int Scheduler::add_controll_points(const bca_res_t list_res[], bca_size_t list_res_size) —— 绑定数据点
int Scheduler::start() —— 建立连接
int Scheduler::stop() —— 断开连接
int Scheduler::report(…) —— 上报数据
int Scheduler::send_content(const void* data, size_t size, bool eof) —— 发送语音数据
int Scheduler::response(const bca_msg_t* req, int msg_code, const char* payload) —— 响应云端的请求
int Scheduler::clear_content() —— 停止上传并清理未上传数据

5.2 HttpClient类

void register_data_handler(data_out_handler_cb data_hdlr_cb, void* p_usr_ctx) —— 注册一个回调函数，通过该函数处理下载到的数据
void register_notify_call_back(check_stop_notify_cb_t chk_stp_cb) —— 注册检查停止下载的回调函数
int get(const char* url, const size_t pos) —— 请求下载数据
void get_download_progress(size_t* total_size, size_t* recv_size) —— 获取下载进度
Http模块使用示例代码

5.3 MediaManager类

bool initialize(…) —— 初始化MediaManager
MediaPlayerStatus play_url(const char* url, bool is_speech, bool is_continue_previous) —— 播放网络媒体文件
MediaPlayerStatus play_local(const char* path) —— 播放本地文件
MediaPlayerStatus pause_or_resume() —— 暂停/播放
MediaPlayerStatus stop() —— 停止
MediaPlayerStatus stop_completely() —— 完全停止
MediaPlayerStatus get_media_player_status() —— 获取播放状态
int register_listener(MediaPlayerListener* listener) —— 注册播放状态的监听者
int unregister_listener(MediaPlayerListener* listener) —— 注销监听者
void set_volume(unsigned char vol) —— 设置音量
unsigned char get_volume() —— 获取音量
void get_download_progress(size_t* total_size, size_t* recv_size) —— 获取http模块的下载进度
void seek(int position) —— 调整播放进度

5.4 RecorderManager类

int start() —— 开始录音
int stop() —— 停止录音
int set_listener(Recorder::IListener* listener) —— 注册监听者
Recoder模块使用示例代码

5.5 PlayCommandManager类

int next() —— 播放下一曲
int previous() —— 播放上一曲
int repeat() —— 循环播放

# 设备端开发手册

# 1. 前言

本手册的编写，是基于开发者对mbed-os的开发有基本的了解。否则，请开发者先去mbed-os开发者网站学习一下。

# 2. 开发环境

| 平台 | 名称 |
| --- | --- |
| 操作系统 | Windows 7/10等 |
| 硬件平台 | RDA 5991H |
| 代码管理工具 | Git |
| 编译工具 | mbed CLI |
| 串口调试工具 | xshell，putty等 |
| 烧录工具 | RDA5981 单口下载工具 |

# 2.1 mbed CLI简介

mbed CLI是ARM提供的mbed命令行编译工具，下面将对它做简单的介绍，详细介绍请参照mbed-os官网。

# 依赖工具

- **Python** 要求使用Python2.7.11以上版本，不兼容Python3
- **Git 和 Mercurial** mbed CLI同时支持Git和Mercurial 的代码库，所以两者都需要
  - Git - 支持1.9.5版本及以上.
  - Mercurial - 支持2.2.2版本及以上.

- **编译器或IDE工具链** DuerOS Light SDK是mbed-os工程，目前仅支持ARM Compiler 5工具链。可以安装ARM Compiler 5编译器或集成了ARM Compiler 5编译器的IDE来进行编译。
  - 编译器: ARM Compiler 5
  - IDE: Keil uVision, DS-5, IAR Workbench.

mbed CLI的依赖工具请自行安装好，可参照ARM提供的视频教程
**注意：** 安装工具时注意添加环境变量Path。安装成功后应该能够通过cmd窗口使用工具。

# 安装mbed CLI

可以安装最新稳定版本

```
pip install mbed-cli
```

也可以安装开发版本

```
git clone https://github.com/ARMmbed/mbed-cli
python setup.py install
```

# 卸载mbed CLI

```
pip uninstall mbed-cli
```

# 配置编译环境ARM_PATH

以Keil uVision 5为例，配置工具链路径。
`"C:\Keil_v5\ARM\ARMCC"` 是Keil uVision 5的默认安装路径，不是默认安装时请自行更改。

```
mbed config -G ARM_PATH "C:\Keil_v5\ARM\ARMCC"
```

**注意**：Keil uVision 5请注册后使用。

# 创建工程

创建名为my-mbed的工程

```
mbed new my-mbed
```

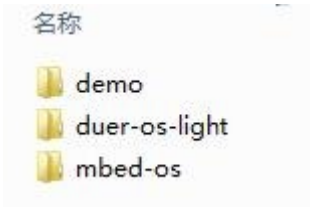
# 导入工程

导入mbed的一个范例工程

```
mbed import https://github.com/ARMmbed/mbed-os-example-blinky
```

# 编译工程

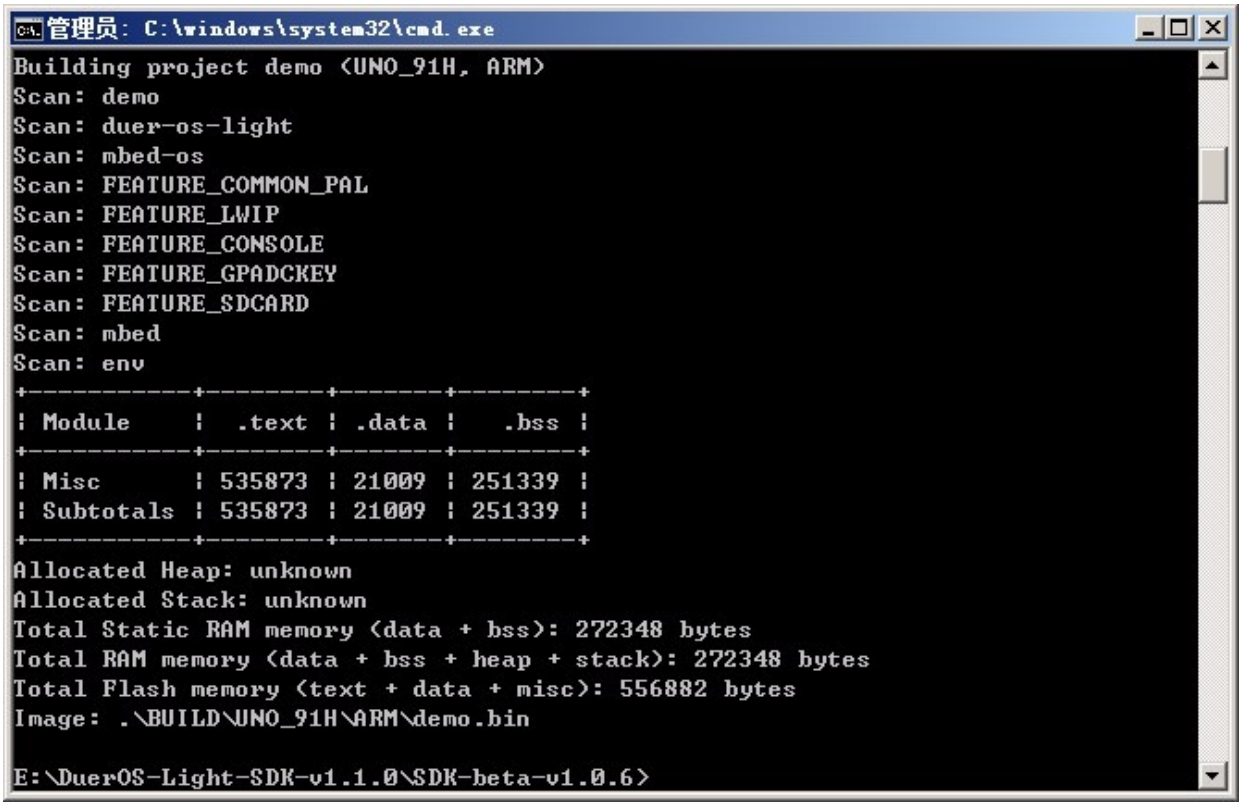
新建工程目录，参照(2.2)下载duerOS Light SDK，解压到工程目录，目录结构如下：




工程目录下执行以下命令进行编译:

```
mbed compile --source demo --source duer-os-light --source mbed-os -m UNO_91H -t ARM
```

`ARM` 表示mbed CLI编译时使用的工具链，可以替换成其他mbed支持的工具链。
`UNO_91H` 表示目标平台，可替换成其他平台，目前SDK中提供的Demo仅支持UNO_91H平台。
编译成功后，终端显示如下：




`SDK-beta-v1.0.6\BUILD\UNO_91H\ARM` 路径下有新编译生成的 `demo.bin` 文件。

编译时报如下错误时，请执行以下命令安装 `jinja2` 和 `prettytable` ：

- 安装命令

```
pip install jinjia2
pip install prettytable
```

- 错误信息

```
C:\dumi>cd C:\dumi\DuerOS-Light-SDK-v1.1.0

C:\dumi\DuerOS-Light-SDK-v1.1.0>mbed compile --source demo --source duer-os-ligh
t --source mbed-os -m UNO_91H -t ARM
[mbed] WARNING: Could not find mbed program in current path "C:\dumi\DuerOS-Ligh
t-SDK-v1.1.0".
[mbed] WARNING: You can fix this by calling "mbed new ." in the root of your pro
gram.
---
Traceback (most recent call last):
  File "C:\dumi\DuerOS-Light-SDK-v1.1.0\mbed-os\tools\make.py", line 43, in <mod
ule>
    from tools.options import get_default_options_parser
  File "C:\dumi\DuerOS-Light-SDK-v1.1.0\mbed-os\tools\options.py", line 18, in <
module>
    from tools.toolchains import TOOLCHAINS
  File "C:\dumi\DuerOS-Light-SDK-v1.1.0\mbed-os\tools\toolchains\__init__.py", l
ine 36, in <module>
    from tools.memap import MemapParser
  File "C:\dumi\DuerOS-Light-SDK-v1.1.0\mbed-os\tools\memap.py", line 11, in <mo
dule>
    from prettytable import PrettyTable
ImportError: No module named prettytable
[mbed] ERROR: "C:\Python27\python.exe" returned error code 1.
[mbed] ERROR: Command "C:\Python27\python.exe -u C:\dumi\DuerOS-Light-SDK-v1.1.0
\mbed-os\tools\make.py -t ARM -m UNO_91H --source demo --source duer-os-light --
source mbed-os --build .\BUILD\UNO_91H\ARM" in "C:\dumi\DuerOS-Light-SDK-v1.1.0"
```

# 2.2 SDK及工具下载

以下为开发者开发过程中会用到的SDK及库下载链接：

| 名称 | 版本 | 下载链接 | 说明 |
| --- | --- | --- | --- |
| DuerOS-Light-SDK | v1.2.9 | 下载 | v1.2.9的SDK中适配了RDA的两款芯片，下载的压缩包中有两个版本的SDK和一个说明文档，请选择适当的SDK解压使用 |
| bootloader | v1.2.9 | 下载 | 开启OTA功能时，会用到 |

下面是烧写时会用到的工具，前三个工具是RDA提供的：

| 名称 | 下载链接 | RDA下载地址 | 功能 |
| --- | --- | --- | --- |
| Merge Tool | 下载 | 由此进入 | 合并bin文件，烧录带OTA功能时使用 |
| RDA5981 单口下载工具 | 下载 | 由此进入 | RDA提供专用烧录工具 |
| image-pack.py | 下载 | 无 | 为工程文件添加版本信息 |

# 3. DuerOS Light SDK介绍

DuerOS Light SDK是由百度提供的基于mbed-os的设备端软件开发工具包（SDK）。适用于故事机、音响、智能手表等轻量级设备的开发。

DuerOS Light SDK提供的API可以帮助开发者简便的实现以下功能：

设备连接百度IoT云，享用百度IoT云提供的服务；
百度IoT云提供了大量的服务，如：儿童故事、百科、娱乐（海量的线上资源）、聊天、天气查询、时间查询等等。
录音、播放媒体文件；
OTA升级
DuerOS Light SDK主要包括CA（Connection Agent），Http，Media，Recoder，OTA等模块。

# 3.1 CA模块

该模块主要提供设备通过profile接入到设备云的能力。设备在连接上设备云后，可以通过相关接口向云端上报数据，并接收云端下发的指令。完成设备数据上报、查询及控制等能力。

CA模块的对外API由Scheduler类提供。 使用时需要包含**baidu_ca_scheduler.h**头文件。
Scheduler是单例类，可以通过 `duer::Scheduler::instance()` 来获取Scheduler的实例。

# 3.2 Http模块

提供设备通过http协议下载server上资源的功能，SDK中主要用于下载网络媒体文件、下载云端推送的OTA升级的固件等。

Http模块的对外API由HttpClient类提供。使用时需要包含**baidu_http_client.h头**文件。

# 3.3 Media模块

提供本地或网络媒体文件的播放功能，目前可支持mp3，wav，m3u8，aac和m4a格式，播放m4a格式要求设备上有SD卡或PSRAM。

Media模块的对外API由MediaManager类提供。 使用时需要包含**baidu_media_manager.h**头文件。
MediaManager是一个单例类，需要调用 `duer::MediaManager::instance()` 来获取MediaManager类的实例。

# 3.4 Recoder模块

提供录音功能。

Recoder模块的对外API由RecorderManager类提供。使用时需要包含**baidu_recorder_manager.h**头文件。

# 3.5 OTA模块

提供固件版本升级功能。使用OTA功能时，开发者可在DuerOS开放平台自行配置升级策略。
主要头文件包括：lightduer_ota_updater.h、lightduer_ota_unpack.h、lightduer_ota_notifier.h、lightduer_dev_info.h、IOtaUpdater.h等。开发者使用时需要参考各个接口的定义来包含相关头文件。

- 设备端OTA功能：

    1. 支持多种下载协议下载升级包
    2. 解压缩升级包
    3. 解密升级包（通过配对的公钥解密，保证升级包的来源）
    4. 校验升级包
    5. 提供用户升级包的配置信息（此功能尚未开放，需要开放，请联系产品经理）
    6. 提供用户原始上传的文件信息

- 云端OTA功能

    1. 提供OTA的推送策略配置，详细推送策略配置请参照论坛FAQ
    2. 提供升级包信息配置功能：在console平台–OTA升级下，添加固件（带OTA功能编译出的原始bin文件）时，会要求输入版本号，云端会将此版本号合入固件中。
    3. 提供云端到设备端的安全通道（通过私钥加密升级包）
    4. 提供压缩升级包功能，减小网络负担

# 3.6 闹钟模块

闹钟模块提供定时闹钟功能，需要开发者调用相关接口进行实现设置闹钟、删除闹钟、上报闹钟事件的功能。
使用闹钟模块需要开闹钟服务，请有需求的开发者联系我们开启闹钟服务。

# 4. 设备端开发

为了能让开发者迅速的掌握使用DuerOS Light SDK开发项目，下面将详细介绍一下设备端开发中的各个重要步骤。

# 4.1 创建工程

1. 创建工程目录，并将DuerOS Light SDK包解压到该目录，目录结构如下：


2. 在工程目录下创建存放开发者自己代码的目录。并在该目录下，创建一个mbed_app.json文件用于配置设备相关的信息，mbed_app.json示例如下：

```
"target_overrides": {
    "UNO_91H": {
        "target.features_add": ["LWIP", "SDCARD", "GPADCKEY", "CONSOLE"],
        "target.macros_add": ["BD_FEATURE_NET_STACK_USING_WIFI", "BD_FEAT
URE_HTTPCLIENT_USING_WIFI"]
     },
    "K64F": {
        "target.features_add": ["LWIP"]
    }
}
```

在文件mbed-os\hal\targets.json中有各种target的定义，包括UNO_91H，K64F。示例代码中target_overrides部分，会override targets.json文件中UNO_91H，K64F的定义。以UNO_91H为例，target.features_add表示在原有的features定义中增加”LWIP”, “SDCARD”, “GPADCKEY”, “CONSOLE”四个feature。target.macros_add表示增加2个宏定义”BD_FEATURE_NET_STACK_USING_WIFI”, “BD_FEATURE_HTTPCLIENT_USING_WIFI”。具体规则请参照mbed-os开发网站的详细说明。

# 4.2 编写代码

以下是编写代码时必须注意的点，其它详细功能的使用请参考5.主要接口类及API

## 4.2.1 初始化工作

- 初始化SD卡，平台不同初始化的方式也不同，开发者需根据自己的平台来做初始化

```
//定义静态变量初始化SD卡
#if defined(TARGET_UNO_91H)
SDMMCFileSystem sd(GPIO_PIN9, GPIO_PIN0, GPIO_PIN3, GPIO_PIN7, GPIO_PIN1
2, GPIO_PIN13, "sd");
#elif defined(TARGET_K64F)
static SDFileSystem sd = SDFileSystem(D11, D12, D13, D10, "sd");
#endif
```

- 初始化flash（仅限TARGET_UNO_91H平台），开发者需根据实际使用硬件情况来做初始化

```
const unsigned int RDA_FLASH_SIZE = 0x400000;        //Flash Size
const unsigned int RDA_SYS_DATA_ADDR = 0x18204000;  //System Data Area, f
ixed size 4KB
const unsigned int RDA_USER_DATA_ADDR = 0x18205000; //User Data Area star
t address
const unsigned int RDA_USER_DATA_LENG = 0x3000;     //User Data Area Leng
th

rda5981_set_flash_size(RDA_FLASH_SIZE);
rda5981_set_user_data_addr(RDA_SYS_DATA_ADDR, RDA_USER_DATA_ADDR, RDA_USE
R_DATA_LENG);
```

- 初始化Media模块

```
duer::MediaManager::instance().initialize();
```

- 连接网络：开发者可根据自身情况，调用mbed-os接口选择连接wifi或ethernet
- 初始化CA库

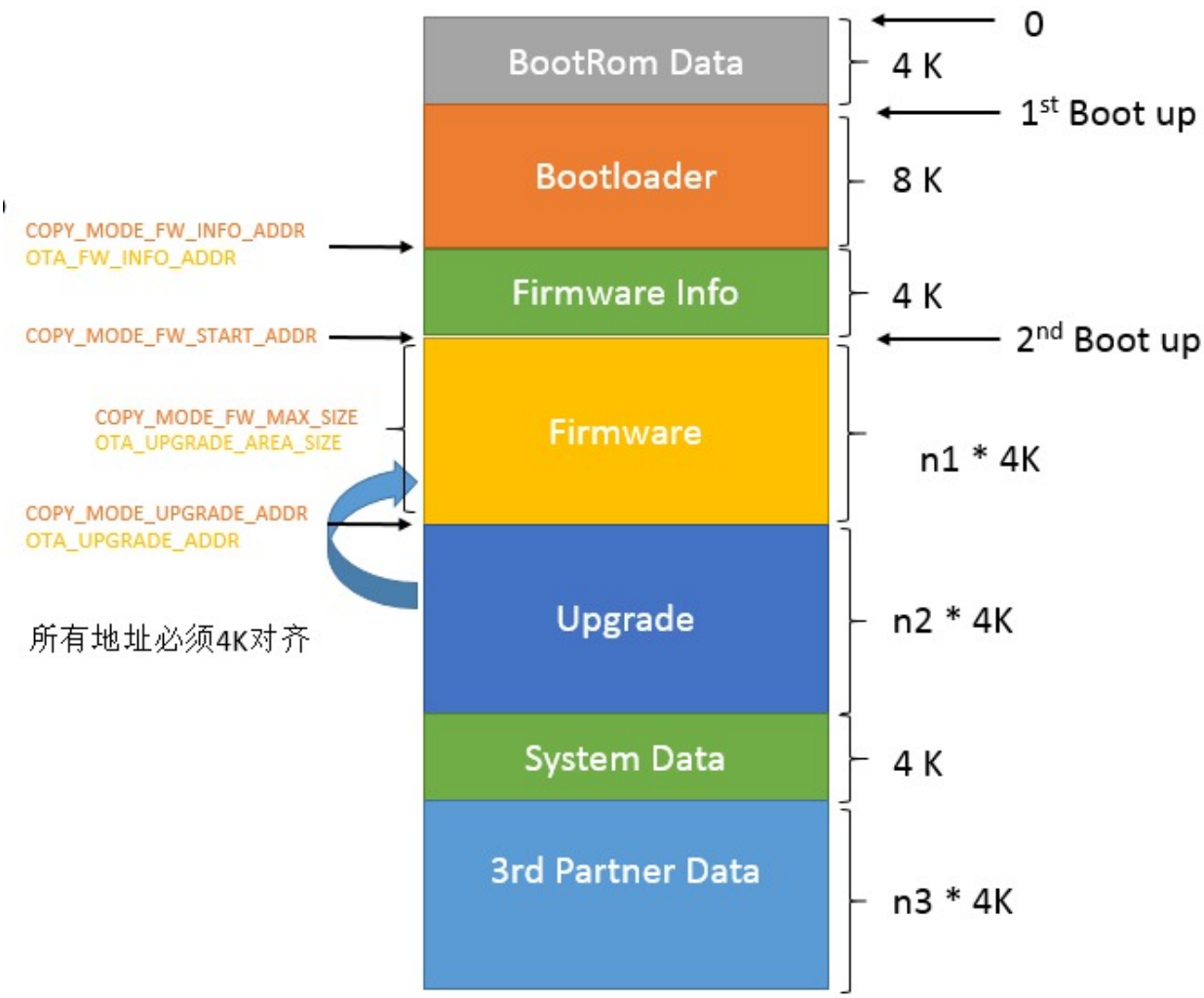
```
duer::Scheduler::instance().start();
```

# 4.2.2 SD卡的使用

SD卡支持FAT12 / FAT16 / FAT32格式，最大支持32Gb。初始化SD卡之后，可以类似”/sd/xxx”的路径访问sd卡上的文件。

# 4.2.3 OTA功能

## 4.2.3.1 启用OTA功能时Flash Layout介绍




`BootRom Data` ：BootRom使用的数据，用户不可读写
`Bootloader` ：bootloader代码区
`Firmware Info` ： 当前执行的固件描述信息
`Firmware` ：当前执行的代码区
`Upgrade` ：待升级的固件存储区
`System Data` ：存放MAC地址，WiFi SSID&PASSWORD
`3rd Partner Data` ：用户数据

## 4.2.3.2 启用OTA功能

1.创建OTA_Config.c文件，内容如下：

```
//COPY_MODE_FW_INFO_ADDR
const unsigned int OTA_FW_INFO_ADDR = 0x18003000;
//COPY_MODE_UPGRADE_ADDR
const unsigned int OTA_UPGRADE_ADDR = 0x18104000;
//COPY_MODE_FW_MAX_SIZE, must be 4K aligned
const unsigned int OTA_UPGRADE_AREA_SIZE = 0x100000;
```

设置 `OTA_FW_INFO_ADDR`，`OTA_UPGRADE_ADDR` ，分别表示当前firmware信息的存放地址和更新包存放分区地址，这两个值要分别与bootloader_config.h里的 `COPY_MODE_FW_INFO_ADDR`，`COPY_MODE_UPGRADE_ADDR` 相同； `OTA_UPGRADE_AREA_SIZE` 设置的值应不大于firmware binary分区的最大值。

**以下几点需要特别注意：**

- 如果不定义这几个常量，工程会链接失败。
- 这几个地址必须与bootloader匹配，不匹配会导致启动失败。
- 上文中的具体数值就是与我们提供的bootloader匹配的值，当使用其他的bootloader时，请自行定义。

2.参照4.3.2进行编译。

3.参照4.4.2进行烧录。

#### 4.2.3.3 OTA功能使用

具体使用参考DuerOS轻量级设备解决方案接入指南。
**注：目前 [2017-9-5] OTA功能对工程文件 `xxx.bin` 的大小有限制，应小于2M。**

## 4.2.4 其它注意事项

开发者需在代码中定义如下函数，以供DuerOS Light使用，否则工程会链接失败。
`void *baidu_get_netstack_instance(void)` ，定义该函数提供WiFiStackInterface或EthernetInterface对象指针。该函数的实现可参照DuerOS Light SDK中Demo下的main.cpp文件来实现。

# 4.3 工程编译

工程的编译使用mbed-cli工具，开发者可参照以下命令编译自己的工程：

## 4.3.1不带OTA功能编译

```
//在工程目录下执行以下命令
mbed compile --source PROJECT_NAME --source duer-os-light --source mbed-o
s -m TARGET -t TOOL_CHAIN
```

## 4.3.2带OTA功能编译

```
//编译带OTA功能的请使用如下命令
mbed compile --source PROJECT_NAME --source duer-os-light -DBD_FEATURE_EN
ABLE_OTA --source mbed-os -m TARGET -t TOOL_CHAIN
```

- PROJECT_NAME为工程名称
- TARGET为目标平台，目前DuerOS Light SDK支持*UNO_91H，K64F，UNO_81C*

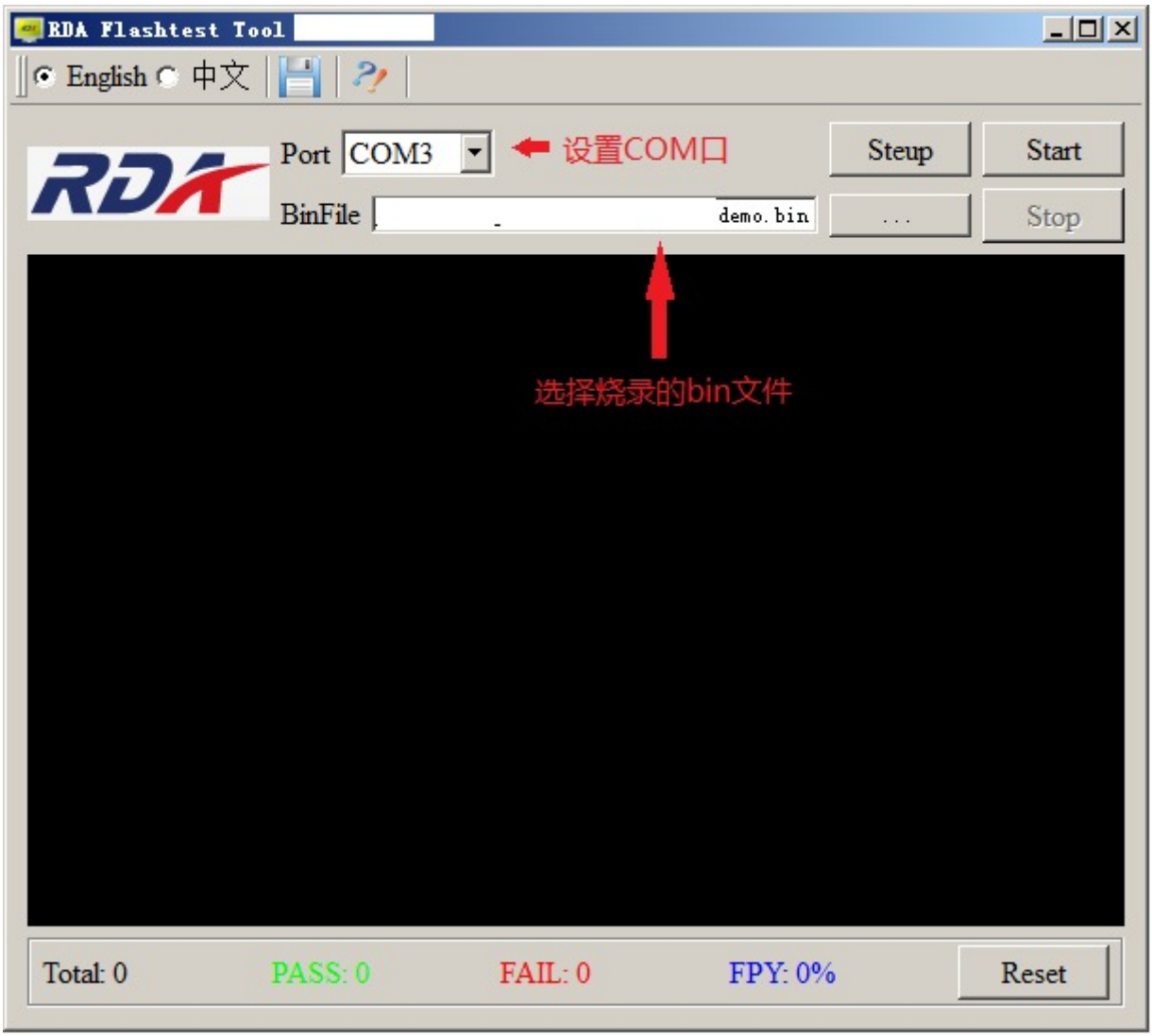
- TOOL_CHAIN为编译工具链，DuerOS Light SDK推荐使用*ARM*
- BD_FEATURE_ENABLE_OTA 开启OTA功能，默认是关闭的

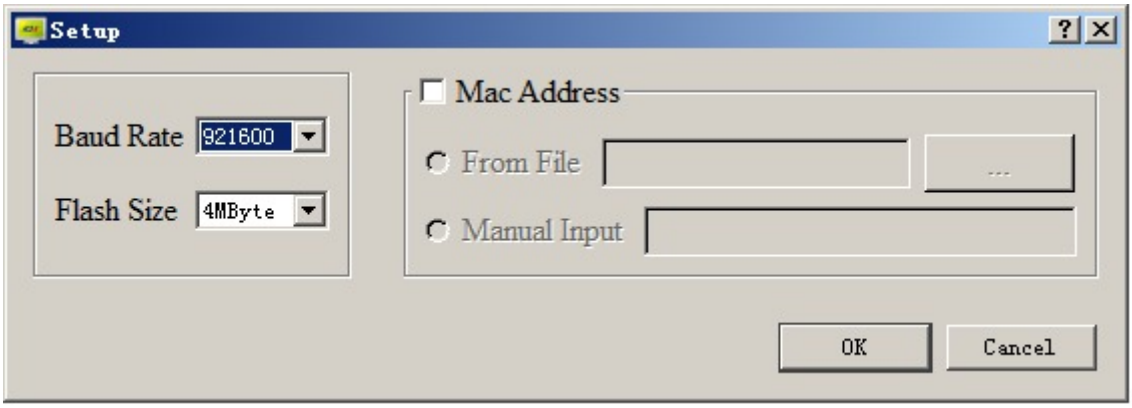
# 4.4 烧录

## 4.4.1 烧录不带OTA功能操作

直接将编出的工程文件烧录到板子中即可。
具体方法如下（建议采用方法一）：

- 方法一、通过RDA5981 单口下载工具
  RDA Flashtest工具打开后如下，参照图示进行设置：



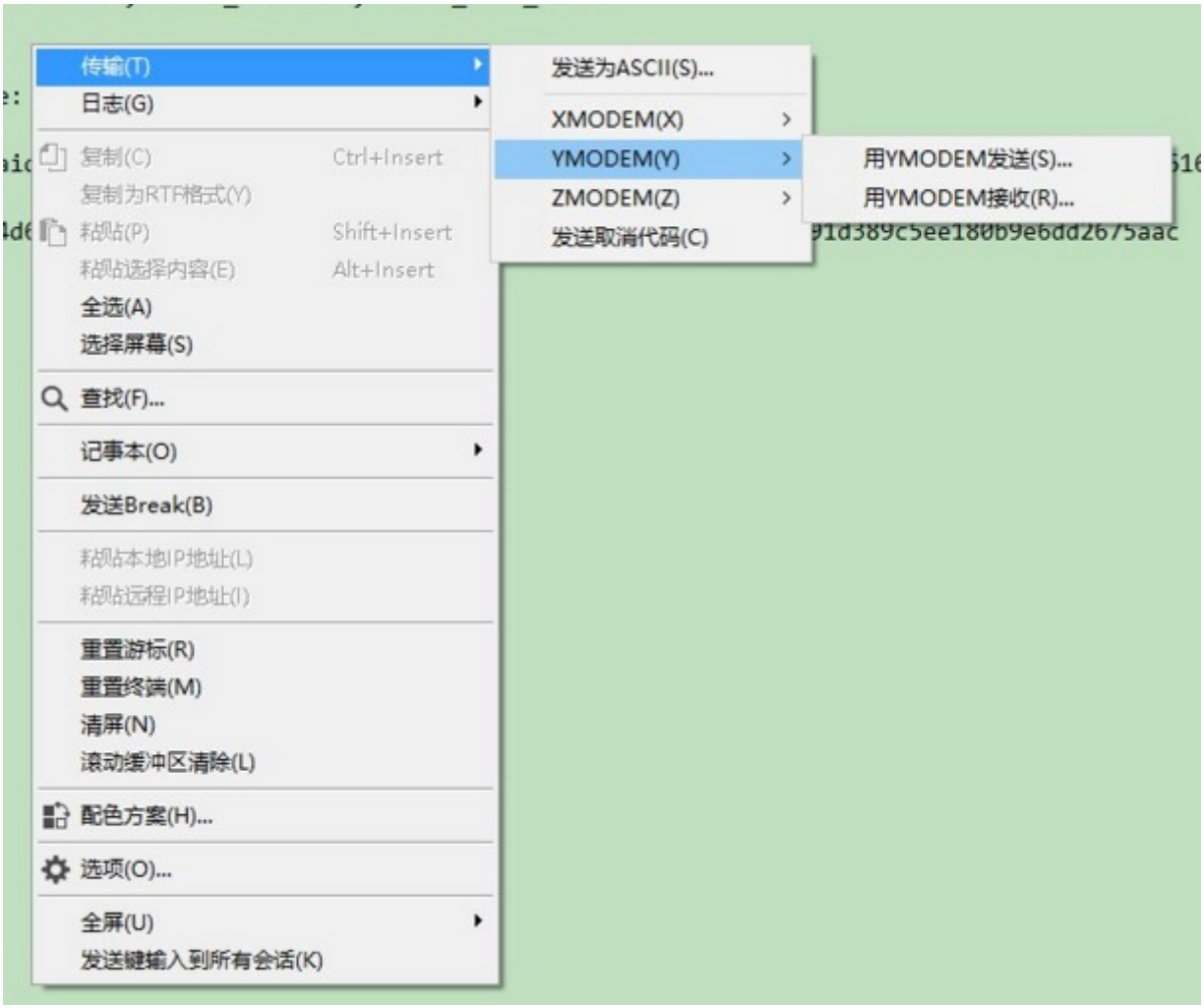



点击Setup按钮会弹出图下方的Setup窗口，参考图中所示进行设置。
之后点击Start按钮，后面待出现下面输出后，点击reset按钮即开始烧录。

```
Open port...
==============================
Running...
Waiting for plug in...
```

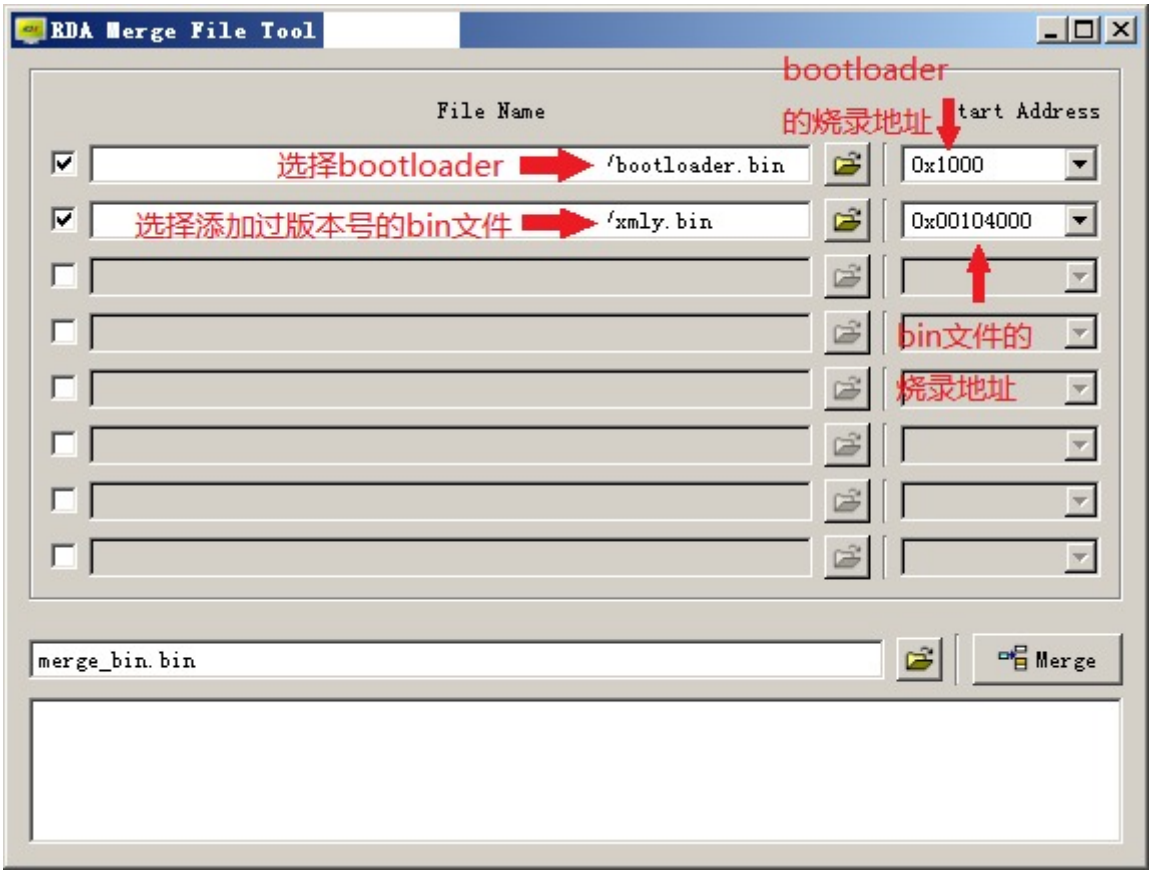
- 方法二、通过板子自带命令loady
    1. 连接串口工具；
    2. 在串口终端出现“count_left=0”之前输入回车，会出现“Boot abort”；
    3. 输入loady指令，等待出现”Ready for binary”后，右键点击选择传输，选择YMODEM，用YMODEM发送；
    4. 选择并传输对应bin文件；

```
RDA Wlan Boot ROM for RDA5991H v1.0
Build Time: 07:20:54 - Jan 24 2017
RDA Microelectronics Copyright(C) 2004-2017

Enter Mcu Mode
count_left=5

Boot >Boot abort

Boot >loady
## Ready for binary (ymodem) download to 0x18001000 at 921600 bps...
CC
```




# 4.4.2 烧录带OTA功能操作

1. 使用脚本文件image-pack.py，为编译出的工程文件(4.3.2)添加版本信息：
   `python image-pack.py xxx.bin y.y.y.y` xxx.bin为工程文件，y.y.y.y为版本号。
   上面命令执行完成会生成对应的xxx.bl文件，将xxx.bl修改为xxx.bin文件。
2. 下载对应的bootloader.bin，使用Merge File工具将bootloader.bin与步骤1生成的xxx.bin合并为一个bin进行烧录
   解压2.2中下载的bootloader文件，根据所使用芯片不同使用不同的bin文件。
   RDA5991H芯片使用bootloader_RDA5991H_U02.bin
   RDA5981C芯片使用bootloader_RDA5991H_U04.bin




3. 参考(4.4.1)将merge后的bin文件进行烧录。

# 4.5 默认语音数据点

通过语音数据点控制设备时需要云端做相应的适配。云端已经适配了以下几个默认的语音数据点。默认的数据点在设备端提供的SDK的demo中都进行了实现，如有需要请自行参考 `device_controller.cpp` 文件进行实现。

## 4.5.1 volume —— 设置音量

此数据点用于语音控制设备的音量或调整音量大小。
可通过以下query触发此数据点：

1. “调高音量”或“调低音量”
2. “把音量调到最小”或“把音量调到最大”
3. “把音量设置为10”

数据点详细信息：

| **标识名** | **volume** |
| --- | --- |
| 类型 | int |
| 取值范围 | 1~100 |

### 4.5.1 docancel —— 停止播放

此数据点用于语音控制设备停止当前播放的内容。
可通过以下query触发此数据点：

1. “我不想听了”
2. “停止”
3. “取消”

数据点详细信息：

| 标识名 | docancel |
| --- | --- |

### 4.5.1 docontinue —— 继续播放

此数据点用于语音控制设备续播之前播放的内容。
可通过以下query触发此数据点：

1. “继续播放”
2. “播放”
3. “接着播放”

数据点详细信息：

| 标识名 | docontinue |
| --- | --- |

# 5. 主要接口类

本节将详细介绍DuerOS Light SDK提供给开发者的主要接口类与API，类中未提供说明的public API开发者可忽略，之后会做优化。

## 5.1 Scheduler类

- **所属头文件**

  ```
  baidu_ca_scheduler.h
  ```

- **功能描述**

  单例类，封装CA模块，提供与云端的交互功能
  通过 `duer::Scheduler::instance()` 获取Scheduler类的实例。

# int Scheduler::set_on_event_listener(IOnEvent* listener) —— 注册监听者

- **功能描述**

注册事件监听者listener，监听3种事件：

- ①设备与云端建立连接；
- ②设备与云端断开连接；
- ③云端向设备发生数据；
- **调用方式**

**以下有两种常规的调用方式，使用任一种均可：**

1.在IOnEvent 的派生类的构造函数中调用（SDK的Demo中采用这种方式）

```
class SchedulerEventListener : public Scheduler::IOnEvent {
...
public:
SchedulerEventListener(DuerApp* app) : _app(app) {
    Scheduler::instance().set_on_event_listener(this);
}
...
}
```

2.在其他函数中调用，例如在main()函数中

```
class SchedulerEventListener : public Scheduler::IOnEvent {
...
}

int main() {
...
    IOnEvent* listener = new SchedulerEventListener();
    Scheduler::instance().set_on_event_listener(listener);
...
}
```

**注：**启用CA模块时，`int set_on_event_listener(IOnEvent* listener)` 接口只能调用一次，仅能注册一位监听者。第二次调用这个接口时，新的监听者会覆盖第一次注册的监听者。

- **返回值**

  非负数：成功
  负数：失败
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| IOnEvent* | listener | IOnEvent是baidu_ca_scheduler.h中定义的抽象类，以下有详细说明 |

- **IOnEvent 类型说明：**

  IOnEvent 中定义了4个接口，通过 `int set_on_event_listener(IOnEvent* listener)` 接口注册监听者后，会监听云端服务的3种事件。（为什么不是4种呢？稍后解释。）
- **IOnEvent定义如下：**

```
class IOnEvent
{
public:
        virtual int on_start() = 0;

    virtual int on_stop() = 0;

    virtual int on_data(const char* data) = 0;

    virtual int on_action(const char* url, bool is_speech, bool is_contin
ue_previous) = 0;
};
```

IOnEvent 是抽象类，使用时需要继承这个类实现自己的方法。

- **监听函数说明：**

1. `virtual int on_start() = 0;`
设备成功与云端建立连接时调用此函数，可在此函数中实现一些连接设备云成功后的初始化工作，可以根据自己的需求仿照DuerOS Light SDK的Demo中代码进行实现。
在DuerOS Light SDK的Demo中的这个接口进行了以下工作：
- 绑定数据点
- 注册按键的监听函数
- 播放成功连接云端的提示音

2. `virtual int on_stop() = 0;`
设备与云端断开连接时调用此函数，可在此函数中实现一些设备云断开时需要进行的工作，可以根据自己的需求仿照DuerOS Light SDK的Demo中代码进行实现。
在DuerOS Light SDK的Demo中的这个接口进行了以下工作：
- 停止播放
- 停止录音
- 注销on_start中注册的监听函数
- 重启，尝试重新连接云端

3. `virtual int on_data(const char* data) = 0;`

云端向设备发送数据时调用此接口，参数 `data` 中是以Json格式存放的云端下发的全部信息，可以在此接口中对 `data` 进行解析、处理。

例如： `data` 中存有云端需要播放的音频的url，解析出url后可以调用播放网络媒体文件的接口，播放声音。

在DuerOS Light SDK的Demo中的这个接口仅仅进行了打印日志的工作，没有进行其他处理。

4. `virtual int on_action(const char* url, bool is_speech, bool is_continue_previous) = 0;`

参数说明：
①url：云端发送的url
②is_speech：表示是否是语音数据
③is_continue_previous：表示是之前播放的音乐是否需要续播

此接口也是在云端向设备发送数据时调用，他与3接口的调用时机相同，云端会先调用 **4** 接口，然后调用 **3** 接口。此接口中的 **3** 接口个参数就是云端通过解析云端要下发的Json数据得出的。此接口的使用与接口 **3** 是相同的。

在DuerOS Light SDK的Demo中的这个接口进行了以下工作：

- 通过语段下发的url，播放网络媒体文件

  **注**：此接口中对下发的数据进行了解析，下发的信息不完整不建议使用。建议使用 **3** 接口。**3** 接口和 **4** 接口不建议同时使用，尤其禁止处理相同的动作，会导致重复执行、浪费资源和一些不可预测的BUG发生。

# int Scheduler::add_controll_points(const bca_res_t list_res[], bca_size_t list_res_size) —— 绑定数据点

- **功能描述**

  绑定云端的数据点，提供云端对设备的回调接口。此接口能够在设备云端的交互中添加一些自定义的功能，是使用云端的数据点自定义控制、查询、上报功能时，必须用到的接口。
- **数据点说明**

对开发者而言，数据点通常指console平台上设置的数据点，参考DuerOS轻量级设备解决方案接入指南理解数据点。
除此之外，还有几个特殊的数据点，如：play、volume、alive、newfirmware。
play：用于控制播放的数据点，不需要开发者进行绑定，DuerOS Light SDK中进行了绑定，支持语音控制功能。
volume：用于控制音量增减，需要开发者自行绑定。支持语音控制功能，但要在console平台自行开启服务。
alive：用于上报心跳，证明设备在线，不需要开发者进行绑定，DuerOS Light SDK中进行了绑定。
newfirmware：用于OTA升级，不需要开发者进行绑定，DuerOS Light SDK中进行了绑定。

- **调用方式**

```
bca_res_t res[] = {
    {BCA_RES_MODE_STATIC, BCA_RES_OP_GET, "alive", .res.s_res = {(vo
id*)"true", 4}},
    {BCA_RES_MODE_DYNAMIC, BCA_RES_OP_PUT, "stop", media_stop},
    {BCA_RES_MODE_DYNAMIC, BCA_RES_OP_PUT | BCA_RES_OP_GET, "volume"
, set_volume},
    {BCA_RES_MODE_DYNAMIC, BCA_RES_OP_PUT, "shutdown", shutdown},
    {BCA_RES_MODE_DYNAMIC, BCA_RES_OP_PUT | BCA_RES_OP_GET, "mode",
 set_mode},
    {BCA_RES_MODE_DYNAMIC, BCA_RES_OP_GET, "power", get_power},
};
Scheduler::instance().add_controll_points(res, sizeof(res) / sizeof
(res[0]));
```

**注：**此接口可以多次调用，用于绑定不同的数据点，通过 `path` 进行区分，每次绑定的 `path` 不能相同，否则会报错。

- **返回值**

非负数：成功
负数：失败

- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| const bca_res_t[] | list_res | list_res是bca_res_t类型的数组，是baidu_ca.h中定义的结构，bca_res_t类型以下有详细说明 |
| bca_size_t | list_res_size | list_res_size记录数组list_res的大小，`是unsigned int类型的 |

- **bca_res_t 定义如下：**

```
typedef struct _bca_resource_s
{
bca_u8_t    mode:2;
bca_u8_t    allowed:6;
char*       path;
union {
    bca_notify_f   f_res;
    struct {
        void*       data;
        bca_size_t  size;
    } s_res;
} res;
} bca_res_t;
```

- **bca_res_t 类型说明：**

  bca_res_t 类型中包含四类信息：

  ①mode：云端调用数据点时设备端返回的资源类型，分为静态资源和动态资源

  - BCA_RES_MODE_STATIC：静态资源，设备端将 `data` 指向的 `size` 大小的资源包装为CoAP message的payload发送给云端
  - BCA_RES_MODE_DYNAMIC：动态资源，执行设备端的相关函数 `f_res` ，返回相应的处理结果

  ②allowed：定义

  - BCA_RES_OP_GET ：用来获取设备端的数据
  - BCA_RES_OP_PUT ：用来更新设备端的数据
  - BCA_RES_OP_POST：在设备端创建数据
  - BCA_RES_OP_DELETE：删除设备端的数据

  **注：**此项定义符合CoAP协议的四种方法，进行设备端开发时请遵守CoAP；一个allowed可以具有多种权限，写代码时可以用按位或( | )运算符连接，如 `BCA_RES_OP_PUT | BCA_RES_OP_GET` 。

  ③path：char*类型，用作与云端数据点匹配的字符串，path字符串内容应与云端数据点的标识名相同，来完成云端数据点与设备端的绑定。

  ④res：按照资源类型（ mode ）分为两种，定义如下：

```
union {
    bca_notify_f   f_res;
    struct {
        void*       data;
        bca_size_t  size;
    } s_res;
```

- s_res：mode是BCA_RES_MODE_STATIC时，使用 `res.s_res` ， `data` 指向静态资源的指针， `size` 资源的大小，bca_size_t 是无符号整型。
- f_res：mode是BCA_RES_MODE_DYNAMIC时，使用 `res.f_res` ， `f_res` 是云端访问数据点时调用函数的函数指针， `bca_notify_f` 类型定义如下：

```
typedef bca_status_t (*bca_notify_f)(bca_context ctx, bca_msg_t
* msg, bca_addr_t* addr);
```

`bca_status_t` 表示函数的返回值是整型， `ctx` 型指针， `msg` 是指向CoAP协议的消息结构的指针， `addr` 表示指向地址信息的指针。

# int Scheduler::start() —— 建立连接

- **功能描述**

  开始与云端建立连接，调用此接口成功后设备与云端连接成功，可以使用云端提供的服务。
- **调用方式**

```
Scheduler::instance().start();
```

- **返回值**

  非负数：成功
  负数：失败
- **参数说明**

  无

# int Scheduler::stop() —— 断开连接

- **功能描述**

  断开与云端的连接，调用此接口成功后设备与云端的连接断开，停止使用云端服务。
  **注：**接口内有状态标志，与云端断开连接时调用此接口。
- **调用方式**

```
duer::MediaManager::instance().stop();
```

- **返回值**

  非负数：成功
  负数：失败
- **参数说明**

无

# int Scheduler::report(...) —— 上报数据

- **功能描述**

  此接口用于设备向云端上报数据，使用上报数据点时需要用到此接口将信息进行上报。

  为方便使用，对此接口进行了重载，支持两种参数：

  1. `int Scheduler::report(baidu_json *data)`
  2. `int Scheduler::report(const Object& data)`

- **调用方式**

```
Object data;
data.putInt("time", us_ticker_read());
Scheduler::instance().report(data);
```

  **注：**这并不是唯一的调用方式，此接口支持两种参数，以下详细介绍。

- **返回值**

| **类型** | **int** |
| --- | --- |

| **返回结果** | **意义** |
| --- | --- |
| 非负数 | 成功 |
| 负数 | 失败 |

- **参数说明**

  以下是对两种参数的分别说明：

| **类型** | **名称** | **描述** |
| --- | --- | --- |
| baidu_json * | data | baidu_json是baidu_json.h中定义的结构体，具体的使用请参考baidu_json.h文件 |

| **类型** | **名称** | **描述** |
| --- | --- | --- |
| const Object& | data | Object类是对baidu_json的一种封装，具体的使用请参考baidu_ca_object.h文件 |

**注：**建议使用const Object&类型的参数，在 `int Scheduler::report(baidu_json *data)` 的实现中，也是将baidu_json *类型转换成const Object&类型后处理的。

object中信息的可以按照 `key : value` 的形式存储，可以参照调用方式的代码书写，object中实现的方法可以参考baidu_ca_object.h文件。

# int Scheduler::send_content(const void* data, size_t size, bool eof) —— 发送语音数据

- **功能描述**

设备与云端进行交互的接口，使用此接口可以向云端发送语音数据。为使录音传输并发进行，语音数据可分段上传。

- **调用方式**

```
Scheduler::instance().send_content(data, size, false);
```

- **返回值**

  非负数：成功
  负数：失败
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| const void* | data | 指向存放语音数据buffer的指针 |
| size_t | size | data指向的语音数据的大小 |
| bool | eof | 表示当前要传输的data是否是最后一段语音数据， `eof` 等于 `true` ，是最后一段数据 |

**注：**判断传输的是最后的一段语音数据的条件并不完全依赖于eof，判断条件 `eof || data == NULL || size == 0` 为 `true` 时表示传输的是最后一个语音片段。数据传输时，请保证 `data` 与 `size` 的值正确。

# int Scheduler::response(const bca_msg_t* req, int msg_code, const char* payload) —— 响应云端的请求

- **功能描述**

  云端通过数据点向设备发送请求，设备端调用相应的处理函数（在绑定数据点接口中有详细说明），对于云端下发的请求，设备端处理结束后应给出相应的应答，向云端报告处理结果。此接口就是完成的这项功能的。
- **调用方式**

  以标识符为 `shutdown` 的数据点为例：

```
 static bca_status_t shutdown(bca_context ctx, bca_msg_t* msg, bca_addr_t
* addr) {
        bca_handler handler = (bca_handler)ctx;
        DUER_LOGV("shutdown");

        if (handler && msg) {
            // 具体调用的代码
            duer::Scheduler::instance().response(msg, BCA_MSG_RSP_CHANGE
D, NULL);
            duer::Scheduler::instance().stop();
        }

        return BCA_NO_ERR;
}
```

- **返回值**

  非负数：成功
  负数：失败
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| const bca_msg_t* | req | 云端发送下来的msg， `bca_msg_t` 类型是CoAP协议的消息类型，定义在baidu_ca.h文件中 |
| int | msg_code | msg_code是CoPA协议中的响应码，详细请阅读协议文档 |
| const char* | payload | 向云端发送的消息，云端识别出消息会进行响应的处理，payload可以为null，表示不发送消息 |

# int Scheduler::clear_content() —— 停止上传并清理未上传数据

- **功能描述**

  设备端主动结束上传数据，并且清空还未上传的数据。调用此接口可以终止当前进行的上传数据动作，来处理下一次的传数据动作或其他动作。

- **调用方式**

```
Scheduler::instance().clear_content();
```

- **返回值**

  非负数：成功
  负数：失败
- **参数说明**

  无

# 5.2 HttpClient类

- **所属头文件**

```
baidu_http_client.h
```

- **功能描述**

  提供通过http协议下载数据的功能，SDK中主要用于下载网络媒体文件和OTA升级固件包。

# void register_data_handler(data_out_handler_cb data_hdlr_cb, void* p_usr_ctx) —— 注册一个回调函数，通过该函数处理下载到的数据

- **功能描述**

  用来注册一个回调函数，通过该函数处理下载到的数据
- **返回值**

  无
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| data_out_handler_cb | data_hdlr_cb | 回调函数指针，由该函数处理http模块从服务器下载的数据 |
| void* | p_usr_ctx | 传入回调函数的用户参数指针，可以为空 |

- **data_out_handler_cb 类型说明：**

  data_out_handler_cb 是拥有5个参数返回类型为int的函数指针，用于设置回调函数，该函数会在server通过http协议向设备传送数据时调用。
- **data_out_handler_cb 定义如下：**

```
typedef enum data_pos {
            DATA_FIRST  = 0x1,
            DATA_MID    = 0x2,
            DATA_LAST   = 0x4
} e_data_pos;
typedef int (*data_out_handler_cb)(void* p_user_ctx, e_data_pos pos, cons
t char* buf, size_t len, const char* type);
```

- **函数指针的参数说明：**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| void* | p_user_ctx | 用户注册回调函数时，传入的参数指针 |
| e_data_pos | pos | 表明此次下载数据段在整个数据中的位置 |
| const char* | buf | http模块下载的数据 |
| size_t | len | buf数据的长度 |
| const char* | type | buf数据的类型。此类型是http请求时server返回的类型，可能是mpeg、mp3等 |

# void register_notify_call_back(check_stop_notify_cb_t chk_stp_cb) —— 注册检查停止下载的回调函数

- **功能描述**

  注册检查停止下载的回调函数。http模块会根据回调函数的返回值判断是否需要继续下载。
- **返回值**

  无
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| check_stop_notify_cb_t | chk_stp_cb | 回调函数指针，由该函数处理判读是否需要停止下载 |

- **回调函数说明：**

http模块会根据此接口注册的回调函数返回值来处理是否继续下载。返回值为1时停止下载，返回值为0时继续下载。

- **check_stop_notify_cb_t 定义如下：**

```
typedef int (*check_stop_notify_cb_t)();
```

# int get(const char* url, const size_t pos) —— 请求下载数据

- **功能描述**

  根据url连接server，请求下载数据，成功后会通过register_data_handler接口设置的回调函数处理下载的数据。

- **返回值**

  此接口的返回类型与e_http_result的意义相同。

  e_http_result 定义如下：

```
//http client results状态码
typedef enum http_result {
    HTTP_OK = 0,              // 成功
    HTTP_PROCESSING,          // 正在处理
    HTTP_PARSE,               // url 解析失败
    HTTP_DNS,                 // 域名解析错误
    HTTP_PRTCL,               // http协议错误
    HTTP_NOTFOUND,            // 错误代码404
    HTTP_REFUSED,             // 错误代码403
    HTTP_ERROR,               // HTTP xxx error
    HTTP_TIMEOUT,             // 连接超时http 的头重定向
    HTTP_CONN,                //  连接错误
    HTTP_CLOSED,              // 远程主机关闭了连接
    HTTP_NOT_SUPPORT,         //  不支持的功能http 的头重定向
    HTTP_REDIRECTTION,        // http 的头重定向
    HTTP_FAILED=-1,
} e_http_result;
```

- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| const char* | url | 请求下载数据的url地址 |

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| const size_t | pos | 通过这个参数可以自定义下载数据的开始位置，下载数据可以从server上资源的任意位置开始下载 |

- **回调函数说明：**

  http模块会根据此接口注册的回调函数返回值来处理是否继续下载。返回值为1时停止下载，返回值为0时继续下载。
- **check_stop_notify_cb_t 定义如下：**

```
typedef int (*check_stop_notify_cb_t)();
```

# void get_download_progress(size_t* total_size, size_t* recv_size) —— 获取下载进度

- **功能描述**

  获取http模块下载时数据总大小和数据开始位置到当前位置的大小，通过计算可得出下载进度。
- **返回值**

  无
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| size_t* | total_size | 所需下载数据的总大小。如果是chunk下载，无法知道数据的总大小，total_size等于0 |
| size_t* | recv_size | 从下载数据的开始位置到当前下载到的位置的大小 |

# Http模块使用示例代码

```
// 下载音频的url
#define URL "http://res.iot.baidu.com/4782182324fffd28daef48.mp3"
// 处理下载数据的函数
static int download_callback(void* p_user_ctx,
                             e_data_pos pos,
                             const char* buf,
                             size_t len,
                             const char* type) {
    // 函数内实现对下载的数据的处理，例如：保存、播放等等
    ...
}
// 检查停止下载的函数
static int check_stop_download(){
    // 函数内实现检查停止下载的功能
    // 需要停止下载时return 1;
    // 不需要停止下载时return 0;
    ...
}


// 实例一个HttpClient的对象
HttpClient http_client;

// 注册处理下载数据的回调函数
http_client.register_data_handler(download_callback, &buffer);

// 注册检查停止下载的回调函数
http_client.register_notify_call_back(check_stop_download);

// 请求下载URL中的数据，从数据的起始位置开始下载
int ret = http_client.get(URL, 0);

// 获取并计算出下载速度
size_t totalSize, recvSize;
float downloadProgress;
http_client.get_download_progress(&totalSize, &recvSize)
if (total_size == 0)
{
    DUER_LOGI("没有获取到当前的下载进度");
}
else
{
    downloadProgress = recvSize * 100.0 / totalSize;
    DUER_LOGI("当前的下载进度: %f%%", downloadProgress);
}
```

# 5.3 MediaManager类

- **所属头文件**

```
baidu_media_manager.h
```

- **功能描述**

单例类，封装MediaManager模块，提供播放的功能。
通过 `duer::MediaManager::instance()` 获取MediaManager类的实例。

- **Media模块播放状态说明**

Media模块定义了MediaPlayerStatus结构用来标记播放状态。

MediaPlayerStatus的定义如下：

```
enum MediaPlayerStatus {
    MEDIA_PLAYER_IDLE,      // 闲置状态，当前无播放内容
    MEDIA_PLAYER_PLAYING,   // 正在播放
    MEDIA_PLAYER_PAUSE      // 暂停状态
};
```

# bool initialize(…) —— 初始化MediaManager

此接口进行了重载，分为下面两种形式：

```
bool initialize();
bool initialize(IBuffer* buffer, size_t size);
```

- **功能描述**

初始化MediaManager，第一次调用有效，使用媒体播放功能前必须调用该接口做初始化。

- **调用方式**

1. 无参数调用：

```
duer::MediaManager::instance().initialize();
```

2. 有参数调用

```
IBuffer* buffer = new PsramBufferr(address, size);
duer::MediaManager::instance().initialize(buffer, size);
```

- **返回值**

true：成功
false：失败

- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| IBuffer* | buffer | 一个buffer的首地址指针，此buffer用于缓存从下载的音频文件，buffer存满后开始播放，buffer不宜过大 |
| size_t* | size | buffer的大小 |

**注：**通过无参接口对MediaManager初始化时，用剩余的data段做缓存音频文件的buffer。

# MediaPlayerStatus play_url(const char* url, bool is_speech, bool is_continue_previous) —— 播放网络媒体文件

- **功能描述**

  根据url下载网络中的媒体文件进行播放。
- **调用方式**

```
MediaManager::instance().play_url(main_url, is_speech, is_continue_previous);
```

- **返回值**

  media player的上一个状态
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| const char* | url | 网络媒体文件的url地址 |
| bool | is_speech | 播放的媒体资源是否是语音数据 |
| bool | is_continue_previous | 此次播放前播放的内容是否需要续播 |

# MediaPlayerStatus play_local(const char* path) —— 播放本地文件

- **功能描述**

根据path访问本地的媒体文件进行播放。

- **调用方式**

```
MediaManager::instance().play_local(path);
```

- **返回值**

media player的上一个状态

- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| const char* | path | 本地媒体文件路径 |

# MediaPlayerStatus pause_or_resume() —— 暂停/播放

- **功能描述**

播放状态下暂停播放，暂停状态下恢复播放，其它状态调用无效

- **调用方式**

```
MediaManager::instance().pause_or_resume();
```

- **返回值**

media player的上一个状态

- **参数说明**

无

# MediaPlayerStatus stop() —— 停止

- **功能描述**

停止播放当前文件

- **调用方式**

```
MediaManager::instance().stop();
```

- **返回值**

media player的上一个状态

- **参数说明**

  无

## MediaPlayerStatus stop_completely() —— 完全停止

- **功能描述**

  彻底停止播放，包括将要续播的内容。
- **调用方式**

```
MediaManager::instance().stop_completely();
```

- **返回值**

  media player的上一个状态
- **参数说明**

  无

## MediaPlayerStatus get_media_player_status() —— 获取播放状态

- **功能描述**

  获取media player的当前状态
- **调用方式**

```
MediaPlayerStatus status = MediaManager::instance().get_media_player
_status();
```

- **返回值**

  media player的上一个状态
- **参数说明**

  无

## int register_listener(MediaPlayerListener* listener) —— 注册播放状态的监听者

- **功能描述**

  注册监听者listener，用于监听播放的状态。
- **调用方式**

```
MediaPlayerListener* media_player_listener = new MediaPlayerListener
();
MediaManager::instance().unregister_listener(&media_player_listene
r);
```

- **返回值**

  0：注册成功
  -1：注册失败
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| MediaPlayerListener* | listener | 监听者的指针，监听者监听了三种事件 |

- **参数类型说明**

  MediaPlayerListener类型定义：

```
class MediaPlayerListener {
public:
    virtual int on_start() = 0;     // 开始播放时触发
    virtual int on_stop() = 0;      // 播放中途停止时触发
    virtual int on_finish() = 0;    // 播放正常结束时触发
    virtual ~MediaPlayerListener() {};
};
```

# int unregister_listener(MediaPlayerListener* listener) —— 注销监听者

- **功能描述**

  注销register_listener接口注册的监听者。
- **调用方式**

```
MediaManager::instance().unregister_listener(&media_player_listener)
;
delete media_player_listener;
```

- **返回值**

  0：注销成功
  -1：注销失败
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| MediaPlayerListener* | listener | 需要注销的监听者指针 |

# void set_volume(unsigned char vol) —— 设置音量

- **功能描述**

  设置音量大小，音量的大小分为16个级别，取值范围0~15。
- **调用方式**

```
duer::MediaManager::instance().set_volume(s_volume);
```

- **返回值**

  无
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| unsigned char | vol | 音量值，有效范围为0~15 |

# unsigned char get_volume() —— 获取音量

- **功能描述**

  获取当前音量大小
- **调用方式**

```
unsigned char s_volume = duer::MediaManager::instance().get_volume
();
```

- **返回值**

  当前音量值，取值范围0~15
- **参数说明**
  无

# void get_download_progress(size_t* total_size, size_t* recv_size) —— 获取http模块的下载进度

- **功能描述**

  获取http模块下载时的下载进度。
- **调用方式**

```
size_t otal_size, recv_size;
MediaManager::instance().get_download_progress(&total_size, &recv_si
ze);
```

- **返回值**

  无
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| size_t* | total_size | 所需下载数据的总大小。如果是chunk下载，无法知道数据的总大小，total_size等于0 |
| size_t* | recv_size | 从下载数据的开始位置到当前下载到的位置的大小 |

# void seek(int position) —— 调整播放进度

- **功能描述**

  调整当前的播放进度，自音频文件开始后position大小开始播放，position的值小于0时不做处理，position大于音频文件的大小时直接终止播放。
- **调用方式**

```
MediaManager::instance().seek(position);
```

- **返回值**

  无
- **参数说明**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| int | position | 播放位置的偏移量 |

# 5.4 RecorderManager类

- **所属头文件**

```
baidu_recorder_manager.h
```

- **功能描述**

Recoder模块提供了录音的功能。

# int start() —— 开始录音

- **功能描述**

  开始录音
- **参数**

  无
- **返回值**

  0：成功
  -1：失败

# int stop() —— 停止录音

- **功能描述**

  停止录音
- **参数**

  无
- **返回值**

  0：成功
  -1：失败

# int set_listener(Recorder::IListener* listener) —— 注册监听者

- **功能描述**

  设置监听者，监听者可在回调函数中获得录音状态及数据
- **参数**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| Recorder::IListener* | listener | 录音状态和数据的监听者 |

IListener

- **IListener类型定义：**

```
    class IListener {
    public:
        virtual int on_start() = 0;  // 开始录音时调用
        virtual int on_resume() = 0; // 暂时未启用，空实现即可
        virtual int on_data(const void* data, size_t size) = 0; // 接收录
音数据时调用
        virtual int on_pause() = 0;  // 暂时未启用，空实现即可
        virtual int on_stop() = 0;   // 结束录音时调用
        virtual ~IListener() = 0;
    };
```

- **返回值**

  0：成功
  -1：失败

# Recoder模块使用示例代码

```
// 写一个IListener类的派生类，根据IListener类型定义中的说明进行相关实现
class RecorderListener : public Recorder::IListener
{
// 录音数据是通过on_data方法获取的
};


// 实例recorder类
RecorderManager recorder;

// 注册监听者
Recorder::IListener* listener = new RecorderListener();
recorder.set_listener(listener);

// 开始录音
recorder.start();

// 结束录音
recorder.stop();
```

# 5.5 PlayCommandManager类

- **所属头文件**

```
duer_play_command_manager.h
```

- **功能描述**

单例类，提供了几种简单的播放控制接口，如：上一曲、下一曲、循环播放。
通过 `duer::PlayCommandManager::instance()` 获取PlayCommandManager类的实例。

# int next() —— 播放下一曲

- **功能描述**

  播放下一曲
- **调用方式**

```
duer::PlayCommandManager::instance().next();
```

- **参数**

  无
- **返回值**

  0：成功
  -1：失败

# int previous() —— 播放上一曲

- **功能描述**

  播放上一曲
- **调用方式**

```
duer::PlayCommandManager::instance().previous();
```

- **参数**

  无
- **返回值**

  0：成功
  -1：失败

## int repeat() —— 循环播放

- **功能描述**

  循环播放
- **调用方式**

```
duer::PlayCommandManager::instance().previous();
```

- **参数**

  无
- **返回值**

  0：成功
  -1：失败

# 5.6 OTA模块

## 5.6.1 功能简介

OTA模块的提供了设备升级的功能。百度提供了OTA升级功能的服务器，不需要开发者自己搭建维护。在console平台上可以自行上传升级固件，配置OTA升级策略等。

同时，想启动OTA升级功能需要设备端做出相应的开发。
设备端OTA功能：

1. 支持多种下载协议下载升级包
2. 解压缩升级包
3. 校验升级包（通过SHA-1和服务器公钥校验升级包，保证升级包的来源）
4. 校验升级包
5. 提供用户升级包的配置信息（此功能尚未开放，需要开放，请联系产品经理）
6. 提供用户原始上传的文件信息

---

## 5.6.2 OTA模块使用说明

OTA模块的API是用C语言写的，主要包含在lightduer_ota_updater.h、lightduer_ota_unpack.h、lightduer_ota_notifier.h、lightduer_dev_info.h、lightduer_ota_installer.h等头文件中。
使用时需要根据接口包含相应的头文件。

---

# 5.6.3 接口（API）说明

## int duer_init_ota(duer_ota_init_ops_t *ops) —— 初始化OTA模块

- **所属头文件**

```
lightduer_ota_updater.h
```

- **功能描述**

  初始化OTA模块，同时注册3个回调函数，在回调函数中需要实现的功能下面会有详细说明。此接口应该在CA模块连接百度云成功后调用。

- **调用方式**

```
duer_ota_init_ops_t s_ota_init_ops = {
    .init_updater = duer_ota_init_updater,
    .uninit_updater = duer_ota_uninit_updater,
    .reboot = reboot,
};
// 此处调用duer_init_ota接口
duer_init_ota(&s_ota_init_ops);
```

- **参数**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| duer_ota_init_ops_t* | ops | duer_ota_init_ops_t是定义在lightduer_ota_updater.h中的结构，包含3个函数指针 |

duer_ota_init_ops_t定义：

```
typedef struct _duer_ota_init_ops_s {
    int (*init_updater)(duer_ota_updater_t *updater);
    int (*uninit_updater)(duer_ota_updater_t *updater);
    int (*reboot)(void *arg);
} duer_ota_init_ops_t;
```

- **返回值**

  DUER_OK：成功
  other: 失败

- **其他说明**

**int (*init_updater)(duer_ota_updater_t *updater)**

功能描述

此回调函数会在OTA开始升级时调用，函数内部需要开发者实现以下功能：

- 注册安装器（调用duer_ota_unpack_register_installer接口）
- 申请用户自定义资源
- 将用户自定义资源传递给安装器（调用duer_ota_unpack_set_private_data接口）

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| duer_ota_updater_t* | updater | 指向升级器的指针，可用于获取或设置安装器资源 |

返回值

DUER_OK：成功
other: 失败

**int (*uninit_updater)(duer_ota_updater_t *updater)**

功能描述

销毁升级器时调用此函数，开发者需要在此函数内释放用户自定义资源。
可使用duer_ota_unpack_get_private_data接口获取 `updater` 中包含的用户自定义资源。

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| duer_ota_updater_t* | updater | 指向升级器的指针，可用于获取或设置安装器的资源 |

返回值

DUER_OK：成功
other: 失败

**int (*reboot)(void *arg)**

功能描述

升级完成时调用此函数，开发者需要在此函数内实现设备的重启工作。

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| void* | arg | 保留参数，目前无实际意义 |

返回值

DUER_OK：成功
other: 失败

## int duer_ota_unpack_register_installer(duer_ota_unpacker_t *unpacker, duer_ota_installer_t *installer) —— 注册安装器

- **所属头文件**

```
lightduer_ota_unpack.h
```

- **功能描述**

  注册安装器，安装器中包含7个回调函数，需要开发者自行实现，回调函数的具体实现可以参照SDK中的Demo。
- **调用方式**

```
duer_ota_installer_t updater = {
    .notify_data_begin  = ota_notify_data_begin,
    .notify_meta_data   = ota_notify_meta_data,
    .notify_module_data = ota_notify_module_data,
    .notify_data_end    = ota_notify_data_end,
    .update_img_begin   = ota_update_img_begin,
    .update_img         = ota_update_img,
    .update_img_end     = ota_update_img_end,
};
// 此函数在 init_updater回调函数内使用，ota_updater是init_updater回调函数的参数
ret = duer_ota_unpack_register_installer(ota_updater->unpacker, &updater)
;
```

- **参数**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| duer_ota_unpacker_t* | unpacker | 指向解包模块实例的指针 |
| duer_ota_installer_t* | installer | 安装器实例，需要开发者进行实现 |

duer_ota_installer_t定义：

```
typedef struct _duer_ota_installer_s {
    int (*notify_data_begin)(void *cxt);
    int (*notify_meta_data)(void *cxt, duer_ota_package_meta_data_t *met
a);
    int (*notify_module_data)(
            void *cxt,
            unsigned int offset,
            unsigned char *data,
            unsigned int size);
    int (*notify_data_end)(void *cxt);
    int (*update_img_begin)(void *cxt);
    int (*update_img)(void *cxt);
    int (*update_img_end)(void *cxt);
} duer_ota_installer_t;
```

- **返回值**

  DUER_OK：成功
  other: 失败
- **其他说明**

**int (*notify_data_begin)(void *cxt)**

功能描述

开始解压升级包时的通知函数。在开始下载解压升级包前调用此函数。

用户可在这个函数里，创建文件，初始化Flash，使能DMA，擦除Flash等操作。

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| void* | cxt | 这个参数就是用户调用duer_ota_unpack_set_private_data接口传递的用户自定义参数 |

返回值

DUER_OK：成功
other: 失败

**int (*notify_meta_data)(void *cxt, duer_ota_package_meta_data_t *meta)**

功能描述

接收升级包描述信息，此接口暂时无效，需要使用此接口请联系我们。

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| void* | cxt | 用户自定义参数 |
| struct package_meta_data* | meta | 升级包的描述信息 |

返回值

DUER_OK：成功
other: 失败

**int (*notify_module_data)(void *cxt, unsigned int offset, unsigned char *data, unsigned int size)**

功能描述

接收用户上传到云端的原始文件，可以将原始文件保存到文件系统或直接写入到Flash里。升级包会分为多个包传送给设备，此函数会被多次调用。

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| void* | cxt | 用户自定义参数 |
| unsigned int | offset | data指向的数据在整个数据包里的偏移量 |
| unsigned char* | data | 升级包数据 |
| unsigned int | size | data指向的升级数据的大小 |

返回值

DUER_OK：成功
other: 失败

**int (*notify_data_end)(void *cxt)**

功能描述

升级包解压缩完成通知函数，升级包发送完整后调用此函数。
用户可以在这个函数里，释放通知函数申请的资源，并校验收到的数据

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| void* | cxt | 用户自定义参数 |

返回值

DUER_OK：成功
other: 失败

**int (*update_img_begin)(void *cxt)**

功能描述

开始更新升级包通知函数，安装升级包前的资源申请和初始化

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| void* | cxt | 用户自定义参数 |

返回值

DUER_OK：成功
other: 失败

**int (*update_img)(void *cxt)**

功能描述

更新升级包函数，用户开始安装升级包

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| void* | cxt | 用户自定义参数 |

返回值

DUER_OK：成功
other: 失败

**int (*update_img_end)(void *cxt)**

功能描述

更新升级包完毕通知函数，用户可以校验安装是否成功

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| void* | cxt | 用户自定义参数 |

返回值

DUER_OK：成功
other: 失败

## int duer_ota_unpack_set_private_data(duer_ota_unpacker_t *unpacker, void *private_data) —— 设置传递给安装器资源

- **所属头文件**

```
lightduer_ota_unpack.h
```

- **功能描述**

  设置安装器使用的自定义资源，此接口在 `init_updater` 回调函数内使用
- **调用方式**

  **注：**详细的调用方式请参照SDK中 `duer_ota_init_updater` 函数的实现。

```
// 申请资源
struct rda_ota_update_context *ota_install_handle = NULL;
ota_install_handle = (struct rda_ota_update_context *)malloc(sizeof(*ota_
install_handle));

// 整合原有资源和新申请资源
ota_install_handle->updater = ota_updater;

// 设置安装器使用的资源
ret = duer_ota_unpack_set_private_data(ota_updater->unpacker, ota_install
_handle);
```

- **参数**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| duer_ota_unpacker_t * | unpacker | 指向解包模块实例的指针 |
| void* | private_data | 用户自定义的资源 |

- **返回值**

  DUER_OK：成功
  other: 失败

## void *duer_ota_unpack_get_private_data(duer_ota_unpacker_t *unpacker) —— 获取用户传递给安装器资源地址

- **所属头文件**

```
lightduer_ota_unpack.h
```

- **功能描述**

  获取安装器使用的资源地址，可用于释放资源
- **调用方式**

  **注：**详细的调用方式请参照SDK中 `duer_ota_uninit_updater` 函数的实现。

```
struct rda_ota_update_context *ota_install_handle = NULL;
ota_install_handle = duer_ota_unpack_get_private_data(ota_updater->unpack
er);
```

- **参数**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| duer_ota_unpacker_t * | unpacker | 指向解包模块实例的指针 |

- **返回值**

  NULL：获取资源地址失败
  Other：获取到的资源地址

## int duer_ota_register_package_info_ops(duer_package_info_ops_t *ops) —— 注册安装包信息

**所属头文件**

```
lightduer_ota_notifier.h
```

- **功能描述**

  注册安装包信息，参数包含一个回调函数指针。开发者需要在这个回调函数中将安装包信息存入此回调函数的参数中。
- **调用方式**

```
static duer_package_info_ops_t s_package_info_ops = {
    .get_package_info = get_package_info,
};
duer_ota_register_package_info_ops(&s_package_info_ops);
```

- **参数**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| duer_package_info_ops_t* | ops | duer_package_info_ops_t是定义在 `lightduer_ota_notifier.h` 中的结构，包含1个函数指针 |

duer_package_info_ops_t定义

```
typedef struct _duer_package_info_ops_s {
    int (*get_package_info)(duer_package_info_t *info);
} duer_package_info_ops_t;
```

- **返回值**

  DUER_OK：成功
  Other: 失败

## int duer_register_device_info_ops(struct DevInfoOps *ops) ——注册设备信息

**所属头文件**

```
lightduer_dev_info.h
```

- **功能描述**

  注册设备信息，设备信息分为4种，通过4个回调函数传入此接口。
- **调用方式**

```
static struct DevInfoOps dev_info_ops = {
    .get_firmware_version = get_firmware_version,
    .get_chip_version     = get_chip_version,
    .get_sdk_version      = get_sdk_version,
    .get_network_info     = get_network_info,
};
duer_register_device_info_ops(&dev_info_ops);
```

- **参数**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| struct DevInfoOps* | ops | DevInfoOps是定义在 `lightduer_dev_info.h` 中的结构，包含4个函数指针 |

DevInfoOps定义

```
struct DevInfoOps {
    int (*get_firmware_version)(char *firmware_version);
    int (*get_chip_version)(char *chip_version);
    int (*get_sdk_version)(char *sdk_version);
    int (*get_network_info)(struct NetworkInfo *info);
};
```

- **返回值**

  DUER_OK：成功
  Other: 失败
- **其他说明**

  **int (*get_firmware_version)(char *firmware_version)**

  功能描述

  获取固件版本号，需要开发者在此函数内将固件版本号存入 `firmware_version` 的地址下，`firmware_version` 指针有16个字节可用，固件版本号所占用空间不能大于16个字节。

  参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| char* | firmware_version | 保存固件版本号信息 |

返回值

DUER_OK：成功
Other: 失败

**int (*get_chip_version)(char *chip_version)**

功能描述

获取芯片型号，需要开发者在此函数内将芯片型号存入 `chip_version` 的地址下， `chip_version` 指针有20个字节可用，芯片型号所占用空间不能大于20个字节。

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| char* | chip_version | 保存芯片型号信息 |

返回值

DUER_OK：成功
Other: 失败

**int (*get_sdk_version)(char *sdk_version)**

功能描述

获取SDK版本号，需要开发者在此函数内将SDK版本号存入 `sdk_version` 的地址下，`sdk_version` 指针有15个字节可用，SDK版本号所占用空间不能大于15个字节。。

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| char* | sdk_version | 保存SDK版本号信息 |

返回值

DUER_OK：成功
Other: 失败

**int (*get_network_info)(struct NetworkInfo *info)**

功能描述

获取网络信息，需要开发者在此函数内将网络信息存入 `info` 的地址下， `info` 指针有18个字节可用，网络信息所占用空间不能大于18个字节。

参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| char* | info | 网络信息 |

返回值

DUER_OK：成功
Other: 失败

## int duer_ota_set_switch(duer_ota_switch flag) —— 设置OTA使能

- **所属头文件**

```
lightduer_ota_updater.h
```

- **功能描述**

  设置是否使能OTA

- **调用方式**

```
duer_ota_set_switch(true)
```

- **参数**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| duer_ota_switch | flag | `ENABLE_OTA` ：使能， `DISABLE_OTA` ：不使能 |

duer_ota_switch定义

```
typedef enum _duer_ota_switch {
    ENABLE_OTA  =  1,
    DISABLE_OTA = -1,
} duer_ota_switch;
```

- **返回值**

  DUER_OK：成功
  Other: 失败

## int duer_ota_get_switch(void) —— 获取OTA开启状态

- **所属头文件**

```
lightduer_ota_updater.h
```

- **功能描述**

  获取OTA是否开启
- **调用方式**

```
OTASwitch ota_switch = duer_ota_get_switch();
```

- **参数**

  无
- **返回值**

  DUER_OK：开启
  DUER_ERR_FAILED： 关闭

## int duer_ota_set_reboot(duer_ota_reboot reboot) —— 设置OTA更新完后重启状态

- **所属头文件**

```
lightduer_ota_updater.h
```

- **功能描述**

  设置OTA更新完后重启状态
- **调用方式**

```
ret = duer_ota_set_reboot(ENABLE_REBOOT);
```

- **参数**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| duer_ota_reboot | reboot | ENABLE_REBOOT ：重启使能， DISABLE_REBOOT ：重启关闭 |

duer_ota_reboot定义

```
typedef enum _duer_ota_reboot {
    ENABLE_REBOOT  =  1,
    DISABLE_REBOOT = -1,
} duer_ota_reboot;
```

- **返回值**

  DUER_OK：成功
  DUER_ERR_FAILED：失败

## int duer_ota_get_reboot(void) —— 获取OTA更新完后重启状态

- **所属头文件**

```
lightduer_ota_updater.h
```

- **功能描述**

  获取OTA更新完后重启状态
- **调用方式**

```
int rebootStatus = duer_ota_get_reboot();
```

- **参数**

  无
- **返回值**

  DUER_OK：重启成功
  DUER_ERR_FAILED：重启失败

# 5.7 Alarm模块

## 5.7.1 功能简介

闹钟功能是用户通过语音设置闹钟，如“今天晚上8点提醒我喝水”，云端解析后下发设置闹钟指令。闹钟触发后，设备端播放对应的音频资源，如“小度提醒您该喝水了”，闹钟结束后调用对应接口删除闹钟信息。

**注：**云端下发的设置指令中包含token、url、time等内容，开发者需要把这些信息保存到本地，待闹钟触发时可播放url对应的音频资源。

## 5.7.2 Alarm模块使用说明

Alarm模块的API是用C语言写的，主要包含在baidu_alarm.h头文件中。
使用时包含**baidu_alarm.h**头文件即可。

## 5.7.3 类型说明

1. alarm_event_type

   该类型为一个枚举类型，表示各类闹钟事件。

   **定义：**

```
typedef enum {
    SET_ALERT_SUCCESS,     // 设置闹钟成功
    SET_ALERT_FAIL,        // 设置闹钟失败
    DELETE_ALERT_SUCCESS, // 删除闹钟成功
    DELETE_ALERT_FAIL,     // 删除闹钟失败
    ALERT_START,           // 开始闹钟
    ALERT_STOP,            // 停止闹钟
} alarm_event_type;
```

2. duer_set_alarm_func

   该类型指向一回调函数，用于实现设置闹钟的功能。

   **定义：**

```
typedef int (*duer_set_alarm_func)(const int id,
                                   const char *token,
                                   const char *url,
                                   const int time);
```

**参数说明：**

| 类型 | 名称 | 说明 |
| --- | --- | --- |
| int | id | 上报设置闹钟成功（SET_ALERT_SUCCESS）事件时作为参数使用 |
| char* | token | 闹钟的唯一标识 |
| char* | url | 闹钟触发时播报的音频链接 |
| int | time | 闹钟被触发的时间（1970年1月1日至今的时间戳，单位“秒”），根据这个time，通过NTP或RTC校准时间，设置定时器。 |

3. duer_set_alarm_func

   该类型为一个函数指针类型，用于实现一个删除闹钟的功能。

   **定义：**

```
typedef int (*duer_delete_alarm_func)(const int id, const char *toke
n);
```

**参数说明：**

| 类型 | 名称 | 说明 |
| --- | --- | --- |
| int | id | 上报删除闹钟成功（DELETE_ALERT_SUCCESS）事件时作为参数使用 |

| 类型 | 名称 | 说明 |
| --- | --- | --- |
| char* | token | 要删除闹钟的token，开发者通过这个token删除设备上的闹钟 |

---

# 5.7.4 接口（API）说明

## void duer_alarm_initialize(duer_set_alarm_func set_alarm_cb, duer_delete_alarm_func delete_alarm_cb) —— 初始化alarm模块

- **功能描述**
  初始化闹钟功能，注册两个回调函数，分别实现设置闹钟和删除闹钟的功能。
- **调用方式**

```
static int duer_alarm_set(const int id,
                          const char *token,
                          const char *url,
                          const int time)
{
    // 实现设置闹钟
}

static int duer_alarm_delete(const int id, const char *token)
{
    // 实现删除闹钟
}

duer_alarm_initialize(duer_alarm_set, duer_alarm_delete);
```

- **参数**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| duer_set_alarm_func | set_alarm_cb | 回调函数指针，需要开发者在此回调函数中实现设置闹钟的功能 |
| duer_delete_alarm_func | delete_alarm_cb | 回调函数指针，需要开发者在此回调函数中实现删除闹钟的功能 |

- **返回值**

  无

## int duer_report_alarm_event(int id, const char *token, alarm_event_type type) —— 上报闹钟事件

- **功能描述**

  该函数用于上报闹钟事件的执行状态。
  开发者在设置闹钟完成 / 删除闹钟完成后，需要上报SET_ALERT_SUCCESS / DELETE_ALERT_SUCCESS事件后，云端会下发一个play指令，播放一段“成功设置/删除了闹钟”之类的音频。闹钟被触发响铃时也需要上报ALERT_START事件。
- **调用方式**

```
// 上报闹钟事件
duer_report_alarm_event(id, token, SET_ALERT_FAIL);
```

- **参数**

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| int | id | ALERT_START事件上报时该参数应为0，SET_ALERT_SUCCESS / DELETE_ALERT_SUCCES事件上报时该参数为其回调函数传入的id |
| char* | token | 上报事件对应的闹钟token |
| alarm_event_type | type | 上报事件类型，详细类型参照alarm_event_type类型说明 |

- **返回值**

  无