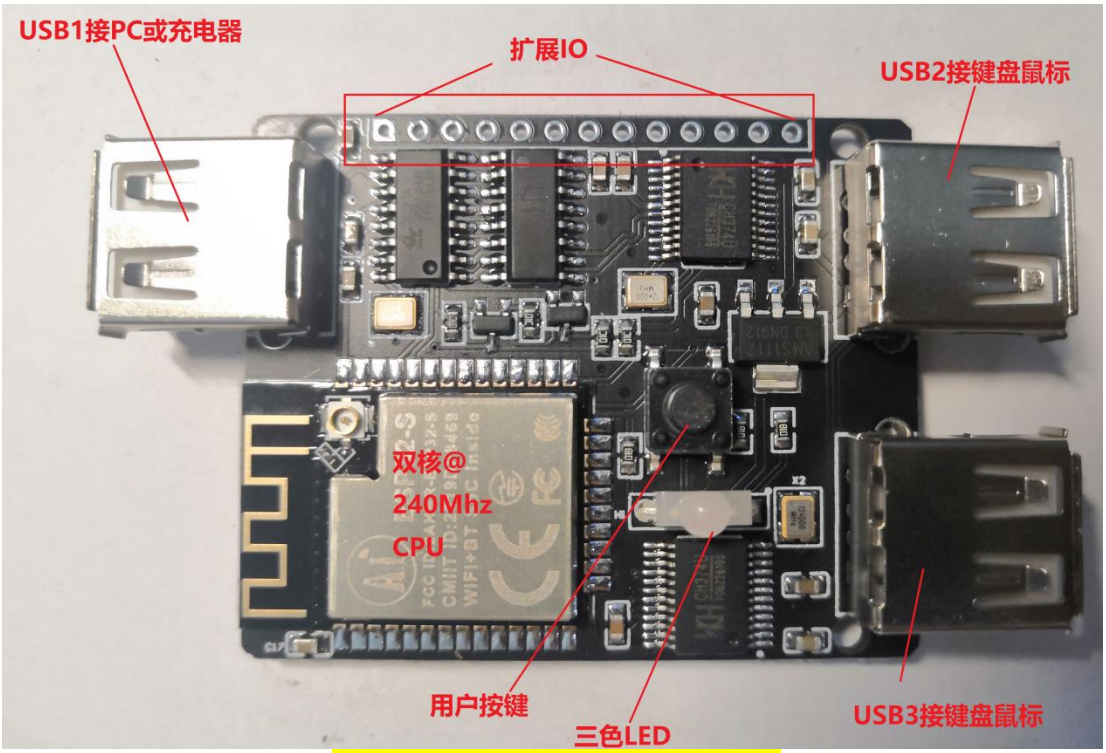
（本文是 kmbox 的使用手册，左边有目录。建议新入坑的小伙伴仔细阅读一遍。我们的 Q 群是 **190702064。里面有丰富的脚本 demo。欢迎进去讨论。如果有好意见也可以提出。谢谢！**

Kmbox 简介

Kmbox 是一款小巧但功能强大的 USB 开发板。他是一个纯硬件 USB 主从设备控制器。能完全硬件级别模拟和控制键盘 44 鼠标等 USB 设备。他无需安装额外驱动。也不需要注入 DLL。其脚本完全脱离 PC，运行在板卡的 CPU 上，能轻松让普通键盘鼠标（非宏键鼠）拥有*可编程功能（宏功能）。你只需要简单的 python 调用即可让普通键盘鼠标化腐朽为神奇。用他可以快速制作纯硬件物理外挂。安全，高效，不怕检测。不仅如此。他还能带你快速入门 python。学习编程，玩转硬件。边学边玩。边玩边用。其乐无穷！下到海底捉鳖，上至九天揽月。不管是底层的硬件逻辑还是上层的框架应用。kmbox 都可以陪你一路打怪升级。绝对是 DIY 爱好者的福音！

kmbox 实物图-



正面默认插针已焊接（DIY 扩展 IO）

7

AB 面

kmbox 硬件特性

Kmbox 包含一个双核 CPU，3 个 USB 硬件端口。一个三色 LED，一个用户按键。以及一组 GPIO 扩展口。下面详细介绍各部分功能。

i

USB1 端口特性

USB1 端口可连接 PC 或者充电器，用于板卡供电。在连接到 PC 等主机设备时。Kmbox 会自动枚举成一个标准的键盘和鼠标。PC 自动识别不需要任何驱动 i’m（键盘鼠标属于标准 HID 设备，系统自带驱动。）。且 kmbox 模拟的键盘鼠标绝非普通键鼠。而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 键盘特性对比

	Kmbox-键盘	其他键盘

回报率	1ms(1000 次/S)	10ms(100 次/S)
报告格式	1+10 按键(可定制任意按键无冲)	1+6 按键
BIOS 模式	不支持（不屑支持）	支持
端点长度	64Byte	8Byte

表 1

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。性能甩他们 10 条街。

2、 报告格式就是一包数据能发送的无冲按键数量。
常规键盘是 1+6 模式，属于 6 键无从冲，kmbox 是 1+10，属于 10 键无冲（PS：如果愿意可以把 kmbox 做到 108 键无冲都行。但是没那个必要呀，毕竟我们只有 10 个手指头）。

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。

举个简单例子你就知道以上参数的意义，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。如果你觉得 20ms 不算差距。那么再对比看看鼠标的技术参数吧。

Kmbox 鼠标特性对比

	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+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

1、 回报率：
kmbox 是回报速度普通鼠标的 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。干嘛用的自己去摸索（PS：可定制任意按键数）就你的鼠标没有侧键，接入 kmbox 后就等于有 5 个侧键。
3、 XY 坐标范围：
kmbox 的 XY 坐标范围是普通鼠标的 256 倍。

同样举个例子就能看到差异性：

假设鼠标从屏幕右下角（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。到这里应该可以感受到什么是硬件差距。一个需要 160ms，一个只要 1ms。有没有爱了爱了的感觉呀？

USB2、 3 端口特性

前面的 USB1 端口是用来模拟 USB 设备的。现在 USB2、3 端口是用来接 USB 设备的。此时的 Kmbox 就是 USB 主机（等价于电脑主机）。不管你 USB2、3 端口接的啥。都归我 kmbox 管。外接的设备是否工作，怎样工作，都由 kmbox 说了算。USB2、3 端口理论上支持所有类型的 USB2.0 全速设备。

目前 kmbox 默认接管外设为 USB 键盘和鼠标（如需控制其他外设请定制）。也就是说如果将键盘鼠标接到 kmbox 上。他就能通过实时获取键盘鼠标的所有数据。如果 kmbox 在中间把这些底层的数据做点手脚。就可以实现一些意想不到的功能。比如任意改键，任意重组。任意屏蔽按键，任意逻辑判断，键盘连点，鼠标连点等等等等。Kmbox 将修改后的数据通过 USB1 端口（或者蓝牙）给 PC。就能让普通键盘鼠标拥有可编程宏功能。详见 kmbox 教程键盘宏和鼠标宏章节。

CPU 特性

Kmbox 内部包含一颗国产高速双核 CPU，运行频率 240MHz。上面跑 Python 脚本。CPU 内部集成蓝牙和 wifi 功能。可以让有线 USB 键盘鼠标通过 kmbox 放飞自我。将有线键鼠蓝牙化。连接到手机、平板、或者电脑。下面是 CPU 的详细规格参数：

CPU	ESP32，Xtensa® 32-bit LX6 双核处理器,运算能力最高可以达 600 DMIPS 448 KByte
ROM	520 KByte SRAM RTC 16 KByte SRAM
通用IO	22
封装	SMD-38
SPI Flash	默认 32Mbit
支持接口	SD卡、UART、SPI、SDIO、I2C、PWM、I2S、IR、GPIO、电容式触摸传感器、ADC、DAC
音频	CVSD 和 SBC 音频
串口速率	默认 115200 bps
板上时钟	40 MHz 晶振
天线形式	板载 PCB 天线和 IPEX 天线座
WiFi 802.11	b/g/n/d/e/i/k/r（802.11n，速度高达150 Mbps）A-MPDU和 A-MSDU聚合，支持0.4
	μs防护间隔2.4 ~ 2.5 GHz
蓝牙	V4.2 BR/EDR和BLE标准,具有 -98 dBm灵敏度的NZIF接收器,Class-1，Class-2和Class-3
	发射器，支持AFH(调频自适应)，支持CVSD和SBC音频格式
电压/电流	2.7V~3.6V(推荐3.3V) / 工作电流：平均80mA，供电电流：最小500mA
尺寸	25.5*18*3(±0.2)mm

注意：由于普通 USB 端口供电电流只有 5V@500mA，kmbox 需求供电电流为 3.3V@500mA. 部分主机会存在供电电流不足问题。导致 kmbox 工作异常。因此 kmbox 默认关闭蓝牙功能，减少功耗。如需使用蓝牙，请保证供电充足。软件脚本开启，详见蓝牙章节。

DIY 扩展特性

Kmbox 板载一个三色 LED，方便脚本运行时提供直观的状态显示。三色 LED 采用 PWM 控制，能实现任意占空比的颜色调制，调制深度是 13 位。也就是 RGB 的精度是 13:13:13.可以支持到 2 的 39 次方种颜色。LED 的三个 GPIO 可以通过脚本释放(详见 rgb free 函数)。可以复用为其他功能。

Kmbox 默认提供一个按键开关，可以用脚本将这个 IO 修改为任意你想要的功能。另外还有一组 GPIO，高阶脚本玩家可以用他们接各种模块，做各种物理外挂。也可以用他们来快速验证硬件模块。驱动各种 IC,DIY 各种智能硬件。详见硬件扩展章节

kmbox 软件特性

1、 kmbox 内置 python 释义器。Python 绝对是公认小白看看就会的编程语言。
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。你可以通过串口与 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。免驱插上就能用的那种哟。
6、 Kmbox 支持文件系统管理，你可以存放多个脚本在板子上，便你调用。

7、 Kmbox 固件更新。不需要其他工具。板子自带下载电路。简单方便。

Kmbox 典型应用

牛皮不是吹的，火车不是推的。Kmbox 能干啥请看下面几个例子。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。8

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)
(2) USB 有线键盘鼠标蓝牙化（点我看视频效果）
(3) 改 USB 游戏手柄蓝牙化玩王者荣耀（点我看视频效果）
(4) 无线鼠标控制小车
(5) WIFI 探针（链接）
(6) 天气预报（链接）
(7) 物联网开关控制（链接）
(8) ……

Kmbox 特点

1、 安全
对电脑来说 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。绝不是那些软件宏，驱动宏可以比拟的。从源头上杜绝封号盗号风险。

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。性能绝对杠杠的！
；
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。小白白都能通过简单的 API 调用轻松写硬件外挂。Python 可是公认看看就会的语言呀。

4、 实用

Kmbox 是一个能让普通键盘鼠标化腐朽为神奇的利器。有了他普通键盘数遍可以随意改键，随意控制，鼠标连点，键盘连点，鼠标宏，键盘宏。吃鸡压枪，自动刷怪，一键 N 技能……完完全全控制所有键盘鼠标数据。甚至可以将普通键盘鼠标蓝牙化。无线连接电脑，手机等设备。

5、 霸道

霸道就是 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 后你就能让你的键鼠设备硬件超频。时时刻刻保持 10 倍的速度领先常人。

6、 内

霸气侧漏的同时，你会发现它低调内敛的一面。它拥有 4MB 存储空间，能存放 400 万行逻辑代码，并且支持文件系统管理。它非常适合系统的学习 USB，学习 python，学习软硬件。它可以外接各种外设模块，无限扩展，随你 DIY。你想拿它干啥他就能干啥。

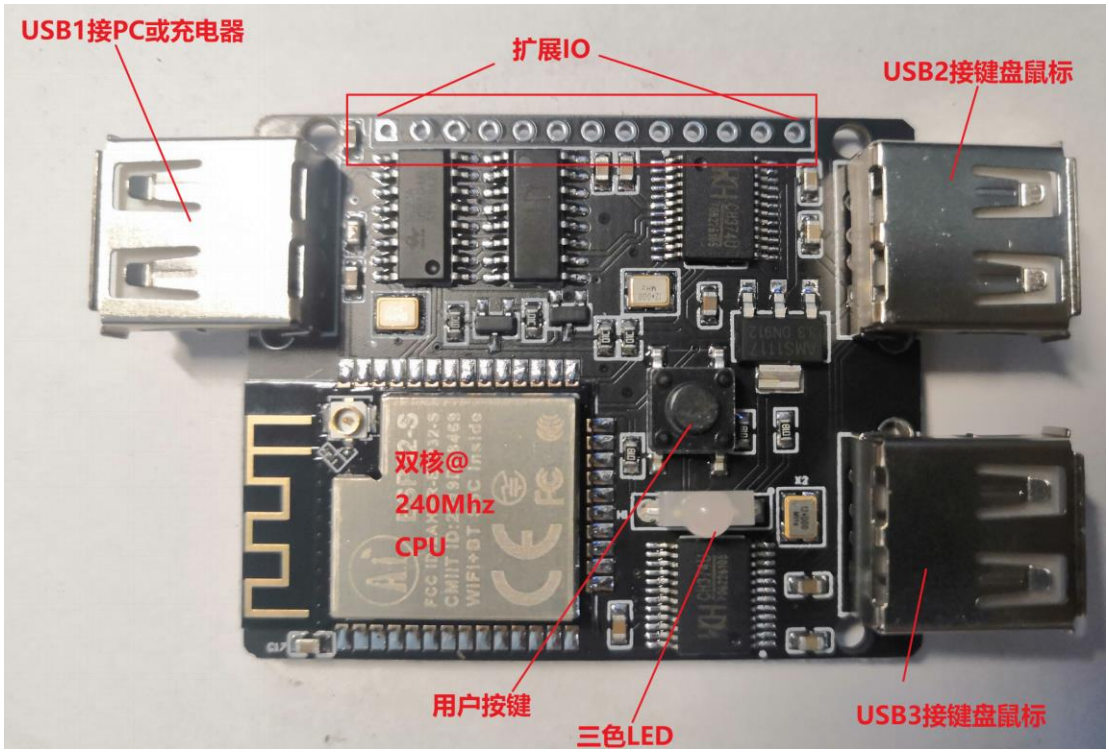
Kmbox 基础教程

这里会介绍 kmbox 的一些基本使用方法。看完本章，你开挂的人生就能开启。废话不多说，直奔主题。所有图片请以实物为准。

拆箱

- 配件清单如下：

物料名称	数量	备注
Kmbox 核心板	1	VerB
USB 延长线	1	长度 1.5 米



Kmbox 核心板

USB 延长线（蓝线黑线二选一，未注明随机发货）

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：
O

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。并不是必须的。但是如果想玩编程和调试就一定得需要。不然代码怎么下载到板子里运行呢？这个驱动只是一个通信口。玩过嵌入式的朋友肯定再熟悉不过了。当安装串口驱动后，可以使用任何串口调试软件登录板卡。
你的开挂人生就从这个串口开始吧。（驱动请去群共享下载 Q 群：190702064）

2、 一个键盘（自动生成）

此键盘是 Kmbox 自动模拟生成的。不要看到模拟二字就以为是假键盘。他就是一个童叟无欺的真键盘。Kmbox 键盘详细规格参数见表一。

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。
。Kmbox 鼠标详细规格参数见表二。
见表二。@

。Kmbox 鼠标详细规格参数见表二。

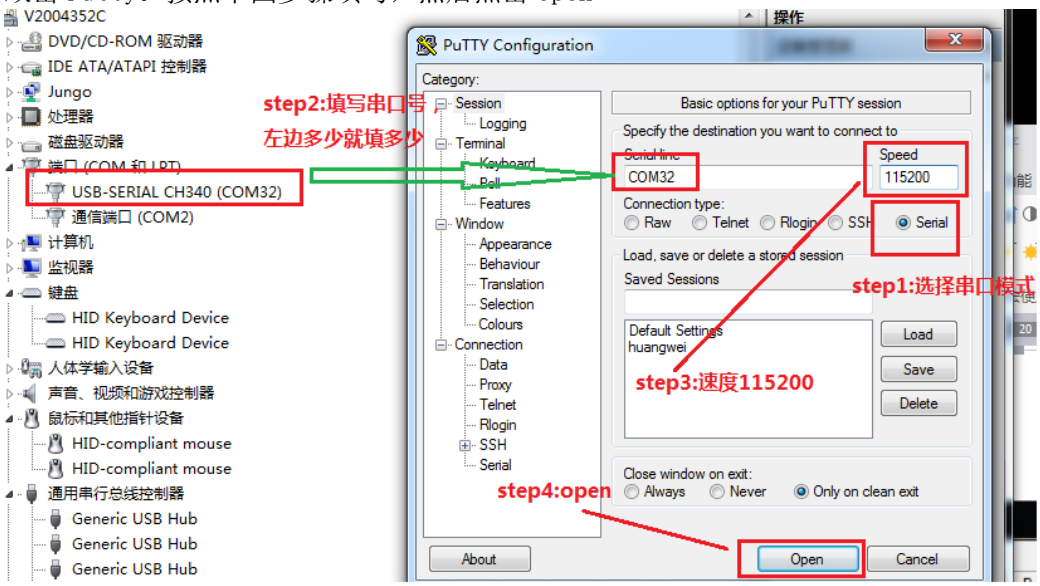
4、一个 HUB 集线器

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。

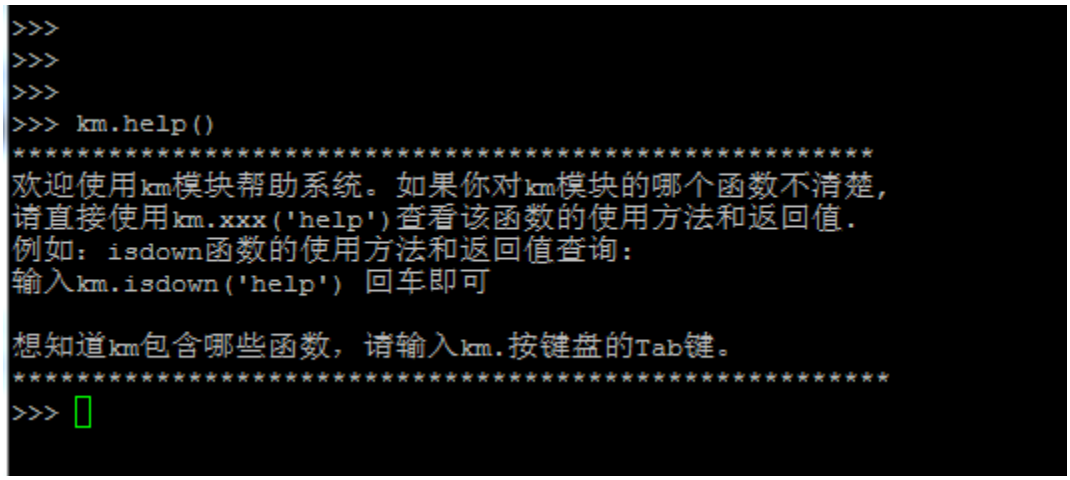
登录 kmbox

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。你熟悉那种就用哪种吧。这里我使用的串口工具是 Putty，才几十 KB 小巧！巴适得很。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



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:



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。如果你从来都没有学习过编程，请不要怕。实话告诉你，Python 真的很简单。真的很简单！真的很简单！不要自己吓唬自己。只要你会敲键盘。你就会编程。如果你有 python 基础，请右上角 X 直接下一章。如果你是小白请接着往下看。

好了，你既然看到这里了。那么给您二鞠躬。热烈欢迎小白白入坑。接下来将用最通俗易懂的方式教你怎么使用 python 脚本控制 kmbox。程序员第一个代码就是 hellow world。那么我们就从 hellow world 开始写起。只需要一行代码：

是不是很简单。其实你会了打印。就能很好知道代码的运行情况。调试代码阶段多加点 print 能提高你代码调试的效率。

前面是一个纯软件的例子，再来个硬核一点的硬件例子。如何用代码控制硬件。看到板子上的绿色 LED 灯了吗？现在就用 python 控制他。一切以实际出发。不搞那些虚的代码。点灯就一个 rgb 函数。

还记得前面说的吗？Python 很简单,只要会敲键盘会读文字就会编程。跟着我!在 putty 的界面输入”(km.rgb ‘help’)“，后回车！先别问为什么，照着敲就是。截图如下：

连接电脑@km

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:

4、 串口（重要）

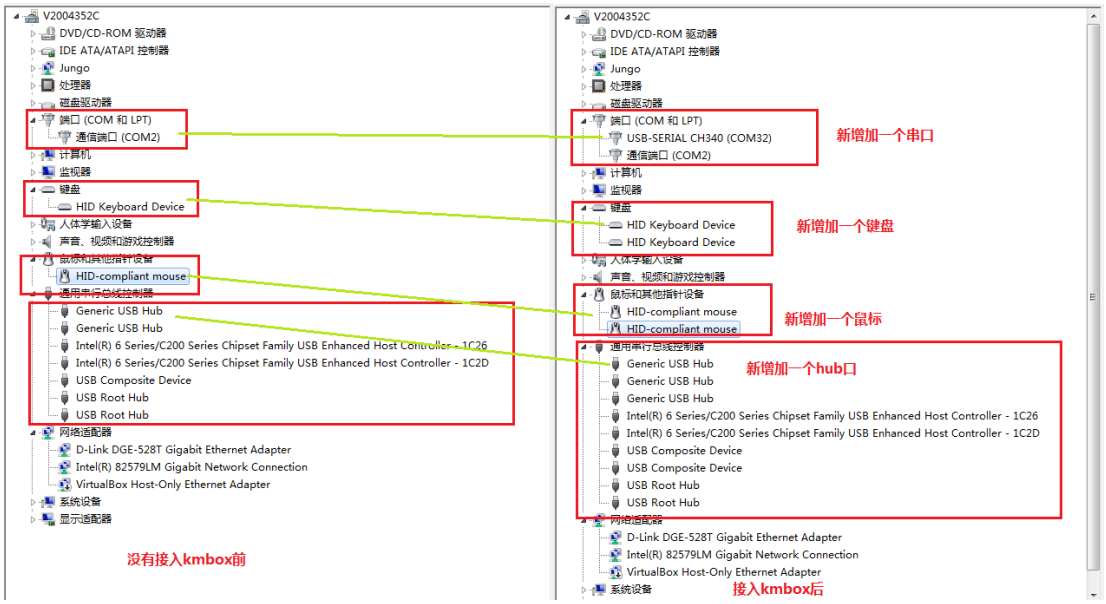
Kmbox 与主机的通信是通过串口完成。如果没有出现串口，请安装串口驱动。串口驱动是用来调试代码,下载脚本的。并不是必须的。但是如果想玩编程和调试就一定得需要。不然代码怎么下载到板子里运行呢？这个驱动只是一个通信口。玩过嵌入式的朋友肯定再熟悉不过了。当安装串口驱动后，可以使用任何串口调试软件登录板卡。

你的开挂人生就从这个串口开始吧。（驱动请去群共享下载 Q 群：190702064）

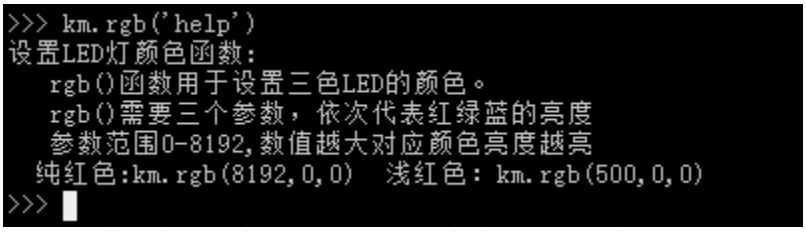
5、 一个键盘（自动生成）

此键盘是 Kmbox 自动模拟生成的。不要看到模拟二字就以为是假键盘。他就是一个童叟无欺的真键盘。Kmbox 键盘详细规格参数见表一。

6、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。



上图是 rgb 函数的帮助文档。已经写的很清楚了。纯红色：Km.rgb(8192,0,0) 就行

这里你可能会问 km 是啥？点是啥？rgb 是啥？小括号是啥？逗号又是啥？没事!!! 程序猿都是这么过来的。我现在简单的说一下，听不懂没关系，反正你只要知道就是这么用就行！用多了你就自然而然的知道了。

“km”是模块。Python 之所以好用是因为有大量丰富的模块。你可以之间按 tab 键可以看到当前有那些模块。

当前已经加载了 gc 模块，km 模块，uos 模块，bdev 模块。

“.”是模块里的函数，比如我们想看看 km 里面有哪些函数可以输入“km.”然后按 tab 键。

```
>>> km.
__class__       Screen          catch_kb        catch_ml
catch_mm        catch_mr        catch_ms1       catch_ms2
click           debug           delay           down
getint          getpos          help            init
isdown          left            lock_ml         lock_mm
lock_mr         lock_ms1        lock_ms2        lock_mx
lock_my         mask            middle          move
moveto          press           reboot          remap
rgb             rgb_free        right           rng
side1           side2           side3           side4
side5           string          table           up
version         wheel           zero
```

Km 模块里的函数都在这里了。有没有找到 rgb 函数?

“rgb”是 km 模块里面的函数。函数都是要打小括号的。别问为什么。因为就是规定。

“,”是函数参数之间的分隔。现在知道让你输入 km.help()是什么意思了吧？就是调用 km 模块 shi 里的 help 函数。同理 km.rgb(100,0,0)想必你也知道了，就是调用 km 模块里的 rgb 函数，把 R 的值设置为 100，G 和 B 的值为 0.这样 LED 灯就变成红色的了。

发散一下思维，绿色，蓝色，紫色，青色等颜色呢？那就改变 RGB 的值呗。大家都懂三原色原理吧。rgb 的范围在 help 中已经说明了是 0-8192. 也许你会有疑问，这个值和亮度有啥关系？这个说来话长了。这里有电路知识，有硬件知识。都是底层的。在嵌入式领域分底层和应用层。每个领域都够你学一辈子的。鄙人不才，已经帮你封装好了 rgb 函数。你只要调用这个 API 就可以。咱只用记住 rgb 函数就能改变灯颜色就行。其他不用管。

强烈建议你开着 putty，跟着教程一个字一个字的敲代码。效果会更好哟！

那么抽象一下，就是只要知道每个函数的作用是什么。那么就能使用好 kmbox。其实现在绝大部分程序猿都是这样。读 API 文档。写测试代码。好比我们买了辆汽车，不用知道轮子是怎么制造的，发动机是怎么工作。读读说明书，只用知道，拧钥匙就能点火，动方向盘就能转弯，挂不同的档就能变速。熟悉这些人机接口就能操作好汽车。老司机们不一定会造车，但一定会开车。同理熟悉这些 API 函数就能用好 kmbox。开车开得好与坏，kmbox 用得牛不牛掰就看你们这些老司机怎么操作这些 API 了。

好了,看完本章你已经一只脚迈进了 python 编程的世界。接下来将的是怎样利用 kmbox 写键盘宏和鼠标宏。在后面的章节中你会逐步了解到 kmbox 的所有 API。笔者会采用一遍教学一边实践的方式介绍玩转这些 API。一边玩一边学。都是实实在在的代码。看得见摸得着的效果。边学边玩，其乐无穷呀！

键盘宏编程

各位老铁们，欢迎来到本章。别拿到板子满脑子就想整个牛逼的外挂。咱先冷静冷静。先来讲一下键盘的基本常识。不要小看理论知识。这些基础的理论能让你后面少走很多弯路。

键盘小常识

首先科普一个小常识，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（划重点）。这是 USB 系统的最基本特点。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：

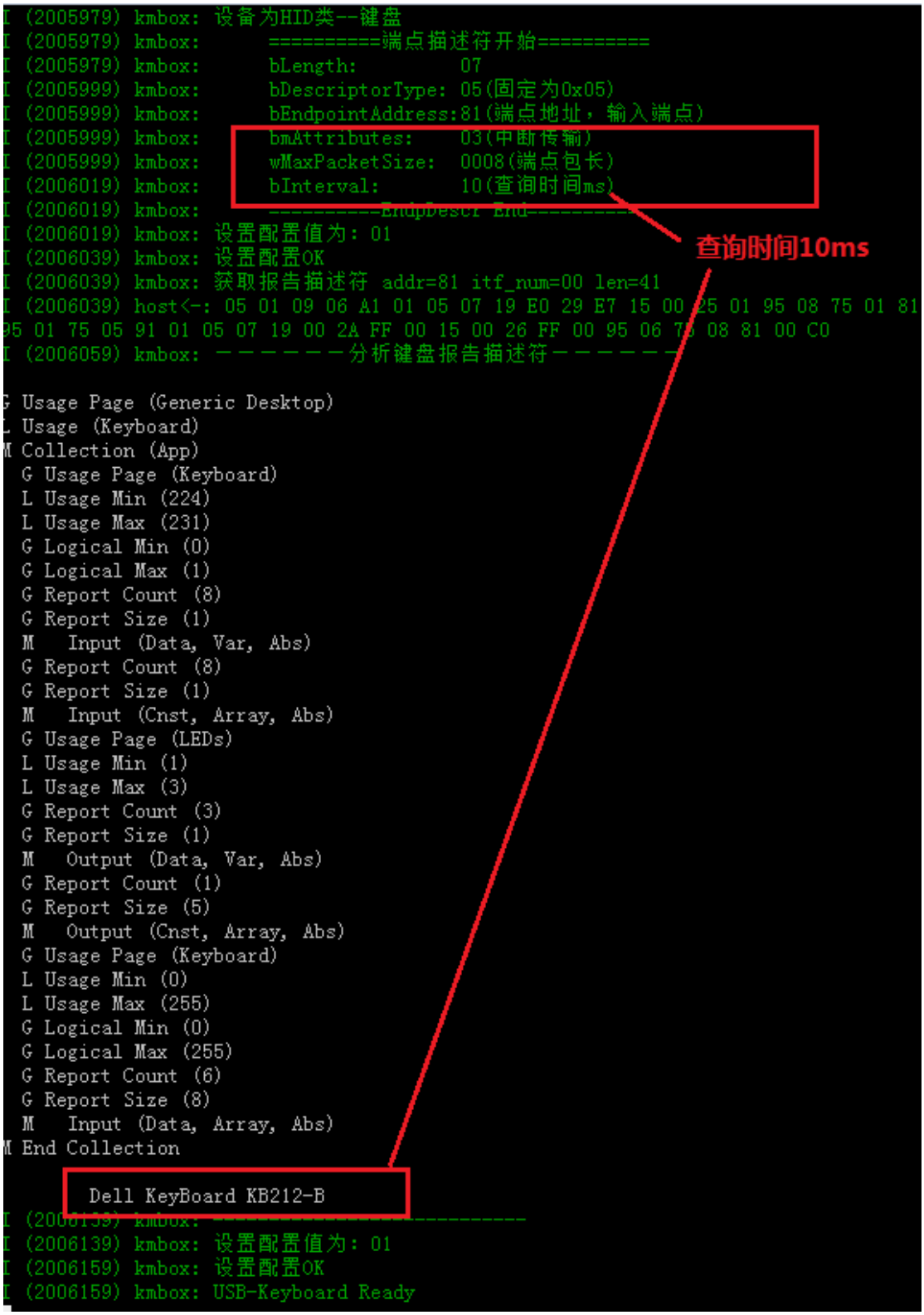
主机：键盘你有按键吗？
键盘：没有！
过了一会儿....（主机等待一个轮询时间）
主机：键盘你有按键吗？
键盘：没有！
过了一会儿....（主机等待一个轮询时间）
主机：键盘你有按键吗？
键盘：我按下了 123456789 这几个按键
.....

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。那么你可能会有疑问。主机什么时候读键盘数据呢？多久读一次呢？我直接告诉你答案，在 USB 设备的配置描述符中会定义这个值。上面的“过了一会儿....”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。记住，这是一般的常规键鼠的报告时间。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。因为你想想，键盘鼠标是物理设备。你可以一秒钟按 50 次键盘或者 50 次鼠标吗？不行吧！如果主机不停问，但是 USB 设备一直回答没有没有。那也是在浪费主机的时间。从这里你应该可以知道，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。
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,我们的 kmbox 是 1ms。性能是他们的 10 倍。

二、 报告长度。
上面的我按下了 123456789 就是报告长度。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（毕竟咱有 10 个手指头）。123456789 一包就能发送完成。节约时间。（PS：如果愿意可以把 kmbox 做到 108 键无冲都行。但是没那个必要呀）

想不想知道你现在用的键盘报告间隔是多久？报告格式是怎样的？很简单。把你的键盘接到 kmbox 的 USB2 或者 USB3 端口。Kmbox 会根据设备描述符等参数自动计算出来，并显示在串口控制台内部：



如图所示，我现在用的戴尔 KB212-B 键盘。查询时间是 10ms。所以 kmbox 会保证每隔 10ms 去读一遍键盘的报告数据。有人会问。主机能读快点吗？能，肯定可以。但是你要记住一点。主机能读是主机的事，设备应答不应答是设备的事。设备已经在描述符中告诉主机，你丫的最好是 10ms 来问我要一次数据，言外之意就是：你要是来早了对不起我很忙，你的请求太快，我处理不过来。你要是来晚了，哟嚯上次的数据有可能被冲刷掉。所以。做主机的还是按照设备的要求让你什么时候来读就什么时候来读吧。想知道 kmbox 的键盘特性点这里跳转。

PS:如果你的键盘或者鼠标接到 kmbox 上无法使用，请戳这里，我可以很自信的告诉你，就没有 kmbox 搞不定的键盘鼠标。

另外出现上面的打印也不是一定有问题。Kmbox 对不识别的键盘鼠标默认当做常规键鼠操作。如果使用没问题可不用理会这条打印。

Kmbox 如何处理键盘数据

看完了通用的 USB 主从设备之间的关系，我们再来看看 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。如下图所示：

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。从上面的过程中可以看出。
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。试想一下，如果在 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，kmbox 有就等效于你的键盘鼠标就有。

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 内部写一个这样的脚本。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。PC 才不管这些。因为他只认 HID 报文。而 HID 报文正好是开火，鼠标下移。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。 到这里你可能会有疑问。我哪里知道怎么改底层 HID 数据？别急，不慌，问题不大。Kmbox 已经封装好这些功能。你只要会调用就行。不用管底层如何实现的。说到这里，才正式开始我们今天的主要内容。键盘宏函数啦。

以下内容边敲代码边看口味更佳哟。你不用一次性搞懂所有函数。用到哪个再查哪个也一样。但前提是你得知道有这个东西才能查。请务必初略的浏览一下每个函数。知道他们是做什么用的。后面自己写脚本就有思路知道怎么写了。以下的函数用途说明也直接可以在代码中输入 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               1               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：2
空格键名字叫”space”,对应的编码值为 44。
回车键名字叫”enter”，对应 table()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。

点这里查看全键值对应表。

aaaaaaaaaaaaaa

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')
>>> aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

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)
>>> aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

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
>>>
```

aaaaaaaaaaaaaaaaaaa

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')
>>>
```

22、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 ctr-l（左 ctrl）+alt-l(左 alt)+delete 键的抬起。你可以这么写：
方法一：km.multidup(‘ctr-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’):

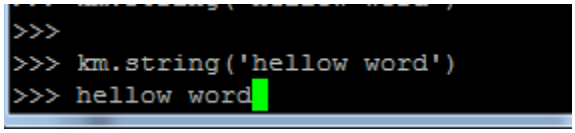
```
>>> km.press('help')
软件强制单击指定按键:
  press()包括按下和松开两个动作,例如单击一次键盘a写法如下:
  一:km.press('a') 输入参数是字符串a
  二:km.press(4)   输入参数是a的键值
  press支持模拟手动操作，即控制一次按键按下的时间，press可以额外增加一个或者两个参数
  press(4,50)------两个参数，表示单击a键，a键按下时间50ms
  press(4,50,100)--三个参数，表示单击a键，a键按下时间50-100ms的随机值
PS:推荐使用键值类型，效率更高,速度更快，
>>>
```

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：



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）

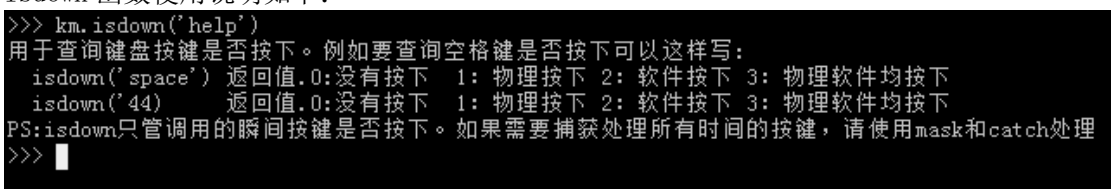
isdown(value)函数用于检测指定的 value 按键是否按下。value 可以是数字类型或者字符串类型（数值类型效率高），value 的值请参考 km.table()函数。这个函数主要用在逻辑判断中，用于判断指定的按键是否被按下。
例如，想检测键盘上的’a’键是否按下。可以有下面两种写法：

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：



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
>>>
```

那么既生瑜何生亮呢？有 table 就没必要用 getint 呀。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按着不松。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。

catch_kb 的用法详见 catch_kb 的介绍。

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

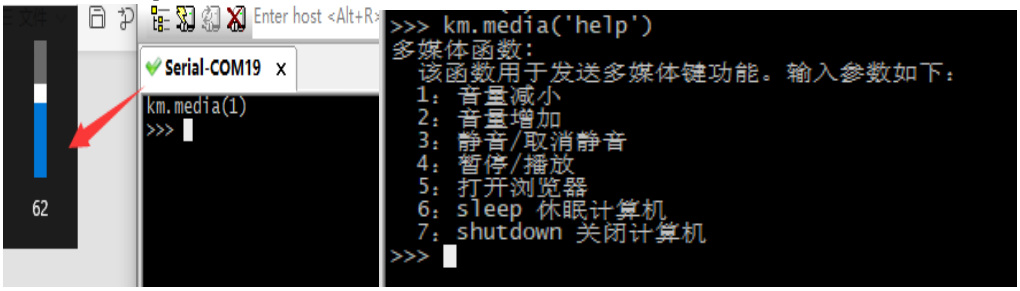
init 函数相当于恢复最初什么都没操作，键盘鼠标什么都没按的状态。因为你可能写脚本会屏蔽很多按键。当你想释放所有脚本的的键盘鼠标按键时可以执行该函数。避免一个一个的还原原来的屏蔽。

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)
>>>
```

你们不要打我哈)

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,需要自己做处理。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！

鼠标宏编程

经过前面几章的学习，想必现在对键盘应该有了新的认识了，本节开始学习鼠标宏的相关知识。首先看看 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。咱们的 kmbox 是 1ms. 10ms 我们能执行 10 条指令。
按键数：普通鼠标只有左中右一共三个。咱们的 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 是占两字节。范围广度是普通鼠标的 256 倍。

如果这些参数你看不出优势。那我就举个简单的例子。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。咱们的 kmbox 是 X 步进是 32767. 同样的移动，只需要一包数据。一次传输。一共耗时 1ms。这下你该知道什么叫硬件差距吧。

想知道目前你的鼠标报告间隔是多久，鼠标报告是什么样的吗？直接怼 kmbox 的 USB 口吧。Kmbox 会自动检测出来。鼠标无法识别请看这里。

好了，牛逼不是吹的。火车不是推的。接下来开始今天的 kmbox 鼠标宏函数的学习。请务必走读一遍，知道这些函数的功能。这样编程时才有思路。咱从上到下一个一个的看。

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。不然检测空气吗？

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

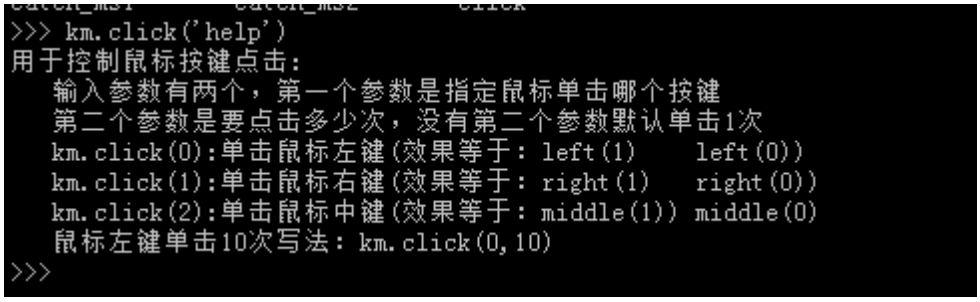
middle 函数用于软件控制鼠标中键。鼠标中键就两个状态，按下和弹起。定义按下是 1，弹起是 0。不给参数时是查询鼠标中键的状态：
0：弹起
1：物理按下
2：软件按下
3：物理软件均按下
当给参数时是设置鼠标中键的状态：
0：鼠标中键弹起
1：鼠标中键按下

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



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。避免你写脚本时太过于底层模拟。另外第二个参数是单击次数。如果你想双击。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

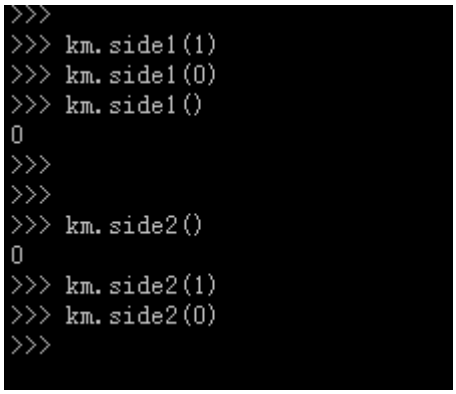
wheel 函数用于控制鼠标滚轮。滚轮可以上下移动，输入参数为滚轮滚动单位值，上移动为正，下移动为负。例如滚轮上移动 10 个单位。和向下移动 10 个单位截图如下：

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

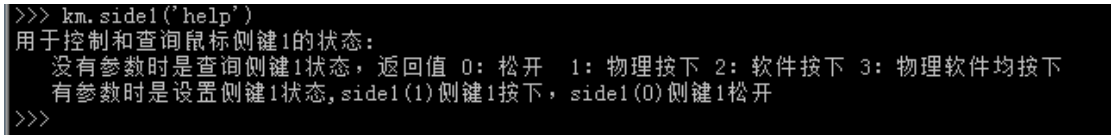
这些函数是昆明。用于控制鼠标的侧键状态。输入参数可有可无。无参数是查询侧键状态：
0：侧键弹起
1：侧键物理按下
2：侧键软件按下
3：侧键物理软件均按下

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：

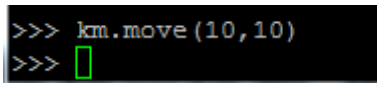


Ps： side1 和 side2 一般是翻页功能，就算你的鼠标没有侧键也没关系。Kmbox 有你就有。

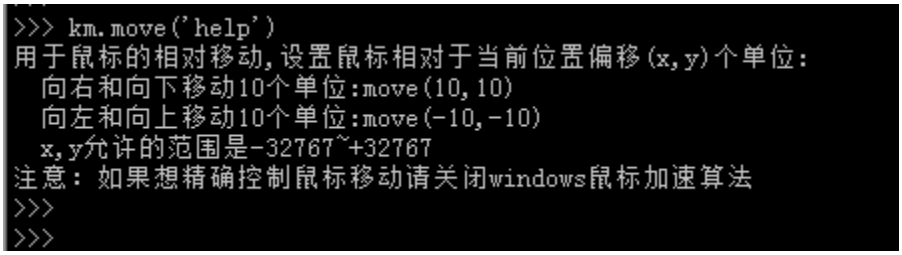


move 鼠标移动控制函数

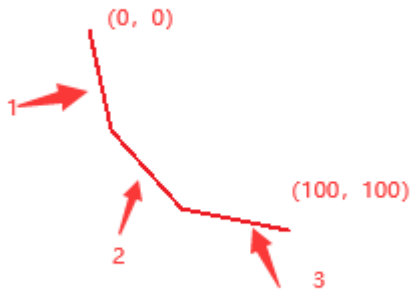
move 用来控制鼠标相对移动，输入参数(x,y)是数值类型。范围为-32767 到+32767。并且规定 x 向右为正，向左为负。Y 向下为正，向上为负。例如让鼠标向右和向下移动 10 个单位，可以这样写：



你就能看到鼠标自动按照这个调指令移动了。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。

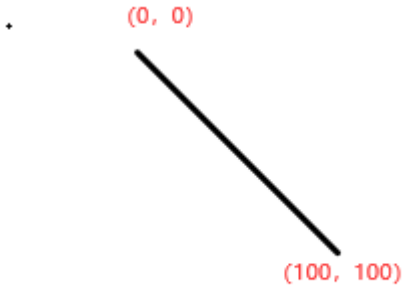


以上是 move 函数的基本用法，move 函数如果只给两个参数，那么鼠标移动将在 1ms 内完成。没有中间过程。实际效果是从当前点阶跃移动到对应的偏移点。这种效率最高，速度最快。如果你需要模拟人工轨迹移动，你可以个 move 函数再加上一个参数，这个参数是用来约束从起点到终点你期望用几条直线来拟合。由常识可知，第三个参数数字越大，拟合出来的曲线就会越平滑，如下图所示：分别用 1 条直线，3 条直线，和 1

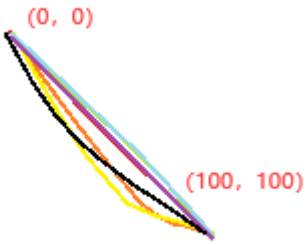


km.move(100,100,3)三段直线随机拟

0 条直线拟合从（0，0）到（100，100）的移动。



km.move(100,100)直接移动



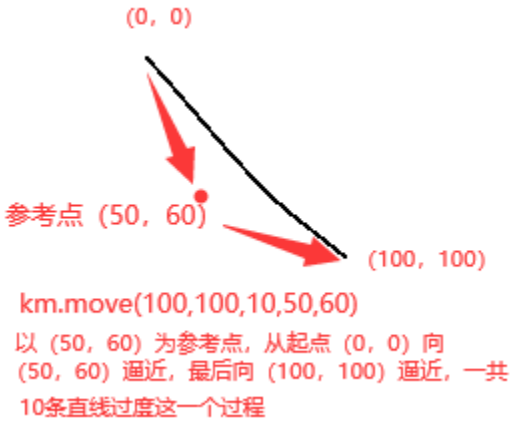
km.move(100,100,10) 10段曲线随机拟

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)



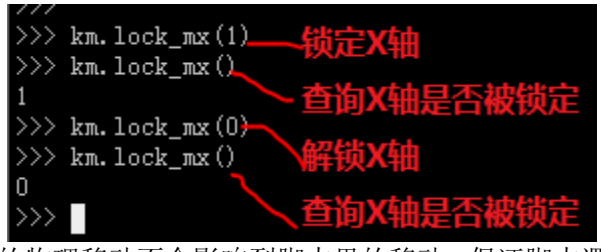
其中的（50，60）就是参考点坐标。kmbox 会自动按照参考点来拟合起点到终点。给定参考点后，kmbox 的轨迹就是唯一确定的。曲线走向怎样，扭曲程度怎样都是由这个参考点决定的。

lock_mx 鼠标 x 方向屏蔽函数

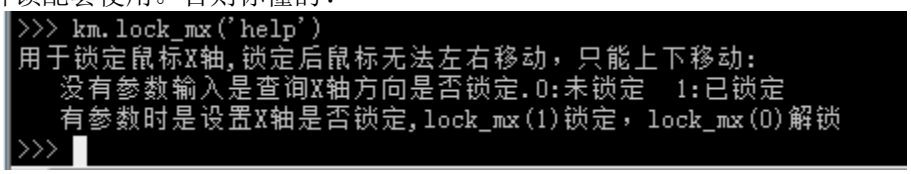
Lock_mx(lock mouse X)函数用于屏蔽鼠标 x 轴方向的物理输入，多用于确保软件 x 方向移动不受鼠标物理移动的影响。
输入参数为 1 时是锁定 x 轴方向：
当 X 轴方向被锁定以后，鼠标左右移动将不起作用，只能上下移动。
输入参数为 0 时是解锁 x 轴方向的锁定。

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：



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。否则你懂的！



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

有了前面的 lock_mx。Lock_my 就请你自己体验一下吧。

```
>>> 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。否则你懂的！

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。否则你懂的！
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。否则你懂的！
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。否则你懂的！
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。否则你懂的！
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：

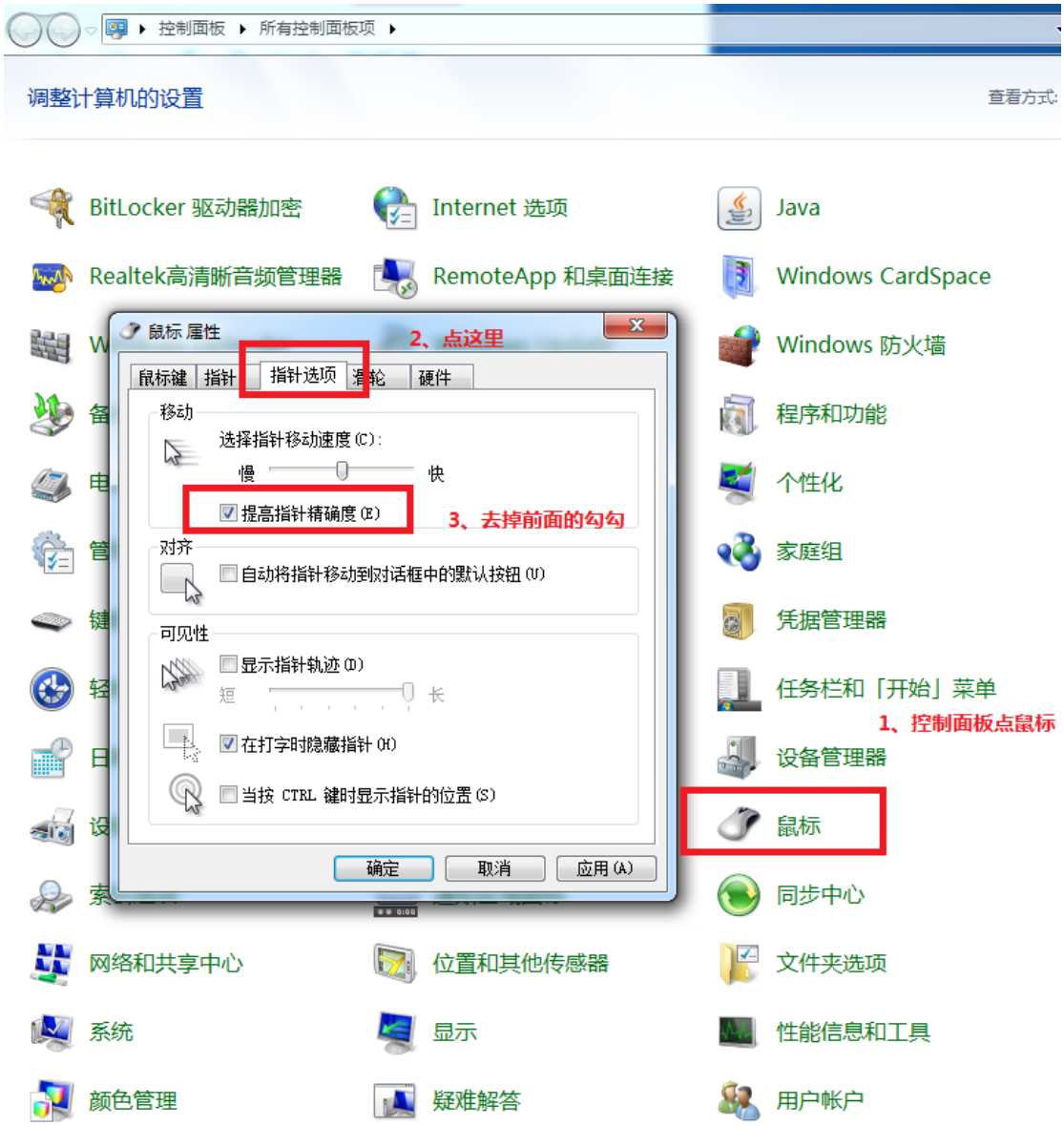
我的显示器是 1920x1200.那么直接将允许移动范围设置为 Screen(1920,1200)

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。不然也会照成误差。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]
>>>
```

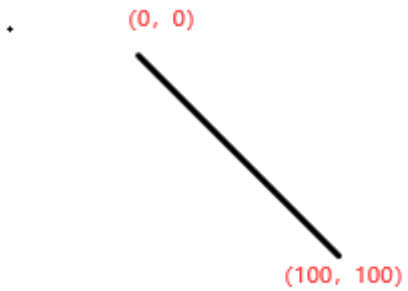
moveto 鼠标精确移动函数

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：

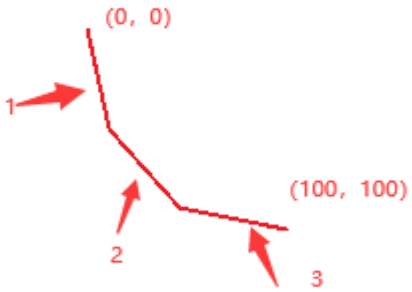
```
>>> km.moveto('help')
用于鼠标的绝对移动，将鼠标移动到绝对坐标(x,y)处:
  此函数需要先调用zero(1).且规定屏幕左上角为坐标原点(0,0)
  推荐使用Screen(x,y)设定屏幕的分辨率大小,这样移动更准确
  moveto(100,200)表示移动到绝对坐标(100,200)处
  x,y的范围为0-32767，不存在负坐标
注意：请关闭windows鼠标加速算法
>>>
```

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。

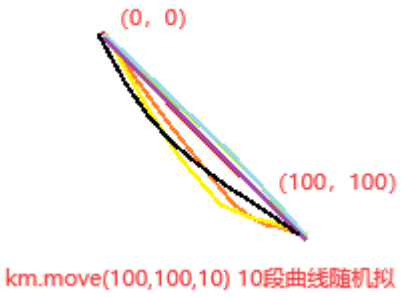
以上是 moveto 函数的基本用法，moveto 函数如果只给两个参数，那么鼠标移动将在 1ms 内完成。没有中间过程。实际效果是从当前点阶跃移动到对应的目标点。这种效率最高，速度最快。如果你需要模拟人工轨迹移动，你可以个 moveto 函数再加上一个参数，这个参数是用来约束从起点到终点你期望用几条直线来拟合。由常识可知，第三个参数数字越大，拟合出来的曲线就会越平滑，如下图所示：分别用 1 条直线，3 条直线，和 10 条直线拟合从（0，0）到（100，100）的移动。



km.move(100,100)直接移动



km.move(100,100,3)三段直线随机拟

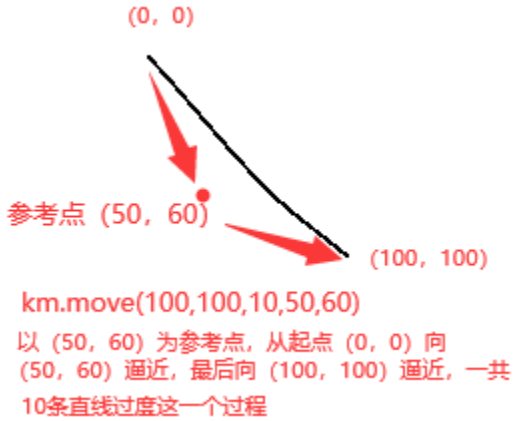


上图是鼠标轨迹。你可以发现，拟合点数越大，起点到终点的过度曲线就越平滑。所以。如果想控制鼠标模拟人操作，可以给 move 加上第三个参数。可以将阶跃状态改变成连续的状态。

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。

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

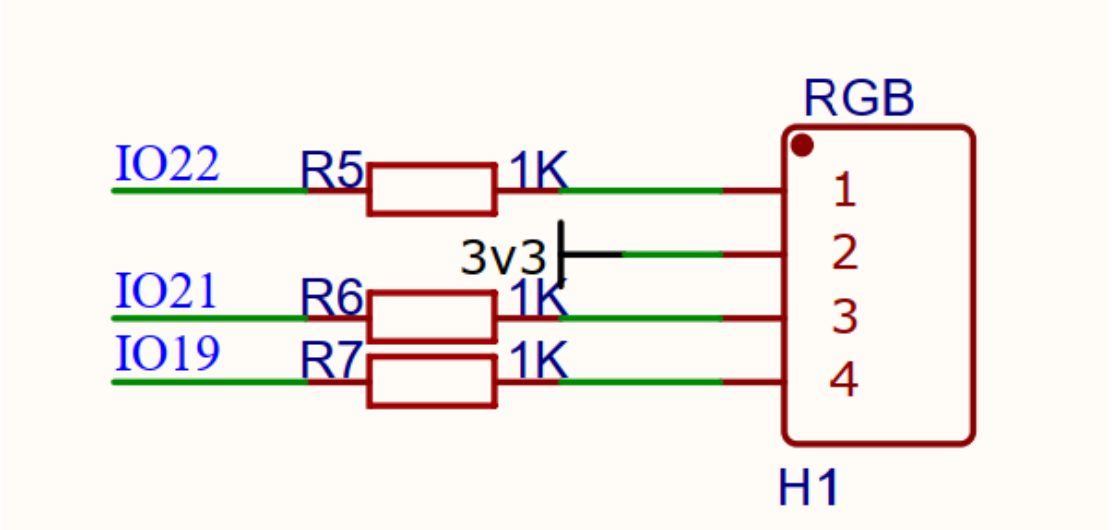
```
>>> km.reboot('help')
软复位重启板卡
  km.reboot()会在3秒后软件复位重启板卡
>>>
```

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()
>>>
```



RGB 引脚控制 IO 如上图所示。

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。

用户按键

Kmbox 上有一个微动开关。这个开关有两个作用：

一：在升级固件时，如果没有自动下载可以长按此按键。（一般不需要）

二：当程序正常运行后，此按键的行为完全由用户自己的脚本决定。按键对应 I00。按下后为低电平。I0 的基本操作请参考这个脚本：

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。但是咱们的 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
_onewire          framebuf          re                upip
_thread           gc                select            upip_utarfile
_webrepl          hashlib           socket            urandom
apa106            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
>>>
```

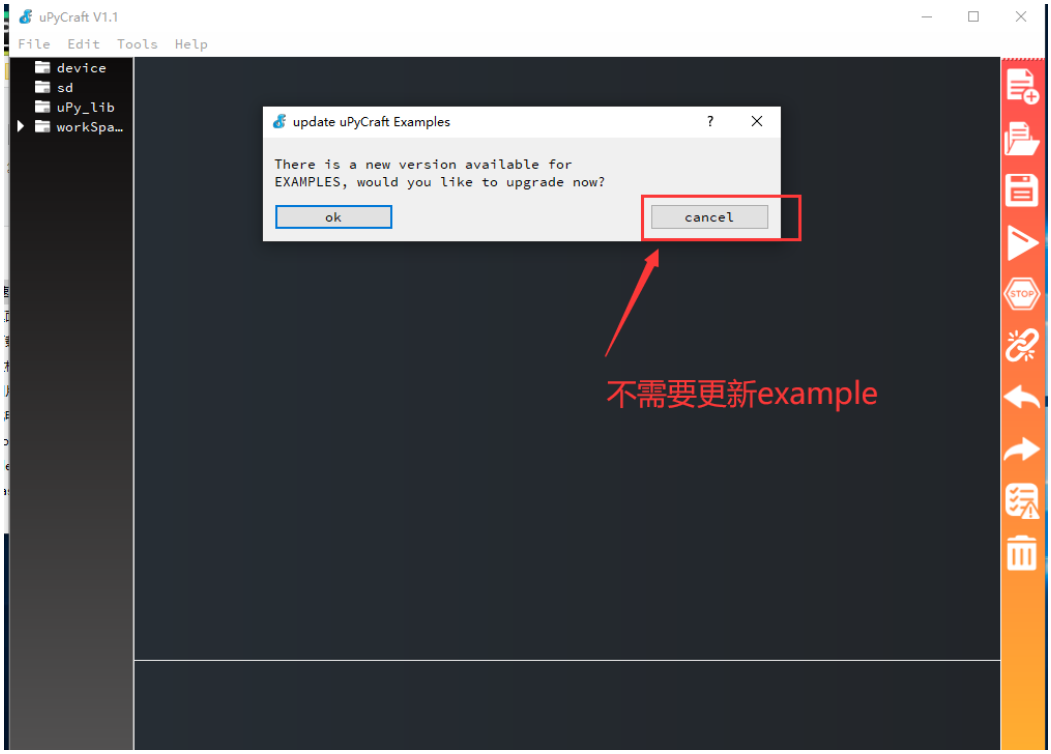
其中 km 仅仅是众多模块中的一个。我也不可能每个模块每个函数一个一个的讲。有了前面 km 模块的学习。其他模块也是以此类推。这里请你自己去探索了。你可以用 kmbox 驱动硬件，也可以用它来写软件算法。DIY 各种电子产品都非常合适。

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。从理论到实践的过度。

写脚本其实很简单，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。

他是一个专门的代码编辑器和下载器。他也可以用于调试。这个小工具十几 MB，能满足你 90%的开发工作。双击 upycraft.会出现这个应用界面。

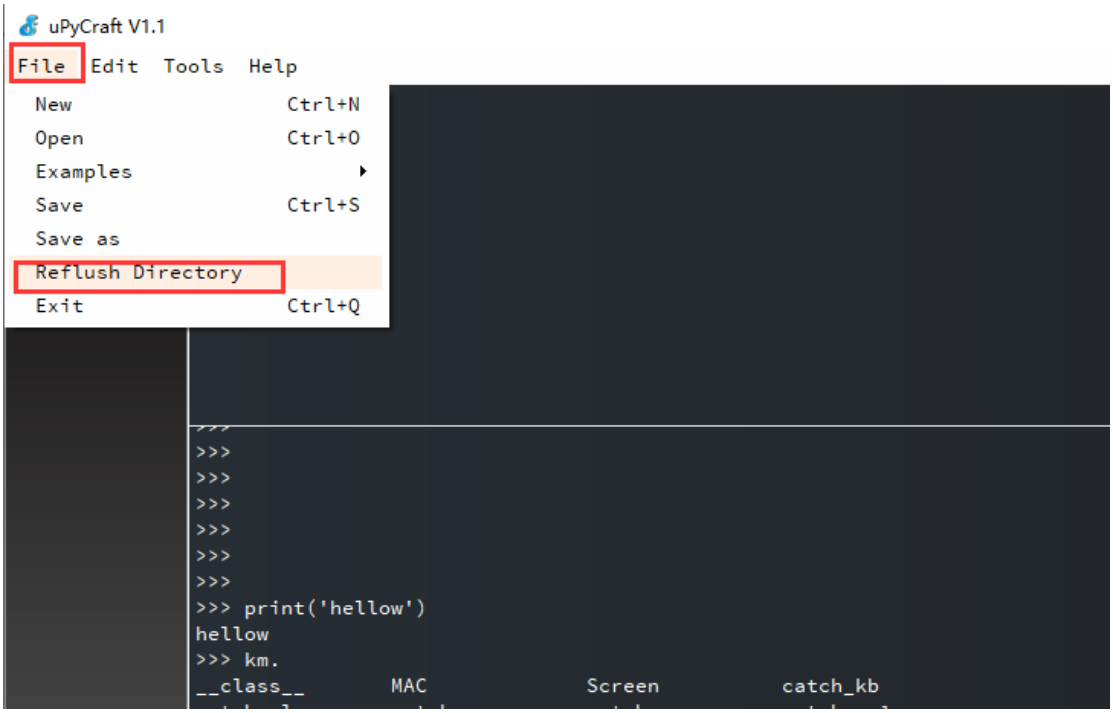


打开后需要连接开发板：依次 tools -->serial-->COMxx 连接到开发板

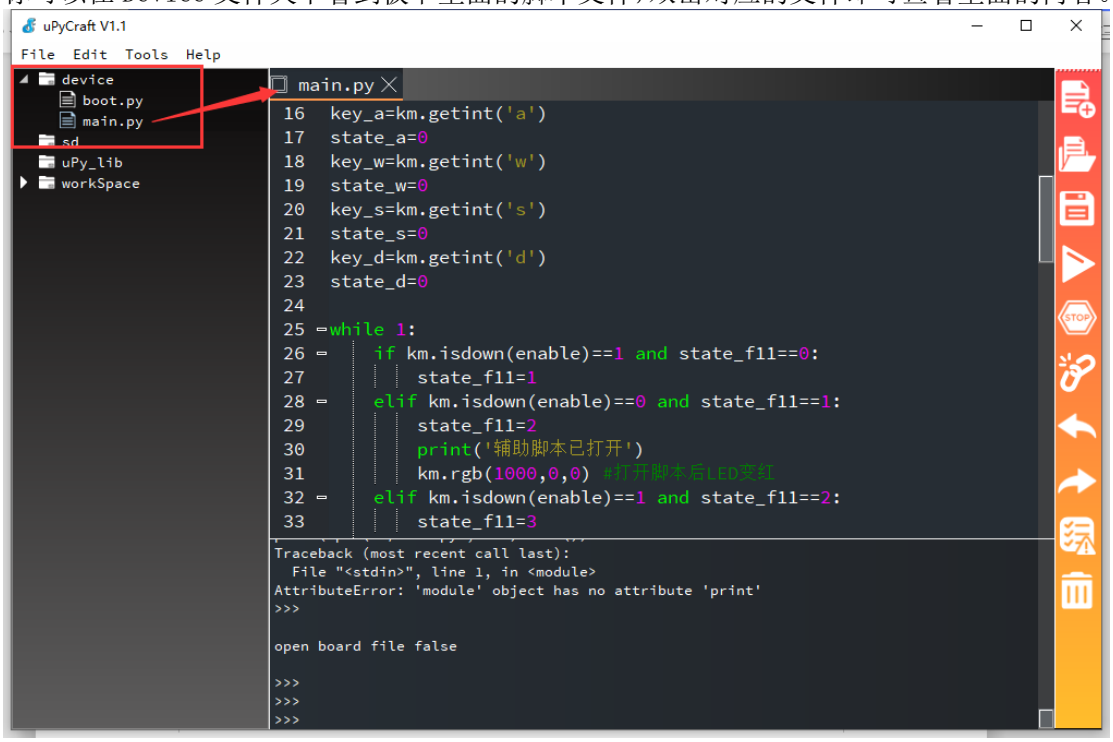
如果连接失败请检测串口驱动。连接后是这样的。这里其实就是一个 putty 的控制台。

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（mpy 脚本无法读取）



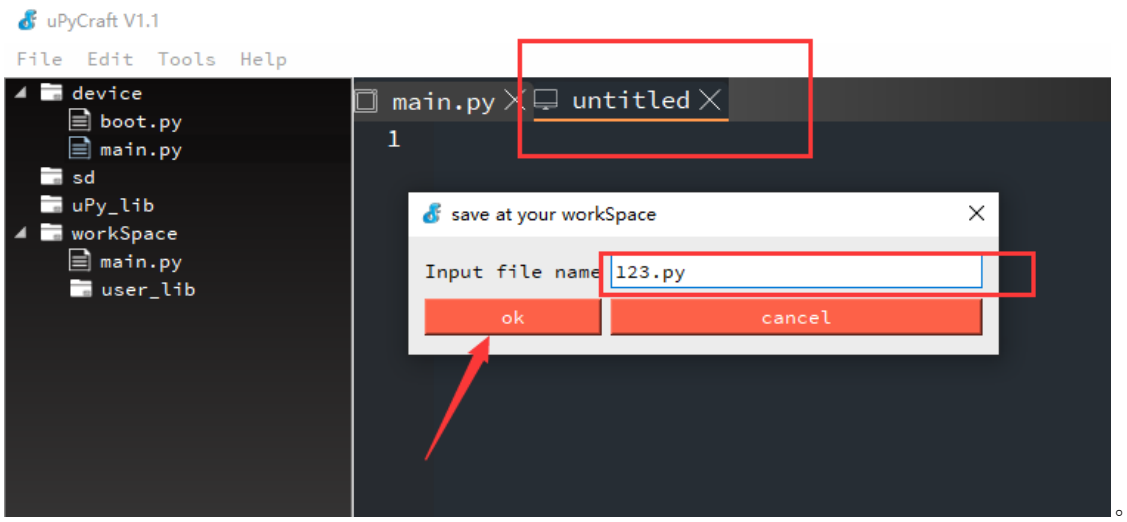
你可以在 Device 文件夹下看到板卡里面的脚本文件，双击对应的文件即可查看里面的内容。



修改，保存，重命名都可以。

新建脚本 3

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 保存文件时给他娶个名字，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



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。Kmbox 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 kmbox 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。

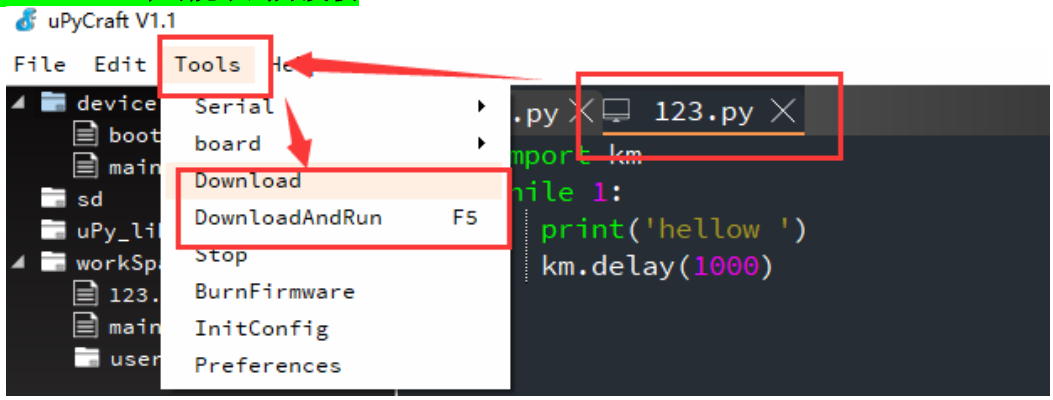
方式二：

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。

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。Mpy 的烧写详见：用户使用加密文件

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/

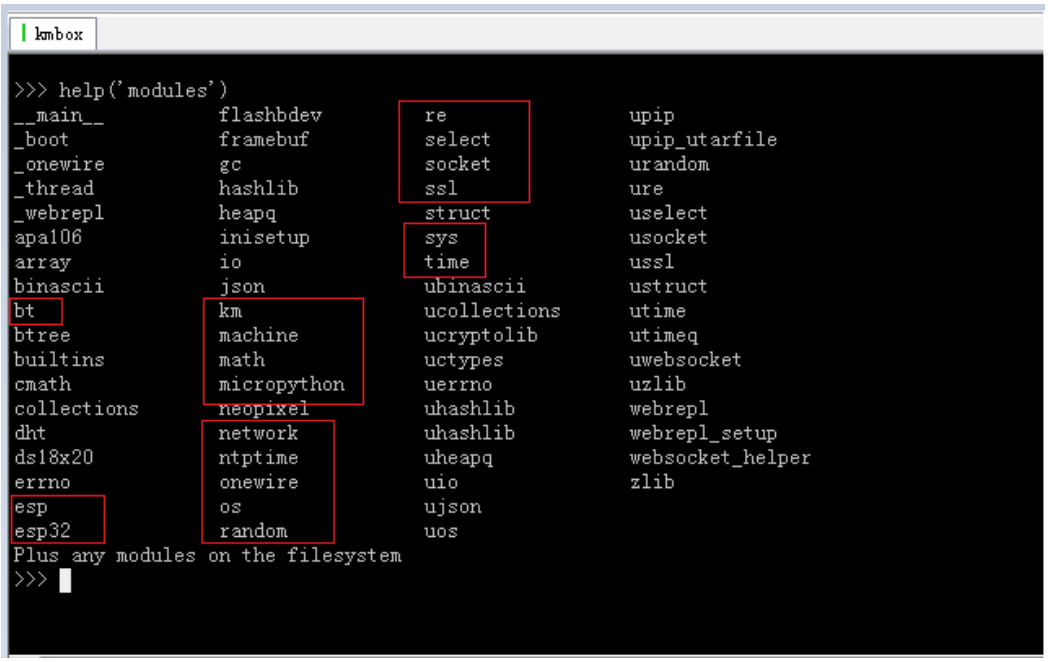
脚本文件请去群里下载。

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？

默认模块

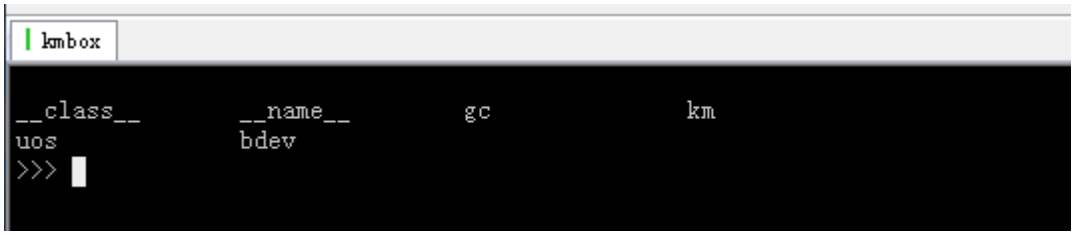
来吧，开始攻克这个问题。俗话说知己知彼百战不殆。要引入其他 module，首先看看自己已经包含哪些 module.你可以在 repl 界面里输入以下指令查看目前 kmbox 内部默认包含哪些模块：



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 。如下图所示：

开机默认只加载 gc(垃圾管理)模块，km(键鼠控制)模块，uos(文件系统管理)模块，bdev（文件管理）模块。你按 tab 后截图如下：



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。

```
kmbox
MicroPython v1.11-265-g12f13ee63-dirty on 2020-06-01; Kmbox with ESP32
www.clion.top

Type "help()" for more information.
>>>
>>>
>>>
>>>
__class__       __name__        gc              km
uos             bdev
>>> import machine
>>>
__class__       __name__        gc              km
machine         uos             bdev
>>>
```

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.
```

那么问题来了。如果你确实要引入一些不在默认库里的东西。是不是 kmbox 就不行了呢？非也非也！引入其他 module 有一下几种方法。今天我们就来一个稍微复制点的工程。
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：

小学生作文，看图说话：
第一步：与 x 轴方向夹角 72°，画一条长度为 L 的线段。
第二步：向右偏转 144°，画一条长度为 L 的线段。
第三步：向右偏转 144°，画一条长度为 L 的线段。
第四步：向右偏转 144°，画一条长度为 L 的线段。
第五步：向右偏转 144°，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。484 很简单？那如何用鼠标宏实现呢。请直接看下面的代码：

```
"""
鼠标左键单击绘制五角星
mp_turtle是海龟绘图模块。移植标准的turble模块
"""
from mp_turtle import *

while 1:
    if m.left()==1:#如果鼠标左键按下
        home()      #将当前鼠标点设置为坐标原点
        left(72)    #龟头72度 准备画图
        for i in range(5): #循环5次画边
            forward(100)    #绘制长度为100的边
            right(144)      #右转144度
            m.delay(100)    #延时100ms，避免绘制过快您看不清过程

        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
            m.delay(100)    #空等延时
```

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，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

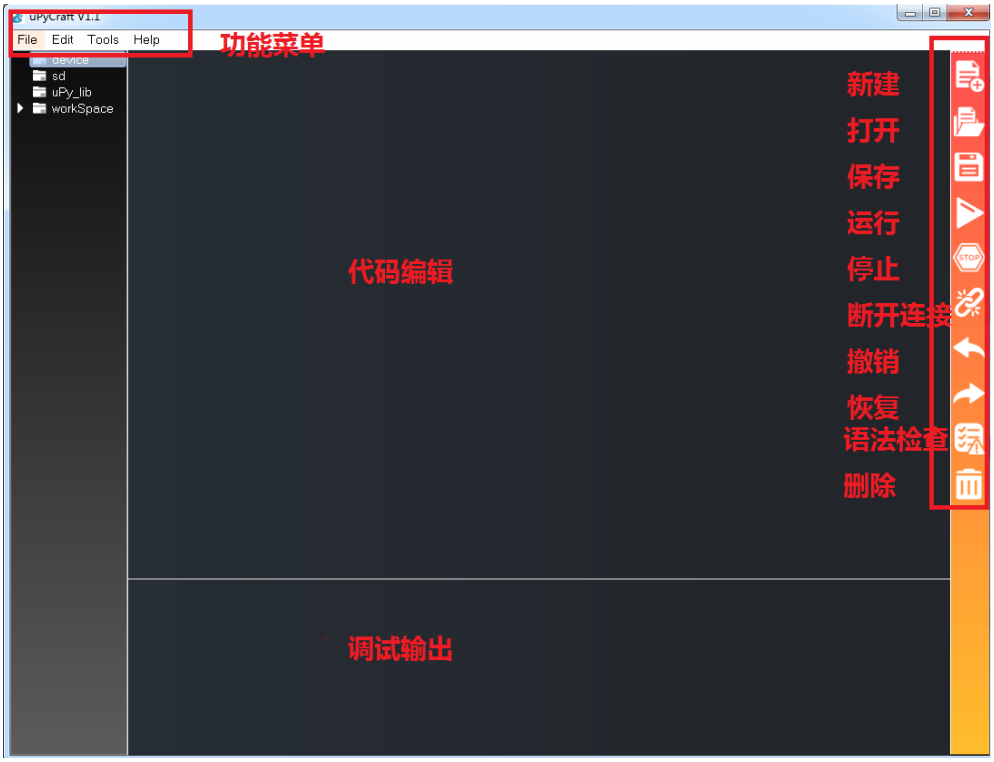
点我看视频点我看视频

您可能纳闷了，home 是啥？Left 是啥?forward 是啥？right 是啥？Km 里好像也没有这些函数呀？？？对！确实没有。以上没见过的函数是在 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。今天这讲主要是学习如何使用多个脚本文件。

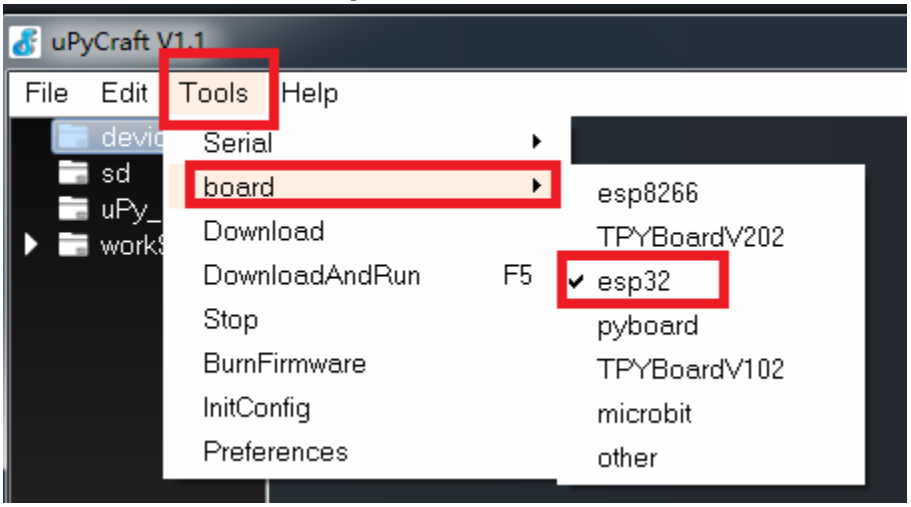
文件存储的方式（入门级）

通俗点讲，文件存储就是将你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。那么怎么把 py 文件保存到 kmbox 内部呢？你可以使用以下工具。

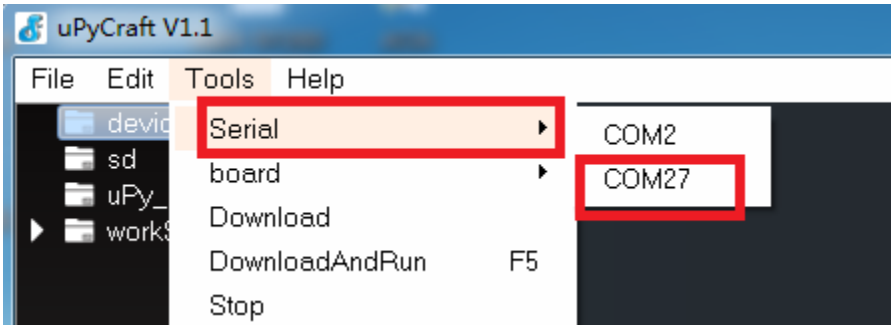
1. Upycraft
   这是一个小巧的代码编辑和调试器，专门为 micropython 设计使用。下载请去群共享。



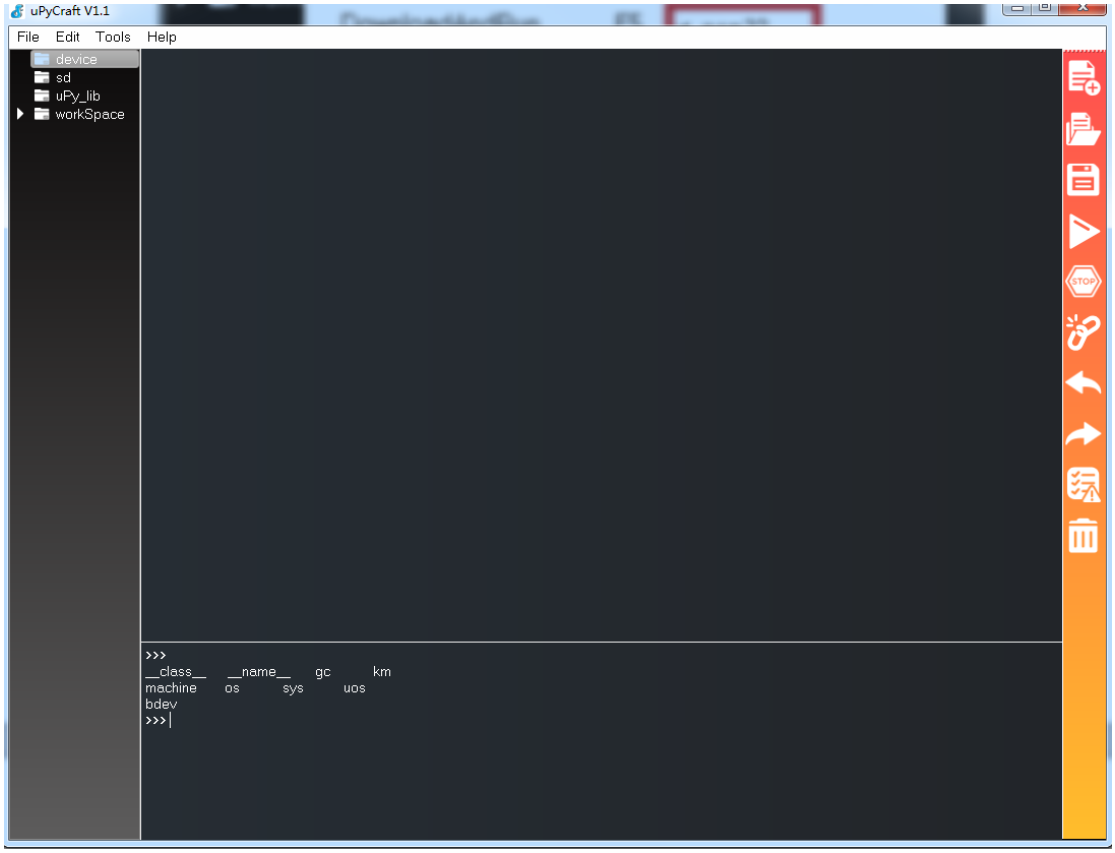
我们以 kmbox 为例，介绍一下怎么使用以及如何传输文件。打开软件后在 tool 菜单下将 board 标签里选择板子芯片型号为 esp32. 如下图：



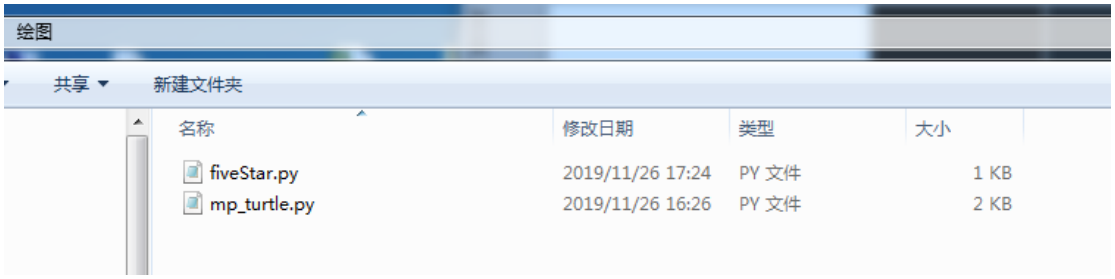
然后将串口号选择 kmbox 实际的串口号即可。



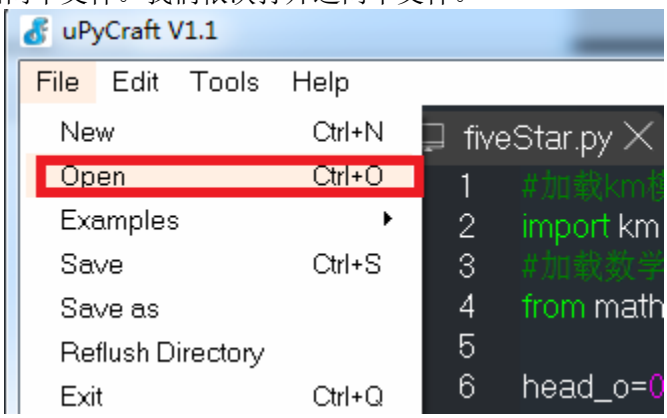
设置完毕后你就能在调试窗口进行之前的 repl 操作了（和前面的 putty 一样）。



那么如何导入第三方的文件或者模块呢？下面举例。例如你在网上找到一个绘图功能的模块。

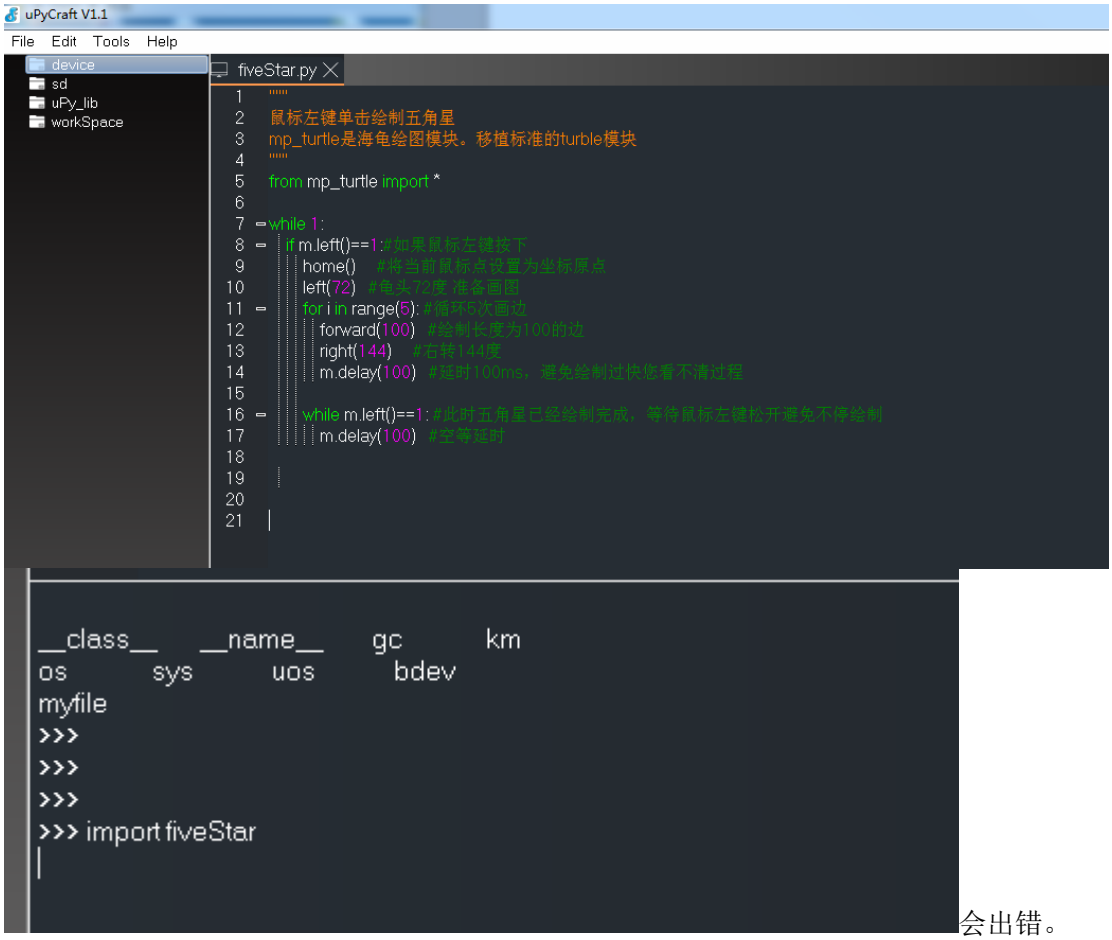


这个绘图模块包括两个文件。我们依次打开这两个文件。

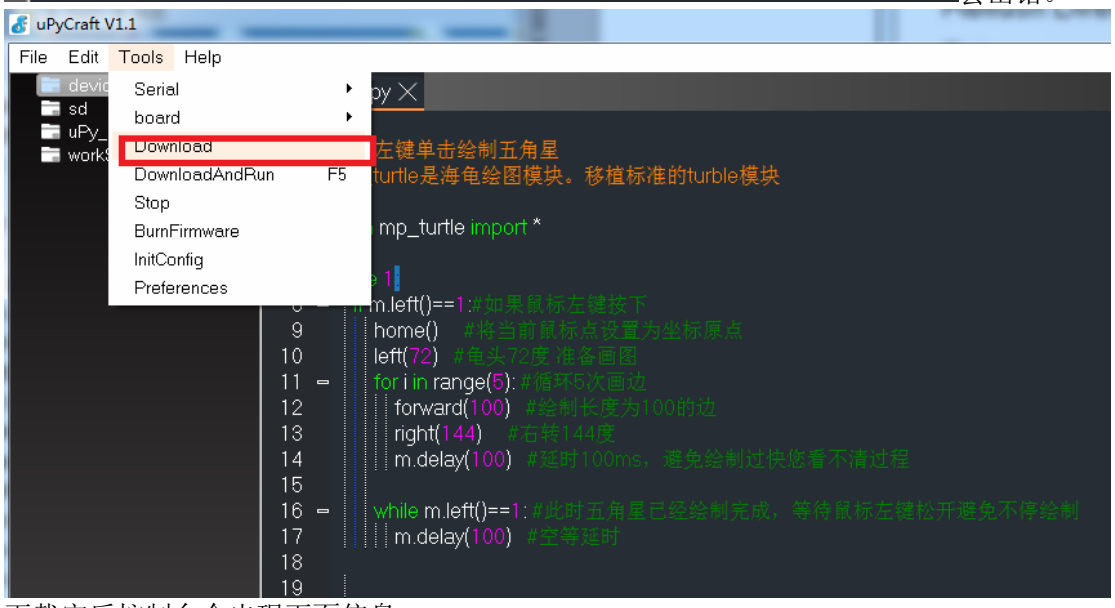


打开后如下图所示：

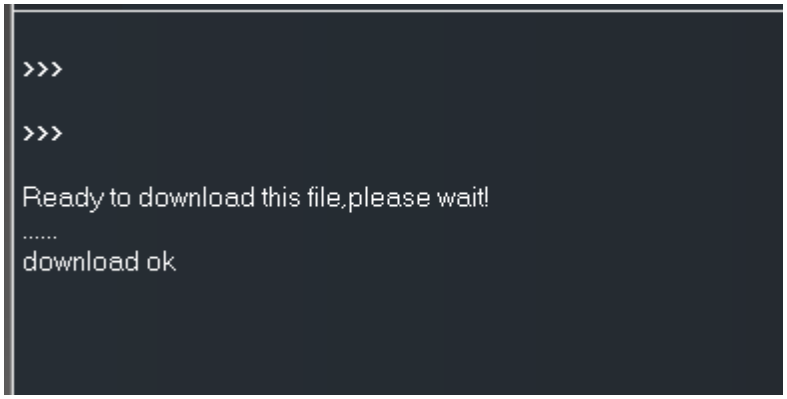
然后下载到开发板。Tools ->Download 。注意，download 是下载，只保存文件，不会运行文件。下面的 downloadAndRun 是保存后就运行文件。从源码中可以看到 fiveStar.py 文件依赖是 mp_turtle.由于现在板卡上还没有 mp_turtle.py 这个文件，所以运行肯定



会出错。

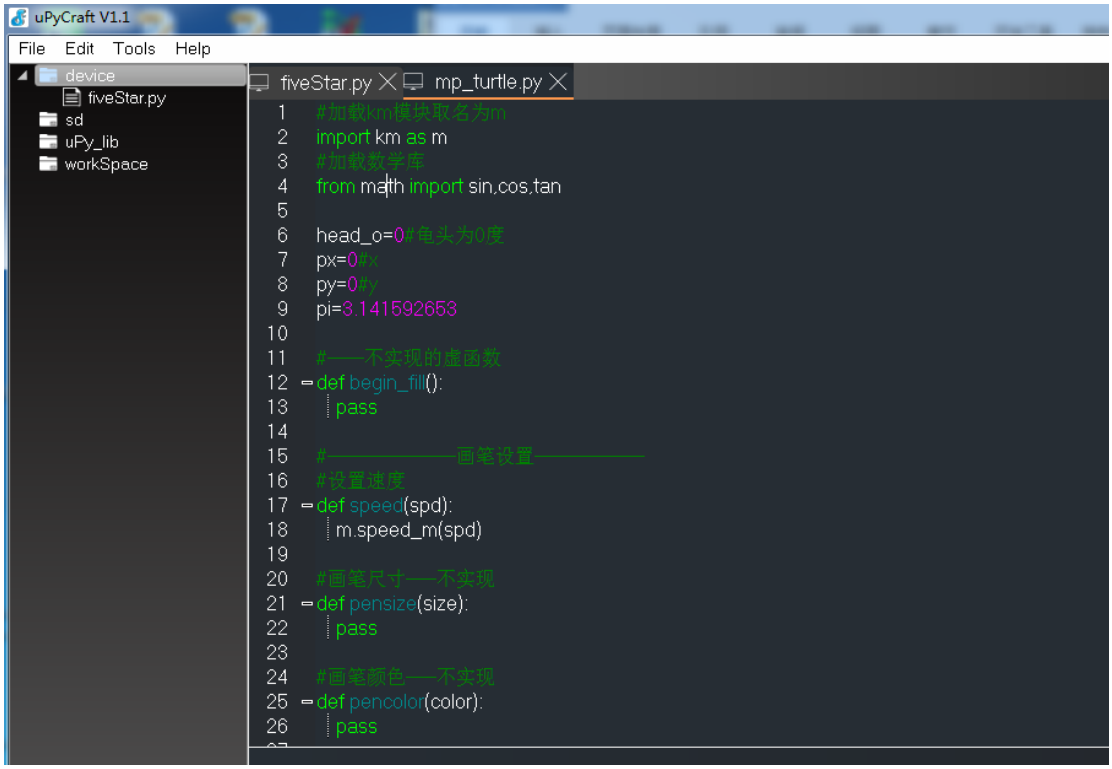


下载完后控制台会出现下面信息。

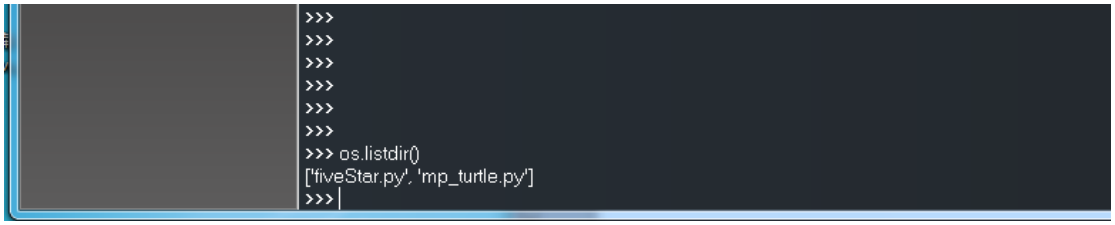


同理，打开 mp_turtle.py 文件烧写到开发板。

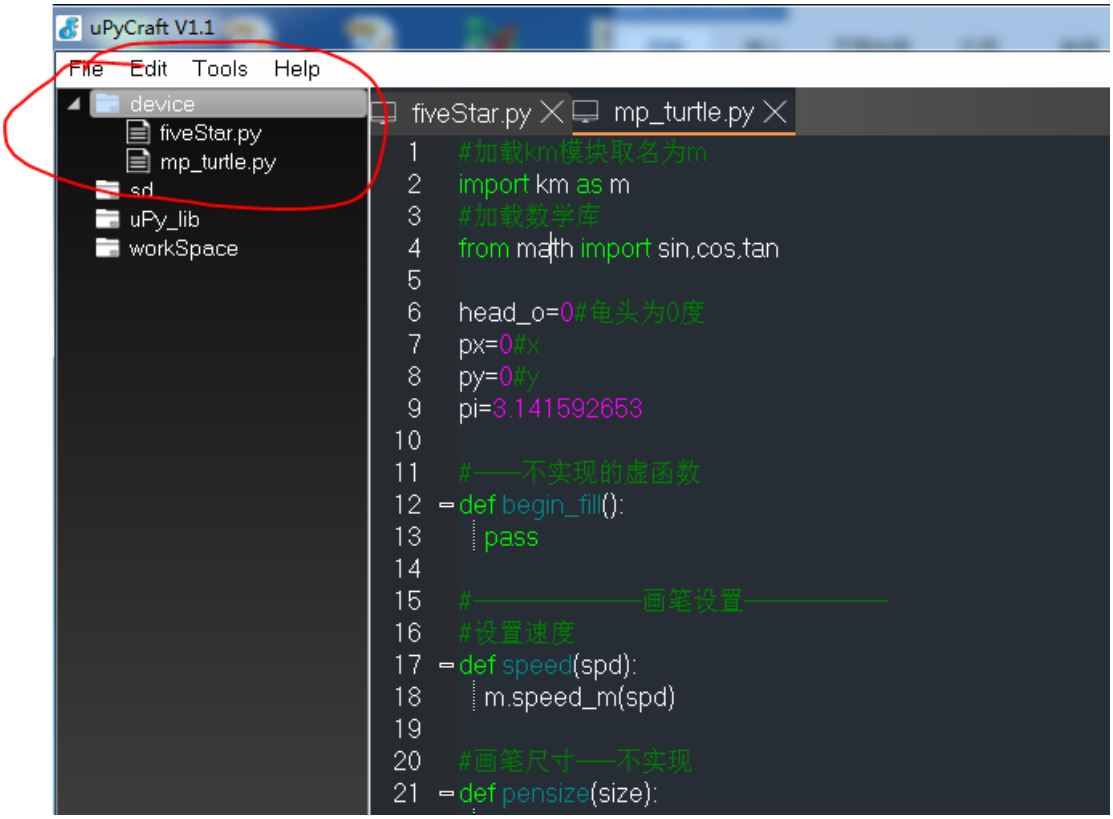
Ps:通过 mp_turtle.py 的源文件可以看到，其底层依赖是 km 模块，和 math 模块。而这两个模块是板卡支持的模块。所以这个脚本所有的依赖关系都满足。那么他肯定能运行。



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：



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。



细心的同学会注意到，如果这个时候如果调用 help（‘modules’）会不会出现刚刚传进来的两个模块呢？答案是不会。你要知道，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。就跟 windows 系统一样。出厂安装的应用和用户安装的应用一个道理。

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。如下图：

怎么死机了？？？？？Nonono!!不是死机。请看 fiveStar.py 的源码

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

他是在 while 1 里等待你的鼠标左键按下。然后画五角星。记住，你的鼠标需要插到 Kmbox 上他才知道鼠标是否按下。你如果接电脑上，电脑是不会告诉 kmbox 鼠标按下了。

如果你想停止脚本，直接在控制台按 ctr+c 即可。
中断后你可以继续使用 import fiveStar 和查看里面的成员函数。如下图

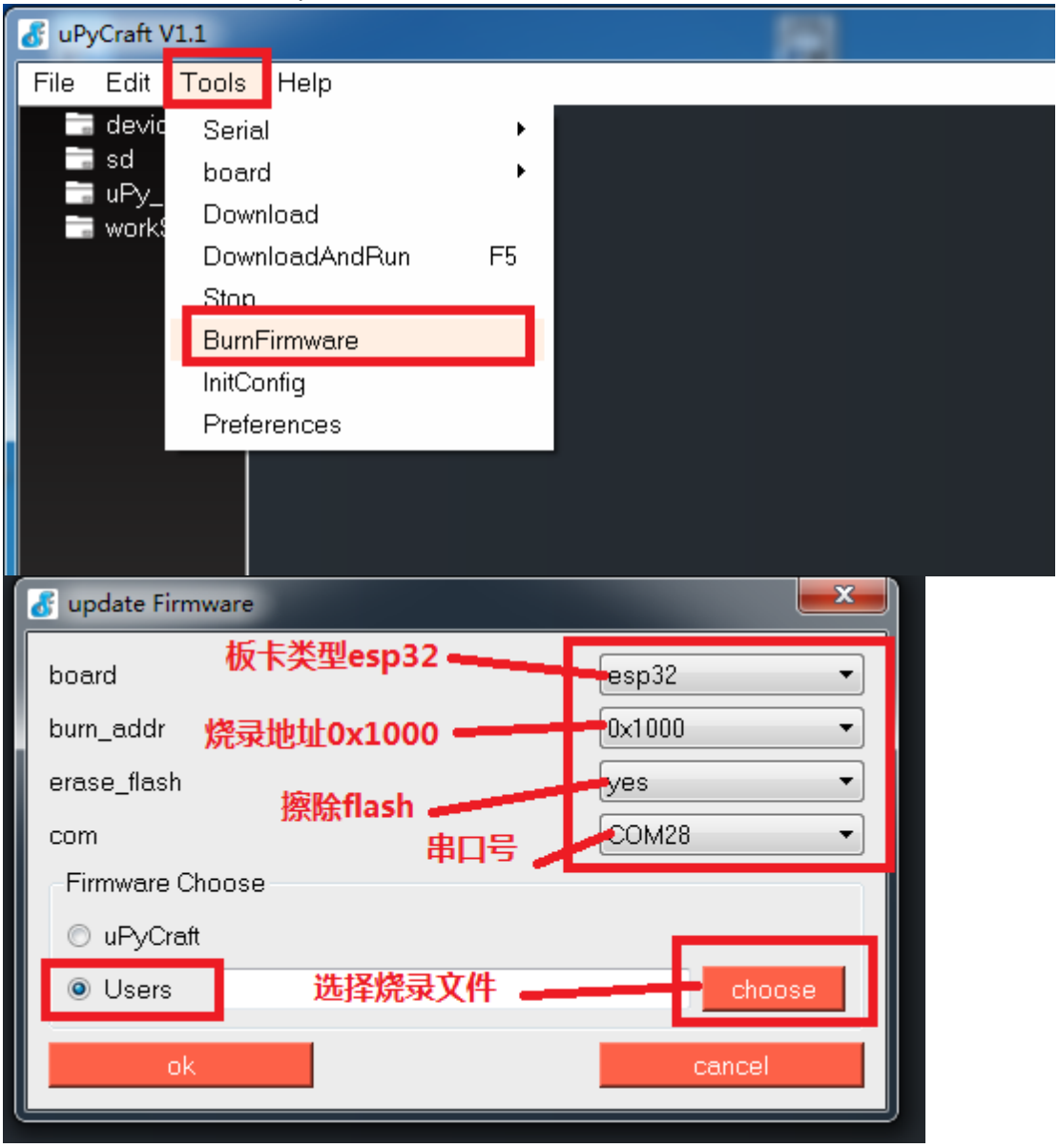
```
>>>
__class__    __name__    gc        km
os        sys        uos        bdev
myfile      fiveStar
>>> fiveStar.
__class__    __name__    __file__     cos
home       left       pi         right
sin        tan        m          forward
head_o      px         py         begin_fill
speed      pensize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

也就是说，只要你有源码，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。

固件升级（板砖修复）

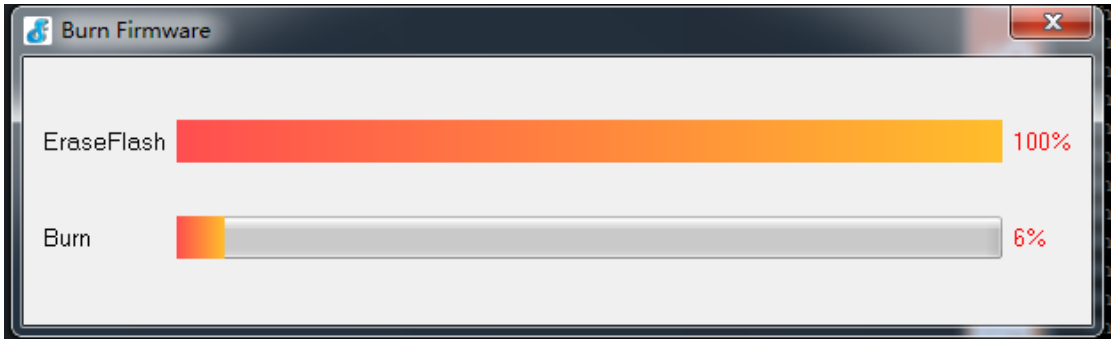
在前面教程中，你应该学会了如何使用脚本。如何自己扩展脚本。俗话说得好，常在河边走，哪有不湿鞋。那如果万一烧录的脚本有问题，运行不正常，板子又不听指令那该如何是好呢？
本章教你最暴力的一招，刷机。顾名思义，就是等同于重装操作系统。Kmbox 的刷机一共有两种方式，**刷机完后一定要断电重启开发板（插拔一下 USB1）.**

第一种方式需要工具：uPyCraft(就是前面那个小编辑器)，打开 tools->BurnFirmware



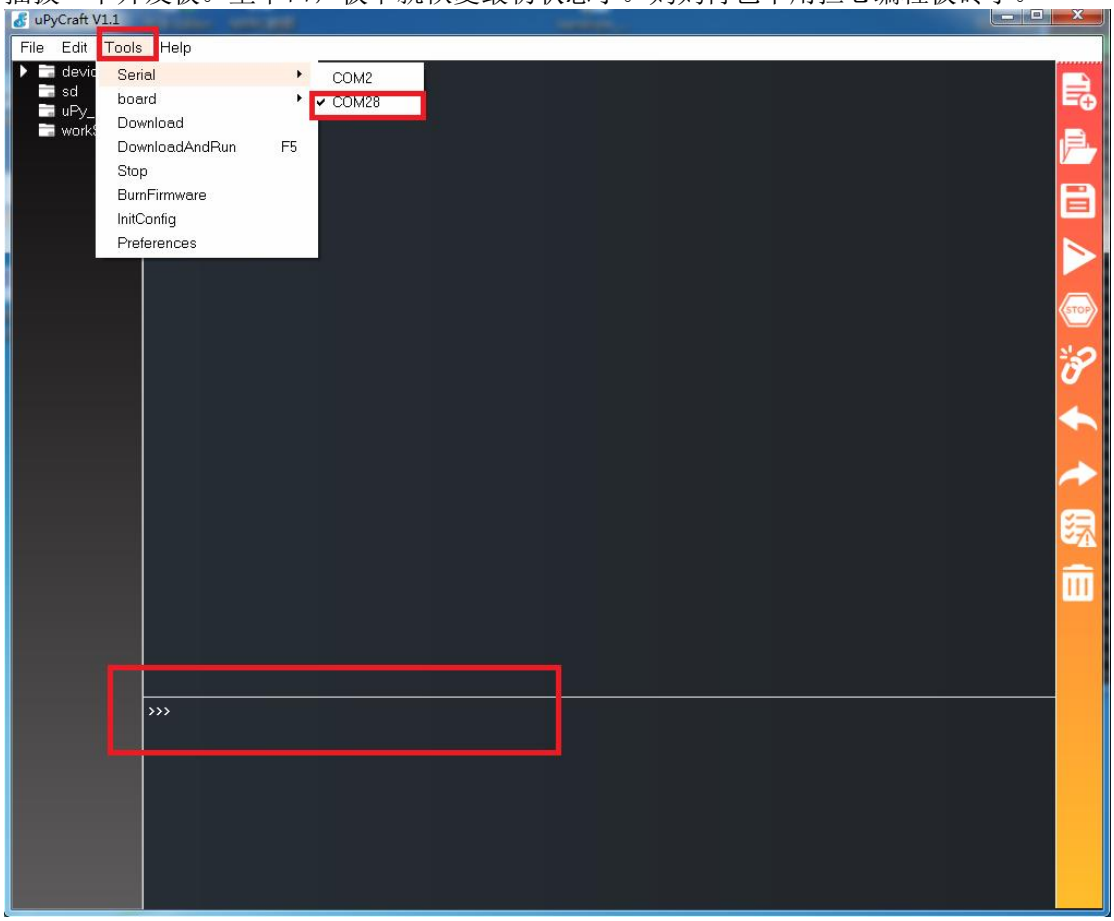
你会看到这个界面：

一看就明白了吧，在 choose 那里选择 kmbox 的固件。点击 OK 即可。



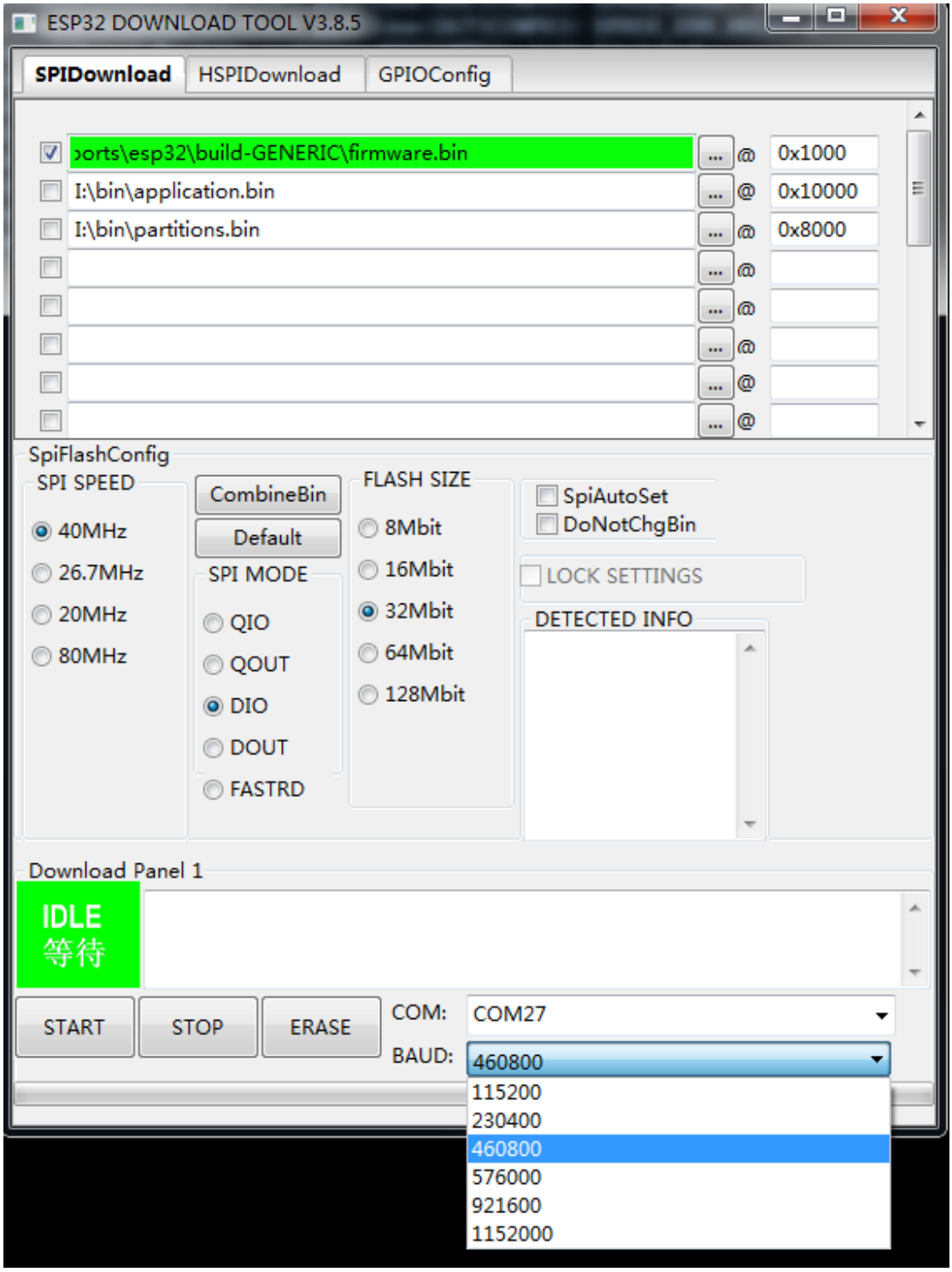
等待两个进度条走完吧。

插拔一下开发板。重串口，板卡就恢复最初状态了。妈妈再也不用担心编程板砖了。



以上操作能满足 90%的应用了。

第二种方式是 flash_download_tool_v3.8.5.exe，这是乐鑫官网的下载工具。他可用于批量生产，还能改变 WIFI，蓝牙的射频特性。如果你有几个 app，可以使用官方烧录工具。速度比上面那个快很多。具体使用就自己摸索吧。反正瞎 JB 折腾板子不会成砖头。


ESP32 DOWNLOAD TOOL V3.8.5
SPIDownload
HSPIDownload
GPIOConfig
orts\esp32\build-GENERIC\firmware.bin
0x1000
I:\bin\application.bin
0x10000
I:\bin\partitions.bin
0x8000
SpiFlashConfig
SPI SPEED
40MHz
26.7MHz
20MHz
80MHz
CombineBin
Default
SPI MODE
QIO
QOUT
DIO
DOUT
FASTRD
FLASH SIZE
8Mbit
16Mbit
32Mbit
64Mbit
128Mbit
SpiAutoSet
DoNotChgBin
LOCK SETTINGS
DETECTED INFO
Download Panel 1
IDLE
等待
START
STOP
ERASE
COM:
COM27
BAUD:
460800
115200
230400
460800
576000
921600
1152000

Kmbox 高级教程

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。那么到底如何开机启动呢？举个例子：让电脑一开机就运行 QQ 一样。首先你得在电脑里安装 QQ，然后将 QQ 设置为开机启动即可。同理，如果想要脚本一开机就运行。其实和电脑上操作一样，第一将脚本下载到 kmbox(相当于安装 QQ)，第二将脚本设置为开机启动。

在《如何使用第三方库》中，你知道了如何下载脚本。那么距离开机启动就只剩下第二步：设置脚本为开机启动。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。好比电脑按下启动后的 BIOS.主要负责初始化系统环境。BIOS 运行完后就进入系统。Kmbox 的系统就相当于 main.py.

如图所示，在 boot.py 里面啥事都没有干。你也看不到 main.py 文件（因为默认没有开机运行脚本）。所以，如果你想让你的脚本开机就运行。其实很简单，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。

单线程代码优化

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。保证执行效率。

例如：需要一个自动打怪脚本，该脚本需要有两个功能。

一、 如果按下 A 键，就自动释放 1、2、3、4、5 这 5 个技能

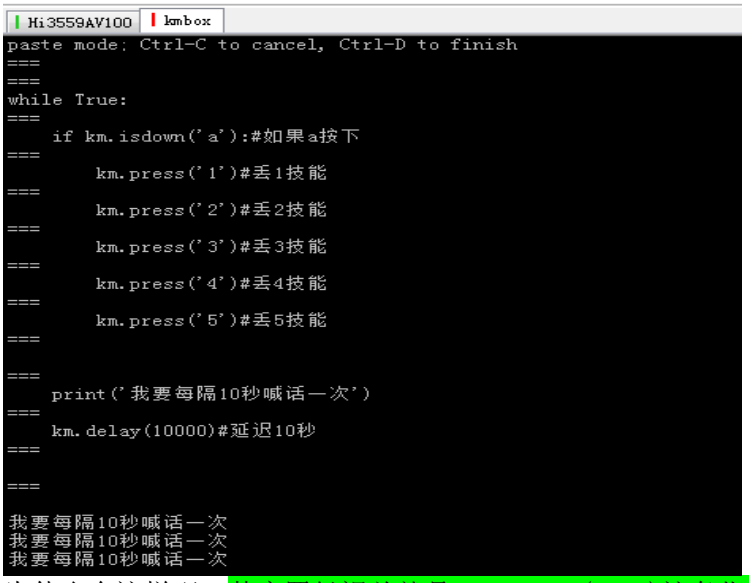
二、 每隔 10 秒自动喊话一次。

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。我们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
```

你可以试试上面的代码。不出意外，你会看见每隔 10 秒会出现一条打印。但是你按 a 键却几乎不太可能会出现 12345。如下图所示：



为什么会这样呢？其实罪魁祸首就是 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。你会发现。这样不是我需要的功能呀。我要的是按下 a 他就立马执行 12345。下面就来谈谈如何优化代码。

前面说到，罪魁祸首是 delay 函数。他白白浪费 CPU 时间。导致循环一次需要 10 秒。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。换句话说。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，理论行讲，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 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
```

```
if time.ticks_ms()-last_time>=10000:
    print('我要每隔 10 秒喊话一次')
    last_time= time.ticks_ms()#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。不错就是前面的几万倍。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。代码是人写的。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。

多线程使用

我们还是拿上一节的自动打怪脚本为例。如何用多线程的方法实现。再来仔细审题一下需求：

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 就按 12345.是没有任何关系的。他们可以看成两个独立的东西。能不能写一个函数 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”。是不是就能满足需求呢？说到这里就是咱们今天要讲的多线程了。

在说多线程时，希望你能了解，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）。你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，你感觉不出来罢了。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:                    #A 线程死循环
      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
```

你可以发现不管在 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。自己将以上代码贴入 kmbox 内部运行体验下有什么不同。这里只是抛砖引玉。多线程的用法远远不止这些。还有线程间的同步，互斥等。你们就自己去探索吧。

PS：以上代码没有写线程退出，请你们自己完善。线程的退出可以参考：http://www.clion.top/forum.php?mod=viewthread&tid=45&extra=page%3D1

脚本加密与授权（有技术卖脚本吧）

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，咱辛辛苦苦写的脚本不想就这么轻易公开给人使用。那有没有方法可以让脚本加密，把这个加密的文件给用户。这样就不用担心源码泄露了。而且你可以授权用户，按时间收费或者次数收费都行。怎么收费完全由你说了算。现在举个简单的例子：

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，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)
```

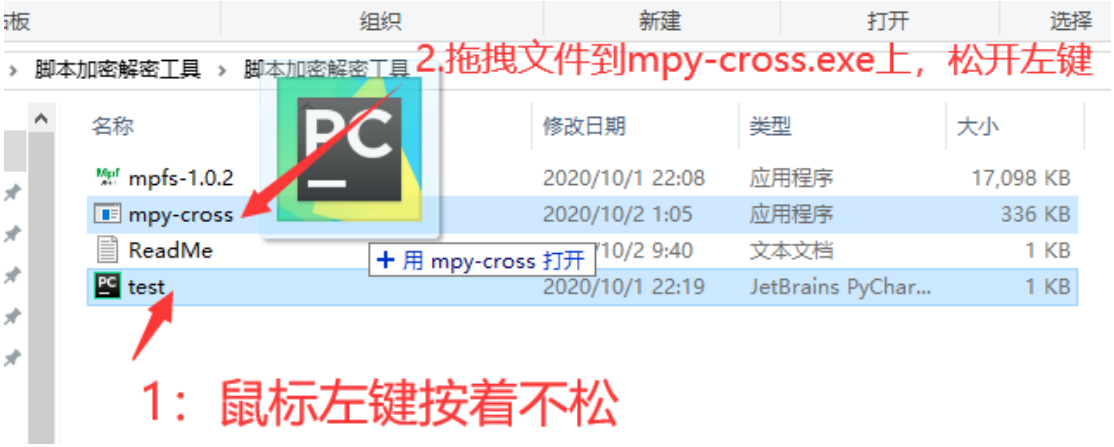
假设脚本的密钥字符串 123456。第一个函数（author）是授权使用，如果授权成功将 IsAuthor 标志置位。在运行 run 函数时会检测是否已经授权。不授权就不运行。如果咱们直接把明文 py 文件发给用户，相信大家都知道怎么破解。所以为了保护作者的劳动成果，我们需要将他加密，使用的工具是 mpy-cross。环境为 windows. 至于加密方式你们随意发挥，反正加密后除了作者本人知道外没人知道。脚本加密是作者的工作。加密脚本的使用

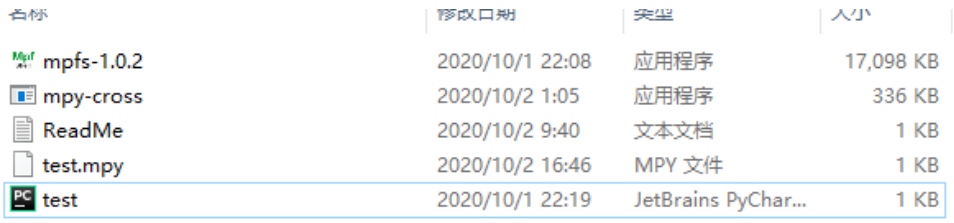
是用户的工作。现在分别从脚本作者和用户的角度讲解以下如何加密和使用。

脚本作者加密脚本

方法一：直接拖拽需要加密的文件（推荐）

我已经编译好 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。我的 python 版本是 3.8.5

```
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
```

安装完毕后咱们就可以将人见人爱花见花开的明文脚本 py 转换为密文 mpy 文件。

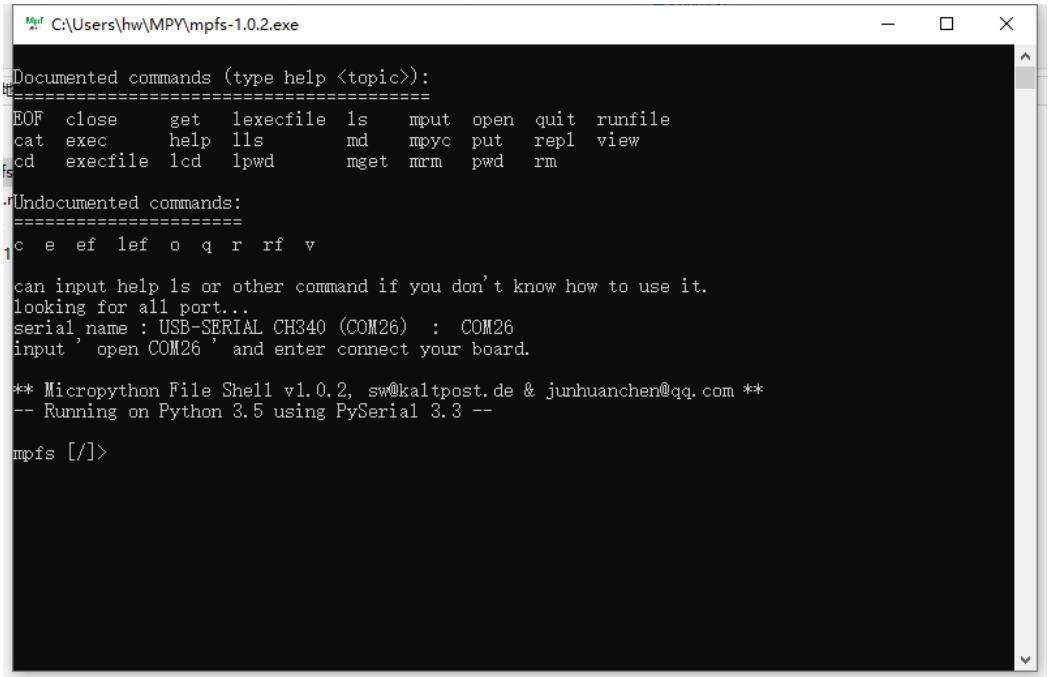
执行如下命令： python -m mpy_cross test.py -----黄色部分 test.py 是要转换的明文文件。

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。再也不用担心源码泄露问题了。到此作者加密工作结束。

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。

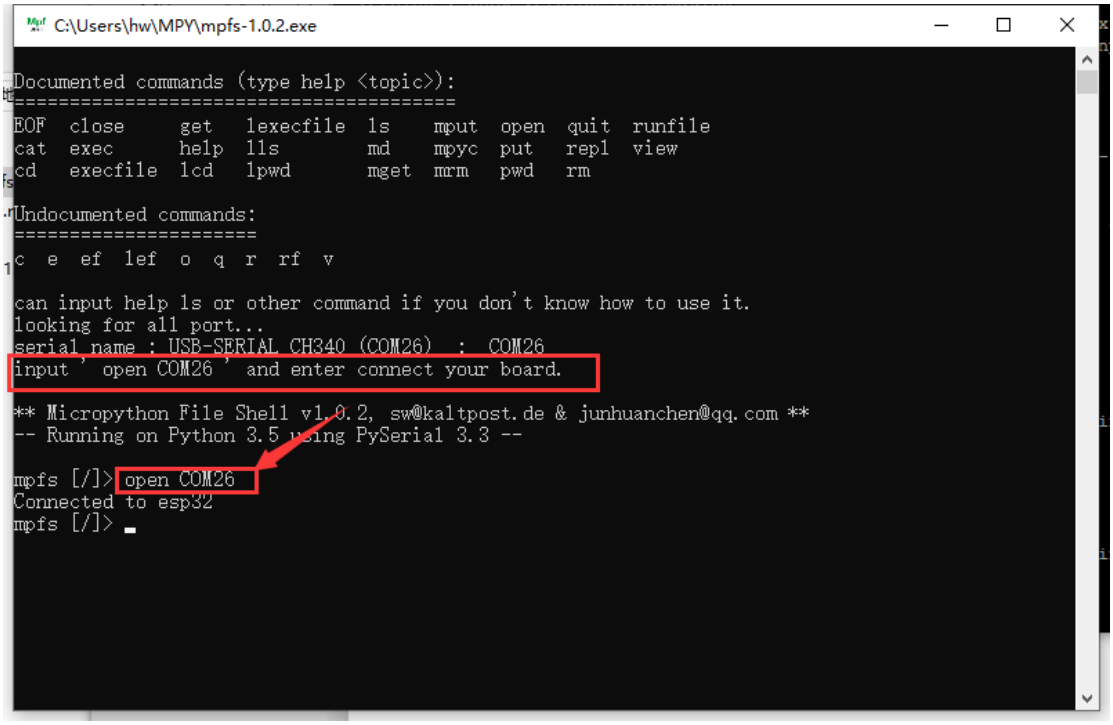
用户使用加密文件

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(去群里下载吧)，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：

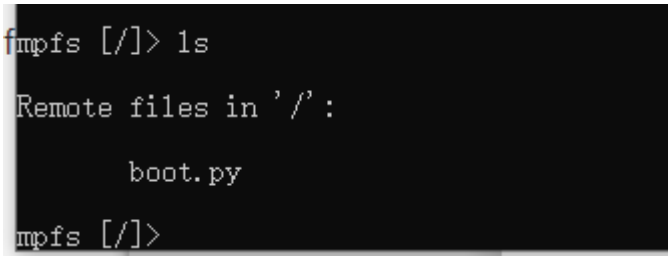


使用这个软件需要板子在 repl 模式下。如果你不知道怎样进入 repl 模式，直接刷机就行。默认就是 repl 模式。

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 COM26 ---连接开发板（COM26 是串口号）

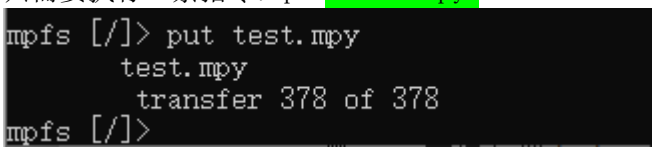


连接完成后可以看到连接成功，这里你可以像在 linux 内部操作一样输入指令 ls mv 等指令..

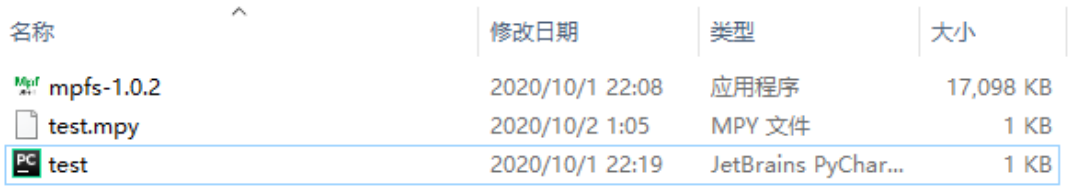


例如 ls 后可以看见当前文件系统内部有 boot.py 脚本。Kmbox 内部是有文件系统的。可以存储各种类型的文件。你可以用 os 库创建文件，修改文件名等等操作都是可以的。现在需要将 mpy 文件写到 kmbox 的文件系统。

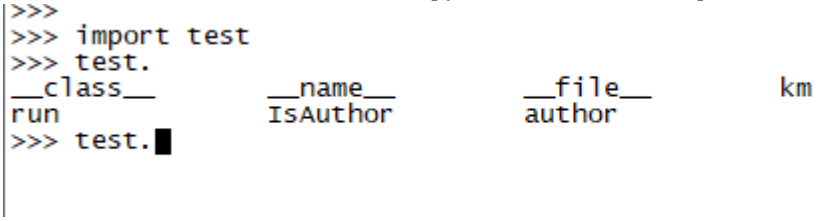
只需要执行一条指令：put test.mpy



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



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。怎样使用呢？其实和普通的 py 文件一样。直接 import test 就行



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。

防火防盗防封号

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 不就是互怼，你电脑上有我没他，有他没我！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。但是如果他真这么做了。那么该如何解决呢？

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。由此造成的后果需自行承担。

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

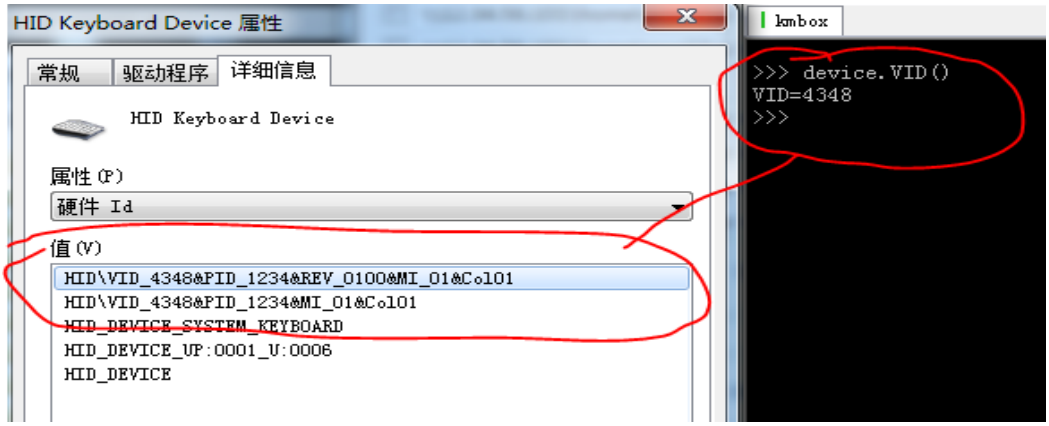
例如下面罗技公司的 VID=046d ，东芝的 VID=046C，kmbox 的 VID=4348

```
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。由此。你应该知道了 VID 和 PID 的作用。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。切记，切记。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。



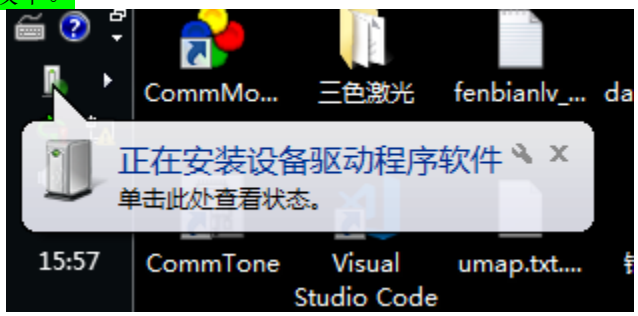
输入 device.VID(‘help’)查看如何使用 VID.

```
>>> 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。



这样你就完美的挂羊头卖狗肉，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。如果他要封 046D。那就看游戏公司有没有这个胆量了。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。谢谢！

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，如果封杀，就看反外挂公司有没有宁可错杀 1000 不可放过一个的魄力了。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。其他都是治标不治本的。因此我没有提供修改名称函数。Kmbox 要门堂堂正正的做自己，要么修改 VID 和 PID 隐姓埋名。

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。讲出来的都很空洞。如果你硬是要每秒点击 500 次，持续 1 天。被封号了。别说我硬件不行。这口大黑锅我可不背。而且你还污染了 0X4348 这个 VID。

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，一条按键指令会已最快的速度执行(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。言尽于此，以上！

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。那么如何让装得像人呢？Kmbox 提供了一些简单但是很实用的 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。一切由你决定。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。我不是数学老师。你们可以自行运行下面脚本查看每个参数的作用。讲得天花乱坠不如实际看看效果。演示脚本和视频效果请去群共享下载。

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吧。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、 不要草木皆兵。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。淘宝上那些软件模拟挂都说自己防封，难道咱这纯硬件的干不过他们？

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。

kmbox 键鼠录制与播放

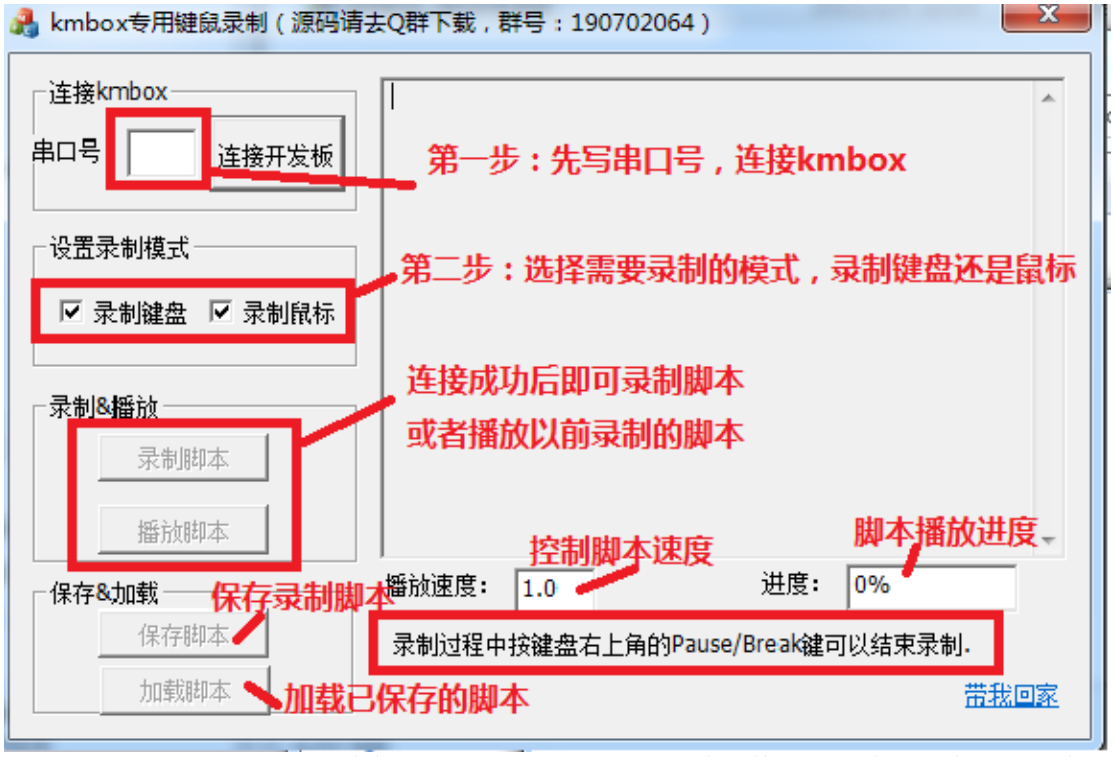
键鼠录制和播放原理上其实很简单，由于 kmbox 是跨接到电脑和键鼠之间的。原则上电脑所有键鼠数据都是 kmbox 发送给 PC 的。所以，录制相当于 kmbox 主动上报所有

键鼠数据，电脑记录下这些数据即可完成录制工作。同理，播放就是将原来录制的数据发送给 kmbox，让他一条一条的执行，这就是播放操作。前面的硬件知识你已经知道，kmbox 内部只有 4MB 的存储空间，且内存有限。不可能存放长时间的录制数据。所以 kmbox 借用 PC 的容量空间，可实现海量脚本的录制和存储。这样就不用受到自身的 RAM 和 ROM 的限制。而且 PC 与 kmbox 的通信是高阶应用的必经之路。有了 PC 当扩展，kmbox 可以实现更多高级功能。例如找图功能，找色功能，AI 识图等功能。这些 kmbox 都不可能直接处理，但是可以用 PC 来处理，让后把需要控制的结果让 kmbox 执行即可。

本小节主要讲 kmbox 的串口通信，kmbox 还支持网络通信。本节的上位机软件源码请去群共享下载。如果需要其他的功能，可以参考此程序增加或者修改。编译环境为 Microsoft Visual Studio 2010。

键鼠录制

编译生成文件为：kmbox_record.exe 双击后即可运行：



如上图所示，kmbox 通过串口和 PC 进行通信，所以此软件需要正确填写串口号。查找串口号请参考前面的章节 “登录 kmbox”。

3

正确连接 kmbox 后录制脚本和加载脚本按钮解锁。右边会出现 kmbox 的版本号。你可以执行录制脚本操作或者加载以前录制好的脚本。

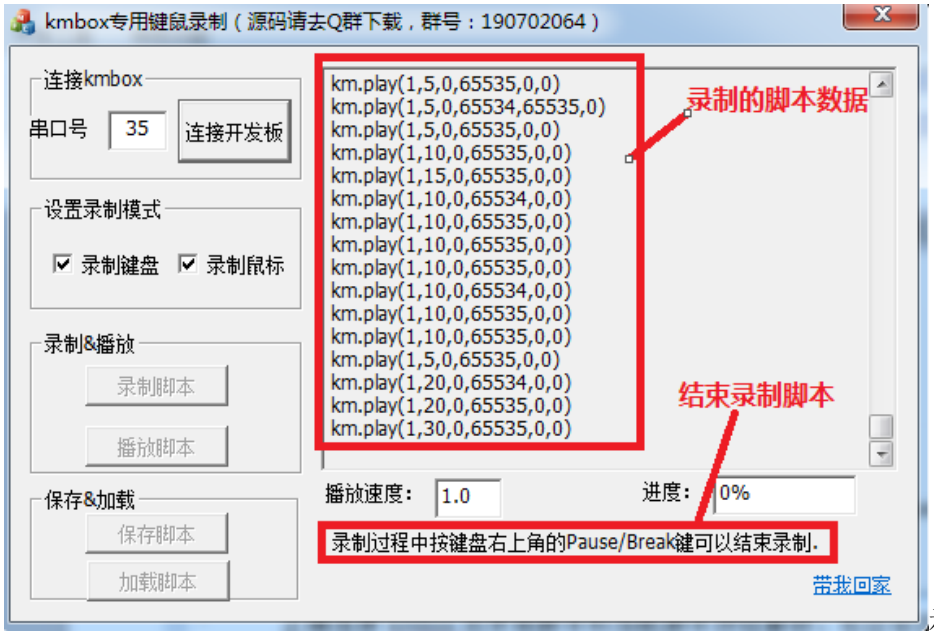
注意：如果软件连不上 kmbox 请检查以下几点：

一、 确保串口没有被其他程序占用。如果有请关闭其他无关程序。

二、 请确保 kmbox 内部没有跑任何脚本。

三、 请确保 kmbox 处于 REPL 状态。

点击录制脚本后右边编辑框中会记录下 kmbox 所有指定数据。如下图所



示：

结束录制请按键盘上的 Pause/Break 按键。

播放录制

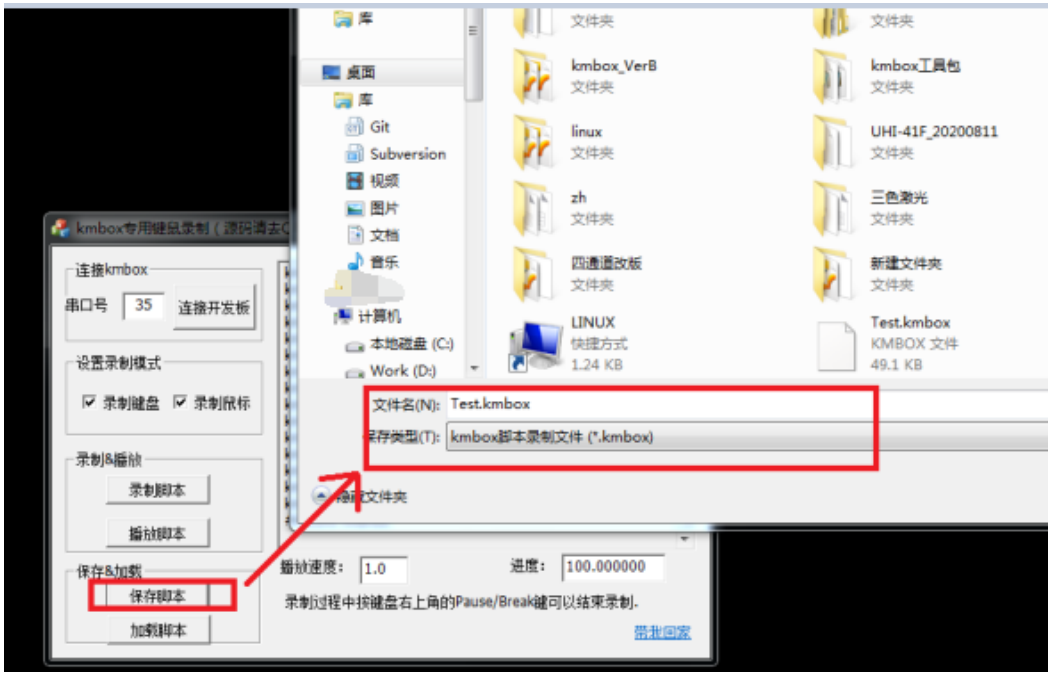
当结束录制后，播放脚本按钮解锁，你可以点击该按钮播放刚刚的录制内容。

右边的播放速度可以试制加快或者减慢脚本的播放速度。进度用于显示当前脚本播放了百分之多少。播放完成后会显示 100%。

PS:别问我能录制多久时间,这完全取决于你的电脑配置。只要你有足够的空间存脚本就行。

保存录制

保存录制用于将录制好的脚本文件保存成文件，方便下次加载调用。



如图所示，点击保存脚本后，你可以给你的脚本取个名字，保存到电脑上。

加载录 1

加载录制是指将前面保存录制的内容重新读回到软件，然后可以点击播放脚本执行。请自行尝试。

ps：此上位机和 kmbox 是配套使用的。不可单独使用。也请不要随意修改录制的脚本。此录制脚本也可以烧写到 kmbox 内部运行，但是注意容量和内存限制。

Kmbox 其他模块

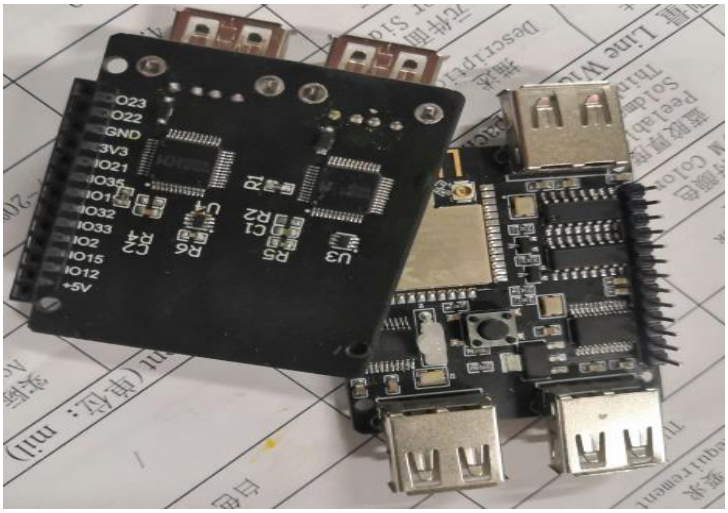
Kvm 同步器、异步器

Kmbox-kvm 是一款既可以实现键鼠同步又能实现键鼠异步的扩展板。用它可以轻松实现一套键鼠同步、异步控制四台 PC(需要搭配 kmbox 主控板)。所有端口 USB 的键鼠数据均可编程。可以随心所欲的掌控每台电脑的键鼠数据。拥有 kmbox-kvm 后可以轻松的一套键鼠游走于四台电脑之间。一个人操作一个小队不再是梦想。绝对是搬砖，单刷，团队下副本的神器。

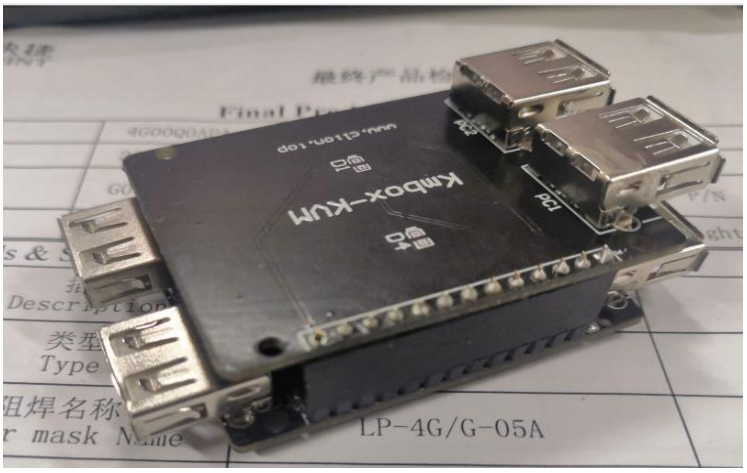
注意：kvm 扩展板必须搭配 kmbox 主板使用。单独购买无法使用！

Kmbox-kvm 扩展板实物图

正面



背面



搭配 kmbox

Kmbox-kvm 扩展板功能框图

Kvm 扩展板包含两个 USB 口，分别对应 USB1 和 USB2。这两个 USB 口连接到电脑后会自动模拟一套键盘和鼠标（纯硬件键盘和鼠标，跟外接键盘鼠标一样）。这套键盘鼠标是标准的 HID 键鼠，电脑端不需要安装任何驱动。USB1 和 USB2 端口可以连接两台独立电脑。通过 KVM 扩展板同步或者异步控制这两台 PC。且 USB 口数据均可编程。编程在 kmbox 主板上实现。Kvm 扩展板负责将键鼠数据发送到不同的 PC 上。Kmbox-kvm 一共可以控制四台电脑。虚线表示蓝牙控制。

kvm 扩展板使用

Kmbox 主板内部已经包含 kvm 模块，使用前只需要 import kvm 即可，如下图所示：

```
>>> import kvm
>>> kvm.
__class__        enable          help            port1
port2            sync
```

enable 函数（使能 kvm 模块）

Kvm 模块使用前必须首先调用 enable 函数，完成 kvm 模块的初始化。

```
>>> kvm.enable('help')
kvm.enable():用于使能kvm功能。使用kvm模块里的函数前必须调用
    调用方法：kvm.enable(1)
    调用后kmbox的键鼠消息数据将同步发送到kvm板。默认PC1和PC2是同步模式
    如需单独控制pc1和pc2的导通模式，请参考kvm.port函数
关闭kvm扩展板可调用kvm.enable(0).
```

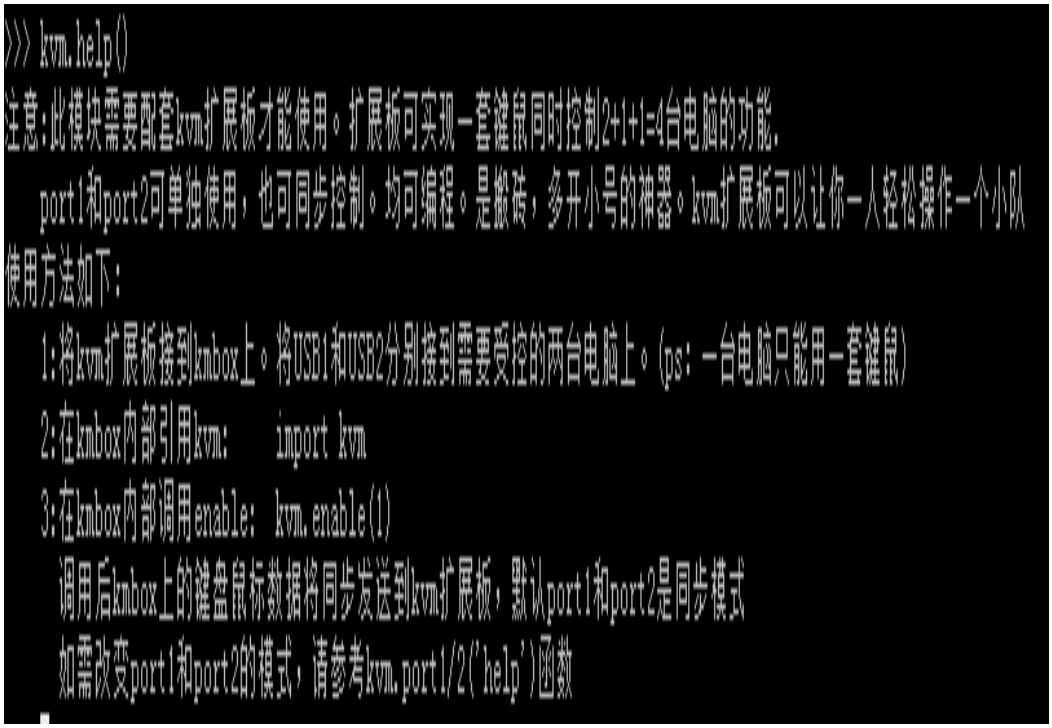
port1/2 函数(键鼠数据走向控制)

port1,port2 函数用于控制键鼠数据走向哪个端口。有时候键鼠操作只需要发给 port1 的电脑而不是 port2。可以用此函数使能 port1，关闭 port2.这样就不会影响到 port2 端口。其用法如下。

```
>>> kvm.port1('help')
USB port1同步使能：
 该函数用于设定port1端口是否接收kmbox的物理键鼠消息，可以用此函数控制port1是否需要跟kmbox的键盘鼠标同步
 kvm.port1(0):表示关闭port1端口，即kmbox的键鼠数据不会主动发送到port1
 kvm.port1(1):表示打开port1端口，即kmbox的键鼠数据会主动发送到port1
 kvm.port1() :不带参数是查询port1端口是否启用，返回值0：未启用 1：启用
```

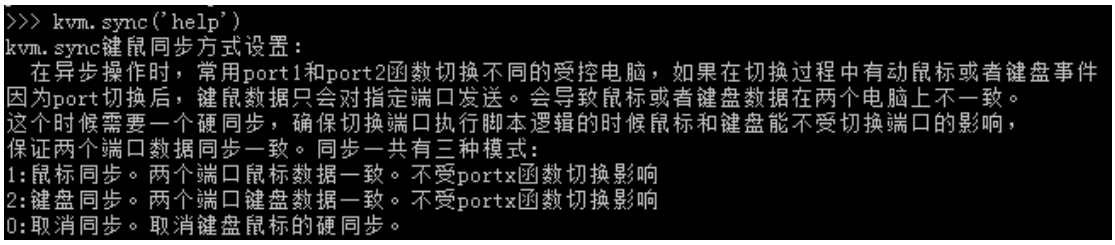
help 函数（帮助文档）

输入 kvm.help()后会显示一般的帮助信息。同 km 模块一样，那个函数不懂就 kvm.xxx(‘help’)吧。



sync 键鼠同步函数

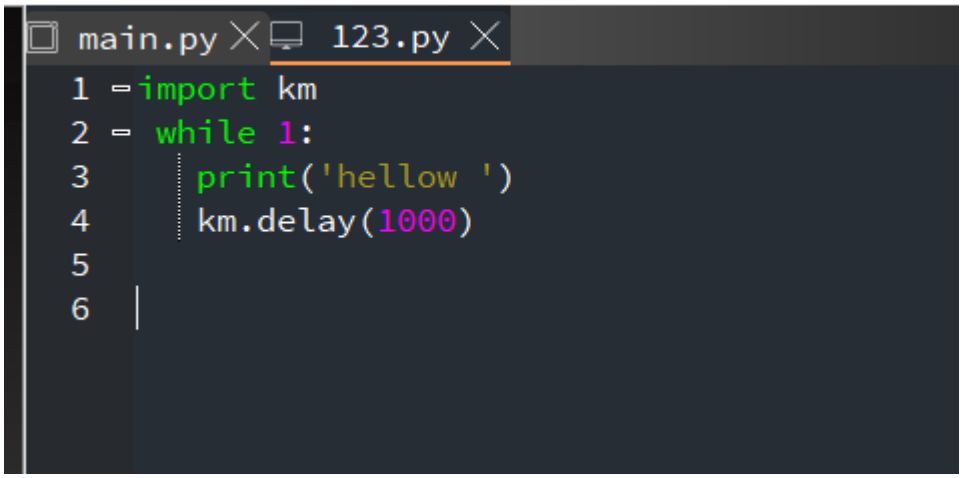
同步函数可以无视 port1/2 的切换开关，例如你期望 USB1 和 USB2 的鼠标数据一样，只是键盘数据不同。你可以使用 sync(1)来实现鼠标硬同步。键盘数据可以做到 port1 和 port2 独立。



Kvm 扩展板典型应用

一、可编程同步器

Kvm 扩展板可以任意切换控制模式，快捷键随你改。再也不用忍受别扭的切屏按键，一切随你定义。单屏，多屏同步随你切换。一切由你的脚本说了算。（脚本请去群共享下载）


main.py
123.py
1 import km
2 while 1:
3 print('hellow ')
4 km.delay(1000)
5
6

```python
'''
kvm模块使用demo
将此脚本命名为main.py保存到kmbox，上电后即可自动运行。本脚本可实现将键盘鼠标
数据任意切换到不同的端口，即可同步切换也可异步切换。
默认kvm模块为键鼠同步模式。切换说明.
win(左)+F1  :键盘鼠标切换到port1
win(左)+F2  :键盘鼠标切换到port2
win(左)+F3  :键盘鼠标切换到port1和port2
win(左)+1  :port1和port2键盘同步，鼠标异步(异步操作受win+f1,f2影响)
win(左)+2  :port1和port2鼠标同步，键盘异步(异步操作受win+f1,f2影响)
win(左)+3  :port1和port2键盘鼠标同步关闭
'''
import kvm
import km
#快捷键切换定义
key_alt=km.getint('gui-l') #左WIN键
key_f1=km.getint('f1')
key_f2=km.getint('f2')
key_f3=km.getint('f3')
key_1=km.getint('1')
key_2=km.getint('2')
key_3=km.getint('3')
km.mask(key_alt,1)#屏蔽左windows键
kvm.enable(1)      #使能kvm模块
while 1:
    if km.isdown(key_alt) and km.isdown(key_f1): #win+f1按下
        kvm.port1(1)#打开port1
        kvm.port2(0)#关闭port2
        print('win(左)+F1  :键盘鼠标切换到port1')
    elif km.isdown(key_alt) and km.isdown(key_f2): #win+f2按下
        kvm.port1(0)#关闭port1
        kvm.port2(1)#打开port2
        print('win(左)+F2  :键盘鼠标切换到port2')
    elif km.isdown(key_alt) and km.isdown(key_f3): #win+f3按下
        kvm.port1(1)#打开port1
        kvm.port2(1)#打开port2
        print('win(左)+F3  :键盘鼠标切换到port1和port2')
    elif km.isdown(key_alt) and km.isdown(key_1): #win+1
        kvm.sync(2)#键盘同步，鼠标异步
        print('win(左)+1   :键盘同步，鼠标异步')
    elif km.isdown(key_alt) and km.isdown(key_2): #win+2
        kvm.sync(1)#键盘异步，鼠标同步
        print('win(左)+2   :键盘异步，鼠标同步')
    elif km.isdown(key_alt) and km.isdown(key_3): #win+3
        kvm.sync(0)#键盘鼠标硬件同步关闭
        print('win(左)+3   :键盘，鼠标同步关闭')
    else:
        km.delay(5)#空闲
```

二、可编程异步器

Kvm 除了同步功能外，还可以异步操作。你可能用过同步器，但是你可能没有用过异步器。何为异步器。举个简单的例子。两台 PC，一套键鼠。在键盘上按下 a 键，PC#1 收到 a 键，PC#2 收到 b 键。对于同样的键盘 a 按下，两台 PC 却收到不一样的按键，这就叫异步。同理，鼠标也可以异步。你可能会问，异步有什么用？全都乱套了嘛。不不不。异步器不像同步器那样死板的复制键鼠消息。他可以实现每个电脑的不同操作。因此更加灵活。

举个例子：某游戏有 3 个职业，战士，法师，道士。战士是近战，法师远程，道士是奶妈。

如果有三台电脑，每个电脑玩其中一个职业。假设走路用鼠标，那么让三台电脑的鼠标同步。也就是走路三个角色一模一样。如果你想刷怪升级。法师远程，适合快速拉怪。战士近战，有近距离 AOE，道士可以加血。那么我们写一个这样的脚本。

1、按下技能键 1，自动切换到法师电脑，丢远程拉怪技能。

2、此时战士和道士不做任何处理，怪物被攻击后自动靠近法师。

3、延迟一会战士自动丢 AOE 群技能。解决蜂拥而至的怪。同时道士奶战士和法师。

这样就能完美的发挥三个职业的优势，从整体来看，你只按了一个 1 键，就同时控制了三台电脑做不同的事。法师拉怪，战士打伤害，道士负责后勤补充血量。一个人轻松控制了三个角色。异步器的功能是同步器实现不了。同步器只可能一个一个的操作，但是异步器可以三个角色同时一起操作。

下面是一个简单的键盘异步鼠标同步 demo（脚本请去群共享下载）。

```
'''
鼠标同步键盘异步demo：
    鼠标同步键盘异步常用在PC1和PC2的鼠标移动必须一致，但键盘按键不一致的情况。
此模式下，鼠标数据不受kvm.portx()函数切换影响，这个切换函数只会影响到键盘数据通向。
利用此模式可以实现同步走位，异步丢技能。

下面这段代码运行后PC1和PC2鼠标运行同步，区别在PC1和PC2上按键反应不一样。
可自行与01-键盘异步.py脚本对比查看运行效果
'''
import kvm
import km
kvm.enable(1)      #使能kvm模块
km.mask('a',1)     #屏蔽键盘a键
kvm.sync(1)        #鼠标同步键盘异步模式
while 1:
    if km.isdown('a'):#如果键盘a键按下
        kvm.port2(0)  #PC2去使能
        kvm.port1(1)  #PC1使能    --准备给PC1发送数据。
        km.press('b') #给PC1发送b,此时km.press('b')只会发送给PC1，PC2不会收到b.因为前面两条指令关闭了PC2的通路
        kvm.port2(1)  #PC2使能
        kvm.port1(0)  #PC1去使能    --准备给PC2发送数据。
        km.press('c') #给PC2发送c,此时km.press('c')只会发送给PC2，PC1不会收到c.因为前面两条指令关闭了PC1的通路
        while km.isdown('a'): #等待键盘a键弹起。避免发送速度过快。
            km.delay(1)#空等待 直到键盘a键弹起
```

如果还需要其他的同步异步骚操作，自己写代码吧。

BT 蓝牙模式

Kmbox 默认支持蓝牙模式，但是蓝牙没有直接开启，如需开启蓝牙模式需要脚本软件打开。在蓝牙模式下，kmbox 连接的键盘鼠标可以直接通过蓝牙的方式传输给上位机 PC 或者手机。可实现有线键鼠的蓝牙化。

如何启动蓝牙模式

启动蓝牙模式需要 import bt 模块。如下图所示。

```
>>> import bt
>>> bt.
__class__        enable          help
```

蓝牙模块比较简单。只有一个帮助（help）函数和一个使能(enable)函数.

help 蓝牙帮助函数

```
>>> bt.help()
bluetooth模块帮助:
设置查询蓝牙模块工作状态
   bt.enable(1):使能蓝牙模式
   bt.enable(0):关闭蓝牙模式
```

enable 蓝牙使能函数

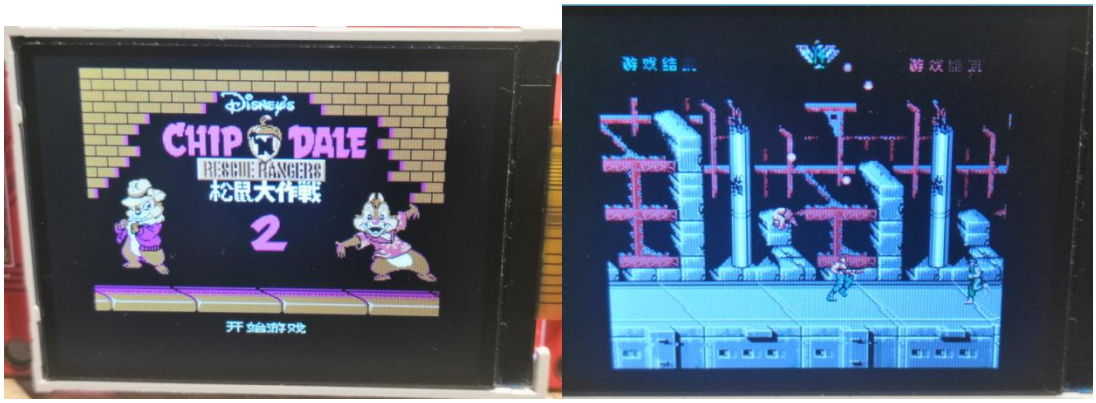
```
>>> bt.enable('help')
设置查询蓝牙模块工作状态
   bt.enable(1):使能蓝牙模式
   bt.enable(0):关闭蓝牙模式
```

也就是说启用蓝牙只需要调用 bt.enable(1)即可。然后是主机与 kmbox 的蓝牙配对。配对完成后就能正常使用。

如果你想开机默认就启用蓝牙，通过前面的学习你应该知道了吧。只需要将 import bt 和 bt.enable(1)这两行代码写到 boot.py 里面即可。

Kmbox-nes(FC 模拟器)

kmbox 除了能做外挂，还能玩 NES 游戏啦。如下图所示



所需材料如下：

一、 USB 键盘（当手柄使用）

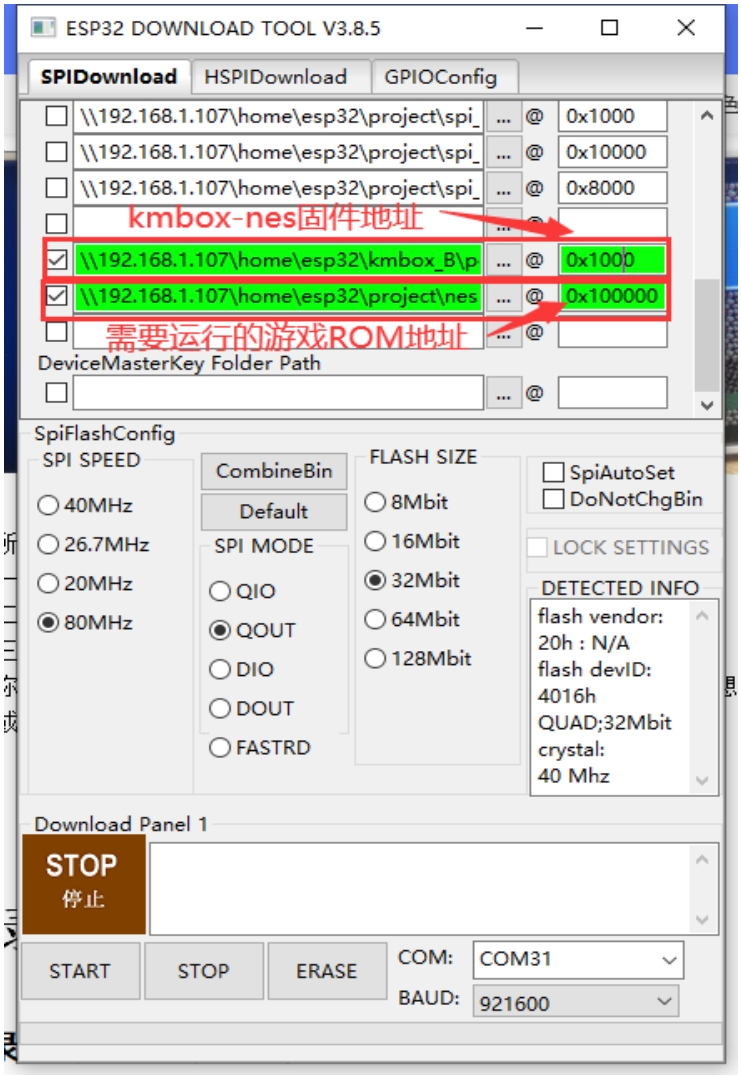
二、 一块 SPI 串口屏（驱动芯片 ILI9341 分辨率 320X240）

三、 游戏 ROM(群共享下载---约有 200+个游戏)

首先你需要刷 kmbox-nes 固件，请直接群共享中下载。然后找一个你想玩的游戏 ROM(群共享有或者你自己去网上下载)。准备就绪后刷机：（注意地址）

固件地址为： 0x1000

ROM 地址为：0x100000



烧录完毕后就可以接上键盘操作 NES 游戏啦。

键盘操作方式如下：

手柄	键盘	手柄	键盘	手柄	键盘
方向上	W 或↑	按钮 A	K 或空格	软 RESET	PgUp
方向下	S 或↓	按钮 B	J	硬 RESET	Esc
方向左	A 或←	选择键 Select	Home		
方向右	D 或→	开始键 Start	Insert 或回车键		

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
```

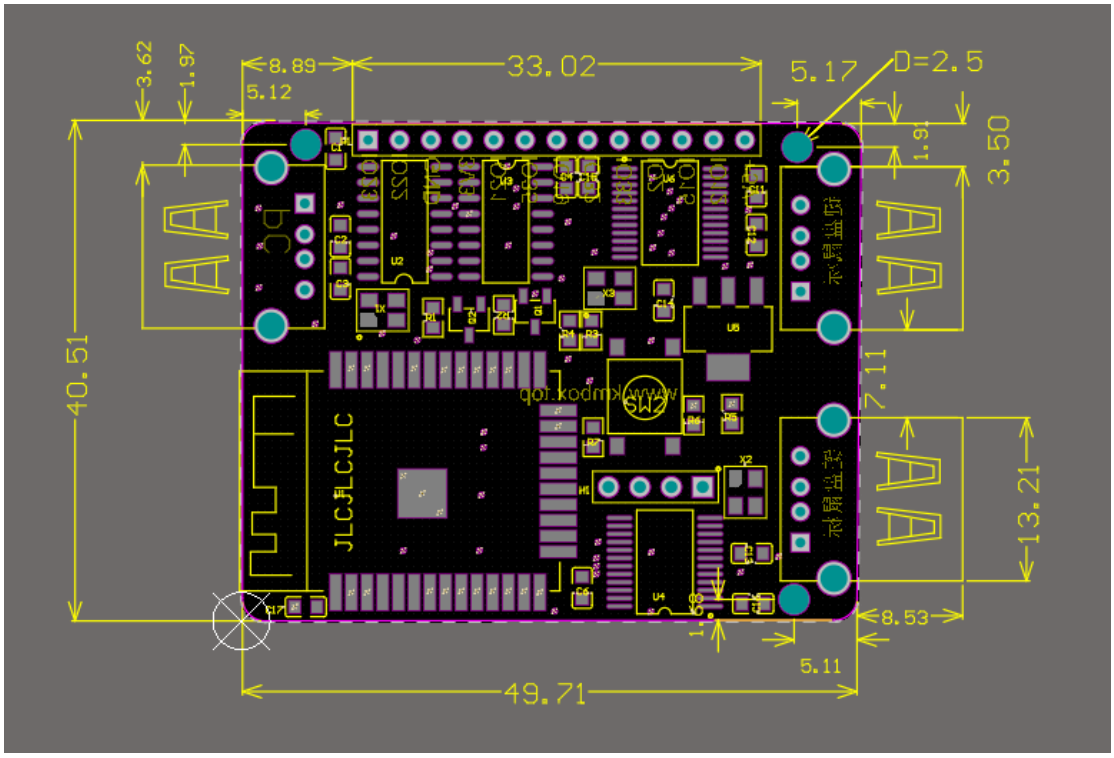
```
#define KEY_F11                                 0x44
#define KEY_F12                                 0x45
#define KEY_PRINTSCREEN                         0x46
#define KEY_SCROLL_LOCK                         0x47
#define KEY_PAUSE                               0x48
#define KEY_INSERT                              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_CBRACE                        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
#define KEY_RIGHTALT                     0xE6
#define KEY_RIGHT_GUI                    0xE7
```

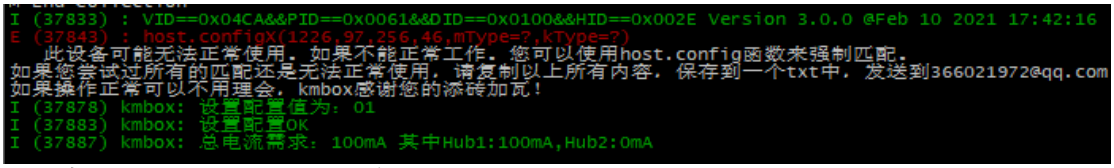
附录 2. kmbox 结构尺寸图



常见问题解决办法

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，请不要慌，问题不大，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。不出意外你肯定会看到下面这行打印。



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：

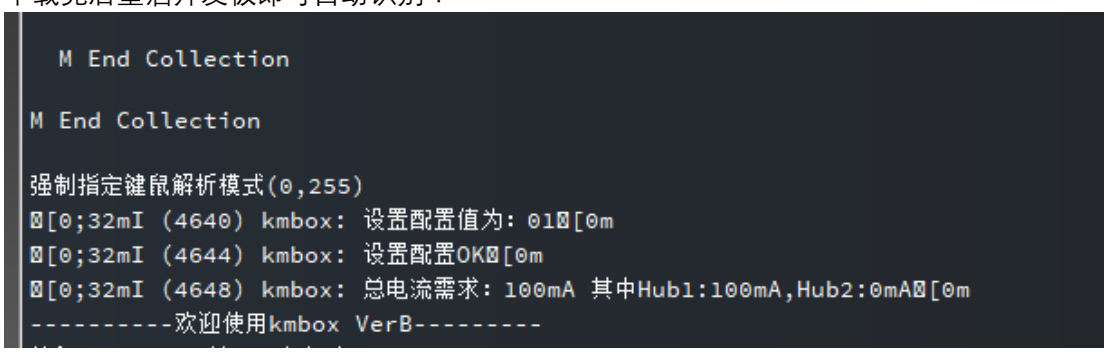
一、 无线键鼠接收器接到 kmbox 上，接收器里会接收键盘和鼠标数据。你需要同时设置 mType 和 kType 的值。如果你的设备是纯键盘，那么请将 mType 的值设置为 255.如果你的设备是纯鼠标，那么将 kType 的值设置为 255。

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：

1

下载完后重启开发板即可自动识别：



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。

一插拔键鼠串口就断开

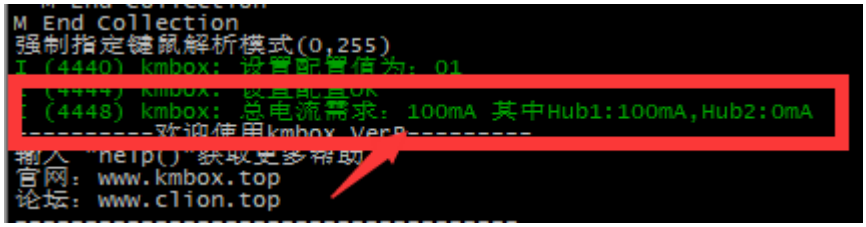
这是典型的供电不足问题。不要将 kmbox 的 USB1 口接任何 HUB，扩展器。请确保直连到电脑的 USB 口。并且保证 USB 口供电充足。

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 (161270) 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：

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，请下载群里的驱动。