

MapleJS
V1.0

开发指南

文档版本 01
发布日期 2019-04-29



华为技术有限公司



版权所有 © 华为技术有限公司 2019。保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明



HUAWEI和其他华为商标均为华为技术有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为技术有限公司

地址：深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址：<http://www.huawei.com>

客户服务邮箱：support@huawei.com

客户服务电话：4008302118

目 录

1 简介.....	1
2 环境准备及工具使用.....	2
2.1 环境准备及使用.....	2
2.2 初始 JS 文件加载工具.....	4
2.3 JavaScript SDK 使用.....	6
3 语法规格.....	11
3.1 语法标准.....	11
3.2 裁剪规格.....	11
4 模块接口.....	19
4.1 基础模块接口.....	19
4.1.1 buffer.....	19
4.1.1.1 介绍.....	19
4.1.1.2 模块接口.....	19
4.1.1.3 约束.....	23
4.1.1.4 样例.....	23
4.1.2 crypto.....	23
4.1.2.1 介绍.....	24
4.1.2.2 模块接口.....	24
4.1.2.3 约束.....	26
4.1.2.4 样例.....	26
4.1.3 fs.....	26
4.1.3.1 介绍.....	26
4.1.3.2 模块接口.....	26
4.1.3.3 约束.....	29
4.1.3.4 样例.....	29
4.1.4 gpio.....	29
4.1.4.1 介绍.....	30
4.1.4.2 模块接口.....	30
4.1.4.3 约束.....	32
4.1.4.4 样例.....	32
4.1.5 i2c.....	33
4.1.5.1 介绍.....	33

4.1.5.2 模块接口.....	33
4.1.5.3 约束.....	37
4.1.5.4 样例.....	37
4.1.6 objectPersistence.....	38
4.1.6.1 介绍.....	38
4.1.6.2 模块接口.....	38
4.1.6.3 约束.....	39
4.1.6.4 样例.....	39
4.1.7 pwm.....	40
4.1.7.1 介绍.....	40
4.1.7.2 模块接口.....	40
4.1.7.3 约束.....	42
4.1.7.4 样例.....	42
4.1.8 rtc.....	43
4.1.8.1 介绍.....	43
4.1.8.2 模块接口.....	43
4.1.8.3 约束.....	47
4.1.8.4 样例.....	47
4.1.9 spi.....	47
4.1.9.1 介绍.....	47
4.1.9.2 模块接口.....	48
4.1.9.3 约束.....	49
4.1.9.4 样例.....	49
4.1.10 timer.....	50
4.1.10.1 介绍.....	50
4.1.10.2 模块接口.....	50
4.1.10.3 约束.....	51
4.1.10.4 样例.....	52
4.1.11 uart.....	52
4.1.11.1 介绍.....	52
4.1.11.2 模块接口.....	52
4.1.11.3 约束.....	55
4.1.11.4 样例.....	55
4.2 行业使能库.....	56
4.2.1 传感器模块接口.....	56
4.2.1.1 超声波传感器.....	56
4.2.1.1.1 介绍.....	56
4.2.1.1.2 模块接口.....	58
4.2.1.1.3 约束.....	59
4.2.1.1.4 样例.....	59
4.2.1.2 二氧化碳传感器.....	59
4.2.1.2.1 介绍.....	59

4.2.1.2.2 模块接口.....	61
4.2.1.2.3 约束.....	62
4.2.1.2.4 样例.....	62
4.2.1.3 红外接收头传感器.....	63
4.2.1.3.1 介绍.....	63
4.2.1.3.2 模块接口.....	65
4.2.1.3.3 约束.....	67
4.2.1.3.4 样例.....	67
4.2.1.4 热电偶传感器.....	67
4.2.1.4.1 介绍.....	67
4.2.1.4.2 模块接口.....	69
4.2.1.4.3 约束.....	70
4.2.1.4.4 样例.....	70
4.2.1.5 陀螺仪加速度传感器.....	71
4.2.1.5.1 介绍.....	71
4.2.1.5.2 模块接口.....	73
4.2.1.5.3 约束.....	74
4.2.1.5.4 样例.....	74
4.2.1.6 温湿度传感器.....	74
4.2.1.6.1 介绍.....	74
4.2.1.6.2 模块接口.....	76
4.2.1.6.3 约束.....	78
4.2.1.6.4 样例.....	78
4.2.1.7 颜色传感器.....	78
4.2.1.7.1 介绍.....	78
4.2.1.7.2 模块接口.....	80
4.2.1.7.3 约束.....	81
4.2.1.7.4 样例.....	81
4.2.1.8 液位传感器.....	81
4.2.1.8.1 介绍.....	82
4.2.1.8.2 模块接口.....	83
4.2.1.8.3 约束.....	84
4.2.1.8.4 样例.....	84
4.2.1.9 震动传感器.....	84
4.2.1.9.1 介绍.....	84
4.2.1.9.2 模块接口.....	85
4.2.1.9.3 约束.....	87
4.2.1.9.4 样例.....	87
4.2.1.10 通用传感器.....	87
4.2.1.10.1 介绍.....	87
4.2.1.10.2 模块接口.....	90
4.2.1.10.3 约束.....	93

4.2.1.10.4 样例.....	93
4.2.2 控制模块接口.....	93
4.2.2.1 5V 步进电机.....	93
4.2.2.1.1 介绍.....	93
4.2.2.1.2 模块接口.....	95
4.2.2.1.3 约束.....	96
4.2.2.1.4 样例.....	96
4.2.2.2 42V 步进电机.....	96
4.2.2.2.1 介绍.....	96
4.2.2.2.2 模块接口.....	100
4.2.2.2.3 约束.....	101
4.2.2.2.4 样例.....	101
4.2.2.3 薄膜键盘.....	101
4.2.2.3.1 介绍.....	101
4.2.2.3.2 模块接口.....	102
4.2.2.3.3 约束.....	104
4.2.2.3.4 样例.....	104
4.2.2.4 舵机.....	104
4.2.2.4.1 介绍.....	105
4.2.2.4.2 模块接口.....	107
4.2.2.4.3 约束.....	114
4.2.2.4.4 样例.....	115
4.2.2.5 继电器.....	116
4.2.2.5.1 介绍.....	116
4.2.2.5.2 模块接口.....	119
4.2.2.5.3 约束.....	120
4.2.2.5.4 样例.....	120
4.2.2.6 旋转编码器 (ky-040)	120
4.2.2.6.1 介绍.....	120
4.2.2.6.2 模块接口.....	122
4.2.2.6.3 约束.....	123
4.2.2.6.4 样例.....	123
4.2.3 显示模块接口.....	124
4.2.3.1 8-8 点阵显示屏.....	124
4.2.3.1.1 介绍.....	124
4.2.3.1.2 模块接口.....	125
4.2.3.1.3 约束.....	126
4.2.3.1.4 样例.....	126
4.2.3.2 LCD1602 显示屏.....	127
4.2.3.2.1 介绍.....	127
4.2.3.2.2 模块接口.....	130
4.2.3.2.3 约束.....	132

4.2.3.2.4 样例.....	132
4.2.3.3 oled 显示屏.....	132
4.2.3.3.1 介绍.....	132
4.2.3.3.2 模块接口.....	134
4.2.3.3.3 约束.....	145
4.2.3.3.4 样例.....	146
4.2.3.4 三色灯.....	150
4.2.3.4.1 介绍.....	150
4.2.3.4.2 模块接口.....	151
4.2.3.4.3 约束.....	153
4.2.3.4.4 样例.....	153
4.2.3.5 tft 显示屏.....	154
4.2.3.5.1 介绍.....	154
4.2.3.5.2 模块接口.....	155
4.2.3.5.3 约束.....	168
4.2.3.5.4 样例.....	168
4.2.3.6 墨水屏.....	177
4.2.3.6.1 介绍.....	177
4.2.3.6.2 模块接口.....	178
4.2.3.6.3 约束.....	190
4.2.3.6.4 样例.....	190
4.2.3.7 像素软屏.....	197
4.2.3.7.1 介绍.....	197
4.2.3.7.2 模块接口.....	198
4.2.3.7.3 约束.....	200
4.2.3.7.4 样例.....	200
4.2.3.8 智能灯.....	200
4.2.3.8.1 介绍.....	200
4.2.3.8.2 模块接口.....	201
4.2.3.8.3 约束.....	205
4.2.3.8.4 样例.....	205
4.2.4 拓展模块接口.....	206
4.2.4.1 hilink.....	206
4.2.4.1.1 介绍.....	206
4.2.4.1.2 模块接口.....	206
4.2.4.1.3 约束.....	208
4.2.4.1.4 样例.....	209
4.2.4.2 ota.....	210
4.2.4.2.1 介绍.....	210
4.2.4.2.2 模块接口.....	210
4.2.4.2.3 约束.....	211
4.2.4.2.4 样例.....	211

4.2.4.2.5 升级包.....	212
4.2.4.3 system.....	212
4.2.4.3.1 介绍.....	213
4.2.4.3.2 模块接口.....	213
4.2.4.3.3 约束.....	214
4.2.4.3.4 样例.....	214
5 调试.....	215
5.1 安装.....	215
5.2 使用选项.....	215
5.3 指令.....	216
6 常见问题.....	220
7 约束与限制.....	222

1 简介

本章为MapleJS基本介绍。

1.1 基本介绍

MapleJS是华为推出的面向物联网（IoT）设备侧应用开发的轻量化JavaScript引擎，及其配套的开发工具集。MapleJS可以运行在LiteOS物联网实时操作系统之上并支持HiLink物联网协议，使得开发者能够在资源受限的嵌入式设备上使用JavaScript进行开发；并通过提供统一的设备能力抽象接口，向开发者屏蔽硬件差异，使其更加聚焦业务实现，从而提升IoT设备应用开发效率。

1.2 特点

- 轻量化：Flash占用小于100KB，空载时RAM占用小于32KB；
- 支持语言标准：支持ECMAScript 5.1标准；
- 垂直整合：与LiteOS整合优化，达到最优能耗/性能比。

1.3 各模块介绍

为了方便广大开发者的开发活动并进一步形成良好的生态，MapleJS提供了一整套完善的开发环境及开发资源。主要划分为以下四个部分。

- MapleJS引擎：对JS代码进行高效的解释执行；
- 开发工具套件：提供了一套完整的从编码、编译，到部署、调试的端到端的集成开发环境，并在开发周期中持续性提供辅助优化的能力；
- 面向设备型开发框架：支持事件驱动的编程模型，并提供统一的硬件抽象接口、系统抽象接口、网络协议接口等，让开发者能够方便快速调用，编程自由度得以进一步释放；
- 行业共享仓库：提供面向行业应用的共享库，便于第三方开发者快速开发行业应用。

2 环境准备及工具使用

本章主要介绍MapleJS运行所需的环境准备和相关工具使用。

2.1 环境准备及使用

2.2 初始JS文件加载工具

2.3 JavaScript SDK使用

2.1 环境准备及使用

2.1.1 版本获取

当前仅支持github路径获取。参考链接<https://github.com/HWMapleJS/MapleJS>。

2.1.2 目录介绍

详情请参考github目录指引。

2.1.3 如何与LiteOS及Hilink集成

i.引擎启动

然后请参考以下代码样例实现在LiteOS上启动JS引擎。

```
void
js_main(void)
{
    TSK_INIT_PARAM_S initparam;
    UINT32 ret = LOS_OK;
    initparam.pfnTaskEntry = (TSK_ENTRY_FUNC) create_maplejs_main_entry;
    initparam.usTaskPrio = 10;
    initparam.pcName = "create_maplejs_main_entry";
    initparam.uwStackSize = MAPLEJS_TASK_STACK_SIZE;
    ret = LOS_TaskCreate (&maplejs_main_task_id, &initparam);
    if (ret != LOS_OK)
    {
        print ("create maplejs_main_entry task fail!\r\n");
    }
}

void create_maplejs_main_entry()
{
    LOS_TaskDelay (2000);
    los_vfs_init ();
    maplejs_main_entry();
}
```

在工程的main()中添加js_main()的调用。

```
void main(void){ uint32_t uwRet; ... js_main(); ... }
```

ii.模块定制

默认当前目录下包含lib、tools、include三个目录，执行如下命令

```
sh ./tools/static-module-gen-gcc.sh
```

界面输出如下

```
1 : apal02      2 : crypto      3 : dht11      4 : fs          5 : gpio       6 : hall
7 : hilink      8 : i2c         9 : ky040     10 : ledrgb     11 : pwm       12 : relays
13 : rtc        14 : spi        15 : system   16 : tcs3200    17 : timer     18 : uart
19 : senseair_s8 20 : canvas     21 : objectPersistence 22 : lattice    23 : servo     24 : keyboard
25 : stepper    26 : nanostepper 27 : sensor   28 : ultrasound

Please Input the Numbers of Modules You Want to Load (Seperated by Comma):
If you want to choose some basic modules, including timer, hilink and uart, you can input number 0.
```

按提示输入想要选择加载的模块对应的编号,用逗号隔开。例如，想要选择timer, crypto, rtc模块，则输入17,2,13。如若想要选择默认模块组合（timer, hilink, uart）可输入0。界面出现新的提示如下

```
Please choose which realtek to be used: 1 for rtl8710. Please Enter 1:
```

要求选择即将使用的realtek类型，目前只支持realtek8710，请输入1。若输入其他数字，则不会得到正确的编译结果，且得到如下提示

```
Please choose valid realtek.
```

按如上提示执行后，在lib的子目录下生成不同的版本libmaplejs.a。例如，希望生成Debug_FS版本的库文件，则执行静态模块定制之后的库文件的路径为lib/Debug_FS/libmaplejs.a。

提示1：如执行报错，请先添加执行权限

```
chmod +x tools/static-module-gen.py
chmod +x tools/static-module-gen-gcc.sh
```

提示2：如出现linux与windows换行符不兼容的问题，请先使用以下命令将文件转成linux环境下可执行

```
dos2unix tools/static-module-gen.py
dos2unix tools/static-module-gen-gcc.sh
```

iii.与hilink工程集成

若使用hilink工程,将maplejs.h放在hilink/hilink_gcc_normalized/component/common/application/maplejs_sdk/include/路径下; 将libmaplejs.a库文件放在hilink/hilink_gcc_normalized/component/common/application/maplejs_sdk/lib/路径下。

然后在hilink/hilink_gcc_normalized/project/realtek_amebaz_va0_example/GCC-RELEASE/目录下执行make命令即可生成bin文件, 生成的bin文件路径为 hilink/hilink_gcc_normalized/project/realtek_amebaz_va0_example/GCC-RELEASE/application/Debug/bin。

2.1.4 如何生成/运行字节码文件

由于Release_Flash与Release_FS版本不支持parser和串口部署，因此无法部署和执行js文件，需先将js文件转换为字节码，然后转换为bin文件直接烧录到flash。

1. 使用tools/MapleJS-snapshot.exe 编译js文件生成字节码文件(后缀为snapshot), 在windows环境下使用下面的命令:

- `./tools/MapleJS-snapshot.exe generate file.js -o file.snapshot`生成一般snapshot文件；
- `./tools/MapleJS-snapshot.exe generate --static-snapshot file.js -o file.snapshot`生成静态snapshot文件，静态snapshot文件通常会消耗更少的引擎heap。

注：Release_FS只支持一般snapshot运行；Release_Flash两种类型的snapshot都支持。如果要生成静态snapshot，则JS脚本中出现number常量只能是28位有符号整形，否则无法生成静态 snapshot。

2. 采用tools/init-js-load-tool.exe将生成的字节码文件转换为bin文件，在windows环境下使用如下命令：`tools/init-js-load-tool.exe -p spiiffs -ping file.snapshot spiiffs_test.bin`关于预置工具的详细使用请参考2.2章节。

提示：如执行报错，请先添加执行权限

```
chmod +x tools/MapleJS-snapshot
chmod +x tools/init-js-load-tool
```

2.1.5 依赖

- LiteOS，支持文件系统；
- 爱联模组；
- Flash空间>100KB，内存>32KB。

2.1.6 JS-Generator压缩包介绍

linux环境下使用命令`tar -zvf MapleJS-Generator.tar.gz`进行解压 目录结构如下：

- MapleJS-Generator
 - README.md: 环境准备与使用说明
 - maplejs-generator.sh: 对接OpenLab平台脚本
 - package.json: 环境依赖文件
 - src/: JS-Generator生成器源码

2.2 初始 JS 文件加载工具

该工具可将JS文件加载至bin文件，生成的bin文件烧录到芯片即可实现JS文件内置到芯片中。

MapleJS内置JS脚本有如下两个特性：

1. 支持乒乓缓冲机制，可内置ping.js和pang.js两个脚本，MapleJS在启动时会依据配置文件加载ping.js或pang.js，其中配置文件将由该工具自动加载至bin文件中。乒乓缓冲机制在内部存在两个缓冲区，其中一个用来存储当前正在运行的JS脚本，当部署新JS脚本或升级JS脚本时会存储至另外一个缓冲区，从而在接收的同时保证当前运行的JS脚本不受影响，当接收完整之后，新接收的JS脚本将切换成正在运行的JS脚本，从而实现部署或升级时平滑过渡。
2. 支持两种方式存储内置JS脚本：
 - 基于raw flash存储内置JS脚本。（注：MapleJS目前版本不支持raw flash方案存储JS文件，但是之后版本将支持，将来工具也无需更新即可使用）
 - 基于spiiffs文件系统储存内置JS脚本。

该工具需通过命令行参数输入MapleJS内置JS脚本特性及其他参数，执行成功时将生成bin文件并打印烧录的起始地址，可使用ImageTool工具将bin文件烧录至芯片。

2.3.1 工具使用步骤

步骤1 打开命令提示符(cmd)并切换至init-js-load-tool.exe工具目录(在Windows文件浏览器地址栏输入cmd并回车可快速打开cmd并切到该目录)



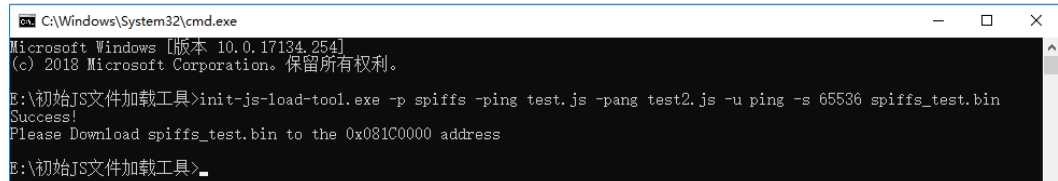
```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.17134.254]
(c) 2018 Microsoft Corporation. 保留所有权利。
E:\初始JS文件加载工具>
```

步骤2 根据实际情况输入相应命令及参数生成bin文件。

下面举例说明

MapleJS基于spiffs文件系统存储内置JS文件，需将本目录下test.js做为ping.js、test2.js作为pang.js，启动时最终加载ping.js：

```
init-js-load-tool.exe -p spiffs -ping test.js -pang test2.js -u ping -s 65536 spiffs_test.bin
```



```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.17134.254]
(c) 2018 Microsoft Corporation. 保留所有权利。
E:\初始JS文件加载工具>init-js-load-tool.exe -p spiffs -ping test.js -pang test2.js -u ping -s 65536 spiffs_test.bin
Success!
Please Download spiffs_test.bin to the 0x081C0000 address
E:\初始JS文件加载工具>
```

生成成功时，将打印：

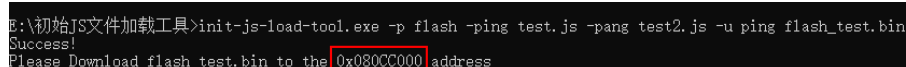
```
Success!
Please Download XXXX.bin to the 0xXXXX address
```

具体命令参数详情解释如下：

- p: 为芯片的存储文件的方式，选项只可为flash或spiffs，flash代表Raw Flash存储方式，spiffs代表spiffs文件系统方式，目前版本只能选择spiffs，选项必填
- ping: 指定目标文件ping.js的原始js文件路径，此选项必填
- pang: 指定目标文件pang.js的原始js文件路径，此选项非必填且无默认值
- u: 指定启动时加载ping.js还是pang.js，选项只可为ping或pang，其中ping表示加载ping.js，pang表示加载pang.js，此选项默认值为ping
- s: 指定spiffs模式时输出文件的大小，此参数须为4096的倍数，此选项有默认值为65536(64K)，取值范围为[12288, 262144]
- c: 指定芯片，当前只支持RTL8710芯片，此参数只可为RTL8710，RTL8710表示芯片为RTL8710，此选项默认值为RTL8710
- 最后一个参数为目标输出文件名，此选项必填且无默认值

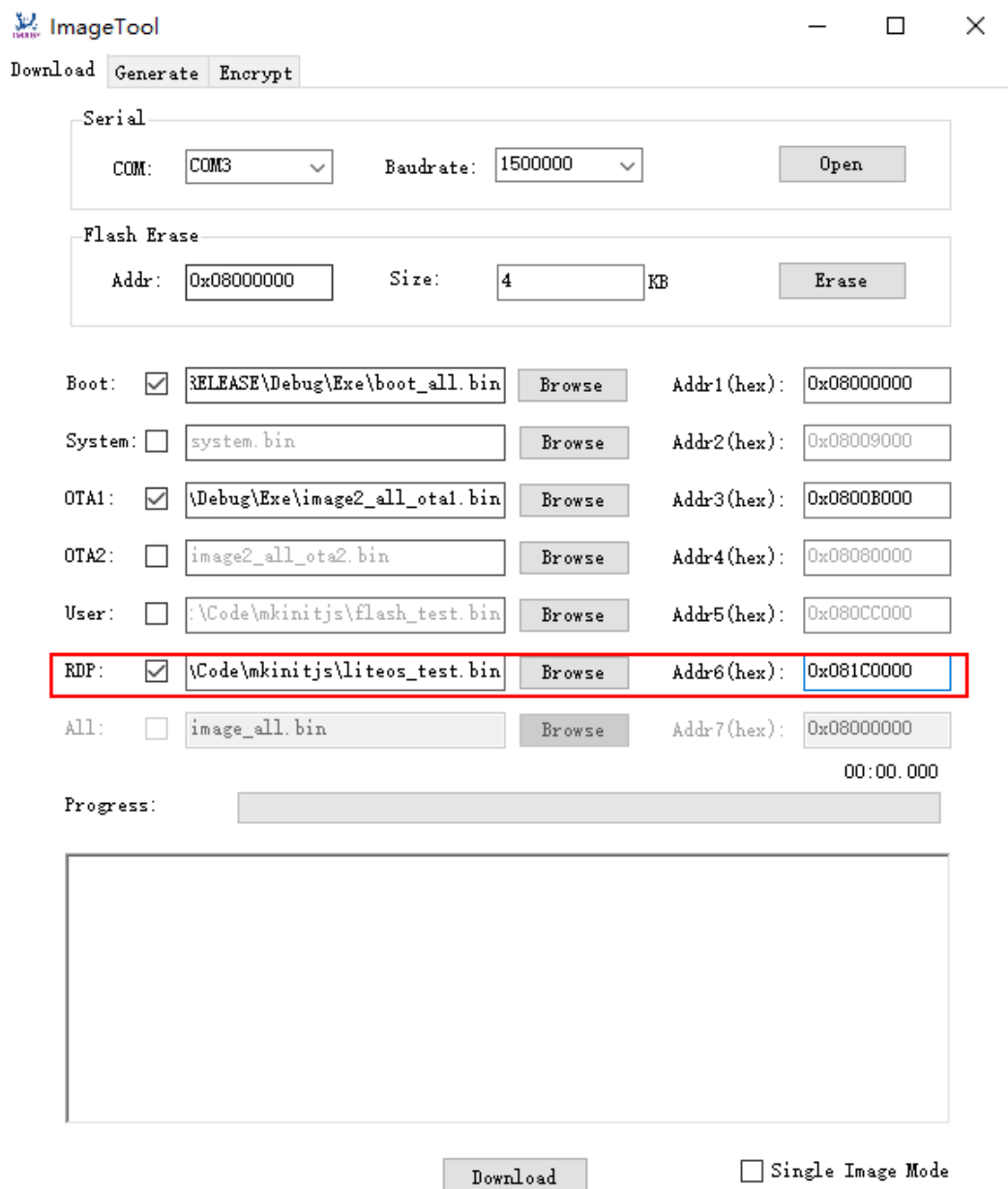
步骤3 选择需烧录的文件及输入地址信息。

在RDP栏中（或其他栏）选择步骤2生成的文件，并填入步骤2打印的烧录起始地址，具体如下图：



```
E:\初始JS文件加载工具>init-js-load-tool.exe -p flash -ping test.js -pang test2.js -u ping flash_test.bin
Success!
Please Download flash_test.bin to the 0x080CC000 address
```

注：请以步骤2打印的地址为准，因采用不同方式存储JS文件时其烧录的起始地址不同。



步骤4 烧录到芯片，烧录方法和烧录其他固件一样，此处省略。

----结束

2.3 JavaScript SDK 使用

JavaScript SDK是方便快捷基于MapleJS开发WIFI模组应用程序的开发包，其中包含部分自动生成的代码，自动生成的代码已支持智能家居App自动发现设备、注册设备、接收APP下发的控制命令等功能，并开放接口添加实际控制具体硬件通信的代码。

2.4.1 获取JavaScript SDK

开发者通过HiLink官网选择使用华为模组及JavaScript开发即可生成JavaScript SDK开发包，JavaScript SDK开发包为一个zip文件

其中zip文件中文件目录及内容说明如下：

```
.
├── doc
│   └── MapleJS开发指南.pdf
├── main.js
└── Tools
    └── init-js-load-tool
```

其中:

MapleJS开发指南.pdf 为MapleJS开发的指南

main.js 为生成的JavaScript代码框架

init-js-load-tool 为Linux下将JS文件加载至bin文件的工具

2.4.2 开发指引

JavaScript SDK中main.js为自动生成的模组侧JavaScript代码框架，可在该文件基础上开发自身业务

main.js主要承担两个功能:

1. 与云端通信（连接云端、接收云端下发的命令、上报数据给云端）
2. 与具体硬件通信（包括与MCU通信或者驱动外围硬件等）

main.js中代码已经包含云端通信的大部分代码，并开放出接口供添加与具体硬件通信的代码

2.4.2.1 main.js代码框架说明

main.js 主要包含三部分

i. 通用module加载

- 加载hilink模块，hilink模块主要提供了与云端通信的接口，这个模块为必选

```
var hilink = require('hilink');
```

- 若涉及定时和倒计时服务时，还需加载timer模块，用来巡检定时时间是否到

```
var timer = require('timer');
```

- 还可新增代码加载其他模块，具体可加载的模块可参考MapleJS开发指南.pdf第三章（模块接口介绍），如：增加uart模块

```
var uart = require('uart');
```

ii. services数据定义及其处理函数定义

与HiLink云端通信都是以service形式，main.js将各个service数据和方法都合并为一个services对象，同时各个具体服务为services中的子对象，键值即为service服务的ID，样例如：

```
var services = {};
services[service_id] = {
  // 该service的数据及方法定义
}
```

大部分service自动生成的代码如下，只需在ctrl中补充云端对该service发送put命令时，实际需操作外围硬件的代码。

```
services["switch1"] = {
  /*
   * this.data.on
   *   bool    0:关;1:开;
   */
  "pm": "GPR",
  "ctrl": function() {
    // print(JSON.stringify(this.data))
    // TODO:Adding code to control MCU or peripheral hardware
  }
}
```

具体某个service对象中主要如下几部分：

- 存放数据部分：定义了该service中数据，该部分在main.js中以注释体现，实际将在该对象的this.data子对象中。
- 操作权限：定义了该service的get/put/report操作权限
- get处理函数：云端获取该service数据时将调该函数，该函数的返回值将返回给云端，大部分情况下都无需定义，未定义时存在默认实现，默认实现为将this.data的数据转成字符串返回给云端
- put处理函数：云端设置该service数据时将调该函数，大部分情况下无需定义，未定义时存在默认实现，默认实现为将云端下发的data数据合并覆盖至this.data中
- control处理函数：云端针对该service下发put命令时，若该函数定义了，则会被调用，可在该函数中添加实际代码将this.data中数据发送给MCU或者控制其他外围硬件

```
services[service_id] = {
  "data" {
    /*
     * 存放该service的数据
     * 考虑到减少main.js的体积大小，这部分数据定义实际存在，但main.js只是用注释形式体现该数据名字及类型等信息
     */
  },
  /* pm为一个字符串，为该service的GET/PUT/REPORT权限，有相应权限则包含该权限的首字母 */
  "pm": "get/put/report permissions",
  "get": function() {
    /*
     * 云端获取该service数据时，将调该函数
     * 该函数的返回值将返回给云端
     * 若该函数若未定义时，有默认实现，即会将this.data的数据转成字符串返回给云端
     */
  },
  "put": function(data) {
    /*
     * 云端设置该service数据时，将调该函数
     * 该函数若未定义时，有默认实现，即会将云端下发的data数据合并覆盖至this.data中
     */
  },
  "ctl": function() {
    /*
     * 云端针对该service下发put命令时，若该函数定义了，则会被调用
     * 在该函数中，可将this.data中数据发送给MCU或者控制其他外围硬件
     */
    // print(JSON.stringify(this.data))
    // TODO:Adding code to control MCU or peripheral hardware
  }
}
```

iii. 函数实现

- hilink初始化参数函数：给MapleJS注入连接云端必要参数，此部分无需修改。

```
var init=function() {
  hilink.initParam('{"sn":"","prodId":"VZzx","model":"qq","dev_t":"005","manu":"FFF","mac":"","hiv":"1.0.0","fwv":"1.0.0","hwv":"1.0.0","swv":"1.0.0","prot_t":1}',
  "switch1,binarySwitch;switch2,binarySwitch;switch3,binarySwitch;switch4,binarySwitch;switchOn,binarySwitch;switchOff,binarySwitch;bb,bb_custom;nn,nn_custom;netInfo,netInfo;",
  "9D436E2CE869018C6D9B7A6341B942CF8527939EA3D1A29A1C8993C9C94F05D9DD217DA917A4DCFCB291FF9ED837C9BBD945E22D626C4D4B69B7507C2A35B458C523C178D9F0AE2FBF50C9E691F3E7FFCEAE71074C75D1F2D3B55408A451AF0EFC8074947CCA8BEECD6248F9E3D0C5B48B24F9DA819E5532C51BC38D0669483BD37357B99C0F27C9CE4A29B844F3D93E0BC39064C224CD4768D1E1889BBE09E1D077270A644A5C7D325AAB282AE53B942EF1EBECE2094BC3087FBD0FD780AEEBF961A326423AF4665071FC2A9159128235AE2298EF1DE5BABAC74E6AD4DB0642B345CDDAD2376DEAEA2C064100F42D48DB0EDBFD0B4E89229CDD8953F0F9A2",
  "71304f63502f61732326574a3a247a37d01e8385c01493215ee67aab5dc5e1817d9dd699c518d245fe92bd0261364fff")
};
init();
```


- hilink get事件处理函数

云端下发get命令时事件处理函数，其中hilink.getHandler是默认处理函数，默认处理为判断该service对应的对象是否存在get函数，若存在，则调get进行处理，若不存在，则使用默认的处理

```
hilink.on(hilink.GET, function(svc_id, instr) {  
    return hilink.getHandler(services, svc_id)  
});
```

- hilink put事件处理函数

云端下发put命令时事件处理函数，其中hilink.puthandler是默认处理函数，默认处理是判断该service对应的对象是否存在put函数，若存在，则调put进行处理，若不存在，则使用默认处理，再之后会判断是否存在ctrl函数，若存在，则调ctrl函数

```
hilink.on(hilink.PUT, function(svc_id, payload) {  
    hilink.putHandler(service, svc_id, payload);  
});
```

- 定时和倒计时时间处理函数

若涉及定时和倒计时服务时，此时下面这段将自动生成，其中hilink.runTimer为巡检定时时间是否满足，若满足之后，将会调注册的hilink.TIMER事件回调函数，可在回调函数中加代码处理定时时间到的逻辑

```
timer.setInterval(function() {  
    hilink.runTimer();  
}, 5000);  
hilink.on(hilink.TIMER, function(svc_id, para, value) {  
    // print("Timely arrival", "svc_id:", svc_id, ",value:", value);  
    // TODO: Add code to process Timely arrival.  
});
```

- 注释的UART处理函数部分

若模组是通过UART和MCU通信，则可把这部分代码注释去掉，并修改uart.open函数的参数。

```
/*  
var uart = require('uart')  
var port = uart.open({uartNum:1});  
*/  
/*  
function map_data(data, fc) {  
    var keys = Object.keys(data);  
    for(var i = 0; i < keys.length; ++i) {  
        fc(data[keys[i]]);  
    }  
}  
port.on(uart.DATA, 10, function(data) {  
    map_data(services, function(obj) {  
        if (obj && obj.sync) {  
            obj.sync(data)  
        }  
    })  
})  
*/
```

这里提供一种方式处理UART数据上报，其中port.on函数使用说明可参考MapleJS开发指南.pdf（参考UART模块接口），其UART回调函数中的逻辑为分别调各个service的sync函数（若sync存在），这样，若某个service需处理UART数据上报，则可在service对应的对象中定义sync函数，并添加如下逻辑：

```
services["switch1"] = {  
    /*  
    * this.data.on  
    *    bool    0:关;1:开;  
    */  
};
```

```
    */  
    "pm": "GPR",  
    "ctrl": function() {  
        // print(JSON.stringify(this.data))  
        // TODO:Adding code to control MCU or peripheral hardware  
    },  
    "sync": function(data) {  
        // TODO  
        // 判断UART数据是否修改该service对象  
        // 若修改，则判断是否改变  
        // 若改变，则  
        //     1. 保存值到this.data中  
        //     2. 若需上报给云端，则调hilink.upload(service_id, "")函数上报  
    }  
}
```

3 语法规格

本章介绍MapleJS支持的JavaScript语法规格。整体上，为了实现引擎轻量化，结合IoT设备侧的使用场景，MapleJS基于通用语法规格进行了部分语法裁剪，由于通用规格可参考业界标准，因此本章着重介绍裁剪部分。

3.1 语法标准

3.2 裁剪规格

3.1 语法标准

JavaScript 是一种动态类型、弱类型、基于原型的脚本语言。变量使用之前不需要类型声明，通常变量的类型是被赋值的那个值的类型。计算时可以不同类型之间对使用者透明地隐式转换，即使类型不正确，也能通过隐式转换来得到正确的类型。新对象继承对象（作为模版），将自身的属性共享给新对象，模版对象称为原型。这样新对象实例化后不但可以享有自己创建时和运行时定义的属性，而且可以享有原型对象的属性。

JavaScript 的核心是 ECMAScript，而 ECMAScript 是一个由 ECMA International 进行标准化，TC39 委员会进行监督的语言。它规定了语言的组成部分：语法、类型、语句、关键字、保留字、操作符、对象。我们支持的语法规则为 ECMAScript 5 (ES5)。它是 ECMAScript 的第五版修订，于 2009 年完成标准化。这个规范在 Web 领域，已经被所有现代浏览器相当完全的实现了。

基于目前 IoT 设备的特点，我们基于 ES 5.1 进行了一些语法的裁剪，即 3.2 裁剪规格展示的裁剪项不被支持。如果使用 3.2 节展示的语法，则会得到未定义错误。

3.2 裁剪规格

3.2.1 裁剪delete操作符 (ES 5.1 11.4.1)

delete 操作符用于删除对象的某个属性；如果没有指向这个属性的引用，那它最终会被释放。

示例：

```
var o = {};  
o.x = new Object();  
delete o.x;
```

3.2.2 裁剪void运算符 (ES 5.1 11.4.2)

`void` 运算符 对给定的表达式进行求值，然后返回 `undefined`。

示例:

```
void(0);
```

3.2.3 裁剪`typeof`运算符 (ES 5.1 11.4.3)

`typeof` 操作符返回一个字符串，表示未经计算的操作数的类型。

示例:

```
print(typeof 42)           // expected output: number
print(typeof 'balabala')  // expected output: string
print(typeof true)        // expected output: boolean
Function("return typeof this;")
```

3.2.4 裁剪`instanceof`运算符 (ES 5.1 11.8.6)

`instanceof` 运算符用来检测 `constructor.prototype` 是否存在于参数 `object` 的原型链上。

示例:

```
function C() {} var o = new C();
o instanceof C; // expected output: true
// 因为Object.getPrototypeOf(o) === C.prototype
```

3.2.5 裁剪`in`运算符 (ES 5.1 11.8.7)

如果指定的属性在指定的对象或其原型链中，则 `in` 运算符返回`true`。

示例1:

```
var arr = [1, 2, 3];
2 in arr // true
3 in arr // true
```

示例2:

```
var evalStr =
'for (var x in this) {\n' + '}\n';
eval(evalStr);
```

3.2.6 裁剪`do...while`语句 (ES 5.1 12.6.1)

`do...while` 语句创建一个执行指定语句的循环，直到 `condition` 值为 `false`。在执行 `statement` 后检测 `condition`，所以指定的 `statement` 至少执行一次。

```
do
  statement
while (condition);
```

statement

执行至少一次的语句，并在每次 `condition` 值为真时重新执行。想执行多行语句，可使用`block`语句（`{ ... }`）包裹这些语句。

condition

循环中每次都会计算的表达式。如果 `condition` 值为真，`statement` 会再次执行。当 `condition` 值为假，则跳到 `do...while` 之后的语句。

示例:

```
var i = 0;
do {
  i += 2;
} while( i<10 ); //do-while
```

3.2.7 裁剪for...in语句 (ES 5.1 12.6.4)

for...in 语句以任意顺序遍历一个对象的可枚举属性。对于每个不同的属性，语句都会被执行。

```
for (variable in object) {...}
```

variable

在每次迭代时，将不同的属性名分配给变量。

object

被迭代枚举其属性的对象。

示例:

```
var arr = [1, 2, 3];
for (var i in arr) { //for in
    var j = i;
}
```

3.2.8 裁剪with语句 (ES 5.1 12.10)

with 语句 扩展一个语句的作用域链。

```
with (expression) {
    statement
}
```

expression

将给定的表达式添加到在评估语句时使用的作用域链上。表达式周围的括号是必需的。

statement

任何语句。要执行多个语句，请使用一个块语句 ({ ... })对这些语句进行分组。

示例:

```
var obj2 = { x: 2 };
with (obj2) { //with
    var t = x;
}
```

3.2.9 裁剪label语句 (ES 5.1 12.12)

label语句可以和 break 或 continue 语句一起使用。标记就是在一条语句前面加个可以引用的标识符。

```
label :
    statement
```

label

任何不是保留关键字的 JavaScript 标识符。

statement

语句。break 可用于任何标记语句，而 continue 可用于循环标记语句。

示例:

```
var i, j;
loop1:
```

```
for (i = 0; i < 3; i++) { //The first for statement is labeled "loop1"
  loop2: for (j = 0; j < 3; j++) { //The second for statement is labeled "loop2"
    if (i == 1 && j == 1) {
      continue loop1;
    }
    print("i = " + i + ", j = " + j);
  }
}

// expected output:
// "i = 0, j = 0"
// "i = 0, j = 1"
// "i = 0, j = 2"
// "i = 1, j = 0"
// "i = 2, j = 0"
// "i = 2, j = 1"
// "i = 2, j = 2"
```

3.2.10 裁剪throw语句 (ES 5.1 12.13)

throw 语句用来抛出一个用户自定义的异常。当前函数的执行将被停止（throw之后的语句将不会执行），并且控制将被传递到调用堆栈中的第一个catch块。如果调用者函数中没有catch块，程序将会终止。

示例：

```
function getRectArea(width, height) {
  if (isNaN(width) || isNaN(height)) {
    throw "Parameter is not a number!";
  }
}

try {
  getRectArea(3, 'A');
} catch(e) {
  print(e);
  // expected output: "Parameter is not a number!"
}
```

3.2.11 裁剪try语句 (ES 5.1 12.14)

try...catch 语句将能引发错误的代码放在 try 块中，并且对应一个响应，然后有异常被抛出。

```
try {
  try_statements
}
[catch (exception_var_1 if condition_1) { // non-standard
  catch_statements_1
}]
...
[catch (exception_var_2) {
  catch_statements_2
}]
[finally {
  finally_statements
}]
```

try_statements

需要被执行的语句。

catch_statements_1, catch_statements_2

如果在try块里有异常被抛出时执行的语句。

exception_var_1, exception_var_2

用于保存关联catch子句的异常对象的标识符。

condition_1

一个条件表达式。

finally_statements

在try语句块之后执行的语句块。无论是否有异常抛出或捕获这些语句都将执行。

示例:

```
try {
    throw "myException"; // generates an exception
} catch (e) {
    // statements to handle any exceptions
    logMyErrors(e); // pass exception object to error handler
}
```

3.2.12 裁剪debugger语句 (ES 5.1 12.15)

debugger 语句调用任何可用的调试功能，例如设置断点。如果没有调试功能可用，则此语句不起作用。

示例:

```
debugger; // do potentially buggy stuff to examine, step through, etc.
```

3.2.13 裁剪NaN属性 (ES 5.1 15.1.1.1)

NaN 属性用于引用特殊的非数字值。

示例:

```
var i = NaN;
```

3.2.14 裁剪Infinity属性 (ES 5.1 15.1.1.2)

全局属性 Infinity 是一个数值，表示无穷大。

示例:

```
var i = Infinity;
```

3.2.15 裁剪parseInt函数 (ES 5.1 15.1.2.2)

parseInt() 函数可解析一个字符串，并返回一个整数。

示例:

```
var i = parseInt("8", 10);
```

3.2.16 裁剪parseFloat函数 (ES 5.1 15.1.2.3)

parseFloat() 函数可解析一个字符串，并返回一个浮点数。

示例:

```
var i = parseFloat("1.2");
```

3.2.17 裁剪isNaN函数 (ES 5.1 15.1.2.4)

isNaN() 函数用于检查其参数是否是非数字值。

示例:

```
isNaN(3); // false
```

3.2.18 裁剪isFinite函数 (ES 5.1 15.1.2.5)

该全局 isFinite() 函数用来判断被传入的参数值是否为一个有限数值（finite number）。在必要情况下，参数会首先转为一个数值。

示例：

```
isFinite(3); // true
```

3.2.19 裁剪decodeURI函数 (ES 5.1 15.1.3.1)

decodeURI() 函数可对 encodeURI() 函数编码过的 URI 进行解码。

示例：

```
var i = "http://www.huawei.com";  
decodeURI(i);
```

3.2.20 裁剪decodeURIComponent函数 (ES 5.1 15.1.3.2)

decodeURIComponent() 函数可对 encodeURIComponent() 函数编码的 URI 进行解码。

示例：

```
var i = "http://www.huawei.com";  
decodeURIComponent(i);
```

3.2.21 裁剪encodeURI函数 (ES 5.1 15.1.3.3)

encodeURI() 函数可把字符串作为 URI 进行编码。

示例：

```
var i = "http://www.huawei.com";  
encodeURI(i);
```

3.2.22 裁剪encodeURIComponent函数 (ES 5.1 15.1.3.4)

encodeURIComponent() 函数可把字符串作为 URI 组件进行编码。

示例：

```
var i = "http://www.huawei.com";  
encodeURIComponent(i);
```

3.2.23 裁剪String对象原型substr方法 (ES 5.1 B.2.3)

substr 方法有两个参数 start 和 length，用于将this对象转换为一个字符串，并返回这个字符串中从 start 位置一直到 length 位置（或如果 length 是 undefined，就一直到了字符串结束位置）的字符组成的子串。

示例：

```
var str = "Hello world!";  
var substr = str.substr(3, 7);
```

3.2.24 裁剪Error对象 (ES 5.1 15.11)

Error对象的实例在运行时遇到错误的情况下会被当做异常抛出。Error对象也可以作为用户自定义异常类的基对象。

```
15.11 Error Objects  
15.11.1 The Error Constructor Called as a Function  
15.11.2 The Error Constructor  
15.11.3 Properties of the Error Constructor  
15.11.4 Properties of the Error Prototype Object  
15.11.5 Properties of Error Instances
```



```
15.11.6 Native Error Types Used in This Standard
15.11.6.1 EvalError
15.11.6.2 RangeError
15.11.6.3 ReferenceError
15.11.6.4 SyntaxError
15.11.6.5 TypeError
15.11.6.6 URIError
15.11.7 NativeError Object Structure
```

除了通用的Error构造函数外,JavaScript还有6个其他类型的错误构造函数: **EvalError**: eval函数没有被正确执行时抛出此错误, 该错误类型已经不再在ES5中出现了, 只是为了保证与以前代码兼容才继续保留。 **RangeError**: 当一个值超出有效范围时发生的错误, 主要有数组长度为负数、number对象的方法参数超出范围、函数堆栈超过最大值。 **ReferenceError**: 引用一个不存在的变量时发生的错误或者将一个值分配给无法分配的对象, 比如对函数的运行结果或者this赋值。 **SyntaxError**: 解析代码时发生的语法错误, 比如变量名错误、缺少括号等。 **TypeError**: 当变量或参数不是预期类型时发生的错误, 比如对字符串、布尔值、数值等原始类型的值使用new命令就会抛出这种错误。 **URIError**: 当URI相关函数的参数不正确时抛出的错误, 主要涉及encodeURIComponent、decodeURIComponent、encodeURIComponent、escape、unescape这六个函数。

示例1:

```
try {
    throw new Error("Whoops!");
} catch (e) {
    print(e.name + ": " + e.message);
}
```

示例2:

```
var x = new Error("This is an error");
if (x.constructor == Error)
    print("Object is an error.");
```

示例3:

```
var check = function(num) {
    if (num < 0 || num > 100) {
        throw new RangeError('Parameter must be between ' + 0 + ' and ' + 100);
    }
};

try {
    check(500);
}
catch (e) {
    if (e instanceof RangeError) {
        print(e.name + ": " + e.message);
    }
}
```

示例4:

```
try {
    var a = b;
}
catch (e) {
    print(e instanceof ReferenceError);
}
```

3.2.25 不带parser时不支持eval语句 (ES 5.1 15.1.2.1)

eval()是全局对象的一个函数属性。eval()的参数是一个字符串。如果字符串表示的是表达式, eval()会对表达式进行求值。如果参数表示一个或多个JavaScript语句, 那么eval()就会执行这些语句。

示例:

```
eval('1+1');  
eval('var a = 1;')
```

3.2.26 裁剪escape函数 (ES 5.1 B.2.1)

escape函数是全局对象的一个属性。它通过将一些字符替换成十六进制转义序列，计算出一个新字符串值。对于代码单元小于等于0xFF的被替换字符，使用 %xx 格式的两位数转义序列。对于代码单元大于0xFF的被替换字符，使用 %uxxxx 格式的四位数转义序列。

示例:

```
escape("hello!"); //expected output: "hello%21"
```

3.2.27 不带parser时不支持Function构造函数 (ES 5.1 15.3.1-15.3.2)

Function构造函数创建一个新的Function对象。

示例:

```
var f = new Function('a', 'b', 'return a + b');
```

3.2.28 裁剪unescape函数(ES 5.1 B.2.2)

unescape函数可对escape编码的字符串进行解码。

示例:

```
unescape("hello%21"); //expected output: "hello!"
```

3.2.29 裁剪Date对象 (ES 5.1 15.9)

Date对象用于处理日期与时间。

示例:

```
var today = new Date()  
var d1 = new Date("September 29, 2018 19:30:00")  
var d2 = new Date(18, 9, 29)  
var d3 = new Date(18, 9, 29, 19, 30, 0)
```

3.2.30 裁剪正则表达式 (ES 5.1 7.8.5 15.10)

正则表达式描述了一种字符串匹配的模式。

示例:

```
var re = /.at/i; //正则表达式字面量，匹配第一个以"at"结尾的3个字符的组合，不区分大小写  
var r = RegExp ("a"); //正则表达式对象
```

4 模块接口

本章主要面向的对象为模组厂商开发人员，智能家电厂商开发人员，智能家居方案厂商开发人员以及个人开发人员。

本章模块接口分为基础模块接口和行业使能库两大部分：基础模块接口主要是芯片内（模组CPU）提供的相关的能力支持；行业使能库主要展示芯片外扩展模块的能力支持。

本章目的是给开发者提供可以直接操作硬件、系统功能、网络连接的能力。开发者可以在使用具体模块时，可在JS代码直接中引用相应模块，从而获得该模块接口所提供的能力。

模块在使用前需要进行模块定制，具体请参见第二章2.1小节的2.1.5。

4.1 基础模块接口

4.2 行业使能库

4.1 基础模块接口

本节为MapleJS内所包含的基础模块信息，可供开发者参考使用。

4.1.1 buffer

4.1.1.1 介绍

该buffer模块与NodeJS中提供的buffer相似，功能有缩减。其中如果参数为小数，则统一向下取整。

4.1.1.2 模块接口

A 申请buffer对象

```
new Buffer(size or string or array);
```

功能描述

申请一个新的buffer对象。

接口约束

无。

参数列表

- **size**: 分配一个大小为 **size** 字节的新建的 **buffer**, **buffer**所有字节初始化为零。要求满足性质 $0 \leq \text{size} \leq 2147483647(0x7fffffff)$ 。
- **string**: 创建一个包含给定字符串 **string** 的 **buffer**, **buffer**长度为**string**的长度。
- **number array**: 将传入的 **number array** 数据拷贝到一个新建的 **buffer**, 如果**array**中的元素不是**number**, 会当做0处理, 若**number**值不在0-255范围内, 会强转为**uint8**类型再存储。**buffer**长度为**array**的长度。

返回值

返回一个新的**buffer**对象。

接口示例

```
buf = new Buffer(10);
```

B 获取buffer长度

```
buf.length();
```

功能描述

获取**buf**的长度, 它是不可修改的数值属性。

接口约束

无。

参数列表

无。

返回值

返回**buf**的长度。

接口示例

```
buf = new Buffer(10);  
print (buf.length());
```

C 读取无符号8位整数

```
buf.readUInt8(offset);
```

功能描述

从指定偏移量的**buf**读取无符号8位整数。

接口约束

无。

参数列表

- **offset** : 整数类型, 开始读取之前要跳过的字节数。要求满足性质 $0 \leq \text{offset} \leq \text{buf.length} - 1$ 。

返回值

从指定偏移量的**buf**读取无符号8位整数。

接口示例

```
buf=new Buffer(10);           //创建一个长度为10的 buf
for (var i=0;i<10;i++) {      //buf的第i个字节写入值i
    buf.writeUInt8(i,i)
}
for (var i=0;i<10;i++) {      //读取buf的第i个字节并打印
    print(buf.readUInt8(i))
}
```

D 将值写入指定偏移量的buf

```
buf.writeUInt8(value, offset);
```

功能描述

将值写入指定偏移量的buf。

接口约束

无。

参数列表

- **value**：整数类型，要写入buf的数字。要求满足性质 $0 \leq \text{value} \leq 255(0\text{xff})$ 。
- **offset**：整数类型，开始写入之前要跳过的字节数。要求满足性质 $0 \leq \text{offset} \leq \text{buf.length} - 1$ 。

返回值

offset加上写入的字节数。

接口示例

```
buf=new Buffer(10);           //创建一个长度为10的 buf
for (var i=0;i<10;i++) {      //buf的第i个字节写入值i
    buf.writeUInt8(i,i)
}
```

E 字符串转码

```
buf.toString(encoding);
```

功能描述

按照指定字符编码将buf解码成字符串。

接口约束

无。

参数列表

- **encoding**：string类型，指定字符的编码，支持utf8, hex, ascii三种字符编码，默认值为utf8。

返回值

按照指定字符编码将buf解码成字符串。

接口示例

```
buf=new Buffer("hello");
print(buf.toString("utf8"));
```

F 字符串拷贝

```
buf.copy(target, targetStart, sourceStart, sourceEnd);
```

功能描述

拷贝 buf 中某个区域的数据到 target 中的某个区域。

接口约束

无。

参数列表

- target 要拷贝进的 buffer。
- targetStart target 中开始写入的偏移量。要求满足 $0 \leq \text{targetStart} \leq \text{target.length} - 1$ ，默认为 0。
- sourceStart buf 中开始拷贝的偏移量。要求满足 $0 \leq \text{sourceStart} \leq \text{buf.length} - 1$ ，默认为 0。
- sourceEnd buf 中结束拷贝的偏移量（不包含）。要求满足 $\text{sourceStart} < \text{sourceEnd} \leq \text{buf.length}$ ，默认为 buf.length。

返回值

- 拷贝的字节数。

接口示例

```
buf_target=new Buffer(10);      //创建一个长度为10的buf_target
for (var i=0;i<10;i++) {        //将buf_target中的值全部置为0
    buf_target.writeUInt8(0,i)
}
buf.copy(buf_target,3,3,10);    //拷贝buf第3至10字节偏移量的数据到buf_target第3字节偏移量开始
```

G buffer填充

```
buf.fill(value, offset, end , encoding);
```

功能描述

用指定的 value 填充 buf指定长度。如果没有指定 offset 与 end，则填充整个 buf。要求满足 $\text{end} - \text{offset} \leq \text{buf.length}$ 。

接口约束

无。

参数列表

- value 是用来填充 buf 的值, value的类型可以是string, buffer, integer或Array。其中如果value是一个数值的话, 则会被转换成介于0到255的整数; value如果是Array的话, 只允许数组的元素全是数值, 同样其中的每个数值都会被转换成介于0到255的整数。
- offset 开始填充 buf 的偏移量。默认为 0。要求满足 $0 \leq \text{offset} \leq \text{buf.length} - 1$ 。
- end 结束填充 buf 的偏移量（不包含）。默认为 buf.length。要求满足 $\text{offset} < \text{end} \leq \text{buf.length}$ 。
- encoding 如果 value 是字符串, 则指定 value 的字符编码。默认和目前只支持utf8编码格式。使用buf.UTF8表示。

返回值

- buf 的引用。

接口示例

```
var val_int=0x0b;
var buf_num=new Buffer([00, 00, 00, 00, 00]);
print(buf_num.fill(val_int).toString("hex"));
```

4.1.1.3 约束

无。

4.1.1.4 样例

介绍

本样例主要包括buffer模块相关的接口调用示例。

例程

```
buf=new Buffer("hello");           //创建一个包含给定字符串 hello 的 buf
print(buf);                         //依据utf8字符编码打印buf

print(buf.toString());              //依据utf8字符编码打印buf
print(buf.toString("utf8"));        //依据utf8字符编码打印buf
print(buf.toString("hex"));         //依据hex字符编码打印buf
print(buf.toString("ascii"));       //依据ascii字符编码打印buf

buf=new Buffer(10);                  //创建一个长度为10的 buf
for (var i=0;i<10;i++) {            //buf的第i个字节写入值i
    buf.writeUInt8(i,i)
}
for (var i=0;i<10;i++) {            //读取buf的第i个字节并打印
    print(buf.readUInt8(i))
}

buf_target=new Buffer(10);           //创建一个长度为10的buf_target
for (var i=0;i<10;i++) {            //将buf_target中的值全部置为0
    buf_target.writeUInt8(0,i)
}

buf.copy(buf_target, 3, 3, 10);      //拷贝buf第3至10字节偏移量的数据到buf_target第3字节偏移量开始
for (var i=0;i<10;i++) {            //读取buf_target的第i个字节并打印
    print(buf_target.readUInt8(i))
}

var val_str='HELLO';
var val_int=0x0b;
var val_arr=[0x01, 0x02, 0x03, 0x04, 0x05];

var buf_num=new Buffer([00, 00, 00, 00, 00]);
var buf_dest=new Buffer("helloworld");
var buf_src=new Buffer(val_str);

print(buf_num.toString('hex'));      //Buffer原始值: <0 0 0 0 0>

print(buf_num.fill(val_int).toString("hex")); //向Buffer中填充数值0x0b,
//填充后结果:<0b 0b 0b 0b 0b>
print(buf_num.fill(val_arr).toString("hex")); //向Buffer中填充数组,
//填充后结果:<01 02 03 04 05>
print(buf_dest);                     //Buffer原始值: helloworld
print(buf_dest.fill(buf_src, 0, 5)); //向Buffer中填充新Buffer中内容,
//填充后结果:HELLOWorld
print(buf_dest.fill('w'));           //向Buffer中填充字符'w',
//填充后结果: wwwwwwwwww
```

4.1.2 crypto

4.1.2.1 介绍

Crypto模块主要提供了多种加密算法供用户使用。

主要提供了如下加解密算法AES_CBC,AES_ECB密钥长度为128位；“DES3_CBC”密钥长度为192位。

提供了如下的hash算法MD5，SHA2_256，以及与HMAC结合的哈希算法HMAC-MD5，HMAC-SHA2_256。不推荐使用MD5，AES_ECB不安全加密算法。

4.1.2.2 模块接口

A 引用模块

```
var crypto = require("crypto");
```

B 加密数据

```
crypto.encrypt(algo, message, key );
```

功能描述

加密字符串或数据（16进制字符串）。

接口约束

入参数数据类型需要与接口原型一致。

参数列表

- **algo**: 加密算法的名称，枚举类型。目前包括AES_CBC，AES_ECB，DES3_CBC。
- **message**: 要加密的信息，buffer类型。可以是字符串或者数据，其长度必须是16字节长度的倍数。
- **key**: 加密算法的密钥，buffer类型，其中AES的密钥长度为16字节，3DES的密钥长度为24字节。

返回值

加密数据，buffer类型。该加密数据为二进制字符串。可以通过result.toString("hex")获取文本格式。

接口示例

```
buf_key = new Buffer("1023456789abcdef12345678");  
buf_message = new Buffer([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5]);  
enresult = crypto.encrypt(crypto.DES3_CBC, buf_message, buf_key);
```

C 解密数据

```
crypto.decrypt(algo, message );
```

功能描述

将加密之后的密文进行解密。

接口约束

解密方式需要与加密方式相同才能正确解密。

参数列表

- algo: 枚举类型，加密算法的名称。目前包括AES_CBC, AES_ECB, DES3_CBC。
- message: buffer类型，加密后的数据（crypto.encrypt（）的返回值）。

返回值

解密数据，buffer类型。该加密数据为二进制字符串。可以通过result.toString("hex")获取文本格式。

接口示例

```
buf_key = new Buffer("1023456789abcdef12345678");
buf_message = new Buffer([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5]);
enresult = crypto.encrypt(crypto.DES3_CBC, buf_message, buf_key);
deresult = crypto.decrypt(crypto.DES3_CBC, enresult);
```

D 哈希加密

```
crypto.hash(algo, message);
```

功能描述

使用不同hash算法对字符串加密。

参数列表

- algo: 枚举类型，哈希算法的名称。目前包括MD5, SHA2_256。
- message: buffer类型，要加密的信息。

返回值

加密结果，buffer类型。该加密数据为二进制字符串。可以通过result.toString("hex")获取文本格式。

接口示例

```
buf_message = new Buffer([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5]);
result = crypto.hash(crypto.SHA2_256, buf_message);
```

E 哈希加密验证

```
crypto.hmac_hash(algo, key, message);
```

功能描述

计算不同hash算法同时结合哈希消息验证码对字符串处理后的值。

参数列表

- algo: 枚举类型，哈希算法的名称。目前包括HMAC_MD5, HMAC_SHA2_256。
- key: buffer类型，加密的密钥，无长度限制。
- message: buffer类型，要加密的信息。

返回值

加密结果，buffer类型。该加密数据为二进制字符串。可以通过result.toString("hex")获取文本格式。

接口示例

```
buf_key = new Buffer("1023456789abcdef12345678");
buf_message = new Buffer([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5]);
result = crypto.hmacHash(crypto.HMAC_SHA2_256, buf_key, buf_message);
```

4.1.2.3 约束

无。

4.1.2.4 样例

介绍

展示各种加密算法的使用。

例程

```
var crypto = require("crypto");
//接口中的参数message 和 key都是通过buffer传递的，即message和key既可以用字符串也可以用数组。
buf_key = new Buffer("1023456789abcdef12345678");
buf_message = new Buffer([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5]);
enresult = crypto.encrypt(crypto.DES3_CBC, buf_message, buf_key);
print("encrypt data:");
print(enresult.toString("hex"));
deresult = crypto.decrypt(crypto.DES3_CBC, enresult);
print("decrypt data:");
print(deresult.toString("hex"));
result = crypto.hash(crypto.SHA2_256, buf_message);
print("hash data:");
print(result.toString("hex"));
hmac_result = crypto.hmacHash(crypto.HMAC_SHA2_256, buf_key, buf_message);
print("hmacHash data:");
print(hmac_result.toString("hex"));
```

4.1.3 fs

4.1.3.1 介绍

fs提供操作文件系统的相关接口。

4.1.3.2 模块接口

A 引用模块

```
var fs = require("fs");
```

B 打开或创建文件

```
openSync (config);
```

功能描述

创建一个新的文件或者打开一个已有的文件，这个调用会产生一个文件描述符(以下简称fd)，fd包含了以特定方式控制文件的所有必要的信息。

接口约束

无。

参数列表

- config主要包含两个基本参数，path和mode
 - path，文件路径
 - mode，是对于相关文件的操作模式选择，支持模式和内容大致如下所示

w	打开一个文本文件，允许写入文件。如果文件不存在，则会创建一个新文件。在这里，程序会从文件的开头写入内容。如果文件存在，则该会被截断为零长度，重新写入。
r	打开一个已有的文本文件，允许读取文件。
a	打开一个文本文件，以追加模式写入文件。如果文件不存在，则会创建一个新文件。在这里，程序会在已有的文件内容中追加内容。
w+	打开一个文本文件，允许读写文件。如果文件已存在，则文件会被截断为零长度，如果文件不存在，则会创建一个新文件。
r+	打开一个文本文件，允许读写文件。如果是写入的话，是从头开始写入。

返回值

文件描述符，用于读写文件被打开/创建的文件。

接口示例

```
/* 以写方式打开（如果文件不存在则创建）hello1.js, */
var config = {
  path: "/hello1.js",
  mode: "w"
};
var fd = fs.openSync(config);
```

C 向文件内写入内容

```
writeSync (fd, str);
```

功能描述

向特定文件中写入特定字符串内容。

接口约束

fd为被写方式打开的文件的文件描述符才能正常写操作。

参数列表

- fd: 被写方式打开的文件的文件描述符
- str: 准备写入的字符串

返回值

写入字符串的长度。

接口示例

```
/* Build a new file and write the context */
var config = {
```

```
    path: "/hello1.js",  
    mode: "w"  
};  
var fd = fs.openSync(config);  
var num = fs.writeSync(fd, "helloworld!!!");
```

D 读取文件内容

```
readSync (fd);
```

功能描述

从特定文件读取内容。

接口约束

fd为被读方式打开的文件的文件描述符才能正常读操作。

被读文件大小不可超过4k。

参数列表

- fd: 被读方式打开的文件的文件描述符。

返回值

读到的字符串。

接口示例

```
/* Read strings from the preset script */  
var config1 = {  
    path: "/hello1.js",  
    mode: "r"  
};  
var fd1 = fs.openSync(config1);  
var data = fs.readSync(fd1);
```

E 关闭文件

```
closeSync (fd);
```

功能描述

关闭文件。

接口约束

无。

参数列表

- fd: 被打开的文件的文件描述符。

返回值

无。

接口示例

```
fs.closeSync (fd);
```

F 删除文件

```
unlinkSync (path);
```

功能描述

删除选中路径中已存在的特定文件。

接口约束

被删除文件需存在，否则程序报错。

参数列表

- path: 被删除文件的绝对路径

返回值

无。

接口示例

```
fs.unlinkSync('/hello1.js');
```

4.1.3.3 约束

1. 文件路径都是以 '/' 开头，文件命名不能以 '!' 开头（避免和系统文件重复），不支持操作系统文件。
2. 文件内容读取最大值为 1024*4 个字节。
3. 只支持 read/write 相应 open 的文件，delete 功能可以独立使用; 不使用文件应及时 close。
4. 当前文件系统操作暂时仅支持同步方式。

4.1.3.4 样例

介绍

文件读、写及删除。

例程

```
var fs = require("fs");

/* Build a new file and write the context */
var config = {
  path: "/hello1.js",
  mode: "w"
};
var fd = fs.openSync(config);
var num = fs.writeSync(fd, "hellowold!!!");
print(num);
fs.closeSync(fd);

/* Read strings from the preset script */
var config1 = {
  path: "/hello1.js",
  mode: "r"
};
var fd1 = fs.openSync(config1);
var data = fs.readSync(fd1);
print(data);
fs.closeSync(fd1);

/* Delete the new built file */
fs.unlinkSync('/hello1.js');
```

4.1.4 gpio

4.1.4.1 介绍

gpio是一种通用的I/O端口,gpio模块可以允许用户通过API调用驱动gpio端口进行读、写、监听等功能。

gpio端口方向的枚举定义

- gpio.DIR_IN: input方向;
- gpio.DIR_OUT: output方向;
- gpio.DIR_INOUT: input/output方向;

gpio端口状态的枚举定义

- gpio.PULLNONE: 不拉输入/输出;
- gpio.PULLUP: 上拉输入/输出;
- gpio.PULLDOWN: 下拉输入/输出;
- gpio.OPENDRAIN: 开漏输出,仅在端口方向为输出时可用;

gpio监听事件的枚举定义

- gpio.INT_RISING: 上升沿监听;
- gpio.INT_FALLING: 下降沿监听;
- gpio.INT_ANY: 任意边沿监听;

4.1.4.2 模块接口

A 引用模块

```
var gpio = require('gpio');
```

B 打开模块

```
gpio.open(config);
```

功能描述

根据配置打开gpio端口。

接口约束

无。

参数列表

- config: gpio端口的配置,包括四个属性:
 - pin: 设置gpio端口对应的管脚号,以爱联模组RTL8710开发板为例,可用引脚号为0, 5, 12, 14, 15, 18, 19, 22, 23。0引脚不推荐使用,具体参见gpio模块约束部分。
 - mode: 设置端口输入/输出模式,具体值参考1.2;
 - dir: 设置端口的方向,具体值参考1.1;

返回值

- GPIOPin: 返回值,gpio接口对象;

接口示例

```
GPIOPin = gpio.open({pin: 5, dir: gpio.DIR_IN, mode: gpio.PULLNONE});
```

C 向gpio端口写出

```
GPIOPin.write (val);
```

功能描述

向gpio端口写入值。

接口约束

无。

参数列表

- **val**: 向gpio端口写入的值,取值为1,或者0;如果写入值不是0或1,输入值大于0视为1,小于0的值视为0。

返回值

无。

接口示例

```
GPIOPin.write(1);
```

D 从gpio端口读入

```
number = GPIOPin.read ();
```

功能描述

从gpio端口读入值。

接口约束

无。

参数列表

无。

返回值

- **number**: 数字类型,当端口处于激活态时返回1,否则返回0。

接口示例

```
var number = GPIOPin.read ();
```

E 关闭模块

```
GPIOPin.close ();
```

功能描述

关闭gpio端口。

参数列表

无。

返回值

无。

接口示例

```
GPIOPin.close ();
```

F gpio端口监听

```
GPIOPin.on (event, func, arg);
```

功能描述

监听gpio端口的事件,比如上升沿、下降沿或者二者兼顾,从而调用相应的回调函数;新增回调函数会使旧的回调函数失效。

接口约束

无。

参数列表

- **event**: 监听的事件类型,具体值参考1.3;
- **func**: 回调函数,当监听事件发生时调用该函数;
- **arg**: 传递给回调函数的参数,只允许一个参数,如果需要传递多个参数,需要把多个参数打包在一个结构体里;如果没有参数,该接口将默认传递undefined;

返回值

无。

接口示例

```
GPIOPin.on (gpio. INT_RISING, function() {led.write(1)});
```

4.1.4.3 约束

在使用爱联提供的RTL8710模组的IO0引脚时,存在开机高电平会导致模块跑飞的问题;主要涉及gpio模块和PWM模块,在使用这两个模块时需禁用IO0接口;爱联提供的RTL8710模组的IO14和IO15是JTAG口复用的,正常情况下是无法输出pwm波形和进行gpio操作,在使用时需要关闭JTAG口才能正常使用,目前已在hilink工程中关闭了IO14和IO15的JTAG口。

4.1.4.4 样例

样例A

介绍

初始化gpio端口,并写入1。

例程

```
var gpio = require('gpio');
var config={
  pin:5,
  dir:gpio.DIR_OUT,
  mode:gpio.PULLNONE
};
var pin=gpio.open(config);
pin.write(1);
```

样例B

介绍

监听gpio端口的上升沿事件,当收到该事件时点亮led灯。

连线图

无。

例程

```
var gpio = require('gpio');
var config = {
  pin: 5,
  dir: gpio.DIR_IN,
  mode: gpio.PULLNONE
};
var pin = gpio.open(config);
config = {
  pin: 12,
  dir: gpio.DIR_OUT,
  mode: gpio.PULLNONE
};
var led = gpio.open(config);
pin.on(gpio.INT_RISING, function() {
  led.write(1);
});
```

4.1.5 i2c

4.1.5.1 介绍

i2c模块支持i2c协议，允许多个从设备与一个或多个主设备通信。每个i2c总线有两个信号：SDA和SCL，SDA是数据信号，SCL是时钟信号。

4.1.5.2 模块接口

A 引用模块

```
var i2c = require('i2c');
```

B 打开模块

```
i2c.open(config);
```

功能描述

i2c.open根据config配置打开引脚并设置相关属性。

接口约束

主设备和从设备的配置需要满足下面的要求：

- 主设备和从设备的speed相同。

参数列表

- config: 为i2c端口的配置对象，主设备包括3个配置属性：mode、i2c 和 speed。从设备包括6个配置属性：mode、i2c、speed、i2c_index、address、mask。
 - 主设备属性:
 - mode: 值为 i2c.MASTER。
 - i2c: 内置i2c编号，值为0或1。
 - 0: SDA引脚为IO23 (即23),SCL引脚为IO18 (即18)。

- 1: SDA引脚为IO19 (即19),SCL引脚为IO22 (即22)。
 - speed: 传输速率，标准模式(0-100kb/s)，快速模式(<400kb/s)。
- 从设备属性:
 - mode: 值为 i2c.SLAVE。
 - i2c 和 speed 含义与主设备属性相同。 速率需要与主设备相同。
 - i2c_index: 值为0或者1。0表示i2c0 device， 1 表示i2c1 device。
 - address: 从设备地址，仅支持7-bit address。
 - mask: 地址位掩码，当地址位掩码某位置 1 (= 1) 时，该位即为“无关位”。无论其在地址的相应位中为0还是1，从模块都会作出响应。例如，mask 为0110000，i2c 从器件将应答并认为地址 0010000 和 0100000 有效。

返回值

I2CPin: 如果调用成功，返回I2CPin对象。

接口示例

例如，下面的配置表示：主设备的SDA引脚为IO23,SCL引脚为IO18，传输速率是100k;从设备的SDA引脚为IO19,SCL引脚为IO22。 传输速率与主设备相同，地址是0xAA。

```
config_master={mode:i2c.MASTER,i2c:0,speed:100000};
config_slave={mode:i2c.SLAVE,i2c:1,speed:100000,i2c_index:0,address:0xAA,mask:0xFF};
```

C 发送数据

```
I2CPin.write();
```

功能描述

如果i2c是

- 主设备， I2CPin.write向从设备发送数据
- 从设备， I2CPin.write向主设备发送数据buf中的数据

接口约束

需要注意：单次读写不能超过256（含256）字节. 如需发送超过256字节，请拆分多次发送。

参数列表

主设备：

- dev_addr: number类型，设备物理地址，单个字节
- control/reg_addr: number/buffer/array类型，控制字节或寄存器地址，最多4个字节，可选
- data: number/buffer/array类型，写入的具体数据

I2CPin.write(dev_addr, data);	所有参数为单个字节的number，返回写成功的字节数
I2CPin.write(dev_addr, data[]);	data为buffer或array，返回写成功的字节数

I2CPin.write(dev_addr, control, data);	所有参数为单个字节的number，返回写成功的字节数
I2CPin.write(dev_addr, reg_addr, data);	所有参数为单个字节的number，返回写成功的字节数
I2CPin.write(dev_addr, control[], data);	control为buffer或array，返回写成功的字节数
I2CPin.write(dev_addr, reg_addr[], data);	reg_addr为buffer或array，返回写成功的字节数
I2CPin.write(dev_addr, control, data[]);	data为buffer或array，返回写成功的字节数
I2CPin.write(dev_addr, reg_addr, data[]);	data为buffer或array，返回写成功的字节数
I2CPin.write(dev_addr, control[], data[]);	control/data为buffer或array，返回写成功的字节数
I2CPin.write(dev_addr, reg_addr[],data[]);	reg_add /data为buffer或array，返回写成功的字节数

从设备：

- buf：向主设备发送的具体数据

返回值

- 如果发送成功，返回发送的字节数。

接口示例

- 主设备：

```
var i2c = require('i2c');
var ma_config={mode:i2c.MASTER, i2c:0, speed:100000};
var I2CPin=i2c.open(ma_config);
I2CPin.write(0x40, 0x00, 0x00);//reset
```

- 从设备：

```
var i2c = require('i2c');
var hz=100000;
var sl_config={mode:i2c.SLAVE, i2c:0, speed:hz, i2c_index:0, address:170, mask:255};
var I2CPin=i2c.open(sl_config);
var size=new Buffer(10);
I2CPin.write(size);
```

D 读取数据

```
I2CPin.read();
```

功能描述

如果i2c是

- 主设备,I2CPin.read从从设备读取数据。
- 从设备,I2CPin.read从主设备接收长度为size的数据。

接口约束

需要注意：单次读写不能超过256（含256）字节。如需发送超过256字节，请拆分多次发送；并且如果接收到的数据量小于设置的size，那么对接口的调用将阻塞，直到收到足够的数据。

参数列表

主设备：

- `dev_addr`: number类型，设备物理地址，单个字节。
- `control/reg_addr`: number/buffer/array类型，控制字节或寄存器地址，最多4个字节，**可选**。
- `data`: buffer/array类型，写入的具体数据，**可选**。

<code>I2CPin.read(dev_addr);</code>	所有参数为单个字节的number，返回读取成功到的 数值
<code>I2CPin.read(dev_addr, control);</code>	所有参数为单个字节的number，返回读取成功到的 数值
<code>I2CPin.read(dev_addr, reg_addr);</code>	所有参数为单个字节的number，返回读取成功到的 数值
<code>I2CPin.read(dev_addr, data[]);</code>	<code>data</code> 为buffer或array，返回读取成功到的字节数
<code>I2CPin.read(dev_addr, control, data[]);</code>	<code>data</code> 为buffer或array，返回读取成功到的字节数
<code>I2CPin.read(dev_addr, reg_addr, data[]);</code>	<code>data</code> 为buffer或array，返回读取成功到的字节数
<code>I2CPin.read(dev_addr, control[], data[]);</code>	<code>control/data</code> 为buffer或array，返回读取成功到的字节数
<code>I2CPin.read(dev_addr, reg_addr[], data[]);</code>	<code>reg_addr/data</code> 为buffer或array，返回读取成功到的字节数

从设备：

- `size`: 期望接收数据的长度，取值范围为[1, 256]的整数。

返回值

- 主设备：

如果读取成功，返回接收的字节数。

- 从设备：

如果接收成功，返回接收的buffer对象

接口示例

- 主设备：

```
var i2c = require('i2c');
var tim = require("timer");
```

```
var ma_config={mode:i2c.MASTER, i2c:0, speed:100000};
var I2CPin=i2c.open(ma_config);
var buf = new
Buffer([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]);
tim.setInterval(function() {
    I2CPin.read(0x40, 0x06, buf);
}, 50);
```

- 从设备:

```
var i2c = require('i2c');
var hz=100000;
var sl_config={mode:i2c.SLAVE, i2c:0, speed:hz, i2c_index:0, address:170, mask:255};
var I2CPin=i2c.open(sl_config);
var size=100;
buf2=I2CPin.read(size);
print(buf2.toString());
```

4.1.5.3 约束

无。

4.1.5.4 样例

介绍

下面例子为i2c的示例代码. 它需要2个realtek设备, 一个主设备, 一个从设备. 根据下面的接线方式连接两个设备后, 主设备向从设备发送数据, 从设备接收并输出。

连接方式:

- 主设备 SDA引脚(IO_23) 连接 从设备SDA引脚 (IO_23), 连线之间接10k上拉电阻。
- 主设备 SCL引脚(IO_18) 连接 从设备SCL引脚 (IO_18), 连线之间接10k上拉电阻。
- 两个设备地线相连(共地)。
- 连线后, 先打开从设备, 再打开主设备。

例程

主设备代码

```
var i2c = require('i2c');
var tim = require("timer");
var ma_config={mode:i2c.MASTER, i2c:0, speed:100000};
var i2c_port=i2c.open(ma_config);
i2c_port.write(0x40, 0x00, 0x00); //resetvar

oldmode = i2c_port.read(0x40, 0x00);
var newmode = (oldmode&0x7F)|0x10; // sleep
i2c_port.write(0x40, 0x00, newmode); //go to sleep
i2c_port.write(0x40, 0xFE, 132); //set the prescaler
i2c_port.write(0x40, 0x00, oldmode);
tim.setDelay(5);
i2c_port.write(0x40, 0x00, oldmode|0xa1);
tim.setDelay(5);
i2c_port.write(0x40, 0xFA, 0x00);
i2c_port.write(0x40, 0xFB, 0x00);
i2c_port.write(0x40, 0xFC, 0x08);
i2c_port.write(0x40, 0xFD, 0x02); //180 degree
var buf = new
Buffer([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]);

tim.setInterval(function() {
    //i2c_port.write(0x40, 0x06, buf);
```

```
i2c_port.read(0x40, 0x06, buf);  
}, 50);  
  
print("js execute done!");
```

从设备代码

```
var i2c = require('i2c');  
var hz=100000;  
var sl_config={mode:i2c.SLAVE, i2c:0, speed:hz, i2c_index:0, address:170, mask:255};  
var sl_port=i2c.open(sl_config);  
var size=100;  
buf2=sl_port.read(size);  
print(buf2.toString());
```

4.1.6 objectPersistence

4.1.6.1 介绍

objectPersistence是AntJS中集成的参数持久化功能模块，可实现object内容的掉电存储。

4.1.6.2 模块接口

A 引用模块

```
var objPer = require('objectPersistence');
```

B 写操作

```
var ret = objPer.write(object);
```

功能描述

- 写入object内容。

参数列表

- object表示待写入的对象。
 - object必须包含name的键值内容（并且必须唯一），用以标识此对象。
- ret表示返回的结果，写入成功时返回写入的字节数，写入失败时返回0。

接口约束无。

接口示例

```
var led_write = {  
  name: 'led',  
  r:    110,  
  g:    120,  
  b:    130  
};  
var ret = objPer.write(led_write);
```

C 读操作

```
var object = objPer.read(nameString);
```

功能描述

- 读取名称为nameString的对象内容。

参数列表

- nameString表示待读取的对象名称字符串。
- object表示返回的结果，读取成功时返回读取对象，读取失败时返回undefined。

接口约束无。

接口示例

```
var led_read = objPer.read('led');
```

D 删除操作

```
var ret = objPer.remove(nameString);
```

功能描述

- 删除名称为nameString的对象。

参数列表

- nameString表示待删除的对象名称字符串。
- ret表示返回结果，删除成功时返回0，删除失败时返回-1。

接口约束无。

接口示例

```
var ret = objPer.remove('led');
```

4.1.6.3 约束

操作的object大小必须不大于64 bytes。

4.1.6.4 样例

例程

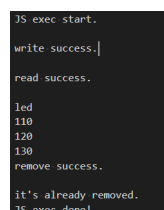
```
var led_write = {
    name: 'led',
    r:    110,
    g:    120,
    b:    130
};
print ("JS exec start.\n");

/*-----加载模块-----*/
var objPer = require('objectPersistence');
/*-----写入-----*/
var ret1 = objPer.write(led_write);
if(ret1 > 0){
    print("write success.\r\n");
}
else{
    print("write fail.\r\n");
};
/*-----读取-----*/
var led_read = objPer.read('led');
if(led_read != undefined){
    print("read success.\r\n");
}
else{
    print("read fail.\r\n");
}
/*-----打印读取信息-----*/
print(led_read.name);
print(led_read.r);
```

```
print(led_read.g);
print(led_read.b);
/*-----删除-----*/
var ret2 = objPer.remove('led');
if(ret2 != -1){
    print("remove success.\r\n");
}
else{
    print("remove fail.\r\n");
}
/*-----再次读取-----*/
var led_read_remove = objPer.read('led');
if (led_read_remove == undefined){
    print ("it's already removed.");
}
print("JS exec done!\n");
```

样例结果

图 4-1



```
JS exec start.
write success.
read success.

led
110
120
130
remove success.
it's already removed.
JS exec done!
```

4.1.7 pwm

4.1.7.1 介绍

pwm模块用于控制支持脉冲宽度调制的引脚产生重复信号,即方波脉冲,其中信号在特定时间上下移动。例如,您可以通过pwm模块将pin IO3引脚每3ms脉冲1ms,它会无限期地执行此操作,直到再次进行设置。通过设置占空比来控制pwm器件,即信号“开启”的时间百分比;例如,控制一个带有pwm的LED,并给它一个50%的占空比(例如1ms开,1ms关),它将以半亮度发光;如果给它一个10%的占空比(例如1ms开启,9ms关闭),它将以10%的亮度发光。

4.1.7.2 模块接口

A 引用模块

```
var pwm = require('pwm');
```

B 打开模块

```
pwm.open(config);
```

功能描述

根据config配置打开pin引脚并设置脉冲周期和占空比。

接口约束

无。

参数列表

- **config** 为pwm端口的配置对象,它包括三个属性: **pin**、**period** 和 **duty**。
 - **pin**: 引脚号,类型为整数`number` (`number`≥0),以爱联模组RTL8710开发板为例,可用引脚号为0, 5, 12, 14, 15, 18, 19, 22, 23。0引脚不推荐使用,具体参见gpio模块约束部分。
 - **period**: 脉冲周期,类型为整数`number` (`number`≥0),单位是μs。
 - **duty**: 占空比,类型是浮点`number`,取值范围是[0,1]。例如:
- **config1**=`{pin:5,period:20000,duty:0}` 表示打开IO5引脚,设置它的脉冲周期是20000μs (20ms),占空比是0(20000μs开,0μs关);如果是控制LED,它将不发光。
- **config2**=`{pin:12,period:10000,duty:0.5}` 表示打开IO12引脚,设置它的脉冲周期是10000μs (10ms),占空比是0.5(5000μs开,5000μs关);如果是控制LED,它将半亮度发光。

返回值

返回PWMPin对象。

接口示例

```
var PWMPin = pwm.open(config1={pin:5, period:20000, duty:0});
```

C 设置脉冲周期

```
PWMPin.set_period(number);
```

功能描述

设置PWMPin对象的脉冲周期。

接口约束

无。

参数列表

- **number** 为设置的脉冲周期,类型为整数`number` (`number`≥0),单位是μs。
 - `number`>0,脉冲周期值将正常设置,不会对占空比设置产生影响。
 - `number`<0,函数会上报错误并返回。
 - `number`=0,脉冲周期被设置为0,此时无论设置占空比的值是多少,返回的占空比的值都将为0,之后若重新更改脉冲周期使`number`>0,需要对占空比进行重新赋值,否则将默认输出占空比为0。

返回值

无。

接口示例

```
PWMPin.set_period(0);
```

D 设置占空比

```
PWMPin.set_duty(number);
```

功能描述

设置PWMPin对象的占空比。

接口约束

无。

参数列表

- number 为设置的占空比,类型是浮点number,取值范围是[0,1]
 - number>1,则设置占空比为1。
 - number<0,则设置占空比为0。
 - 占空比的值为0表示信号不“开启”,1表示信号在脉冲周期内完全“开启”;如果是控制LED,占空比的值为0表示不发光,1表示以100%的亮度发光。

返回值

无。

接口示例

```
PWMPin.set_duty(1);
```

E 获取占空比

```
number = PWMPin.get_duty();
```

功能描述

获取PWMPin对象的占空比。

接口约束

无。

参数列表

无。

返回值

- number : 返回值是[0,1]的浮点数。

接口示例

```
var number = PWMPin.get_duty();
```

4.1.7.3 约束

在使用爱联提供的RTL8710模组的IO0引脚时,存在开机高电平会导致模块跑飞的问题;主要涉及gpio模块和pwm模块,在使用这两个模块时需禁用IO0接口。爱联提供的RTL8710模组的IO14和IO15是JTAG口复用的,正常情况下是无法输出pwm波形和进行gpio操作,在使用时需要关闭JTAG口才能正常使用,目前已在hilink工程中关闭了IO14和IO15的JTAG口。

4.1.7.4 样例

介绍

下面例子是pwm的示例代码;它需要1-3个LED灯或者1个三色(RGB)的LED灯,根据下面的接线方式连接LED和pwm引脚后,LED将逐渐变亮,然后逐渐变暗(感觉好像是人在呼吸,所以也叫呼吸灯)。

- 1-3个LED灯
 - 连接一个LED到IO_23并接地
 - 连接一个LED到IO_5并接地

- 连接一个LED到IO_12并接地
- 3色LED灯
 - 连接R引脚到IO_23
 - 连接G引脚到IO_5
 - 连接B引脚到IO_12
 - 接地

例程

```
var pwm = require('pwm');
var timer = require('timer');
var pwm_period=20000;
var config1={pin:23, period:pwm_period,duty:0};
var config2={pin:5, period:pwm_period,duty:0};
var config3={pin:12, period:pwm_period,duty:0};
var ports=new Array(3)ports[0]=pwm.open(config1);
ports[1]=pwm.open(config2);
ports[2]=pwm.open(config3);

var PWM_STEP=0.01;
var steps=new Array(PWM_STEP, PWM_STEP, PWM_STEP);
var pwms=new Array(0.0, 0.0, 0.0);

while(1) {
  for(var i=0;i<3;i++) {
    ports[i].set_duty(pwms[i]);
    pwms[i]=pwms[i]+steps[i];
    if(pwms[i]>=1.0) {
      steps[i]=-PWM_STEP;
      pwms[i]=1.0;
    }
    if(pwms[i]<0.0) {
      steps[i]=PWM_STEP;
      pwms[i]=0.0;
    }
  }
  timer.setDelay(20);
}
```

4.1.8 rtc

4.1.8.1 介绍

rtc（Real-Time Clock）模块可用于获取系统当前时。

4.1.8.2 模块接口

A 引用模块

```
var rtc = require('rtc');
```

B 获取当前日期秒数

```
rtc.getTime();
```

功能描述

用于获取当前日期的秒数。

接口约束

这个数是从1970.1.1 00:00:00开始计算的。

参数列表

无。

返回值

当前日期秒数。

接口示例

```
var rtc = require('rtc');  
rtc.getTime();
```

C 获取当前年份

```
rtc.getYear();
```

功能描述

用于获取当前年份。

接口约束

无。

参数列表

无。

返回值

当前年份。

接口示例

```
var rtc = require('rtc');  
rtc.getYear();
```

D 获取当前月份

```
rtc.getMonth();
```

功能描述

用于获取当前月份。

接口约束

无。

参数列表

无。

返回值

当前月份。

接口示例

```
var rtc = require('rtc');  
rtc.getMonth();
```

E 获取当前月份内日期

```
rtc.getMday();
```

功能描述

获取当前月份中的第几天。

接口约束

无。

参数列表

无。

返回值

返回当前月份中的第几天。

接口示例

```
var rtc = require('rtc');  
rtc.getMday();
```

F 获取当前周内日期

```
rtc.getWday();
```

功能描述

获取当前周内的日期，（其中星期日为第一天，以0表示，依次类推）。

接口约束

无。

参数列表

无。

返回值

返回当前周内的日期。

接口示例

```
var rtc = require('rtc');  
rtc.getWday();
```

G unix时间戳转换

```
rtc.strToTime(time_string);
```

功能描述

用于将输入的 utc 时间字符串转变为 unix 时间戳。

接口约束

无。

参数列表

- **time_string**: string类型，这里输入的时间字符串格式固定为"1970-01-01 00:00:00"，输入非法字符串或非法日期都会上报错误。时间字符串取值范围为 1970-01-01 00:00:00 到 2036-02-07 06:28:15。

返回值

utc 时间。

接口示例

```
var rtc = require('rtc');  
var m = rtc.strToTime("2018-08-17 12:00:00");
```

H utc 时间字符串转换

```
rtc.timeToStr(time_t t);
```

功能描述

用于将输入的 unix 时间戳转变为 utc 时间字符串。

接口约束

无。

参数列表

- t: number类型，时间戳取值范围为0-2085978495。

返回值

unix 时间戳字符串。

接口示例

```
var rtc = require('rtc');  
rtc.timeToStr(1533744000);
```

I utc 时间字符串转换

```
rtc.setTimeShift(int tz);
```

功能描述

接受一个参数用于设置时间偏置。

接口约束

无。

参数列表

- tz: number类型，该参数取值范围为-12到12之间，当设备rtc寄存器保存的并非是 utc时间而是localtime时。这种情况下设置rtc.setTimeShift(-8)（以北京时间为例）来进行校正。

返回值

无。

接口示例

```
var rtc = require('rtc');  
rtc.setTimeShift(-8);
```

J utc 时间字符串转换

```
rtc.setTime();
```

功能描述

用于以unix时间戳来设置日期。

接口约束

无。

参数列表

- **t**: number类型, 参数设置范围为0-2085978495, 对应utc时间为1970-01-01 00:00:00 到 2036-02-07 06:28:15。

返回值

无。

接口示例

```
var rtc = require('rtc');  
rtc.setTime(1533744000);
```

4.1.8.3 约束

无。

4.1.8.4 样例

介绍

下面是rtc相关接口的样例, 主要包括秒、年月日等当前时间的获取及表达形式的转换, 时间的偏置及以Unix时间戳来对时间进行设置。

例程

下面例子是rtc的示例代码:

```
var rtc = require('rtc');  
var unixtime = rtc.getTime();  
var year = rtc.getYear();  
var month = rtc.getMonth();  
var day = rtc.getMday();  
var week = rtc.getWday();  
var m = rtc.strToTime("2018-08-17 12:00:00");  
var date = rtc.timeToStr(1533744000);  
rtc.setTimeShift(-8);  
rtc.setTime(1533744000);
```

4.1.9 spi

4.1.9.1 介绍

spi(Serial peripheral interface)即串行外围设备接口,是由Motorola首先在其MC68HCxx系列单片机上定义的,基于高速全双工总线的通讯协议; spi通讯需要使用4条线: 3条总线和1条片选; spi遵循主从模式,3条总线分别是SCLK、MOSI和MISO,片选线为SS(低电平有效)。

- **SS (Slave Select)**: 片选信号线,用于选中spi从设备;当从设备上的SS引脚被置拉低时表明该从设备被主机选中;
- **SCLK (Serial Clock)**: 时钟信号线,通讯数据同步用;时钟信号由通讯主机产生,它决定了spi的通讯速率。
- **MOSI (Master Output Slave Input)**: 主机(数据)输出/从设备(数据)输入引脚,即这条信号线上传输从主机到从机的数据。

- MISO (Master Input Slave Output): 主机(数据)输入/从设备(数据)输出引脚,即这条信号线上传输从从机到主机的数据。
- 主从机通过两条信号线来传输数据。

spi 模块支持SPI协议。

4.1.9.2 模块接口

A 引用模块

```
var spi = require('spi');
```

B 打开模块

```
spi.open(config);
```

功能描述

根据config配置打开引脚并设置相关属性;如果调用成功,返回SPIPin对象。

接口约束

主机和从机需要工作在相同的模式下才能正常通讯。

参数列表

- config 为spi端口的配置对象,主设备包括5个配置属性:mode、spi、bits、speed、transmit_mode;从设备包括4个配置属性: mode、spi、bits、transmit_mode。
 - 主设备属性:
 - mode: 值为 SPI.MASTER。
 - spi: 内置spi编号,值为0
 - 0: MOSI引脚为23,MISO引脚为22,SCLK 引脚为18,SS 引脚为19。
 - speed: 传输速率,最高到31.25MHz,类型为浮点数number。
 - bits: 数据帧大小,值为4-16的整数;有一点需要注意的是,主机和从机数据帧需要一样。
 - transmit_mode: spi有4种工作模式,值分别为 0、1、2和3;工作模式用于设置时钟极性(CPOL)和时钟相位(CPHA)。时钟极性指通讯设备处于空闲状态(spi开始通讯前、SS线无效)时,SCLK的状态。时钟相位指数据的采样时刻位于SCLK的偶数边沿采样还是奇数边沿采样。
 - 0: CPOL=0(空闲时刻SCLK低电平),CPHA=0(第一个边沿采样(奇))
 - 1: CPOL=0(空闲时刻SCLK低电平),CPHA=1(第二个边沿采样(偶))
 - 2: CPOL=1(空闲时刻SCLK高电平),CPHA=0(第二个边沿采样(奇))
 - 3: CPOL=1(空闲时刻SCLK高电平),CPHA=1(第二个边沿采样(偶))
 - 从设备属性: 含义与主设备相同。

返回值

返回SPIPin对象。

接口示例

```
SPIPin = spi.open({mode:spi.SLAVE, spi:0, bits:8, transmit_mode: 0});
```

C 向主设备或者从设备写入数据

```
spi.write(buf);
```


功能描述

主设备,SPIPin.write向SS为低电平从设备发送buf中的数据,如果发送成功,返回发送的字节数;从设备,SPIPin.write向主设备发送数据buf中的数据,如果发送成功,返回发送的字节数。

接口约束

无。

参数列表

- buf: 发送的数据。

接口示例

```
var number = spi.write(buf);
```

D 向主设备或者从设备读取数据

```
spi.read(size);
```

功能描述

主设备,SPIPin.read向SS为低电平的从设备接收长度是size的数据,如果接收成功,返回一个保存数据的Buffer对象,长度是size;从设备,I2CPin.read从主设备接收长度为size的数据,返回一个保存数据的Buffer对象,长度是size。

接口约束

无。

参数列表

- size: 期望接收数据的长度,类型为整数number (number>=0)。

接口示例

```
var Buffer = spi.read(1024);
```

4.1.9.3 约束

无。

4.1.9.4 样例

介绍

下面的例子展示了Slave 发送数据给Master;首先启动从设备,然后启动主设备;

例程

Master

```
var spi = require('spi');  
var hz=2000000;  
var ma_config={mode:spi.MASTER, spi:0, speed:hz, bits:8, transmit_mode: 3};  
var ma_port=spi.open(ma_config);  
var size=2048;  
buf=ma_port.read(size);  
print(buf.toString());
```

Slave

```
var spi = require('spi');
var hz=2000000;
var sl_config={mode:spi.SLAVE, spi:0, bits:8, transmit_mode: 3};
var sl_port=spi.open(sl_config);
var size=2048;buf=new Buffer(size);
for(var i=0;i<size;i++) {
    buf.writeUInt8(i+1,i);
}
sl_port.write(buf);
```

4.1.10 timer

4.1.10.1 介绍

timer模块可以提供异步的周期性和一次性定时器功能,以及同步的延时功能。

4.1.10.2 模块接口

A 引用模块

```
var timerHandler = require('timer');
```

B 异步周期性定时器接口

```
timerHandler.setInterval (func, interval, arg);
```

功能描述

提供异步的,周期性的定时器功能。

接口约束

无。

参数列表

- **func**: 定时器超时后的回调函数。
- **interval**: 定时器的周期长度,以毫秒 (ms) 为单位, 类型为整数number (number>=0); 注意: 当interval过小时,会导致引擎来不及处理,导致引擎的事件队列溢出,所以建议 number>=10。
- **arg**: 传递给回调函数的参数;只允许一个参数,如果需要传递多个参数,需要把多个参数打包在一个结构体里;如果没有参数,该接口将默认传递undefined。

返回值

timerID: 返回值,timer的ID,可以用于timer的关闭。

接口示例

```
var timerID = timerHandler.tim.setInterval(function() { /* pseudo code,read data from a port. */
len += port.read(...); if (len > 100) {    tim.stopTimer(id); }}, 100);
```

C 异步一次性定时器接口

```
timerHandler.setDelay (func, delay, arg);
```

功能描述

提供异步的,一次性的定时器功能,即超时后func只被调用一次,timer随后被关闭。

接口约束

无。

参数列表

- `func` : 定时器超时后的回调函数。
- `delay` : 定时器的周期长度,以毫秒 (ms) 为单位, 类型为整数`number` (`number`>=0)。
- `arg` : 传递给回调函数的参数;只允许一个参数,如果需要传递多个参数,需要把多个参数打包在一个结构体里;如果没有参数,该接口将默认传递`undefined`。

返回值

`timerID` : 返回值,timer的ID,可以用于timer的关闭。

接口示例

```
timerHandler.setDelay (func() {}), 100)。
```

D 同步延时接口

```
void setDelay (delay);
```

功能描述

提供同步的延时功能。

接口约束

无。

参数列表

- `delay` : 延时的时间长度,以毫秒 (ms) 为单位,类型为整数`number` (`number`>=0) ;

返回值

无。

接口示例

```
timerHandler.setDelay (100);
```

E 关闭模块

```
void timerHandler.stopTimer(id);
```

功能描述

停止异步的定时器,包括周期性和一次性的。

参数列表

- `tiemrID` : 定时器的id。

返回值

无。

接口示例

```
timerHandler.stopTimer(id);
```

4.1.10.3 约束

同时生效的异步定时器最大数为10。

4.1.10.4 样例

样例A

介绍

定时器;从某端口读取数据（大于100字节）,每隔100ms查询一次,读完退出。

例程

```
var tim = require('timer');
var len = 0;
for (;;) {
    /* pseudo code, read data from a port. */
    len += port.read(...);

    if (len < 100) {
        tim.setDelay(100);
    } else {
        break;
    }
}
```

样例B

介绍

定时器;使用异步timer实现相同功能。

连线图

无。

例程

```
var tim = require('timer');
var len = 0;
var id = tim.setInterval(function()
{
    /* pseudo code, read data from a port. */
    len += port.read(...);

    if (len > 100) {
        tim.stopTimer(id);
    }
}, 100);
```

4.1.11 uart

4.1.11.1 介绍

通用异步收发传输器（Universal Asynchronous Receiver/Transmitter，通常称为UART）是一种异步收发传输器，将数据通过串行通信和并行通信间作传输转换。UART模块可以使设备通过串口发送/接收数据。

4.1.11.2 模块接口

A 引用模块

```
var uart = require('uart');
```

B 打开串口

```
uart.open(config);
```

功能描述

uart.open根据config初始化串口。如果初始化成功，返回uart对象。

接口约束

config是一个json对象，不允许其他类型数据当参数。

参数列表

- config 为uart端口的配置对象，它包括四个属性：uartNumber、baudRate 和 dataBits、stopBits。
 - uartNumber: 内置uart编号，值为0或1。该属性是可选属性，默认为1 (0为调试端口，调试时用于部署脚本，此时，由于io29与io30同usb接口有复用关系，因此不能进行相关操作)。
 - 0: 接收引脚为IO29 (即29)，发送引脚为IO30 (即30)。
 - 1: 接收引脚为IO18 (即18)，发送引脚为IO23 (即23)。
 - baudRate: 波特率，取值范围是[110, 6000000]。可选，默认为115200。
 - dataBits: 数据位，这是衡量通信中实际数据位的参数，取值为 7或8。该属性是可选属性，默认值是8。如何设置取决于你想传送的信息。比如，标准的ASCII码是0~127(7位)。扩展的ASCII码是0~255(8位)。实际数据位取决于通信协议的选取。如果输入数值不是7和8，按照数据位为默认值处理。
 - stopBits: 停止位，用于表示单个包的最后一位。取值为1或2位。该属性是可选属性，默认为1。由于数据是在传输线上定时的，并且每一个设备有其自己的时钟，很可能在通信中两台设备间出现了小小的不同步。因此停止位不仅仅是表示传输的结束，并且提供计算机校正时钟同步的机会。适用于停止位的位数越多，不同时钟同步的容忍程度越大，但是数据传输率同时也越慢。如果输入数值不是1或2，按照停止位为默认值处理。

返回值

- uartObj：串口对象，可以通过该对象直接进行串口功能操作。

接口示例

```
//表示 uart接收引脚是IO29，发送引脚是IO30，波特率是9600，数据位是8位，停止位是1位。  
var config={uartNumber:0,baudRate:9600,dataBits:8,stopBits:1};  
var port = uart.open(config);// 根据config创建一个串口对象
```

C 串口发送数据

```
uartObj.write(data);
```

功能描述

向UART发送data中的数据。

接口约束

data是一个buffer对象，存放要发送的数据，其他数据类型无法发送。

参数列表

- data 发送的数据，类型为buffer (可参考buffer模块)。

返回值

无。

接口示例

```
var buf = new Buffer(10); // 申请一个大小为10的buffer
port.write(buf); // uart对象将buf中的内容发送出去
```

D 串口读取数据

```
uartObj.read(buf, size, timeout);
```

功能描述

从接受引脚接收size字符的数据，存放到buf中，如果 size 大于buf的长度，那么只读取buf长度的数据。如果提供了超时返回时间，那么达到超时时间后，返回已经读取到的字节数。它是一个同步读取接口。

接口约束

入参数数据类型需要与接口原型一致。

参数列表

- buf 为存放读取的数据的buffer，类型为buffer。
- size 为要读取的字符的个数，类型为整数number（number>=0）。
- timeout 可选参数，超时返回时间，类型为整数number（number>=0）。如果timeout为零或者不提供该参数，那么只有接收size字符的数据才返回。

返回值

无。

接口示例

```
var buf = new Buffer(3); // 申请一个大小为10的buffer
port.read(buf, 3); //从uart 接收引脚读取3个字节的内容到buf中
```

Or

```
var buf = new Buffer(3); // 申请一个大小为10的buffer
port.read(buf, 3, 10); //从uart 接收引脚读取3个字节的内容到buf中, 10秒内如果没有读到3个字符，则返回已读到的字符
```

E 串口读取数据

```
uartObj.on(method, [number/end_char], [function]);
```

功能描述

为UART event设置回调函数。当前仅支持UART.DATA事件。多次调用on接口会覆盖掉之前的回调函数。

接口约束

无。

参数列表

- method：目前只支持 UART.DATA event。
- number/end_char 可以是一个大于0的数字或者一个字符。
 - 如果是大于零的数字n，那么当每次接收到n个字节时，回调函数被调用。
 - 如果是字符c，那么当每次接收到字符c时 或者当接收到最大长度为255个字符时，回调函数被调用。
- function 回调函数，event UART.DATA 的回调函数形如: function(data){...}，data是一个buffer，表示已经被uart接收到的数据。

当只提供method参数时，可以注销该event的回调函数。例如：
UARTPin.on(UART.DATA) 可以注销UART.DATA event的回调函数。

返回值

无。

接口示例

```
port.on (uart.DATA, '\r', function(data) {  
    print ('receive from uart:', data)  
    if (data.toString () == '717569740d') //hex code of "quit\r"  
    {  
        port.on (uart.DATA) //注销回调事件  
    }  
})
```

4.1.11.3 约束

串口监听数据的频率太高会导致事件队列溢出，当出现队列溢出时可做如下调整：

- 1.需降低串口发送数据的频率。
- 2.缩短数据的处理时间。
- 3.调大queue的数量。

做上面的几个调整时，需要遵循以下原则之一：

- 1.每个事件的处理时间要小于2个监听事件之间的间隔时间。
- 2.需要处理的事件数量要小于当前空闲queue的数量。

4.1.11.4 样例

样例A

介绍

UART读入用户输入的3个字符然后输出。

例程

```
var uart = require('uart');  
var config={uartNumber:1};  
var port=uart.open(config); //打开uart number 1 端口，即 接收引脚为I018（即18），发送引脚为I023（即23）。  
buf = new Buffer(100); // 申请一个大小为100的buffer  
port.read(buf, 3); //从uart 接收引脚读取3个字节的内容到buf中  
port.write(buf); // 将buf的内容发送到uart 发送引脚
```

样例B

介绍

注册了一个回调函数，每读4个字节，输出读到的内容。 当读到quit时，注销回调函数。

例程

```
var uart = require('uart');  
var config={uartNumber:1};  
var port=uart.open(config); //打开uart number 1 端口，即 接收引脚为I018（即18），发送引脚为I023（即23）。  
port.on(uart.DATA, 4, function(data) { // 设置回调函数，每4个字节调用一次，  
    print('receive from uart:', data);
```

```
if(data.toString() == '71756974') //hex code of "quit"
{
    port.on(uart.DATA)           //注销回调函数
}
});
```

样例C

介绍

注册了一个回调函数，每读到'\r'字符(或者255个字符)，输出读到的内容。当读到'quit\r'时，注销回调函数。

例程

```
var uart = require('uart');
var config={uartNumber:1};
var port=uart.open(config);
port.on(uart.DATA, '\r', function(data) {
    print('receive from uart:', data);
    if(data.toString() == '717569740d') //hex code of "quit\r"
    {
        port.on(uart.DATA)
    }
});
```

4.2 行业使能库

行业使能库模块主要分为四大模块接口，分别为传感器模块接口，控制模块接口，显示模块接口，以及拓展模块接口。

4.2.1 传感器模块接口

本章主要介绍各类传感器单元的使用方法，主要包括超声波传感器、二氧化碳传感器、热电阻传感器、热电偶传感器、陀螺仪加速度传感器、液位传感器、震动传感器以及通用传感器。其中通用传感器部分包含烟雾传感器、煤气传感器、酒精传感器、光敏电阻传感器、火焰传感器、声音传感器、霍尔传感器、土壤湿度传感器以及漫反射红外传感器，因上述传感器初始化等操作均一致，故归类为通用传感器。

4.2.1.1 超声波传感器

4.2.1.1.1 介绍

超声波传感器是将超声波信号转换成其他能量信号（通常是电信号）的传感器。超声波是振动频率高于20kHz的机械波。它具有频率高、波长短、绕射现象小，特别是方向性好、能够成为射线而定向传播等特点。超声波对液体、固体的穿透本领很大，尤其是在不透明的固体中。超声波碰到杂质或分界面会产生显著反射形成反射回波，碰到活动物体能产生多普勒效应。超声波传感器广泛应用于工业、国防、生物医学等方面。

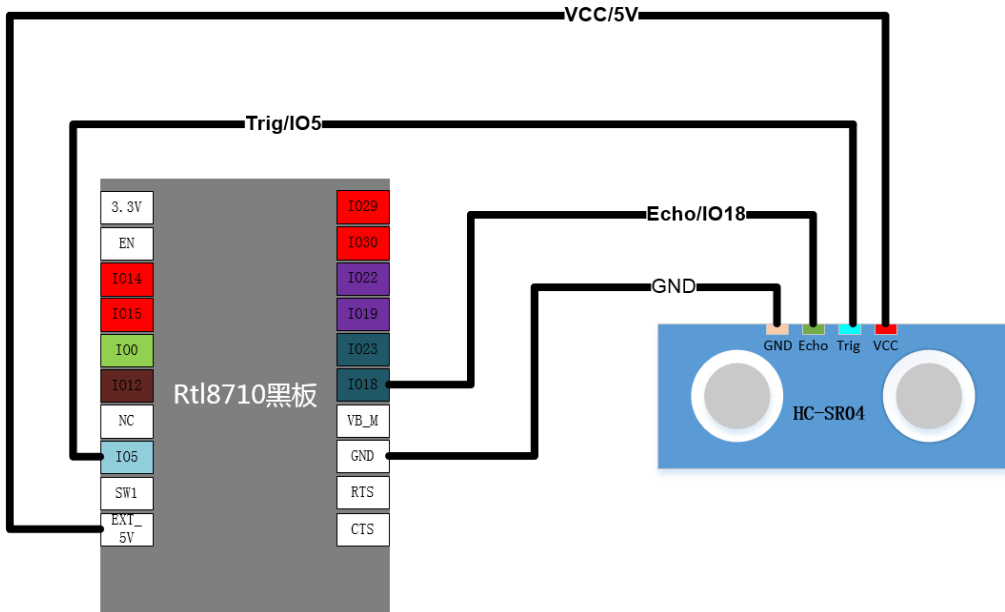
A 模块参数

HC-SR04模块性能稳定，测度距离精确，能和国外的SRF05、SRF02等超声波测距模块相媲美。模块高精度，盲区(2cm)超近，最大识别距离为450cm。



使用电压	5V
静态电流	15mA
工作频率	40Hz
最远射程	4m
最近射程	2cm
测量角度	15度
输入触发信号	10us的TTL脉冲
输出回响	输出TTL电平信号，与射程成正比
规格尺寸	45 * 20 * 15mm

B 连线方式



引脚说明

1	VCC	Ext_5V	正极5V电源
2	GND	GND	负极接地
3	Trig	IO5	接收控制板传来的信号
4	Echo	IO18	将测出的距离结果返回给控制板

4.2.1.1.2 模块接口

A 引用模块

```
var ultrasound = require('ultrasound');
```

B 打开模块

```
ultrasound.open(freq_div, Trig);
```

功能描述

超声波传感器使用IO口与开发板相连，因此需要对通信IO初始化。

接口约束

无。

参数列表

- freq_div: 分频，分频越小测量的精度越高。当freq_div为1时，测量距离范围是0-27cm，当freq_div为10时，测量范围是0-270cm，当freq_div为100时，测量距离范围是0-2700cm。
- trig: number类型，以爱联模组RTL8710开发板为例，可用引脚号为0，5，12，14，15，18，19，22，23。0引脚不推荐使用，具体参见gpio模块约束部分。

返回值

无。

接口示例

```
ultrasound.open(1, 5);
```

C 获取距离

```
ultrasound.getDistance();
```

功能描述

通过串口读取距离障碍物的距离，单位cm。

接口约束

无。

参数列表

无。

返回值

距离障碍物的距离，单位cm。

接口示例

```
var distance = ultrasound.getDistance();
```

D 关闭模块

```
ultrasound.close();
```

功能描述

关闭ultrasound模块。

参数列表

无。

返回值

无。

接口示例

```
ultrasound.close();
```

4.2.1.1.3 约束

- i. 由于定时器timer4指定18号引脚，所以ECHO引脚只能连接18号引脚。
- ii. 被测目标必须垂直于超声波传感器。
- iii. 被测目标表面必须平坦。
- iv. 测量时在超声波传感器周围没有其他可反射超声波的物体。
- v. 由于发射功率有限，传感器无法测量10m外的物体。

4.2.1.1.4 样例

介绍

用超声波传感器测出到物体之间的距离。

例程

```
var ultrasound = require('ultrasound');
var time = require('timer');
ultrasound.open(10, 5);
time.setInterval(function() {
    var distance = getDistance();
    print("distance:", distance, "cm");
}, 1000);
```

4.2.1.2 二氧化碳传感器

4.2.1.2.1 介绍

二氧化碳传感器多用于检测空气中二氧化碳的浓度。在大自然环境里，空气中二氧化碳的正常含量是0.04%（400 PPM），在大城市里有时候达到500 PPM。室内没有人的情况下，二氧化碳浓度一般在500到700 PPM左右。

当二氧化碳的浓度达到1%(1000 PPM)时，人们会感到沉闷，注意力开始不集中，心悸。如果在不透气的卧室里二氧化碳达到1000 PPM，而我们连续睡觉8个小时，早上起床时我们会感觉没有休息好，不想起床。如果办公室的空气中CO2含量达到1000PPM，员工们的工作效率会下降。

二氧化碳浓度达到1500-2000 PPM时，人们会感到气喘、头痛、眩晕。两个人在密闭的卧室里睡一个晚上，二氧化碳的浓度很容易达到2000 PPM. 办公室的空气中CO2浓度达到2000PPM时，员工们会感觉很困，注意力不集中，精神疲劳。超过了2000PPM后，我们甚至不想继续工作，思考能力明显下降。

5000PPM以上时人体机能严重混乱，使人丧失知觉、神志不清。

A 模块参数

二氧化碳传感器S8是瑞典森尔senseair。



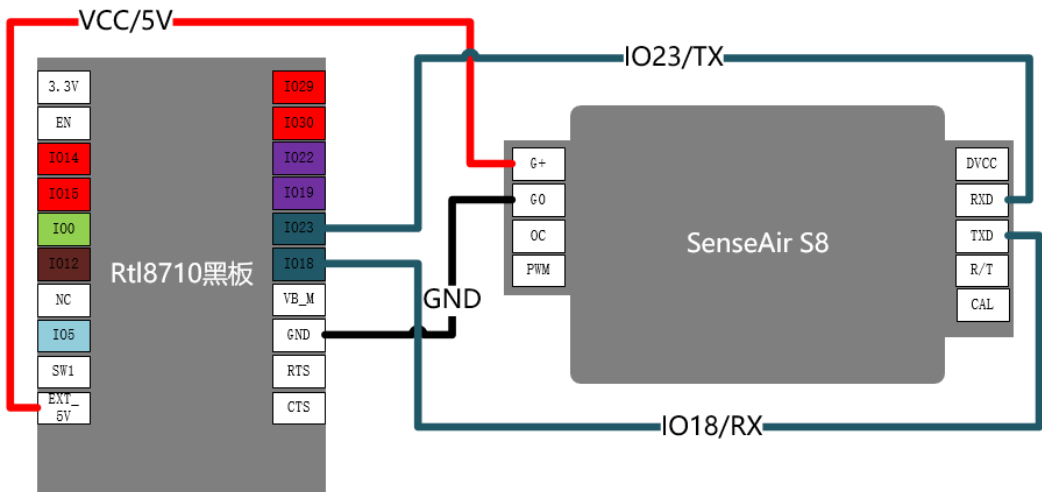
二氧化碳传感器S8参数列表:

测量气体	CO2
工作原理	非色散红外（NDIR）
测量范围	400~2000ppm，高达10000ppm的扩展范围
精度	±70ppm 读数的±3%
数据更新时间	2s
反应时间（t90）	2 分钟
工作温度	0~50℃
工作湿度	0~85%，非冷凝
存储温度	-40~70 ℃
尺寸	33.5 x 20 x 8.5 mm

重量	< 8克
电源	4.5V-5.25V
耗电	300毫安峰值，30mA平均
使用寿命	15年以上

数据通过UART（Modbus协议），方向控制引脚直接连接到RS485 接收器集成电路。报警输出，集电极开路 1000/800 正常状态进行大100毫安。在CO2高，或低功率，或传感器故障晶体管打开。PWM 输出（1kHz）0到100%占空比为0至2000ppm。

B 模块连线



开发板管脚与模块管脚的连线表：

1	Ext_5V	G+	G+是传感器电源，范围是4.5V-5.25V
2	GND	G0	G0是传感器参考地
3	IO23/TX	RXD	RXD，即 UART_RxD，是传感器UART的接收
4	IO18/RX	TXD	TXD，即 UART_TxD，是传感器UART的发送

4.2.1.2.2 模块接口

A 引用模块

```
var s8 = require('senseair_s8');
```

B 打开模块

```
s8.open(config);
```

功能描述

S8使用串口与开发板相连，因此需要对通信串口初始化。

S8串口的配置参数是9600 bps，8bits，N，1 stopbit，即“9600 8 N 1”。

接口约束

无。

参数列表

- config 为uart端口的配置对象，它包括属性：uartNumber。
 - uartNumber: 内置uart编号，爱联rtl8710为例，值为0或1。该属性是可选属性，默认为1 (0为调试端口，调试时用于部署脚本，此时，由于io29与io30同usb接口有复用关系，因此不能进行相关操作)。
 - 0: 接收引脚为IO29 (即29)，发送引脚为IO30 (即30)。
 - 1: 接收引脚为IO18 (即18)，发送引脚为IO23 (即23)。

返回值

无。

接口示例

```
var port = s8.open({portNumber:1});
```

C 读取二氧化碳浓度

```
number port.read();
```

功能描述

S8通过串口读取二氧化碳浓度值。读取浓度单位是ppm。

接口约束

无。

参数列表

无。

返回值

二氧化碳浓度，取值范围是[0,2000]，正整数。

接口示例

```
var concentration = port.read();
```

4.2.1.2.3 约束

模块的浓度数据2s更新一次，启动时间2分钟，因此注意读取接口的调用时机。

4.2.1.2.4 样例

介绍

每隔5000毫秒读取一次二氧化碳浓度。

例程

```
var s8 = require('senseair_s8');
var tim = require("timer");
var port = s8.open({portNumber:1});
tim.setInterval(function() {
    print("浓度: "+port.read()+" ppm");
}, 2000);
print("js execute done!");
```

4.2.1.3 红外接收头传感器

4.2.1.3.1 介绍

红外接收头传感器用于接收红外遥控器发出的信号。

A 模块参数

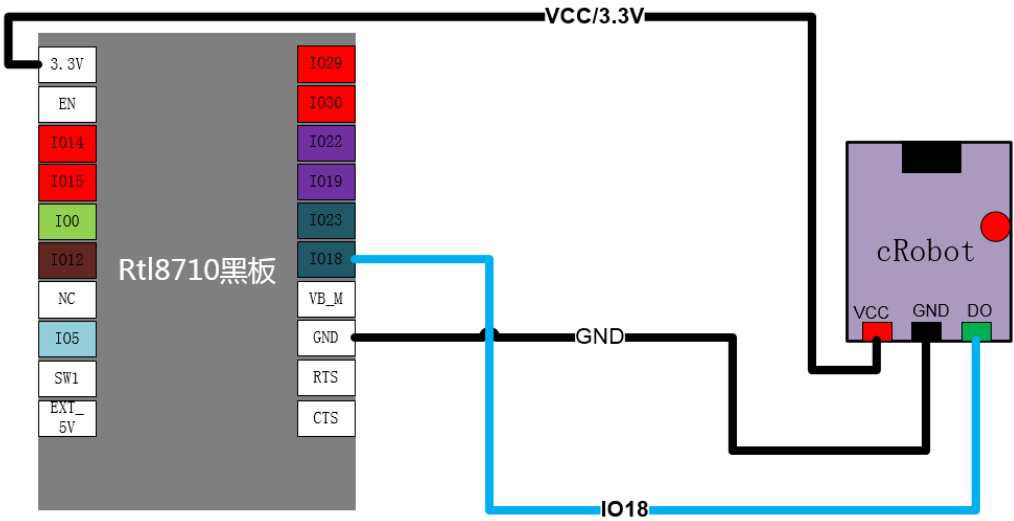


红外接收头传感器参数列表：

工作电压	2.7-4.5V
工作电流	1.7-2.7mA
接收频率	37.9kHz
峰值波长	940nm
静态输出	高电平
输出低电平	<=0.4V
输出高电平	接近工作电压

红外遥控器发出的信号是一连串的二进制脉冲码。为了使其在无线传输过程中免受其他红外信号的干扰,通常都是先将其调制在特定的载波频率上,然后再经红外发射二极管发射出去,而红外线接收装置则要滤除其他杂波,只接收该特定频率的信号并将其还原成二进制脉冲码,也就是解调。内置接收管将红外发射管发射出来的光信号转换为微弱的电信号,此信号经由IC内部放大器进行放大,然后通过自动增益控制、带通滤波、解调变、波形整形后还原为遥控器发射出的原始编码,经由接收头的信号输出脚输入到电器上的编码识别电路。

B 连线方式



引脚说明:

序号	开发板引脚	模块引脚	说明
1	Ext_5V	VCC	3.3-5V电源电压
2	GND	GND	负极接地
3	IO18	DO	数字信号输出

4.2.1.3.2 模块接口

A 引用模块

```
var infrared = require('infrared');
```

B 打开模块

```
infrared.open(freq_div);
```

功能描述

红外接收头传感器与开发板相连，因此需要对通信IO口进行初始化。

接口约束

无。

参数列表

freq_div: 定时器分频，当freq_div为1时，脉冲宽度取值范围是0~1.6ms，当freq_div为10时，脉冲宽度取值范围是0~16ms等等，当freq_div为100时，脉冲宽度取值范围是0~160ms等等，freq_div越小精度越高，由于该款传感器需要捕捉的脉冲宽度在1.5ms~2.2ms之间，所以freq_div的值为10。

1	0~1.6ms
10	0~16ms
100	0~160ms
...	...

返回值

打开对象。

接口示例

```
var ir = infrared.open(10);
```

C 监听遥控器解码

```
ir.on(func, args);
```

功能描述

监听infrared端口返回的解码值。

接口约束

无。

参数列表

func: 回调函数，当监听事件发生时调用该函数；
arg: 传递给回调函数的参数，只允许一个参数，如果需要传递多个参数，需要把多个参数打包在一个结构体里。如果没有参数，该接口将默认传递undefined；

序号	返回值	对应键值
1	104	0
2	48	1
3	24	2
4	122	3
5	16	4
6	56	5
7	90	6
8	66	7
9	74	8
10	82	9
11	162	CH-
12	98	CH
13	226	CH+
14	34	<<
15	2	>>
16	194	>
17	224	-
18	168	+
19	144	EQ
20	152	100+
21	176	200+

返回值

无。

接口示例

```
var rcv = ir.on(function(data){});
```

D 关闭模块

```
ir.close();
```

功能描述

关闭infrared模块。

参数列表

无。

返回值

无。

接口示例

```
ir.close();
```

4.2.1.3.3 约束

i 由于使用定时器timer4，该模块只能应用18号引脚。

ii 测试距离理论上小于8m。

4.2.1.3.4 样例

介绍

红外发射器发射信号，读取红外接收头接收信号并转换为十进制。

例程

```
var infrared = require('infrared');  
var time = require('timer');  
var ir = infrared.open(10);  
ir.on(function(data) {  
    print("rcv:", data);  
});
```

4.2.1.4 热电偶传感器

4.2.1.4.1 介绍

热电偶传感器是工业中使用最为普遍的接触式测温装置。这是因为热电偶具有性能稳定、测温范围大、信号可以远距离传输等特点，并且结构简单、使用方便。热电偶能够将热能直接转换为电信号，并且输出直流电压信号，使得显示、记录和传输都很容易。

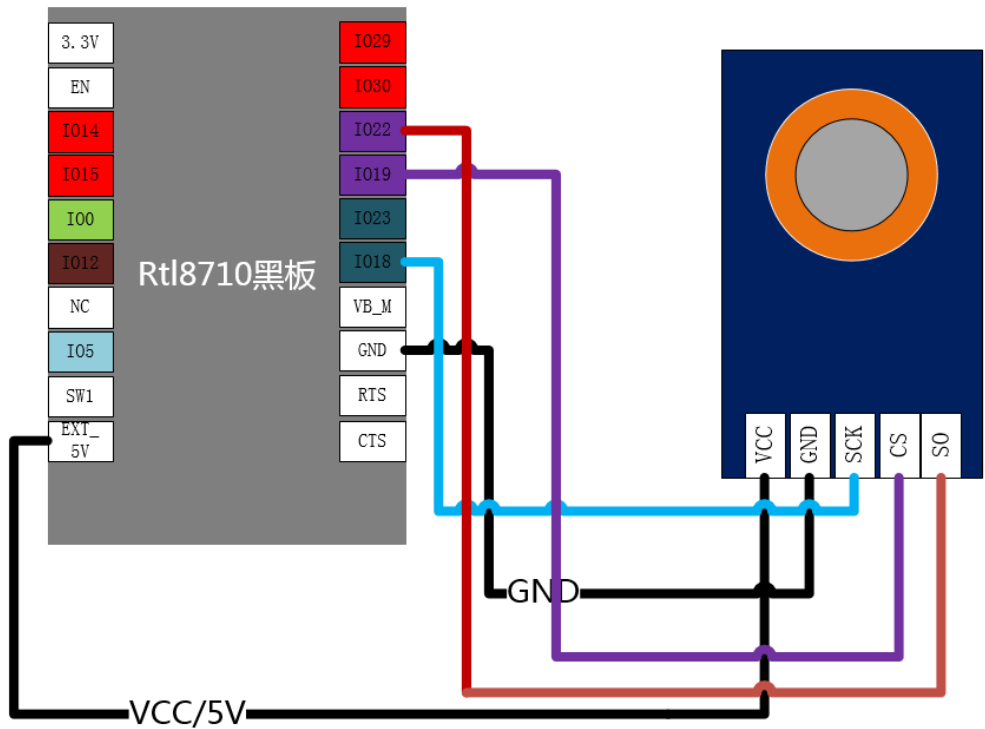
A 模块参数

K型热电偶传感器,可以直接测量各种生产中从0℃到1300℃范围的液体蒸汽和气体介质以及固体的表面温度;通常和显示仪表,记录仪表和电子调节器配套使用,是目前用量最大的廉金属热电偶。



MAX6675冷端温度补偿、热电偶数字转换器可进行冷端温度补偿,并将K型热电偶信号转换成数字信号;数据输出为12位分辨率、SPI兼容、只读格式。转换器温度分辨率为0.25°C,可读取温度达+1024°C,热电偶在0°C至+700°C温度范围内精度为8 LSB。

B 模块连线



开发板管脚与模块管脚的连线表：

序号	开发板管脚	模块管脚	说明
1	Ext_5V	VCC	VCC是传感器电源，范围是3.3V-5V
2	GND	GND	传感器接地端
3	IO18	SCK	时钟信号线，通讯数据同步用。时钟信号由通讯主机产生，它决定了SPI的通讯速率
4	IO19	CS	片选信号线，用于选中SPI从设备。当从设备上的SS引脚被置拉低时表明该从设备被主机选中
5	IO22	SO	主机(数据)输入/从设备(数据)输出引脚，即这条信号线上传输从从机到主机的数据

4.2.1.4.2 模块接口

A 引用模块

```
var Thermocouple = require('Thermocouple');
```

B 打开模块

```
thermocouple.open(config);
```

功能描述

打开热电偶传感器模块。

接口约束

无。

参数列表

- config:
 - spi : 0 (内置SPI编号,值为0，具体参见SPI模块)。
 - speed : 传输速率,最高到31.25MHz,类型为浮点数number。
 - bits : 数据帧大小，值为4-16的整数(这里必须是16,因为max6675的特性)。有一点需要注意的是，主机和从机数据帧需要一样。
 - transmit_mode : 3 (SPI有4种工作模式，值分别为 0、1、2和3。工作模式用于设置时钟极性(CPOL)和时钟相位(CPHA)),具体解释参见SPI模块。

返回值

thermocouple对象。

接口示例

```
var thermocouple = thermocouple.open({spi:0, speed:2000000, bits:16, transmit_mode:3});
```

C 读取温度值

```
thermocouple.read();
```

功能描述

获取温度值。

接口约束

无。

参数列表

无。

返回值

温度值（0 ~ 1024摄氏度）。

接口示例

```
var temp = thermocouple.read();
```

D 关闭模块

```
thermocouple.close();
```

功能描述

关闭热电偶传感器模块。

参数列表

无。

返回值

无。

接口示例

```
thermocouple.close();
```

4.2.1.4.3 约束

由于max6675寄存器中的数据格式限制，模块的数据帧大小只能为16bits。

4.2.1.4.4 样例

介绍

1000毫秒读取一次温度。

例程

```
var thermocouple = require('thermocouple');  
var tim = require('timer');  
var config = {spi:0, speed:2000000, bits:16, transmit_mode:3};  
var thermocouple = thermocouple.open(config);
```

```
tim.setInterval(function() {
    print("temp" + thermocouple.read());
}, 1000);
```

4.2.1.5 陀螺仪加速度传感器

4.2.1.5.1 介绍

陀螺仪又叫角速度传感器，是不同于加速度（G-sensor）的，可获取物理量是偏转、倾斜时的转动角速度。多用于测试物体的运动状态。

A 模块参数



陀螺仪加速传感器MPU-6050参数列表：

使用芯片	MPU-6050
供电电源	3-5V（内部低压差稳压）
通讯方式	标准I2c通讯协议
芯片内置	16bitAD转换器，16位数据输出
陀螺仪范围	±250 500 1000 2000 °/s
加速范围	±2 ±4 ±8 ±16g
采用	沉金PCB，机器焊接工艺保证质量
引脚间距	2.54mm
温度传感器测量范围	-40~+85度
温度传感器线性误差	±1度
陀螺仪线性误差	0.1°/s
加速度输出频率	最高1000Hz
陀螺仪最高分辨率	131LSB/（°/s）
DMP姿态解算频率	最高200Hz
陀螺仪输出频率	最高8000Hz

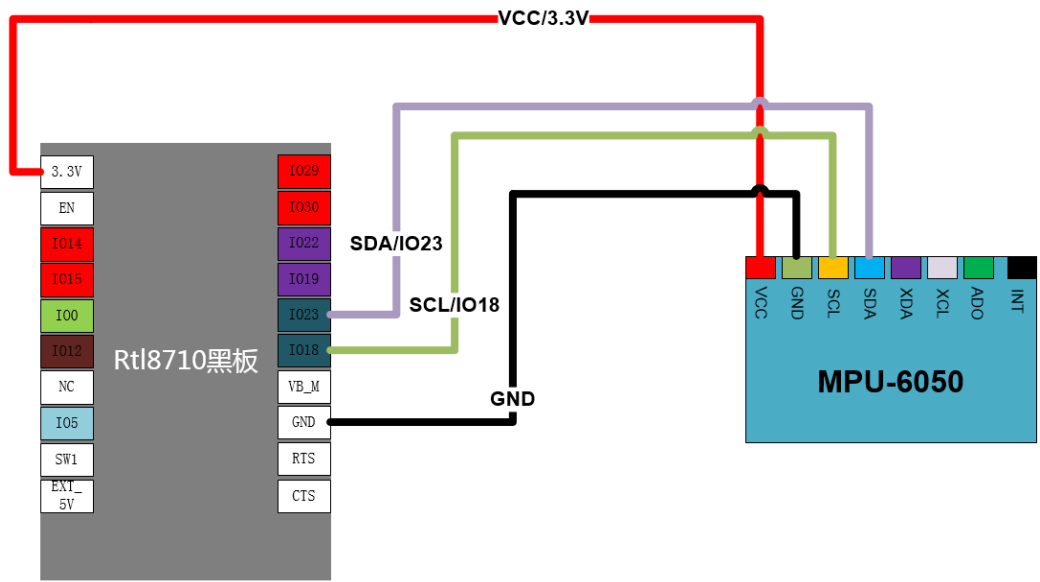
MPU-6050为整合性6轴运动处理组件，相较于多组件方案，免除了组合陀螺仪与加速器时之轴间差的问题，减少了大量的包装空间。MPU-6050整合了3轴陀螺仪，3轴加速器，并含可藉由第二个I2c端口连接其他厂牌的加速器，磁力传感器，或者其他传感器的数位运动处理硬件加速引擎，由主要I2c端口以单一的数据流形式，向应用端输出完整的9轴融合演算技术。

InvenSense的运动处理资料库，可处理运动感测的复杂数据，降低了运动处理运算对操作系统的负荷，并为应运开发提供架构化的API。

MPU-6050的角速度全格感测范围为±250、±500、±1000、±2000°/sec（dps），可准确追踪快速与慢速动作，并且，用户可程式控制的加速器全格感测范围为±2g，±4g，±8g与±16g。产品传输可透过最高至400kHz的I2c或者最高达20MHz的SPI。

MPU-6050内建频率产生器在所有温度范围仅有±1%频率变化。

B 模块连线



引脚说明：

序号	开发板引脚	模块接口	备注
1	3.3V	VCC	3~5V电源电压
2	GND	GND	负极接地
3	IO23/TX	SCL	连接MCU的i2c接口
4	IO18/RX	SDA	连接MCU的i2c接口
5		XDA	连接外部从机设备
6		XCL	连接外部从机设备
7		ADO	IIC接口的地址控制引脚，该引脚控制IIC地址的最低位

序号	开发板引脚	模块接口	备注
8		INT	输入端

4.2.1.5.2 模块接口

A 引用模块

```
var mpu = require('mpu6050');
```

B 打开模块

```
mpu.open(i2c_num, frequency, address);
```

功能描述

mpu6050使用串口与开发板相连，因此需要对通信串口初始化。

接口约束

无。

参数列表

- i2c_num: number类型，当num为0时，SDA引脚为IO23 (即23),SCL引脚为IO18 (即18)。当num为1时，SDA引脚为IO19 (即19),SCL引脚为IO22 (即22)。
- frequency: 传输速率，标准模式(0-100kb/s)，快速模式(<400kb/s)。
- address: 陀螺仪加速传感器地址。

返回值

mpu对象。

接口示例

```
mpu.open(1, 100000, 0x68);
```

C 获取gyroscope加速度，角速度，温度

```
gyr.getMpuData(tmp); //以获取温度为例
```

功能描述

读取温度，X、Y、Z轴的加速度和绕X、Y、Z轴的角速度。

接口约束

无。

参数列表

-tmp: mpu寄存器类型，参数如下：

- TEMP: 获取当前温度（-40~+85度）；
- ACC_X: 获取加速度X轴分量；
- ACC_Y: 获取加速度Y轴分量；
- ACC_Z: 获取加速度Z轴分量；

- GYR_X: 获取绕X轴旋转的角速度;
- GYR_Y: 获取绕Y轴旋转的角速度;
- GYR_Z: 获取绕Z轴旋转的角速度。

返回值

根据所传入的参数返回温度值，或各个轴的加速度和角速度的值。

接口示例

```
var data = gyr.getMpuData(gyr.TEMP);
```

4.2.1.5.3 约束

无。

4.2.1.5.4 样例

介绍

获取温度；获取X轴加速度；获取Y轴加速度；获取Z轴加速度；获取绕X轴的角速度；获取绕Y轴的角速度；获取绕Z轴的角速度。

例程

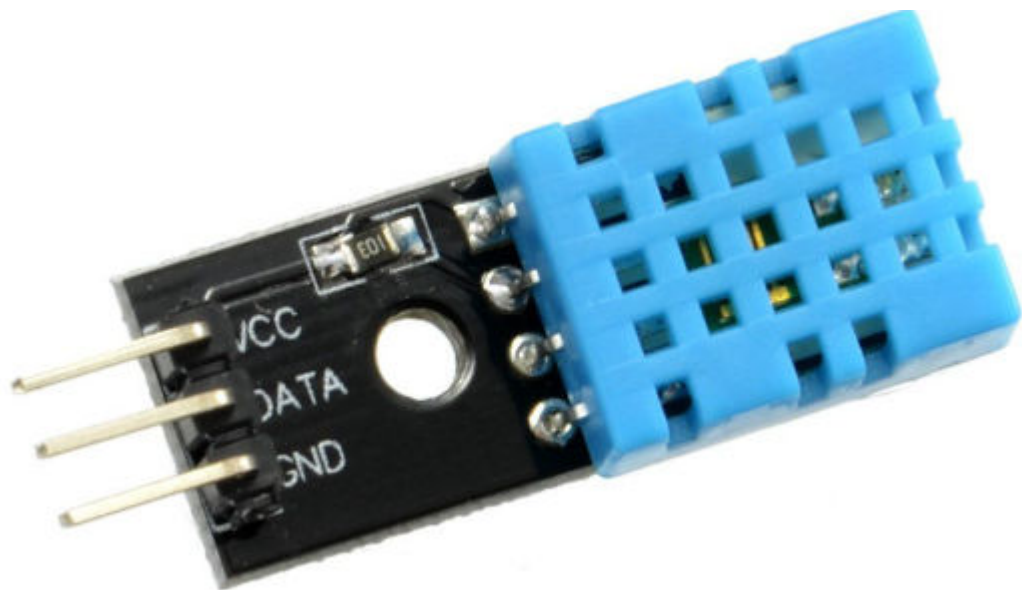
```
var mpu = require('mpu6050');
var gyr = mpu.open(0, 100000, 0x68);
var temp = gyr.getMpuData(gyr.TEMP);           //获取温度
print("temp=: ", temp);
var acc_x = gyr.getMpuData(gyr.ACC_X);          //获取X轴加速度
print("acc_x=: ", acc_x);
var acc_y = gyr.getMpuData(gyr.ACC_Y);          //获取Y轴加速度
print("acc_y=: ", acc_y);
var acc_z = gyr.getMpuData(gyr.ACC_Z);          //获取Z轴加速度
print("acc_z=: ", acc_z);
var gyr_x = gyr.getMpuData(gyr.GYR_X);          //获取绕X轴的角速度
print("gyr_x=: ", gyr_x);
var gyr_y = gyr.getMpuData(gyr.GYR_Y);          //获取绕Y轴的角速度
print("gyr_y=: ", gyr_y);
var gyr_z = gyr.getMpuData(gyr.GYR_Z);          //获取绕Z轴的角速度
print("gyr_z=: ", gyr_z);
```

4.2.1.6 温湿度传感器

4.2.1.6.1 介绍

A 模块参数

DHT11数字温湿度传感器是一款含有已校准数字信号输出的温湿度复合传感器。

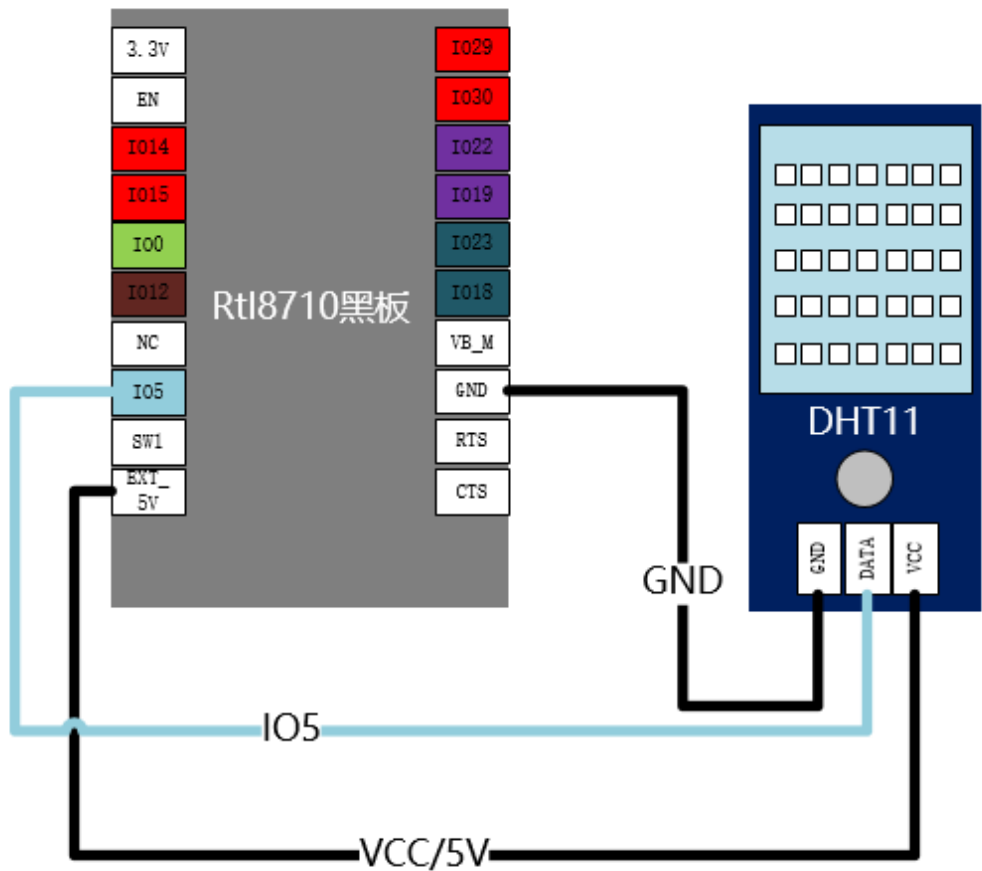


温湿度传感器参数：

参数	数值
接口	单线制串行接口，4针单排引脚封装
湿度精度	±5%RH
温度精度	±2℃
湿度范围	20~90%RH
温度范围	0~50℃

单个数据引脚端口完成输入输出双向传输，其数据包由5Byte（40Bit）组成；一次通讯时间最大3ms,数据分小数部分和整数部分。DHT11是单总线的，采用私有时序。DHT11应用于暖通空调、测试及检测设备、汽车、数据记录器、消费品自动控制、气象站、家电、湿度调节器、医疗、除湿器等场景。

B 模块连线



开发板与模块管脚连接表：

序号	开发板管脚	模块管脚	备注
1	Ext_5v	VDD	VDD是3.5V-5.5V DC
2	IO5	DATA	DATA是串行数据,单总线
3	GND	GND	GND是接地,电源负极

4.2.1.6.2 模块接口

A 引用模块

```
var dht11 = require('dht11');
```

B 初始化模块

```
dht11.init(ionum)
```

功能描述

根据配置初始化DHT11端口。

接口约束

无。

参数列表

- `ionum`是使用开发板的IO号，以爱联模组RTL8710开发板为例，可用引脚号为0，5，12，14，15，18，19，22，23。0引脚不推荐使用，具体参见gpio模块约束部分。

返回值

无。

接口示例

```
dht11.init(5);
```

C 读取湿度

```
dht11.getHumidity()
```

功能描述

从DHT11端口读取湿度值。

接口约束

无。

参数列表

无。

返回值

湿度值，单位是%，范围是[0.0,100.0]的正数，可以有1位小数。

接口示例

```
var humi = dht11.getHumidity();
```

D 读取温度

```
dht11.getTemp()
```

功能描述

从DHT11端口读取温度值。

接口约束

无。

参数列表

无。

返回值

温度值，单位摄氏度，范围是[0.0,50.0]的正数，可以有1位小数。

接口示例

```
var temp = dht11.getTemp();
```

E 读取温湿度

```
dht11.getTempHumi()
```

功能描述

从DHT11端口同时读取温度值和湿度值。DHT11总线是一次能输出温度值和湿度值。

接口约束

无。

参数列表

无。

返回值

js对象，包括属性temp和humidity，temp范围是[0.0,50.0]的正数，可以有1位小数，humidity范围是[0.0,100.0]的正数，可以有1位小数。

接口示例

```
var obj =dht.getTempHumi();
print("温度: "+obj.temp+" C");
print("湿度: "+obj.humidity+" %");
```

4.2.1.6.3 约束

模块的温湿度读取频次不能高于2s一次。

4.2.1.6.4 样例

介绍

温湿度读取并显示。

例程

```
var dht11 = require('dht11');
var tim = require('timer');
dht11.init(5);
tim.setInterval(function() {
    print("humi:"+dht11.getHumidity()+"%");
    tim.setDelay(2000);
    print("temp:"+dht11.getTemp());
    tim.setDelay(2000);
}, 6000);
print("js execute done!");
```

4.2.1.7 颜色传感器

4.2.1.7.1 介绍

A 模块参数

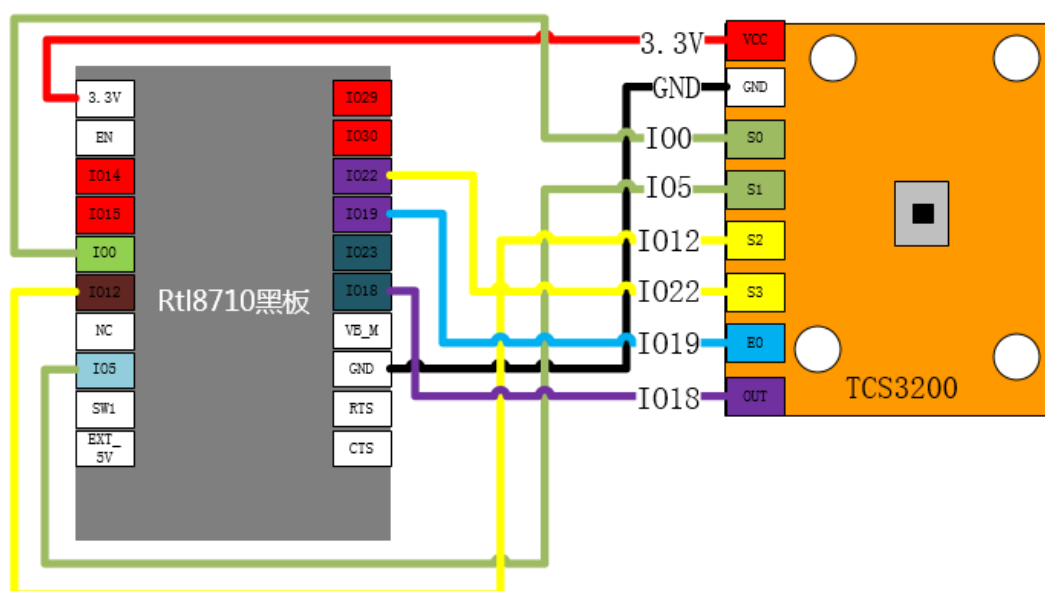
TCS3200颜色传感器是一款全彩的颜色检测器，包括了一块TAOS TCS3200RGB感应芯片和4个白色LED灯，TCS3200能在一定的范围内检测和测量几乎所有的可见光。TCS3200有大量的光检测器，每个都有红绿蓝和清除4种滤光器。每6种颜色滤光器均匀地按数组分布来清除颜色中偏移位置的颜色分量。内置的振荡器能输出方波，其频率与所选择的光的强度成比例关系。



引脚说明

序号	模块引脚	说明
1	VCC	3~5V电源电压
2	GND	负极接地
3	S0,S1	选择输出比例因子或电源关断模式
4	S2,S3	选择滤波器类型
5	EO	频率输出使能引脚，可控控制输出状态
6	OUT	频率输出引脚

B 模块连线



开发板管脚与模块管脚的连线表:

序号	开发板管脚	模块管脚	说明
----	-------	------	----

1	3.3V	VCC	电源VCC
2	GND	GND	接地
3	IO0	S0	与S1配合，设置输出频率比例
4	IO5	S1	与S0配合，设置输出频率比例
5	IO12	S2	与S3配合，选择颜色滤波器
6	IO22	S3	与S2配合，选择颜色滤波器
7	IO19	E0	输出使能引脚，低电平使能
8	IO18	OUT	输出

4.2.1.7.2 模块接口

A 引用模块

```
var tcs3200 = require('tcs3200');
```

B 打开模块

```
tcs3200.open(a, b, c, d, e, f);
```

功能描述

根据配置打开颜色传感器端口。

接口约束

无。

参数列表

- a,b:对应s0,s1引脚，选择不同的输出比例因子;
- c,d:对应s2,s3引脚，选择不同色光的滤波器;
- e:对应out引脚，输出RGB三原色对应的频率;
- f:对应EO引脚，控制颜色传感器开关;
- a, b, c, d, e, f为number类型，以爱联模组RTL8710开发板为例，可用引脚号为0, 5, 12, 14, 15, 18, 19, 22, 23。0引脚不推荐使用，具体参见gpio模块约束部分。

返回值

颜色传感器接口对象。

接口示例

```
var color = tcs3200.open(0, 5, 12, 18, 19, 22);
```

C 设置白平衡


```
colorPin.balance(void)
```

功能描述

模块根据目标颜色设置白平衡，后续测量时将根据白平衡结果基数去计算颜色测量结果。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
color.balance();
```

D 测量目标颜色

```
color.measure(void)
```

功能描述

测量目标颜色，根据白平衡结果基数计算并输出颜色RGB值。

参数列表

无。

返回值

number类型，目标RGB配色值，如白色0xFFFFFF;

接口示例

```
var value = color.measure();
```

4.2.1.7.3 约束

无。

4.2.1.7.4 样例

介绍

打开颜色传感器，设置白平衡，五秒后测量颜色值并打印输出。

例程

```
var color = require('tcs3200');  
var time = require('timer');  
var port = color.open(0, 5, 12, 18, 19, 22);  
port.balance();  
time.setDelay(5000);  
print("rgb:" + port.measure);
```

4.2.1.8 液位传感器

4.2.1.8.1 介绍

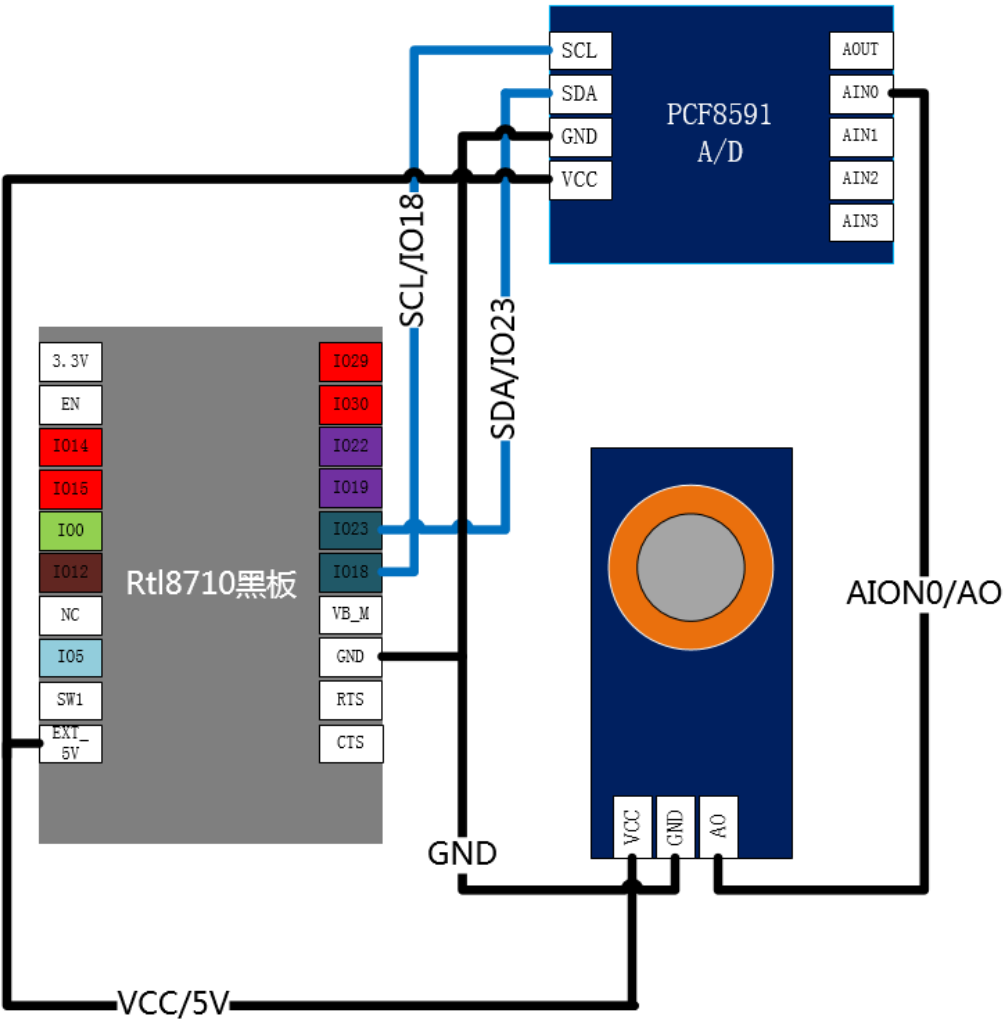
液位传感器（静压液位计/液位变送器/液位传感器/水位传感器）是一种测量液位的压力传感器。

A 模块参数

静压投入式液位变送器（液位计）是基于所测液体静压与该液体的高度成比例的原理，采用国外先进的隔离型扩散硅敏感元件或陶瓷电容压力敏感传感器，将静压转换为电信号，再经过温度补偿和线性修正，转化成标准电信号。传感器如下图：



B 模块连线



AO模拟量,开发板管脚、AD转换器和模块管脚的接线表:

序号	开发板管脚	AD转换器管脚	模块管脚	说明
1	5V	VCC	VCC	外接 3.3V-5V
2	GND	GND	GND	外接GND
3	IO18/SCL	SCL	AO(连接AD转换器AINO)	传感器模拟量输出接口
4	IO23/SDA	SDA	AO(连接AD转换器AINO)	传感器模拟量输出接口

4.2.1.8.2 模块接口

A 引用模块

```
var liquid = require ('liquid');
```

B 打开模块

```
liquid.open (config);
```

功能描述

根据配置打开端口。

接口约束

无。

参数列表

- config :
- mode : liquid.AO,模拟量模式。
 - i2c : 内置i2c编号,值为0或1;详细解释参见I2C模块。
 - speed : 传输速率,标准模式 (0-100kb/s),快速模式(<400kb/s)。
 - address : 从设备地址,仅支持7-bit address。

返回值

遵循I2C协议的liquid端口对象。

接口示例

```
var i2c_port = liquid.open({mode:liquid.A0, i2c:0, speed:100000, address:0x48});
```

C 读取AO模拟量

```
i2c_port.read();
```

功能描述

通过AD转换器，读到传感器输出的模拟值。

接口约束

无。

参数列表

无。

返回值

level 传感器读到的模拟值(值越低, 说明水位深度越深, 范围 (0~255))。

D 关闭模块

```
i2c_port.close();
```

功能描述

关闭传感器功能。

参数列表

无。

4.2.1.8.3 约束

无。

4.2.1.8.4 样例

介绍

获取液位模拟量。

例程

```
var liquid = require ('liquid');  
var tim = require ("timer");  
var config = {mode:liquid.A0, i2c:0, speed:100000, address:0x48};  
var i2c_port = liquid.open(config);  
tim.setInterval(function() {  
    var level = i2c_port.read();  
    print("level", level);  
}, 1000);
```

4.2.1.9 震动传感器

4.2.1.9.1 介绍

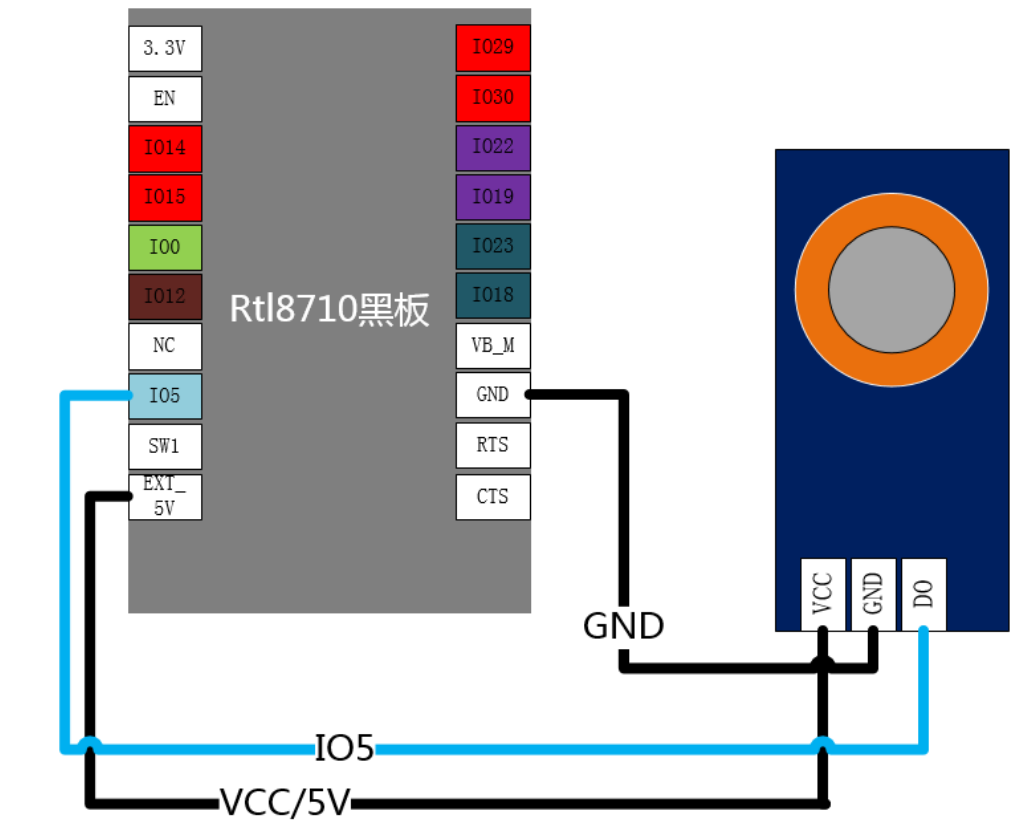
震动传感器, 可用于检测物体是否处于震动状态。

A 模块参数

内部含有滚珠开关。滚珠开关, 其内部含有导电珠子, 器件一旦震动, 珠子随之滚动, 就能使两端的导针导通;传感器如下图:



B 模块连线



DO数字量,开发板管脚和模块管脚的连线表:

序号	开发板管脚	模块管脚	说明
1	5V	VCC	VCC是传感器电源正极,范围是3.3V-5V
2	GND	GND	GND是传感器参考地
3	IO5	DO	DO是传感器数字量输出接口

4.2.1.9.2 模块接口

A 引用模块

```
var sensor = require ('shock');
```

B 打开模块

```
shock.open (config);
```

功能描述

打开震动传感器。

接口约束

无。

参数列表

config:

- pin : 引脚号,以爱联模组RTL8710开发板为例,可用引脚号为0, 5, 12, 14, 15, 18, 19, 22, 23。0引脚不推荐使用,具体参见gpio模块约束部分。
- mode : 模式,shock.DO。

返回值:

- pin_do : 传感器对象;

C 读取AO数字量

```
pin_do.read ();
```

功能描述

读取震动传感器AO数字量。

接口约束

无。

参数列表

无。

返回值

- num : 读到的值
 - 1 (表示传感器正常)。
 - 0 (表示传感器被外界条件触发)。

接口示例

```
var level = pin_do.read();
```

D 关闭模块

```
pin_do.close();
```

功能描述

关闭震动传感器端口。

参数列表

无。

返回值

无。

接口示例

```
pin_do.close();
```

E 监听端口

```
pin_do.on (event, func, arg);
```

功能描述

设置震动产生时触发的回调事件。注：可以通过旋转传感器上的十字螺母，以调节震动传感器的灵敏度。

接口约束

无。

参数列表

- **event**：回调事件的事件类型
 - **shock.SHOCK_DOWN**：震动事件。震动传感器检测到震动。
 - **shock.SHOCK_UP**：震动结束事件。震动传感器检测到震动后恢复正常时触发的事件。
- **func**：回调函数,监听高低电平转换事件触发时调用该函数。
- **arg**：传递给回调函数的参数,只允许一个参数,如果需要传递多个参数,需要把多个参数打包在一个结构体里;如果没有参数,该接口将默认传递undefined。

返回值

无。

4.2.1.9.3 约束

无。

4.2.1.9.4 样例

介绍

传感器检测到震动时，触发“打印DO口读到的值”的事件。

例程

```
var shock = require('shock');
var tim = require('timer');
var config = {mode:shock.DO,pin:5};
var pin_do = shock.open(config);
pin_do.on(shock.SHOCK_UP,function(){
    var level = pin_do.read();
    print("sensor status is changing! level is ",level);
});
```

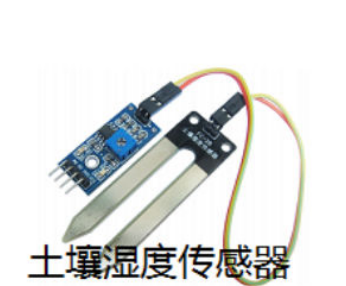
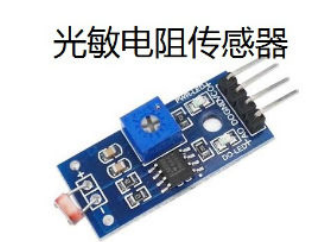
4.2.1.10 通用传感器

4.2.1.10.1 介绍

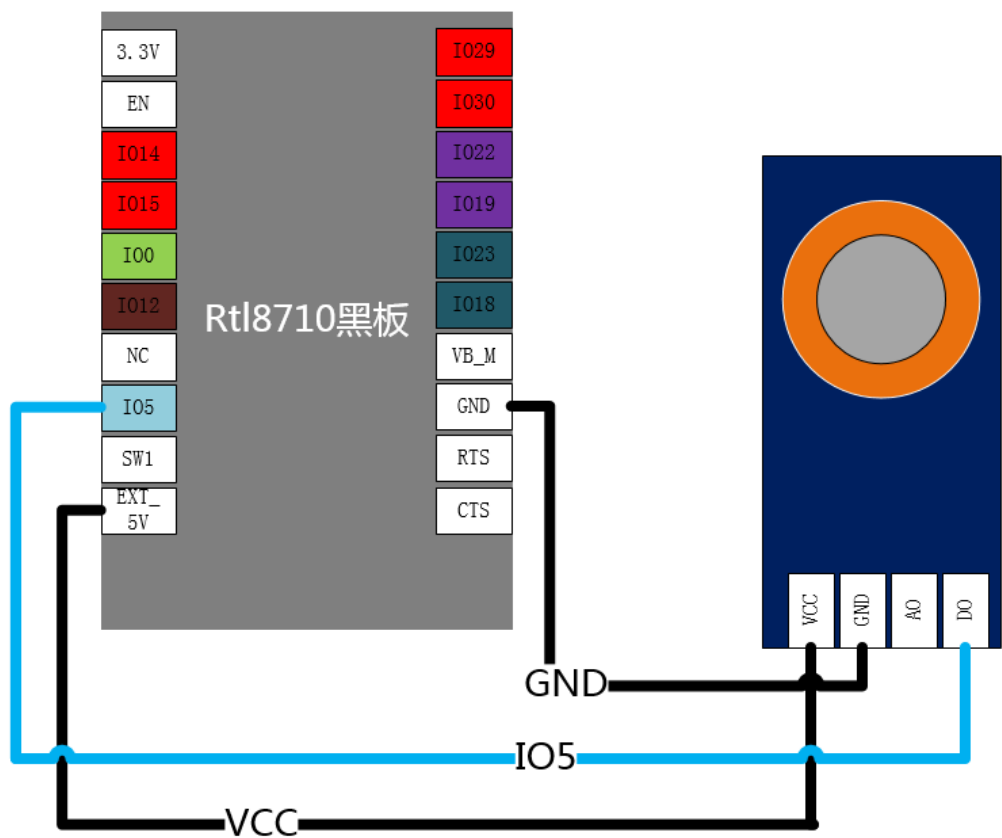
本节包含烟雾传感器、煤气传感器、酒精传感器、光敏电阻传感器、火焰传感器、声音传感器、霍尔传感器、土壤湿度传感器以及漫反射红外传感器；本模块中所提及传感器初始化等操作均一致,故归类为通用传感器，详细支持型号请参见模块参数。

A 模块参数

环境条件达不到设定的阈值,DO输出高电平，当环境条件超过设定的阈值,DO输出低电平；适用传感器有：烟雾传感器MQ-2、煤气传感器MQ-5、酒精传感器MQ-3、光敏电阻传感器、火焰传感器、声音传感器、霍尔传感器FC-03、土壤湿度传感器YL-69以及漫反射红外传感器；相关传感器图片如下：

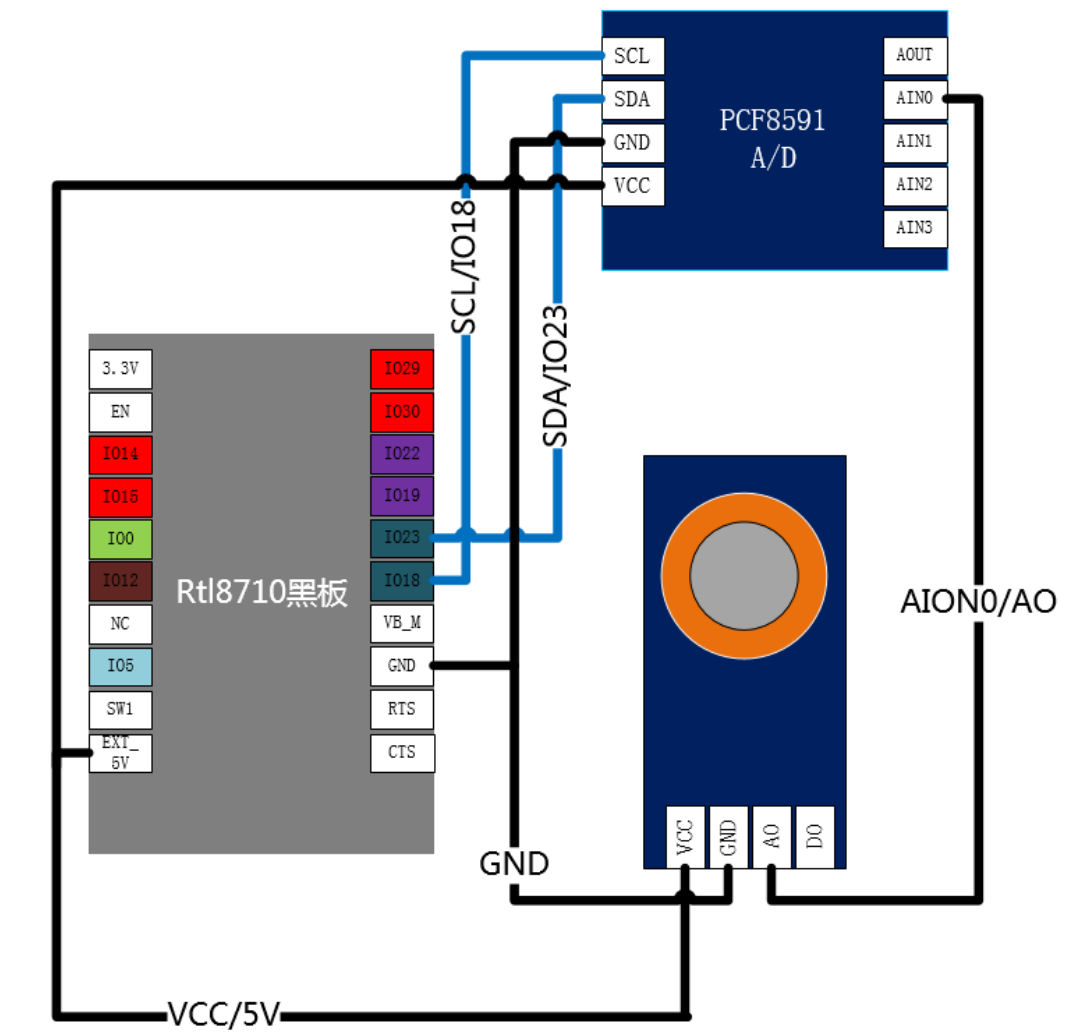


B 模块连线



DO数字量,开发板管脚和模块管脚的连线表:

序号	开发板管脚	模块管脚	说明
1	5V	VCC	VCC是传感器电源正极,范围是3.3V-5V
2	GND	GND	GND是传感器参考地
3	IO5	DO	DO是传感器数字量输出接口



AO模拟量,开发板管脚、AD转换器和模块管脚的接线表:

序号	开发板管脚	AD转换器管脚	模块管脚	说明
1	5V	VCC	VCC	外接 3.3V-5V
2	GND	GND	GND	外接GND
3	IO18/SCL	SCL	AO(连接AD转换器AIN0)	传感器模拟量输出接口
4	IO23/SDA	SDA	AO(连接AD转换器AIN0)	传感器模拟量输出接口

开发板和传感器直接相连或者开发板通过AD转换器间接和传感器相连。

4.2.1.10.2 模块接口

A 引用模块

```
var sensor = require ('sensor');
```

B 打开模块

```
sensor.open (config);
```

功能描述

打开并初始化传感器。

功能描述

打开传感器DO端口。

接口约束

无。

参数列表

config:

- pin : 引脚号,以爱联模组RTL8710开发板为例,可用引脚号为0, 5, 12, 14, 15, 18, 19, 22, 23。0引脚不推荐使用,具体参见gpio模块约束部分。
- mode : 模式,sensor.DO。

返回值:

pin_do : 返回一个sensor中GPIO端口对象。

接口示例

```
var config = {mode:sensor.DO,pin:5};  
var pin_do = sensor.open (config);
```

功能描述

打开一个遵循I2C协议的sensor端口,此端口读取通过AD转化的模拟量。

接口约束

无。

参数列表

config_1 :

- mode : sensor.AO,模拟量模式。
- i2c : 内置i2c编号,值为0或1;详细解释参见I2C模块。
- speed : 传输速率,标准模式 (0-100kb/s),快速模式(<400kb/s)。
- address : 从设备地址,仅支持7-bit address。

返回值

遵循I2C协议的sensor端口。

接口示例

```
var config_1 = {mode:sensor.A0,i2c:0,speed:100000,address:0x48};  
var i2c = sensor.open(config_1);
```

C 从打开端口中获取数据

```
pin_do.read ();
```

功能描述

调用sensor中读功能,根据打开DO端口或者AO端口,分别获取数字量或模拟量数据。

接口约束

无。

参数列表

无。

返回值

- DO方式打开：获得数字量,值域(0或1)。
- AO方式打开：获得模拟量,值域为(0-255)。

接口示例

```
var num = pin_do.read();
```

D 关闭模块

```
pin_do.close();
```

功能描述

关闭传感器端口。

参数列表

无。

返回值

无。

接口示例

```
pin_do.close();
```

E 监听端口

```
pin_do.on (event, func, arg);
```

功能描述

调用sensor模块,打开监听功能。

参数列表

- event : 事件类型
 - sensor.SENSOR_UP(外界条件正常)。
 - sensor.SENSOR_DOWN(外界条件超过阈值,传感器触发)。
- func : 回调函数,当事件event触发时所调用的函数。
- arg : 传递给回调函数的参数,通常只允许一个参数;如需传递多个参数,需要把多个参数打包在一个结构体里;如无参数,该接口将默认传递undefined。

返回值

无。

接口示例

```
pin_do.on(sensor.SENSOR_UP,function() {  
    var level = pin_do.read();
```

```
print ("sensor status is changing! level is ",level);  
});
```

4.2.1.10.3 约束

无。

4.2.1.10.4 样例

样例A

介绍

传感器超过阈值（如烟雾传感器浓度超过阈值）时,触发事件。

例程

```
var sensor = require (' sensor');  
var tim = require (' timer');  
var config = {mode:sensor.D0,pin:5};  
var pin_do = sensor.open (config);  
pin_do.on(sensor.SENSOR_UP,function() {  
    var level = pin_do.read();  
    print ("sensor status is changing! level is ",level);  
});
```

样例B

介绍

一秒为周期,周期性获取AD转换器读到的模拟量。

例程

```
var sensor = require (' sensor');  
var tim = require ("timer");  
var config_1 = {mode:sensor.A0,i2c:0, speed:100000, address:0x48};  
var i2c = sensor.open(config_1);  
tim.setInterval(function() {  
    var level = i2c.read();  
    print("level", level);  
}, 1000);
```

4.2.2 控制模块接口

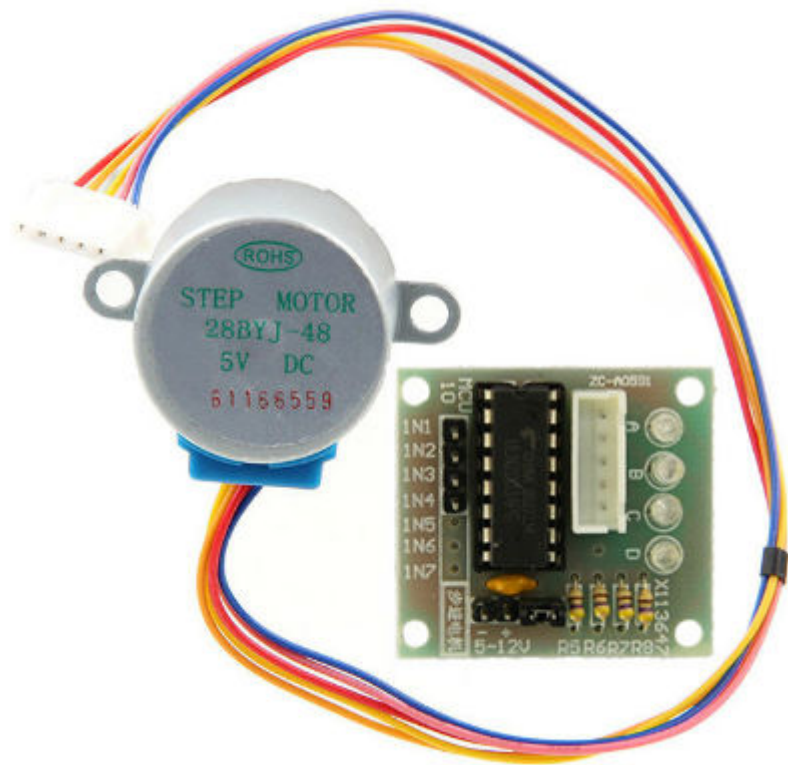
4.2.2.1 5V 步进电机

4.2.2.1.1 介绍

步进电机是将电脉冲信号转变为角位移或线位移的开环控制电机，是现代数字程序控制系统中的主要执行元件，应用极为广泛。

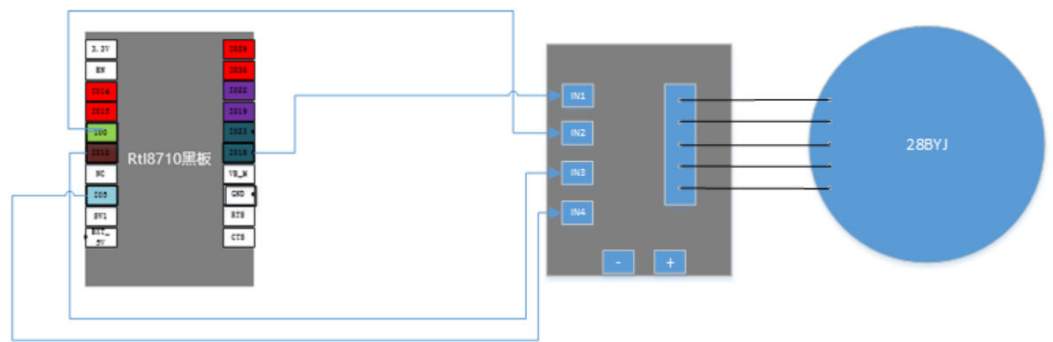
A 模块参数

此模块提供接口给用户控制28BYJ-48步进电机旋转。28BYJ-48步进电机如下图：



B 模块连线

由于板子输出的电流不足以带动28BYJ-48电机，需要ULN2003驱动模块才能驱动。ULN2003驱动板上的接口有防呆口设置，电机接上ULN2003后，开发板与驱动模块引脚对应连线如下图：



序号	开发板管脚	模块引脚	说明
1	IO18	IN1	与电机蓝线短接，用于输出高/低电平控制电机运动
2	IO0	IN2	与电机粉线短接，用于输出高/低电平控制电机运动

3	IO12	IN3	与电机黄线短接， 用于输出高/低电平 控制电机运动
4	IO5	IN4	与电机橙线短接， 用于输出高/低电平 控制电机运动
5		+	外接电源正极， 5~12V
6		-	外接电源负极

4.2.2.1.2 模块接口

A 引用模块

```
var motor= require ('nanostepper');
```

B 打开模块

```
motor.open(pin_array,work_mode);
```

功能描述

motor.open根据参数配置打开引脚并设置相关属性，成功初始化后返回一个motorObj对象。

接口约束

无。

参数列表

- pin_array : 控制引脚数组，数组内引脚顺序按照与IN1~IN4连接的顺序排列，以爱联模组RTL8710开发板为例，可用引脚号为0，5，12，14，15，18，19，22，23。0引脚不推荐使用，具体参见gpio模块约束部分。
- work_mode: 步进电机的运行模式，根据设置值不同，以不同方式运行。
 - 0 : 单四拍
 - 1 : 双四拍
 - 2 : 八拍

返回值

motorobj : 电机对象。

接口示例

```
motor.open([18, 0, 12, 5], 0);
```

C 驱动电机旋转

```
myMotor.to(angle,dir);
```

功能描述

控制电机旋转指定角度。

接口约束

无。

参数列表

- **angle**: 范围(任意整数)，电机旋转的角度。
- **dir**: 范围(0,1)，电机旋转的方向，设置为0时逆时针移动，设置为1时顺时针移动，缺省值为0。

返回值

无。

接口示例

```
myMotor.to(90, 1);
```

4.2.2.1.3 约束

无。

4.2.2.1.4 样例

介绍

电机以单四拍运行，顺时针移动90°。

连线图

参考4.2.2.1.1模块连线。

例程

```
var motor = require("nanostepper");
var myMotor = motor.open([18, 0, 12, 5], 0);
/* 参数说明:
io18连IN1
io0连IN2
io12连IN3
io5连IN4
电机以单四拍运行。
*/
myMotor.to(90, 1);
/* 电机顺时针移动90° */
```

4.2.2.2 42V 步进电机

4.2.2.2.1 介绍

步进电机是将电脉冲信号转变为角位移或线位移的开环控制电机，是现代数字程序控制系统中的主要执行元件，应用极为广泛。

A 模块参数

此模块提供接口给用户控制SL42STH34-1504A 步进电机旋转, 需结合TB600驱动模块使用, TB6600是两相步进电机驱动。可实现正反转控制, 通过 3 位拨码开关, 可以选择 7 档细分控制 (1、2/A、2/B、4、8、16、32), 8 档电流控制 (0.5A、1A、1.5A、2A、2.5A、2.8A、3.0A、3.5A)。适合驱动 57、42 型两相、四相混合式步进电机, 能达到低振动、低噪声、高速度的驱动效果。电机如下图:



TB6600输入输出

- 信号输入端
 - PUL+ : 脉冲信号输入正
 - PUL- : 脉冲信号输入负
 - DIR+ : 电机正、反转控制正
 - DIR- : 电机正、反转控制负
 - EN+ : 电机脱机控制正
 - EN- : 电机脱机控制负
- 电机绕组连接
 - A+ : 连接电机绕组 A+相
 - A- : 连接电机绕组 A-相
 - B+ : 连接电机绕组 B+相
 - B- : 连接电机绕组 B-相
- 电源电压连接
 - VCC : 电源正端 “+”
 - GND : 电源负端 “-” 注意：DC9-42V,不可以超过此范围，否则将无法正常工作，甚至损坏驱动器。

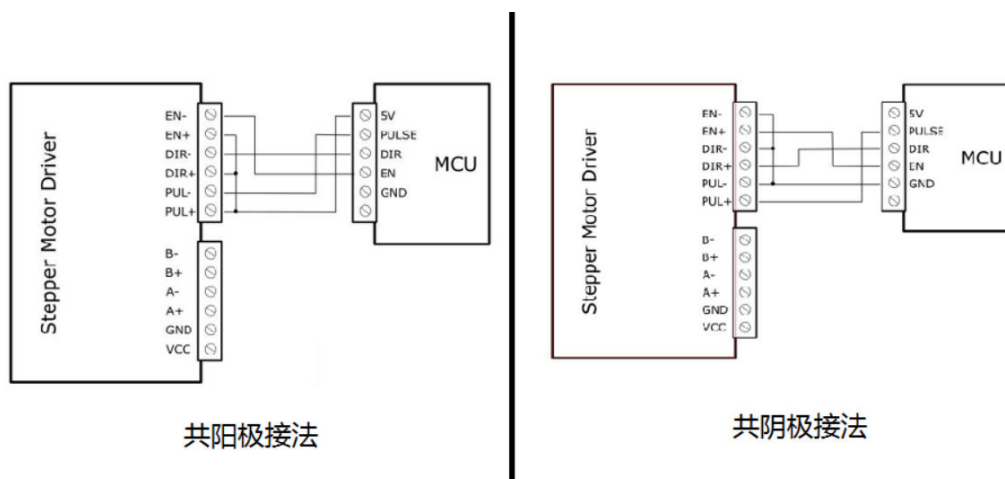
输入端接线说明

输入信号共有三路，它们是：

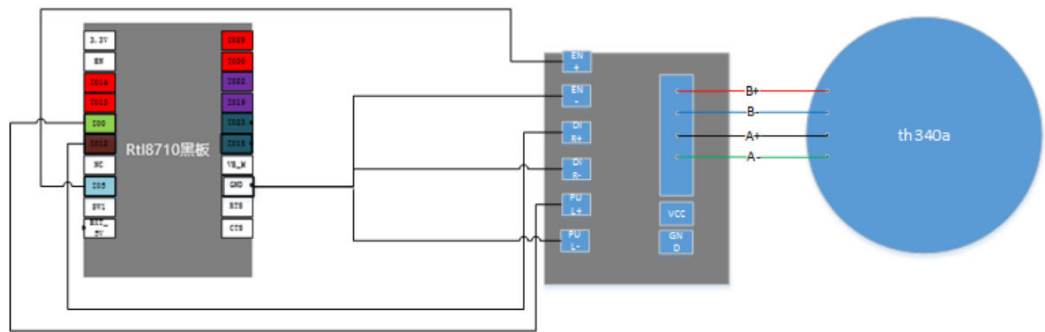
- 步进脉冲信号 PUL+, PUL-。
- 方向电平信号 DIR+, DIR-。
- 脱机信号 EN+, EN-。

B 模块连线

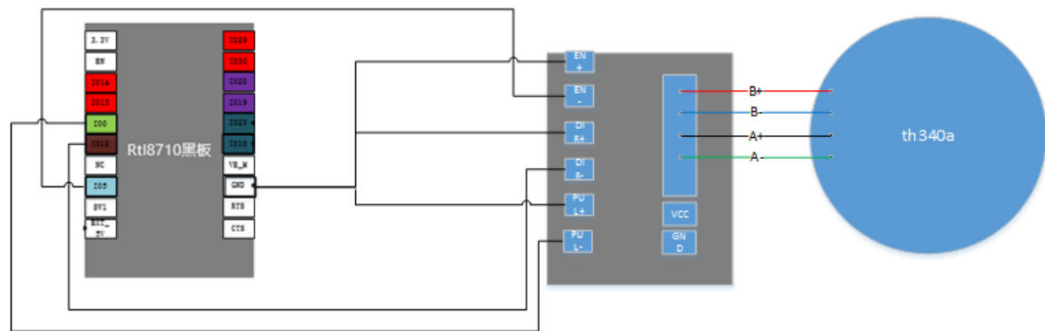
TB6600驱动板连线方式分为两种，用户可根据需要采用共阳极接法或共阴极接法，接法如下图：



共阴极接法



共阳极接法



序号	开发板管脚	TB6600驱动板	步进电机引脚	说明
1	IO5	EN+(共阴)/EN-(共阳)		共阴极接法 IO5接EN+，共 阳极接法IO5 接EN-
2	IO12	DIR+(共阴)/ DIR-(共阳)		共阴极接法 IO5接DIR+， 共阳极接法 IO5接DIR-
3	IO0	PUL+(共阴)/ PUL-(共阳)		共阴极接法 IO5接PUL+， 共阳极接法 IO5接PUL-
4		B+	红线	与电机红线短 接，用于输出 高/低电平控制 电机运动

序号	开发板管脚	TB6600驱动板	步进电机引脚	说明
5		B-	蓝线	与电机蓝线短接，用于输出高/低电平控制电机运动
6		A+	黑线	与电机黑线短接，用于输出高/低电平控制电机运动
7		A-	绿线	与电机绿线短接，用于输出高/低电平控制电机运动
8		GND		外接电源地
9		VCC		外接电源正极(9-12V)

由于SL42STH34-1504A模块的驱动电流为1.5A,需要将TB6600的S4~S6设置为ON、ON、OFF状态。

4.2.2.2.2 模块接口

A 引用模块

```
var motor= require ('stepper');
```

B 打开模块

```
motor.open(pin_array,connect_mode,step);
```

功能描述

motor.open根据config配置打开引脚并设置相关属性，成功初始化后返回一个motorObj对象。

接口约束

无。

参数列表

- pin_array : 控制引脚数组，数组内引脚顺序按照EN、DIR、PUL顺序输入，以爱联模组RTL8710开发板为例，可用引脚号为0，5，12，14，15，18，19，22，23。0引脚不推荐使用，具体参见gpio模块约束部分。
- connect_mode: 驱动器的连接方式。根据实际连线方式设置该值：
 - 0 : 共阴极连接
 - 1 : 共阳极连接
- step : 细分控制的档位，按照实际S1~S3的打开情况按16进制设置。例如设置成16细分档时，S1~S3的状态分别为OFF、OFF、ON，则应设置该值为0x01(S1对应bit2，S2对应bit1，S3对应bit0)。

返回值

motorObj: 电机对象。

接口示例

```
motor.open([5, 12, 0], 0, 0x0);  
/* 参数说明:  
采用共阴极连线方式, 故connect_mode值为0,  
设置成32细分档, S1~S3都为OFF, 即step值为0x0,  
io5连EN+  
io12连DIR+  
io0连PUL+  
*/
```

C 驱动电机旋转

```
motorObj.to(angle);
```

功能描述

控制电机旋转指定角度。

参数列表

angle: 电机旋转的角度。

返回值

无。

接口示例

```
motorObj.to(90);
```

4.2.2.2.3 约束

无。

4.2.2.2.4 样例

介绍

发射脉冲驱使电机顺时针移动90°，再次发射脉冲驱使电机逆时针移动90°。

例程

```
var motor = require("stepper");  
var myMotor = motor.open([5, 12, 0], 0, 0x0);  
  
myMotor.to(90);  
/* 电机顺时针移动90° */  
myMotor.to(-90);  
/* 电机逆时针移动90° */
```

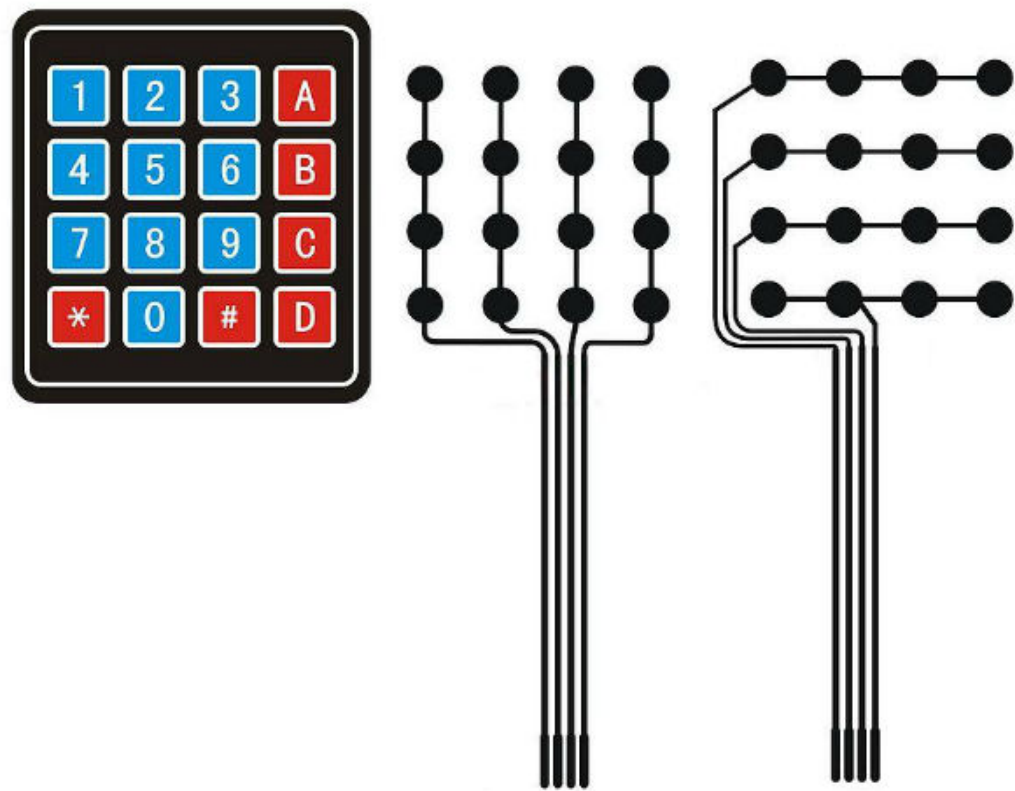
4.2.2.3 薄膜键盘

4.2.2.3.1 介绍

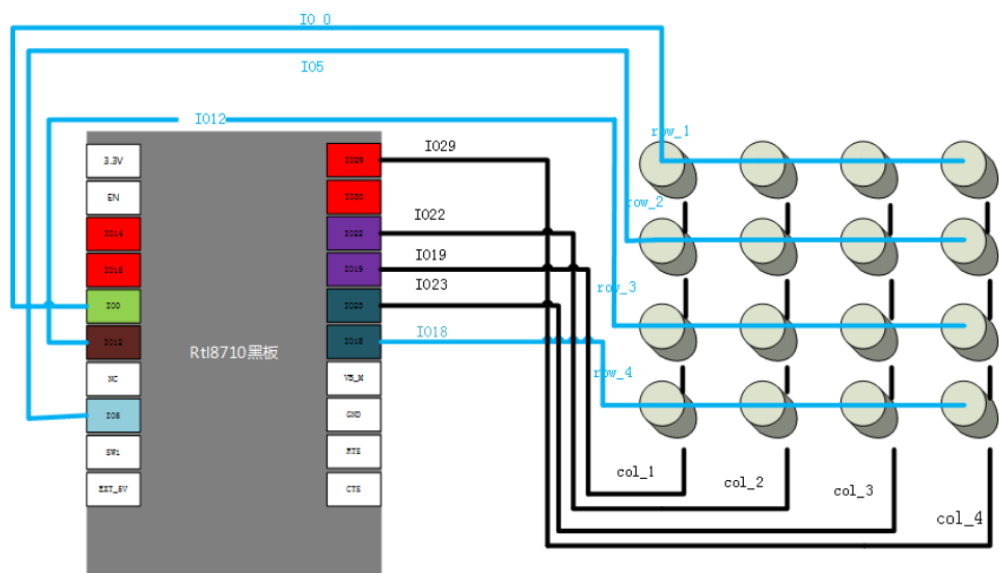
按键式控制输入设备。

A 模块参数

矩阵键盘是单片机外部设备中所使用的排布类似于矩阵的键盘组。在键盘中按键数量较多时，为了减少I/O口的占用，通常用矩阵键盘来替代独立按键。4×4薄膜键盘如下图所示。（注：此模块支持自定义n*m的矩阵键盘（ $n > 0, m > 0, 0 < n + m < \text{可使用的IO数}$ ））。



B 模块连线



4.2.2.3.2 模块接口

A 引用模块

```
var keyboard = require("keyboard");
```

B 打开模块

```
keyboard.open(rowPinArray, colPinArray);
```

功能描述

keyboard.open打开引脚并设置相关属性，成功初始化后返回一个keyboard对象。

接口约束

无。

参数列表

- rowPinArray: 行IO引脚号数组
- colPinArray: 列IO引脚号数组 注：目前使用的IO口为：以爱联模组RTL8710开发板为例，可用引脚号为0, 5, 12, 14, 15, 18, 19, 22, 23。0引脚不推荐使用，具体参见gpio模块约束部分。此模块支持自定义row*col的矩阵键盘 ($0 < \text{row} < 7$, $0 < \text{col} < 7$, $2 < \text{row} + \text{col} < 8$ (所有使用的IO数))。

返回值

打开成功时返回一个keyboard对象。

接口示例

```
var rowPin = [0, 5, 12, 18]; //键盘1~4行分别连接I00, I05, I012, I018
var colPin = [19, 22, 23, 29]; //键盘1~4列分别连接I019, I022, I023, I029
var myKB = keyboard.open(rowPin, colPin);
```

C 设置键值

```
keyboardPin.setValue(valueArray);
```

功能描述

设置矩阵键盘不同键位所对应的值。

接口约束

无。

参数列表

- valueArray: 矩阵键盘对应键位的值。键值为一个字符，如果设置成字符串，则以字符串的第一个字符为键值。键值数组需要和打开时设置的行列数相同，否则设置失败。

返回值

无。

接口示例

```
var rowPin = [0, 5, 12, 18];
var colPin = [19, 22, 23, 29];
var myKB = keyboard.open(rowPin, colPin);
/* 键值个数为： rowPin * colPin */
myKB.setValue(['1', '2', '3', 'A'], // 键盘列数为4，每个数组需要存4个键值
               ['4', '5', '6', 'B'], // 键盘行数为4，需要设置4个数组
               ['7', '8', '9', 'C'],
               ['*', '0', '#', 'D']);
```

D 设置按键事件

```
keyboardPin.on(event, function);
```

功能描述

设置矩阵键盘键位按下时触发的事件。

接口约束

无。

参数列表

- **event** : 事件类型
 - **KEY_DOWN** : 按键按下事件
- **function** : **event**发生时，触发的回调函数。如: `function(data){...}`，**data**是一个buffer，表示触发**event**的键值。

返回值

无。

接口示例

```
myKB.on(keyboard.KEY_DOWN, function(data) {  
    if (data.toString() == '1')  
    {  
        print ("pull down key 1");  
    }  
});
```

4.2.2.3.3 约束

由于IO29与IO30复用uart功能，所以在使用该模块时，如果使用IO29连接键盘，并且使用IO30，可能出现键盘功能异常。

4.2.2.3.4 样例

介绍

打印矩阵键盘键值，按下按键会打印对应的键值。

例程

```
var keyboard = require("keyboard");  
var rowPin = [0, 5, 12, 18];  
var colPin = [19, 22, 23, 29];  
  
var myKeyboard = keyboard.open(rowPin, colPin);  
myKeyboard.setValue(['1', '2', '3', 'A'],  
                    ['4', '5', '6', 'B'],  
                    ['7', '8', '9', 'C'],  
                    ['*', '0', '#', 'D']);  
myKB.on(keyboard.KEY_DOWN, function(data) {  
    if (data.toString() == '1')  
    {  
        print ("pull down key 1");  
    }  
});
```

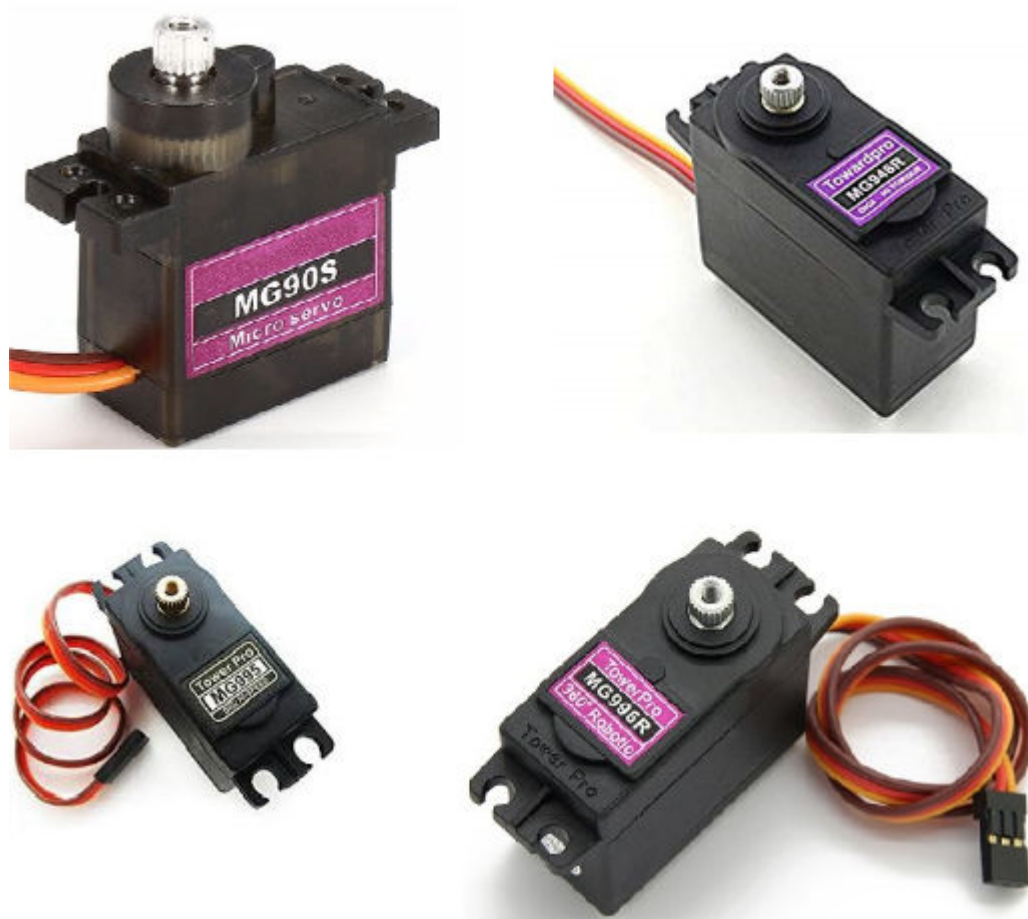
4.2.2.4 舵机

4.2.2.4.1 介绍

舵机通常包括一个小型直流电机，加上传感器、控制芯片、减速齿轮组，装进一体化外壳。能够通过输入信号（一般是PWM信号，也有的是数字信号）控制旋转角度。因传统上用于飞行器舵面控制而得名。

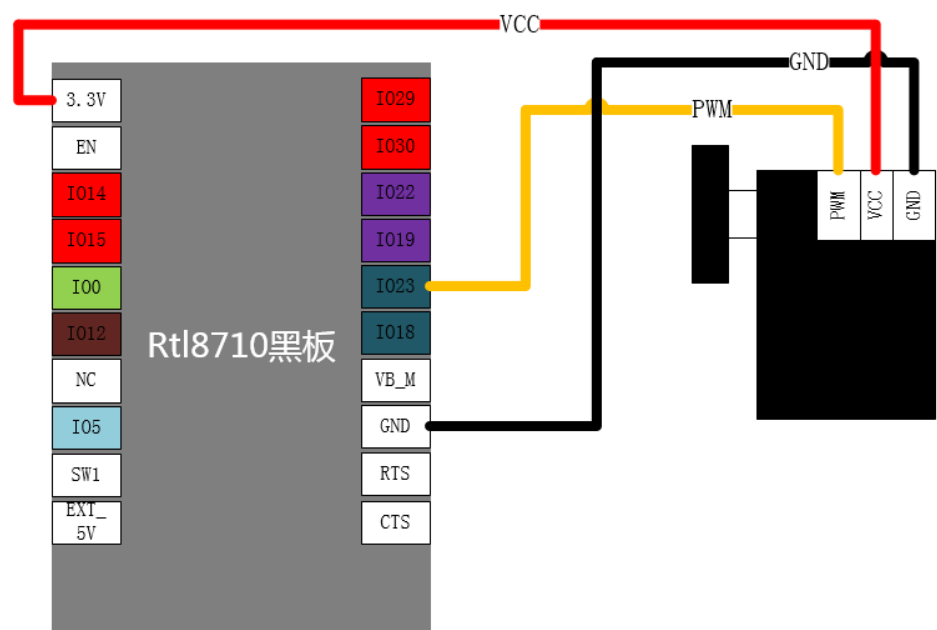
A 模块参数

舵机模块提供驱动舵机的接口给用户，适用于工作方式与MG90S、MG996R模块相似的舵机。

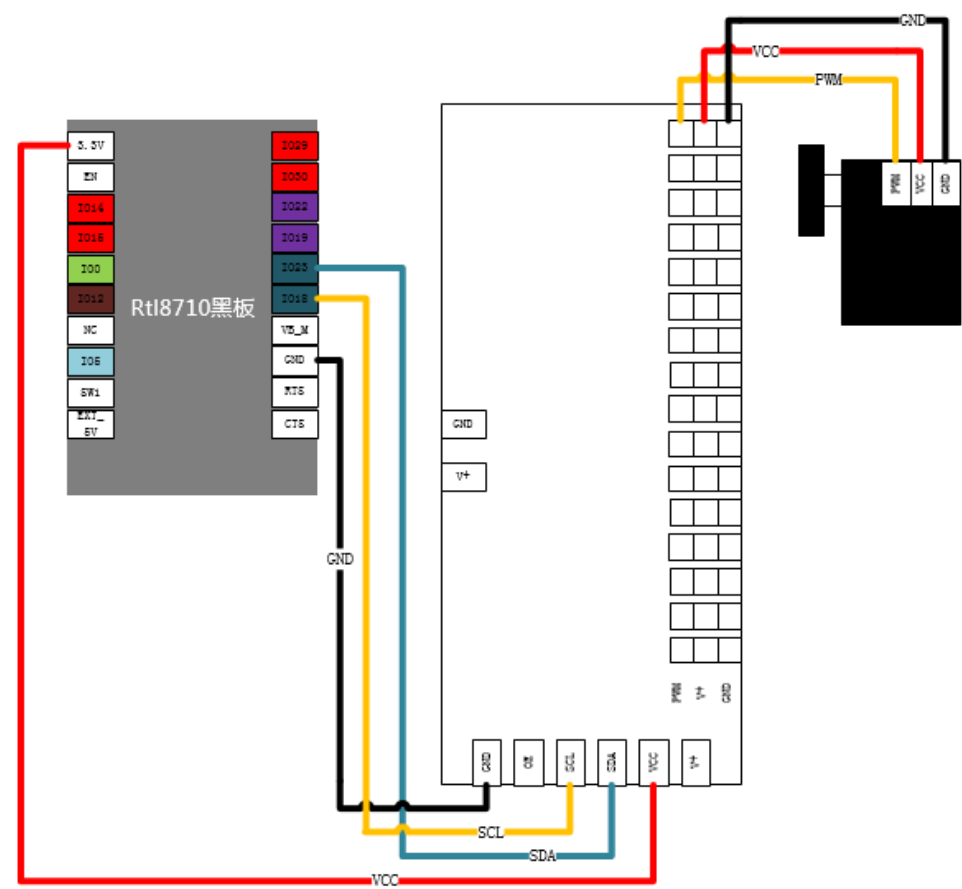


该模块可以选用有PWM功能的GPIO口产生输出信号，进行单个舵机控制，也可以采用PCA9685模块转接板进行多个舵机的控制，选用PCA9685转接板进行控制时，开发板和转接板采用I2C通信方式，转接板接受开发板发出的数据并进行处理，产生多个PWM信号对多个舵机进行控制，转接板由另外的5V电源进行供电。

B 模块连线



或者



开发板管脚与模块管脚的连线表：

序号	开发板管脚	模块管脚	说明
1	Ext_5V	VCC	电源
2	GND	GND	接地
3	GPIO	PWM	PWM控制信号

或

序号	开发板管脚	模块管脚	说明
1	Ext_5V	VCC	5V电源
2	GND	GND	接地
3	IO23/I2C1_SDA	SDA	SDA，即 I2C_SDA，是I2C 数据线
4	IO18/I2C1_SCL	SCL	SCL，即 I2C_SCL，是I2C时 钟线

4.2.2.4.2 模块接口

A 引用模块

```
var servo = require("servo");
```

B 打开模块

```
servo.open(config);
```

功能描述

对需要使用的信号输出引脚进行配置打开并设置相关属性。

接口约束

无。

参数列表

在对舵机进行控制的时候，必须要配置的为可以产生PWM引脚pin。

- pin: 选用的可以产生PWM信号的引脚。

除了引脚外，其余可以根据选择进行配置的属性有id、type、rangeMin、rangeMax、invert、startAt、center、controller、i2cAddr、i2cNumber、model、minDuty、maxDuty。

- id: 用户自定义ID号，缺省时默认为空。
- type: 舵机的类型，设为“servo.STANDARD”则为标准舵机，可旋转角度为为0~180°；设置为“servo.CONTINUOUS”为连续旋转舵机旋转角度为360°。缺省时为“servo.STANDARD”。目前暂只支持servo.STANDARD类型。

- **rangeMin**: 舵机旋转角度的最小值, 缺省时为0。
- **rangeMax**: 舵机旋转角度的最大值。type为servo.STANDARD时, 缺省值为180°; type为servo.CONTINUOUS时, 缺省值为360°。
- **invert**: 翻转舵机移动方向, 缺省时为false。该值为false时, 舵机默认角度变大方向为正方向。(如舵机默认为逆时针旋转为正方向。设置为true后, 舵机以顺时针旋转为正方向)。
- **startAt**: 用于初始化舵机的角度, 缺省时为rangeMin。创建对象过程中会改变舵机位置至该角度。
- **center**: 如果为true, 则覆盖startAt并将舵机移动到舵机移动范围的中间值。
- **controller**: 设置控制驱动板, 缺省时直接通过IO口输出PWM控制; 设置为servo.PCA9685时, 通过I2C驱动PCA9685输出PWM控制舵机, 默认addr为 0x40。
- **i2cNumber**: 用于设置板子使用的i2c引脚。取值范围为0或1; 缺省时, 该值为0。默认speed为1Mb/s。
 - 0: SDA引脚为IO23 (即23), SCL引脚为IO18 (即18)。
 - 1: SDA引脚为IO19 (即19), SCL引脚为IO22 (即22)。
- **i2cAddr**: 用于设置PCA9685控制板的7位从机地址。缺省时, 该值为0x40; 注: 当controller缺省时, 此属性设置无效。
- **model**: 设置舵机的类型。servo.MG996R代表MG996R舵机, 其移动0~180°对应的高电平时长为600us-2200us, MG995舵机与MG996R相同, 控制演示如下图所示; servo.MG90S代表MG90S舵机, 其移动0~180°对应的高电平时长为500us-2500us。缺省时舵机类型为MG90S。
- **minDuty**: 用于设置舵机最小角度对应的高电平时长, 单位为us。该值必须与maxDuty一起被设置才能生效。
- **maxDuty**: 用于设置舵机最大角度对应的高电平时长, 单位为us。该值必须与minDuty一起被设置才能生效。注: minDuty/maxDuty两个参数是预留给用户, 用于微调舵机。

返回值

servo对象。

接口示例

使用GPIO口产生PWM时:

```
var servo = require("servo");
var config = { pin : 5 };
var myServo = servo.open(config);
```

使用PCA9685转接板驱动舵机时:

```
var config = { pin : 5,
               rangeMin : 30,
               rangeMax : 150,
               startAt : 90,
               center : true,
               controller : servo.PCA9685,
               model : servo.MG90S
             };
var myServo = servo.open(config);
```

C 设置角度

```
servo.to(degrees, time, rate);
```

功能描述

将舵机移动到指定角度，单位为度，范围为0-180（或range的范围）。如果设置了time，舵机将在这段时间内移动到该位置；如果同时设置了rate，则将在time时间内进行rate次移动到degrees角度。如果指定的角度与当前角度相同，则不发送任何命令。

接口约束

无。

参数列表

- degrees: 控制舵机移动的角度位置，范围为0~180°。
- time: 控制舵机偏移角度所用的时长，单位为ms。
- rate: 转到目标位置所移动的步数，默认为1，可以根据需要设定，只能为正整数。

返回值

无。

接口示例

```
myServo.to(90); //舵机旋转90°
```

或

```
myServo.to(90, 500); //舵机在500ms内旋转90°
```

或

```
servo.to(90, 500, 10); // 舵机在500ms内用10步旋转90°
```

D 设置到最小角度

```
myServo.min();
```

功能描述

将舵机移动到最小值(rangeMin)。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
var servo = require("servo");
var config = { pin : 5 };
var myServo = servo.open(config);
myServo.min(); // 舵机旋转到角度最小值，未设置是最小值为0。
```

或

```
var servo = require("servo");
var config = { pin : 5,
               rangeMin: 45
             };
var myServo = servo.open(config);
myServo.min(); // 舵机旋转至最小值为45°。
```

E 设置到最大角度

```
myServo.max();
```

功能描述

将舵机移动到最大值(rangeMax)。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
var servo = require("servo");
var config = { pin : 5 };
var myServo = servo.open(config);
myServo.max();// 舵机旋转至最大值，默认为180°。
```

或

```
var servo = require("servo");
var config = { pin : 5,
               rangeMax: 150
             };
var myServo = servo.open(config);
myServo.max();// 舵机旋转至最大值150°。
```

F 设置到中间角度

```
myServo.center();
```

功能描述

将舵机移动到中间 ((rangeMax+rangeMin)/2)。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
var servo = require("servo");
var config = { pin : 5};
var myServo = servo.open(config);
myServo.center();//默认最小值为0°，最大值为180°，此时中间值为90°。
```

或

```
var servo = require("servo");
var config = { pin : 5,
               rangeMin: 40,
```

```
        rangeMax: 80
    };
    var myServo = servo.open(config);
    myServo.center();// (40 + 80) / 2 = 60, 旋转到60°。
```

G 设置到初始角度

```
myServo.home();
```

功能描述

将舵机的角度移动到startAt的值。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
var servo = require("servo");
var config = { pin : 5 };
var myServo = servo.open(config);
myServo.to(90, 500);// 舵机旋转到90°。
myServo.home();// 默认startAt的角度为0°，舵机旋转到0°。
```

或

```
var servo = require("servo");
var config = { pin : 5,
               startAt : 20
             };
var myServo = servo.open(config);
myServo.to(90, 500);// 舵机旋转到90°。
myServo.home();// 设置startAt的角度为20°，舵机旋转到20°。
```

H 停止舵机

```
myServo.stop();
```

功能描述

在无延时情况下，停止移动的舵机。

参数列表

无。

返回值

无。

接口示例

```
var servo = require("servo");
var config = { pin : 5 };
var myServo = servo.open(config);
myServo.to(90);
myServo.stop();//* 在无延时情况下停止舵机，如果在该函数之前有延时，则此函数无效 */
```

I 初始化舵机数组

```
servo.initServos();
```

功能描述

初始化一个舵机数组，用于保存和控制多个舵机对象。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
var myServos = servo.initServos();
```

J 初始化舵机

```
myServos.multOpen (config, ...);
```

功能描述

根据config初始化舵机，并将舵机添加到舵机数组中。

接口约束

无。

参数列表

- config：舵机的初始化信息，配置信息参照1.2。
- ...：省略号，可一次性传入多个config，一次性打开多个舵机。

返回值

无。

接口示例

```
var servo = require("servo");  
var config1 = {pin : 1, controller : servo.PCA9685 };  
var config2 = {pin : 2, controller : servo.PCA9685 };  
var config3 = {pin : 3, controller : servo.PCA9685 };  
var myServos = servo.initServos();  
myServos.multOpen (config1, config2, config3);
```

K 添加对象

```
myServos.addServo (servoPin, ...);
```

功能描述

向舵机数组中添加舵机对象。

接口约束

无。

参数列表

- servoPin : 已经初始化成功的舵机对象。
- ... : 省略号, 可以一次传入多个舵机对象。

返回值

无。

接口示例

```
var servo = require("servo");
var config1 = {pin : 1, controller : servo.PCA9685 };
var config2 = {pin : 2, controller : servo.PCA9685 };
var config3 = {pin : 3, controller : servo.PCA9685 };
var servo1 = servo.open(config1);
var servo2 = servo.open(config2);
var servo3 = servo.open(config3);
var myServos = servo.initServos();
myServos.addServo (servo1, servo2, servo3);
```

L 设置舵机组角度

```
myServos.to (degrees, time, rate);
```

功能描述

与servoPin.to(degrees, time, rate)函数相似, 区别在于传入的degrees可以是一个值, 也可以是一个数组。

接口约束

无。

参数说明

- degrees: 控制舵机移动的位置。
 - 当degrees是一个数值时, 所有舵机都会移动到该角度, 范围为0~180°;
 - 当degrees是一个数组时, 数组中的数值与舵机数组中的舵机一一对应。当degrees数组中的数值个数n小于舵机个数m时, 舵机数组只有前n个舵机会移动, 剩余舵机不移动; 当n大于m时, 多余的数值会被忽略, 范围为0~180°。
- time: 控制舵机偏移角度所用的时长, 单位为ms。
- rate : 转到目标位置所移动的步数。

返回值

无。

接口示例

```
var servo = require("servo");
var tim = require ('timer');
var config1 = {pin : 1, controller : servo.PCA9685 };
var config2 = {pin : 2, controller : servo.PCA9685 };
var config3 = {pin : 3, controller : servo.PCA9685 };
var myServos = servo.initServos();
myServos.multOpen (config1, config2, config3);
myServos.to (90); /* 数组中所有舵机移动到90° */
tim.setDelay(1000);/* 延时1000ms, 等待舵机移动到90° */
/* 数组中第一个舵机移动到0° ,
 * 数组中第二个舵机移动到180° ,
 * 数组中第三个舵机不移动,
 * 所有舵机在1000ms内, 分3步移动到指定位置。
 */
myServos.to ([0, 180], 1000, 3);
```

M 设置舵机组动作

```
myServos.setSegment (config);
```

功能描述

设置一连串舵机组合动作，并让舵机按动作移动。

接口约束

无。

参数列表

config: 为servos设置舵机动作，主要包括两个个主要属性：**keyFrames**和**moveTimes**

- **operateMode**: 表示输入舵机角度的输入模式，有且只有0和1两种输入模式。
 - 0: 表示输入舵机角度为绝对值。
 - 1: 表示输入舵机的角度为上一个角度的相对值。
- **keyFrames**: 关键帧数组，数值数组的数组。存放舵机组每个片刻的角度，可存放多个片刻。
- **moveTimes**: 每个关键帧预留的移动时间。

返回值

无。

接口示例

```
var servo = require("servo");
var config1 = {pin : 1, controller : servo.PCA9685 };
var config2 = {pin : 2, controller : servo.PCA9685 };
var config3 = {pin : 3, controller : servo.PCA9685 };

var myServos = servo.initServos();

var ser1 = servo.open(config1);
var ser2 = servo.open(config2);
var ser3 = servo.open(config3);

myServos.addServo (ser1, ser2, ser3);
var segment = {
  operateMode:0,
  keyFrames:[
    [20, 30, 40], /* 片刻1: ser1移动至20° , ser2移动到30° , ser3移动到40° */
    [30, 40, 50], /* 片刻2: ser1移动至30° , ser2移动到40° , ser3移动到50° */
    [40, 50, 60] /* 片刻3: ser1移动至40° , ser2移动到50° , ser3移动到60° */
  ],
  moveTimes:[1000, 500, 2000]
  /* 在舵机数组从初始状态向片刻1的状态进行改变时，会预留1000ms 的时间，以保证所有舵机可达到预计角度；
  在舵机数组从片刻1的状态向片刻2的状态进行改变时，会预留500ms 的时间，以保证所有舵机可达到预计角度；
  在舵机数组从片刻2的状态向片刻3的状态进行改变时，会预留2000ms 的时间，以保证所有舵机可达到预计角度。*/
}
myServos.setSegment (segment); /* 将动作片段传递舵机数组，让舵机数组按照设定完成动作 */
```

4.2.2.4.3 约束

无。

4.2.2.4.4 样例

样例A

介绍

通过 pwm/pca9685控制板使用舵机。

例程

```
var servo = require("servo");
var tim = require('timer');
var config = { pin : 5 };
/* 在此配置下，要将板子的I05连接到舵机的橙色线，3.3v连接到舵机红色线， GND连接到舵机的黑色线 */
/*
var config = {
  id: 0,
  pin: 10,
  type: servo.STANDARD,
  rangeMin: 20,
  rangeMax: 150,
  invert: false,      // Invert all specified positions
  startAt: 90,       // Immediately move to a degree
  center: true,
  controller : servo.PCA9685
};
// 在此配置下，的连线方式：
// 1. 要将板子的I023 连接到pca9685控制板(后简称控制板)的SCK, I018连接到控制
// 板的SDA， 3.3v连接控制板的VCC， GND连接控制板的GND， 5V连接控制板的V+。
// 注：控制板尽量单独供电。
// 2. 将舵机插到控制板的10排。
*/
var myServo = new servo.open(config);

/* Servo API */
// set the servo to the minimum degrees
// defaults to 0
myServo.min();
tim.setDelay(1000);/* 延时1000ms,等待舵机移动到最小角度 */
// set the servo to the maximum degrees
// defaults to 180
myServo.max();
tim.setDelay(1000);/* 延时1000ms,等待舵机移动到最大角度 */
// centers the servo to 90°
myServo.center();
tim.setDelay(1000);/* 延时1000ms,等待舵机移动到中间角度值 */
//
// Moves the servo to position by degrees
//
myServo.to(90);
```

样例B

介绍

通过舵机数组控制多个舵机完成自定义动作。

例程

```
var servo = require("servo");
var config1 = {pin : 1, controller : servo.PCA9685, startAt: 90 };
var config2 = {pin : 2, controller : servo.PCA9685, startAt: 90 };
var config3 = {pin : 3, controller : servo.PCA9685, startAt: 90 };

var myServos = servo.initServos();
var ser1 = servo.open(config1);
var ser2 = servo.open(config2);
var ser3 = servo.open(config3);
```

```
/*
‘o’ 的位置代表舵机的位置，下图为舵机和支架构成的简易腿部的示意图。

      o ①
      |
      o ②
      |
③ o_
A 腿部直立
  ① ②
  o - o
    |
  ③ o-
B 抬起膝盖
动作A->B的过程中，各个舵机的变化：
① 由90° 逆时针旋转45° 到达135° ；
② 由90° 顺时针旋转45° 到达45° ；
③ 90° 没有变化；
*/
myServos.addServo (ser1, ser2, ser3);
var up_leg = {
  operateMode:0,
  keyFrames:[
    [90,90,90], /* 片刻1：三个舵机都为90°，效果如动作A(腿部直立) */
    [110,70,90], /* 片刻2：ser1移动至110°，ser2移动到70°，ser3不动°，该片刻是为了防止角度移动过大而设置的过渡帧 */
    [135,45,90] /* 片刻3：三个舵机移动到B动作的预设位置 */
  ],
  moveTimes:[500, 1000, 1000]
}; /* 将动作片段传递舵机数组，让舵机数组按照设定完成动作 */
myServos.setSegment (up_leg);
```

4.2.2.5 继电器

4.2.2.5.1 介绍

继电器（Relay），也称电驿，是一种电子控制器件，它具有控制系统（又称输入回路）和被控制系统（又称输出回路），通常应用于自动控制电路中，它实际上是用较小的电流去控制较大电流的一种“自动开关”。

A 模块参数

本模块使用的是电磁继电器，如下图所示。

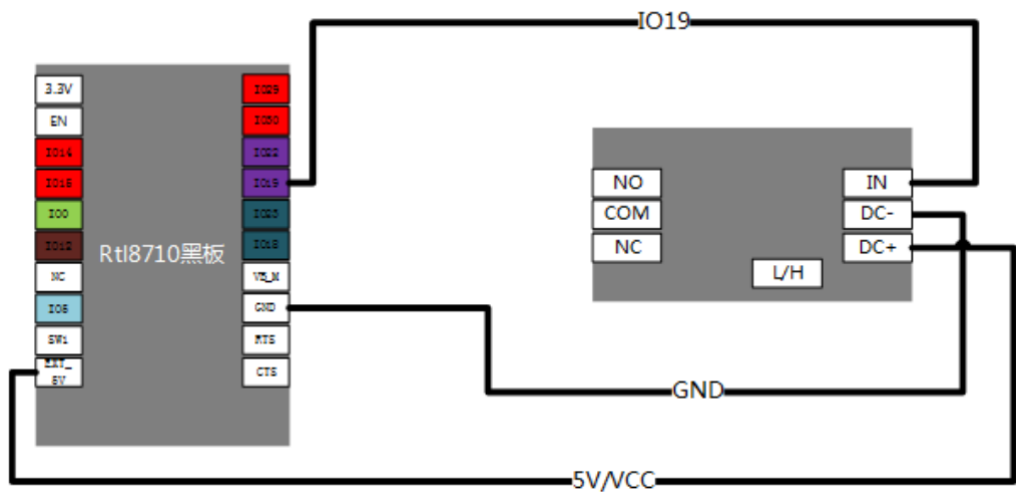


5V继电器参数列表：

参数	数值
电源	< 5V
电流	5 ~190 mA
常开接口最大负载	交流250V/10A ； 直流 30V/10A
尺寸	50 * 25 * 18.5 mm

1.2 模块连线

开发板管脚与模块管脚连线

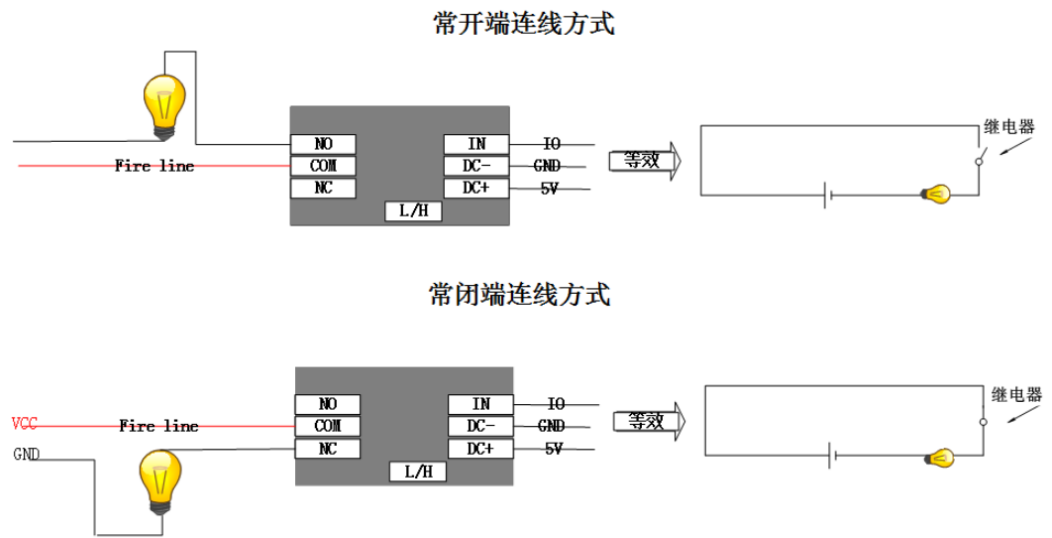


开发板管脚与模块管脚的连线表：

序号	开发板管脚	模块管脚	说明
1	Ext_5V	DC+	DC+是传感器电源，范围是 3.3 < 5V
2	GND	DC-	DC-是传感器参考地
3	IO19	IN	输入，用于继电器使能/关闭目标电路
4	—————	NO	继电器常开端
5	—————	COM	继电器公共端，连接目标电路的火线
6	—————	NC	继电器常闭端
7	—————	L/H	继电器上的电平触发选择，需要用跳线帽短接

模块与目标电路连线

继电器模块与目标电路的连线方式：



4.2.2.5.2 模块接口

A 引用模块

```
var relays = require("relays");
```

B 打开模块

```
relays.open(config);
```

功能描述

relays.open根据config配置打开引脚并设置相关属性，成功初始化后返回一个relays对象。

接口约束

无。

参数列表

- config：配置参数，包含三个参数值： pin, work_level, connect_mode。
 - pin：gpio引脚编号，以爱联模组RTL8710开发板为例，可用引脚号为0， 5， 12， 14， 15， 18， 19， 22， 23。0引脚不推荐使用，具体参见gpio模块约束部分。
 - work_level：低电平或高电平触发开关的模式选择。(注， 设置需要和板子实际连线方式相同)
 - relays.LOW：跳线与LOW短接， 低电平触发。
 - relays.HIGH：跳线与HIGH短接， 高电平触发。
 - connect_mode：常开/闭端接法的模式设置。(注， 设置需要和板子实际连线方式相同)
 - relays.NORM_OPEN：常开端的接法。
 - relays.NORM_CLOSED：常闭端的接法。

返回值

打开成功时返回一个relays对象。

接口示例

```
var config = {pin: 5, workLevel: relays.LOW, connectMode: relays.NORM_OPEN};  
/* config需要与实际连线匹配，当前的配置意味着IN→IO5，L/H → L，常开接法 */  
var myrelays = relays.open(config);
```

C 继电器使能目标电路

```
relaysPin.set_work(state);
```

功能描述

通过该接口使能或关闭控制电路。

接口约束

无。

参数列表

- state : 使能或关闭控制电路。
 - relays.OFF : 关闭控制电路；
 - relays.ON : 使能控制电路。

返回值

无。

4.2.2.5.3 约束

无。

4.2.2.5.4 样例

介绍

用于自动控制电路中，它实际上是用较小的电流去控制较大电流的一种“自动开关”。

连线图

参考4.2.2.5.1模块连线，常开电路。

例程

```
var relays = require("relays");  
var config = {pin: 5, workLevel: relays.LOW, connectMode: relays.NORM_OPEN};  
var myrelays = relays.open(config);  
myrelays.set_state(relays.OFF);
```

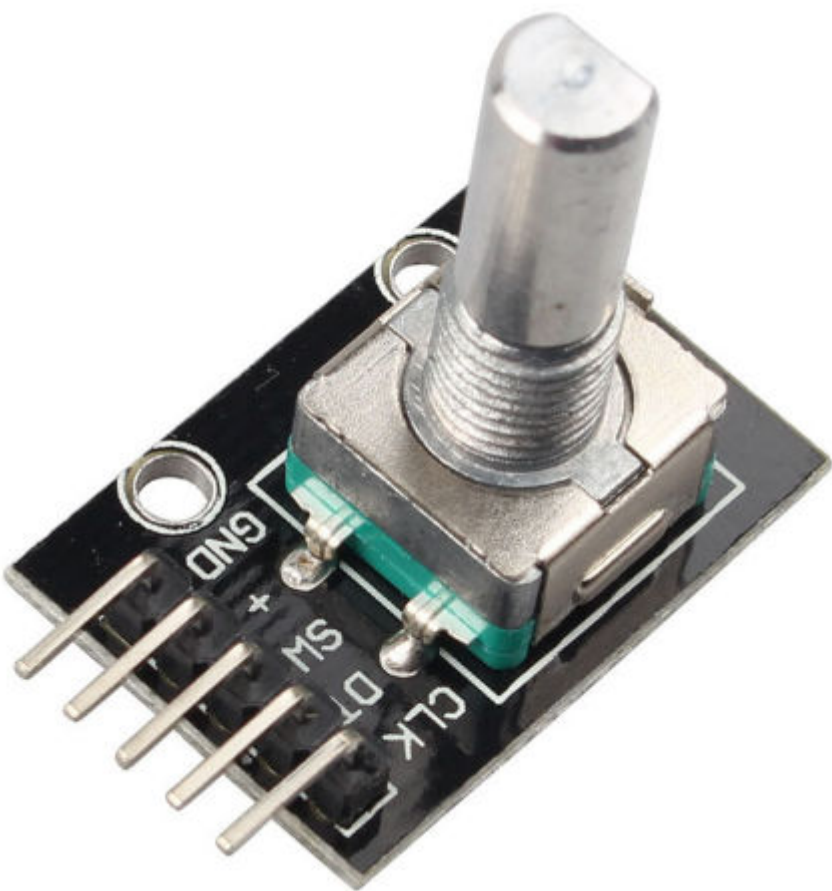
4.2.2.6 旋转编码器（ky-040）

4.2.2.6.1 介绍

旋转编码器是用来测量转速并配合PWM技术可以实现快速调速的装置，光电式旋转编码器通过光电转换，可将输出轴的角位移、角速度等机械量转换成相应的电脉冲以数字量输出（REP）。

A 模块参数

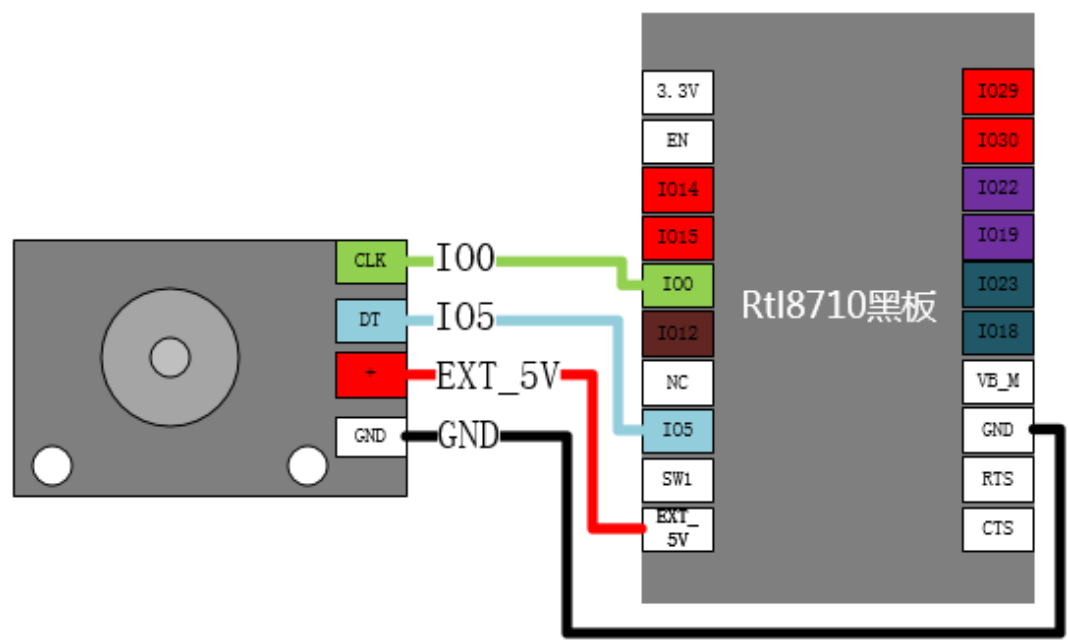
旋转编码器（ky040）可通过旋转可以计数正方向和反方向转动过程中输出脉冲的次数，旋转计数不像电位计，这种转动计数是有限制的。配合旋转编码器上的按键，可以复位到初始状态，即从0开始计数。



旋转编码器（ky040）参数列表:

参数	数值
工作电压:	5V
一圈脉冲数	20

B 模块连线



开发板管脚与模块管脚的连线表：

序号	开发板管脚	模块管脚	说明
1	IO0	CLK	CLK上拉，旋钮逆时针旋转时CLK先输出一段低电平
			旋钮顺时针旋转时CLK后输出一段低电平
2	IO5	DT	DT上拉，旋钮顺时针旋转时DT先输出一段低电平
			旋钮逆时针旋转时DT后输出一段低电平
3	IO12	SW	SW上拉，按键按下时输出下降沿
4	Ext_5V	+	+, 5伏电源输入
5	GND	GND	GND，接地

4.2.2.6.2 模块接口

A 引用模块

```
var ky040 = require('ky040');
```

B 打开模块

```
ky040.open(config)
```

功能描述

根据配置打开旋钮模块获得模块对象。

接口约束

无。

参数列表

config为一个json值，包含CLK、DT和SW，CLK:连接旋钮编码器CLK的引脚编号、DT:连接旋钮编码器DT的引脚编号、SW:连接旋钮编码器SW的引脚编号，此处的引脚编号只能是非负整数；

返回值

返回旋转编码器模块对象；

接口示例

```
var config = {CLK: 0, DT : 5, SW :12};  
var ky040Port = ky040.open(config);
```

C 注册监听事件

```
void ky040Port.on(eventType, func, arg);
```

功能描述

监听旋钮的事件，如旋钮正旋（顺时针）、旋钮逆旋或按键按下，从而调用相应的回调函数。新增回调函数会使旧的回调函数失效。

接口约束

无。

参数列表

- event: 事件类型枚举值，ky040.CW:旋钮顺时针旋转，CWW:旋钮逆时针旋转，BTN_DOWN:按键按下；
- func: 回调函数，当监听事件发生时调用该函数；
- arg: 传递给回调函数的参数，只允许一个参数，如果需要传递多个参数，需要把多个参数打包在一个结构体里。如果没有参数，该接口将默认传递undefined；

返回值

无。

接口示例

```
myEncoder.on(rotary_encoder.CW, function () { print ("编码器正旋");  
});
```

4.2.2.6.3 约束

无。

4.2.2.6.4 样例

```
var ky040 = require('ky040');  
var config = {CLK:0, DT:5, SW:12};
```

```
var ky040Port = ky040.open(config);

ky040Port.on(rotary_encoder.CW, function (num) {
    print (num+"编码器正旋");
}, 1);
ky040Port.on(rotary_encoder.CCW, function (num) {
    print (num+"编码器逆旋");
}, 2);
ky040Port.on(rotary_encoder.BTN_DOWN, function (num) {
    print (num+"编码器按键按下");
}, 3);
```

4.2.3 显示模块接口

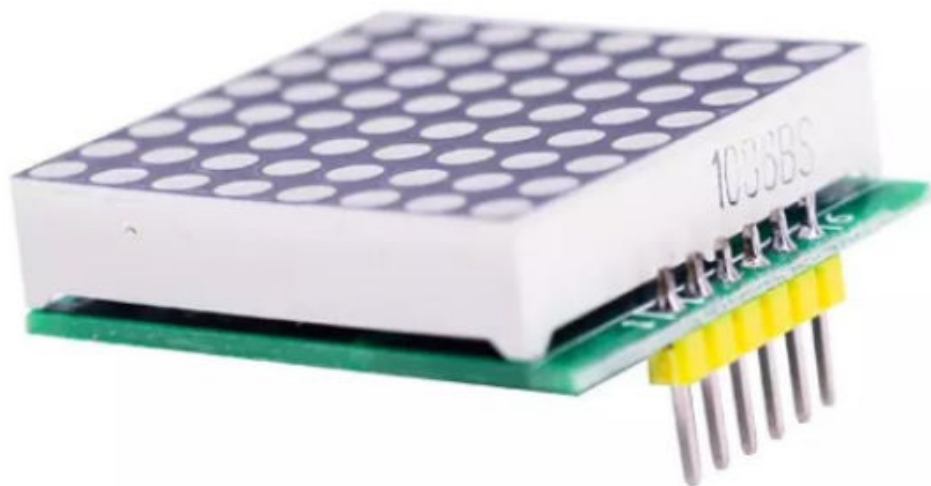
本章主要介绍MapleJS所支持的显示模块接口的使用。其中包括8-8点阵显示屏、LCD1602显示屏、oled显示屏、tft显示屏、墨水屏，以及像素软屏。

4.2.3.1 8-8 点阵显示屏

4.2.3.1.1 介绍

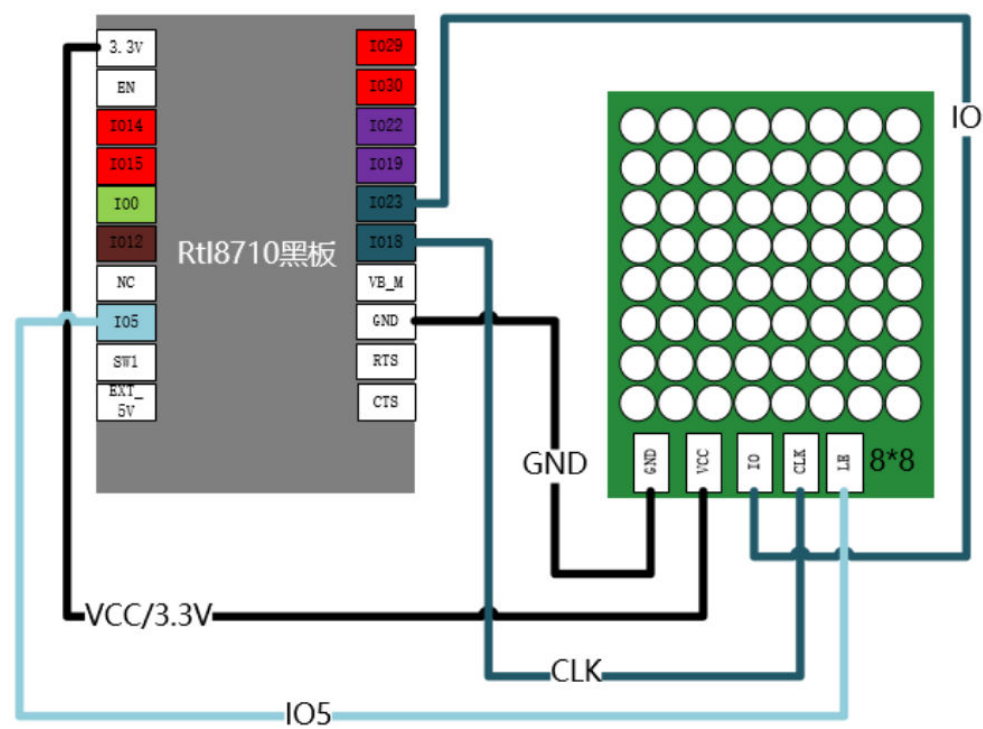
点阵显示屏格式为8乘以8，用来显示8乘以8格式的字符串。

A 模块参数



点阵显示屏我们采用74HC595芯片作为转换驱动芯，采用GPIO口模拟SPI的通信方式进行数据传输，转换芯片将接收到的数据进行处理，对点阵显示屏引脚进行对应操作，从而达到显示信息的目的。

B 模块连线



开发板管脚与模块管脚的连线表：

序号	开发板管脚	模块管脚	说明
1	c	VCC	电源
2	GND	GND	接地
3	IO12	CLK	使用GPIO口模拟SPI的时钟线SCLK
4	IO5	OI	使用GPIO口模拟SPI通信的数据输出线MOSI
5	IO0	LE	使用GPIO口作为输出锁存器的时钟线RCK

4.2.3.1.2 模块接口

A 引用模块

```
var point = require('lattice');
```

B 打开模块

```
point.open();
```

功能描述

点阵显示屏使用GPIO口与开发板相连，因此需要对通信使用的GPIO进行初始化。

接口约束

无。

参数列表

点阵显示屏需要配置使用的引脚为SCLK, MOSI, RCK。

- **SCLK:** 使用GPIO口模拟SPI通信的时钟引脚，以爱联模组RTL8710开发板为例，可用引脚号为0, 5, 12, 14, 15, 18, 19, 22, 23。0引脚不推荐使用，具体参见gpio模块约束部分。
- **MOSI:** 使用GPIO口模拟SPI通信的数据输出引脚，以爱联模组RTL8710开发板为例，可用引脚号为0, 5, 12, 14, 15, 18, 19, 22, 23。0引脚不推荐使用，具体参见gpio模块约束部分。
- **RCK:** 以GPIO口作为输出存储器锁存时钟线，以爱联模组RTL8710开发板为例，可用引脚号为0, 5, 12, 14, 15, 18, 19, 22, 23。0引脚不推荐使用，具体参见gpio模块约束部分。

返回值

点阵显示屏对象。

接口示例

```
var config = {SCLK : 12, MOSI : 5, RCK : 0};  
var lattice = point.open(config);
```

C 输入字符串

```
lattice.write("string", times);
```

功能描述

将传入的ASCII字符串进行逐个显示，每个字符扫描指定次数（使用该功能需要提前将相关字库文件部署至开发板中，文件部署步骤请参考相关说明文档）。

接口约束

无。

参数列表

- **string:** 要进行显示的字符串。
- **times:** 每个字符进行扫描的次数，每次扫描大约为8ms，最大值不应超过4294967295(32位)，且只能为正整数。

返回值

无。

接口示例

```
point.write("huawei", 500);
```

4.2.3.1.3 约束

因点阵显示屏的像素低，无法进行汉字显示，所以只能进行字符显示，显示屏显示内容无法长久保存。

4.2.3.1.4 样例

介绍

将传入的字符串"huawei"进行逐个显示，每个字符扫描指定次数：500。

例程

```
var lattice = require('lattice');  
var config = {SCLK:0, MOSI:5, RCK:12};  
var point = lattice.open(config);  
point.write("huawei", 500);
```

4.2.3.2 LCD1602 显示屏

4.2.3.2.1 介绍

LCD1602是一种工业字符型液晶，能够同时显示16*02即32个字符，支持内置CGROM中的字母、数字、符号等显示，具体可参考下图。

A 模块参数



Upper 4 Bit Hexadecimal													
Lower 4 bits \ Upper 4 bits	0000 (0)	0010 (2)	0011 (3)	0100 (4)	0101 (5)	0110 (6)	0111 (7)	1010 (A)	1011 (B)	1100 (C)	1101 (D)	1110 (E)	1111 (F)
Lower 4 Bit Hexadecimal \ Upper 4 bits	CG RAM (1)		0	1	P	\	P		—	3	3	3	P
xxxx0000 (0)													
xxxx0001 (1)	(2)	!	1	A	Q	a	4	a	7	7	4	3	q
xxxx0010 (2)	(3)	"	2	B	R	b	r	r	イ	ツ	×	P	e
xxxx0011 (3)	(4)	#	3	C	S	c	s	↓	ウ	7	E	e	∞
xxxx0100 (4)	(5)	4	4	D	T	d	t	、	E	ト	ト	μ	e
xxxx0101 (5)	(6)	%	5	E	U	e	u	・	オ	ナ	1	3	U
xxxx0110 (6)	(7)	&	6	F	V	f	v	ヲ	加	ニ	ヨ	P	Σ
xxxx0111 (7)	(8)	'	7	G	W	g	w	ア	キ	ヌ	ウ	g	π
xxxx1000 (8)	(1)	<	8	H	X	h	x	イ	ウ	本	リ	J	又
xxxx1001 (9)	(2)	>	9	I	Y	i	y	ウ	ウ	7	リ	ル	“
xxxx1010 (A)	(3)	*	#	J	Z	j	z	エ	コ	ハ	レ	j	キ
xxxx1011 (B)	(4)	+	8	K	E	k	く	オ	サ	ヒ	ロ	”	天
xxxx1100 (C)	(5)	、	<	L	7	l	l	ト	シ	フ	ワ	Φ	天
xxxx1101 (D)	(6)	—	=	M	I	m	>	ユ	ズ	へ	ン	も	÷
xxxx1110 (E)	(7)	、	>	N	^	n	→	ヨ	セ	ホ	°	天	
xxxx1111 (F)	(8)	/	?	O	_	o	+	ウ	リ	マ	°	も	天

本模块需加装PCF8574转接板，通过I2C通信协议来驱动屏幕。

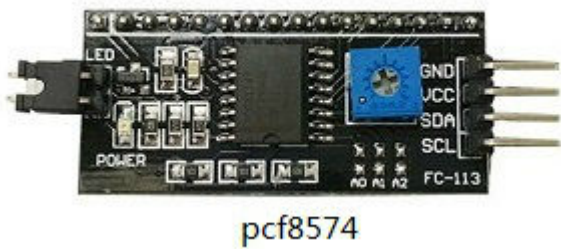
LCD1602参数列表：

参数	数值
显示容量	16×2个字符
芯片工作电压	4.5—5.5V
工作电流	2.0mA(5.0V)
块最佳工作电压	5.0V

参数	数值
字符尺寸	2.95×4.35(W×H)mm

PCF8574是CMOS电路，通过两条双向总线I2C可使大多数MCU实现远程I/O口扩展。该器件包含一个8位准双向口和一个I2C总线接口。

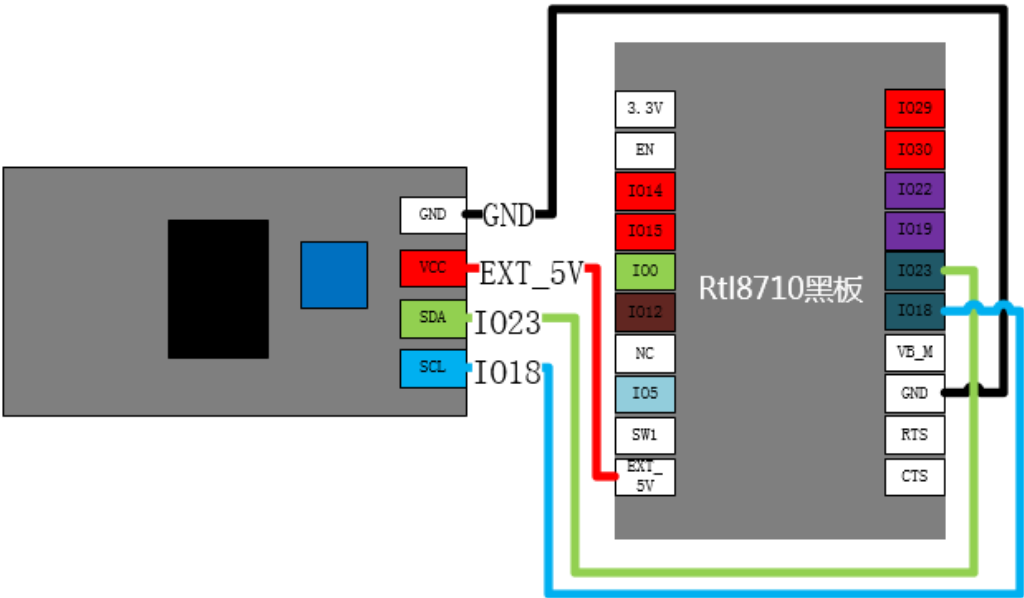
PCF8574功耗较低，输出锁存，具有大电流驱动能力，可直接驱动LED。此外它还拥有一条中断接线(INT)可MCU的中断逻辑相连，通过INT发送中断信号，远端I/O口不必经过总线通信便可通知MCU是否有数据从端口输入。这使得PCF8574可以作为一个单被控器。



PCF8574T参数列表:

参数	数值
工作温度范围	-40°C to +85°C
操作电压	2.5—6.0V
低备用电流	≤10μA
时钟频率	0.1MHz

B 模块连线



开发板管脚与模块管脚的连线表：

序号	开发板管脚	模块管脚	说明
1	Ext_5V	VCC	电源
2	GND	GND	接地
3	IO23/I2C1_SDA	SDA	SDA，即I2C_SDA，是I2C数据线
4	IO18/I2C1_SCL	SCL	SCL，即I2C_SCL，是I2C时钟线

4.2.3.2.2 模块接口

A 引用模块

```
var lcd1602 = require('lcd1602');
```

B 打开模块

```
lcd.open(i2c_num);
```

功能描述

打开指定i2c引脚上的lcd1602端口。

接口约束

无。

参数列表

- i2c_num:number类型，开发板的i2c编号，可为0或者1，及其他正整数。

返回值

返回lcd1602对象。

接口示例

```
var lcd = lcd1602.open(0);
```

C 显示字符串

```
void lcdPort.fillText(x, y, string);
```

功能描述

在指定位置显示5*8格式字符串，需要注意1602屏幕只能显示16*2个字符，若显示过多字符将忽略超出一行边界的字符。

接口约束

无。

参数列表

- x:number类型，表示显示字符串起始位置的水平坐标，只能0~15中任意一个整数。
- y:number类型，表示显示字符串起始位置的垂直坐标，只能0~1中任意一个整数。
- string:要显示的字符串。

返回值

无。

接口示例

```
lcd.fillText(0, 0, 'hello');
```

D 清屏

```
void lcdPort.clear(void);
```

功能描述

对显示屏进行清屏处理。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
lcd.clear();
```

E 控制屏幕背光

```
void lcdPort.backlight(switch);
```

功能描述

打开或关闭显示屏背光。

接口约束

无。

参数列表

- switch:lcd1602开关枚举值
 - lcd1602.ON : 开
 - lcd1602.OFF : 关

返回值

无。

接口示例

```
lcd.backlight(lcd1602.ON);
```

F 关闭模块

```
lcdPort.close(void);
```

功能描述

关闭LCD1602模块。

参数列表

无。

返回值

无。

4.2.3.2.3 约束

无。

4.2.3.2.4 样例

介绍

使用LCD1602显示屏进行字符串开背光显示。

例程

```
var lcd = require('lcd1602');
var time = require('timer');
var lcdPort= lcd.open(0);
lcdPort.backlight(lcd.ON);
lcdPort.fillText(0, 1, 'Hello');
time.setDelay(3000);
lcdPort.fillText(1, 9, 'world!');
time.setDelay(3000);
lcdPort.clear();
time.setDelay(3000);
lcdPort.close();
```

4.2.3.3 oled 显示屏

4.2.3.3.1 介绍

A 模块参数

canvas module是AntJS系统基于不同屏幕的专用画图子模块，其接口与HTML5的canvas API保持兼容。本文是canvas module在oled屏上具体实现的指导说明。

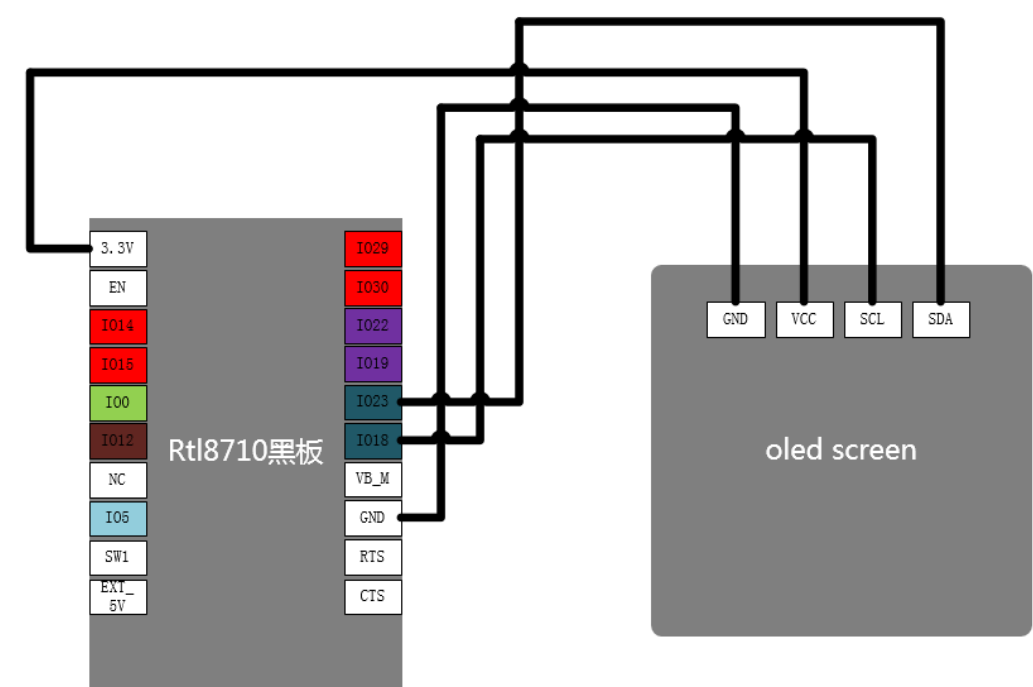
oled screen具有功耗低、视角宽、响应速度快显示等优点，经常被用于货架标签、工业仪表等显示应用。



oled screen模块参数列表：

参数	数值
尺寸	0.96寸
分辨率	128*64
可视角度	>160°
功耗	0.04W
输入电压	3.3~5VDC
通信方式	IIC
字库支持	无

B 模块连线



序号	开发板管脚	模块管脚	说明
1	GND	GND	GND是模块的接地引脚
2	3.3V	VCC	VCC是模块的电源引脚
3	SCL	IO18	SCL是模块的IIC时钟线
3	SDA	IO23	SDA是模块的IIC数据线

4.2.3.3.2 模块接口

A 引用模块

```
var canvas = require("canvas");
```

B 打开模块

```
canvas.open(cfg);
```

功能描述

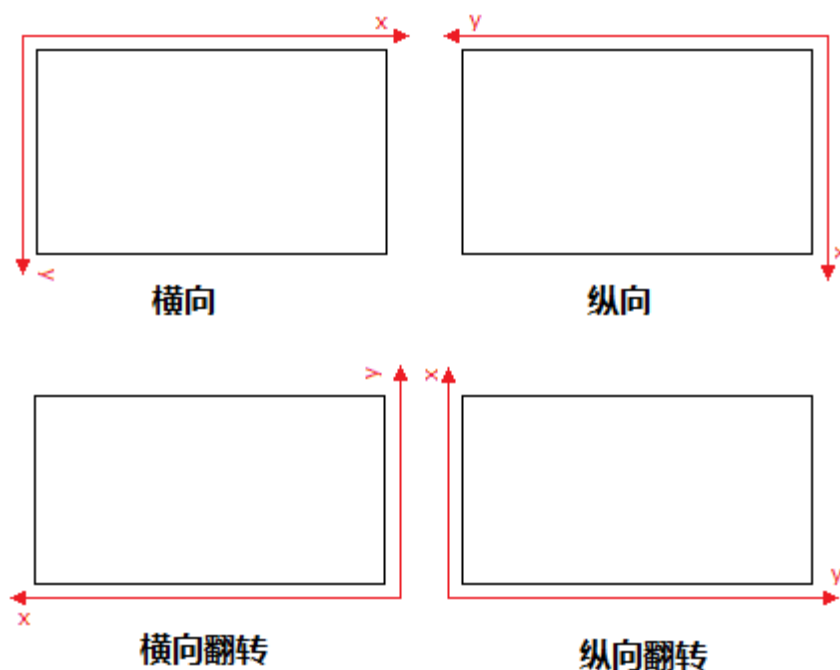
- 按照cfg的配置打开模块。

接口约束

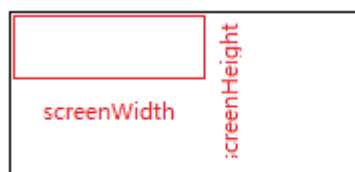
无。

参数列表

- `cfg`表示打开的配置参数，其中应包含：
 - `screenType`信息。`screenType`表示屏幕类型：
 - 字符串“`ink`”表示ink屏；
 - 字符串“`oled`”表示oled屏；
 - 字符串“`tft`”表示tft屏；
 - `screenType`必须指定，不可省略。
 - `orientation`信息。`orientation`表示显示方向：
 - 取值选项如下：
 - 1：横向
 - 2：纵向
 - 3：横向翻转
 - 4：纵向翻转



- `orientation`必须显示指定，不可省略。
- `screenWidth`信息。`screenWidth`表示自定义屏幕宽度：
 - 范围： $1 \leq \text{screenWidth} \leq 128$ 。
 - `screenWidth`可省略，省略时`screenWidth`将默认设置为128。
- `screenHeight`信息。`screenHeight`表示自定义屏幕高度：
 - 范围： $1 \leq \text{screenHeight} \leq 64$ 。
 - `screenHeight`可省略，省略时`screenHeight`将默认设置为64。



以横向为例

- canvasPort为方法返回的端口句柄。

返回值

无。

接口示例

```
/* 显示指定自定义屏幕宽高 */  
var canvas = require("canvas");  
var cfg = {screenType:"oled", screenWidth:64, screenHeight:32, orientation:1};  
var canvas_port = canvas.open(cfg);  
  
/* 省略自定义屏幕宽高 */  
var canvas = require("canvas");  
var cfg = {screenType:"oled",orientation:1};  
var canvas_port = canvas.open(cfg);
```

C 新建路径

```
canvas_port.beginPath();
```

功能描述

新建一条路径，路径一旦创建成功，图形绘制命令被指向到路径上生成路径。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
canvas_port.beginPath();
```

D 关闭路径

```
canvas_port.closePath();
```

功能描述

关闭路径，之后图形绘制命令又重新指向到上下文中。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
canvas_port.closePath();
```

E 设置画笔起点


```
canvas_port.moveTo(x, y);
```

功能描述

设置画笔的起始点坐标。

接口约束

无。

参数列表

- x为起始点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为起始点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.moveto(0, 0);
```

F 绘制线段

```
canvas_port.lineTo(x, y);
```

功能描述

绘制线段。

接口约束

无。

参数列表

- x为终点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为终点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.lineto(50, 50);
```

G 绘制矩形

```
canvas_port.strokeRect(x1, y1, x2, y2);
```

功能描述

绘制矩形。

接口约束

无。

参数列表

- (x1,y1)和(x2,y2)分别是矩形对角顶点的坐标。
- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.strokeRect(0, 0, 29, 29);
```

H 填充矩形

```
canvas_port.fillRect(x1, y1, x2, y2);
```

功能描述

填充矩形。

接口约束

无。

参数列表

- (x1,y1)和(x2,y2)分别是矩形对角顶点的坐标。
- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.fillRect(0, 0, 29, 29);
```

I 绘制三角形

```
canvas_port.strokeTriangle(x1, y1, x2, y2, x3, y3);
```

功能描述

绘制三角形。

接口约束

无。

参数列表

- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。
- x3为顶点3横坐标：
 - 范围： $0 \leq x3 \leq (\text{screenWidth}-1)$ 。
- y3为顶点3纵坐标：
 - 范围： $0 \leq y3 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.strokeTriangle(10, 10, 20, 20, 30, 30);
```

J 填充三角形

```
canvas_port.fillTriangle(x1, y1, x2, y2, x3, y3);
```

功能描述

填充三角形。

接口约束

无。

参数列表

- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。
- x3为顶点3横坐标：
 - 范围： $0 \leq x3 \leq (\text{screenWidth}-1)$ 。

- y3为顶点3纵坐标:
 - 范围: $0 \leq y3 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.fillTriangle(10, 10, 20, 20, 30, 30);
```

K 绘制多边形

```
canvas_port.strokePolygon(verticesArray);
```

功能描述

绘制多边形。

接口约束

无。

参数列表

- verticesArray为多边形顶点坐标array。
 - 顶点的横坐标范围 $0 \leq x \leq (\text{screenWidth}-1)$ 。
 - 顶点的纵坐标范围 $0 \leq y \leq (\text{screenHeight}-1)$ 。
 - 顶点个数 $3 \leq n \leq 10$

返回值

无。

接口示例

```
var verticesArray1 = [[10, 20], [20, 10], [40, 10], [30, 20]];
canvas_port.strokePolygon(verticesArray1);
```

L 填充多边形

```
canvas_port.fillPolygon(verticesArray);
```

功能描述

填充多边形。

接口约束

无。

参数列表

- verticesArray为多边形顶点坐标array。
 - 顶点的横坐标范围 $0 \leq x \leq (\text{screenWidth}-1)$ 。
 - 顶点的纵坐标范围 $0 \leq y \leq (\text{screenHeight}-1)$ 。
 - 顶点个数 $3 \leq n \leq 10$

返回值

无。

接口示例

```
var verticesArray1 = [[10, 20], [20, 10], [40, 10], [30, 20]];
canvas_port.fillPolygon(verticesArray1);
```

M 绘制圆弧

```
canvas_port.arc(x, y, radius, start_angle, end_angle, anti_clockwise);
```

功能描述

绘制圆弧。

接口约束

无。

参数列表

- x为圆心横坐标：
 - 范围： $0 \leq x \leq 128$ 。
- y为圆心纵坐标：
 - 范围： $0 \leq y \leq 64$ 。
- radius为圆弧半径：
 - 范围： $1 \leq radius \leq 64$ 。
- start_angle为起始角度：
 - 范围： $0 \leq start_angle \leq 360$ 。
- end_angle为终止角度：
 - 范围： $0 \leq end_angle \leq 360$ 。
- anti_clockwise为圆弧方向：
- 0表示顺时针，1表示逆时针。

返回值

无。

接口示例

```
canvas_port.arc(50, 30, 10, 0, 135, 0);
```

N 填充圆形

```
canvas_port.fillCircle(x, y, radius);
```

功能描述

填充圆形。

接口约束

无。

参数列表

- x为圆心横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为圆心纵坐标：

- 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。
- radius为圆弧半径：
 - 范围： $1 \leq \text{radius} \leq 200$ 。

返回值

无。

接口示例

```
canvas_port.fillCircle(60, 30, 25);
```

O 绘制椭圆

```
canvas_port.strokeEllipse(x1, x2, y1, y2);
```

功能描述

绘制椭圆。

接口约束

无。

参数列表

- x1为最左侧点横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- x2为最右侧点横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y1为最上侧点纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- y2为最下侧点纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.strokeEllipse(10, 40, 5, 60);
```

P 填充椭圆

```
canvas_port.fillEllipse(x1, x2, y1, y2);
```

功能描述

填充椭圆。

接口约束

无。

参数列表

- x1为最左侧点横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。

- x2为最右侧点横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y1为最上侧点纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- y2为最下侧点纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.fillEllipse(10, 40, 5, 60);
```

Q 显示字符

```
canvas_port.fillText(x, y, "textString", "fontString");
```

功能描述

显示字符。（使用该功能需要提前将相关字库文件部署至开发板中，文件部署步骤请参考相关说明文档。）

接口约束

无。

参数列表

- x为显示起始点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为显示起始点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。
- "textString"为待显示字符串。
- "fontString"为字体字符串：
 - "0816"表示英文字体；
 - "1616"表示中文字体。

返回值

无。

接口示例

```
canvas_port.fillText(0, 0, "Hello World!", "0816");  
canvas_port.fillText(0, 20, "一丁", "1616");
```

R 显示图片

```
canvas_port.drawImage("imageNameString", x, y);
```

功能描述

显示图片。（使用该功能需要提前将相关图片文件部署至开发板中，文件部署步骤请参考相关说明文档。）

接口约束

无。

参数列表

- "imageNameString"为图片文件名字符串。
 - "imageNameString"不可包含文件后缀名，例如部署的图片文件为test.pin则填入"test"。
- x为显示起始点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为显示起始点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.drawImage("test", 0, 0);
```

S 显示轮廓

```
canvas_port.stroke();
```

功能描述

显示轮廓。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
canvas_port.stroke();
```

T 清除屏幕

```
canvas_port.clear();
```

功能描述

清除屏幕。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
canvas_port.clear();
```

U 设置线宽属性

```
canvas_port.lineWidth = settingValue;
```

功能描述

设置线宽属性。

接口约束

无。

参数列表

- settingValue为所设置的线宽值。
 - 范围： $1 \leq \text{settingValue} \leq 8$
 - 线宽值可省略，省略时线宽值将默认为1。

返回值

无。

接口示例

```
canvas_port.lineWidth = 2;
```

V 设置画笔颜色属性

```
canvas_port.strokeStyle = settingValue;
```

功能描述

设置画笔颜色属性。

接口约束

无。

参数列表

- settingValue为所设置的画笔颜色属性。
 - 取值： 0x000000表示画笔为黑色， 0xFFFFFFFF表示画笔为白色。
 - 画笔颜色属性可省略，省略时画笔颜色属性值将默认为0x000000（黑色）。

返回值

无。

接口示例

```
canvas_port.strokeStyle = 0x000000;
```

4.2.3.3.3 约束

- 模块接口中所涉及到的数值的属性必须是不小于零的整数，浮点数、负数等均为非法。
- 在显式指定了screenWidth及screenHeight的情况下，Canvas模块将只对指定范围内的画笔内容进行处理显示并自动忽略超出范围部分。

- screenWidth和screenHeight的设定值是相对于lineWidth设为1而言的，当lineWidth被设置为其他合法数值时，screenWidth和screenHeight所实际覆盖的范围会相应缩小，用户需自行计算并确保所绘制图形在其覆盖范围之内。

4.2.3.3.4 样例

样例A

介绍

orientation对比。

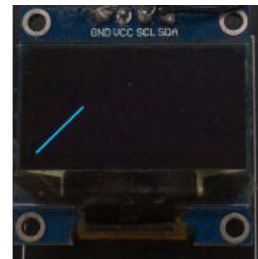
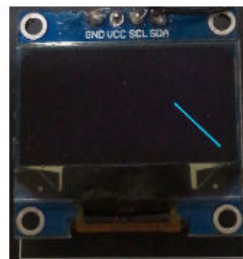
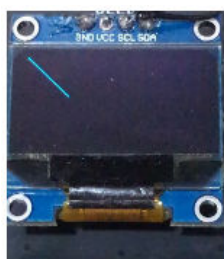
样例代码

```
canvas = require("canvas");
cfg = {screenType:"oled",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();
canvas_port.lineWidth = 4;
canvas_port.moveTo(0, 0);
canvas_port.lineTo(30, 30);
canvas_port.stroke();
canvas_port.closePath();
```

然后依次将代码模板中的cfg修改为：

```
cfg = {screenType:"oled",orientation:2};
cfg = {screenType:"oled",orientation:3};
cfg = {screenType:"oled",orientation:4};
```

样例结果



orientation:1 orientation:2 orientation:3 orientation:4

样例B

介绍

绘制线段。

样例代码

```
canvas = require("canvas");
cfg = {screenType:"oled",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.lineWidth = 1;
canvas_port.moveTo(0, 5);
canvas_port.lineTo(50, 5);

canvas_port.lineWidth = 2;
canvas_port.moveTo(0, 10);
canvas_port.lineTo(50, 10);
```

```
canvas_port.lineWidth = 4;
canvas_port.moveTo(0, 20);
canvas_port.lineTo(50, 20);

canvas_port.lineWidth = 8;
canvas_port.moveTo(0, 30);
canvas_port.lineTo(50, 30);

canvas_port.stroke();
canvas_port.closePath();
```

代码中可通过设置`canvas_port.lineWidth`来调整线宽，线宽的取值范围为 $1 \leq \text{lineWidth} \leq 8$ 。

样例C

介绍

绘制矩形。

样例代码

```
canvas = require("canvas");
cfg = {screenType:"oled",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.lineWidth = 1;
canvas_port.strokeRect(0, 0, 29, 29);

canvas_port.lineWidth = 2;
canvas_port.strokeRect(40, 0, 69, 29);

canvas_port.lineWidth = 4;
canvas_port.strokeRect(80, 0, 109, 29);

canvas_port.stroke();
canvas_port.closePath();
```

代码中可通过设置`canvas_port.lineWidth`来调整线宽，线宽的取值范围为 $1 \leq \text{lineWidth} \leq 8$ 。

样例D

介绍

填充矩形。

样例代码

```
canvas = require("canvas");
cfg = {screenType:"oled",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.fillRect(0, 0, 29, 29);
canvas_port.fillRect(40, 0, 69, 29);
canvas_port.fillRect(80, 0, 109, 29);

canvas_port.stroke();
canvas_port.closePath();
```

样例E

介绍

绘制圆弧。

样例代码

```
canvas = require("canvas");
cfg = {screenType:"oled",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.lineWidth = 1;
canvas_port.arc(50, 30, 10, 0, 135, 0);

canvas_port.lineWidth = 2;
canvas_port.arc(50, 30, 20, 0, 135, 0);

canvas_port.lineWidth = 4;
canvas_port.arc(50, 30, 30, 0, 135, 0);
canvas_port.stroke();
canvas_port.closePath();
```

代码中可通过设置`canvas_port.lineWidth`来调整线宽，线宽的取值范围为 $1 \leq \text{lineWidth} \leq 8$ 。

样例F

介绍

绘制圆形。

样例代码

```
canvas = require("canvas");
cfg = {screenType:"oled",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.lineWidth = 1;
canvas_port.arc(20, 30, 20, 0, 360, 0);
canvas_port.lineWidth = 2;
canvas_port.arc(60, 30, 20, 0, 360, 0);
canvas_port.lineWidth = 4;
canvas_port.arc(100, 30, 20, 0, 360, 0);

canvas_port.stroke();
canvas_port.closePath();
```

代码中可通过设置`canvas_port.lineWidth`来调整线宽，线宽的取值范围为 $1 \leq \text{lineWidth} \leq 8$ 。

样例G

介绍

填充圆形。

样例代码

```
canvas = require("canvas");
cfg = {screenType:"oled",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.fillCircle(60, 30, 25);

canvas_port.stroke();
canvas_port.closePath();
```

样例H

介绍

显示字符。

样例代码

```
canvas = require("canvas");
cfg = {screenType:"oled",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.fillText(0, 0, "Hello World!", "0816");
canvas_port.fillText(0, 20, "一丁", "1616");

canvas_port.stroke();
canvas_port.closePath();
```

样例I

介绍

显示图片

样例代码

```
canvas = require("canvas");
cfg = {screenType:"oled",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.drawImage("smile", 0, 0);

canvas_port.stroke();
canvas_port.closePath();
```

样例J

介绍

通过设置画笔颜色实现反显。

样例代码

```
canvas = require("canvas");
cfg = {screenType:"oled",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.strokeStyle = 0x000000;
canvas_port.fillCircle(60, 30, 25);

canvas_port.strokeStyle = 0xFFFFFF;
canvas_port.fillCircle(60, 30, 15);

canvas_port.stroke();
canvas_port.closePath();
```

样例K

介绍

清除屏幕。

样例代码

```
canvas = require("canvas");
cfg = {screenType:"oled",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();
```

```
canvas_port.arc(30, 30, 20, 0, 360, 0);
canvas_port.clear();

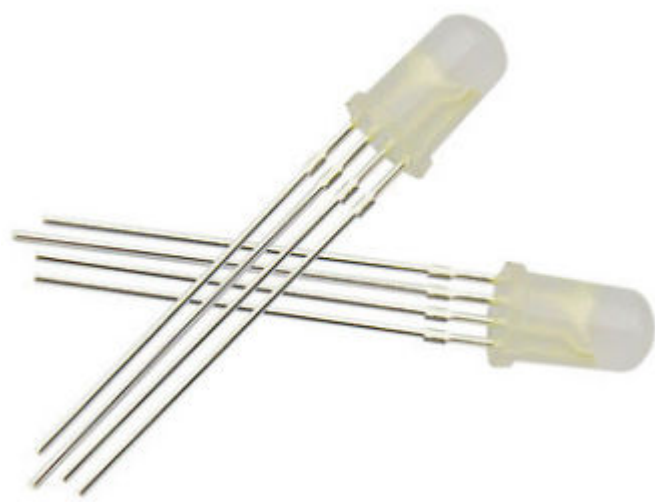
canvas_port.stroke();
canvas_port.closePath();
```

4.2.3.4 三色灯

4.2.3.4.1 介绍

A 模块参数

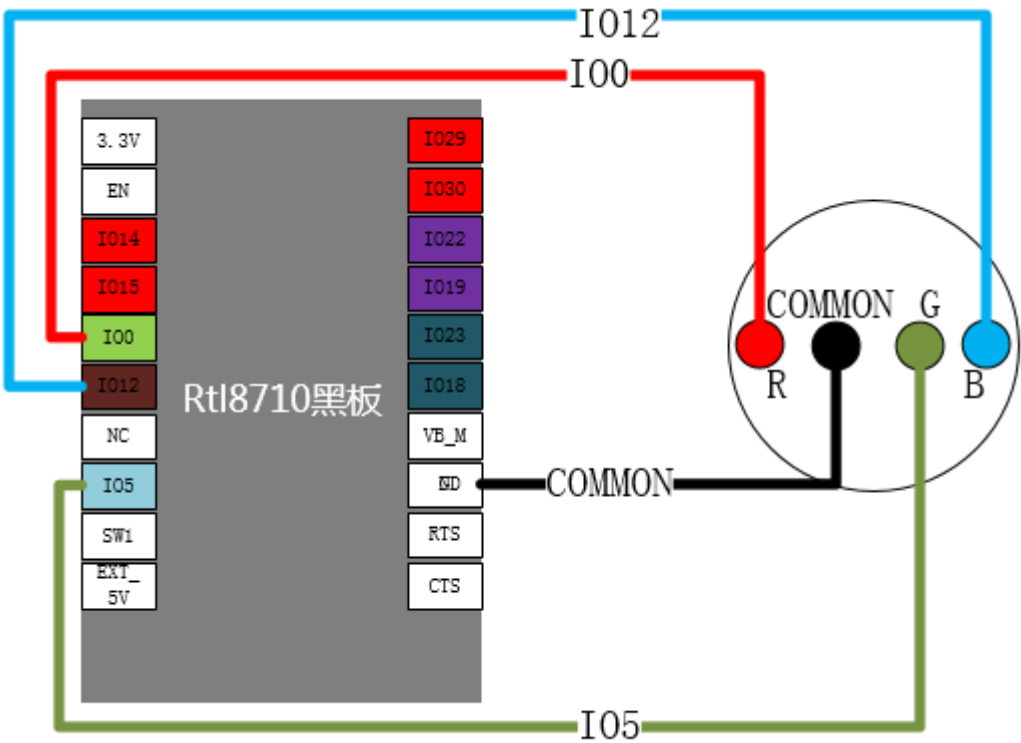
RGB三色LED由三个不同颜色的管芯组成，有共阳、共阴接法 三色LED模块可以通过控制通过PWM脉宽调制实现色彩渐变 realtek有效的PWM管脚为PWM0:IO14， PWM1:IO15， PWM2:IO0， PWM3:IO12， PWM4:IO5 本模块所有工作模式以周期为8ms的pwm脉冲驱动。



三色LED参数列表:

参数	数值
红色波长	1000-1200MCD
绿色波长	3000-5000MCD
蓝色波长	2000-3000MCD
红色电源	1.8-2.0
绿色电源	3.2-3.4
蓝色电源	3.2-3.4

B 模块连线



开发板管脚与模块管脚的连线表：

序号	开发板管脚	模块管脚	说明
1	IO14/PWM0	R	红色管脚
2	GND/VCC	COMMON	公共管脚，共阴或共阳
3	IO15/PWM1	G	绿色管脚
4	IO0/PWM2	B	蓝色管脚

4.2.3.4.2 模块接口

A 引用模块

```
var ledrgb = require('ledrgb');
```

B 打开模块

```
ledrgb.open(pin_r, pin_g, pin_b, type);
```

功能描述

根据配置参数打开三色LED端口。

接口约束

无。

参数列表

- `pin_r`、`pin_g`、`pin_b`: `number`类型，是红、绿、蓝连接的pwm引脚编号，如0、5、12，只能是非负整数，默认为0、5、12;
- `type`: 三色LED器件极性(共阴、共阳)枚举值，`ledrgb.TYPE_CC`: 共阴、`ledrgb.TYPE_CA`: 共阳，默认为共阴类型;

返回值

返回三色LED接口对象;

接口示例

```
var rgb = ledrgb.open(0, 5, 12, rgb.TYPE_CA);
```

C 长亮模式

```
void rgbPin.keep( RGB );
```

功能描述

指定颜色点亮三色灯并保持长亮。

接口约束

无。

参数列表

RGB: RGB配色值,`number`类型,(0x000000~0xFFFFFFFF);

返回值

无。

接口示例

```
rgb.keep(0xFFFFFFFF);
```

D 渐变模式

```
void rgbPin.change(speed);
```

功能描述

红绿蓝循环渐变，需要配合`timer`模块使用，使用时需注意定时器周期需大于渐变周期，否则将导致事件队列溢出，建议先给较大的定时器周期，估算出渐变周期，再对定时器周期做相应调整。这里的定时器周期指的是`timer`模块接口的入参周期值，渐变周期指的是一种颜色全亮到下一种颜色全亮所需的时间。

接口约束

无。

参数列表

`speed`: 渐变速度，`number`类型，分10档，[1-10]，数值越大渐变越快;

返回值

无

接口示例

```
rgbPin.change(10);
```

E 呼吸灯模式


```
void rgbPin.breath(speed, RGB);
```

功能描述

以固定颜色进行周期性呼吸，需要配合timer模块使用，使用时需注意定时器周期需大于呼吸周期，否则将导致事件队列溢出，建议先给较大的定时器周期，估算出呼吸周期，再对定时器周期做相应调整。这里的定时器周期指的是timer模块接口的入参周期值，呼吸周期指的是一次呼吸所需的时间，也就是呼吸灯从灭到全亮再到全灭所需的时间。

接口约束

无。

参数列表

- speed: 呼吸速度，number类型，分10档[1-10]，数值越大,呼吸越快;
- RGB: RGB配色值(0x000000~0xFFFFFF);-

返回值

无。

接口示例

```
rgbPin.breath(5, 0xAABBCC);
```

F 关闭三色LED

```
void rgbPin.close(void);
```

功能描述

熄灭LED并关闭三色LED端口。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
rgb.close();
```

4.2.3.4.3 约束

无。

4.2.3.4.4 样例

样例A

介绍

打开共阴三色LED，设置rgb三色引脚分别为IO14、IO15、IO0，保持白色显示。

例程

```
var rgb = require('ledrgb');
var rgbPin = rgb.open(14, 15, 0, rgb.TYPE_CC);
rgbPin.keep(0xFFFFFFFF);
```

样例B

介绍

打开共阳三色LED，设置rgb三色引脚分别为IO0，IO5，IO12，运行循环渐变模式，渐变速度为最快速度10档，周期3秒。

例程

```
var rgb = require('ledrgb');
var time = require('timer');
var rgbPin = rgb.open(0, 5, 12, rgb.TYPE_CA);
var timeID = time.setInterval (function () {
    rgbPin.change(10);
}, 3000);
```

样例C

介绍

打开共阴三色LED，设置rgb三色引脚分别为IO0，IO5，IO12，运行呼吸灯模式，颜色为红色，呼吸速度为5档，周期5秒。

例程

```
var rgb = require('ledrgb');
var time = require('timer');
var rgbPin = rgb.open(0, 5, 12, rgb.TYPE_CC);
var timeID = time.setInterval (function () {
    rgbPin.breath(5, 0xFF0000);
}, 5000);
```

4.2.3.5 tft 显示屏

4.2.3.5.1 介绍

A 模块参数

canvas module是AntJS系统基于不同屏幕的专用画图子模块，其接口与HTML5的canvas API保持兼容。本文是canvas module在tft屏上具体实现的指导说明。

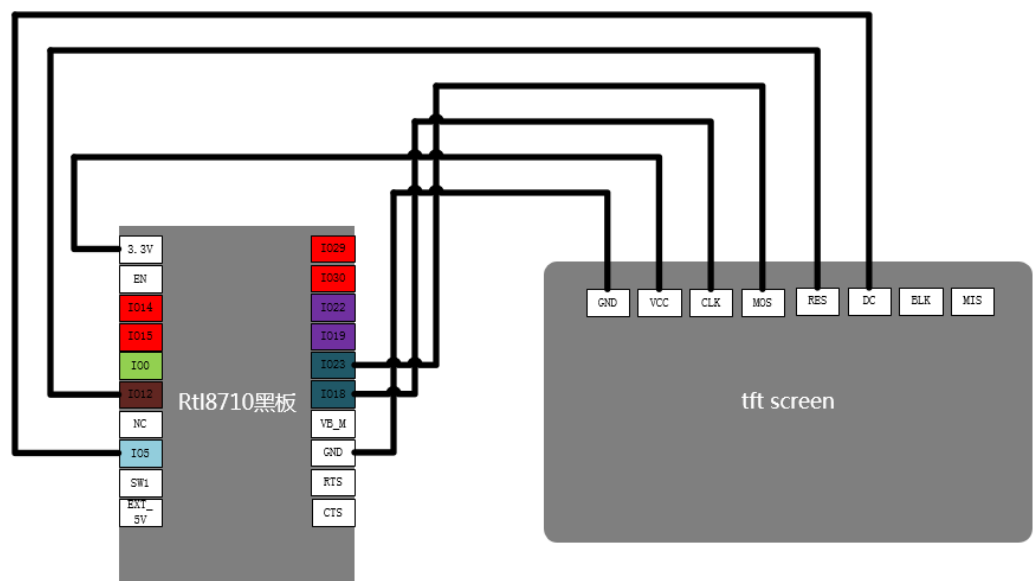
tft screen具有功耗低，可视角度大等优点，常用于电子产品等显示应用。

tft screen模块参数列表：

参数	数值
尺寸	3.5寸
控制芯片	ILI9486
分辨率	320 × 480
显示区域	48.96mm × 73.44mm
接口类型	SPI
显示颜色	全彩色

像素点大小	0.153mm × 0.153mm
工作温度	-20 ~ 70℃
工作电压	3.3VDC

B 模块连线



序号	开发板管脚	模块管脚	说明
1	GND	GND	GND是模块的接地引脚
2	3.3V	VCC	VCC是模块的电源引脚
3	IO18	CLK	CLK是模块的SPI通信SCK引脚
4	IO23	MOS	MOS是模块的SPI通信MOSI引脚
5	IO12	RES	RES是模块的复位
6	IO5	DC	DC是模块的寄存器/数据选择引脚

4.2.3.5.2 模块接口

A 引用模块

```
var canvas = require("canvas");
```

B 打开模块

```
canvas.open(cfg);
```

功能描述

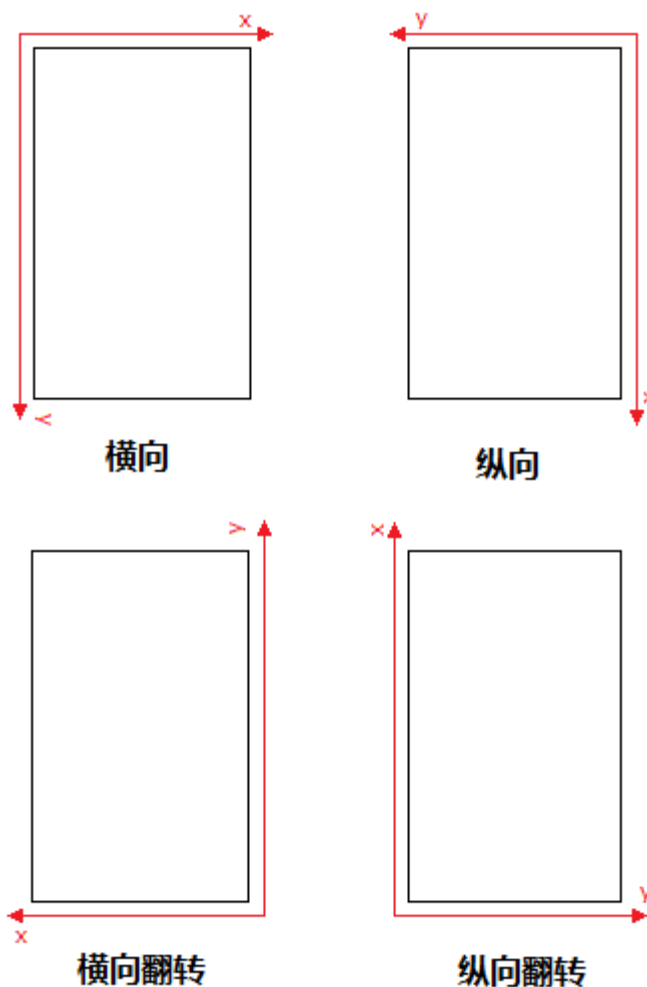
- 按照cfg的配置打开模块。

接口约束

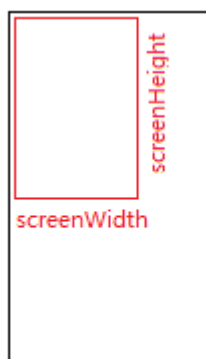
无。

参数列表

- cfg表示打开的配置参数，其中应包含：
 - screenType信息。screenType表示屏幕类型：
 - 字符串“ink”表示ink屏；
 - 字符串“oled”表示oled屏；
 - 字符串“tft”表示tft屏；
 - screenType必须指定，不可省略。
 - orientation信息。orientation表示显示方向：
 - 取值选项如下：
 - 1：横向
 - 2：纵向
 - 3：横向翻转
 - 4：纵向翻转



- orientation必须显示指定，不可省略。
- screenWidth信息。screenWidth表示自定义屏幕宽度：
 - 范围： $1 \leq \text{screenWidth} \leq 320$ 。
 - screenWidth可省略，省略时screenWidth将默认设置为320。
- screenHeight信息。screenHeight表示自定义屏幕高度：
 - 范围： $1 \leq \text{screenHeight} \leq 480$ 。
 - screenHeight可省略，省略时screenHeight将默认设置为480。



以横向为例

- canvasPort为方法返回的端口句柄。

返回值

无。

接口示例

```
/* 显示指定自定义屏幕宽高 */
var canvas = require("canvas");
var cfg = {screenType:"tft", screenWidth:64, screenHeight:32, orientation:1};
var canvas_port = canvas.open(cfg);

/* 省略自定义屏幕宽高 */
var canvas = require("canvas");
var cfg = {screenType:"tft", orientation:1};
var canvas_port = canvas.open(cfg);
```

C 新建路径

```
canvas_port.beginPath();
```

功能描述

新建一条路径，路径一旦创建成功，图形绘制命令被指向到路径上生成路径。

接口约束

无。

参数列表

无。

接口示例

```
canvas_port.beginPath();
```

D 关闭路径

```
canvas_port.closePath();
```

功能描述

关闭路径，之后图形绘制命令又重新指向到上下文中。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
canvas_port.closePath();
```

E 设置画笔起点

```
canvas_port.moveTo(x, y);
```

功能描述

设置画笔起始点坐标。

接口约束

无。

参数列表

- x为起始点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为起始点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.moveTo(0, 0);
```

F 绘制线段

```
canvas_port.lineTo(x, y);
```

功能描述

绘制线段。

接口约束

无。

参数列表

- x为终点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为终点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.lineto(100, 100);
```

G 绘制矩形

```
canvas_port.strokeRect(x1, y1, x2, y2);
```

功能描述

绘制矩形。

接口约束

无。

参数列表

- (x1,y1)和(x2,y2)分别是矩形对角顶点的坐标。
- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.strokeRect(0, 0, 29, 29);
```

H 填充矩形

```
canvas_port.fillRect(x1, y1, x2, y2);
```

功能描述

填充矩形。

接口约束

无。

参数列表

- (x1,y1)和(x2,y2)分别是矩形对角顶点的坐标。
- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.fillRect(0, 0, 29, 29);
```

I 绘制三角形

```
canvas_port.strokeTriangle(x1, y1, x2, y2, x3, y3);
```

功能描述

绘制三角形。

接口约束

无。

参数列表

- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。
- x3为顶点3横坐标：
 - 范围： $0 \leq x3 \leq (\text{screenWidth}-1)$ 。
- y3为顶点3纵坐标：
 - 范围： $0 \leq y3 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.strokeTriangle(10, 10, 20, 20, 30, 30);
```

J 填充三角形

```
canvas_port.fillTriangle(x1, y1, x2, y2, x3, y3);
```

功能描述

填充三角形。

接口约束

无。

参数列表*

- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。
- x3为顶点3横坐标：
 - 范围： $0 \leq x3 \leq (\text{screenWidth}-1)$ 。
- y3为顶点3纵坐标：

- 范围： $0 \leq y3 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.fillTriangle(10, 10, 20, 20, 30, 30);
```

K 绘制多边形

```
canvas_port.strokePolygon(verticesArray);
```

功能描述

绘制多边形。

接口约束

无。

参数列表

- verticesArray为多边形顶点坐标array。
 - 顶点的横坐标范围 $0 \leq x \leq (\text{screenWidth}-1)$ 。
 - 顶点的纵坐标范围 $0 \leq y \leq (\text{screenHeight}-1)$ 。
 - 顶点个数 $3 \leq n \leq 10$

返回值

无。

接口示例

```
var verticesArray1 = [[10, 20], [20, 10], [40, 10], [30, 20]];
canvas_port.strokePolygon(verticesArray1);
```

L 填充多边形

```
canvas_port.fillPolygon(verticesArray);
```

功能描述

填充多边形。

接口约束

无。

参数列表

- verticesArray为多边形顶点坐标array。
 - 顶点的横坐标范围 $0 \leq x \leq (\text{screenWidth}-1)$ 。
 - 顶点的纵坐标范围 $0 \leq y \leq (\text{screenHeight}-1)$ 。
 - 顶点个数 $3 \leq n \leq 10$

返回值

无。

接口示例

```
var verticesArray1 = [[10, 20], [20, 10], [40, 10], [30, 20]];
canvas_port.fillPolygon(verticesArray1);
```

M 绘制圆弧

```
canvas_port.arc(x, y, radius, start_angle, end_angle, anti_clockwise);
```

功能描述

绘制圆弧。

接口约束

无。

参数列表

- x为圆心横坐标：
 - 范围： $0 \leq x \leq 320$ 。
- y为圆心纵坐标：
 - 范围： $0 \leq y \leq 480$ 。
- radius为圆弧半径：
 - 范围： $1 \leq \text{radius} \leq 320$ 。
- start_angle为起始角度：
 - 范围： $0 \leq \text{start_angle} \leq 360$ 。
- end_angle为终止角度：
 - 范围： $0 \leq \text{end_angle} \leq 360$ 。
- anti_clockwise为圆弧方向：
- 0表示顺时针，1表示逆时针。

返回值

无。

接口示例

```
canvas_port.arc(50, 30, 10, 0, 135, 0);
```

N 填充圆形

```
canvas_port.fillCircle(x, y, radius);
```

功能描述

填充圆形。

接口约束

无。

参数列表

- x为圆心横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为圆心纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。

- radius为圆弧半径：
 - 范围： $1 \leq \text{radius} \leq 320$ 。

返回值

无。

接口示例

```
canvas_port.fillCircle(60, 30, 25);
```

O 绘制椭圆

```
canvas_port.strokeEllipse(x1, x2, y1, y2);
```

功能描述

绘制椭圆。

接口约束

无。

参数列表

- x1为最左侧点横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- x2为最右侧点横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y1为最上侧点纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- y2为最下侧点纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.strokeEllipse(10, 40, 5, 60);
```

P 填充椭圆

```
canvas_port.fillEllipse(x1, x2, y1, y2);
```

功能描述

填充椭圆。

接口约束

无。

参数列表

- x1为最左侧点横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- x2为最右侧点横坐标：

- 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y1为最上侧点纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- y2为最下侧点纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.fillEllipse(10, 40, 5, 60);
```

Q 显示字符

```
canvas_port.fillText(x, y, "textString", "fontString");
```

功能描述

显示字符。（使用该功能需要提前将相关字库文件部署至开发板中，文件部署步骤请参考相关说明文档。）

接口约束

无。

参数列表

- x为显示起始点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为显示起始点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。
- "textString"为待显示字符串。
- "fontString"为字体字符串：
 - "0816"表示英文字体；
 - "1616"表示中文字体。

返回值

无。

接口示例

```
canvas_port.fillText(0, 0, "Hello World!", "0816");  
canvas_port.fillText(0, 20, "一丁", "1616");
```

R 绘制条形码

```
canvas_port.barcode(x, y, "codeNumberString");
```

功能描述

绘制条形码。

接口约束

无。

参数列表

- x为显示起始点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为显示起始点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。
- "codeNumberString"为待生成条形码的数字字符串。
 - "codeNumberString"的内容应符合条形码规范，是一个12位数字。

返回值

无。

接口示例

```
canvas_port.barcode(50, 50, "692116859353");
```

S 显示图片

```
canvas_port.drawImage("imageNameString", x, y);
```

功能描述

显示图片。（使用该功能需要提前将相关图片文件部署至开发板中，文件部署步骤请参考相关说明文档。）

接口约束

无。

参数列表

- "imageNameString"为图片文件名字符串。
 - "imageNameString"不可包含文件后缀名，例如部署的图片文件为test.pin则填入"test"。
- x为显示起始点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为显示起始点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.drawImage("test", 0, 0);
```

T 显示轮廓

```
canvas_port.stroke();
```

功能描述

显示轮廓。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
canvas_port.stroke();
```

U 清除屏幕

```
canvas_port.clear();
```

功能描述

清除屏幕。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
canvas_port.clear();
```

V 设置线宽属性

```
canvas_port.lineWidth = settingValue;
```

功能描述

设置线宽属性。

接口约束

无。

参数列表

- settingValue为所设置的线宽值。
 - 范围： $1 \leq \text{settingValue} \leq 8$
 - 线宽值可省略，省略时线宽值将默认为1。

返回值

无。

接口示例

```
canvas_port.lineWidth = 2;
```

W 设置画笔颜色属性

```
canvas_port.strokeStyle = settingValue;
```

功能描述

设置画笔颜色属性。

接口约束

无。

参数列表

- `settingValue`为所设置的画笔颜色属性。
 - 取值： $0x000000 \leq \text{settingValue} \leq 0xFFFFFFFF$ 。
 - 画笔颜色属性可省略，省略时画笔颜色属性值将默认为`0x000000`（黑色）。

返回值

无。

接口示例

```
canvas_port.strokeStyle = 0x000000;
```

4.2.3.5.3 约束

1. 模块接口中所涉及到的数值的属性必须是不小于零的整数，浮点数、负数等均为非法。
2. 在显式指定了`screenWidth`及`screenHeight`的情况下，`Canvas`模块将只对指定范围内的画笔内容进行处理显示并自动忽略超出范围部分。
3. `screenWidth`和`screenHeight`的设定值是相对于`lineWidth`设为1而言的，当`lineWidth`被设置为其他合法数值时，`screenWidth`和`screenHeight`所实际覆盖的范围会相应缩小，用户需自行计算并确保所绘制图形在其覆盖范围之内。

4.2.3.5.4 样例

样例A

介绍

初始化。

例程

```
canvas = require("canvas");
cfg = {screenType:"tft",orientation:1};
canvas_port = canvas.open(cfg);
```

样例B

介绍

`orientation`对比。

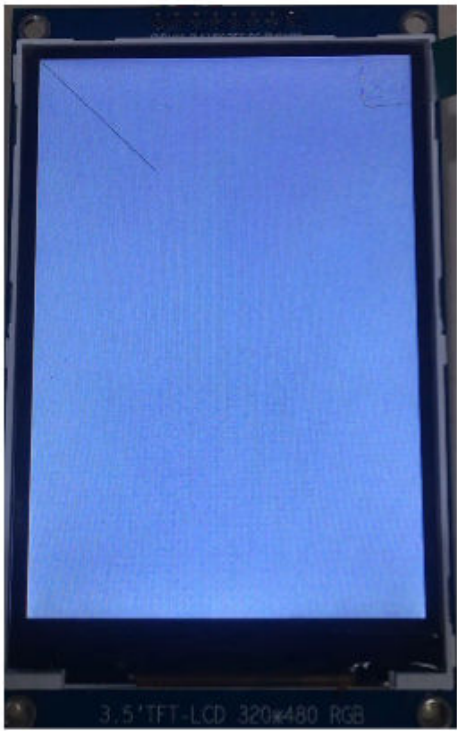
例程

```
canvas = require("canvas");
cfg = {screenType:"tft",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();
canvas_port.moveTo(0,0);
canvas_port.lineTo(100,100);
canvas_port.stroke();
canvas_port.closePath();
```

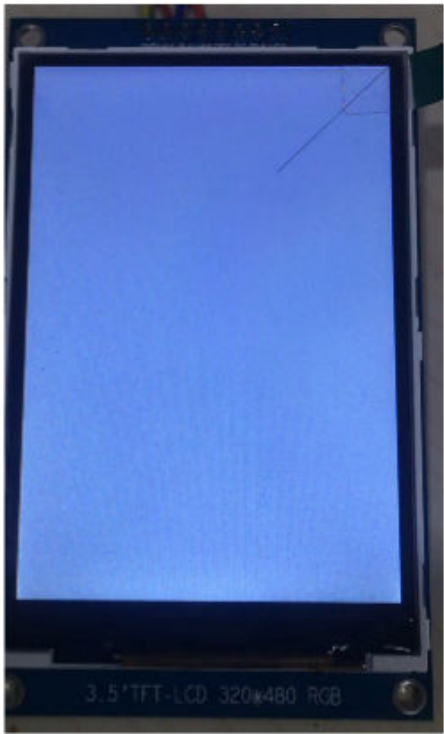
然后依次将代码模板中的`cfg`修改为：


```
cfg = {screenType:"tft",orientation:2};  
cfg = {screenType:"tft",orientation:3};  
cfg = {screenType:"tft",orientation:4};
```

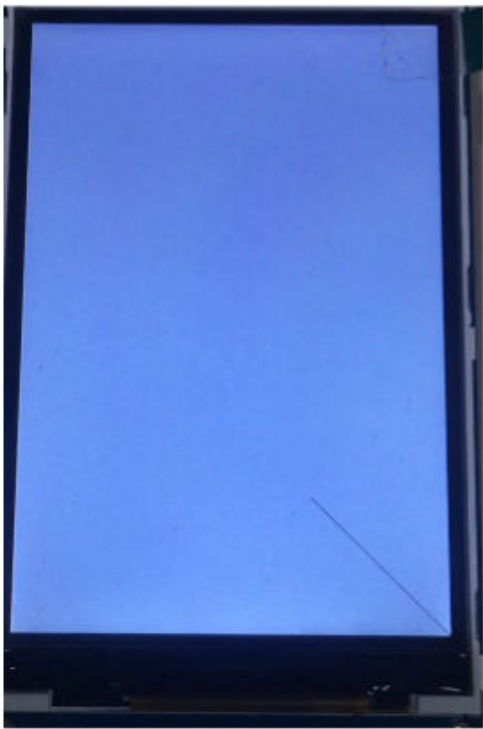
样例结果



orientation:1



orientation:2



orientation:3



orientation:4

样例C

介绍

绘制线段。

例程

```
canvas = require("canvas");
cfg = {screenType:"tft",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.lineWidth = 1;
canvas_port.strokeStyle = 0x000000;
canvas_port.moveTo(0, 0);
canvas_port.lineTo(50, 0);

canvas_port.lineWidth = 2;
canvas_port.strokeStyle = 0xFF0000;
canvas_port.moveTo(0, 10);
canvas_port.lineTo(50, 10);

canvas_port.lineWidth = 4;
canvas_port.strokeStyle = 0x00FF00;
canvas_port.moveTo(0, 20);
canvas_port.lineTo(50, 20);

canvas_port.lineWidth = 8;
canvas_port.strokeStyle = 0x0000FF;
canvas_port.moveTo(0, 30);
canvas_port.lineTo(50, 30);

canvas_port.stroke();
canvas_port.closePath();
```

代码中可通过设置`canvas_port.lineWidth`来调整线宽，线宽的取值范围为 $1 \leq \text{lineWidth} \leq 8$ 。代码中可通过设置`canvas_port.strokeStyle`来调整颜色，颜色的取值范围为 $0x000000 \leq \text{strokeStyle} \leq 0xFFFFFFFF$ 。

样例D

介绍

绘制矩形。

例程

```
canvas = require("canvas");
cfg = {screenType:"tft",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.lineWidth = 1;
canvas_port.strokeStyle = 0xFF0000;
canvas_port.strokeRect(0, 0, 29, 29);

canvas_port.lineWidth = 2;
canvas_port.strokeStyle = 0x00FF00;
canvas_port.strokeRect(40, 0, 69, 29);

canvas_port.lineWidth = 4;
canvas_port.strokeStyle = 0x0000FF;
canvas_port.strokeRect(80, 0, 109, 29);

canvas_port.stroke();
canvas_port.closePath();
```

代码中可通过设置`canvas_port.lineWidth`来调整线宽，线宽的取值范围为 $1 \leq \text{lineWidth} \leq 8$ 。代码中可通过设置`canvas_port.strokeStyle`来调整颜色，颜色的取值范围为 $0x000000 \leq \text{strokeStyle} \leq 0xFFFFFFFF$ 。

样例E

介绍

填充矩形。

例程

```
canvas = require("canvas");
cfg = {screenType:"tft",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();
canvas_port.strokeStyle = 0xFF0000;
canvas_port.fillRect(0, 0, 29, 29);

canvas_port.stroke();
canvas_port.closePath();
```

代码中可通过设置`canvas_port.strokeStyle`来调整颜色，颜色的取值范围为 $0x000000 \leq \text{strokeStyle} \leq 0xFFFFFFFF$ 。

样例结果



样例F

介绍

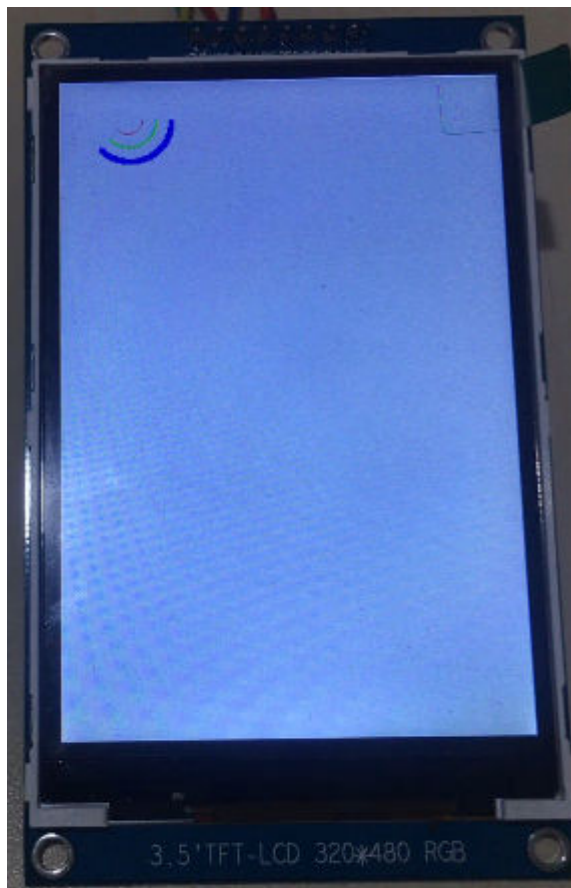
绘制圆弧。

例程

```
canvas = require("canvas");
cfg = {screenType:"tft",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();
canvas_port.lineWidth = 1;
canvas_port.strokeStyle = 0xFF0000;
canvas_port.arc(50, 30, 10, 0, 135, 0);
canvas_port.lineWidth = 2;
canvas_port.strokeStyle = 0x00FF00;
canvas_port.arc(50, 30, 20, 0, 135, 0);
canvas_port.lineWidth = 4;
canvas_port.strokeStyle = 0x0000FF;
canvas_port.arc(50, 30, 30, 0, 135, 0);
canvas_port.stroke();
canvas_port.closePath();
```

代码中可通过设置`canvas_port.lineWidth`来调整线宽，线宽的取值范围为 $1 \leq \text{lineWidth} \leq 8$ 。代码中可通过设置`canvas_port.strokeStyle`来调整颜色，颜色的取值范围为 $0x000000 \leq \text{strokeStyle} \leq 0xFFFFFFFF$ 。

样例结果



顺时针

样例G

介绍

显示字符。

例程

```
canvas = require("canvas");
cfg = {screenType:"tft",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.strokeStyle = 0xFF0000;
canvas_port.fillText(0, 0, "Hello World!", "0816");
canvas_port.strokeStyle = 0x0000FF;
canvas_port.fillText(0, 20, "一丁", "1616");

canvas_port.stroke();
canvas_port.closePath();
```

代码中可通过设置`canvas_port.strokeStyle`来调整颜色，颜色的取值范围为 $0x000000 \leq \text{strokeStyle} \leq 0xFFFFFFFF$ 。

样例结果



样例H

介绍

绘制条形码。

例程

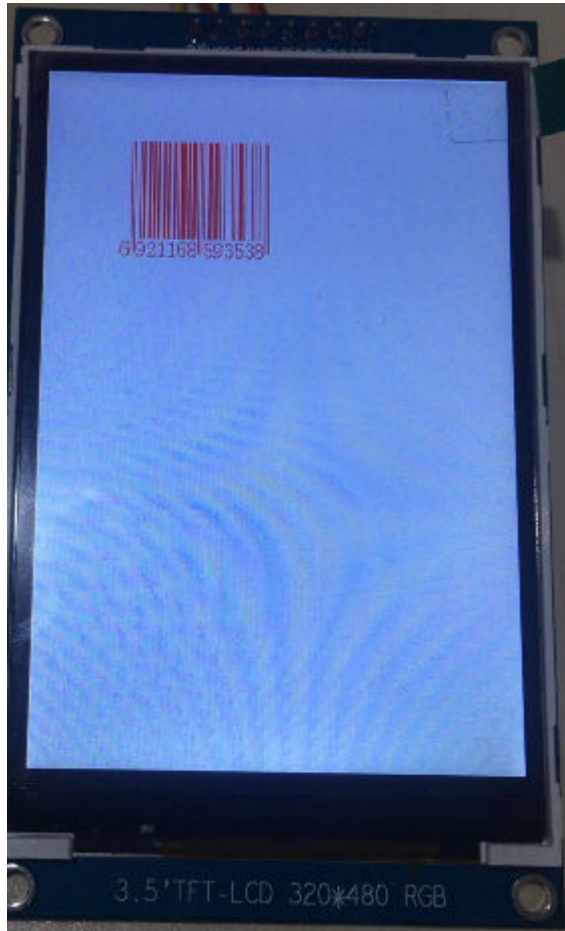
```
canvas = require("canvas");
cfg = {screenType:"tft",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.strokeStyle = 0xFF0000;
canvas_port.barcode(50, 50, "692116859353");

canvas_port.stroke();
canvas_port.closePath();
```

代码中可通过设置`canvas_port.strokeStyle`来调整颜色，颜色的取值范围为 $0x000000 \leq \text{strokeStyle} \leq 0xFFFFFFFF$ 。

样例结果



样例I

介绍

显示图片。

例程

```
canvas = require("canvas");
cfg = {screenType:"tft",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.drawImage("huawei", 0, 0);

canvas_port.stroke();
canvas_port.closePath();
```

样例结果



样例J

介绍

通过设置画笔颜色实现反显。

例程

```
canvas = require("canvas");
cfg = {screenType:"tft",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.strokeStyle = 0xFF0000;
canvas_port.fillCircle(60, 30, 25);

canvas_port.strokeStyle = 0xFFFFFF;
canvas_port.fillCircle(60, 30, 15);

canvas_port.stroke();
canvas_port.closePath();
```

样例K

介绍

清除屏幕。

例程

```
canvas = require("canvas");
cfg = {screenType:"tft",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.arc(30, 30, 20, 0, 360, 0);

canvas_port.strokeStyle = 0xFFFFFF; /*清屏前将色彩设置为白色*/
canvas_port.clear();

canvas_port.stroke();
canvas_port.closePath();
```


4.2.3.6 墨水屏

4.2.3.6.1 介绍

A 模块参数

canvas module是AntJS系统基于不同屏幕的专用画图子模块，其接口与HTML5的canvas API保持兼容。本文是canvas module在ink屏上具体实现的指导说明。

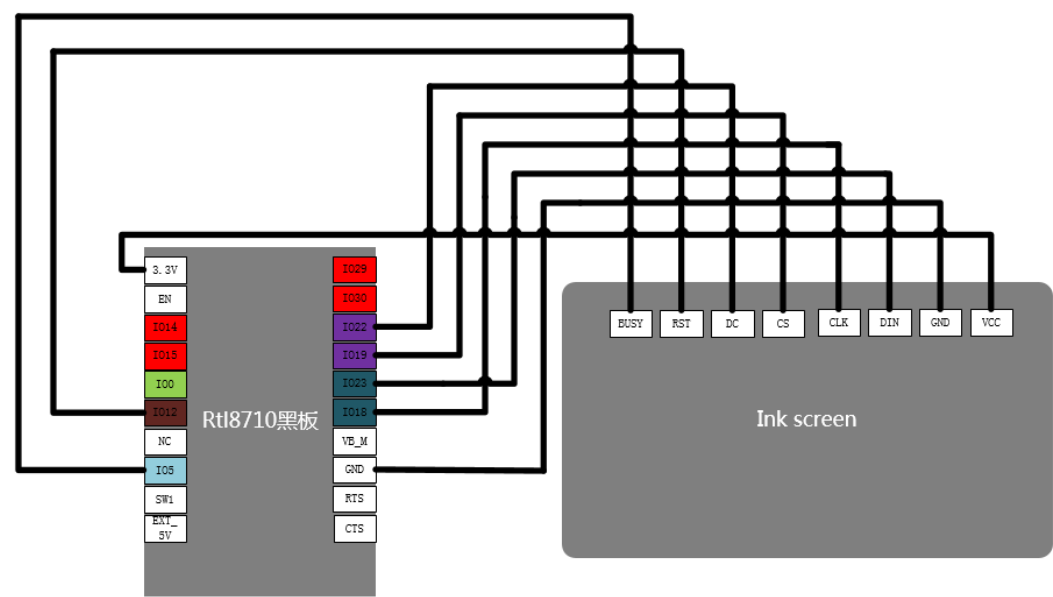
ink screen具有功耗低、视角宽、阳光直射下仍可清晰显示等优点，经常被用于货架标签、工业仪表等显示应用。



ink screen模块参数列表：

参数	数值
工作电压	3.3V
通信接口	SPI
外形尺寸	48mm
显示尺寸	27.6mm × 27.6mm
点距	0.138 × 0.138
分辨率	200 × 200
刷新功耗	26.4mV(typ.)
待机功耗	<0.17mV
可视角度	>170°

B 模块连线



序号	开发板管脚	模块管脚	说明
1	3.3V	VCC	VCC是模块的电源引脚
2	GND	GND	GND是模块的接地引脚
3	IO23	DIN	DIN是模块的SPI通信MOSI引脚
4	IO18	CLK	CLK是模块的SPI通信SCK引脚
5	IO19	CS	CS是模块的SPI片选引脚
6	IO22	DC	CS是模块的数据/命令控制引脚
7	IO12	RST	RST是模块的复位引脚
8	IO5	BUSY	BUSY是模块的忙状态输出引脚

4.2.3.6.2 模块接口

A 引用模块

```
var canvas = require("canvas");
```

B 打开模块

```
var canvasPort = canvas.open(cfg);
```

功能描述

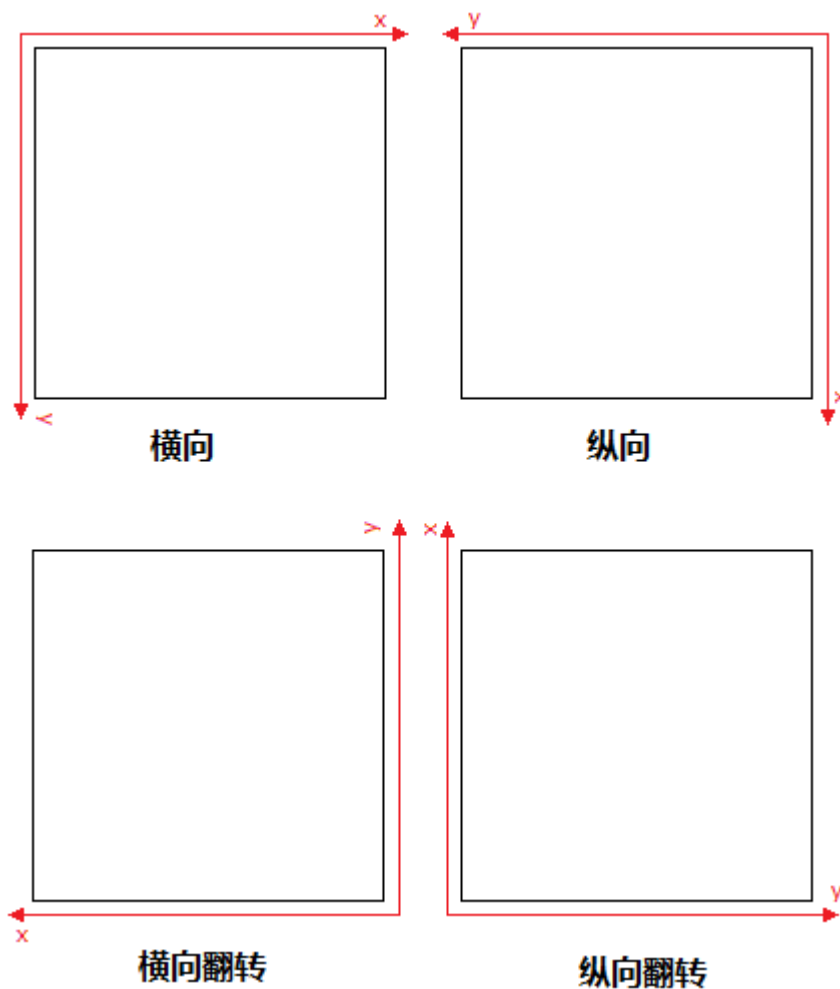
- 按照cfg的配置打开模块。

接口约束

无。

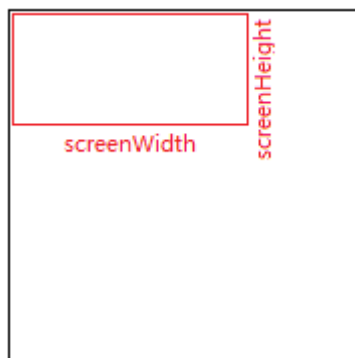
参数列表

- `cfg`表示打开的配置参数，其中应包含：
 - `screenType`信息。`screenType`表示屏幕类型：
 - 字符串“`ink`”表示ink屏；
 - 字符串“`oled`”表示oled屏；
 - 字符串“`tft`”表示tft屏；
 - `screenType`必须显示指定，不可省略。
 - `orientation`信息。`orientation`表示显示方向：
 - 取值选项如下：
 - 1：横向
 - 2：纵向
 - 3：横向翻转
 - 4：纵向翻转



- `orientation`必须显示指定，不可省略。
- `screenWidth`信息。`screenWidth`表示自定义屏幕宽度：
 - 范围： $1 \leq \text{screenWidth} \leq 200$ 。
 - `screenWidth`可省略，省略时`screenWidth`200。
- `screenHeight`信息。`screenHeight`表示自定义屏幕高度：
 - 范围： $1 \leq \text{screenHeight} \leq 200$ 。

- screenHeight可省略，省略时screenHeight将默认设置为200。



以横向为例

- canvasPort为方法返回的端口句柄。

返回值

无。

接口示例

```
/* 显示指定自定义屏幕宽高 */
var canvas = require("canvas");
var cfg = {screenType:"ink", screenWidth:64, screenHeight:32, orientation:1};
var canvas_port = canvas.open(cfg);

/* 省略自定义屏幕宽高 */
var canvas = require("canvas");
var cfg = {screenType:"ink",orientation:1};
var canvas_port = canvas.open(cfg);
```

C 新建路径

```
canvas_port.beginPath();
```

功能描述

新建一条路径，路径一旦创建成功，图形绘制命令被指向到路径上生成路径。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
canvas_port.beginPath();
```

D 关闭路径

```
canvas_port.closePath();
```

功能描述

关闭路径，之后图形绘制命令又重新指向到上下文中。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
canvas_port.closePath();
```

E 设置画笔起点

```
canvas_port.moveTo(x, y);
```

功能描述

设置画笔的起始点坐标。

接口约束

无。

参数列表

- x为起始点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为起始点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.moveTo(0, 0);
```

F 绘制线段

```
canvas_port.lineTo(x, y);
```

功能描述

绘制线段。

接口约束

无。

参数列表

- x为终点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为终点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.lineto(100, 100);
```

G 绘制矩形

```
canvas_port.strokeRect(x1, y1, x2, y2);
```

功能描述

绘制矩形。

接口约束

无。

参数列表

- (x1,y1)和(x2,y2)分别是矩形对角顶点的坐标。
- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.strokeRect(0, 0, 29, 29);
```

H 填充矩形

```
canvas_port.fillRect(x1, y1, x2, y2);
```

功能描述

填充矩形。

接口约束

无。

参数列表

- (x1,y1)和(x2,y2)分别是矩形对角顶点的坐标。
- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：

- 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.fillRect(0, 0, 29, 29);
```

I 绘制三角形

```
canvas_port.strokeTriangle(x1, y1, x2, y2, x3, y3);
```

功能描述

绘制三角形。

接口约束

无。

参数列表

- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。
- x3为顶点3横坐标：
 - 范围： $0 \leq x3 \leq (\text{screenWidth}-1)$ 。
- y3为顶点3纵坐标：
 - 范围： $0 \leq y3 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.strokeTriangle(10, 10, 20, 20, 30, 30);
```

J 填充三角形

```
canvas_port.fillTriangle(x1, y1, x2, y2, x3, y3);
```

功能描述

填充三角形。

接口约束

无。

参数列表

- x1为顶点1横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- y1为顶点1纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- x2为顶点2横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y2为顶点2纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。
- x3为顶点3横坐标：
 - 范围： $0 \leq x3 \leq (\text{screenWidth}-1)$ 。
- y3为顶点3纵坐标：
 - 范围： $0 \leq y3 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.fillTriangle(10, 10, 20, 20, 30, 30);
```

K 绘制多边形

```
canvas_port.strokePolygon(verticesArray);
```

功能描述

绘制多边形。

接口约束

无。

参数列表

- verticesArray为多边形顶点坐标array。
 - 顶点的横坐标范围 $0 \leq x \leq (\text{screenWidth}-1)$ 。
 - 顶点的纵坐标范围 $0 \leq y \leq (\text{screenHeight}-1)$ 。
 - 顶点个数 $3 \leq n \leq 10$

返回值

无。

接口示例

```
var verticesArray1 = [[10, 20], [20, 10], [40, 10], [30, 20]];
canvas_port.strokePolygon(verticesArray1);
```

L 填充多边形

```
canvas_port.fillPolygon(verticesArray);
```

功能描述

填充多边形。

接口约束

无。

参数列表

- verticesArray为多边形顶点坐标array。
 - 顶点的横坐标范围 $0 \leq x \leq (\text{screenWidth}-1)$ 。
 - 顶点的纵坐标范围 $0 \leq y \leq (\text{screenHeight}-1)$ 。
 - 顶点个数 $3 \leq n \leq 10$

返回值

无。

接口示例

```
var verticesArray1 = [[10, 20], [20, 10], [40, 10], [30, 20]];
canvas_port.fillPolygon(verticesArray1);
```

M 绘制圆弧

```
canvas_port.arc(x, y, radius, start_angle, end_angle, anti_clockwise);
```

功能描述

绘制圆弧。

接口约束

无。

参数列表

- x为圆心横坐标：
 - 范围： $0 \leq x \leq 199$ 。
- y为圆心纵坐标：
 - 范围： $0 \leq y \leq 199$ 。
- radius为圆弧半径：
 - 范围： $1 \leq \text{radius} \leq 200$ 。
- start_angle为起始角度：
 - 范围： $0 \leq \text{start_angle} \leq 360$ 。
- end_angle为终止角度：
 - 范围： $0 \leq \text{end_angle} \leq 360$ 。
- anti_clockwise为圆弧方向：
- 0表示顺时针，1表示逆时针。

返回值

无。

接口示例

```
canvas_port.arc(50, 30, 10, 0, 135, 0);
```

N 填充圆形

```
canvas_port.fillCircle(x, y, radius);
```

功能描述

填充圆形。

接口约束

无。

参数列表

- x为圆心横坐标：
 - 范围： $0 \leq x \leq 199$ 。
- y为圆心纵坐标：
 - 范围： $0 \leq y \leq 199$ 。
- radius为圆弧半径：
 - 范围： $1 \leq \text{radius} \leq 200$ 。

返回值

无。

接口示例

```
canvas_port.fillCircle(60, 30, 25);
```

O 绘制椭圆

```
canvas_port.strokeEllipse(x1, x2, y1, y2);
```

功能描述

绘制椭圆。

接口约束

无。

参数列表

- x1为最左侧点横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- x2为最右侧点横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y1为最上侧点纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- y2为最下侧点纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.strokeEllipse(10, 40, 5, 60);
```

P 填充椭圆

```
canvas_port.fillEllipse(x1, x2, y1, y2);
```

功能描述

填充椭圆。

接口约束

无。

参数列表

- x1为最左侧点横坐标：
 - 范围： $0 \leq x1 \leq (\text{screenWidth}-1)$ 。
- x2为最右侧点横坐标：
 - 范围： $0 \leq x2 \leq (\text{screenWidth}-1)$ 。
- y1为最上侧点纵坐标：
 - 范围： $0 \leq y1 \leq (\text{screenHeight}-1)$ 。
- y2为最下侧点纵坐标：
 - 范围： $0 \leq y2 \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.fillEllipse(10, 40, 5, 60);
```

Q 显示字符

```
canvas_port.fillText(x, y, "textString", "fontString");
```

功能描述

显示字符。（使用该功能需要提前将相关字库文件部署至开发板中，文件部署步骤请参考相关说明文档。）

接口约束

无。

参数列表

- x为显示起始点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为显示起始点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。
- "textString"为待显示字符串。
- "fontString"为字体字符串：
 - "0816"表示英文字体；
 - "1616"表示中文字体。

返回值

无。

接口示例

```
canvas_port.fillText(0, 0, "Hello World!", "0816");  
canvas_port.fillText(0, 20, "一丁", "1616");
```

R 绘制条形码

```
canvas_port.barcode(x, y, "codeNumberString");
```

功能描述

绘制条形码。

接口约束

无。

参数列表

- x为显示起始点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为显示起始点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。
- "codeNumberString"为待生成条形码的数字字符串。
 - "codeNumberString"的内容应符合条形码规范，是一个12位数字。

返回值

无。

接口示例

```
canvas_port.barcode(50, 50, "692116859353");
```

S 显示图片

```
canvas_port.drawImage("imageNameString", x, y);
```

功能描述

显示图片。（使用该功能需要提前将相关图片文件部署至开发板中，文件部署步骤请参考相关说明文档。）

接口约束

无。

参数列表

- "imageNameString"为图片文件名字符串。
 - "imageNameString"不可包含文件后缀名，例如部署的图片文件为test.pin则填入"test"。
- x为显示起始点横坐标：
 - 范围： $0 \leq x \leq (\text{screenWidth}-1)$ 。
- y为显示起始点纵坐标：
 - 范围： $0 \leq y \leq (\text{screenHeight}-1)$ 。

返回值

无。

接口示例

```
canvas_port.drawImage("test", 0, 0);
```

T 显示图形

```
canvas_port.stroke();
```

功能描述

显示图形。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
canvas_port.stroke();
```

U 清除屏幕

```
canvas_port.clear();
```

功能描述

清除屏幕。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
canvas_port.clear();
```

V 设置线宽属性

```
canvas_port.lineWidth = settingValue;
```

功能描述

设置线宽属性。

接口约束

无。

参数列表

- settingValue为所设置的线宽值。
 - 范围： $1 \leq \text{settingValue} \leq 8$

- 线宽值可省略，省略时线宽值将默认为1。

返回值

无。

接口示例

```
canvas_port.lineWidth = 2;
```

W 设置画笔颜色属性

```
canvas_port.strokeStyle = settingValue;
```

功能描述

设置画笔颜色属性。

接口约束

无。

参数列表

- settingValue为所设置的画笔颜色属性。
 - 取值：0x000000表示画笔为黑色，0xFFFFFFFF表示画笔为白色。
 - 画笔颜色属性可省略，省略时画笔颜色属性值将默认为0x000000（黑色）。

返回值

无。

接口示例

```
canvas_port.strokeStyle = 0x000000;
```

4.2.3.6.3 约束

1. 模块接口中所涉及到的数值的属性必须是不小于零的整数，浮点数、负数等均为非法。
2. 在显式指定了screenWidth及screenHeight的情况下，Canvas模块将只对指定范围内的画笔内容进行处理显示并自动忽略超出范围部分。
3. screenWidth和screenHeight的设定值是相对于lineWidth设为1而言的，当lineWidth被设置为其他合法数值时，screenWidth和screenHeight所实际覆盖的范围会相应缩小，用户需自行计算并确保所绘制图形在其覆盖范围之内。

4.2.3.6.4 样例

样例A

介绍

初始化。

例程

```
canvas = require("canvas");  
cfg = {screenType:"ink",orientation:1};  
canvas_port = canvas.open(cfg);
```

样例B

介绍

orientation对比。

例程

```
canvas = require("canvas");
cfg = {screenType:"ink",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.lineWidth = 4;
canvas_port.moveTo(0, 0);
canvas_port.lineTo(30, 30);

canvas_port.stroke();
canvas_port.closePath();
```

然后依次将代码模板中的cfg修改为:

```
cfg = {screenType:"ink",orientation:2};
cfg = {screenType:"ink",orientation:3};
cfg = {screenType:"ink",orientation:4};
```

样例结果



orientation:1



orientation:2



orientation:3



orientation:4

样例C

介绍

绘制线段。

例程

```
canvas = require("canvas");
cfg = {screenType:"ink",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.lineWidth = 1;
canvas_port.moveTo(0, 5);
canvas_port.lineTo(50, 5);

canvas_port.lineWidth = 2;
canvas_port.moveTo(0, 10);
canvas_port.lineTo(50, 10);

canvas_port.lineWidth = 4;
canvas_port.moveTo(0, 20);
canvas_port.lineTo(50, 20);

canvas_port.lineWidth = 8;
canvas_port.moveTo(0, 30);
canvas_port.lineTo(50, 30);
```

```
canvas_port.stroke();  
canvas_port.closePath();
```

代码中可通过设置`canvas_port.lineWidth`来调整线宽，线宽的取值范围为 $1 \leq \text{lineWidth} \leq 8$ 。

样例结果



样例D

介绍

绘制矩形。

例程

```
canvas = require("canvas");  
cfg = {screenType:"ink",orientation:1};  
canvas_port = canvas.open(cfg);  
canvas_port.beginPath();  
  
canvas_port.lineWidth = 1;  
canvas_port.strokeRect(0, 0, 29, 29);  
  
canvas_port.lineWidth = 2;  
canvas_port.strokeRect(40, 0, 69, 29);  
  
canvas_port.lineWidth = 4;  
canvas_port.strokeRect(80, 0, 109, 29);  
  
canvas_port.stroke();  
canvas_port.closePath();
```

代码中可通过设置`canvas_port.lineWidth`来调整线宽，线宽的取值范围为 $1 \leq \text{lineWidth} \leq 8$ 。

样例结果



样例E

介绍

填充矩形。

例程

```
canvas = require("canvas");
cfg = {screenType:"ink",orientation:1};
canvas_port = canvas.open(cfg);

canvas_port.beginPath();
canvas_port.fillRect(0, 0, 29, 29);
canvas_port.fillRect(40, 0, 69, 29);
canvas_port.fillRect(80, 0, 109, 29);

canvas_port.stroke();
canvas_port.closePath();
```

样例结果



样例F

介绍

绘制圆弧。

例程

```
canvas = require("canvas");
cfg = {screenType:"ink",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.lineWidth = 1;
canvas_port.arc(50, 30, 10, 0, 135, 0);

canvas_port.lineWidth = 2;
canvas_port.arc(50, 30, 20, 0, 135, 0);

canvas_port.lineWidth = 4;
canvas_port.arc(50, 30, 30, 0, 135, 0);

canvas_port.stroke();
canvas_port.closePath();
```

代码中可通过设置`canvas_port.lineWidth`来调整线宽，线宽的取值范围为 $1 \leq \text{lineWidth} \leq 8$ 。

样例结果



样例G

介绍

显示字符。

例程

```
canvas = require("canvas");
cfg = {screenType:"ink",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.fillText(0, 0, "Hello World!", "0816");
canvas_port.fillText(0, 20, "一丁", "1616");

canvas_port.stroke();
canvas_port.closePath();
```

样例结果



样例H

介绍

绘制条形码。

例程

```
canvas = require("canvas");
cfg = {screenType:"ink",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.barcode(50, 50, "692116859353");

canvas_port.stroke();
canvas_port.closePath();
```

样例结果



样例I

介绍

显示图片。

例程

```
canvas = require("canvas");
cfg = {screenType:"ink",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.drawImage("huawei", 0, 0);

canvas_port.stroke();
canvas_port.closePath();
```

样例结果



样例J

介绍

通过设置画笔颜色实现反显。

例程

```
canvas = require("canvas");
cfg = {screenType:"ink",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.strokeStyle = 0x000000;
canvas_port.fillCircle(60, 30, 25);
canvas_port.strokeStyle = 0xFFFFFF;
canvas_port.fillCircle(60, 30, 10);

canvas_port.stroke();
canvas_port.closePath();
```

样例K

介绍

清除屏幕。

例程

```
canvas = require("canvas");
cfg = {screenType:"ink",orientation:1};
canvas_port = canvas.open(cfg);
canvas_port.beginPath();

canvas_port.arc(30, 30, 20, 0, 360, 0);
canvas_port.clear();

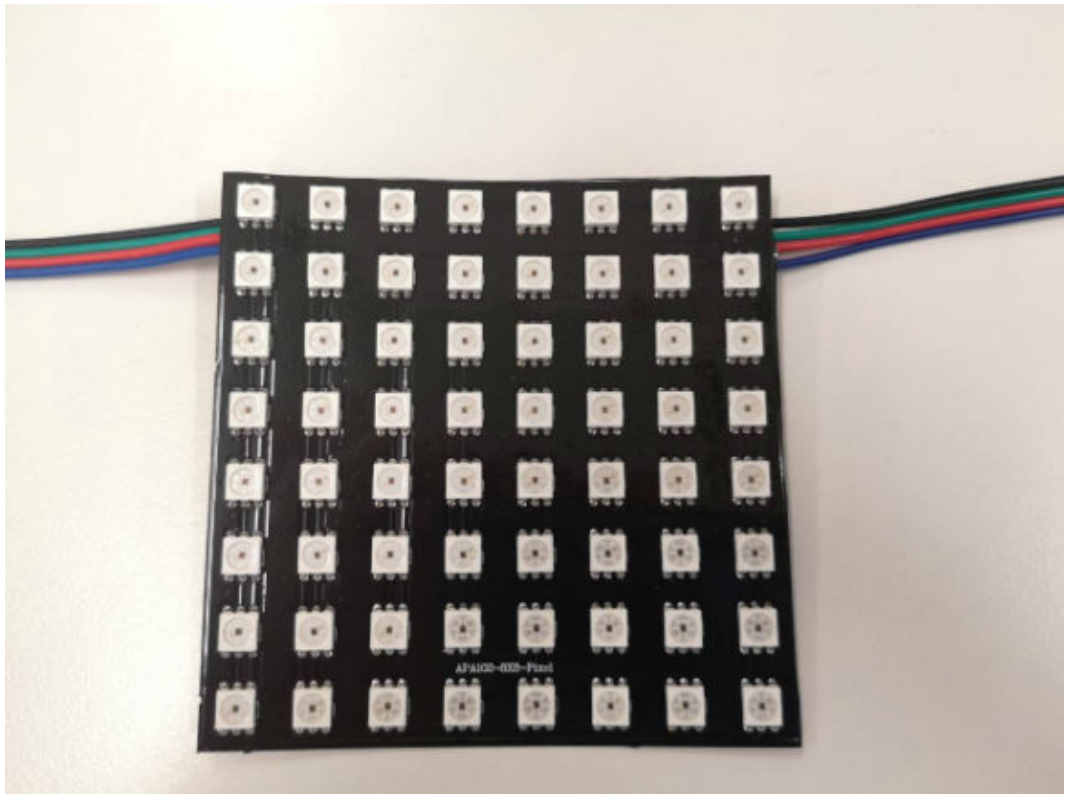
canvas_port.stroke();
canvas_port.closePath();
```

4.2.3.7 像素软屏

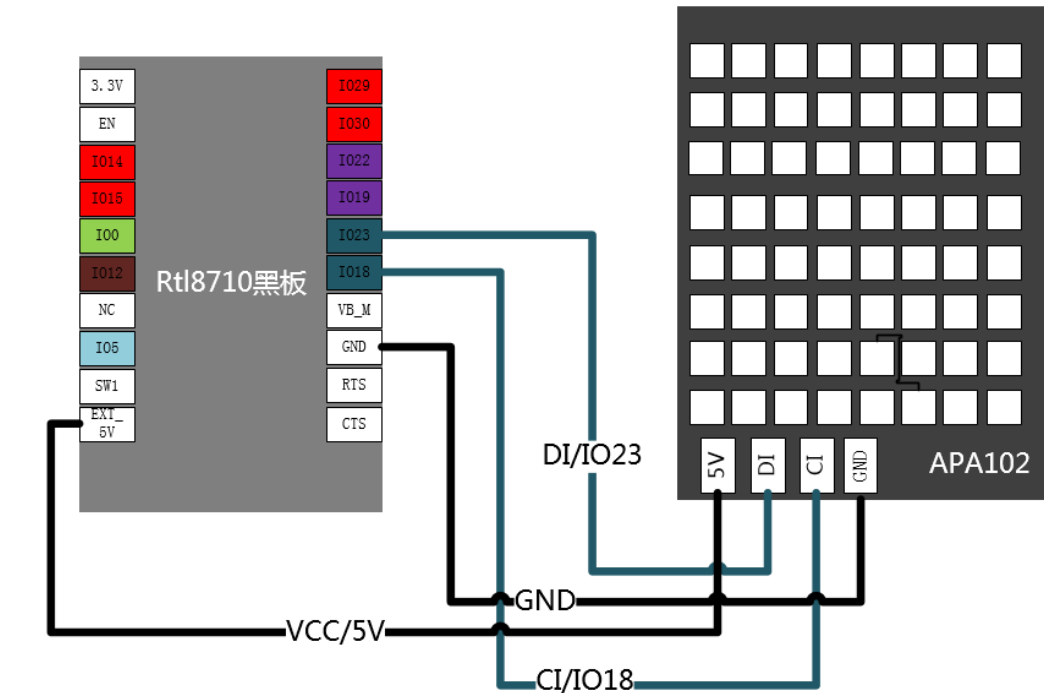
4.2.3.7.1 介绍

A 模块参数

APA102是一种RGB全彩LED，该模块采用的是APA102灯珠集成的软屏，参考图片如下：



B 模块连线



开发板与模块引脚连接表：

序号	开发板管脚	模块管脚	备注
1	EXT_5V	5V	灯珠需要5v的供电电压
2	IO23	DI	DI是数据输入线
3	IO18	CI	CI是时钟输入线
4	GND	GND	接地

4.2.3.7.2 模块接口

A 引用模块

```
var apa102 = require('apa102');
```

B 初始化模块

```
apa102.init(ionum1, ionum2);
```

功能描述

根据配置初始化apa102端口。

接口约束

无。

参数列表

- `ionum1`是使用开发板的IO号，取值范围是[0, 5, 12, 18, 19, 22, 23, 29, 30]，且只能取正整数，建议不要使用IO0。
- `ionum2`是使用开发板的IO号，取值范围是[0, 5, 12, 18, 19, 22, 23, 29, 30]，且只能取正整数，建议不要使用IO0。

返回值

初始化成功则返回0，不成功则返回报错信息。

接口示例

```
apa102.init(18, 23);
```

C 运行模式

```
apa102.runMode(light_mode, num_leds, value);
```

功能描述

设置APA102软屏的亮灯模式，以及软屏的灯珠个数，以及灯珠闪亮的颜色。

接口约束

无。

参数列表

- `light_mode`：目前设置了三种灯亮的方式“LED_FLOW”（流水灯模式），“LED_BREATH”（呼吸灯模式），“LED_COLOR”（全亮模式）。
- `num_leds`：目前使用的软屏是8*8的规格，共64个灯珠。（根据软屏上灯珠的个数确定）
- `value`：RGB颜色值或者十六进制颜色码。

返回值

设置成功返回0，失败返回错误信息。

接口示例

```
apa.runMode(apa.LED_FLOW, 64, 0xf05261);
```

D 设置亮度

```
apa102.setBright(bright_val);
```

功能描述

设置灯珠的亮度。

接口约束

无。

参数列表

- `bright_val`：亮度的范围是0~31，默认值为5。

返回值

设置成功返回0，失败返回错误信息。

接口示例

```
apa102.setBright(10);
```

4.2.3.7.3 约束

无。

4.2.3.7.4 样例

介绍

显示流水灯。

例程

```
var apa=require(' apa102');
var tim=require(' timer');
apa.init(18,23);//I018和I023，采用的是爱联第一代的板子(绿板子)
tim.setInterval(function() {
    apa.runMode(apa.LED_FLOW, 64, 0xf05261);
}, 50);
/*
mode:
    apa.LED_FLOW    流水灯
    apa.LED_BREATH  呼吸灯
    apa.LED_COLOR   全亮
num_leds:1~64
color:
    red:0xf05261
    blue:0x1d64b1
    pink:0xb11da2
    yellow:0xf0d752
    还提供了apa.setBright(int val)接口，val的范围是0~31的亮度，不设置的话val默认为5
*/
apa.runMode(apa.LED_FLOW, 64, 0xf05261);
print("js execute done");
```

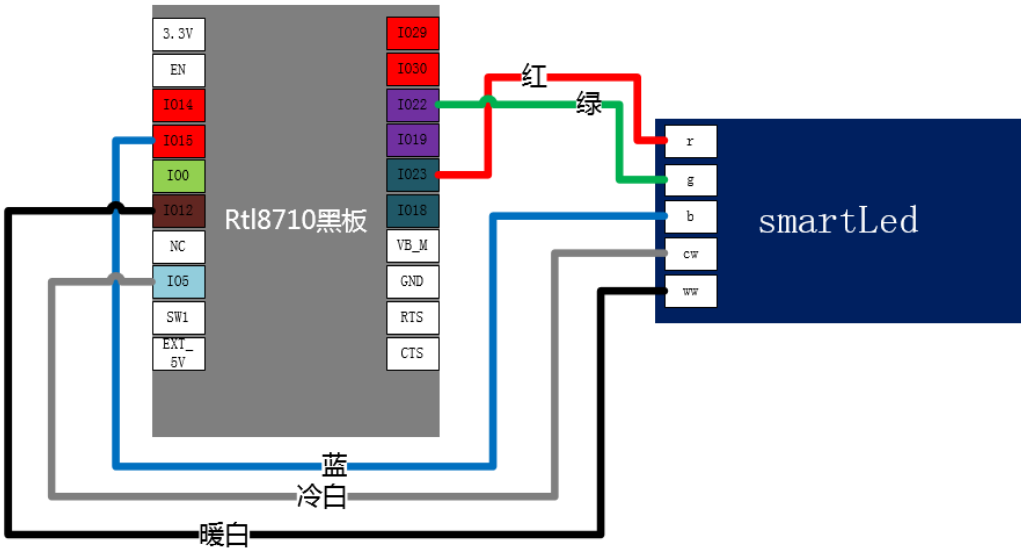
4.2.3.8 智能灯

4.2.3.8.1 介绍

A 模块参数

smartLed模块可使用5路PWM波去控制智能灯模块的红、绿、蓝、冷白和暖白五种颜色。

B 模块连线



开发板与模块管脚连接表：

序号	开发板管脚	模块管脚	说明
1	IO5	cw	冷白
2	IO12	ww	暖白
3	IO15	b	蓝
4	IO22	g	绿
5	IO23	r	红

4.2.3.8.2 模块接口

A 引用模块

```
var smartLed = require('smartLed')
```

B 打开smartLed端口

```
smartLed.open(config);
```

功能描述

打开smartLed的端口

接口约束

无。

参数列表

```
- config: 打开操作的配置信息，包括如下属性：
- connection: led的连接方式，module.CA表示共阳型连接，module.CC表示共阴型连接，默认为module.CA；
- period: 脉冲周期，类型为整数number（number>=0），单位是us，默认脉冲周期为1ms，即为1000(us)；
- cwPin: 色温型冷白驱动管脚，以爱联模组RTL8710开发板为例，可用引脚号为0，5，12，14，15，18，19，22，23。0引脚不推荐使用，具体参见gpio模块约束部分，默认为5；
- wwPin: 色温型暖白驱动管脚，以爱联模组RTL8710开发板为例，可用引脚号为0，5，12，14，15，18，19，22，23。0引脚不推荐使用，具体参见gpio模块约束部分，默认为5；
- rPin: 色彩型红色驱动管脚，以爱联模组RTL8710开发板为例，可用引脚号为0，5，12，14，15，18，19，22，23。0引脚不推荐使用，具体参见gpio模块约束部分，默认为5；
- gPin: 色彩型绿色驱动管脚，以爱联模组RTL8710开发板为例，可用引脚号为0，5，12，14，15，18，19，22，23。0引脚不推荐使用，具体参见gpio模块约束部分，默认为5；
- bPin: 色彩型蓝色驱动管脚，以爱联模组RTL8710开发板为例，可用引脚号为0，5，12，14，15，18，19，22，23。0引脚不推荐使用，具体参见gpio模块约束部分，默认为5
```

返回值

smartLed端口。

接口示例

```
var config = {
  connection:module.CA,
  pwmFreq:1,
  cwPin:5,
  wwPin:12,
  rPin:23,
  gPin:22,
  bPin:15
};
var port = smartLed.open(config);
```

C 点亮

```
void switchOn();
```

功能描述

点亮智能灯。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
port.switchOn();
```

D 熄灭

```
void switchOff();
```

功能描述

熄灭智能灯。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
port.switchOff();
```

E 亮度调节

```
void brightSet(brightLevel);
```

功能描述

调节当前智能灯的亮度。

接口约束

无。

参数列表

brightLevel: 亮度值，其取值范围为[0,255]。

返回值

无。

接口示例

```
port.brightSet(128);
```

F 色温调节

```
void cctSet(cctVal);  
或  
void cctSet(cwLevel, wwLevel);
```

功能描述

设置智能灯的色温，支持两种方式接口。

接口约束

无。

参数列表

- cctVal: 色温值，其取值范围为[2700,6500];
- cwVal: 冷白值，其取值范围为[0,255];
- wwVal: 暖白值，其取值范围为[0,255];

返回值

无。

接口示例

```
port.cctSet(4000);port.cctSet(50, 100);
```

G 色彩调节

```
void colorSet(colorVal);  
或  
void colorSet(rLevel, gLevel, bLevel);
```

功能描述

设置智能灯的色彩，支持两种方式的接口。

接口约束

无。

参数列表

- colorVal: 颜色设置值，其取值范围为[0x000000,0xFFFFFFFF]，第一个字节表示红色值，第二个字节表示绿色值，第三个字节表示蓝色值；
- rLevel: 红色值，其取值范围为[0,255];
- gLevel: 绿色值，其取值范围为[0,255];
- bLevel: 蓝色值，其取值范围为[0,255];

返回值

无。

接口示例

```
port.colorSet(0x0000FF);  
port.colorSet(0, 0, 255);
```

H 静态模式

```
void staticMode(cctVal, colorVal, brightLevel)  
或  
void staticMode(cwValue, wwValue, colorVal, brightLevel)
```

功能描述

设置智能灯为静态模式，同时输入冷暖白和RGB及亮度值。

接口约束

无。

参数列表

- cctVal: 色温值，其取值范围为[2700,6500];
- cwVal: 冷白值，其取值范围为[0,255];
- wwVal: 暖白值，其取值范围为[0,255];
- colorVal: 颜色设置值，其取值范围为[0x000000,0xFFFFFF]，第一个字节表示红色值，第二个字节表示绿色值，第三个字节表示蓝色值;
- brightLevel: 亮度值，其取值范围为[0,255];

返回值

无。

接口示例

```
port.staticMode(2700, 0, 50);  
port.staticMode(120, 200, 0, 50);
```

I 持续周期性变化接口

```
port.on(module.TIMER, period, duration, function);
```

功能描述

运用该接口可实现闪烁模式、呼吸模式及渐变模式。

接口约束

无。

参数列表

- module.TIMER:事件回调函数的类型，目前只支持module.TIMER，表示持续周期性变化接口。
- period:调用时间间隔，单位为ms。
- duration:总共持续时间，-1为无穷大，单位为ms，是period的倍数。
- function: 为类似function(count){}形式的函数，其中count为当前计数，从1开始，每隔period时间，将调用一次。

返回值

无。

接口示例

```
//闪烁间隔为1s，总共闪烁1分钟  
port.on(module.TIMER, 1000, 60000, function(count){  
    if (count % 2 == 0){  
        port.brightSet(0);  
    }  
});
```

```
    }  
    else{  
        port.brightSet(255);  
    }  
});  
  
//20ms刷新一次呼吸灯亮度，最亮到下次最亮的周期为1s，保持一直呼吸  
port.on(module.TIMER, 20, -1, function(count){  
    var data = count % 50; // 呼吸周期为1s，20ms刷新一次，那么50次为一个周期  
    var right = data > 25 ? data - 25 : 25 - data; // 亮度为 (|x - 25|) / 25  
    port.brightSet(right / 25 * 255);  
});  
  
//20ms刷新一次渐变颜色，共持续1分钟  
port.on(module.TIMER, 20, 60000, function(count){  
    // 伪代码，target和origin是目标颜色和起始颜色  
    var curRGB = count * (target - origin) / (60000 / 20) + origin;  
    port.colorSet(curRGB);  
});
```

J pwm通道设置操作

```
void pwmChannelSet(channel, dutyCycle);
```

功能描述

系统原生操作5路PWM的接口。

接口约束

无。

参数列表

- channel pwm通道，如下：
- module.CW 表示冷白pwm通道；
- module.WW 表示暖白pwm通道；
- module.R 表示红色pwm通道；
- module.G 表示绿色pwm通道；
- module.B 表示蓝色pwm通道；
- dutyCycle: 占空比，取值范围为[0, 1]。

返回值

无。

接口示例

```
port.pwmChannelSet(module.CW, 0.5);
```

4.2.3.8.3 约束

无。

4.2.3.8.4 样例

介绍

smart LED使用。

例程

```
smartled = require('smartLed');  
var config = { 'connection': smartled.CA, 'period': 10};
```

```
led = smartled.open(config);
led.switchOn();
led.brightSet(255);
led.brightSet(0);
led.cctSet(2700);
led.cctSet(6500);
led.cctSet(255, 200);
led.colorSet(120, 255, 240);
led.colorSet(0xffccee);
led.colorSet(0);
led.colorSet(0xffffffff);
led.staticMode(2800, 0xffffffff, 255);
led.staticMode(255, 255, 0xffffffff, 255);
led.pwmChannelSet(smartled.WW, 0.5);
led.pwmChannelSet(smartled.R, 0.5);
led.pwmChannelSet(smartled.G, 0.5);
led.pwmChannelSet(smartled.B, 0.5);
led.pwmChannelSet(smartled.CW, 1);
led.on(smartled.TIMER, 100, 50000, function(count){
    print(count);
    led.colorSet(0);
});
timer.setInterval(function(){
    led.on(smartled.TIMER, 100, 10000, function(count){
        print("new:", count);
        led.colorSet(100);
    });
}, 5000);
```

4.2.4 拓展模块接口

本章主要介绍MapleJS与上层应用以及底层系统交互模块，主要包括HiLink liteOS以及OTA升级部分。

4.2.4.1 hilink

4.2.4.1.1 介绍

hilink是华为开发的智能家居开放互联平台，hilink模块提供设备侧的hilink接口，使设备具备快速连接到hilink平台的能力。

4.2.4.1.2 模块接口

A 引用模块

```
var hilink = require('hilink');
```

B 初始化参数接口

```
hilink.initParam(devinfo, svcinfo, BI, AC);
```

功能描述

该接口主要是初始化设备信息。

接口约束

参数必须包含hilink规定的字段，具体参见参数列表。

参数列表

- devinfo: 设备信息，可以为json对象，也可以为字符串，但须注意字符串格式必须为‘{ “xx” : “xx”, ... }’，即单引号在最外面，里面的元素必须使用双引号，否则会解析错误。

- **svcinfo**: 设备功能信息，字符串格式为“服务类型”：“服务ID”，具体的设备中svcinfo是固定的。
- **BI**: 与hilink有关参数，字符串类型，由hilink给出。
- **AC**: 与加密相关参数，字符串类型，由hilink给出。

返回值

初始化成功则为0，失败返回报错信息。

接口示例

```
var devinfo={'sn':'','prodId':'1000','model':'iot','dev_t':'000','manu':'000','mac':'','hiv':'1.0.0','fwwv':'1.0.0','hwv':'1.0.0','swv':'1.0.0','prot_t':1};  
var svcinfo="switch,switch;"+"netInfo,netInfo;";  
var BI= "00000000000000000000000000000000"+  
    "3D294E73EE637A1CBC80DB19055F227D"+  
    "5C6D60DB4467A1E223FCABA94CDA4A9D"+  
    "A492700F7C703E2D372F607810F6DC4C"+  
    "3BF2FE6C5D76E32190326D48E2DB4B59"+  
    "B9883D73307BEEC83D566661AACB4CDB"+  
    "3E8A592ECDD72EE0F1CD06E87B5C14E0"+  
    "950C969D5C8C067923CBFDACAA4BD1A1"+  
    "CC76931244C517C34FCF4E213B508944"+  
    "00000000000000000000000000000000"+  
    "89C9EEBC41429A477267704EF081B8C8"+  
    "23EC28165DC4B9D60B5F6A784D8067F2"+  
    "00000000000000000000000000000000"+  
    "71DE4C105AF60925FF6AF031E6EF1AF4"+  
    "3667A64D94CA75A09293B46892B97D44"+  
    "7AD9C92F69A708229E37B8587AE52F7F";  
var AC= "00000000000000000000000000000000"+  
    "7837F9F02FB5B3B99C690A999E039455"+  
    "92DE3268C94247F37B63764A1B007631";  
hilink.initParam(JSON.stringify(devinfo),svcinfo, BI, AC);
```

C 上传接口

```
hilink.upload(svc_id, payload);
```

功能描述

服务字段状态发生改变主动上报到云平台（连接云平台时）或者hilink网关（连接hilink网关时）。

接口约束

由hilink提供参数。

参数列表

- `svc_id`: 服务ID, 字符串格式。
- `payload`: json格式的字段与值。

返回值

上报状态:

- 0为服务状态上报成功
- 非0服务状态上报不成功

接口示例

```
hilink.upload("currentLevel", JSON.stringify(py));
```

D 断开wifi接口

```
hilink.disconnectWIFI();
```

功能描述

断开已经连接上的wifi。

接口约束

无

参数列表

无

返回值

断开状态：

- 0为断开wifi成功
- 非0为断开wifi失败。

接口示例

```
hilink.disconnectWIFI();
```

E hilink端口监听

```
hilink.on(action, func);
```

功能

监听hilink端口的动作，从而调用相应的回调函数。

参数列表

- **event**: 监听的动作类型，GET动作对应于设备状态的上报，是一条上行路径；PUT动作对应于网络命令的下发，是一条下行路径。
- **func**: 回调函数，当监听事件发生时调用该函数。

返回值

无。

接口示例

```
hilink.on(hilink.GET, function(svc_id, instr) {
    var ret = "";
    var pass = {};
    ...
    else if(svc_id == "netInfo"){
        hilink.response(hilink.getNetInfo());
        return(hilink.getNetInfo());
    }
    ret = JSON.stringify(pass);
    print(ret);
    hilink.response(ret);
    return ret;
});
```

4.2.4.1.3 约束

无。

4.2.4.1.4 样例

介绍

某hilink智能设备的示例代码，主要演示了定义用户逻辑的功能。

例程

```
var hilink=require('hilink');
var uart = require('uart');
var config={uartNumber:1, baudRate:9600, dataBits:8, stopBits:1};
var port=uart.open(config);
var
devinfo={'sn':'','prodId':'1000','model':'iot','dev_t':'000','manu':'000','mac':'','hiv':'1.0.0','f
wv':'1.0.0','hwv':'1.0.0','swv':'1.0.0','prot_t':1};
var svcinfo="switch,switch;"+"netInfo,netInfo;";
var BI= "00000000000000000000000000000000"+
      "3D294E73EE637A1CBC80DB19055F227D"+
      "5C6D60DB4467A1E223FCABA94CDA4A9D"+
      "A492700F7C703E2D372F607810F6DC4C"+
      "3BF2FE6C5D76E32190326D48E2DB4B59"+
      "B9883D73307BEEC83D566661AACB4CDB"+
      "3E8A592ECDD72EE0F1CD06E87B5C14E0"+
      "950C969D5C8C067923CBFDACAA4BD1A1"+
      "CC76931244C517C34FCF4E213B508944"+
      "00000000000000000000000000000000"+
      "89C9EEBCA1429A477267704EF081B8C8"+
      "23EC28165DC4B9D60B5F6A784D8067F2"+
      "00000000000000000000000000000000"+
      "71DE4C105AF60925FF6AF031E6EF1AF4"+
      "3667A64D94CA75A09293B46892B97D44"+
      "7AD9C92F69A708229E37B8587AE52F7F";
var AC= "00000000000000000000000000000000"+
      "7837F9F02FB5B3B99C690A999E039455"+
      "92DE3268C94247F37B63764A1B007631";
hilink.initParam(JSON.stringify(devinfo),svcinfo, BI, AC);

port.on(uart.DATA, 9, function(data) {
    var val;
    ...
    if(type==0x01 && cmd== 0xF0){
        hilink.disconnectWiFi();
        ...
    }
    else if(type == 0x03){
        py.switch=val;
        hilink.upload("switch",JSON.stringify(py));
    }
});
hilink.on(hilink.GET,function(svc_id,instr){
    var ret = "";
    var pass = {};
    ...
    else if(svc_id == "netInfo"){
        hilink.response(hilink.getNetInfo());
        return(hilink.getNetInfo());
    }
    ret = JSON.stringify(pass);
    print(ret);
    hilink.response(ret);
    return ret;
});
hilink.on(hilink.PUT,function(svc_id,payload){
    var pass=JSON.parse(payload);
    var perc=0;
    if(svc_id == "targetLevel")
        ...
        hilink.response(0);
    return 0;
});
```

4.2.4.2 ota

4.2.4.2.1 介绍

OTA模块用于JS脚本和MCU的在线升级。JS脚本的升级过程不需要用户参与，MCU的升级需要用户实现提供的接口共同完成。

4.2.4.2.2 模块接口

A MCU升级

在MCU的升级过程中，需要用户在JS脚本中注册相应的hilink回调函数，以明确MCU在升级过程中面对各种事件需要触发的动作，比如收到数据长度，数据块，哈希值时应当怎么处理这些数据。比如应当注册如下回调函数来处理收到的数据块：

```
hilink.on (hilink.OTA_CHUNK, function (name, data)
{
    print ("mcu file name:", name);
    //process data
    return {"error":0}; //返回数据处理结果，0代表正常
});
```

用于注册的回调函数的接口hilink.on有2参数，第一个参数表明升级过程中的事件，第二个参数是回调函数

a.事件类型

在升级MCU时需要用户全部注册下面四个事件的回调函数：

- hilink.FILELIST_COMPONENT：传递MCU的版本号
- hilink.OTA_START：传递MCU的大小；
- hilink.OTA_CHUNK：可能多次调用，分块传递MCU的数据；
- hilink.OTA_END：传递MCU的升级的状态，hash值来结束此次升级；

b.回调函数参数以及返回值

- FILELIST_COMPONENT 对应的函数原型为 Object function (name, version)
version: 字符串类型，表示MCU版本。在IAR版本中第二个参数是一个对象，有version这个属性。
返回值应当是一个对象，包含属性update(number)，代表升级与否(1代表升级)。
- OTA_START 对应 object function (name, len)
len:number类型，表示MCU长度。
返回值是一个对象，包含属性error,表示数据处理过程中是否出错，0代表无错，-1表示有错。
- OTA_CHUNK 对应 object function (name, data)
data: Buffer类型，表示MCU升级数据。
返回值是一个对象，包含属性error表示数据处理过程中是否出错，0代表无错，-1表示有错。
- OTA_END 对应 object function (name, state, hash)
state: number类型，表示升级状态(0表示正常)。 hash: MCU的hash值，buffer类型，32字节，目前仅支持sha256。
返回值是一个对象，有一个属性error表示是否出错，0代表无错，-1表示有错。

以上参数中的name(string)均代表升级的MCU部件名称。在现在只有一个MCU的情况下，默认是mcu_ota_all.bin。

B 版本信息查询

在hilink模块中，有两个接口用于用户对当前版本信息进行查询，分别是getVersion和getJSVersion，前者获取整体版本号，后者获取JS脚本版本号，均是string类型。

4.2.4.2.3 约束

1. 由于内存受限的原因，目前基于文件系统的版本不支持OTA升级，仅基于Flash的版本支持。
2. 目前允许升级的最大JS脚本为6k。
3. JS升级不需要注册以上回调函数，MCU升级需要注册全部4个回调函数，如果只注册部分，则回调函数无法使用。同时只要升级包里有MCU，则需要注册回调函数。
4. JS,MCU如果都需要升级，则顺序为先JS,后MCU。如果JS升级成功，会保存JS版本号到FLASH,可以通过getJSVersion读到；之后会用MCU升级会用新的JS脚本进行升级。
5. 只有升级整体成功，即要升级的部件全部升级成功，才会将整体版本号保存，否则通过getVersion读到版本号跟之前一致，不会变化。
6. Debug版本暂不支持OTA升级。Release和Release_Parser中的pl-cnv版本仅支持JS文件(或者snapshot文件)升级，Release和Release_Parser中的full版本支持JS文件(或者snapshot文件)和MCU升级。

4.2.4.2.4 样例

介绍

- 1.获取整体版本号和JS脚本版本号。
- 2.MCU升级过程。

例程

```
var hilink = require('hilink');

print(hilink.getVersion()); //整体版本号
print(hilink.getJSVersion()); //JS脚本版本号

hilink.on(hilink.FILELIST_COMPONENT, function(name, version){
    print("mcu name:", name);
    //process version.
    update = 1;
    return {"update":update};
})

hilink.on(hilink.OTA_START, function(name, len){
    //process mcu length
    return {"error":0};
})

hilink.on(hilink.OTA_CHUNK, function(name, buf){
    //process mcu data
    return {"error":0};
})

hilink.on(hilink.OTA_END, function(name, state, hash){
    if (state != 0){
        //inform mcu the wrong state
    }
    else{
        //process hash value
    }
})
```

```
    }  
    return {"error":0};  
  })
```

4.2.4.2.5 升级包

升级包制作

ota-package-making-tool.py是用于制作升级包的工具，需要python3运行。

用户需要提供升级包的整体版本号(必选), js脚本(或者snapshot)及其版本号，mcu文件及其版本号，至少提供一个文件。

版本号长度均限制在64个字节内。

参数说明：

1. -v或--version，指定整体版本号。
2. -j或--js，指定JS文件；-jv或--js-version指定JS版本号。
3. -m或--mcu，指定MCU文件；-mv或--mcu-version指定MCU版本号。

样例：

```
python3 ota-package-making-tool.py -v V2.0 -j js_file.js -jv V1.1 -m mcu.bin -mv V1.2
```

成功运行后输出一个二进制文件mcu_ota_all.bin.

filelist.json样例

```
{  
  "version": "V1.0",           //整体版本号，要与制作升级包时的整体版本号一致  
  "mcu_ota_all.bin":          //制作的升级包  
  {  
    "hash": "sha256",          //升级包的哈希算法  
    "value": "123456789..." //升级包的哈希值  
  }  
}
```

注：IAR版本不需要制作升级包，filelist.json如下：

```
{  
  "version": "1.0",           //overall version  
  "file":                    //component name  
  {  
    "version": "js_1001"      //component name  
    "hash": "sha256",  
    "value": "123456789...",  
    "user_hash": "MD5",  
    "user_value": "22345346..."  
  },  
  "mcu_ota_all.bin":          //component name  
  {  
    "version": "rtlambaz-fw0100" //component version  
    "hash": "sha256",  
    "value": "123456789...",  
    "user_hash": "MD5",  
    "user_value": "22345346..."  
  }  
}
```

4.2.4.3 system

4.2.4.3.1 介绍

system模块允许用户通过调用API来查询引擎和操作系统的堆内存使用状况，以及对board进行复位和恢复出厂设置等操作。使用system模块首先要先获得该模块句柄：var system = require ('system');。

4.2.4.3.2 模块接口

A 引用模块

```
var system = require ('system');
```

B 查询堆内存

```
system.engineHeap (type);  
system.osHeap (type);
```

功能描述

引擎的可分配内存可以是一个静态编译的数组，也可以是通过调用操作系统的接口来分配的系统堆内存，通过一个宏来控制，默认使用静态数组。

查询引擎堆内存使用情况：

```
system.engineHeap (type);
```

查询操作系统堆内存使用情况：

```
system.osHeap (type);
```

接口约束

使用引擎内存查询接口时应当注意，除了debug版本外，其他发布的版本只支持查询TOTAL和USED两种类型，如果查询，则只会返回undefined。

内存最大块：OS内存和引擎内存都是以链表的方式来组织的，内存是分为多个块，通过指针来连接，所谓内存最大块是指内存最大的那个块，表示一次申请内存不能超过的最大值。

浪费内存大小：由于在申请引擎的堆内存时，是按8字节对齐的，所以多余申请的字节即为浪费的内存。

参数列表

- type: 要查询内存信息的类型，可选类型如下：
 - system.TOTAL: 总内存大小，engineHeap和osHeap均支持。
 - system.USED: 已使用内存大小，engineHeap和osHeap均支持。
 - system.MAX_BLOCK: 内存最大块的大小，engineHeap和osHeap均支持。
 - system.ALLOC_COUNT: 内存分配次数，engineHeap和osHeap均支持。
 - system.FREE_COUNT: 内存回收次数，engineHeap和osHeap均支持。
 - system.PEAK_ALLOC: 内存使用峰值，仅engineHeap支持。
 - system.WASTE: 内存浪费大小，仅engineHeap支持。
 - system.PEAK_WASTE: 内存浪费峰值，仅engineHeap支持。
 - system.GC: 内存GC情况，仅engineHeap支持。
 - system.FREE: 已回收的内存大小，仅osHeap支持。

返回值

在以上类型中，除GC外，返回值均为number类型；GC返回值类型为object，含有两个属性，一个是size，代表总共GC的内存大小，一个是count，代表GC次数。

接口示例

```
var system = require ('system');  
var gc = system.engineHeap (system.GC);  
var gc = system.osHeap (system.TOTAL);
```

C 重启

```
system.reset ();
```

功能描述

重启系统。

接口约束

无。

参数列表

无。

返回值

无。

接口示例

```
var system = require ('system');  
system.reset ();
```

D 恢复出厂设置

```
system.restore ();
```

4.2.4.3.3 约束

无。

4.2.4.3.4 样例

介绍

下面是system模块的使用样例，包括了内存查询，与恢复出厂设置。

例程

```
/* 内存查询 */  
var system = require ('system');  
print ("total heap size of engine:" + system.engineHeap (system.TOTAL));  
var gc = system.engineHeap (system.GC);  
print ("engine gc times:" + gc.count + ", heap size freed by gc:" + gc.size);  
print ("os available heap size:" + (system.osHeap (system.TOTAL) - system.osHeap (system.USED)));  
  
/* 恢复出厂设置 */  
var hilink = require ('hilink');  
hilink.disconnectWiFi () /* 断开wifi连接 */  
system.restore () /* 恢复出厂设置 */  
system.reset () /* 重启 */
```

5 调试

本章介绍调试功能。

该调试功能仅限用户在开发调试阶段使用，依赖串口与调试端口连接，产品发布后由于在现网中调试端口关闭，所以功能无法使用，不涉及网上安全问题。

注意：不支持对Bytecode文件的调试。当前本版不支持对含有回调函数的JS脚本的调试。

5.1 安装

5.2 使用选项

5.3 指令

5.1 安装

当前版本支持Windows和Linux操作系统，Python版本 2.7 (不支持Python 3)

```
# 仅Windows平台下需要安装pywin32
pip install pywin32

# Windows和Linux平台均需安装pyserial
pip install pyserial
```

5.2 使用选项

uart 选项

该选项必须指定，其值为板子连接的串口号，对大小写敏感，Windows平台下形如COM4，Linux平台下形如/dev/ttyS3

```
$ python maple-client-serial.py --uart=<port number>
>>>>Reboot your board first!<<<
Waiting for UART connection...
```

help 选项

可选，可以列出客户端支持的所有选项，查看每个选项对应的格式和描述

```
$ python maple-client-serial.py --help
usage: maple-client-serial.py [-h] [--uart UART] [--upload UPLOAD] [-v]
                             [--non-interactive] [--color]
                             [--display DISPLAY] [--exception {0,1}]
```

```
JerryScript debugger client optional arguments:
-h, --help          show this help message and exit
--uart UART         specify a COM port
--upload UPLOAD      specify a filename and a COM port via --uart=port
-v, --verbose        increase verbosity (default: False)
--non-interactive    disable stop when newline is pressed (default: False)
--color             enable color highlighting on source commands (default:
False)
--display DISPLAY    set display range
--exception {0,1}    set exception config, usage 1: [Enable] or 0: [Disable]
```

verbose 选项

可选，可以打印出调试过程中客户端的debug信息，默认为False

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --verbose
DEBUG: Debug logging mode: ON
Waiting for UART connection...
```

non-interactive 选项

可选，控制程序运行过程中是否可被打断，默认为False，即可以打断

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --non-interactive
```

color 选项

可选，控制客户端输出信息的颜色显示，默认为False

注意：color 选项在 cmd 和 PowerShell 中会导致显示异常

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --color
```

display 选项

可选，控制调试过程中是否显示源码，其值为显示当前行前后源码的行数

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --display=10
```

exception 选项

可选，配置是否显示Exception信息，其值为 0 或 1，默认为 0

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --exception=1
```

upload 选项

可选，其值为部署到板子里的文件，部署完成后，程序退出。

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --upload=./hello.js
-----
Upload JS with content:print("Hello, world!");
-----
Finished writing source code to COM port!
Finished restart device through COM port!
```

5.3 指令

help 指令

打印帮助信息，无参时打印所有支持的命令，后面可跟参数为调试器支持的命令，打印对应命令的帮助信息

```
(maplejs-debugger) help
```



```
Documented commands (type help <topic>):
```

```
=====
abort      bt      display  exception  list      next      source
b          c          dump     f          memstats  quit      src
backtrace  continue e        finish    ms        s         step
break      delete  eval     help      n         scroll    throw
```

eval 指令

执行JavaScript源代码，必须跟参数，为JavaScript表达式，返回表达式的计算结果

```
(maplejs-debugger) eval count
1
(maplejs-debugger) eval count+1
2
```

e 指令

同 eval指令

throw 指令

抛出异常

```
(maplejs-debugger) throw new Error("Throwing an exception")
Stopped at trycatch.js:7Source: trycatch.js
1  try
2  {
3    print("In try block");
4  }
5  catch(err)
6  {
7 >   print("In catch block");
8  }
9  finally {
10   print("In finally");
11 }
(maplejs-debugger)
```

abort 指令

抛出异常

```
(maplejs-debugger) abort new Error("Fatal error")
err: Script Error: Error: Fatal error
```

memstats 指令

显示内存使用信息

```
(maplejs-debugger) memstats
Allocated bytes: 0
Byte code bytes: 0
String bytes: 0
Object bytes: 0
Property bytes: 0
```

ms 指令

同 memstats指令

display 指令

设置最多一次显示源码的行数

```
(maplejs-debugger) display 20
```

scroll 指令

向上或向下滚动源码，w+回车 向上，s+回车 向下，q+回车退出，可与 display 配合使用

```
(maplejs-debugger) scroll  
w  
s  
q
```

dump 指令

打印所有调试数据

```
(maplejs-debugger) dump  
{325: Function(byte_code_cp:0x145, source_name:'string-load', name:'', line:1, column:1 { string-load:1 })}
```

backtrace 指令

回溯堆栈，列出当前函数调用关系

```
(maplejs-debugger) backtrace
```

bt 指令

同 backtrace

step 指令

执行下一条指令，遇到函数会进入函数内部执行，不接受参数

```
(maplejs-debugger) step
```

s 指令

同 step

next 指令

可跟一个参数，默认为1，执行下n条指令，不会进入函数体

注意：该命令的参数必须为大于0的正整数，且当其大小大于当前函数的最大行数时，该参数并不会实际作为next命令的执行次数，取而代之的是当前函数的最大行数

```
(maplejs-debugger) next  
(maplejs-debugger) next 3
```

n 指令

同 next

finish 指令

继续运行直到当前函数返回，不接受参数

```
(maplejs-debugger) finish
```

f 指令

同 finish

continue 指令

继续执行程序，直到遇到下一个断点或者执行结束

```
(maplejs-debugger) continue
```

c 指令

同 continue

break 指令

此指令用来在文件中设置断点，目前支持两种方式：

- 通过函数名设置断点：执行到对应函数处暂停执行
- 通过文件名+行号设置断点：执行到对应行时暂停执行

断点分为Active和Pending两种，Pending类型的断点一般是在空行、不存在的函数等处，不会影响文件执行

```
(maplejs-debugger) break main          /* 根据函数名设置断点 */  
(maplejs-debugger) break ping.js:10    /* 根据文件名和行号设置断点 */
```

b 指令

同 break指令

list 指令

列出当前所有断点，不接受参数

```
(maplejs-debugger) list
```

delete 指令

删除指定断点，参数可为索引、all、active、pending，索引为可通过list指令查看

```
(maplejs-debugger) delete 1
```

source 指令

显示当前行前后n行的源码

```
(maplejs-debugger) source 5
```

src 指令

同 source

exception 指令

配置异常处理程序模块，可以接收的参数为0、1，分别为关闭和开启

```
(maplejs-debugger) exception 0
```

quit 指令

退出串口调试器

注意：当程序运行结束时，必须通过Ctrl+C组合键退出调试器

```
(maplejs-debugger) quit
```

6 常见问题

本章介绍常见问题。

1. **MapleJS引擎目前分配的堆内存为10KB，由于引擎本身消耗一定内存，所以JS应用实际可使用内存小于该值，当前测试可使用内存估计为8KB左右。**
2. **回调与循环的陷阱**

引擎采用事件队列运行机制（同Node.js）：先执行当前运行的脚本，当前脚本执行完后依次从事件队列中取下一个事件执行。引擎的timer.setInterval、timer.setDelay、gpio.on、uart.on都是异步接口，当循环中嵌套这些异步接口的回调函数时会遇到一些“问题”。看下面的一个例子：

```
var time = require('timer');
var tid = new Array(2)
var j = 1;

function startTimer() {
    print("this is " + j + " cycle!");
    j++;
}

for(var i=0; i<2; i++) {
    print("start timer " + i);
    tid[i] = time.setInterval(startTimer, 100);
}

time.setDelay(120);

for(var i=0; i<2; i++) {
    time.stopTimer(tid[i]);
    print("stop timer " + i);
}
```

下面是实际运行起来的输出结果：

```
start timer 0
start timer 1
stop timer 0
stop timer 1
this is 1 cycle!
this is 2 cycle!
```

从运行结果我们发现startTimer在stoptimer之后执行，这是因为引擎采用事件队列运行机制，即引擎首先执行上面的JS脚本，执行完第一个for循环后，引擎会sleep 120 ms，第一个for循环执行完100ms后，tid[0]和tid[1]会向事件队列中发送2个startTimer事件，这两个事件需要等到上面的JS脚本执行完之后才可以执行，所以引擎在120 ms之后会继续运行第二个for循环。当执行完上面的脚本后，引擎从事件队列中取下一个事件执

行。因为现在事件队列中已经有两个startTimer事件，所以它们会依次执行. 最终得到上面的运行结果.

3.死循环问题

如果JS脚本有死循环，会导致当前JS脚本一直在引擎上运行，无法运行下一个事件。所以不建议使用死循环以及运行时间比较长的操作。建议使用异步的方式来代替死循环。

如果JS脚本有死循环，而且存在停止执行当前脚本的需求，可通过IDE工具或命令行工具中提供的 stopEngine / stopScript 功能来实现（仅适用于循环结构）。

注意：向设备发送的stop指令并不会立即得到执行，而是执行到循环语句时才能停止。

7 约束与限制

本章介绍MapleJS引擎的限制与约束

函数调用参数个数限制

函数调用参数的个数限制，MapleJS引擎支持函数调用的参数的最大个数为255，当函数的调用的入参个数大于255时，会出现：

Script Error: SyntaxError: Maximum number of function arguments reached

函数定义参数个数限制

函数定义参数的个数限制，MapleJS引擎支持函数定义的参数的最大个数为256，当函数定义的入参个数大于256时，会出现：

Script Error: Maximum number of registers is reached

字符串长度限制

MapleJS引擎支持字符串的长度最大为65535，当字符串的长度超过65535时，会出现：

Script Error: SyntaxError:String is too long

文件源码最大字节数限制

MapleJS引擎支持加载js文件的最大字节数为1048576，当源码字符超过最大限制后，后续的字符会被丢弃。

引用计数限制

MapleJS引擎支持对象的最大引用次数为1023，当引用次数超过最大限制后，会引起引擎重启。

引用计数（reference count）增减的基本规则是：

1. 当声明了一个变量并将一个引用类型值赋给该变量时，则该值的引用次数就是1；
2. 如果同一个值又被赋给另一个变量，则该值的引用次数加1；
3. 如果包含对该值引用的变量又取得了另外一个值，则该值的引用次数减1。

栈使用限制

默认MapleJS应用可使用的栈空间为3KB，超过限制会提示：Potential stack overflow, leaving only xx bytes of stack space!

Timer使用限制

定时器的事件触发频率太高会导致事件队列溢出，报错：Fail to push event to event queue. 当出现队列溢出时可做出以下调整：

1. 降低定时器触发事件的频率；
2. 缩短数据的处理时间；
3. 调大Queue的数量。

做上面的几个调整时，需要遵循以下原则：

1. 每个事件的处理时间要小于2个监听事件之间的间隔时间；
2. 需要处理的事件数量要小于当前空闲queue的数量。