

MapleJS  
V1.0

# 开发指南

文档版本 01  
发布日期 2019-04-29



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](mailto:support@huawei.com)

客户服务电话：4008302118

# 目 录

<b>1 简介.....</b>	<b>1</b>
<b>2 环境准备及工具使用.....</b>	<b>2</b>
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
<b>3 语法规格.....</b>	<b>11</b>
3.1 语法标准.....	11
3.2 裁剪规格.....	11
<b>4 模块接口.....</b>	<b>19</b>
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.....	24
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.....	30
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.....	52
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.....	53
4.1.11.3 约束.....	55
4.1.11.4 样例.....	55
4.2 拓展模块接口.....	56
4.2.1 hilink.....	56
4.2.1.1 介绍.....	56
4.2.1.2 模块接口.....	56
4.2.1.2.1 引用模块.....	56
4.2.1.2.2 初始化参数接口.....	57
4.2.1.2.3 上传接口.....	58
4.2.1.2.4 断开 wifi 接口.....	58
4.2.1.2.5 hilink 端口监听.....	59

4.2.1.3 约束.....	59
4.2.1.4 样例.....	59
4.2.2 ota.....	60
4.2.2.1 介绍.....	60
4.2.2.2 模块接口.....	60
4.2.2.2.1 MCU 升级.....	60
4.2.2.2.2 版本信息查询.....	61
4.2.2.3 约束.....	61
4.2.2.4 样例.....	62
4.2.2.5 升级包.....	62
4.2.3 system.....	63
4.2.3.1 介绍.....	63
4.2.3.2 模块接口.....	63
4.2.3.2.1 引用模块.....	63
4.2.3.2.2 查询堆内存.....	63
4.2.3.2.3 重启.....	64
4.2.3.3 约束.....	65
4.2.3.4 样例.....	65
<b>5 调试.....</b>	<b>66</b>
5.1 安装.....	66
5.2 使用选项.....	66
5.3 指令.....	67
<b>6 常见问题.....</b>	<b>71</b>
<b>7 约束与限制.....</b>	<b>73</b>

# 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 : apa102      2 : canvas      3 : crypto      4 : dht11      5 : fs          6 : gpio
7 : hilink      8 : i2c         9 : infrared    10 : keyboard   11 : ky040      12 : lattice
13 : lcd1602    14 : ledRGB     15 : liquid     16 : mpu6050    17 : objectPersistence 18 : pwm
19 : relays     20 : rtc        21 : senseair_s8 22 : sensor     23 : spi        24 : system
25 : tcs3200    26 : timer      27 : uart       28 : servo      29 : stepper     30 : nanostepper
31 : smartLed   32 : ultrasound 33 : thermocouple 34 : shock
Please Input the Numbers of Modules You Want to Load (Seperated by Comma):
```

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26,3,20。如若想要选择默认的组合（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)



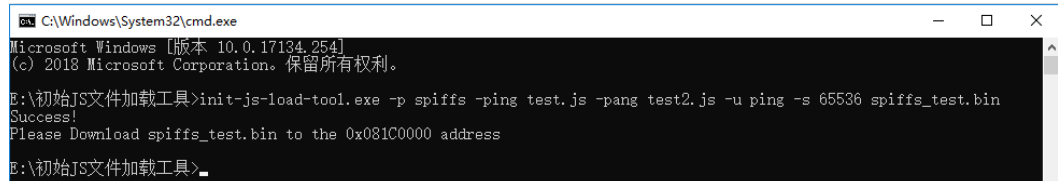
```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.17134.254]
(c) 2018 Microsoft Corporation。保留所有权利。
E:\初始JS文件加载工具>
```

**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：

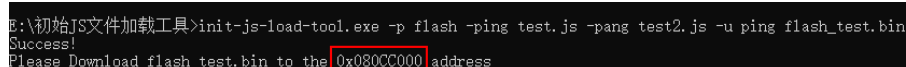
```
Success!
Please Download XXXX.bin to the 0xXXXX address
```

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

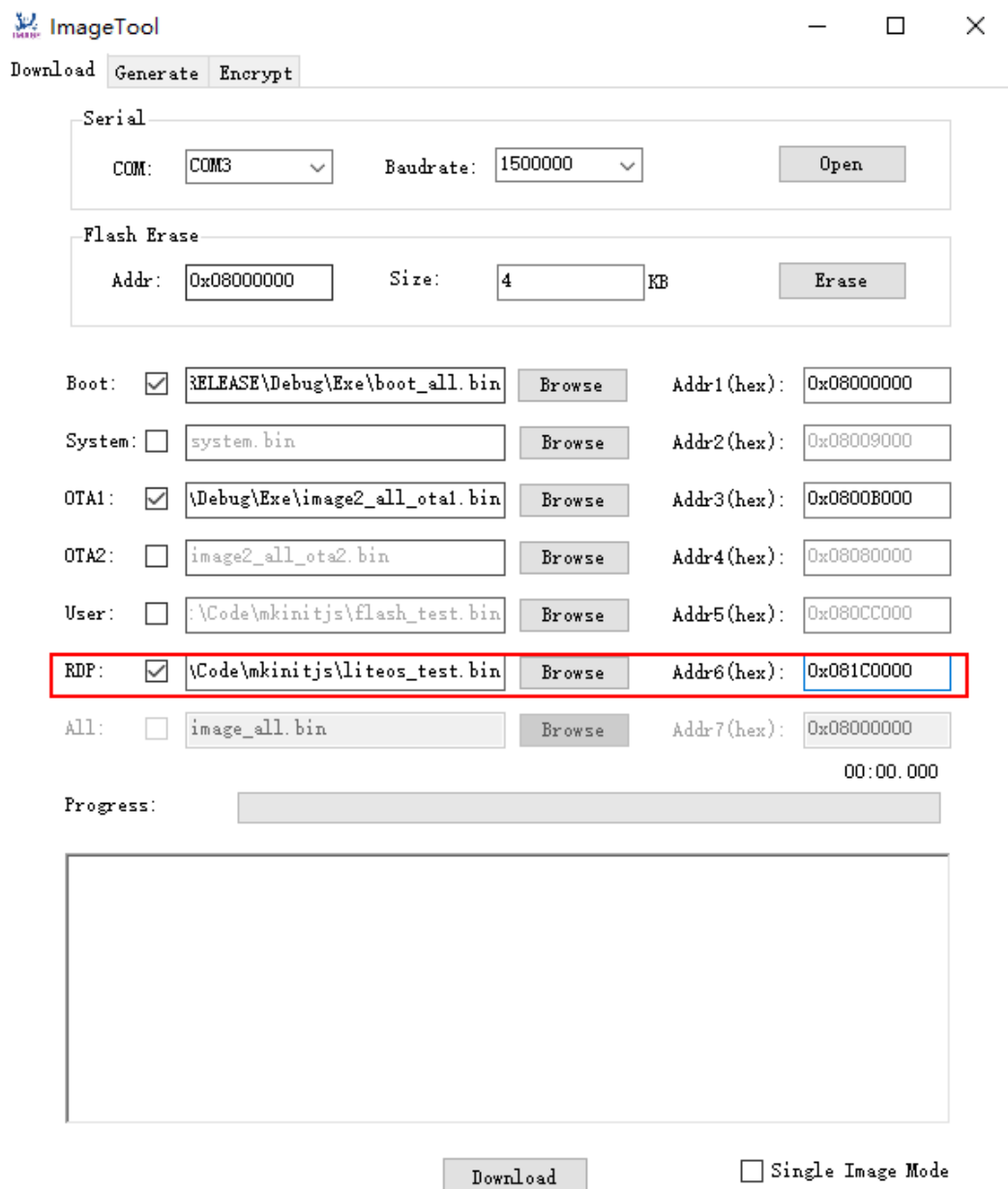
**步骤3** 选择需烧录的文件及输入地址信息。

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.exe
    └── MapleJS-snapshot.exe
```

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.exe 为Windows下将JS文件加载至bin文件的工具

MapleJS-snapshot.exe 为Windows下将JS源码编译成字节码的工具

## 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mian.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，需返回为一个json字符串  
     * 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','fw':'1.0.0','hw':'1.0.0','sw':'1.0.0','prot_t':1},  
  'switch1,binarySwitch;switch2,binarySwitch;switch3,binarySwitch;switch4,binarySwitch;switch0n,binarySwitch;switchOff,binarySwitch;bb,bb_custom;nn,nn_custom;',  
  '9D436E2CE869018C6D9B7A6341B942CF8527939EA3D1A29A1C8993C9C94F05D9DD217DA917A4DCFCB291FF9ED837C9DBD945E22D626C4D4B69B7507C2A35B458C523C178D9F0AE2FBF50C9E691F3E7FFECEAE71074C75D1F2D3B55408A451AF0EFC8074947CCA8BEED6248F9E3D0C5B48B24F9DA819E5532C51BC38D0669483BD37357B99C0F27C9CE4A29B844F3D93E0BC39064C224CD4768D1E1889BBE09E1D077270A644A5C7D325AAB282AE53B942EF1EBECE2094BC3087FBD0FD780AEEBF961A326423AF4665071FC2A9159128235AE2298EF1DE5BABAC74E6AD4DB0642B345CDDAD2376DEAAEA2C064100F42D48DB0EDBFD0B4E89229CDD8953F0F9A2',
```

```
'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。

另外，MapleJS还拥有丰富的行业硬件使能库，具体可参考：<https://github.com/HWMMapleJS/MapleJS>。

## 4.1 基础模块接口

### 4.2 拓展模块接口

## 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  $0 \leq \text{value} \leq 255(0xff)$ 。
- **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的整数值; 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的整数值。
- 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: wwwwwwwww
```

## 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, "hellowold!!!");
```

### 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_addr /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**。

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

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

A terminal window with a black background and white text. The text shows the output of a JavaScript program: "JS exec start.", "write success.", "read success.", "led", "110", "120", "130", "remove success.", "it's already removed.", and "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 $\mu\text{s}$ 。
  - `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20000 $\mu\text{s}$  (20ms),占空比是0(20000 $\mu\text{s}$ 开,0 $\mu\text{s}$ 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10000 $\mu\text{s}$  (10ms),占空比是0.5(5000 $\mu\text{s}$ 开,5000 $\mu\text{s}$ 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 $\mu\text{s}$ 。
  - `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

- timerID : 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IO18 (即18),发送引脚为IO23 (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 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。

zh-cn\_topic\_0168574280.xml

zh-cn\_topic\_0168574291.xml

zh-cn\_topic\_0168574300.xml

### 4.2.1 hilink

#### 4.2.1.1 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。

#### 4.2.1.2 模块接口

##### 4.2.1.2.1 引用模块

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

#### 4.2.1.2.2 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。以下是Hilink设备信息的一个示例：

```
var
devinfo={ 'sn': '', 'prodId': '1000', 'model': 'iot', 'dev_t': '000', 'manu': '000', 'mac': '', 'hiv': '1.0.0', 'fwv': '1.0.0', 'hwv': '1.0.0', 'swv': '1.0.0', 'prot_t': 1 };

typedef struct {
    char* sn;           /**<设备唯一标识，比如sn号，长度范围 (0, 40]*/
    char* prodId;       /**<设备HiLink认证号，长度范围 (0, 5]*/
    char* model;        /**<设备型号，长度范围 (0, 32]*/
    char* dev_t;        /**<设备类型，长度范围 (0, 4]*/
    char* manu;         /**<设备制造商，长度范围 (0, 4]*/
    char* mac;          /**<设备MAC地址，固定32字节*/
    char* hiv;          /**<设备Hilink协议版本，长度范围 (0, 32]*/
    char* fwv;          /**<设备固件版本，长度范围[0, 64]*/
    char* hwv;          /**<设备硬件版本，长度范围[0, 64]*/
    char* swv;          /**<设备软件版本，长度范围[0, 64]*/
    int prot_t;         /**<设备协议类型，取值范围[1, 3]*/
} dev_info_t;
```

- **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', 'fwv': '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" +
    "3E8A592ECD72EE0F1CD06E87B5C14E0" +
    "950C969D5C8C067923CBFDACAA4BD1A1" +
    "CC76931244C517C34FCF4E213B508944" +
    "00000000000000000000000000000000" +
    "89C9EEBC41429A477267704EF081B8C8" +
    "23EC28165DC4B9D60B5F6A784D8067F2" +
    "00000000000000000000000000000000" +
    "71DE4C105AF60925FF6AF031E6EF1AF4" +
    "3667A64D94CA75A09293B46892B97D44";
```

```
"7AD9C92F69A708229E37B8587AE52F7F";  
var AC= "00000000000000000000000000000000"+  
"7837F9F02FB53B99C690A999E039455"+  
"92DE3268C94247F37B63764A1B007631";  
hilink.initParam(JSON.stringify(deinfo),svcinfo,BI,AC);
```

#### 4.2.1.2.3 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));
```

#### 4.2.1.2.4 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();
```

#### 4.2.1.2.5 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;
});

```

### 4.2.1.3 约束

无。

#### 4.2.1.4 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', 'fw': '1.0.0', 'hwv':'1.0.0', 'swv':'1.0.0', 'prot_t':1};
var svcinfo="switch, switch:";
var BI= "00000000000000000000000000000000"+
"3D294E73EE637A1CBC80DB19055F227D"+
"5C6D60DB4467A1E223FCABA94CDA4A9D"+
"A492700F7C703E2D372F607810F6DC4C"+
"3BF2FE6C5D76E32190326D48E2DB4B59"+
"B9883D73307BEEC83D566661AACB4CDB"+
"3E8A592ECDD72EE0F1CD06E87B5C14E0"+
"950C969D5C8C067923CBFDACAA4BD1A1"+
"CC7693124C517C34FCF4E213B508944"+
"00000000000000000000000000000000"+
"89C9EEBC41429A477267704EF081B8C8"+
"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;
    /* more codes. */
    if(type==0x01 && cmd== 0xF0){
        hilink.disconnectWiFi();
    /* more codes. */
    }
    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 = {};
    /* more codes. */
    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")
/* more codes. */
hilink.response(0);
return 0;
});
```

#### 4.2.2 ota

### 4.2.2.1 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。

#### 4.2.2.2 模块接口

#### 4.2.2.2.1 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。

### 4.2.2.2 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。

### 4.2.2.3 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。

#### 4.2.2.4 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};
})
```

#### 4.2.2.5 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..."
  }
}
```

## 4.2.3 system

### 4.2.3.1 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');。

### 4.2.3.2 模块接口

#### 4.2.3.2.1 引用模块

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

#### 4.2.3.2.2 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);
```

## 4.2.3.2.3 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 ();
```

## 4.2.3.3 约束

无。

## 4.2.3.4 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

```
1.1.1.1.1 /* 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。