模块帮助系统。如果你对km模块的哪个函数不清楚,
请直接使用km.xxx('help')查看该函数的使用方法和返回值。
例如: isdown函数的使用方法和返回值查询:
输入km.isdown('help') 回车即可

想知道km包含哪些函数, 请输入km.按键盘的Tab键。
*****
>>> 
```

这里就是 kmbox 内部的运行 python 环境。你可以按照python 的语法书写任何你想要的算法和功能。

按 tab 键可以看到当前有哪些模块

```
>>>
__class__      __name__      Pin      SPI
bt             device       gc       km
math          os          sys      time
uos           bdev        TFT     sysfont
spi           tft         model   btl
>>> |
```

看模块里面有哪些函数，例如KM模块，输入“km.”然后按 tab键，以下是KM模块包含的函数

```
>>>
>>> km.
__class__      MAC          Screen      baud
baud_index     catch_kb    catch_ml    catch_mm
catch_mr       catch_ms1   catch_ms2   click
debug          delay       down        freq
getint         getpos     help        init
isdown        keyboard   left        lock_ml
lock_mm        lock_mr     lock_ms1    lock_ms2
lock_mx        lock_my     mask        media
middle         mode       mouse       move
moveAuto       moveto     multidown   multipress
multiup        play       press       reboot
record         remap      rgb         rgb_free
right          rng         setpos      side1
side2          side3      side4       side5
string         table      trace_debug trace_enable
trace_type     up         version     vertime
wheel          zero
>>> |
```

要知道每个函数的作用是什么，可以调用 ‘help’ 参数功能，打印函数帮助信息。例如press函数，可以输入km.press(‘help’)回车，得到如下信息

```
>>> km.press('help')
软件强制单击指定按键：
    press() 包括按下和松开两个动作，例如单击一次键盘a写法如下：
    一:km.press('a') 输入参数是字符串a
    二:km.press(4)   输入参数是a的键值
    press支持模拟手动操作，即控制一次按键按下的时间，press可以额外增加一个或者两个参数
    press(4,50)-----两个参数，表示单击a键，a键按下时间50ms
    press(4,50,100)--三个参数，表示单击a键，a键按下时间50-100ms的随机值
    PS: 推荐使用键值类型，效率更高,速度更快，手动按键正常人速度约为8次每秒。如需模拟请设置合适的按下参数
>>>
```

## 4. 键盘宏编程

### 1) 键盘小常识

电脑是如何识别键盘或者鼠标的按键的。在 USB 系统中，所有数据通信都是主机（PC）发起的。例如键盘上按下一个按键。并不是键盘主动把按键值发送给主机。而是主机主动过来读键盘按了哪些按键。所有的 USB设备都是这样。主机和键盘之间的通信类似这种模式：

主机：键盘你有按键吗？

键盘：没有！

过了一会....（主机等待一个轮询时间）

主机：键盘你有按键吗？

键盘：没有！

过了一会....（主机等待一个轮询时间）

主机：键盘你有按键吗？

键盘：我按下了 123456789 这几个按键

.....

从上面的模式可以看到，USB设备属于从设备，PC是主设备。主设备和从设备之间采用一问一答的方式进行通信。那么主机什么时候读键盘数据呢？多久读一次呢？在 USB 设备的配置描述符中会定义这个值。上面的“过了一会....”对应的就是这个轮询时间。通常键盘鼠标这种低速设备是8ms-10ms（想了解更多请查看 USB 协议文档、HID文档）。也就是主机1秒钟100到125次读取。这个数值越小，主机读数据越勤快，反应就更灵敏。高速USB设备是125us读一次。但并不是越小就越好，键盘鼠标是物理设备，人做不到一秒钟按50次键盘或者50次鼠标，主机过快读取是在浪费主机的资源。其实衡量一个键盘性能的好坏应该主要包括以下两个因素：

#### 一、回报率

回报率也就是键盘的轮询时间间隔。这个间隔越短说明键盘的硬件运行速度越快。常规键盘是10ms，kmbox是1ms，性能10倍于常规键盘。

#### 二、报告长度

常规键盘报告是6键无冲，如果要发送123456789按下，需要分两包发送。第一包123456，第二包 789。

Kmbox目前采用的是10键无冲，发送123456789一包就能发送完成，节约了时间。

## 2) Kmbox如何处理键盘数据

kmbox 与主机和键盘之间的关系。kmbox是串在电脑和USB设备之间的。

接电脑的一端属于 USB 设备。Kmbox需要模拟标准的 USB 设备。出厂默认情况下kmbox模拟的是键盘和鼠标。Kmbox模拟的键盘参数前面的表 1 中已经列举了参数。右边的 USB 端口接的是 USB 外设。那么kmbox 右端则是 USB 主机的角色。作为主机，kmbox 默认识别 HID 类键鼠设备。Kmbox 会按照外设的描述符访问和配置外设。

所有键盘和鼠标的原始数据都会经过 kmbox。然后再发送到电脑。在kmbox内部运行脚本，实时的处理原始的键盘鼠标数据，按照用户脚本的逻辑重新修改打包。再发送给主机。对于主机 PC 来说。他接收是宏处理后的数据，而且是真实的 HID 数据。脚本里面运行的逻辑就相当于是外接键盘和鼠标的逻辑。也就是即使外接的键盘鼠标没有宏功能，有kmbox就等效于键盘鼠标有宏功能。

举个例子，kmbox如何让普通鼠标实现吃鸡压枪的功能。鼠标左键按下是开火。如果kmbox不处理。那么PC主机收到的就是鼠标左键按下，开火。这是一个正常流程。但是如果接上kmbox。在kmbox内部写一个脚本（如果鼠标左键按下，那么再让鼠标向下移动）。这两种数据包发送给主机，开火是我们实现的，鼠标下移是 kmbox 脚本实现的。整个过程中kmbox帮助我们自动鼠标下移压枪，减少了操作难度。Kmbox已经封装好这些功能，只要学会调用就行了。

## 3) Kmbox函数介绍

以下函数用途说明也可以在代码中输入 `km.xxx('help')` 获得。xxx 是你要查询的函数名。

### table 函数

调用方法：`km.table()`

备注说明：该函数用于显示常用键盘按键名称与键值的对应表。

因为 PC 识别按键只认 HID 数据的键值，该函数用于查询键盘上的物理按键对应的10进制数值，所以如果需要查询哪按键对应哪个键值。可以直接调用 `km.table()` 查看。返回如下：

```
kmbox

>>> km.table()

按键名称      按键键值      按键名称      按键键值
space          44          w             26
a              4           s             22
d              7           l             30
2             31           3             32
4             33           5             34
6             35           7             36
8             37           9             38
0             39           b             5
c              6           .            53
e              8           f             9
g             10          h            11
i             12          j            13
k             14          l            15
m             16          n            17
o             18          p            19
q             20          r            21
t             23          u            24
v             25          x            27
y             28          z            29
tab           43          back          42
enter         40          del           76
esc           41          f1            58
f2            59          f3            60
f4            61          f5            62
f6            63          f7            64
f8            65          f9            66
f10           67          f11           68
f12           69          up            82
down          81          left          80
right         79          -            45
=            46          insert        73
home          74          [            47
]            48          \            49
end           77          caps          57
;            51          '            52
,            54          .            55
/            56          print         70
scroll        71          pause         72
pgup          75          f13           104
f14           105         f15           106
f16           107         f17           108
f18           109         f19           110
f20           111         pgdown        78
aplcat        101         ctr-l         224
shift-l       225         alt-l         226
gui-l         227         ctr-r         228
shift-r       229         alt-r         230
gui-r         231         help          254
>>>
```

如图所示空格键” space”，对应的编码值为44。回车键” enter”，对应编码值为40。键值不用死记硬背，用的时候查一查就是啦。如果你要的按键不在这个表里面也是支持的。比如power键，其键值是0x66. 只是我们不常用罢了，也就没有写到这个table中。

## down 函数

调用方法: km.down(value)

down(value) 函数用于控制键盘按键按下, 等同于手动按下物理按键。value 可以是数字类型 (例如 a 键键值4) 或者字符串类型 ('a' 字符串)。推荐使用数值类型。效率更高。value 的值请参考km.table()函数。down()一般与 up()函数配套使用。

例如, 想要键盘 a 键保持按下, 可以有两种书写方式:

1、km.down('a')----直接输入 a 的名称, 这个名称就是 table()中的 key name

```
>>> km.down('a')
>>> aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

注意输入上面这条指令后会一直打印 a, 要停止请按 ctrl+C。

2、km.down(4)----直接输入 a 的键值(4), 这个键值就是table()中的按键键值

```
>>> km.down(4)
>>> aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

down 函数是用脚本控制按键按下, 就算你物理上没有按下, 只要调用了 km.down('a'), 那么PC端也会识别到 a 键按下。

PS: 如果你在控制台操作, 可以按 ctrl+c 终止当前脚本的运行。

Down 函数使用说明如下:

```
>>> km.down('help')
软件强制指定按键一直按下:
例如想让a键按着不松可以有以下两种写法
一:km.down('a') 输入参数是字符串a
二:km.down(4)   输入参数是a的键值
PS:推荐使用键值类型, 效率更高, 速度更快
>>>
```

## multidown 函数

multidown 函数用法和 down 函数一致。唯一的区别是 multidown 函数支持一次性按下多个按键。最多支持 10 个按键同时按下(最多 10 个参数)。

例如你想调用任务管理器, 快捷键是 ctr-l (左 ctrl) +alt-l(左 alt)+delete 键。你可以这么写:

方法一: km.multidown('ctr-l', 'alt-l', 'del') ----直接传字符

方法二: km.multidown(224, 226, 76) ----直接传键值

注意 multi down 一般和 multiup 配套使用

## up 函数

调用方法: km.up(value)

up(value)用于设置键盘指定按键弹起, 等同于手动松开物理按键value 可以是数字类型 (例如空格键键值 44) 或者字符串类型 ('space' 字符串)。推荐使用数值类型。效率更高。value 的值请参考km.table()一般与 down()函数配套使用。

例如要松开空格键可以有下面两种写法:



1、 km.up('space') ----直接输入空格键的名称，这个名称就是 table() 中的 key name

```
>>> km.up('space')
>>>
```

2、 Km.up(44) ----直接输入空格键的键值，这个键值就是 table() 中的 key value

```
>>> km.up(44)
>>>
```

Km.up(44) 后空格键弹起，光标不继续移动。

PS: 这个函数调用后相当于松开指定按键，就算你物理上按住空格不松，调用该函数后空格也会松开。

Up 函数使用说明如下：

```
>>> km.up('help')
软件强制松开指定按键：
  例如你按着键盘上的a不松，通过软件让a松开。可以有以下两种写法
  一: km.up('a') 输入参数是字符串a
  二: km.up(4)   输入参数是a的键值
PS: 推荐使用键值类型，效率更高，速度更快
>>>
```

## multiup 函数

multiup 函数用法和 up 函数一致。唯一的区别是 multiup 函数支持一次性抬起多个按键。最多支持 10 个按键同时抬起(最多 10 个参数)。

例如你想解除调用任务管理器，快捷键是 ctrl-l (左 ctrl) + alt-l (左 alt) + delete 键的抬起。你可以这么写：

方法一： km.multidup('ctrl-l', 'alt-l', 'del') ----直接传字符

方法二： km.multiup(224, 226, 76) ----直接传键值

注意 multi down 一般和 multiup 配套使用。

## press 函数

press(value) 函数用于键盘单次敲击指定按键，等同于手动单击一次物理按键。

value 可以是数字类型或者字符串类型（数值类型效率高），value 的值请参考 km.table() 函数。其功能等同于先调用 down() 再调用 up()。

例如，想按一下键盘的 'a' 键。可以有下面两种写法：

1、 km.press('a') ----直接输入 a 的名称，这个名称就是 table() 中的 key name

```
>>> km.press('a')
>>> a
```

2、 Km.press(4) ----直接输入 a 的键值，这个键值就是 table() 中的 key value

```
>>> km.press(4)
>>> a
```

press 函数还可以带两到三个参数，用于指定按下指定按键的时间，可以模拟人工操作。例如按下 a 键 200ms，可以这样写：km.press(4, 200)

这里的 200 是指 a 键按下 200ms，主要用于模拟长按 200ms 后松开。也可以带三个参数：km.press(4, 80, 100)

这里的 80, 100 指的是 a 持续按下的时间在80-100ms 之间的随机值。主要用于模拟人操作。如果不带参数，kmbox 会以最快速度执行完一次按键。一般 1ms 时间。正常人不可能做到，带参数后，会让你的脚本更像人在操作。可以过行为检测。请注意时间 参数的合理性，press 的详细使用方法可查询 km.press(‘help’):

```
>>> km.press('help')
软件强制单击指定按键:
press() 包括按下和松开两个动作, 例如单击一次键盘a写法如下:
一:km.press('a') 输入参数是字符串a
二:km.press(4) 输入参数是a的键值
press支持模拟手动操作, 即控制一次按键按下的时间, press可以额外增加一个或者两个参数
press(4, 50)-----两个参数, 表示单击a键, a键按下时间50ms
press(4, 50, 100)---三个参数, 表示单击a键, a键按下时间50-100ms的随机值
PS:推荐使用键值类型, 效率更高, 速度更快,
>>>
```

### multipress 函数

multipress 函数用法和 press 函数一致。唯一的区别是 multipress 函数支持一次性单击多个按键。最多支持 10 个按键同时按下(最多 10 个参数)。

例如你想调用任务管理器，快捷键是 ctr-l (左 ctrl) +alt-l(左 alt)+delete 键。你可以这么写：

方法一：km.multipress(‘ctr-l’, ‘alt-l’, ‘del’) ----直接传字符

方法二：km.multipress(224, 226, 76) ----直接传键值

### string 函数

string 函数用于模拟键盘敲击输入一连串的字符，例如我们想直接输入“hellow word”。最简单的方法就是调用 km.string(‘hellow word’)函数，当然如果你不嫌麻烦也可以 h, e, l, l, o, w... 一一调用 km.press() 函数。其实 string 函数内部就是这么干的。其运行截图如下：

```
>>>
>>> km.string('hellow word')
>>> hellow word
```

注意：string 的字符串支持 0-9 的数字以及主键盘区域的所有大小写字母以及字符。不支持小键盘。因为小键盘会和主键盘冲突。使用时注意转义字符(\\, ‘, ”)的使用。

String 函数的参数为字符串，支持转义字符(\\r(回车) \\n(空格) \\t(Tab))，例如需要做自动登录脚本。一般是输入账号，Tab, 密码，回车。可以使用 string 函数一步完成，例如：

账号：KmBox



密码: www.clion.top

可以这样写: `km.string( 'KmBox\\twww.clion.top\\r' )`

其中\\t和\\r 表示的是 Tab 和回车。(注意转义字符需要多加一个斜杠)

**isdown 函数 (需将被检测的键盘接到 kmbox 上)**

`isdown(value)` 函数用于检测指定的 value 按键是否按下。value 可以是数字类型或者字符串类型 (数值类型效率高), value 的值请参考 `km.table()` 函数。这个函数主要用在逻辑判断中, 用于判断指定的按键是否被按下。

例如, 想检测键盘上的 'a' 键是否按下。可以有下面两种写法:

- 1、`km.isdown( 'a' )` ----直接输入 a 的名称, 这个名称就是 `table()` 中的 key name
- 2、`km.isdown(4)` ----直接输入 a 的键值, 这个键值就是 `table()` 中的 key value

返回值有四个:

- 0: 没有按下--物理软件均没有按下
- 1: 物理按下--物理按下但是软件没按下
- 2: 软件按下--软件按下了但是物理没有按下
- 3: 物理和软件均按下--软件和物理均按下了

这里我们做个约定, 真实键盘按下叫做物理按下, 用软件脚本按下叫做软件按下。

所以 `isdown` 函数的返回值有以上四个就很好理解了。

`Isdown` 函数使用说明如下:

```
>>> km.isdown('help')
用于查询键盘按键是否按下。例如要查询空格键是否按下可以这样写:
  isdown('space') 返回值:0:没有按下 1: 物理按下 2: 软件按下 3: 物理软件均按下
  isdown('44')     返回值:0:没有按下 1: 物理按下 2: 软件按下 3: 物理软件均按下
PS:isdown只管调用的瞬间按键是否按下。如果需要捕获处理所有时间的按键, 请使用mask和catch处理
>>> █
```

注意: `isdown` 函数检测的是 HID 报告里的内容。如果你的键盘压根就没插到 kmbox。Kmbox 检测不到任何东西。不要天真的以为电脑会告诉 kmbox。能是能, 自己去写驱动。目前 kmbox 检测功能仅限于接到 kmbox 上的键鼠。

**getint 函数**

此函数和 `table` 函数一样, 用于查询键盘字符串对应的按键值。例如我们需要知道键盘上 "esc" 按键对应的键值是多少, 可以直接使用 `table()` 然后找到 "esc" 的值。还有一种方法就是使用 `getint` 函数。例如:

```
>>>
>>> km.getint('esc')
41
>>> █
```

有 table 就没必要用getint了吧，存在必然有道理。我们先看看下面这段代码

```
Run=getint('f1')
if km.isdown(Run):
    km.string("welcom")
else
    km.string("good bye")
```

如上所示，我们把 f1 按键的键值给 Run, 如果 Run 按下，我们就打印欢迎。松开就打印再 见。如果你想更换快捷键，不是 F1 而是 F10，脚本改动就只需要改 getint 内的字符串即可，其他地方都不需要改动。如果没有 getint. 那么每个 Run 的地方你都需要修改一次。所以这样的好处是可以写脚本时设置变量，暴露接口比较方便。便于脚本的通用性。getint 函数使用说明如下：

```
>>> km.getint('help')
用于查询常规按键对应的10进制键值：
  例如要返回空格键对应的键值:km.getint('space')
  返回值为44。也可以通过table()函数查看常用按键键值对照表
>>> █
```

#### mask 屏蔽函数（需将被屏蔽的键盘接到 kmbox 上）

mask 函数用于屏蔽或者查询键盘的物理按键是否被屏蔽。例如，你想屏蔽键盘上的回车键(enter)。你可以这么写：km.mask('enter',1)  
第一个参数是字符串 enter. 详见 table() 表。第二个参数是1表示屏蔽。

```
>>> km.mask('enter',1)
>>> █
```

屏蔽完回车键后，你键盘上的回车键不管是按下还是松开，PC 都无法识别到。相当于你把 回车键从你键盘上去掉了。另外一种写法是回车键的键值。

```
>>> km.mask(40,1)
```

这里的 40 是 enter 对应的键值。

你可以通过 isdown() 函数来查询被屏蔽按键的状态(按下还是弹起)。

```
>>> km.isdown('enter')
0
>>> km.isdown('enter')
1
>>> █
```

如上图，屏蔽回车后按着回车不松，通过 isdown 查询。返回值是 1，说明物理按下。注意。Mask 后虽然 kmbox 不会将键值发送给 PC，但是它还是会接收这个键值。因为我们在做 键盘连点时，希望物理按键不会影响到我们的脚本。例如你想做个空格连点。期望按着空 格不松就不停的空格 down 和 up. 由于你物理上是一直接着不松。脚本软件执行 up 的一瞬间。键盘是松了。但是紧接着又会按下。所以为了保证脚本不受物理按键的影响。可

以使用 mask 函数来屏蔽物理按键。这样空格键的状态就完全由软件来决定了。物理的按键不会 干预到脚本的逻辑。

如果你发现某个按键不起作用。首先可以查一查这个按键有没有被屏蔽。直接输入：

Km.mask(按键名或者键值)

```
>>> km.mask('enter')
1
>>>
```

例如刚刚屏蔽和回车，看到其返回值是 1，说明被屏蔽了。要解除按键屏蔽，将 mask 的第二个参数改成 0 即可。

```
>>> km.mask('enter',0)
>>> km.mask('enter')
0
>>>
```

解除屏蔽后，再次查询回车是否被屏蔽，返回值为 0 说明没有屏蔽。再次按回车键就能正 常使用了。

Mask 函数还有一个比较高级的用法。那就是用来设置某个按键为捕获模式。例如你想知道 空格键按了多少下，如果采用普通的 isdown 的方法，可能轮询得不够快。导致你错过了检 测指定按键。这时候你可以采用 mask('space',2)来设定空格键实时捕获。保证不漏掉 任何一次的按键。此时你可以通过 catch\_kb 函数来提取按下的按键。

Mask 函数使用说明如下：

```
>>> km.mask('help')
设置和查询物理按键是否被软件屏蔽：
mask(44)      :查询空格键是否被软件屏蔽.返回值：0没有被屏蔽  1已被屏蔽  2软件捕获模式
mask('space') :查询空格键是否被软件屏蔽.返回值：0没有被屏蔽  1已被屏蔽  2软件捕获模式
mask(44,0)    :解除空格键被软件屏蔽状态
mask('space',1):设置空格键被软件屏蔽
mask(44,2)    :设置空格被屏蔽且捕获所有空格。通过调用catch返回被捕获的按键和次数
PS:注意屏蔽了按键后，物理按键值不会发送给主机。可以通过isdown查询或者catch查询被屏蔽
按键是否被按下。如果需要捕获记录所有时刻屏蔽的按键，推荐使用catch函数
>>>
```

## init 初始化函数

init 函数相当于恢复最初什么都没操作，键盘鼠标什么都没按的状态。因为你可能写脚本会屏蔽很多按键。当你想释放所有脚本的的键盘鼠标按键时可以执行该函数。避免一个一个的还原原来的屏蔽。

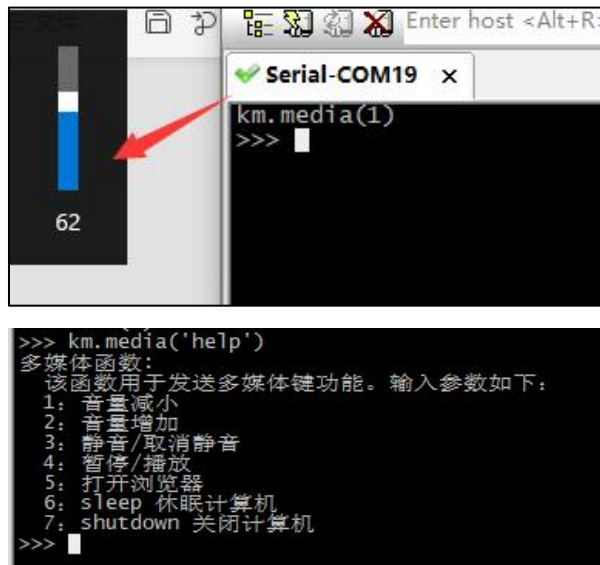
## media 多媒体功能键函数

media 函数用于模拟多功能键的作用，你的键盘可以没有多功能键，kmbox 有就等于外接键盘有。该函数调用一次相当于 press 一次对应的多功能键。多功能键包括以下几个功能：

- 1) 音量减小
- 2) 音量增加

- 3) 静音/取消静音
- 4) 暂停/播放（一般播放器有效）
- 5) 打开浏览器
- 6) sleep 休眠计算机
- 7) shutdown 关闭计算机

例如：你想减小电脑音量，可以直接输入 `km.media(1)` 。如下图所示，详细用法见 `km.media('help')`



### remap 改键函数（需将被改键的键盘接到 kmbox 上）

remap 函数是一个很好玩的函数。俗称改键函数。原理是按下键盘上的 A 键，kmbox 自动把 A 键重新映射成 B 键。也就是就实现了 A 键的改键功能。使用 remap 函数你可以重新自定义键盘上的所有按键。还是一样，不懂看帮助文档。

这个函数有个好处是能随时改键。玩游戏时经常会有按键不顺手。用这个函数就能随意将你需要的按键改到任何地方。

### catch\_kb 捕获按键函数（需将被检测键盘接到 kmbox 上）

catch\_kb 函数比较特殊，它需要调用 [mask\(xxx, 2\)](#) 后才会被激活。catch\_kb 函数是用来捕捉所有 mask 函数指定的按键。首先需要指出。代码运行是需要时间的。比如你想检测 a 键有没有按下时。前面很自然的想到使用 `isdown('a')`。例如你有个单线程的脚本。

```
If isdown(a):
```

```
    Print('a is down')
```

```
    Sleep(10)
```

这个脚本看似在检测 a。但是实际效果很糟糕。因为那个 `sleep(10)` 直接让代码停止 10 秒。这 10 秒内不管你怎么按 a，`isdown` 函数是检测不到的。因为代码没有运行到 `isdown` 函数。所以 `catch_kb` 函数就应运而生。前面说到 `catch_kb` 函数是需要先调用 `mask` 函数才能激活。

首先调用 `km.mask('a', 2)`。Mask 函数返回后a键按都会捕捉到`catch_kb()`函数中。现在 你默数一下按了a多少次。然后你再调用`km.catch_kb()`函数。你会发现，每调用一次返回 一个4，返回5次后变为0。(ps:楼主是两个键盘。所以调用屏蔽后用另外键盘输入的 ‘a’ 。

```
>>> km.remap('help')
按键重映射(改键)函数:
重映射函数俗称改键函数。可以将任意按键映射为其他按键
例如你想把a键映射为b键。写法如下:
写法1:km.remap('a','b')即当按a时，等同于按b,a消息不会上传PC
写法2:km.remap(4,5) 其中4是a的键值，5是b的键值
如果要清除所有的重映射设定请调用:km.remap(0)
PS:重映射仅支持ctrl, shift, alt, gui以外的按键。被重映射的按键不可以调用mask屏蔽。
重映射之后的的按键不支持获取按键状态。(因为实际的物理键没按下嘛，只是软件设置按下的)
>>>
```

`catch_kb` 的用法如下:

```
>>> km.catch_kb('help')
捕获键盘指定按键:
当调用mask(xxx, 2)后，xxx按键将被实时监测,可以截获xxx按键所有按下消息
xxx按键值不会发送给pc,需要自己处理。catch有两种模式，阻塞模式和非阻塞模式
km.catch_kb():是不带参数的非阻塞模式。没有发生捕捉事件返回0.捕捉到指定按键后返回按键值。
非阻塞模式常用在快速轮询,km.catch_kb(0)也是非阻塞模式
km.catch_kb(1):是带参数的阻塞模式。函数在没有捕捉到指定的按键前是不会返回的。
所以阻塞模式建议在多线程中使用。不然当前线程会被阻塞
PS:一定要调用mask(xxx, 2)函数后才能使用catch_kb函数。
>>>
```

`catch` 函数还有个用法就是一键 N 技能。例如：你在脚本里写一个这样的逻辑。

如果 a 键按下了，那么就依次按 12345. 要实现这种逻辑写法很多。

方法 1: 轮询法

```
If km.isdown('a'):
```

```
Km.string('12345')
```

```
Do something ...
```

这个方法的好处是简单，缺点是如果 `Do something ...` 的时间超过了你按键的时间。那么a的检测可能就不及时。

方法 2: 多线程捕捉

首先我们开启a的捕捉，然后启用一个线程专门执行 `catch` 函数。主线程里面执行 `Do something ...`，这样可以保证 a 的捕捉不会受到 `do something` 的影响。多线程的方法后面有专门的探讨。

以上内容就是 `kmbox` 为大家提供的键盘有关的 API 函数。其实键盘也就两种状态，按键按下，按键弹起。如果你有更多的需求也可以与我联系。欢迎提意见！

## 5. 鼠标宏编程

kmbox 模拟的鼠标技术参数：

	Kmbox-鼠标	其他鼠标
回报率	1ms (1000 次/S)	10ms (100 次/S)
按键数	8 按键 (左中右+5 侧键。可定制任意按键)	3 按键 (左中右键)
X 坐标范围	-32767 ~ 32767 (可任意定制)	-127 ~ 127
Y 坐标范围	-32767 ~ 32767 (可任意定制)	-127 ~ 127
滚轮范围	-127 ~ 127 (可任意定制)	-127 ~ 127
报告长度	6 字节 (可任意定制)	4 字节

回报率：普通鼠标报告10ms一条。kmbox是1ms一条，10ms能执行10条指令。

按键数：普通鼠标只有左中右一共三个。kmbox是8个按键（可任意定制按键数）。

坐标范围：普通鼠标一字节。kmbox是两字节。范围广度是普通鼠标的256倍。

举个简单的例子。在一个 1920x1080的显示器上。普通鼠标从屏幕右下角（1920, 1080）移动到左上角（0, 0），按照最大步进127来算（一字节8bit，1bit符号+7bit(127)数据）。需要x轴方向移动  $1920/127=15.118$  次，取整就是16次。普通鼠标需要发送 16 包数据。每包数据间隔10ms，一共需要160ms。kmbox的X步进是32767，同样的移动，只需要一包数据，一次传输耗时1ms。

接下来开始kmbox鼠标宏函数的学习。

### left 鼠标左键控制函数

left 函数用于软件控制鼠标左键。鼠标左键就两个状态，按下和弹起。定义按下是1，弹起是0。不给参数时是查询鼠标左键的状态，返回值如下：

0：弹起

1：物理按下

2：软件按下

3：物理软件均按下

当给参数时是设置鼠标左键的状态：

0：设置鼠标左键弹起

1：设置鼠标左键按下

```
>>> km.left('help')
用于控制和查询鼠标左键状态：
没有参数时是查询鼠标左键状态，返回值 0：松开 1：物理按下 2：软件按下 3：物理软件均按下
有参数时是设置鼠标左键状态，left(1)鼠标左键按下，left(0)鼠标左键松开
>>>
```

注意，如果要检测鼠标左键是否按下，请将被检测的鼠标接到kmbox上。



### right鼠标右键控制函数

right函数用于软件控制鼠标右键。鼠标右键就两个状态，按下和弹起。定义按下是1，弹起是0。不给参数时是查询鼠标右键的状态：

0：弹起

1：物理按下

2：软件按下

3：物理软件均按下

当给参数时是设置鼠标右键的状态：

0：设置鼠标右键弹起

1：设置鼠标右键按下

用法如下图所示：

```
>>> km.right('help')
用于控制和查询鼠标右键状态：
    没有参数时是查询鼠标右键状态，返回值 0：松开 1：物理按下 2：软件按下 3：物理软件均按下
    有参数时是设置鼠标右键状态，right(1)右键按下，right(0)右键松开
>>> |
```

### middle鼠标中键控制函数

middle函数用于软件控制鼠标中键。鼠标中键就两个状态，按下和弹起。定义按下是1，弹起是0。不给参数时是查询鼠标中键的状态：

0：弹起

1：物理按下

2：软件按下

3：物理软件均按下

当给参数时是设置鼠标中键的状态：

0：鼠标中键弹起

1：鼠标中键按下

用法如下图所示：

```
>>> km.middle('help')
用于控制和查询鼠标中键状态：
    没有参数时是查询鼠标中键状态，返回值 0：松开 1：物理按下 2：软件按下 3：物理软件均按下
    有参数时是设置鼠标中键状态，middle(1)中键按下，middle(0)中键松开
>>> |
```

Ps:很多鼠标中键是滚轮，滚轮是可以按的。

## click 鼠标按键单击函数

```
>>> km.click('help')
用于控制鼠标按键点击:
    输入参数有两个, 第一个参数是指定鼠标单击哪个按键
    第二个参数是要点击多少次, 没有第二个参数默认单击1次
km.click(0):单击鼠标左键(效果等于: left(1)    left(0))
km.click(1):单击鼠标右键(效果等于: right(1)   right(0))
km.click(2):单击鼠标中键(效果等于: middle(1)) middle(0)
鼠标左键单击10次写法: km.click(0,10)
>>>
```

click函数用于软件控制鼠标按键单击, 单击哪个按键由click的参数决定。如上图所示, 第一个参数是要单击哪个按键。第二个参数是单击次数。目前单击左键是 0, 中键是 1, 右键是 2, click 函数将down和up打包成一个整体。另外第二个参数是单击次数。如果想双击, 那么把第二个参数改成 2 就是双击。依次类推。当然, 你也可以使用 down和up模拟单击, 中间加延迟, 可以控制一次单击按下时间和抬起后的时间。如果需要模拟人操作, 尽量使用 down和up来模拟。Click的速度会很快, 快得不像人操作。

## wheel 鼠标滚轮控制函数

wheel 函数用于控制鼠标滚轮。滚轮可以上下移动, 输入参数为滚轮滚动单位值, 上移动 为正, 下移动为负。例如滚轮上移动10个单位。和向下移动10个单位截图如下:

<pre>&gt;&gt;&gt; km.wheel(10) 滚轮上移动 &gt;&gt;&gt; &gt;&gt;&gt; km.wheel(-10) 滚轮下移动 &gt;&gt;&gt;</pre>	<pre>&gt;&gt;&gt; km.wheel('help') 鼠标滚轮设置: wheel(100) :滚轮上移动100个单位 wheel(-100):滚轮下移动100个单位 滚轮移动范围为: -127~+127 &gt;&gt;&gt;</pre>
---	--

## side1、side2、side3、side4、side5 鼠标侧键控制函数

这些函数用于控制鼠标的侧键状态, 输入参数可有可无, 无参数是查询侧键状态:

- 0: 侧键弹起
- 1: 侧键物理按下
- 2: 侧键软件按下
- 3: 侧键物理软件均按下

有参数是设置侧键状态:

- 0: 是侧键弹起
- 1: 是侧键按下 使用截图如下:



```

>>>
>>> km.side1(1)
>>> km.side1(0)
>>> km.side1()
0
>>>
>>>
>>> km.side2()
0
>>> km.side2(1)
>>> km.side2(0)
>>>

```

Ps: side1和side2一般是翻页功能，就算你的鼠标没有侧键也没关系。Kmbox可以帮助实现。

```

>>> km.side1('help')
用于控制和查询鼠标侧键1的状态:
没有参数时是查询侧键1状态, 返回值 0: 松开 1: 物理按下 2: 软件按下 3: 物理软件均按下
有参数时是设置侧键1状态, side1(1)侧键1按下, side1(0)侧键1松开
>>>

```

### move鼠标移动控制函数

move用来控制鼠标相对移动，输入参数(x, y)是数值类型。范围为-32767 到+32767。并且 规定x向右为正，向左为负。Y向下为正，向上为负。例如让鼠标向右和向下移动10个单位，可以这样写：

```

>>> km.move(10,10)
>>> 

```

move 移动是相对移动。任何时候都是以当前坐标点为参考。相对移动x和y个单位。如果要精确定位移动，请使用后面的moveto函数。

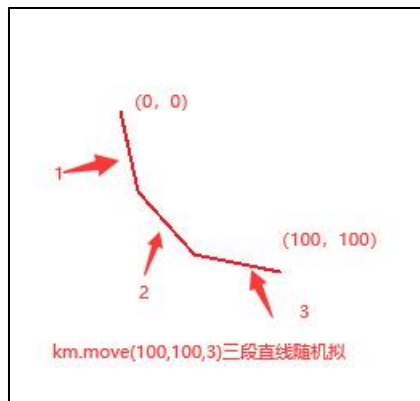
```

>>> km.move('help')
用于鼠标的相对移动, 设置鼠标相对于当前位置偏移(x, y)个单位:
向右和向下移动10个单位: move(10, 10)
向左和向上移动10个单位: move(-10, -10)
x, y允许的范围是-32767~+32767
注意: 如果想精确控制鼠标移动请关闭windows鼠标加速算法
>>>
>>>

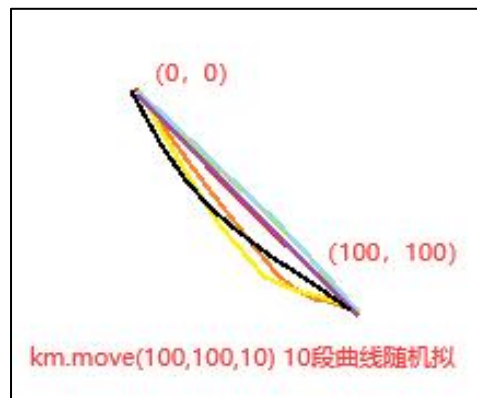
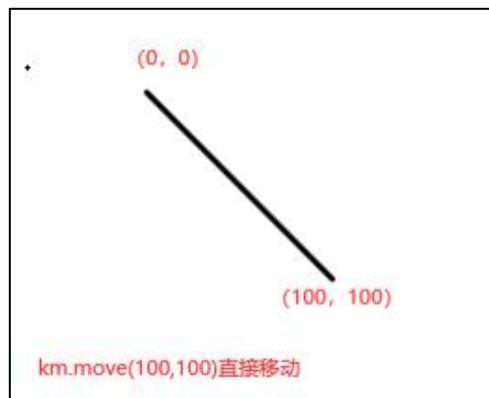
```

以上是 move 函数的基本用法，move 函数如果只给两个参数，那么鼠标移动将在 1ms 内完成。没有中间过程，实际效果是从当前点阶跃移动到对应的偏移点。这种效率最高，速度最快。如果你需要模拟人工轨迹移动，可以给move函数再加上一个参数，这个参数是用来约束从起点到终点你期望用几条直线来拟合。由常识可知，第三个参数数字越大，拟合出来的曲线就会越平滑。

3条直线拟合从（0，0）到（100，100）的移动。



1条直线和10条直线拟合从  $(0, 0)$  到  $(100, 100)$  的移动。

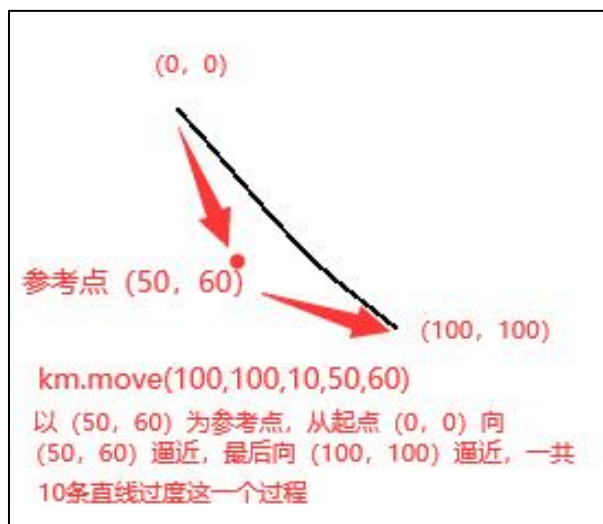


上图是鼠标轨迹。你可以发现，拟合点数越大，起点到终点的过度曲线就越平滑。所以，如果想控制鼠标模拟人操作，可以给 move 加上第三个参数。可以将阶跃状态改变成连续的状态。

另外你会发现，每次调用 move(100, 100, 10) 得到的鼠标轨迹是不一样的。因为过两个点的曲线有无数条，kmbox内部会自动随机取点。保证平滑的由起点到终点。

如果你不想让kmbox自动随机取点，而是想实际控制中间的过程曲线再给move函数增加两个参数，用来做参考逼近。

km.move(100, 100, 10, 50, 60)



其中的 (50, 60) 就是参考点坐标。kmbox 会自动按照参考点来拟合起点到终点。给定参考点后，kmbox的轨迹就是唯一确定的。曲线走向怎样，扭曲程度怎样都是由这个参考点决定的。

### lock\_mx 鼠标 x 方向屏蔽函数

Lock\_mx(lock mouse X)函数用于屏蔽鼠标 x 轴方向的物理输入，多用于确保软件 x 方向移动不受鼠标物理移动的影响。

输入参数为 1 时是锁定 x 轴方向：

当 X 轴方向被锁定以后，鼠标左右移动将不起作用，只能上下移动。

输入参数为 0 时是解锁 x 轴方向的锁定。

无输入参数时是查询 x 轴的锁定状态。

返回值 0：鼠标 x 轴方向未锁定

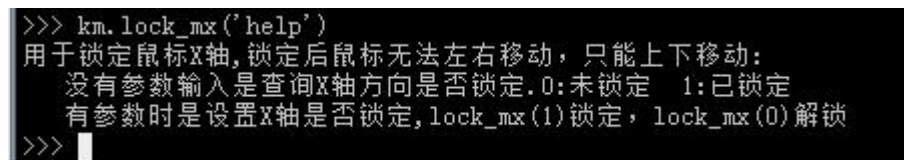
返回值 1：鼠标 x 轴方向已锁定

其用法截图如下：



```
>>> km.lock_mx(1)
>>> km.lock_mx()
1
>>> km.lock_mx(0)
>>> km.lock_mx()
0
>>>
```

在锁定 X 轴后，鼠标的物理移动不会影响到脚本里的移动。保证脚本逻辑不被破坏。请记住加锁和解锁配套使用。



```
>>> km.lock_mx('help')
用于锁定鼠标X轴, 锁定后鼠标无法左右移动, 只能上下移动:
  没有参数输入是查询X轴方向是否锁定. 0: 未锁定  1: 已锁定
  有参数时是设置X轴是否锁定, lock_mx(1) 锁定, lock_mx(0) 解锁
>>>
```

### lock\_my 鼠标 y 方向屏蔽函数

Lock\_my(lock mouse Y)函数用于屏蔽鼠标 y 轴方向的物理输入，多用于确保软件 y 方向移动不受鼠标物理移动的影响。

输入参数为 1 时是锁定 y 轴方向：

当 Y 轴方向被锁定以后，鼠标上下移动将不起作用，只能左右移动。输入参数为 0 时是解锁 y 轴方向的锁定。

无输入参数时是查询 y 轴的锁定状态。

返回值 0：鼠标 x 轴方向未锁定

返回值 1：鼠标 x 轴方向已锁定

```
>>> km.lock_my('help')
用于锁定鼠标Y轴,锁定Y轴后鼠标无法上下移动,只能左右移动:
  没有参数输入是查询Y轴方向是否锁定.0:未锁定 1:已锁定
  有参数时是设置Y轴是否锁定,lock_my(1)锁定,lock_my(0)解锁
>>>
```

### lock\_ml 鼠标左键屏蔽函数

Lock\_ml(lock mouse left)函数用于屏蔽鼠标左键的物理输入,多用于确保软件左键控制不受鼠标物理左键的影响。

输入参数为 1 时是锁定鼠标左键:

当鼠标左键被锁定以后,物理上怎么点鼠标都没用。

输入参数为 0 时是解锁鼠标左键。

无输入参数时是查询左键的锁定状态。

返回值 0: 鼠标左键未锁定

返回值 1: 鼠标左键已锁定

请记住加锁和解锁配套使用。

注意 lock\_ml 后,可以调用 catch\_ml()来捕捉鼠标左键按下和抬起,详见 catch\_ml 的用法。

```
>>> km.lock_ml('help')
用于软件锁定鼠标左键,锁定鼠标后物理点击左键不会有反应:
  没有参数输入是查询鼠标左键是否锁定.0:未锁定 1:已锁定
  有参数时是设置鼠标左键是否锁定,lock_ml(1)锁定,lock_ml(0)解锁
  锁定后可以用left()查询左键实际状态或者catch_ml捕获状态
>>>
```

### lock\_mm 鼠标中键屏蔽函数

Lock\_mm(lock mouse middle)函数用于屏蔽鼠标中键的物理输入,多用于确保软件中键控制不受鼠标物理中键的影响。

输入参数为 1 时是锁定鼠标中键:

当鼠标中键被锁定以后,物理上怎么点鼠标都没用。

输入参数为 0 时是解锁鼠标中键。

无输入参数时是查询中键的锁定状态。

返回值 0: 鼠标中键未锁定

返回值 1: 鼠标中键已锁定

请记住加锁和解锁配套使用。

注意 lock\_mm 后,可以调用 catch\_mm()来捕捉鼠标中键按下和抬起,详见 catch\_mm 的用法。

```
>>> km.lock_mm('help')
用于锁定鼠标中键,锁定鼠标后物理点击中键不会有反应:
  没有参数输入是查询鼠标中键是否锁定.0:未锁定  1: 已锁定
  有参数时是设置鼠标中键是否锁定,lock_mm(1)锁定, lock_mm(0)解锁
  锁定后可以用middle()查询中键实际状态或者catch_mr捕获状态
>>>
```

### lock\_mr 鼠标右键屏蔽函数

Lock\_mr(lock mouse right)函数用于屏蔽鼠标右键的物理输入，多用于确保软件右键控制不受鼠标物理右键的影响。

输入参数为 1 时是锁定鼠标右键：

当鼠标右键被锁定以后，物理上怎么点鼠标都没用。

输入参数为 0 时是解锁鼠标右键。

无输入参数时是查询右键的锁定状态。

返回值 0：鼠标右键未锁定

返回值 1：鼠标右键已锁定

请记住加锁和解锁配套使用。

注意 lock\_mr 后，可以调用catch\_mr来捕捉鼠标右键按下和抬起，详见catch\_mr的用法。

```
>>> km.lock_mr('help')
用于锁定鼠标右键,锁定鼠标后物理点击右键不会有反应:
  没有参数输入是查询鼠标右键是否锁定.0:未锁定  1: 已锁定
  有参数时是设置鼠标右键是否锁定,lock_mr(1)锁定, lock_mr(0)解锁
  锁定后可以用right()查询右键状态或者catch_mr捕获状态
>>>
```

### lock\_ms1 鼠标侧键 1 屏蔽函数

Lock\_ms1(lock mouse side 1)函数用于屏蔽鼠标侧键 1 的物理输入，多用于确保软件侧键 1 控制不受鼠标物理侧键 1 的影响。

输入参数为 1 时是锁定鼠标侧键：

当鼠标侧键被锁定以后，物理上怎么点鼠标都没用。

输入参数为 0 时是解锁鼠标侧键。

无输入参数时是查询侧键的锁定状态。

返回值 0：鼠标侧键未锁定

返回值 1：鼠标侧键已锁定

请记住加锁和解锁配套使用。

注意lock\_ms1后，可以调用catch\_ms1来捕捉鼠标右键按下和抬起，详见catch\_ms1的用法。

```
>>> km.lock_ms1('help')
用于软件锁定鼠标侧键1,锁定鼠标后物理点击侧键1不会有反应:
  没有参数输入是查询鼠标侧键1是否锁定.0:未锁定  1: 已锁定
  有参数时是设置鼠标侧键1是否锁定,lock_ms1(1)锁定, lock_ms1(0)解锁
  锁定后可以用catch_ms1捕获侧键实际状态
>>>
```

## lock\_ms2 鼠标侧键 2 屏蔽函数

Lock\_ms2(lock mouse side 2)函数用于屏蔽鼠标侧键 2 的物理输入，多用于确保软件侧键 2 控制不受鼠标物理侧键2 的影响。

输入参数为 1 时是锁定鼠标侧键：

当鼠标侧键被锁定以后，物理上怎么点鼠标都没用。

输入参数为 0 时是解锁鼠标侧键。

无输入参数时是查询侧键的锁定状态。

返回值 0：鼠标侧键未锁定

返回值 1：鼠标侧键已锁定

请记住加锁和解锁配套使用。

注意lock\_ms2后，可以调用catch\_ms2来捕捉鼠标右键按下和抬起，详见catch\_ms2的用法。

```
>>> km.lock_ms2('help')
用于软件锁定鼠标侧键2, 锁定鼠标后物理点击侧键2不会有反应:
  没有参数输入是查询鼠标侧键2是否锁定. 0: 未锁定   1: 已锁定
  有参数时是设置鼠标侧键2是否锁定, lock_ms2(1) 锁定, lock_ms2(0) 解锁
  锁定后可以用 catch_ms2捕获侧键实际状态
>>>
```

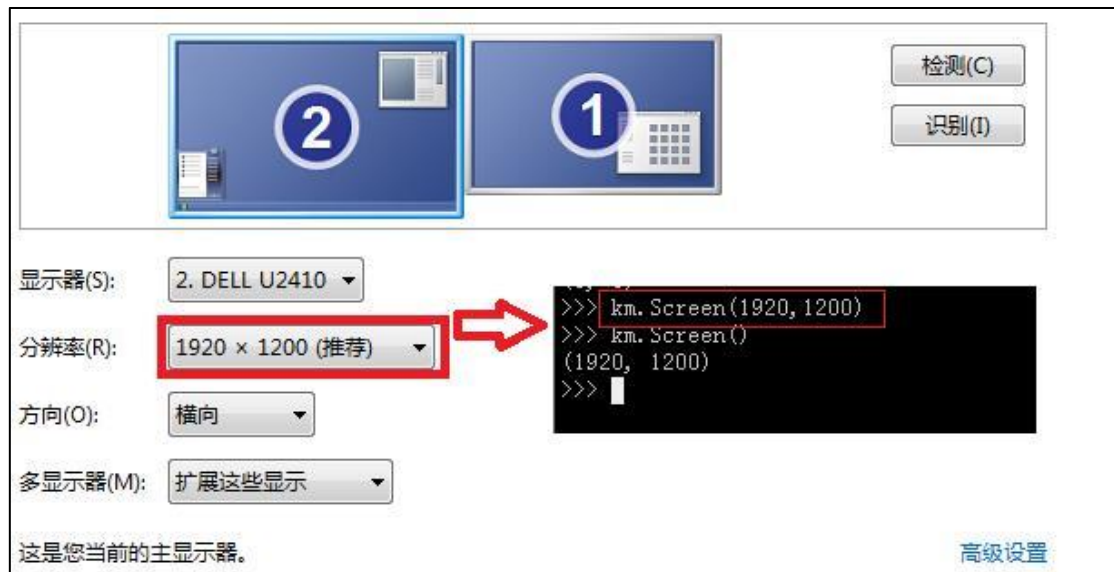
## Screen 鼠标精确控制范围限定函数

前面我们知道，电脑上接收的所有鼠标数据都是 kmbox 转发给 PC 的。原理上 Kmbox 是知道当前鼠标 X,Y 方向分别移动了多少。如果我们能记录下当前的 XY, 是不是就能知道当前鼠标在屏幕的哪个点呢？原理上是讲得通的。Kmbox内部的坐标统计也是基于这个思路。但是还有些情况会出现特例。例如：当你把鼠标移到屏幕左上角时，这时候你再继续往左上角移动。鼠标的指针是不会移动的。因为 window 认为你的鼠标已经在他的边界了，不可能更小了。但是此时鼠标底层的发码还是一直存在，往左移动，往上移动，但是实际表现是鼠标不会移动。如果这个时候还是统计底层数据，那么肯定有问题。为了解决这个问题，我们引入一个鼠标移动范围限制函数Screen。这个函数对应我们的鼠标能移动的最大范围。比如我们的显示器分辨率是1920X1200。那么我们可以限定死鼠标只能在这个范围内部移动。超出这个范围的移动就不算。这样就能实现屏幕坐标和kmbox的坐标一一对应。首先不给参数调用一下screen函数，发现返回值是（0,0），也就是没有对屏幕尺寸进行任何限制。

现在我们根据实际情况限定一下鼠标的移动范围：

比如显示器是 1920x1200. 那么直接将允许移动范围设置为 Screen(1920, 1200)



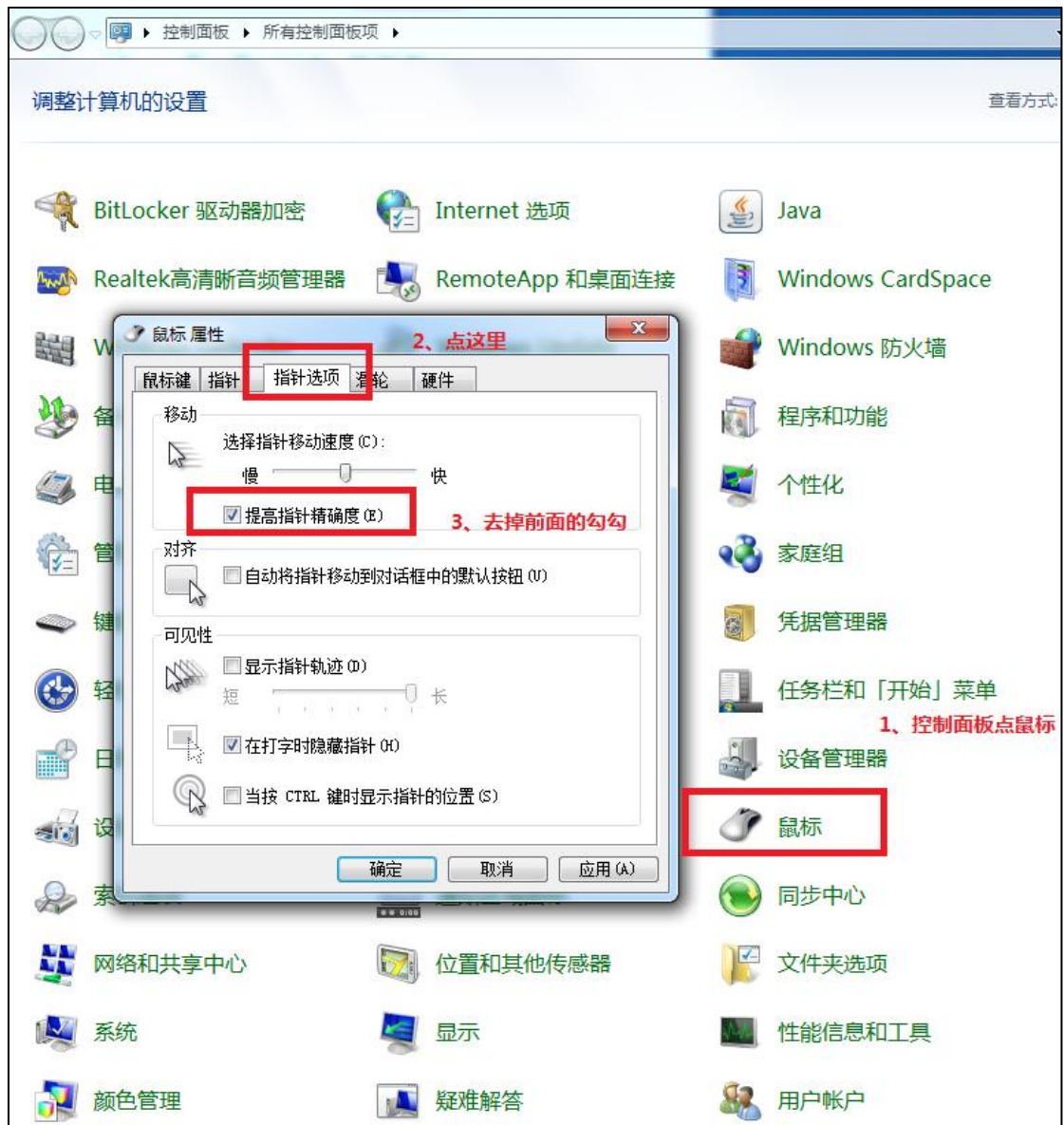


这时候kmbox内部就限定了鼠标的移动的范围了。但是这个时候并没有生效。因为 kmbox 之前不知道鼠标在屏幕的哪个位置，有没有越界。接下来需要调用一个校准函数 zero。

```
>>> km.Screen('help')
用于设置鼠标允许移动的最大区域。避免鼠标边界错误
Screen():没有参数时是查询当前设置的限制尺寸
Screen(1920,1080):有参数表示设置鼠标最大移动范围为:1920x1080
注意:当鼠标超过上面的范围时,虽然底层在发移动数据,但是鼠标已到达屏幕边缘
PC端不会有任何移动效果。在精确控制鼠标时会用到这个函数,避免鼠标统计坐标出错
>>>
```

### zero 鼠标精确控制归零函数

Zero 函数会使鼠标自动校准。即鼠标移动到屏幕左上角 (0,0) 处，并且后面会实时的记录当前鼠标的坐标点。可用于精确控制鼠标在屏幕上的移动。并且当鼠标超过 screen 函数定义的 X,Y 后会自动丢弃数据包（因为就算发了底层数据，鼠标在边界处也不会移动）。需要注意的是，请去掉windows的鼠标加速算法，不然鼠标不会按照底层鼠标的实际数据移动。Windows 为了提高鼠标的速度，采用了一定的加速算法，他会按照鼠标报文的加速度预测目标位置。我们为了精确控制，需要关闭这个功能，不然也会造成误差。去掉鼠标加速算法方法如下：



Zero 函数用法如下:

```
>>> km.zero('help')
鼠标坐标模式设置和查询
zero(0): 设置为相对模式坐标。坐标原点为当前位置。对应移动函数为move()
zero(1): 设置为绝对模式坐标。坐标原点在屏幕的左上角。对应移动函数为moveto()
zero(): 无参数是查询当前鼠标模式, 0: 相对模式 1: 绝对模式
>>>
```

### getpos 鼠标精确控制获取当前位置函数

getpos 函数是获取当前鼠标的物理位置。也就是 kmbox 内部统计的坐标。这个函数只有在 zero 调用后才有意义。他能实现 kmbox 坐标与屏幕坐标的一一对应。理论上在屏幕上任意选一个点, 无论怎样移动鼠标, 在这个点的物理坐标都是恒定的。Getpos 函数的返回值就



是鼠标当前点的物理坐标，第一个元素是 x 坐标，第二个元素是 y 坐标。截图如下：

```
>>>
>>>
>>>
>>> km.getpos()
[109, 24]
>>>
```

### moveto 鼠标精确移动函数

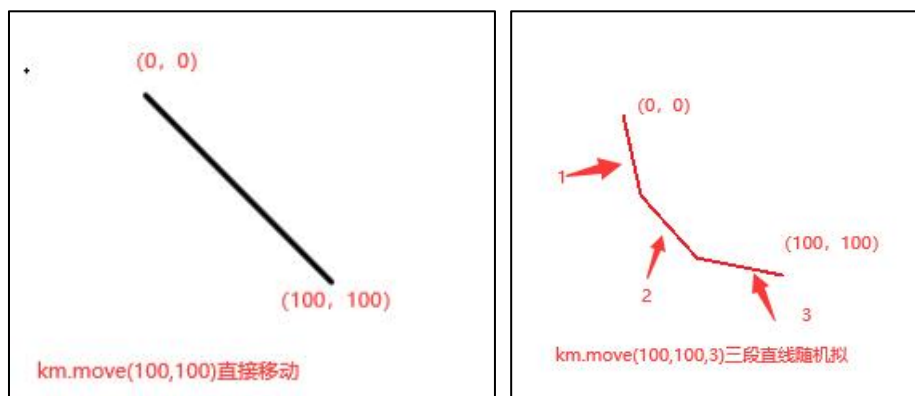
moveto 是将鼠标移动到绝对物理坐标(x, y)处。所以这个函数也是在执行zero函数后才可以使用。与 move 函数不同的是，moveto 的坐标原点是屏幕的左上角(0,0)处。而 move 函数的坐标原点是当前鼠标位置。Moveto能快速精确的将鼠标移动到指定位置。

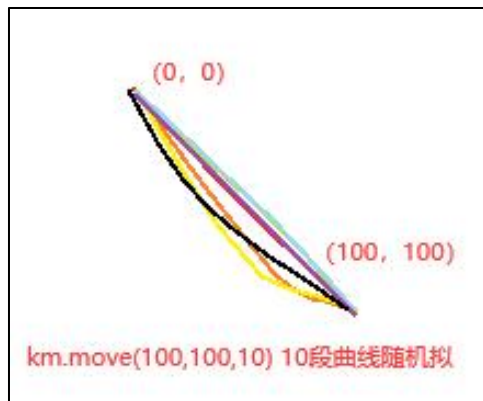
此函数用法如下：

```
>>> km.moveto('help')
用于鼠标的绝对移动，将鼠标移动到绝对坐标(x,y)处：
此函数需要先调用zero(1).且规定屏幕左上角为坐标原点(0,0)
推荐使用Screen(x,y)设定屏幕的分辨率大小,这样移动更准确
moveto(100,200)表示移动到绝对坐标(100,200)处
x,y的范围为0-32767，不存在负坐标
注意：请关闭windows鼠标加速算法
>>>
```

注意，一般这个函数是配合getpos使用。比如你想移动到某个坐标点。这个点的坐标可以通过 getpo 获取。然后调用moveto 就能分毫不差的移动到这个点。

以上是 moveto 函数的基本用法，moveto 函数如果只给两个参数，那么鼠标移动将在1ms内完成，没有中间过程。实际效果是从当前点阶跃移动到对应的目标点。这种效率最高，速度最快。如果你需要模拟人工轨迹移动，你可以个 moveto 函数再加上一个参数，这个参数是用来约束从起点到终点你期望用几条直线来拟合。由常识可知，第三个参数数字越大，拟合出来的曲线就会越平滑，如下图所示：分别用 1 条直线，3 条直线，和 10 条直线拟合从(0, 0)到(100, 100)的移动。

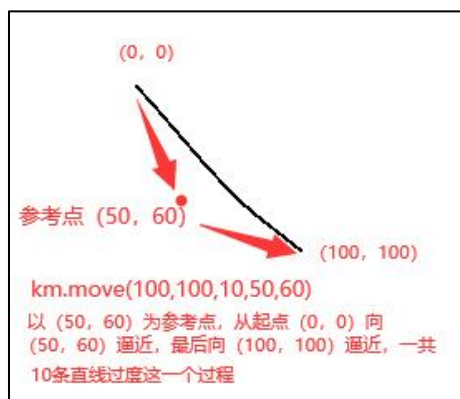




上图是鼠标轨迹，你可以发现，拟合点数越大，起点到终点的过度曲线就越平滑。所以，如果想控制鼠标模拟人操作，可以给 move 加上第三个参数。可以将阶跃状态改变成连续的状态。

另外你会发现，每次调用 move(100, 100, 10) 得到的鼠标轨迹是不一样的。因为过两个点的曲线有无数条，kmbox 内部会自动随机取点。保证平滑的由起点到终点。如果你不想让 kmbox 自动随机取点，而是想实际控制中间的过程曲线再给 moveto 函数增加两个参数，用来做参考逼近。

km.moveto(100, 100, 10, 50, 60)



其中的 (50, 60) 就是参考点坐标。kmbox 会自动按照参考点来拟合起点到终点。给定参考点后，kmbox 的轨迹就是唯一确定的。曲线走向怎样，扭曲程度怎样都是由这个参考点决定的。

### catch\_ml 鼠标左键捕获函数

catch\_ml(catch mouse left) 函数用于捕获鼠标左键的物理输入，多用于需要对左键进行特殊处理的情况。其用法和键盘的 catch\_kb 一样。下图是 catch\_ml 的使用方法

```
>>> km.catch_ml('help')
捕获鼠标左键：
    当调用lock_ml(1)后，鼠标左键被实时监测，且鼠标左键按下和弹起的消息
    不会发送给pc，需要自己做处理。catch_ml有两种模式，阻塞模式和非阻塞模式
    km.catch_ml():是不带参数的非阻塞模式。没有发生捕捉事件返回0.按下返回1，松开返回2
    非阻塞模式常用在快速轮询，km.catch_ml(0)也是非阻塞模式
    km.catch_ml(1):是带参数的阻塞模式。函数在没有捕捉到指定的按键前是不会返回的。
    所以阻塞模式建议在多线程中使用。不然当前线程会被阻塞
PS:一定要调用lock_ml(1)函数后才能激活使用catch_ml函数。
>>>
```

### catch\_mm 鼠标中键捕获函数

catch\_mm(catch mouse middle)函数用于捕获鼠标中键的物理输入，多用于需要对中键进行特殊处理的情况。其用法和键盘的 catch\_kb 一样。下图是 catch\_mm 的使用方法

```
>>> km.catch_mm('help')
捕获鼠标中键：
    当调用lock_mm(1)后，鼠标中键被实时监测，且鼠标中键按下和弹起的消息
    不会发送给pc，需要自己做处理。catch_mm有两种模式，阻塞模式和非阻塞模式
    km.catch_mm():是不带参数的非阻塞模式。没有发生捕捉事件返回0.按下返回1，松开返回2
    非阻塞模式常用在快速轮询，km.catch_mm(0)也是非阻塞模式
    km.catch_mm(1):是带参数的阻塞模式。函数在没有捕捉到指定的按键前是不会返回的。
    所以阻塞模式建议在多线程中使用。不然当前线程会被阻塞
PS:一定要调用lock_mm(1)函数后才能激活使用catch_mm函数。
>>> █
```

### catch\_mr 鼠标右键捕获函数

catch\_mr(catch mouse right)函数用于捕获鼠标右键的物理输入，多用于需要对右键进行特殊处理的情况。其用法和键盘的 catch\_kb 一样。下图是 catch\_mr 的使用方法

```
>>> km.catch_mr('help')
捕获鼠标右键：
    当调用lock_mr(1)后，鼠标右键被实时监测，且鼠标右键按下和弹起的消息
    不会发送给pc，需要自己做处理。catch_mr有两种模式，阻塞模式和非阻塞模式
    km.catch_mr():是不带参数的非阻塞模式。没有发生捕捉事件返回0.按下返回1，松开返回2
    非阻塞模式常用在快速轮询，km.catch_mr(0)也是非阻塞模式
    km.catch_mr(1):是带参数的阻塞模式。函数在没有捕捉到指定的按键前是不会返回的。
    所以阻塞模式建议在多线程中使用。不然当前线程会被阻塞
PS:一定要调用lock_mr(1)函数后才能激活使用catch_mr函数。
>>> █
```

### catch\_ms1 鼠标侧键 1 捕获函数

catch\_ms1(catch mouse side 1)函数用于捕获鼠标侧键 1 的物理输入，多用于需要对侧键 1 进行特殊处理的情况。其用法和键盘的 catch\_kb 一样。下图是 catch\_ms1 的使用方法

```
>>> km.catch_ms1('help')
捕获鼠标侧键1：
    当调用lock_ms1(1)后，鼠标侧键被实时监测，且鼠标侧键按下和弹起的消息
    不会发送给pc，需要自己做处理。catch_ms1有两种模式，阻塞模式和非阻塞模式
    km.catch_ms1():是不带参数的非阻塞模式。没有发生捕捉事件返回0.按下返回1，松开返回2
    非阻塞模式常用在快速轮询，km.catch_ms1(0)也是非阻塞模式
    km.catch_ms1(1):是带参数的阻塞模式。函数在没有捕捉到指定的按键前是不会返回的。
    所以阻塞模式建议在多线程中使用。不然当前线程会被阻塞
PS:一定要调用lock_ms1(1)函数后才能激活使用catch_ms1函数。
>>> █
```

## catch\_ms2 鼠标侧键 2 捕获函数

catch\_ms1(catch mouse side 2)函数用于捕获鼠标侧键 2 的物理输入，多用于需要对侧键 2 进行特殊处理的情况。其用法和键盘的 catch\_kb 一样。下图是 catch\_ms2 的使用方法

```
>>> km.catch_ms2('help')
捕获鼠标侧键2:
  当调用lock_ms2(1)后，鼠标侧键被实时监测,且鼠标侧键按下和弹起的消息
  不会发送给pc,需要自己做处理。catch_ms2有两种模式，阻塞模式和非阻塞模式
  km.catch_ms2():是不带参数的非阻塞模式。没有发生捕捉事件返回0.按下返回1，松开返回2
  非阻塞模式常用在快速轮询,km.catch_ms2(0)也是非阻塞模式
  km.catch_ms2(1):是带参数的阻塞模式。函数在没有捕捉到指定的按键前是不会返回的。
  所以阻塞模式建议在多线程中使用。不然当前线程会被阻塞
  PS:一定要调用lock_ms2(1)函数后才能激活使用catch_ms2函数。
>>>
```

以上，就是目前鼠标宏支持的函数。如果你有更多新需求或者意见也可以与我联系。

## 6. 其他功能函数与模块

在前面的章节。你应该已经掌握了 kmbox 的键盘宏和鼠标宏包含哪些函数了。本节讲讲其 他函数。可能会用到。

### debug 调试函数

```
>>> km.debug('help')
打印调试数据:(开发人员专用)
  当需要查看底层传输数据时可以打开这个调试。
  输入参数，0: 关闭所有调试打印
  输入参数，1: 打开鼠标底层报文数据
  输入参数，2: 打开键盘底层报文数据
>>>
```

### delay 延迟函数

```
>>> km.delay('help')
delay(...):延迟函数
  1个参数是精确延时，例如延迟10ms,km.delay(10)
  2个参数是随机延时，例如随机延迟100ms-200ms,km.delay(100,200)
>>>
```

### reboot 开发板重启函数

```
>>> km.reboot('help')
软复位重启板卡
  km.reboot()会在3秒后软件复位重启板卡
>>>
```



## version 版本查询函数

```
>>> km.version('help')
返回当前kmbox的版本号:
  最新固件请去官网:www.kmbox.top
  或者论坛      :www.clion.top
  也欢迎提交bug或者意见!
>>> km.version()
kmbox:  2.0.0 Aug 31 2020 21:49:51
>>>
```

## rgb\_free 释放三色 LED 的 GPIO

```
>>> km.rgb_free('help')
释放三色LED占用的3个GPIO:
  当你需要复用这个3个IO时, 你需要调用这个函数将3个GPIO从kmbox释放出来
  不然你在操作IO, kmbox也在操作IO. 那么最后就乱套了
  释放IO调用:km.rgb_free()
>>>
```

## mode 模式切换函数

Kmbox 一共有7种传输模式, 分别是 USB 模式、蓝牙模式、kvm 模式。在多主机中可以使用此函数切换不同的工作模式。

```
>>> km.mode('help')
kmbox传输模式设置:
  kmbox一共有7种传输协议, 分别对应如下:
  0:usb+bt+kvm模式 (等同于模式1+模式2+模式3)
  1:USB模式 (模式1, 有且仅有USB传输--kmbox的PC端口传输)
  2:蓝牙模式 (模式2, 有且仅有蓝牙传输)
  3:kvm模式 (模式3, 有且仅有kvm传输, 需要kvm扩展卡)
  4:usb+kvm模式 (等同于模式1+模式3)
  5:bt+kvm模式 (等同于模式2+模式3)
  6:usb+bt模式(等同于模式1+模式2)
使用方法: km.mode(1)->切换到USB模式
          km.mode() ->查询当前模式
ps:此函数的用途一般是多系统切换, 例如你可以USB连接A电脑, 蓝牙连接B电脑
    kvm接C电脑, 调用此函数可在多个系统中切换键鼠连接的设备
```

## rng 随机数产生函数

```
>>> km.rng('help')
产生随机函数:
  无参数是产生一个随机数,km.rng()
  2个参数是产生一个指定范围内的随机数, 例如需要一个100-200之间的随机数,km.rng(100,200)
>>>
```

## freq 键盘鼠标报告频率超频设置函数

km.freq() 函数用于强制设置键盘鼠标的报告频率。默认情况下无需设置。默认值为200, 即键盘鼠标每秒钟上传200此报告。如果不给参数是查询当前报告频率, 给参数就是强制设置报告频率, 其取值范围为【100, 1000】. 报告频率是USB设备的固有属性。如果强制超频可能会带来设备的不稳定性。常规键盘和鼠标的报告频率是 125Hz. 如果你嫌鼠标卡顿较慢可以强制提高报告频率。freq 的用法详见 km.freq('help') 函数说明。

## MAC 获取机器码函数

机器码是板卡身份证, 每个板子的机器码具有唯一性。机器码多用于脚本绑定机器。实现注册授权功能, 防止脚本在非授权的设备上使用。什么加密算法由脚本作者自己定义。

MAC 函数的返回值是组元，一共六个元素。每个板子的返回值各不相同。

```
>>> km.MAC()
(124, 158, 189, 193, 30, 80)
>>>
```

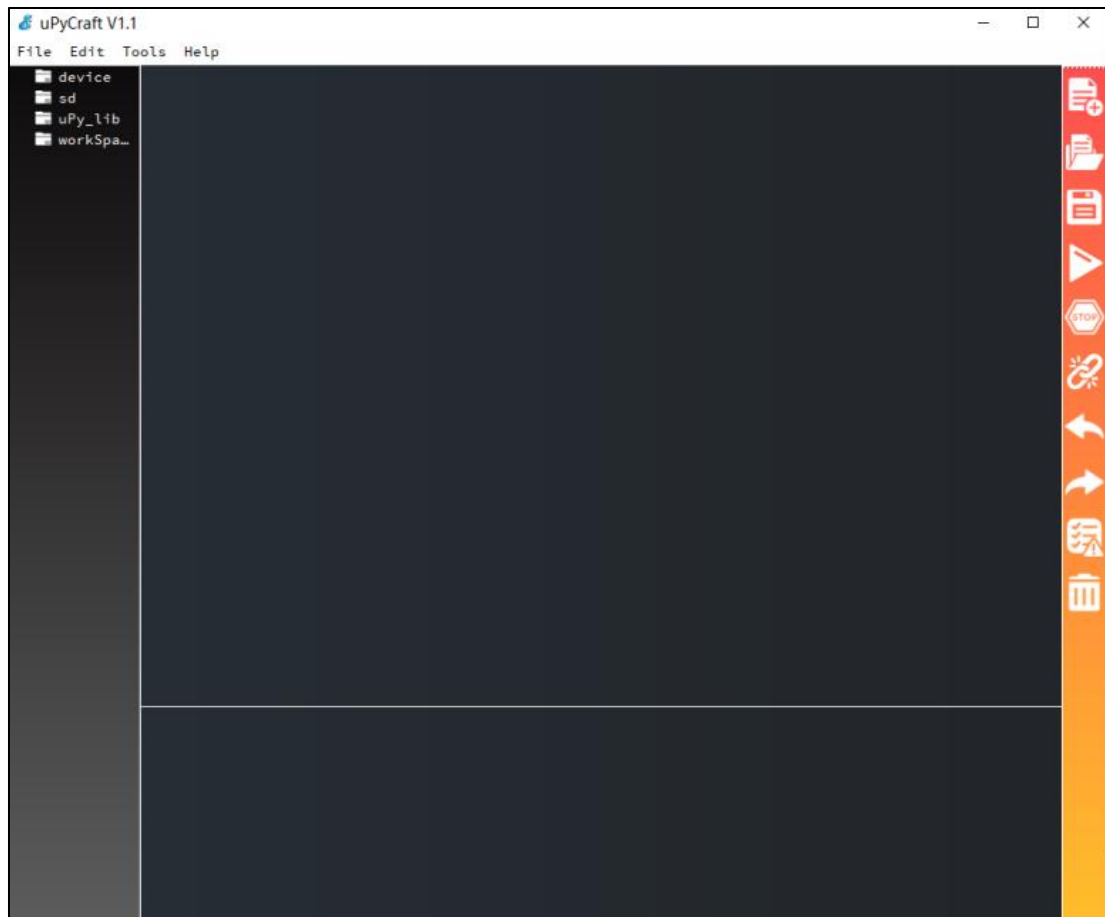
以上是 km 模块提供的 API 函数。键鼠操作用 km 模块应该基本够用。但是 kmbox 绝不是仅仅只能操作键盘鼠标那么简单。键盘鼠标只是其中一部分功能。kmbox 还有好多模块呢！你可以输入 help(‘modules’) 查看板卡默认包含的模块：

```
>>> help('modules')
__main__      esp32          os              ujson
_boot         flashbdev      random          uos
_owewire      framebuf      re              upip
_thread       gc            select          upip_utarfile
_webrepl      hashlib      socket          urandom
apal06        heapq        ssl             ure
array         host          struct          uselect
binascii      inisetup     sys             usocket
bt            io            time            ussl
btree         json          ubinascii       ustruct
builtins      km            ucollections    utime
cmath         machine      ucryptolib      utimeq
collections   math         uctypes         uwebsocket
device        micropython  uerrno          uzlib
dht           neopixel     uhashlib        webrepl
ds18x20       network      uhashlib        webrepl_setup
errno         ntptime      uheapq          websocket_helper
esp           onewire      uio             zlib
Plus any modules on the filesystem
>>>
```

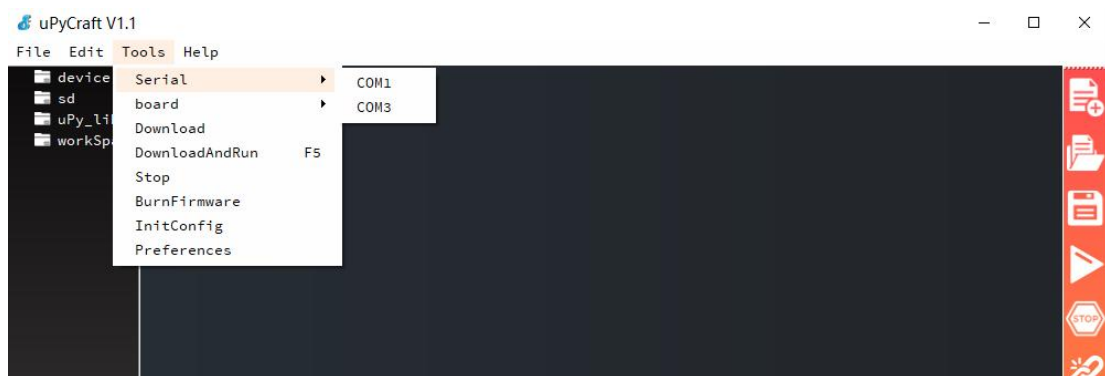
其中 km 仅仅是众多模块中的一个。这里就不一个一个的讲。有了前面 km 模块的学习。其他模块也是以此类推。你可以用 kmbox 驱动硬件，也可以用它来写软件算法。DIY 各种电子产品都非常合适。

## 四、kmbox 实战篇

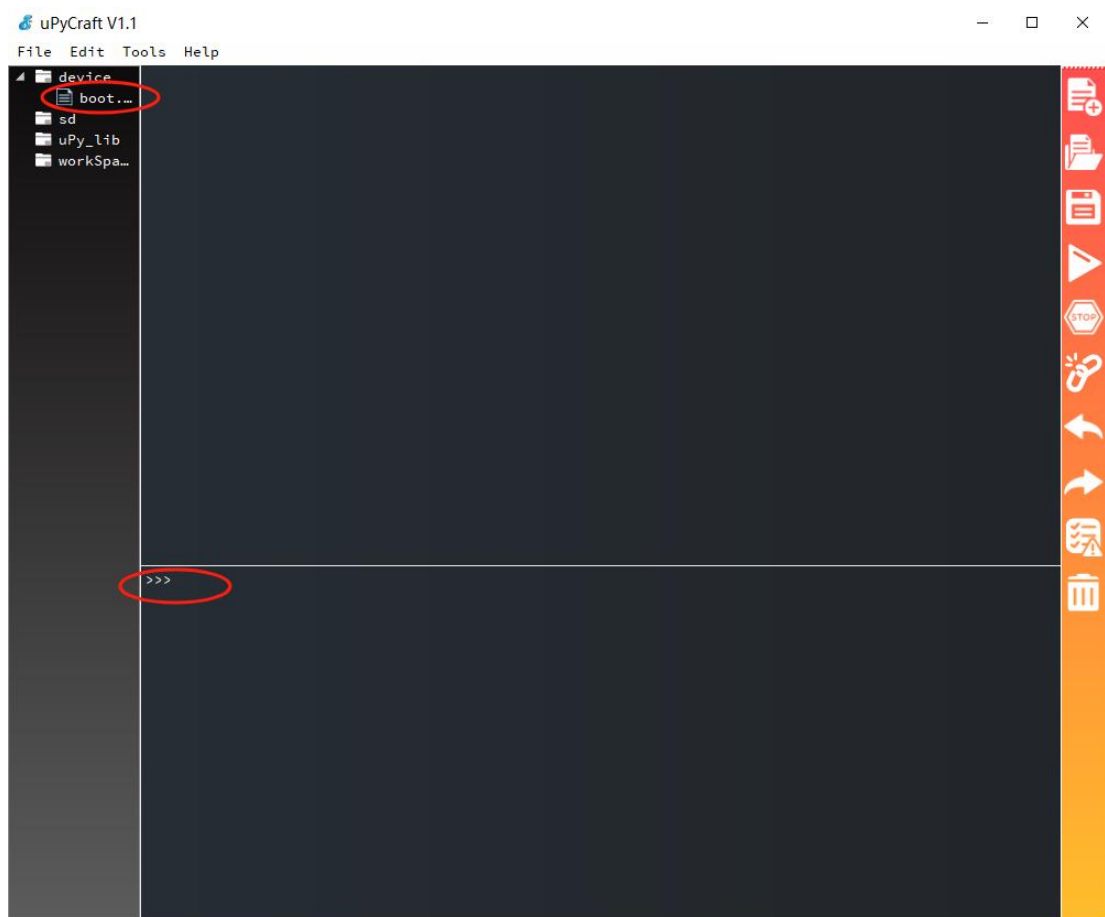
在前面的章节中，通过 putty 可以轻松的尝试每个函数的使用方法，但是这距离实际应用的脚本还有一点距离。本章就学习一下如何写脚本，将前面的 API 串联起来。从理论到实践过度。写脚本很简单，就是前面 API 的调用与组合，外加条件判断、循环等。这些都是根据你要实现的功能来确定的。从前面可以知道，python 是在线释义的。你敲完代码他就立即执行。如果代码成百上千行，用 putty 写可能会让你发狂。本节引进一个新工具 uPyCraft.exe。这是一个专门的代码编辑器和下载器。他也可以用于调试。双击 uPyCraft.exe 会出现如下应用界面：



打开后需要连接开发板：依次 tools --> serial-->COMxx 连接到开发板

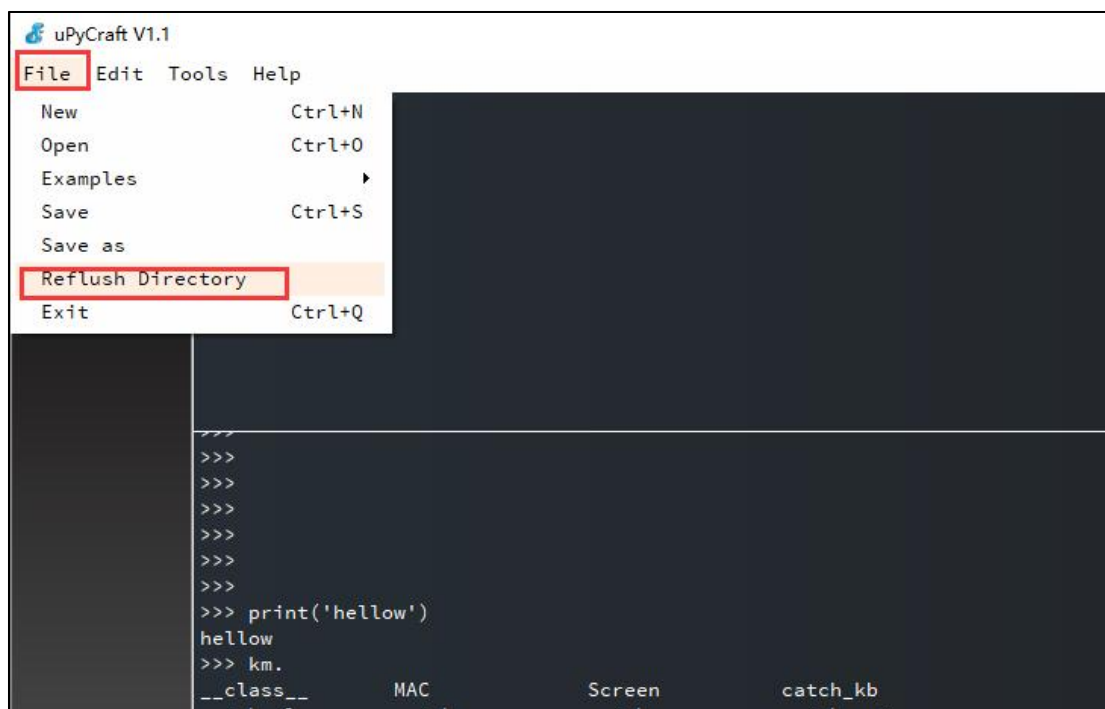


如果连接失败请检测串口驱动。连接后是这样的。这里其实就是一个 putty 的控制台。



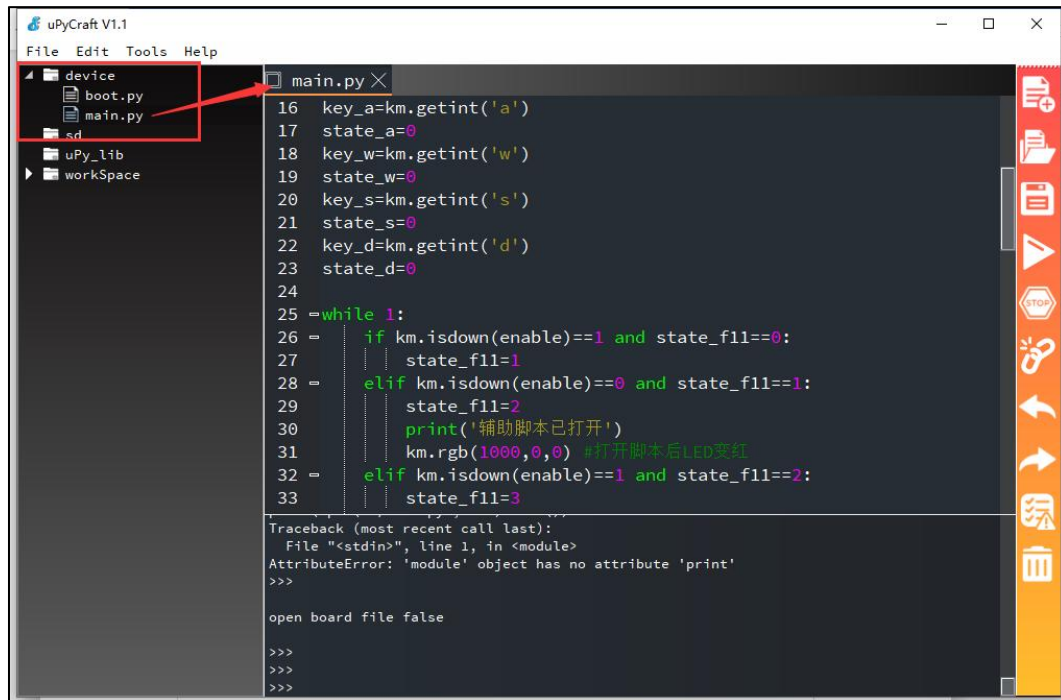
## 1. 读取 kmbox 内部的脚本

依次点击 File -->Reflush Directory 即可读取板卡内部的脚本（**copy 脚本无法读取**）



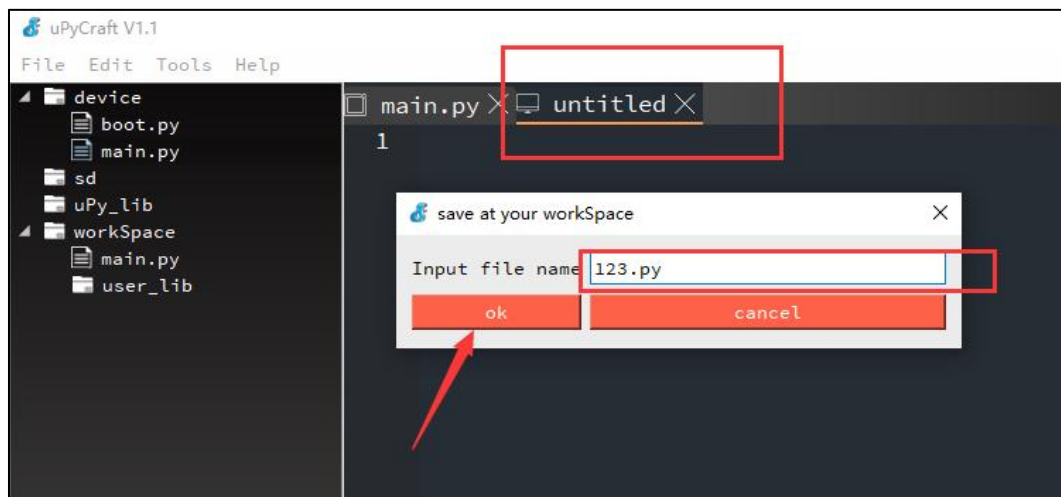


你可以在Device文件夹下看到板卡里面的脚本文件，双击对应的文件即可查看里面的内容。修改，保存，重命名都是可以的。



## 2. 新建脚本

File -->New 即可新建一个文档（快捷键是 Ctrl+N）.新建的文件没有名字，按 ctrl+s 保存文件时给他取个名字，点击OK此时就可以将你需要的脚本功能写到这个文件里面了。



## 3. 调试脚本

例如在 123.py 中写了个简单脚本。每隔1秒打印一次 hellow。

调试这个脚本很简单。有以下几种方式：

方式一：

内存中运行（推荐），在下面的控制台按ctrl+e。Kmbbox 会进入复制模式。

直接复制123.py 文件里的内容（ctrl+c），然后在控制台右键粘贴（**不要用快捷键 ctrl+v**）。

粘贴完毕后在控制台按 ctrl+d，那么刚刚复制的代码就会在 kmbbox内部运行。

Ctrl+d 后脚本就已经开始运行了，如下图所示

```
===    print('hellow ')
===    km.delay(1000)
===
===
hellow
hellow
hellow
hellow
hellow
```

如果需要结束运，在控制台按ctrl+c。

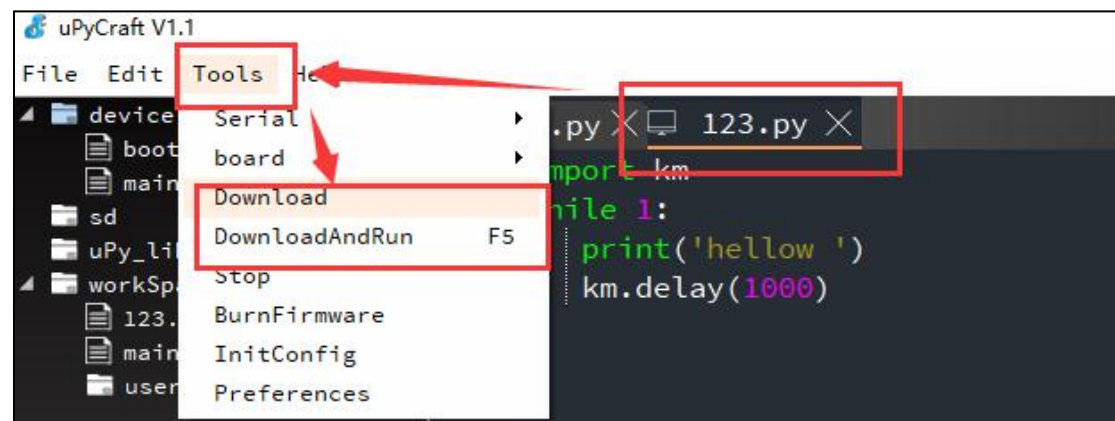
方式二：

直接点右边的 RUN，这个方法不太推荐。因为这个会下载到板卡里面去。Flash 都是有写寿命的，一般10万次就不行了。建议用方式1，脚本OK确认无误了再烧写到板卡内部。

## 4. 烧写脚本

如果调试脚本没有问题了，我们希望永久保存脚本，就需要将脚本烧写到开发板里面。例如前面的123.py确认功能无误后需要烧写到开发板。

依次点击 123.py--->Tools--->Download 即可烧录到开发板



其中downloadAndRun 是下载后立即运行该脚本。如果不需要运行只用下载就行。烧写脚本不支持mpy类型的文件。

## 5. 鼠标宏的简单应用（吃鸡压枪宏制作）

本章就用前面的知识做一个简单有实用价值的吃鸡压枪宏。其实在前面的章节中已经提到 过压枪是怎么实现的。基本思路是左键按下，鼠标自动下移来实现。

但是你要问我具体下移多少？我也只能很遗憾的告诉你，我也不知道。知道这个参数的估计只有写吃鸡的程序猿了。但是我们可以测试呀。那今天就拿荒野行动做个简单的牛刀小试吧。手把手教你怎样用 kmbox 写一个吃鸡压枪外挂。

详细步骤见这个视频：

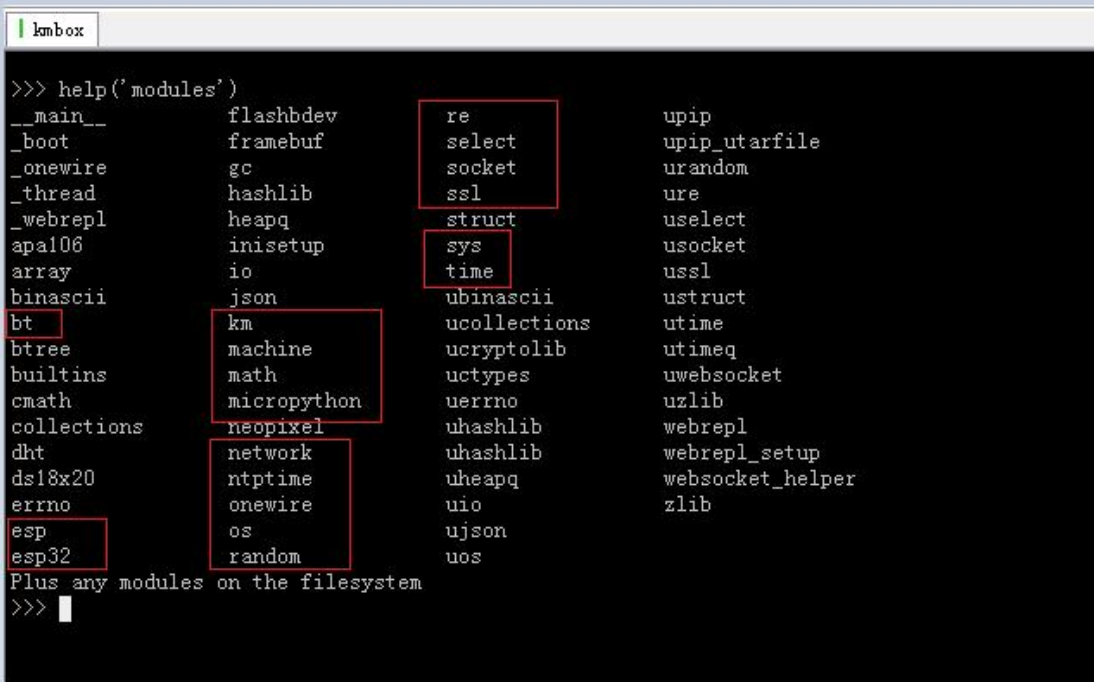
<https://www.bilibili.com/video/BV1Ya4y1Y7ku/>

## 6. 如何使用第三方库

在上面的基础教程中你已经能在 Repl 环境下测试和验证简单的逻辑代码。但是真正的项目 可能需要很多 py 文件，他们各自相互依赖。那么怎么在 kmbox 里面调用其他的库或者模块 呢？

## 7. 默认模块

要引入其他 module，首先看看已经包含哪些 module. 你可以在repl界面里输入 `help( 'module' )`指令查看目前kmbox内部默认包含哪些模块：



```
>>> help('modules')
__main__      flashbdev      re            upip
__boot__      framebuf      select        upip_utarfile
__onewire     gc            socket        urandom
__thread      hashlib      ssl           ure
__webrepl     heapq        struct        uselect
apa106        inisetup     sys           usocket
array         io           time          ussl
binascii      json         ubinascii     ustruct
bt            km            ucollections  utime
btree         machine      ucryptolib    utimeq
builtins      math         uctypes       uwebsocket
cmath         micropython  uerrno        uzlib
collections   neopixel     uhashlib      webrepl
dht           network      uhashlib      webrepl_setup
ds18x20       ntptime      uheapq        websocket_helper
errno         onewire      uio           zlib
esp           os            ujson
esp32         random       uos
Plus any modules on the filesystem
>>>
```

从上面我们可以看到常用的km模块，sys 模块，time 模块，machine 模块，os 模块等等，也就是说上面的这些模块名就能直接使用 `import xxxx` 包含到我们的脚本里面。

按tab可以查看开机默认加载模块，如下图所示：

```
>>>
Pin          SPI          __dict__      gc
os           spi           sys           vfs
KMMMain      TFT           bdev          sysfont
km           uart_speed  km_trace_type km_trace_enable
bt_enable    km_mode      km_freq       km_vertime
tft
>>> |
```

如果如果你想使用 machine（硬件控制）模块。你可以直接使用 ” import machine” 再按 tab 后你可以看到 machine 模块已经包好到我们的环境中了。

```
>>> import machine
>>>
Pin          SPI          __dict__      gc
machine      os           spi           sys
vfs          KMMMain     TFT           bdev
sysfont      km           uart_speed    km_trace_type
km_trace_enable
km_freq      km_vertime  tft           uos
>>>
```

Machine 模块里面包含一些硬件操作，例如 ADC，DAC，PWM，I2C，GPIO 等操作。

```
>>> machine.
__class__    __name__      ADC           DAC
DEEPSLEEP    DEEPSLEEP_RESET EXT0_WAKE
EXT1_WAKE    HARD_RESET    I2C           PIN_WAKE
PWM          PWRON_RESET   Pin           RTC
SDCard       SLEEP         SOFT_RESET    SPI
Signal       TIMER_WAKE    TOUCHPAD_WAKE Timer
TouchPad     UART          ULP_WAKE      WDT
WDT_RESET    deepsleep     disable_irq    enable_irq
freq         idle          lightsleep     mem16
mem32        mem8          reset          reset_cause
sleep        time_pulse_us unique_id      wake_reason
>>> machine. |
```

如果你确实要引入一些不在默认库里的东西。引入其他module有以下几种方法。

kmbox自动画个五角星，写脚本前首先要想好用鼠标怎么画五角星。思路如下：

第一步：与 x 轴方向夹角  $72^\circ$ ，画一条长度为 L 的线段。

第二步：向右偏转  $144^\circ$ ，画一条长度为 L 的线段。

第三步：向右偏转  $144^\circ$ ，画一条长度为 L 的线段。

第四步：向右偏转  $144^\circ$ ，画一条长度为 L 的线段。

第五步：向右偏转  $144^\circ$ ，画一条长度为 L 的线段。

以上 5 个步骤，完毕后就得到了一个五角星。那如何用鼠标宏实现，请看下面的代码：

```

1  """
2  鼠标左键单击绘制五角星
3  mp_turtle是海龟绘图模块。移植标准的turtle模块
4  """
5  from mp_turtle import *
6
7  while 1:
8      if m.left()==1:#如果鼠标左键按下
9          home()      #将当前鼠标点设置为坐标原点
10         left(72)     #龟头72度 准备画图
11         for i in range(5): #循环5次画边
12             forward(100)  #绘制长度为100的边
13             right(144)    #右转144度
14             m.delay(100)  #延时100ms, 避免绘制过快您看不清过程
15
16         while m.left()==1: #此时五角星已经绘制完成, 等待鼠标左键松开避免不停绘制
17             m.delay(100)  #空等延时
18
19
20
21

```

上图中鼠标左键按下，设置当前坐标为原点，然后向左偏移 72 度。在for循环中5次画长度为100的线和偏转，即得到了五角星。

脚本实际运行效果视频如下：[点我看视频](#)

这里用到的home、Left、forward、right函数是在mp\_turtle模块中定义的。mp\_turtle中又会调用 km 模块。相当于重新封装一遍km模块。

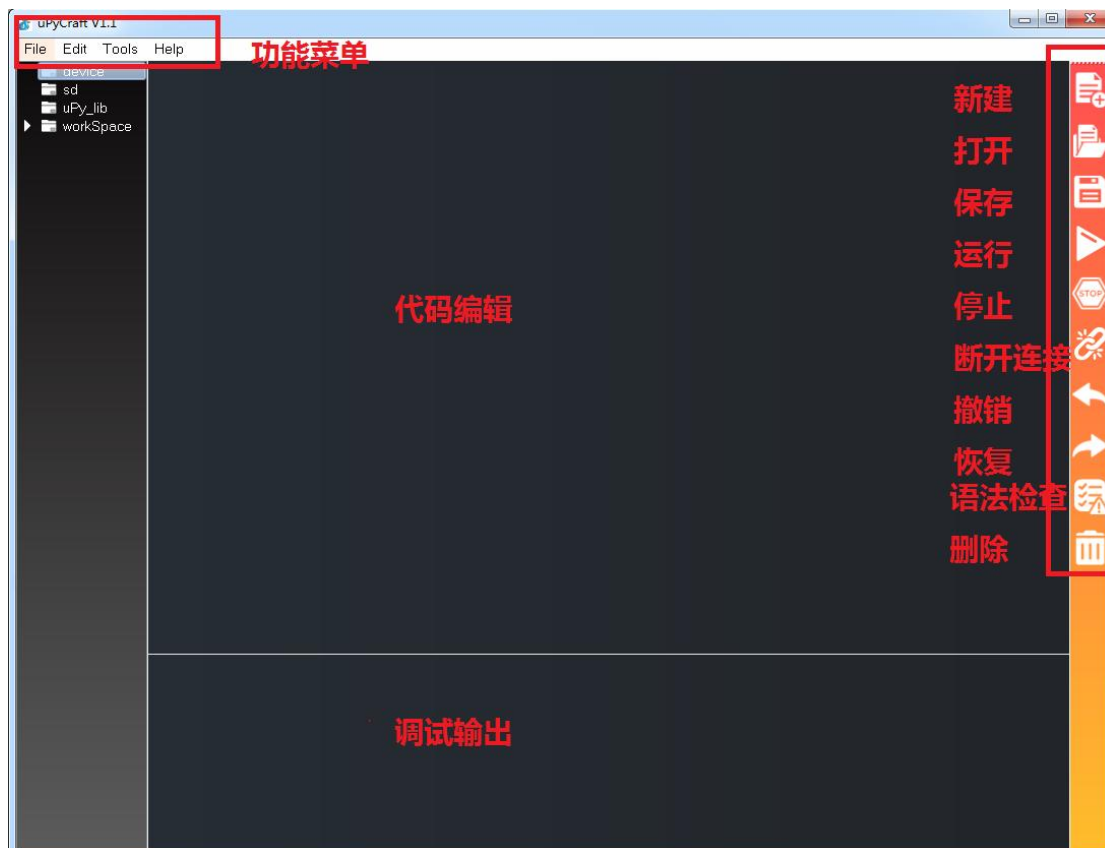
## 8. 文件存储的方式（入门级）

文件存储就是将要引入的模块源码保存到kmbox里面。然后你就能直接import使用这个模块了。可以使用以下工具把py文件保存到kmbox内部。

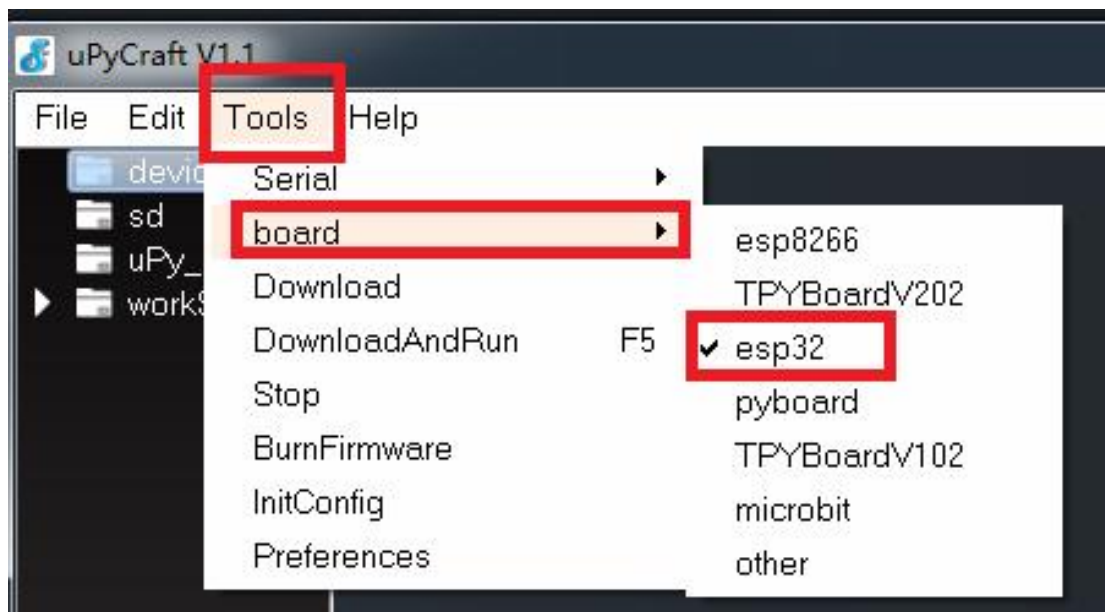
### 1. Upycraft

这是一个小巧的代码编辑和调试器，专门为micropython设计使用。

下载请访问：<https://github.com/SDRRADIO001/Keyboard-Mouse-Box>

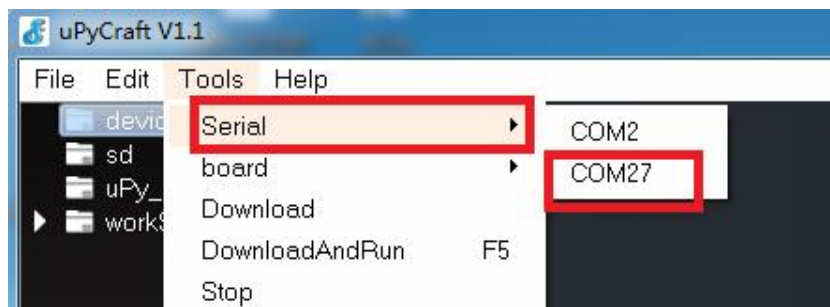


我们以 kmbox 为例，介绍一下怎么使用以及如何传输文件。打开软件后在 tool 菜单下将 board 标签里选择板子芯片型号为 esp32 如下图：

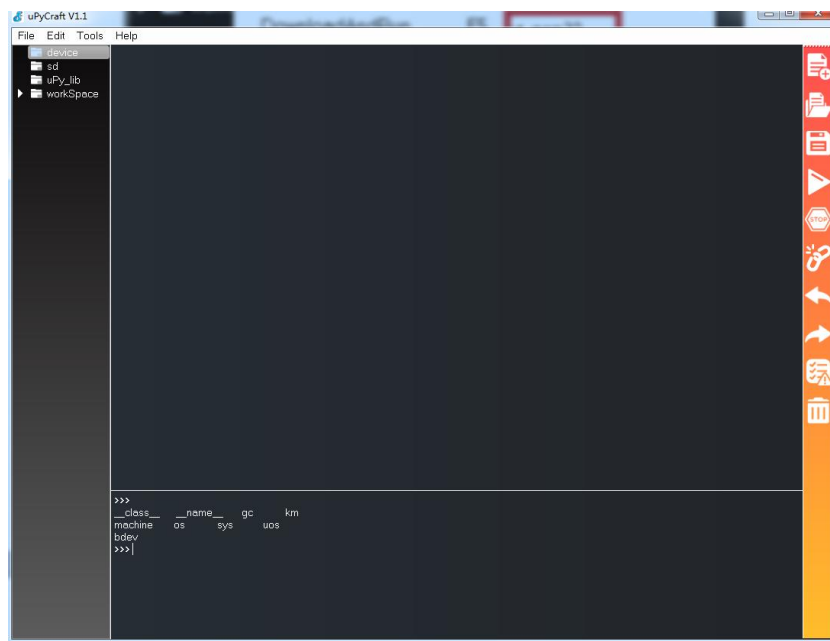




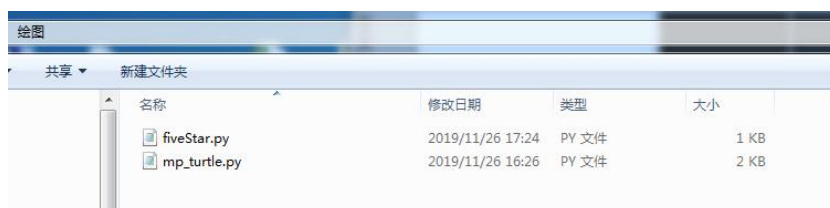
然后将串口号选择 kmbox 实际的串口号即可。



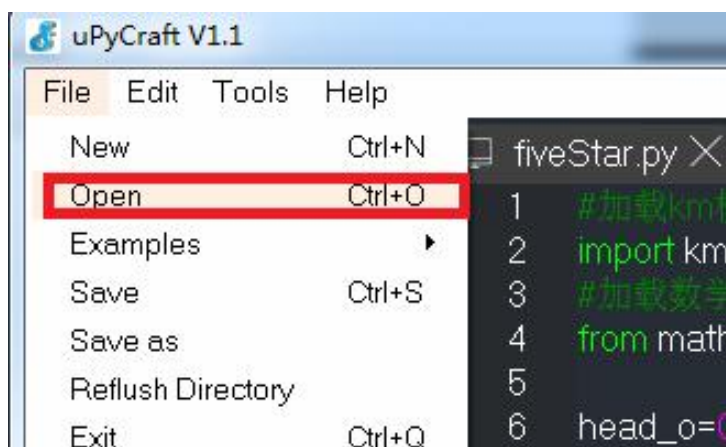
设置完毕后你就能在调试窗口进行之前的 repl 操作了（和前面的 putty 一样）。



下面介绍如何导入第三方的文件或者模块，例如你在网上找到一个绘图功能的模块。



这个绘图模块包括两个文件。我们依次打开这两个文件。



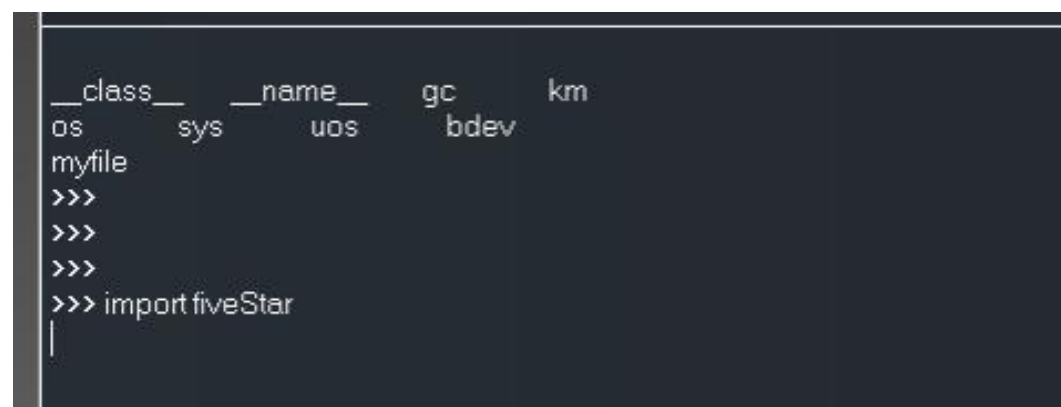
打开后如下图所示：

然后下载到开发板Tools ->Download 。注意，download 是下载，只保存文件，不会运行文件，下面的 downloadAndRun 是保存后就运行文件。从源码中可以看到 fiveStar.py 文件依赖是 mp\_turtle. 由于现在板卡上还没有 mp\_turtle.py 这个文件，所以运行肯定会出错。



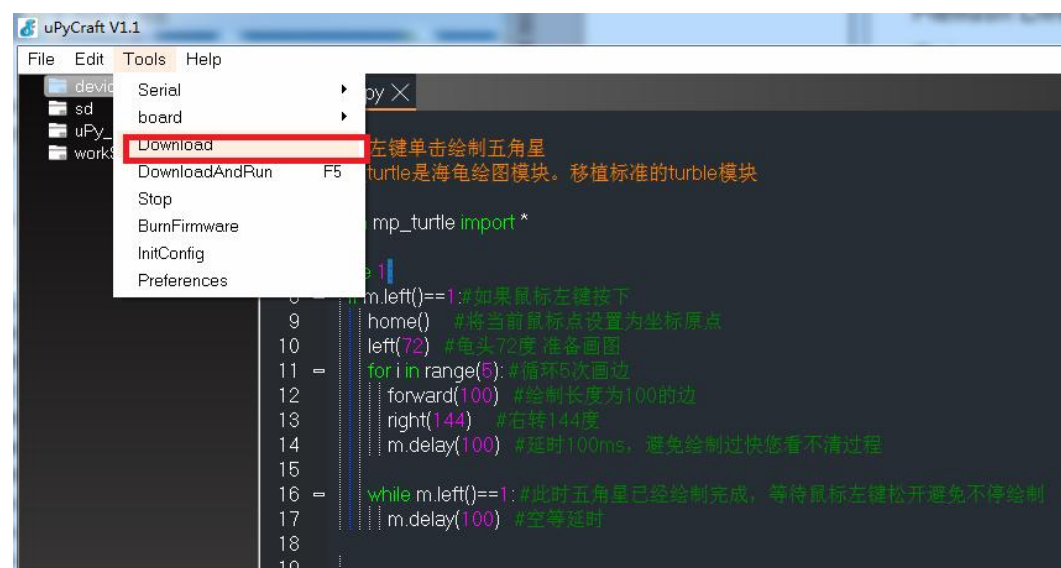
The screenshot shows the uPyCraft V1.1 software interface. On the left is a file explorer with folders 'device', 'sd', 'uPy\_lib', and 'workSpace'. The main editor window is titled 'fiveStar.py' and contains the following Python code:

```
1  """
2  鼠标左键单击绘制五角星
3  mp_turtle是海龟绘图模块。移植标准的turtle模块
4  """
5  from mp_turtle import *
6
7  = while 1:
8  = if m.left()==1: #如果鼠标左键按下
9      home() #将当前鼠标点设置为坐标原点
10     left(72) #龟头72度 准备画图
11     = for i in range(5): #循环5次画边
12         forward(100) #绘制长度为100的边
13         right(144) #右转144度
14         m.delay(100) #延时100ms，避免绘制过快您看不清过程
15
16     = while m.left()==1: #此时五角星已经绘制完成，等待鼠标左键松开避免不停绘制
17         m.delay(100) #空等延时
18
19
20
21
```



The screenshot shows a terminal window with the following commands and output:

```
__class__  __name__  gc      km
os          sys        uos      bdev
myfile
>>>
>>>
>>>
>>> import fiveStar
|
```



The screenshot shows the uPyCraft V1.1 software interface with the 'Tools' menu open. The 'Download' option is highlighted with a red rectangle. The menu options are: Serial, board, Download, DownloadAndRun, Stop, BurnFirmware, InitConfig, and Preferences. The background editor window shows the same Python code as the previous screenshot.

下载完后控制台会出现下面信息。

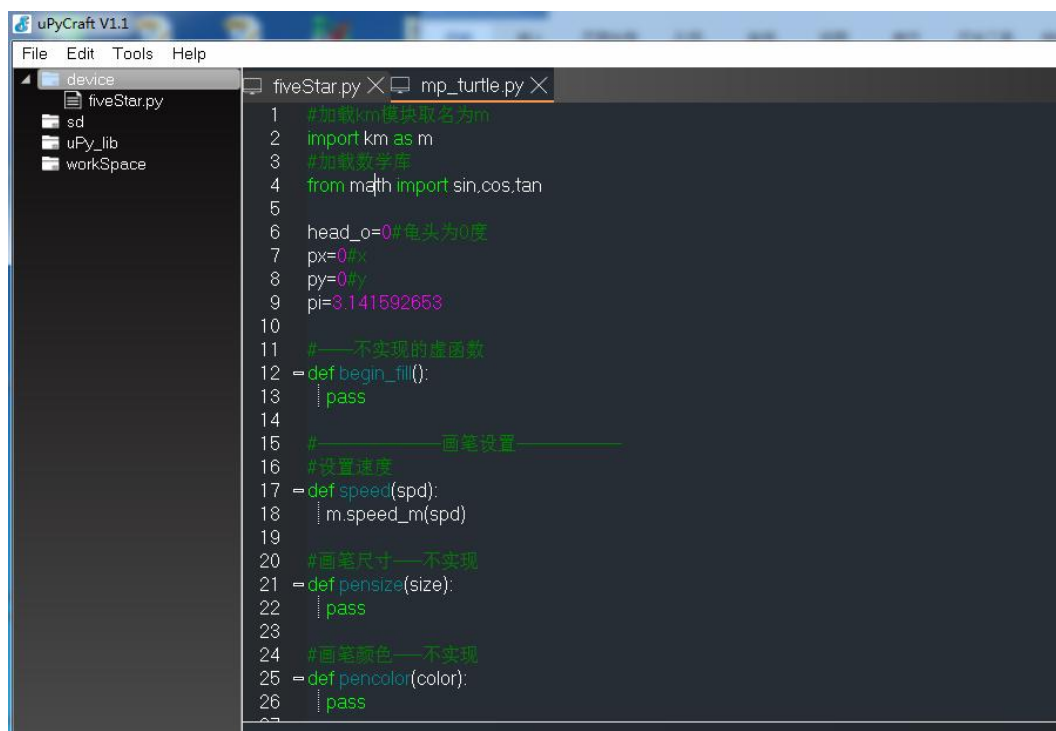
```
>>>

>>>

Ready to download this file,please wait!
.....
download ok
```

同理，打开 mp\_turtle.py 文件烧写到开发板。

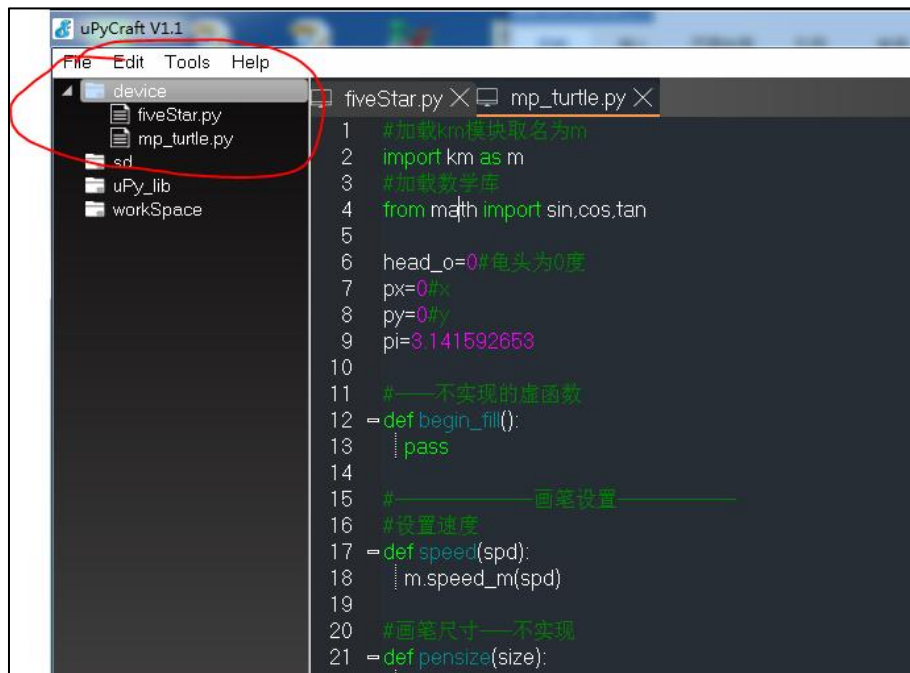
Ps:通过 mp\_turtle.py 的源文件可以看到，其底层依赖是km模块和math模块。而这两个模块是板卡支持的模块。所以这个脚本所有的依赖关系都满足。那么他肯定能运行。



烧写完毕后你就能看到在左边看到这两个文件。你也可以在控制台输入 os.listdir() 查看当前文件系统下挂载哪些文件。如下图所示：

```
>>>
>>>
>>>
>>>
>>>
>>> os.listdir()
['fiveStar.py', 'mp_turtle.py']
>>>
```

到此这两个文件已经在kmbox内部。你就可以直接调用fiveStar.py这个文件实现的功能了。

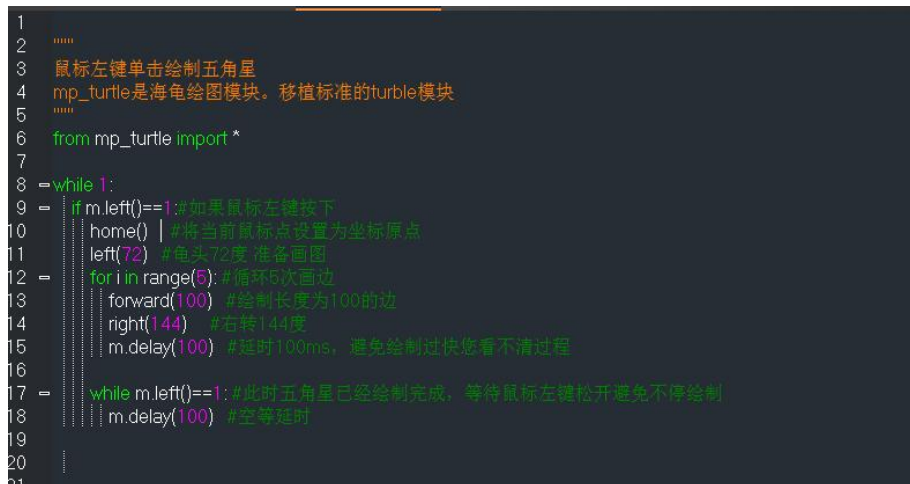


如果这个时候调用 `help('modules')` 不会出现刚刚传进来的两个模块。`help('modules')` 里显示的是通过源码方式直接编译进来的模块。而上面通过文件传输的方式是用户自定义的模块。

### 使用传进来的第三方模块：

接下来可以直接在控制台输入 `import fiveStar` 即可使用这个模块。

注意，这里运行后怎么像死机了，其实不是死机，请看 `fiveStar.py` 的源码



在 `while 1` 里等待你的鼠标左键按下，然后画五角星。鼠标需要插到Kmbox上才知道鼠标是否按下。如果鼠标接电脑上，电脑是不会告诉kmbox鼠标已按下了。

如果你想停止脚本，直接在控制台按 `ctr+c` 即可。

中断后可以继续使用 `import fiveStar` 和查看里面的成员函数。如下图

```
>>>
__class__  __name__  gc      km
os          sys      uos      bdev
myfile      fiveStar
>>> fiveStar.
__class__  __name__  __file__  cos
home       left      pi        right     cos
sin        tan      m        forward
head_o     px        py        begin_fill
speed      pensize  pencolor  pendown
penup      backward goto      setx
sety       setheading heading   position
>>> fiveStar|
```

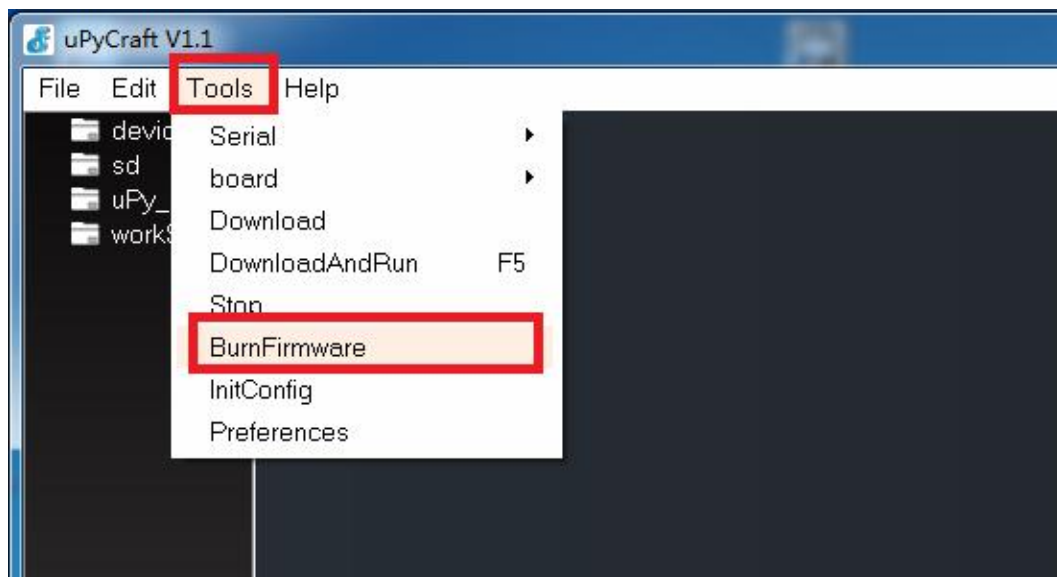
只要有源码，解决完依赖问题，最终都能在kmbox内部运行。通常烧录到板子里面的py文件是已经调试好了的。如果是开发阶段，建议在内存里运行，需要发布时才往板子里面烧。

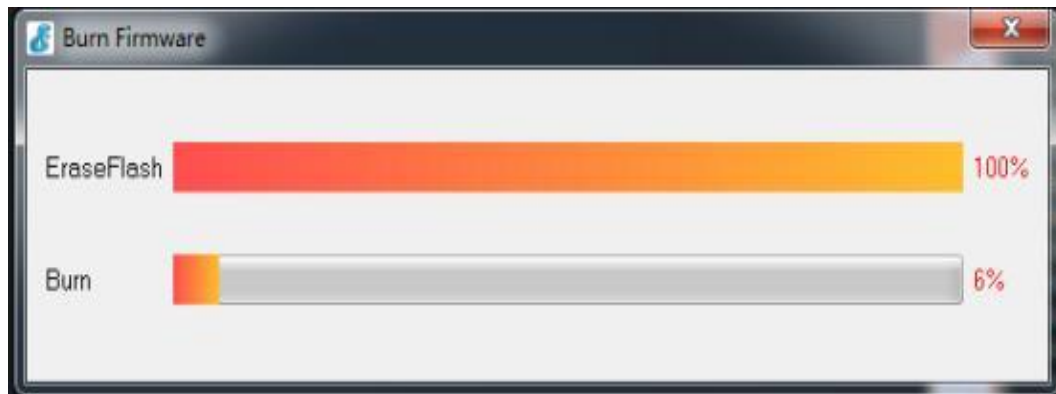
## 五、固件升级（板砖修复）

在前面教程中，描述了如何使用脚本，如何自己扩展脚本。如果万一烧录的脚本有问题，运行不正常，可以采用刷机方式恢复。Kmbox的刷机一共有两种方式，刷机完后一定要断电重启开发板（插拔USB）。

第一种方式需要工具：uPyCraft，打开 tools->BurnFirmware

出现以下这个界面，在 choose 那里选择kmbox 的固件，点击 OK 即可。

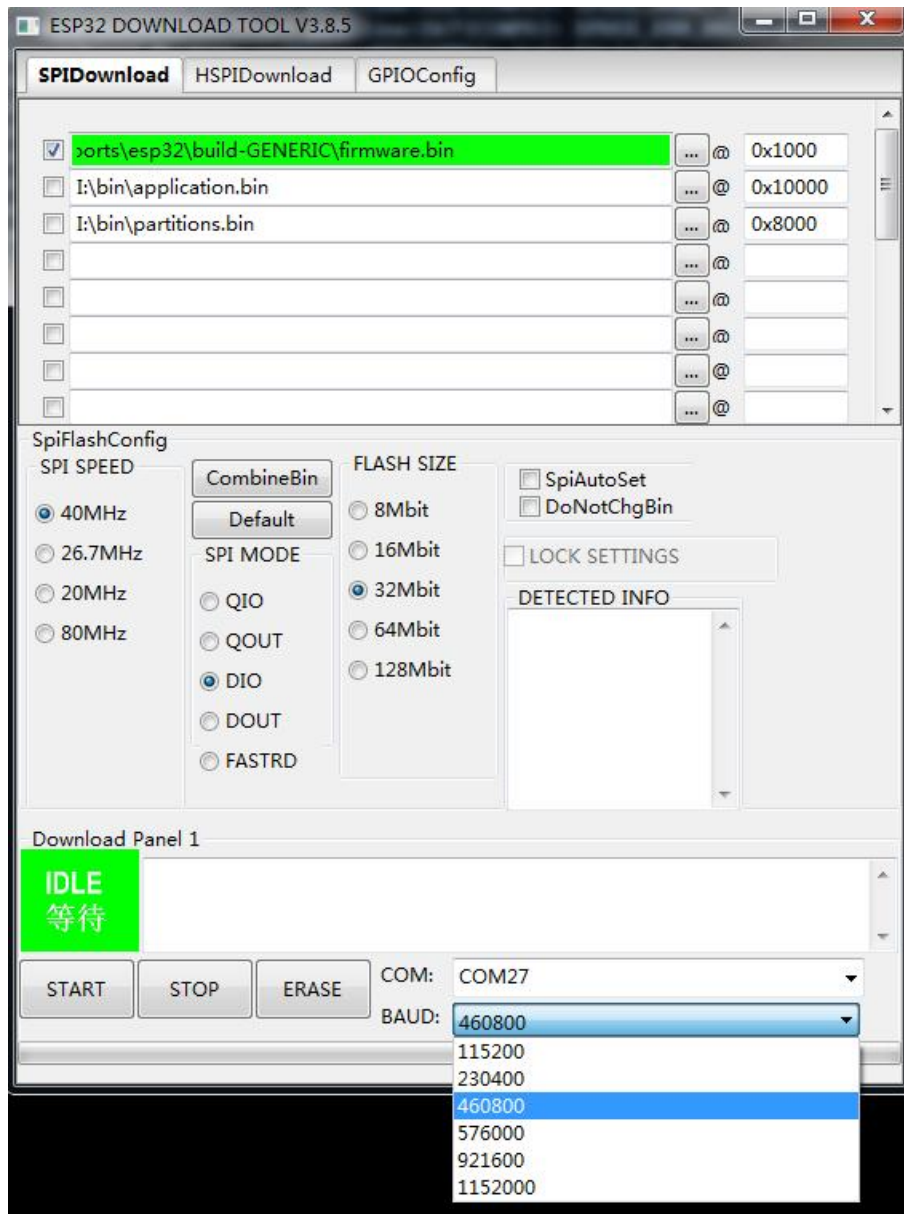




等待两个进度条走完。断电重启，KMB0X板卡就恢复最初状态。

第二种方式是 `flash_download_tool_v3.8.5.exe`，这是乐鑫官网的下载工具。他可用于批量烧录。





## 六、Kmbox 高级教程

### 1. 如何让脚本上电后自动运行

很多时候写好了脚本，我们希望脚本保存到 kmbox 内部，只要一上电 kmbox 就能自动运行。而不是搞个控制台输入指令来调用。其实在前面的《[如何使用第三方库](#)》章节中已经有相关的介绍。如果想要脚本一开机就运行。第一将脚本下载到 kmbox，第二将脚本设置为开机启动。在《[如何使用第三方库](#)》中，介绍了如何下载脚本。Kmbox的开机启动流程是固定的。他会首先运行boot.py脚本，然后运行main.py脚本。通常boot.py里面一般是系统的一些初始化配置参数。所以，如果你想让你的脚本开机就运行，将要运行的脚本文件名改成main.py或者boot.py就行。建议用main.py这个名字，boot.py后面可能用于其他前置配置。

## 2. 多线程代码优化

在前面的教程中，咱们大多是使用一个 `while True` 的大循环，然后在里面写检测逻辑。当逻辑量很小，执行耗时很少时。基本上可以不用考虑效率问题。但是有时候需要考虑效率。下面举个例子。学习一下如何优化脚本，提高执行效率。

例如：需要一个自动打怪脚本，该脚本需要有两个功能。

- 1) 如果按下 A 键，就自动释放 1、2、3、4、5 这 5 个技能
- 2) 每隔 10 秒自动喊话一次

按照前面学习到的很容易想到判断 A 键按下用 `isdown`，丢技能就用 `press`，每隔 10 秒就用 `delay`。很容易将代码写成这样。

```
while True:
    if km.isdown('a'):#如果 a 按下
        km.press('1')#丢 1 技能
        km.press('2')#丢 2 技能
        km.press('3')#丢 3 技能
        km.press('4')#丢 4 技能
        km.press('5')#丢 5 技能
    print('我要每隔 10 秒喊话一次')
    km.delay(10000)#延迟 10 秒
```

运行上面的代码，每隔10秒会出现一条打印。但是按a键却几乎不太可能会出现12345。如下图所示：

```
Hi3559AV100 | kmbox
paste mode: Ctrl-C to cancel, Ctrl-D to finish
===
while True:
===
    if km.isdown('a'):#如果a按下
===
        km.press('1')#丢1技能
===
        km.press('2')#丢2技能
===
        km.press('3')#丢3技能
===
        km.press('4')#丢4技能
===
        km.press('5')#丢5技能
===

    print('我要每隔10秒喊话一次')
===
    km.delay(10000)#延迟10秒
===

我要每隔10秒喊话一次
我要每隔10秒喊话一次
我要每隔10秒喊话一次
```

其实是因为 `km.delay(1000)` 这条指令。因为代码是一行一行的运行。`km.delay(10000)` 这条指令从开始运行到返回需要 10s 的时间。在这 10 秒内，CPU 啥都不干，仅仅等待这 10 秒结束，然后再运行 `km.isdown('a')` 这条指令。所以只有你碰巧在他执行 `isdown('a')` 时按下 a 键，kmbox 才会执行 12345 这几条指令。

下面谈谈如何优化代码。如果去掉这个 `delay`。你会发现这个主循环一秒钟可以达到几十万次。那么就相当于每秒检测 `isdown` 几十万次。根据奈奎斯特采样定理可知，抽样频率大于 2 倍以上的就能保证抽样信息不被丢失。如果主循环一秒钟运行 1 万次。意味每秒按 5000 次数据都不会丢失。实际情况是你一秒钟按不了 50 次按键。所以理论上，只要主循环能每秒执行 100 次基本上不会丢失按键。所以要确保 a 键按下不丢失。我们应该尽可能的提高 `isdown` 的调用次数，使它大于 100 以上即可。所以这里介绍第一种方式，短轮询。

```

import time #用于时间统计
last_time=0 #记录喊话的时间
while True:
    if km.isdown('a'):#如果 a 下
        km.press('1')#丢 1 技能
        km.press('2')#丢 2 技能
        km.press('3')#丢 3 技能
        km.press('4')#丢 4 技能
        km.press('5')#丢 5 技能
    if time.time()-last_time>=10000:
        print('我要每隔 10 秒喊话一次')
        last_time= time.time()#刷新上次喊话的时间
    #km.delay(10000)#延迟 10 秒

```

对比一下前面的代码，上面这段代码的效率是前面的几万倍。这段代码没有浪费 CPU 时间。把延迟 10 秒的操作采用 if 语句来快速判断。当时间到达时执行打印。以上代码每秒至少运行 1 万次以上。不会出现检测不到 a 按下的情况。所以，这时候你看见，不仅你按下 a 它能立马执行 12345. 而且每隔 10 秒会打印一次数据。但是上面这段代码还是会有问题。问题就出在它运行速度太快。会出现你不期望的现象。这个你们自己去板子里面运行一下就知道结果了。当你找到问题后你就能想办法解决。运行太快就想办法让他不那么快运行就行。实现同一个功能不同人有不同的方法。没有最好，只要能达到需求就行。

### 3. 多线程使用

以上一节的自动打怪脚本为例。如何用多线程的方法实现。需要一个自动打怪脚本，该脚本需要有两个功能：

- 一：如果按下 A 键，就自动释放 1、2、3、4、5 这 5 个技能
- 二：每隔 10 秒自动喊话一次。

从逻辑上讲，每隔 10 秒喊话和按下 A 就释放12345是没有任何关系的。他们可以看成两个独立的東西。可以写一个函数 A，让他检测 A 有没有按下。一个函数 B 让他自动喊话，然后这两个函数“同时运行”。这里就是现在要讲的多线程了。

虽然在逻辑上 A,B 两个函数是同时运行的。但是 CPU 只有一个。任何时候 A 在运行必然 B 不运行（多核 CPU 除外）。看上去的同时运行实际上是串行AB交替运行。只是他们交替的速度太快，感觉不出来罢了。下面就是多线程的例子：

```

'''
kmbox 支持多线程操作，示例如下：
需要快速处理的放在线程 A 里面。
比较耗时的操作放在线程 B 里面。
A 和 B 逻辑上是独立的。他们是同时运行的 '''
import _thread #引入多线程模块
import time    #引入时间模块

```

```

keyval_a=km.getint('a') #a
km.mask(keyval_a,2) #屏蔽监测 a 并加入捕获队列 通过 catch_kb

#A 线程---快速响应任务
def thread_A():
    while 1:
        retVal=km.catch_kb() #非阻塞捕获按键数据
        if retVal==keyval_a:#捕获到 1 次 a 按下
            print('a 按下...执行 a 按下后的逻辑,例如一键 N 技能。')

#B 线程---耗时任务
def thread_B(): #B 线程函数
    while 1: #B 线程死循环
        print('B 线程运行中...cputick=',time.ticks_ms())
        time.sleep(1.5) #1.5 秒休眠
    _thread.start_new_thread(thread_A, ()) #启动 A 线程
    _thread.start_new_thread(thread_B, ()) #启动 B 线程

```

可以发现不管在 B 里面怎么延迟,都不会影响到 A 函数的检测。A 的运行十分流畅。多线程的用法远远不止这些。还有线程间的同步,互斥等。这里就不一一介绍了  
PS: 以上代码没有写线程退出,请你们自己完善。

## 4. 脚本加密与授权

前面的章节中,我们的脚本都是明文的形式。也就是只要拿到 py 文件。就能无限制的修改和使用。但是很多时候,写的脚本不想就这么轻易公开给人使用。那么可以让脚本加密,再把这个加密的文件给用户。这样就不用担心源码泄露了。而且还可以授权用户,按时间收费或者次数收费都行。

举个简单的例子:假设这是个一锤子买卖。你实现某个功能,卖给用户。采用密码验证的方式。用户从你 这里买密钥。密钥正确就给用,不正确就不给用。例如明文源码如下 (test.py):

```

import km
IsAuthor=0 #是否授权初始化为 0 ---授权条件可以自己定义,例如密钥,机器码,
0 时间等。这里只做简单演示
def author( key):
    global IsAuthor
    if key=='123456': #设置脚本密钥为 123456
        IsAuthor=1 #密钥正确允许允许 run 函数
        print('授权码正确 ')
    else:
        print('授权码错误! 请联系脚本作者: xxx@xx.com')
def run():
    if IsAuthor==0:#密钥不正确

```

```

print('对不起您还未授权，请调用 author 输入注册码')
return
while 1:
    print('脚本已授权正在运行中...')
    km.delay(1000)

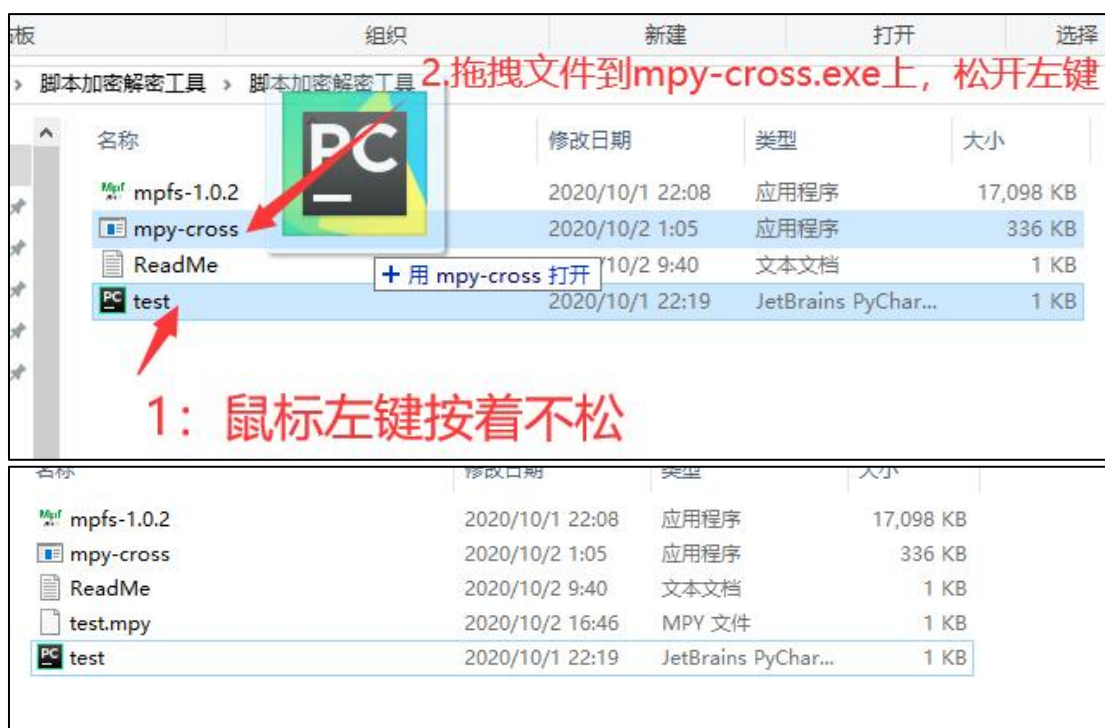
```

假设脚本的密钥字符串123456。第一个函数（author）是授权使用，如果授权成功将IsAuthor 标志置位。在运行 run 函数时会检测是否已经授权。不授权就不运行。为了保护作者的劳动成果，我们需要加密，使用的工具是 mpy-cross。环境为 windows。至于加密方式可以自己定，加密后除了作者本人知道外没人知道。现在分别从脚本作者和用户的角度讲解如何加密和使用。

## 5. 脚本作者加密脚本

方法一：直接拖拽需要加密的文件（推荐）

编译好mpy\_cross。直接将要加密的脚本拖到mpy-cross.exe 上，松开鼠标左键即可。在 exe 所在的文件夹内会自动生成 mpy 文件：



方法二：编译 mpy-cross，并在 python 下使用命令生成

第一步：加密需要用到的工具是 **mpycross**（python 环境自行搭建）—注意版本号是 1.11。其他版本可能无法运行。

```

C:\Users\hw\MPY>python
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

```

安装执行这个命令：pip install mpy-cross==1.11 安装过程如下图所示：



```
C:\Users\hw\MPY>pip install mpy-cross==1.11
Defaulting to user installation because normal site-packages is not writeable
Collecting mpy-cross==1.11
  Downloading mpy_cross-1.11-py2.py3-none-win_amd64.whl (151 kB)
    | 151 kB 5.8 kB/s
Installing collected packages: mpy-cross
Successfully installed mpy-cross-1.11
```

安装完毕后就可以将明文脚本 py 转换为密文 mpy 文件。

执行如下命令：`python -m mpy_cross test.py` -----黄色部分 test.py 是要转换的明文文件。

执行此命令后 test.py 文件会自动生成一个 test.mpy 的文件。这个 test.mpy 就是密文文件。这个密文文件你就可以发布给你的客户。你可以对比看看 test.py 和 test.mpy 两个文件。

test.mpy 文件更小，其效率比 test.py 高。而且 test.mpy 是不可见的。也不用担心源码泄露问题了。

**PS:脚本加密实际是将源码转换为机器码，所以 mpy 文件的体积比原始文件小，而且执行效率也比 py 文件高。**

## 6. 用户使用加密文件

用户使用加密文件（mpy）和使用普通的明文文件（py）一样。可以认为 mpy 就是等价于 py。只是用户不知道这个 mpy 文件里的内容罢了。使用 mpy 文件首先需要烧写到 kmbox 内部，Import 后即可调用里面的函数。Upycraft 这个软件无法传输 mpy 文件（因为作者最开始就没考虑 mpy 文件）。这里介绍一个新工具 Mpfs，它的作用就是将任意类型的文件下载到 kmbox 内部。你可以把它看作一个下载器，双击运行后如下图所示：

```

Mpfs C:\Users\hw\MPY\mpfs-1.0.2.exe
Documented commands (type help <topic>):
=====
EOF close get lexecfile ls mput open quit runfile
cat exec help llis md mpyc put repl view
cd execfile lcd lpwd mget mrm pwd rm
Undocumented commands:
=====
c e ef l ef o q r rf v

can input help ls or other command if you don't know how to use it.
looking for all port...
serial name : USB-SERIAL CH340 (COM26) : COM26
input ' open COM26 ' and enter connect your board.

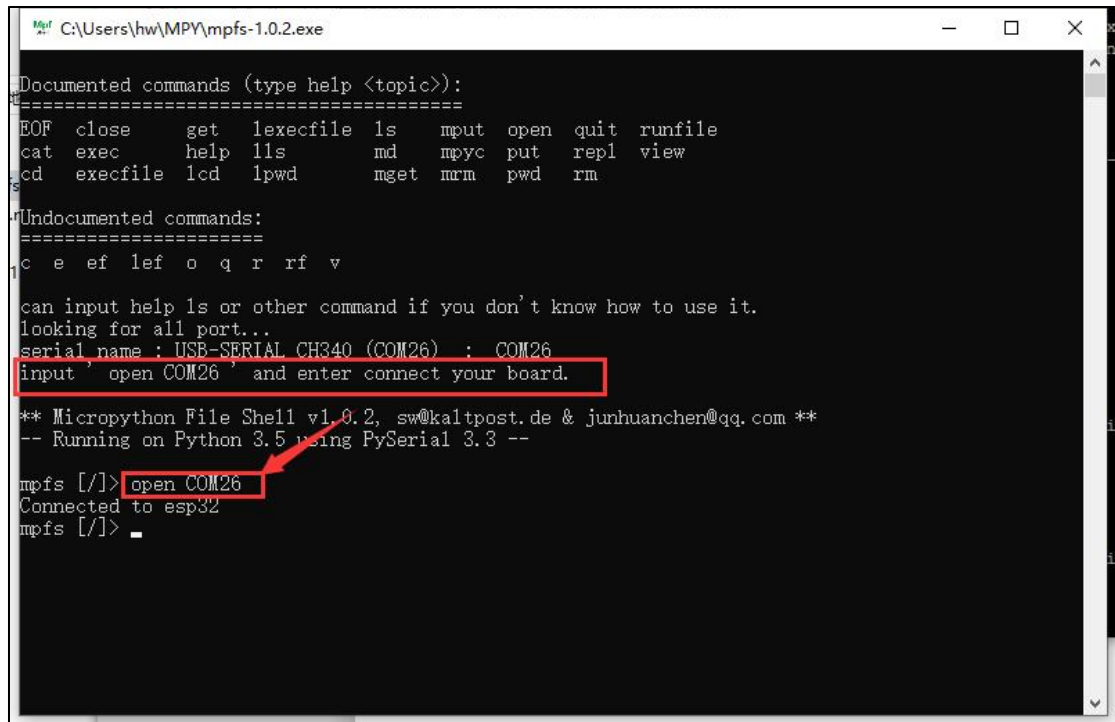
** Micropython File Shell v1.0.2, sw@kaltpost.de & junhuanchen@qq.com **
-- Running on Python 3.5 using PySerial 3.3 --

mpfs [/]>

```

使用这个软件需要板子在 repl 模式下。如果你不知道怎样进入 repl 模式，直接刷机就行。默认就是 repl 模式。

由于 mpy 需要放在板卡内部文件系统里。所以需要连接 kmbox。输入：open COMxx ---连接开发板（COMxx 是串口号）



```
C:\Users\hw\MPY\mpfs-1.0.2.exe

Documented commands (type help <topic>):
=====
EOF  close    get    lexecfile  ls    mput  open  quit  runfile
cat  exec     help   ll       md    mpyc  put   repl  view
cd   execfile  lcd    lpwd     mget  mrm   pwd   rm

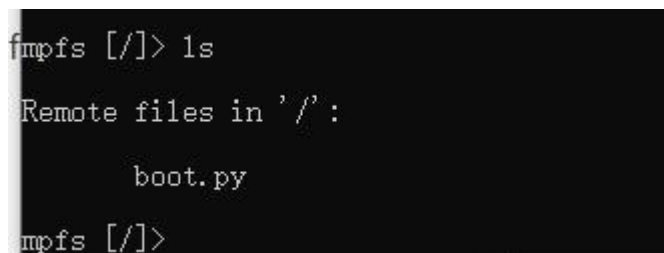
Undocumented commands:
=====
c e ef lef o q r rf v

can input help ls or other command if you don't know how to use it.
looking for all port...
serial_name : USB-SERIAL CH340 (COM26) : COM26
input 'open COM26' and enter connect your board.

** Micropython File Shell v1.0.2, sw@kaltpost.de & junhuanchen@qq.com **
-- Running on Python 3.5 using PySerial 3.3 --

mpfs [/]> open COM26
Connected to esp32
mpfs [/]> _
```

连接完成后可以看到连接成功，这里你可以像在 linux 内部操作一样输入指令 ls mv 等指令。



```
mpfs [/]> ls

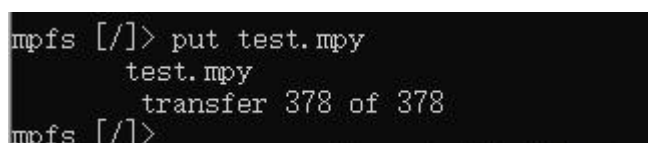
Remote files in '/':

    boot.py

mpfs [/]>
```

例如 ls 后可以看见当前文件系统内部有 boot.py 脚本。Kmbox 内部是有文件系统的。可以存储各种类型的文件。你可以用 os 库创建文件，修改文件名等等操作都是可以的。现在需要将 mpy 文件写到 kmbox 的文件系统。

只需要执行一条指令：put test.mpy



```
mpfs [/]> put test.mpy
    test.mpy
    transfer 378 of 378
mpfs [/]>
```

执行这条指令时，注意 test.py 和 mpfs.exe 需要在同级目录内：

名称	修改日期	类型	大小
mpfs-1.0.2	2020/10/1 22:08	应用程序	17,098 KB
test.mpy	2020/10/2 1:05	MPY 文件	1 KB
test	2020/10/1 22:19	JetBrains PyChar...	1 KB

此时 mpy 文件已经烧写到 kmbox 内部。此时 mpfs 可以关闭了。接下来就是运行刚刚的加密脚本。怎样使用，其实和普通的 py 文件一样，直接 `import test` 就行。

```
>>>
>>> import test
>>> test.
__class__      __name__      __file__      km
run            IsAuthor     author
>>> test.█
```

可以看到，作者写的函数在这里都有。你可以调用。从作者端的源码知道，必须首先调用 `author` 并且传入的密钥是 123456 才能使用和后面的 `run` 函数。我们先调用以下 `run` 试试

:

```
>>>
>>>
>>> import test
>>> test.
__class__      __name__      __file__      km
run            IsAuthor     author
>>> test.run()
对不起您还未授权，请调用author输入注册码
>>>
```

接下来假设我是从作者那里花了一块钱知道了密钥是 123456，调用以下 `author('123456')`

```
>>>
>>>
>>> import test
>>> test.
__class__      __name__      __file__      km
run            IsAuthor     author
>>> test.run()
对不起您还未授权，请调用author输入注册码
>>> test.author('123456')
授权码正确
>>>
```

密钥正确后我就能使用 `run` 函数了。

```

>>>
>>>
>>> import test
>>> test.
__class__      __name__      __file__      km
run            IsAuthor    author
>>> test.run()
对不起您还未授权，请调用author输入注册码
>>> test.author('123456')
授权码正确
>>>
>>>
>>> test.run()
脚本已授权正在运行中...
脚本已授权正在运行中...
脚本已授权正在运行中...
脚本已授权正在运行中...
脚本已授权正在运行中...
脚本已授权正在运行中...
脚本已授权正在运行中...
脚本已授权正在运行中...
脚本已授权正在运行中...
脚本已授权正在运行中...

```

当然以上代码只是演示，isAuthor 都直接给到变量里面了，应该把他封装成 class，设置为私有变量。加密也不是简单的字符串。可以使用机器码的方式绑定。每个设备的密钥都不一样。规则由你自己定。

## 七、预防封号

### 1. 硬件防封

本章主要讲解一下如何避免硬件封号。虽然 kmbox 属于纯硬件物理外挂，想封 kmbox 前面说过除非不让用键盘鼠标。在前面你应该知道。USB 设备都是有 VID 和 PID。如果有的游戏已经撕破脸，就是不允许指定的 VID 和 PID 设备玩游戏。那也是没办法的。毕竟游戏公司有权这么做。当年 360 和 QQ 就是互怼！但是一般游戏公司不敢这么做。因为有 VID 和 PID 的设备本身就是合法设备。禁用一个 VID 全世界可能几百万设备不能使用，kmbox 的 VID 和 PID 是正规合法的。该 VID 和 PID 也可能用于其他标准的键盘鼠标。如果游戏公司限定了这个 VID 和 PID，那么就会将用到该 IC 的所有其他类型的键盘鼠标全部屏蔽。为了搞 kmbox，需要误伤一大堆垫背的，这是得不偿失的。所以很少有游戏这么做。但是如果真这么做了，那么该如何解决。

那就修改 VID 和 PID。注意：VID 和 PID 是需要交钱给 USB 协会，请不要随意修改，擅自使用其他公司的 VID 和 PID 都是违法侵权行为。由此造成的后果自行承担。

查询常见的各大供应商的 VID 和 PID：<http://www.linux-usb.org/usb.ids>

例如下面罗技公司的 VID=046d , 东芝的 VID=046C,  
kmbox 的 VID=4348 046b American Megatrends,  
Inc.

- 0001 Keyboard
- 0101 PS/2 Keyboard, Mouse & Joystick Ports
- 0301 USB 1.0 Hub
- 0500 Serial & Parallel Ports
- ff10 Virtual Keyboard and Mouse

046c Toshiba Corp., Digital Media

Equipment 046d Logitech, Inc.

- 0082 Acer Aspire 5672 Webcam
- 0200 WingMan Extreme Joystick
- 0203 M2452 Keyboard
- 0242 Chillstream for Xbox 360
- 0301 M4848 Mouse
- 0401 HP PageScan
- 0402 NEC PageScan

4348 WinChipHead

- 5523 USB->RS 232 adapter with Prolific PL 2303 chipset
- 5537 13.56Mhz RFID Card Reader and Writer
- 5584 CH34x printer adapter cable

如果 kmbox (4348) 被加入了黑名单, 我们可以手动修改 VID。改成 046D, 那么电脑就会认为现在接的设备是罗技公司的产品。而不再是 kmbox。修改 VID 和 PID 都是违法侵权行为, 除非这个 VID 和 PID 属于你自己。以下内容仅供学习参考。其中 VID 是厂商号, PID 是产品号。

## 如何修改VID

Vid 属于 USB 设备的制造厂商编号，由 USB 协会分配。Kmbox 默认 VID 是 0x4348. 读写 VID 需要用到 device 模块。首先 import device 你可以输入下面指令查看 VID。



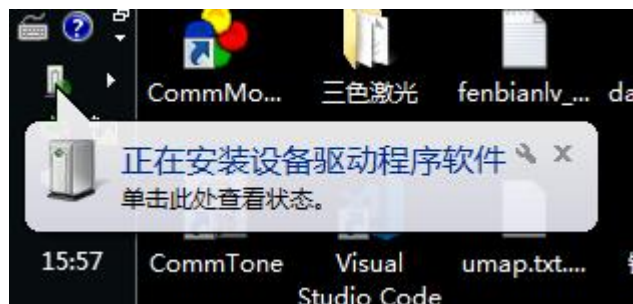
输入 device.VID( 'help' ) 查看如何使用 VID.

```
>>> device.VID()
VID=4348
>>> device.VID('help')
自定义Kmbox的VID:
  Kmbox的默认VID是4348. VID相当于设备的身份证，不可以随意修改. 主机会根据VID匹配不同的驱动
  有的游戏检测到特定VID后会禁用设备数据，此时可以通过修改VID绕开这个屏蔽，修改VID是侵权行为。
  请不要乱用别人的VID。本函数仅供学习使用。修改VID方法如下：
  device.VID()      : 无参数是查询当前VID
  device.VID('046D') : 有参数是设置VID=046D(戴尔设备)
  设置完VID后需要调用enable(1)函数重新枚举
>>> █
```

例如现在修改 VID=046D(伪造罗技公司的产品)。

```
>>> device.VID('046D')
自定义VID=046d, 启用新参数请调用enable(1)函数
>>> █
```

修改 VID 后并不是立即生效，需要调用 device.enable(1). 让电脑重新识别 USB 设备。调用后你的电脑会重新枚举。



这样你就完美的把自己伪装成罗技的键盘或者鼠标了。原来针对 4348 的 VID 封锁就当然无存了。再次强调，VID侵权属于非法行为，请不要随意修改，否则后果自负。



## 如何修改 PID

PID 是产品编号，用于区分同一公司的不同产品。例如罗技有各种型号的键盘鼠标。他们的VID 一样，所以用PID 来进行区分。PID 的修改方式和 VID 一样。详细参考帮助文档。

Kmbox 默认 PID=0x1234

```
>>> device.PID('help')
自定义Kmbox的PID:
PID是产品ID，不同的PID可以区分同一厂商的不同设备，便于主机加载不同驱动
device.PID()      : 无参数是查询当前PID
device.PID('046D'): 有参数是设置PID=046D
设置完PID后需要调用enable(1)函数重新枚举
>>> █
```

在此就不过多举例了。请自行尝试。

## 如何修改 USB 设备名称

Kmbox 插上电脑后，默认名称是 kmbox。有人说反外挂机制可能会根据设备名称来限制设备。那么如何修改名称呢？其实你要知道，绝大部分的键盘鼠标为了节省成本。都没有设备的字符描述符（需要 ROM 存储）。默认没有设备名称。万一真的是按照设备名称来禁用设备该怎么解决呢？前面 VID 和 PID 两个函数，只要你使能了任意一个。Kmbox 的字符描述就会被清空。主机 PC 就不会读到设备的名称。读不到名字就不能随意封杀。而且个人觉得依据名称来识别设备的不太可能。如果我是反外挂的程序员，我会通过 VID 和 PID 入手。行为检测为主，因此我没有提供修改名称函数。

## 2. 软件防封

在前面的硬件防封中可以解决硬件原因导致封号，但是硬件领域有句话叫：防呆不防傻。很多封号并不是硬件导致的，而是软件引起的。举个例子，你做了一个每秒狂点鼠标 500 次的连点器，持续 1 天。是个人想想都不可能，第一，每秒 500 次，试问哪个大神能做到。每秒能点 10 次就已经非人哉了。500次想都不用想，绝对是外挂。第二持续一天，这也不太可能。还是这么高强度的点击持续 1 天。所以写硬件挂要尽量模拟人操作。用挂就要装得像点，别那么假，约束好自己的行为。写脚本的时候最好按照实际情况来，多用随机数，多模拟过程，一切控制在合理的范围内才行。这里只能自己慢慢写脚本去体会了。

举几个经常容易犯错的例子：

- 1、延迟太过准确（每次键盘鼠标事件的延迟规律太准确，不像人操作。）
- 2、按键时间太快（kmbox 为了提高代码效率，一条按键指令会以最快的速度执行约 1ms，然而实际按键，从按下到抬起，时间至少 80ms 以上。模拟按键请尽量用 down 和 up 模拟，少用 press）
- 3、鼠标移动跨度太大。（尽量将大跨度的鼠标移动分割成小跨度移动，少用move(800, 600)这样的代码，而是用 move(1, 1)，X 方向循环 800 次，y 方向循环 600 次。）

总而言之，尽量装得像人在操作吧。

### 3. kmbox 的行为模仿

在上一节的行为检测中，一再强调，用外挂要会装，尽量装得像人。那么如何装得像人呢？Kmbox 提供了一些简单而且很实用的 API，可以避免你的操作太机械化。

1、延迟太过准确。

针对这个 kmbox 提供了一个随机延迟函数。详见 delay 函数用法

2、按键时间太快。

针对这个问题，kmbox 提供了 press 函数，你可以给 press 带上三个参数，用来指定按键按下的时间。详见 press 函数用法。

3、鼠标移动跨度太大。

针对这个问题。Kmbox 内置 move 和 moveto 函数已支持过程模拟。你无需写复杂的移动逻辑，只用给几个参数，kmbox 就能自动让鼠标移动变成平滑的曲线。再也不用担心直去直来，横平竖直的机械操作，取而代之的是同样的起点到终点移动，kmbox 能顺滑的连续过度。而且既可以做到每次路径不一致，又可以做到特定的平滑过度。Kmbox 的平滑过度原理是基于贝塞尔曲线（如需了解原理请自行百度）。

下面就再来谈谈 move 和 moveto 两个函数吧。move 和 moveto 如果只给两个参数。就表示一次性的快速移动到给定坐标。如果给三个参数。就表示从起点到终点用几条折线来拟合。由常识可以知道，第三个参数数值越大，那么拟合出来的曲线就会越平滑。而且过两点的曲线有无数条。所以当不给其他辅助限定条件时，每调用一次函数就会有一条不同的曲线。这就是 kmbox 的随机曲线。如果你想控制曲线的扭曲程度，你可以给一个参考点坐标。这个参考点坐标可以修正曲线的扭曲方向，使得曲线向参考点方向扭曲。所以 move 和 moveto 函数有三个参数或者 5 个参数，三个参数是随机拟合出一条贝塞尔曲线，五个参数是拟合一个向参考点靠近的贝塞尔曲线。你们可以自行运行下面脚本查看每个参数的作用。

再强调一下：

一、使用多条直线来拟合曲线方式肯定比单一的横平竖直要慢。目前 kmbox 每增加一段拟合直线需要 1ms 的时间。拟合长度越大曲线越平滑，但是耗时就越高。拟合长度越小，曲线拐点阶跃越明显。所以最好根据实际情况来拟合。推荐以 127 为最大刻度单位来计算需要多少条拟合直线。例如从 (0, 0) 移动到 (1920, 1080) 最大跨度是 1920 个单位。那么用 1920 除以 127 等于 15.118. 因此推荐拟合直线最小值为 16。

二、目前 kmbox 没有受到任何限制，除非有需要模拟过程可以使用过程模拟，否则正常情况该怎么写脚本就怎么写。

三、如果不知道怎么使用参考点来约束曲线。建议直接用三个参数的函数。设置多少条直线拟合即可。Kmbox 会自动随机在起点和终点范围内选取一个参考点，根据拟合参数自动拟合。

## 八、附录

### 附录 1 键盘全键值对应表

此表是键盘 HID 数据与按键对应值，也就是 table 函数的内容。左边是按键名称，右边是 HEX 值。如果使用不在 table 函数中的按键，请记得将 HEX 转换为 10 进制使用。

#define KEY_NONE	0x00
#define KEY_ERRORROLLOVER	0x01
#define KEY_POSTFAIL	0x02
#define KEY_ERRORUNDEFINED	0x03
#define KEY_A	0x04
#define KEY_B	0x05
#define KEY_C	0x06
#define KEY_D	0x07
#define KEY_E	0x08
#define KEY_F	0x09
#define KEY_G	0x0A
#define KEY_H	0x0B
#define KEY_I	0x0C
#define KEY_J	0x0D
#define KEY_K	0x0E
#define KEY_L	0x0F
#define KEY_M	0x10
#define KEY_N	0x11
#define KEY_O	0x12
#define KEY_P	0x13
#define KEY_Q	0x14
#define KEY_R	0x15
#define KEY_S	0x16
#define KEY_T	0x17
#define KEY_U	0x18
#define KEY_V	0x19
#define KEY_W	0x1A
#define KEY_X	0x1B
#define KEY_Y	0x1C
#define KEY_Z	0x1D
#define KEY_1_EXCLAMATION_MARK	0x1E
#define KEY_2_AT	0x1F
#define KEY_3_NUMBER_SIGN	0x20
#define KEY_4_DOLLAR	0x21
#define KEY_5_PERCENT	0x22
#define KEY_6_CARET	0x23
#define KEY_7_AMPERSAND	0x24
#define KEY_8_ASTERISK	0x25
#define KEY_9_OPARENTHESIS	0x26
#define KEY_0_CPARENTHESIS	0x27
#define KEY_ENTER	0x28

#define KEY_ESCAPE	0x29
#define KEY_BACKSPACE	0x2A
#define KEY_TAB	0x2B
#define KEY_SPACEBAR	0x2C
#define KEY_MINUS_UNDERSCORE	0x2D
#define KEY_EQUAL_PLUS	0x2E
#define KEY_OBRACKET_AND_OBRACE	0x2F
#define KEY_CBRACKET_AND_CBRACE	0x30
#define KEY_BACKSLASH_VERTICAL_BAR	0x31
#define KEY_NONUS_NUMBER_SIGN_TILDE	0x32
#define KEY_SEMICOLON_COLON	0x33
#define KEY_SINGLE_AND_DOUBLE_QUOTE	0x34
#define KEY_GRAVE_ACCENT_AND_TILDE	0x35
#define KEY_COMMA_AND_LESS	0x36
#define KEY_DOT_GREATER	0x37
#define KEY_SLASH_QUESTION	0x38
#define KEY_CAPS_LOCK	0x39
#define KEY_F1	0x3A
#define KEY_F2	0x3B
#define KEY_F3	0x3C
#define KEY_F4	0x3D
#define KEY_F5	0x3E
#define KEY_F6	0x3F
#define KEY_F7	0x40
#define KEY_F8	0x41
#define KEY_F9	0x42
#define KEY_F10	0x43
#define KEY_F11	0x44
#define KEY_F12	0x45
#define KEY_PRINTSCREEN	0x46
#define KEY_SCROLL_LOCK	0x47
#define KEY_PAUSE	0x48
#define KEY_INSERT	0x49
#define KEY_HOME	0x4A
#define KEY_PAGEUP	0x4B
#define KEY_DELETE	0x4C
#define KEY_END1	0x4D
#define KEY_PAGEDOWN	0x4E
#define KEY_RIGHTARROW	0x4F
#define KEY_LEFTARROW	0x50
#define KEY_DOWNARROW	0x51
#define KEY_UPARROW	0x52
#define KEY_KEYPAD_NUM_LOCK_AND_CLEAR	0x53
#define KEY_KEYPAD_SLASH	0x54
#define KEY_KEYPAD_asteriks	0x55
#define KEY_KEYPAD_MINUS	0x56
#define KEY_KEYPAD_PLUS	0x57
#define KEY_KEYPAD_ENTER	0x58

#define KEY_KEYPAD_1_END	0x59
#define KEY_KEYPAD_2_DOWN_ARROW	0x5A
#define KEY_KEYPAD_3_PAGEDN	0x5B
#define KEY_KEYPAD_4_LEFT_ARROW	0x5C
#define KEY_KEYPAD_5	0x5D
#define KEY_KEYPAD_6_RIGHT_ARROW	0x5E
#define KEY_KEYPAD_7_HOME	0x5F
#define KEY_KEYPAD_8_UP_ARROW	0x60
#define KEY_KEYPAD_9_PAGEUP	0x61
#define KEY_KEYPAD_0_INSERT	0x62
#define KEY_KEYPAD_DECIMAL_SEPARATOR_DELETE	0x63
#define KEY_NONUS_BACK_SLASH_VERTICAL_BAR	0x64
#define KEY_APPLICATION	0x65
#define KEY_POWER	0x66
#define KEY_KEYPAD_EQUAL	0x67
#define KEY_F13	0x68
#define KEY_F14	0x69
#define KEY_F15	0x6A
#define KEY_F16	0x6B
#define KEY_F17	0x6C
#define KEY_F18	0x6D
#define KEY_F19	0x6E
#define KEY_F20	0x6F
#define KEY_F21	0x70
#define KEY_F22	0x71
#define KEY_F23	0x72
#define KEY_F24	0x73
#define KEY_EXECUTE	0x74
#define KEY_HELP	0x75
#define KEY_MENU	0x76
#define KEY_SELECT	0x77
#define KEY_STOP	0x78
#define KEY_AGAIN	0x79
#define KEY_UNDO	0x7A
#define KEY_CUT	0x7B
#define KEY_COPY	0x7C
#define KEY_PASTE	0x7D
#define KEY_FIND	0x7E
#define KEY_MUTE	0x7F
#define KEY_VOLUME_UP	0x80
#define KEY_VOLUME_DOWN	0x81
#define KEY_LOCKING_CAPS_LOCK	0x82
#define KEY_LOCKING_NUM_LOCK	0x83
#define KEY_LOCKING_SCROLL_LOCK	0x84
#define KEY_KEYPAD_COMMA	0x85
#define KEY_KEYPAD_EQUAL_SIGN	0x86
#define KEY_INTERNATIONAL1	0x87
#define KEY_INTERNATIONAL2	0x88

#define KEY_INTERNATIONAL3	0x89
#define KEY_INTERNATIONAL4	0x8A
#define KEY_INTERNATIONAL5	0x8B
#define KEY_INTERNATIONAL6	0x8C
#define KEY_INTERNATIONAL7	0x8D
#define KEY_INTERNATIONAL8	0x8E
#define KEY_INTERNATIONAL9	0x8F
#define KEY_LANG1	0x90
#define KEY_LANG2	0x91
#define KEY_LANG3	0x92
#define KEY_LANG4	0x93
#define KEY_LANG5	0x94
#define KEY_LANG6	0x95
#define KEY_LANG7	0x96
#define KEY_LANG8	0x97
#define KEY_LANG9	0x98
#define KEY_ALTERNATE_ERASE	0x99
#define KEY_SYSREQ	0x9A
#define KEY_CANCEL	0x9B
#define KEY_CLEAR	0x9C
#define KEY_PRIOR	0x9D
#define KEY_RETURN	0x9E
#define KEY_SEPARATOR	0x9F
#define KEY_OUT	0xA0
#define KEY_OPER	0xA1
#define KEY_CLEAR_AGAIN	0xA2
#define KEY_CRSEL	0xA3
#define KEY_EXSEL	0xA4
#define KEY_KEYPAD_00	0xB0
#define KEY_KEYPAD_000	0xB1
#define KEY_THOUSANDS_SEPARATOR	0xB2
#define KEY_DECIMAL_SEPARATOR	0xB3
#define KEY_CURRENCY_UNIT	0xB4
#define KEY_CURRENCY_SUB_UNIT	0xB5
#define KEY_KEYPAD_OPARENTHESIS	0xB6
#define KEY_KEYPAD_CPARENTHESIS	0xB7
#define KEY_KEYPAD_OBRACE	0xB8
#define KEY_KEYPAD_CBACE	0xB9
#define KEY_KEYPAD_TAB	0xBA
#define KEY_KEYPAD_BACKSPACE	0xBB
#define KEY_KEYPAD_A	0xBC
#define KEY_KEYPAD_B	0xBD
#define KEY_KEYPAD_C	0xBE
#define KEY_KEYPAD_D	0xBF
#define KEY_KEYPAD_E	0xC0
#define KEY_KEYPAD_F	0xC1
#define KEY_KEYPAD_XOR	0xC2
#define KEY_KEYPAD_CARET	0xC3



#define KEY_KEYPAD_PERCENT	0xC4
#define KEY_KEYPAD_LESS	0xC5
#define KEY_KEYPAD_GREATER	0xC6
#define KEY_KEYPAD_AMPERSAND	0xC7
#define KEY_KEYPAD_LOGICAL_AND	0xC8
#define KEY_KEYPAD_VERTICAL_BAR	0xC9
#define KEY_KEYPAD_LOGIACL_OR	0xCA
#define KEY_KEYPAD_COLON	0xCB
#define KEY_KEYPAD_NUMBER_SIGN	0xCC
#define KEY_KEYPAD_SPACE	0xCD
#define KEY_KEYPAD_AT	0xCE
#define KEY_KEYPAD_EXCLAMATION_MARK	0xCF
#define KEY_KEYPAD_MEMORY_STORE	0xD0
#define KEY_KEYPAD_MEMORY_RECALL	0xD1
#define KEY_KEYPAD_MEMORY_CLEAR	0xD2
#define KEY_KEYPAD_MEMORY_ADD	0xD3
#define KEY_KEYPAD_MEMORY_SUBTRACT	0xD4
#define KEY_KEYPAD_MEMORY_MULTIPLY	0xD5
#define KEY_KEYPAD_MEMORY_DIVIDE	0xD6
#define KEY_KEYPAD_PLUSMINUS	0xD7
#define KEY_KEYPAD_CLEAR	0xD8
#define KEY_KEYPAD_CLEAR_ENTRY	0xD9
#define KEY_KEYPAD_BINARY	0xDA
#define KEY_KEYPAD_OCTAL	0xDB
#define KEY_KEYPAD_DECIMAL	0xDC
#define KEY_KEYPAD_HEXADecimal	0xDD
#define KEY_LEFTCONTROL	0xE0
#define KEY_LEFTSHIFT	0xE1
#define KEY_LEFTALT	0xE2
#define KEY_LEFT_GUI	0xE3
#define KEY_RIGHTCONTROL	0xE4
#define KEY_RIGHTSHIFT	0xE5

## 附录 2 kmbox 结构尺寸图



## 九、常见问题解决办法

### 1. 键盘鼠标无法正常工作

当键盘或者鼠标接到 kmbox 上无法正常工作时，kmbox 作为主机，需要兼容所有的键盘和鼠标，笔者手头设备有限，市面上的键盘鼠标千千万，无法知道其特性。所以记得打开串口，插入 USB 设备。你可能会看到下面这行打印。

```
I (37833) : VID==0x04CA&&PID==0x0061&&OID==0x0100&&HID==0x002E Version 3.0.0 @Feb 10 2021 17:42:16
E (37843) : host.configX(1226,97,256,46,mType=?,kType=?)
此设备可能无法正常使用。如果不能正常工作，您可以使用host.config函数来强制匹配。
如果您尝试过所有的匹配还是无法正常使用，请复制以上所有内容，保存到一个txt中，发送到366021972@qq.com
如果操作正常可以不用理会，kmbox感谢您的添砖加瓦！
I (37879) kmbox: 设置配置值为: 01
I (37883) kmbox: 设置配置OK
I (37887) kmbox: 总电流需求: 100mA 其中Hub1:100mA,Hub2:0mA
```

这里有两种方法可以让你的键盘鼠标工作：

方法一：调用 `host.configX` 函数。调试哪个参数适合你的键盘鼠标，在控制台输入 `import host`。 `host` 模块里面有两个 `config` 函数，分别是 `config0` 和 `config1`。 因为

kmbox 有两个USB口。这两个 USB 口插键盘或者鼠标都行。configX (X=0 或 1) 函数用来强制指定两个端口的 USB 配置属性。configX 的前四个参数你可以从 log 中得到。

E (37843) : `host.configX(1226,97,256,46,mType=?,kType=?)` 其中

`mType` : 表示鼠标报告解析方式,其取值范围是 0 到 12 .

`kType` : 表示键盘报告解析方式,其取值范围是 0 到6.

如果鼠标不能正常工作,你需要改变 `mType` 的值,其取值范围是 0 到 12. 你可以依次将 `mType` 的值设置为0 到 12 中的一个值。调用完毕后,看看鼠标能不能正常工作。如果可以正常工作,那么你的鼠标对应的 `mType` 值就已经确定了。同理键盘不能正常工作的话就依次修改 `kType` 的值。直到键盘可以正常工作为止。

注意:

一、无线键鼠接收器接到 kmbox 上,接收器里会接收键盘和鼠标数据。你需要同时设置 `mType` 和 `kType` 的值。如果你的设备是纯键盘,那么请将 `mType` 的值设置为 255。如果你的设备是纯鼠标,那么将 `kType` 的值设置为255。

二、`config0`和`config1`不对应USB口,对应的是设备。设备可以插入任意USB口。如果你有一个设备无法使用,你用了`config0`来强制匹配。第二个不能使用的设备你需要用`config1`函数。如果继续用`config0`函数,那么就会冲刷掉`config0`的配置。kmbox最多控制两个设备。`config0` 和`config1`是用来对应这两个设备的。

当你用上面的方法成功适配好键盘鼠标后,你可以创建一个 `config.py` 文件。将配置值写入`config.py`文件中。然后下载到开发板。那么以后就能永久自动识别和匹配了。如下图所示:

下载完后重启开发板即可自动识别:

```
M End Collection

M End Collection

强制指定键鼠解析模式(0,255)
[0;32mI (4640) kmbox: 设置配置值为: 01[0m
[0;32mI (4644) kmbox: 设置配置OK[0m
[0;32mI (4648) kmbox: 总电流需求: 100mA 其中Hub1:100mA,Hub2:0mA[0m
-----欢迎使用kmbox VerB-----
```

如果你尝试过所有键鼠参数的匹配还是无法正常使用。那么请记复制所有打印内容到一个txt 文件中。并以设备名称命名。例如,罗技 G102 鼠标无法识别。请复制所有打印,命名为“罗技 G102 鼠标.txt”(如下)。并将该文件[发送给作者](#),增加适配后即可支持。

## 2. 一插拔键鼠串口就断开

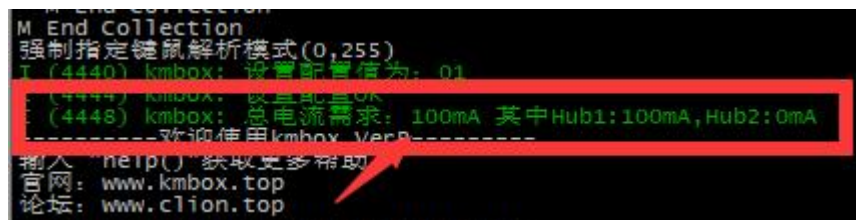
这是典型的供电不足问题。不要将 kmbox 的 USB1 口接任何 HUB扩展器。请确保直连到电脑的 USB 口。并且保证 USB 口供电充足。

常识：一个 USB 端口供电电压是5V，最大电流是500mA。当电脑USB接到kmbox后，kmbox上再接键盘或者鼠标，其电源还是电脑的USB。Kmbox工作需要消耗100mA电流（蓝牙模式300mA）。如果你的键盘鼠标消耗电流较大，其总电流需求加上kmbox的超过500mA。那么kmbox可能无法正常工作。因为你的电脑USB口无法提供足够的电流。所以请保证USB端口的充足供电。

键盘鼠标需求电流可以在开机log中找到，kmbox 没有做电源管理。默认你电脑电能充足供应：

```
I (161224) kmbox:      =====配置描述符开始=====
I (161230) kmbox:      bLength:          09
I (161234) kmbox:      bDescriptorType:  02(固定为 02)
I (161240) kmbox:      wTotalLengthH/L:  0054(集合长度)
I (161246) kmbox:      bNumInterfaces:  03(接口数)
I (161251) kmbox:      bConfigValue:     01(配置值)
I (161257) kmbox:      iConfiguration:  04(配置字符索引)
I (161263) kmbox:      bmAttributes:     A0(总线供电 支持远程唤醒)
I (61270) kmbox:      MaxPower:          31(最大电流 98mA)
```

请务必确保所有设备电流总和小于 USB 规范 500mA. 且你的电脑能提供充足电流。目前板卡已经集成了总电流需求提示。



如果总电流大于500mA，kmbox可能无法正常工作。后面的键鼠也不可能正常工作。毕竟一个USB电流只有500mA. 原来键盘鼠标是分开端口供电。现在一个USB端口给kmbox供电，外接键盘和供电，外接鼠标供电。如果出现这种情况建议使用双端口USB线缆，提高USB供电能力。双头USB线连接如下：

### 3. 插上板卡后就不停的重启

这个可能是串口驱动安装不对导致板卡频繁重启，请下载安装驱动。