国网电动汽车服务有限公司物联SDK使用手册

目录

[1 概述 1](#_Toc47625703)

[1.1 功能说明 1](#_Toc47625704)

[1.2 SDK结构框图 1](#_Toc47625705)

[2 适用范围 1](#_Toc47625706)

[3 快速体验 1](#_Toc47625707)

[4 系统移植 7](#_Toc47625708)

[4.1 移植指南 7](#_Toc47625709)

[4.2 MCU上集成集成SDK 24](#_Toc47625710)

[4.3 模组上集成SDK 38](#_Toc47625711)

[4.4 高级系统集成SDK 43](#_Toc47625712)

[5 接口说明 46](#_Toc47625713)

[5.1 功能列表 46](#_Toc47625714)

[5.2 功能接口说明 47](#_Toc47625715)

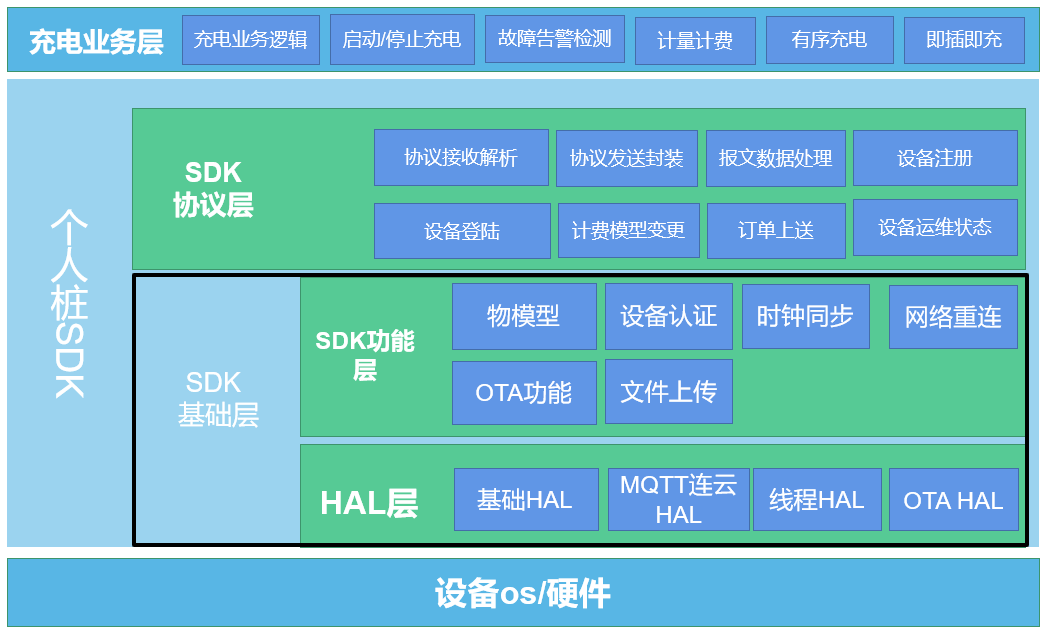
[6 版本变更说明 50](#_Toc47625716)

1. 概述
   1. 功能说明

EVS-SDK提供给设备厂商，由设备厂商集成到设备上后通过该SDK将设备快速安全地接入到国网电动汽车统一物联平台，继而让设备可以被国网电动汽车统一物联平台进行管理。设备需要支持TCP/IP协议栈才能集成EVS-SDK。

* 1. SDK结构框图

充电业务层需要设备厂家调用SDK协议层接口实现。HAL层需要设备厂家根据自己的硬件平台进行实现，其他功能则为SDK自身实现。



1. 适用范围

C语言EVS-SDK适用于使用C语言开发业务处理逻辑的设备，由于C语言运行速度快, 目前大多数的充电设备使用C语言进行功能开发，设备需要支持TCP/IP协议栈或通过AT指令外挂网络通信模组才能集成EVS-SDK。

1. 快速体验

本章描述如何在64位Ubuntu主机上通过物模型的编程方式进行业务报文的上报和接收，编译环境是Ubuntu16.04, 在其它Linux版本上尚未验证过, 推荐安装相同的编译环境以避免碰到兼容性方面的问题。

如果您使用Windows操作系统, 可以安装虚拟机软件Virtualbox获得Linux开发环境, 下载地址: <https://www.virtualbox.org/wiki/Downloads>

然后安装64位的Desktop版本的Ubuntu 16.04.x LTS, 下载地址: <http://releases.ubuntu.com/16.04>

**安装必备软件**

本SDK的开发编译环境使用如下软件: make-4.1, git-2.7.4, gcc-5.4.0, gcov-5.4.0, lcov-1.12, bash-4.3.48, tar-1.28, mingw-5.3.1, gawk-4.1.3

可使用如下命令行安装必要的软件:

$ sudo apt-get install -y build-essential make git gcc gawk

下面介绍如何接入物联网测试平台：

**添加设备**

首先向平台管理人员申请测试账号，测试平台地址： <http://121.196.185.161:8081/>

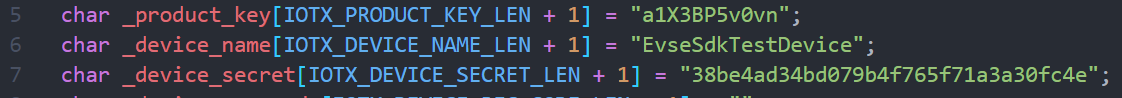
登陆到平台后，选择左侧功能栏中【设备列表】-【设备添加】，添加成功后，生成的设备资产码、设备秘钥、产品密钥即为设备的三元组，在接下来会用到。



**产品功能实现**

填写设备三元组到例程中

将src\dev\_model\examples\device\_example\_pile.c 中的三元组替换成测试平台创建的设备的三元组



**编译与运行程序**

在SDK顶层目录执行如下命令:

$ make distclean

$ make

编译成功完成后，生成的例子程序在当前路径的 output/release/bin 目录下, 名为device\_example\_pile

在SDK顶层目录执行如下命令:

$ ./output/release/bin/device\_example\_pile

**观察数据**

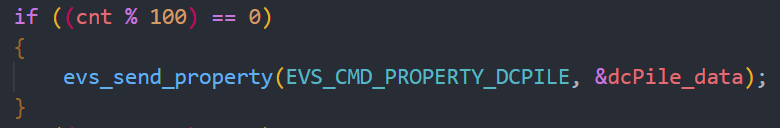
示例程序会定期将dcDeRealIty属性的数值上报云端, 因此可以在平台查看收到的属性等数据；



**属性上报**

示例中使用evs\_send\_property作为上报属性的例子该示例会循环上报各种情况的payload, 用户可观察在上报错误payload时返回的提示信息:

代码中上报属性的代码片段如下:

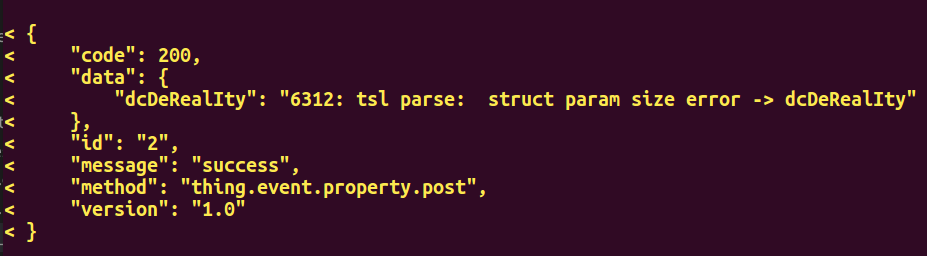
下面是上报正常属性时的日志：

evs\_sdk\_state\_dump.61: EVS--- received state event, -0x0928(pub-uri: /sys/a1X3BP5v0vn/EvseSdkTestDevice/thing/event/propert)

这里是发送给云端的消息：



收到的云端应答：



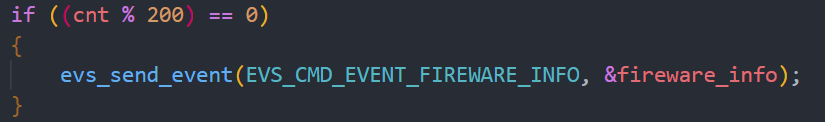
用户回调函数的日志

evs\_report\_reply\_event\_handler.42: EVS--- Message Post Reply Received, Message ID: 44, Code: 200, Reply: {}

**事件上报**

示例中使用 evs\_send\_event 上报属性，该示例会循环上报各种情况的payload，用户可观察在上报错误payload时返回的提示信息:

正常上报事件的情况：



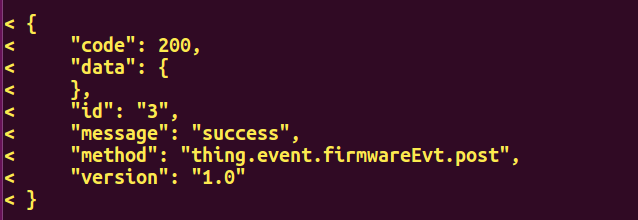
示例程序中 fireware\_info 事件(Event)是约每20s上报一次, 在以上各种情况中循环. 其中正常上报的日志如下:

evs\_sdk\_state\_dump.61: EVS--- received state event, -0x0928(pub-uri: /sys/a1X3BP5v0vn/EvseSdkTestDevice/thing/event/firmwar)

向云端上报的事件消息内容及日志：



从云端收到的应答消息内容及日志：



evs\_sdk\_state\_dump.61: EVS--- received state event, -0x0927(cloud response msgid: 45, code: 200, data: {})

用户回调函数 evs\_trigger\_event\_reply\_event\_handler() 中的日志:

evs\_trigger\_event\_reply\_event\_handler.53: EVS--- Trigger Event Reply Received, Message ID: 45, Code: 200, EventID: firmwareEvt, Message: success

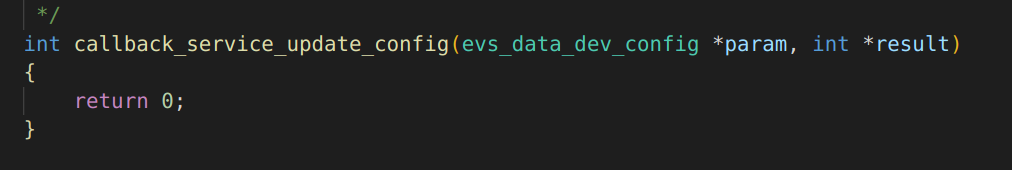
^[aevs\_sdk\_state\_dump.61: EVS--- received state event, -0x0928(pub-uri: /sys/a1X3BP5v0vn/EvseSdkTestDevice/thing/event/propert)

**服务调用**

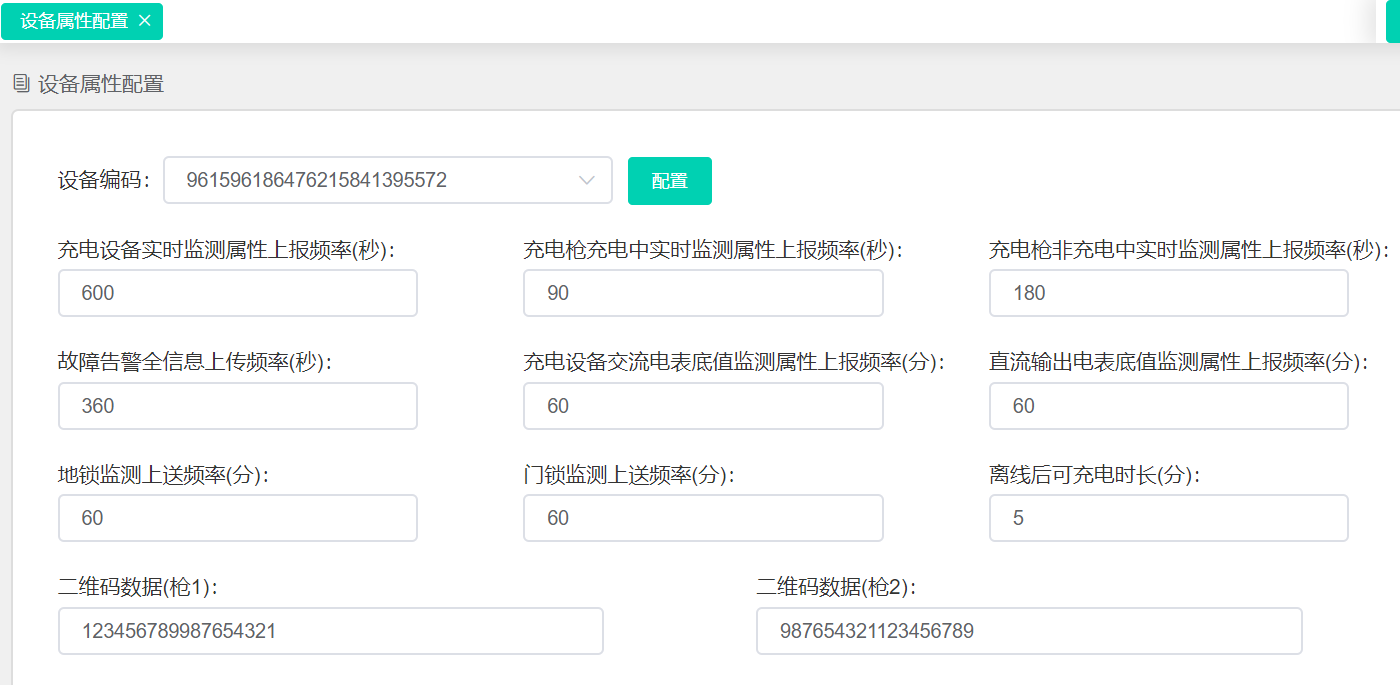
注册服务消息的处理函数，例：

EVS\_RegisterCallback(EVS\_CONF\_UPDATE\_SRV, callback\_service\_update\_config);

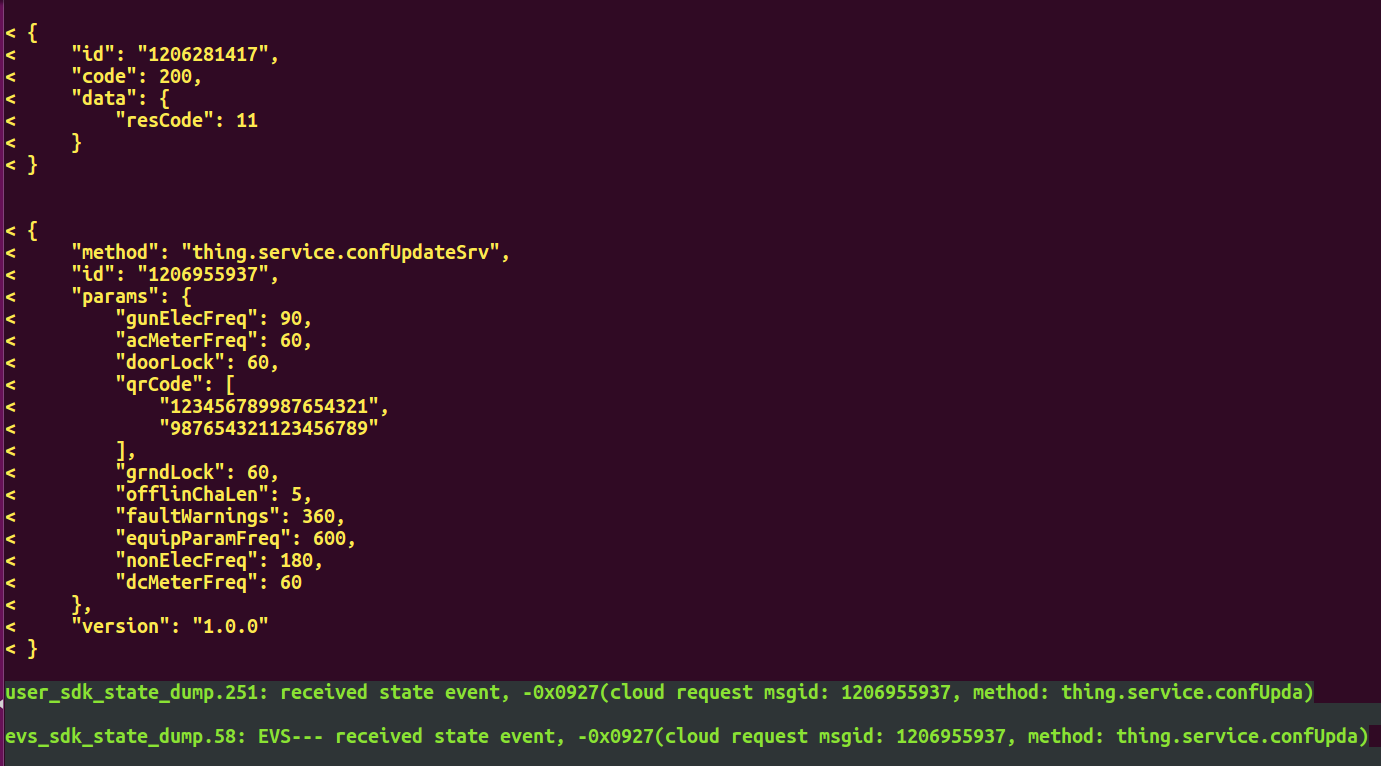
收到平台更新设备配置信息的服务消息后, SDK将对数据进行解析，解析完毕后会将参数送至回调函数callback\_service\_update\_config中如下所示，设备厂家可在此函数中进行参数配置



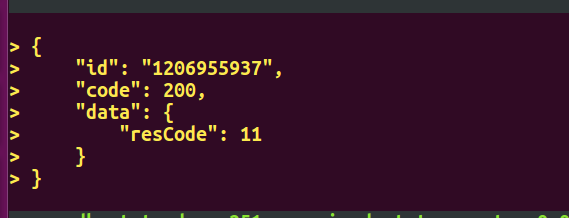
在物联网平台属性配置输入如下参数



配置下发后可在设备端看到如下日志：

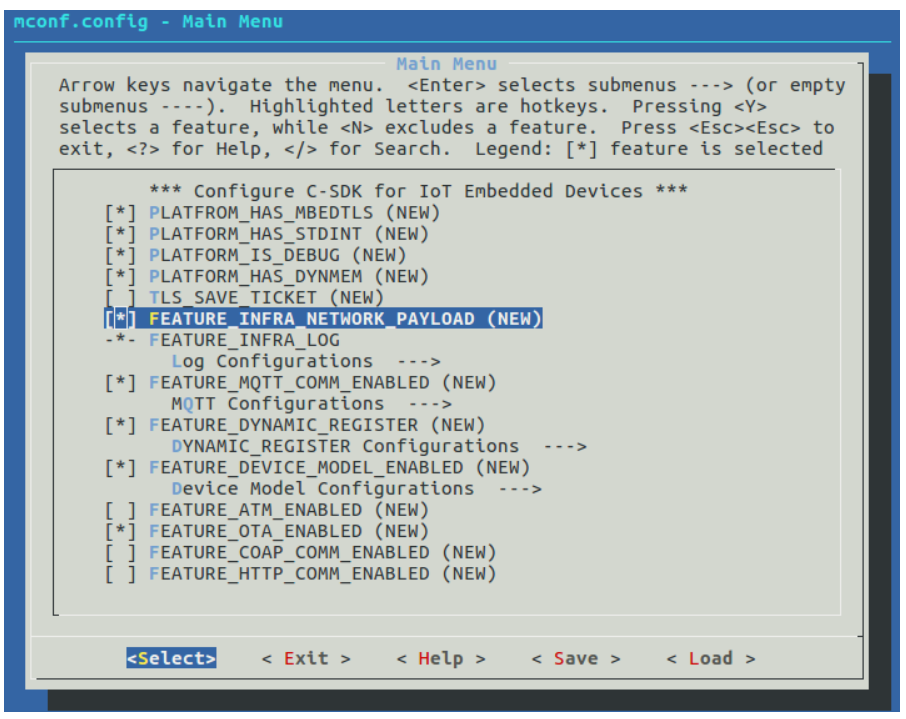


上报设置结果到云端



用 make.settings 文件裁剪 EVS-SDK 详解

EVS-SDK可以用 make menuconfig 命令或者 config.bat 脚本, 分别在Linux/Windows主机上图形化的配置SDK



在上面的界面中

按下空格键可以选中或者失效某个功能, 使用小键盘的上下键来在不同功能之间选择

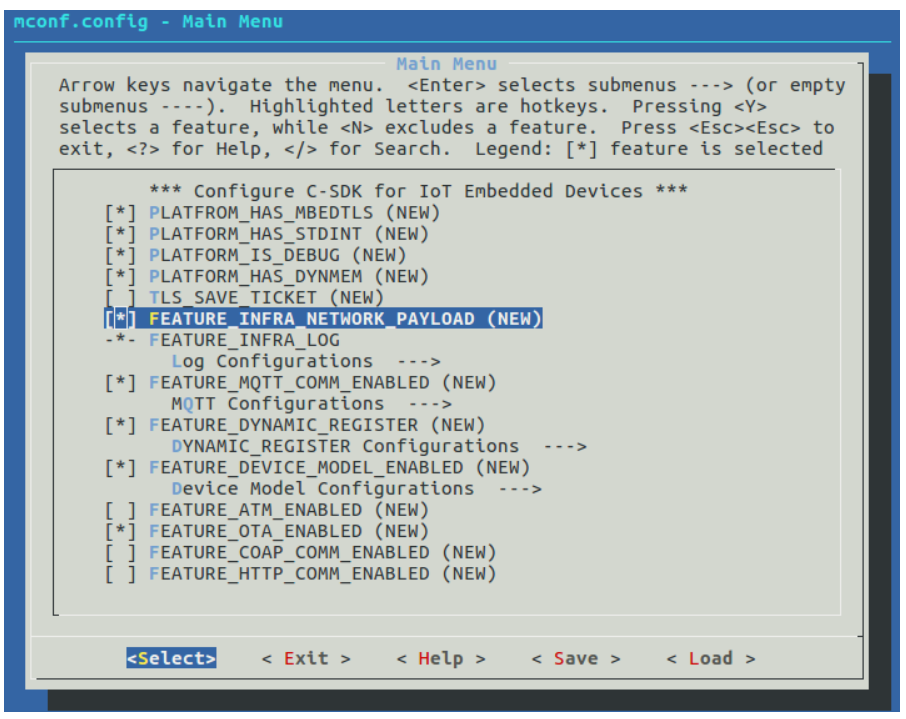
如果想知道每个选项的具体含义, 先用方向键将高亮光条移到那个选项上, 再按键盘上的 **"h"按键**, 将出现帮助文本, 对选项进行详细说明。

注意: 不建议手动编辑 make.settings 文件改动配置, 一切配置都需通过上面的图形界面进行

**PLATFORM\_IS\_DEBUG**

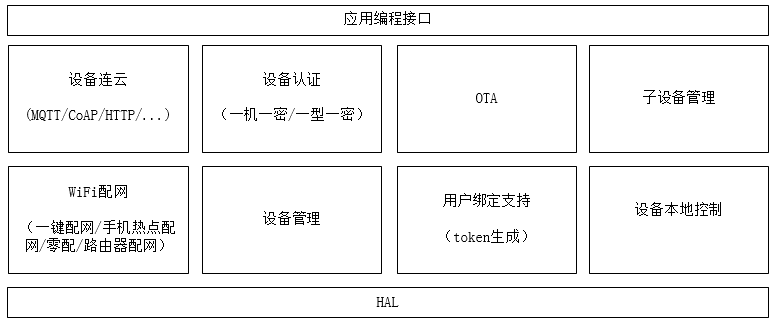
如果设备在开发调试过程中使能该配置，设备接入公网调试环境，在程序进入生产流程后关闭该配置项，设备通过物联卡接入国网生产环境

直连充电设备make menuconfig典型配置如下图所示



1. 系统移植
   1. 移植指南
      1. 移植概述

C语言EVS-SDK的结构如下图所示：



SDK被设计为可以在各种OS/硬件平台上运行，因此凡是与OS、目标硬件相关的操作都被抽象为HAL函数，设备商在开发产品时除了需要调用SDK的API实现产品业务逻辑外，还需要对SDK依赖的HAL函数进行实现。

**a.设备商下载SDK之后，进行的操作如下：**

* 进行HAL函数的实现
* 调用SDK提供的API进行产品功能实现
* 对SDK进行编译

**b.SDK编译方式**

将SDK移植到目标嵌入式平台有两种方式：

* 使用SDK自带的编译系统，当开发环境为Linux时使用
* 抽取SDK的代码文件加入到用户的开发环境编译

**c.使用SDK自带的编译系统**

如果使用Linux作为开发环境，可以直接使用SDK自带的编译系统对SDK进行编译，

使用SDK自带的编译系统时, 您需要

* 准备一台 Linux 的开发主机, 安装 Ubuntu16.04 64位或以上版本
* 运行 make menuconfig 命令配置功能点
* 在 tools/board 目录下新增自己的硬件平台描述文件，进行交叉编译配置
* 运行 make reconfig 选择自己的平台
* 运行 make, 产物是已经交叉编译好的 libiot\_sdk.a, 在 output/release/lib 目录下

用户可以通过4.1.2节了解编译系统如何使用，参考4.1.3节了解交叉编译的过程

**d.抽取代码加入用户的开发环境**

KEIL、IAR这样的开发工具无法使用make，需要将SDK中的功能代码抽取出来加入工程进行编译。

代码抽取以及编译过程如下：

* 准备一台 Linux 的开发主机, 安装 Ubuntu16.04 64bits；或者准备一台 Windows 的开发主机
* 运行 make menuconfig 命令(Ubuntu) 或者 config.bat 脚本(Windows)配置需要使用到的SDK功能点
* 运行 extract.sh 脚本(Ubuntu) 或者 extract.bat 脚本(Windows)根据被选择的功能点抽取代码
* 用户选中的SDK功能相关的代码将被放置在 output/eng 目录下
* 用户将output/eng下的代码添加到开发工具的项目中进行编译

用户可参考4.1.4节了解相关过程。

* + 1. 基于Make的编译说明

本章对make系统进行更详细的描述, 以及讲解如何对SDK进行裁剪。

**常用命令**



**输出说明**

使用make指令进行编译,将会打印类似如下的表格, 这是每个模块的ROM占用, 以及静态RAM占用的统计



用户需要关注的输出产物都在 output/release 目录下:







**配置系统组成部分**

**用户输入**

设备端EVS-SDK的构建配置系统, 有以下输入文件可接受用户的配置, 您可以通过编辑它们, 将配置输入到构建系统中

**功能配置文件:** 即顶层目录的 make.settings 文本文件

**平台配置文件:** 即目录 tools/board 下的 config.xxx.yyy 系列文件, 也称config文件

构建系统最终是依据 config.xxx.yyy 文件进行编译, 然而由于功能配置/裁剪更为常用, 我们将它额外抽取到了 make.settings 中

**config.xxx.yyy** 主要关注目标嵌入式硬件平台的工具链程序和编译/链接选项的指定, 用于跨平台移植

**config.xxx.yyy** 此外也能以 CONFIG\_ENV\_CFLAGS += ... 的语法新增自定义 CFLAGS, 同理 CONFIG\_ENV\_LDFLAGS += ... 可以指定链接选项

**make.settings** 则是在已被确定的目标硬件平台上, 专注于EVS-SDK的功能模块裁剪或者配置, 用于裁剪功能模块

**构建单元**

从工程顶层目录以下, 每一个含有iot.mk的子目录, 都被构建系统认为是一个构建单元；

每一个构建单元, 若相对顶级makefile的路径是bar, foo/bar1, 则可以用make bar, make foo/bar1这样的命令单独编译；

tools/board/config.xxx.yyy

文件名形式为config.<VENDOR>.<MODEL>的文本文件, 会被构建系统认为是硬件平台配置文件, 每个文件对应一个嵌入式软硬件平台；

其中 <VENDOR> 部分, 一般是指明嵌入式平台的软件OS提供方如mxchip, ubuntu, win7等；

另外, 这也会导致构建系统到 $(IMPORT\_DIR)/<VENDOR> 目录下, 寻找预编译库的二进制库文件和头文件，其中 <MODEL> 部分, 一般是标明嵌入式平台的具体硬件型号，如mtk7687, qcom4004等；

不过也可以写上其它信息, 因为构建系统不会去理解它, 比如mingw32, x86-64等；

例如, 您编译时在 make reconfig 之后选择的是 config.arm-linux.demo 配置文件，则构建系统会在 src/ref-impl/hal/os/arm-linux/ 目录下寻找 HAL\_XXX 的实现(V2.3.0及以前版本)，或者构建系统会在 wrappers/os/arm-linux/ 目录下寻找 HAL\_XXX 的实现；

**调试方式**

在make ...命令行中, 设置TOP\_Q变量为空, 可打印工程顶层的执行逻辑, 例如硬件平台的选择, SDK主库的生成等

*make .... TOP\_Q=*

在make ...命令行中, 设置Q变量为空, 可打印模块内部的构建过程, 例如目标文件的生成, 头文件搜寻路径的组成等

*make .... Q=*

可以用make foo/bar单独对foo/bar进行构建, 不过, 这可能需要先执行make reconfig

可以进入.O/foo/bar路径, 看到完整的编译临时目录, 有makefile和全部源码, 所以在这里执行make, 效果和make foo/bar等同

**交叉编译相关**

以下是常在 tools/board/config.xxx.yyy 平台配置文件中使用的变量

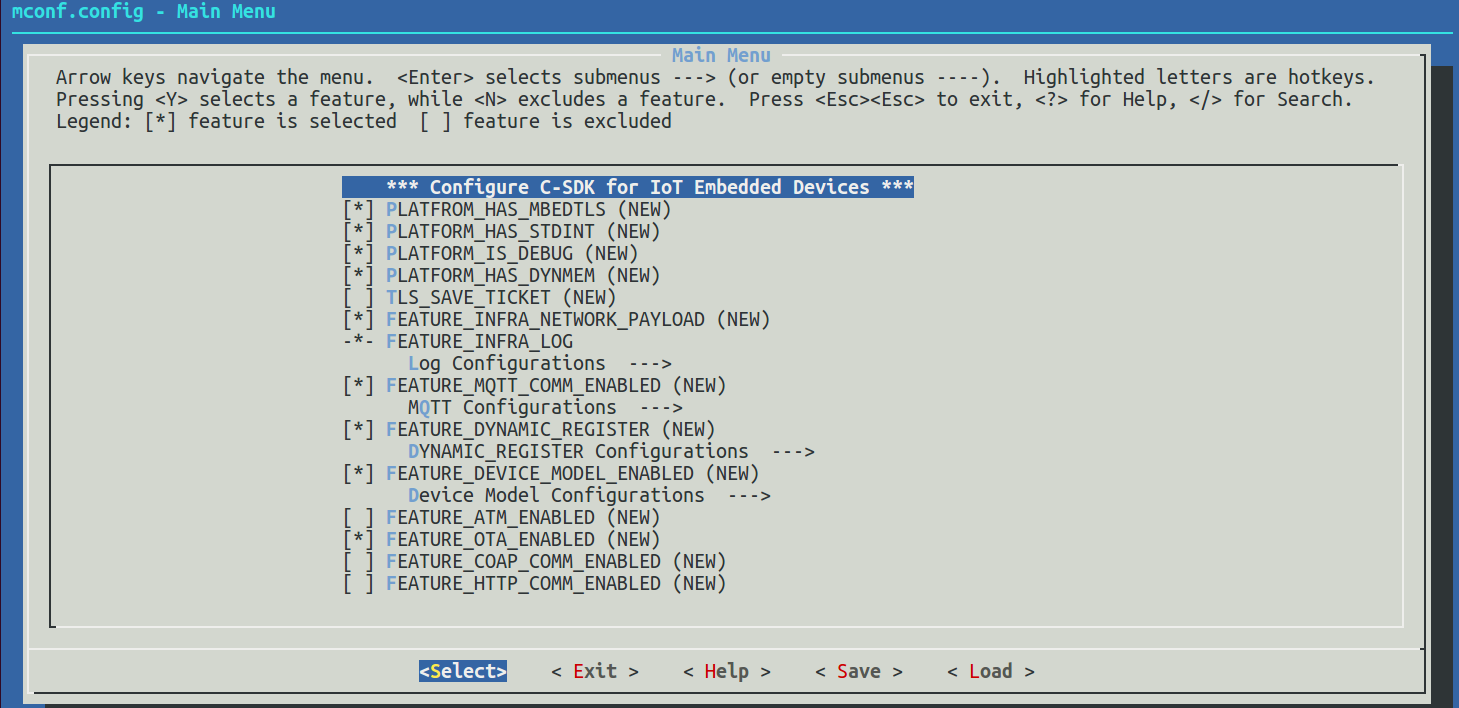
 

**目录文件相关**



**用 make.settings 文件裁剪 EVS-SDK 详解**

用 make menuconfig 命令或者 config.bat 脚本, 分别在Linux/Windows主机上图形化的配置SDK



在上面的界面中

按下空格键可以选中或者失效某个功能, 使用小键盘的上下键来在不同功能之间选择

如果想知道每个选项的具体含义, 先用方向键将高亮光条移到那个选项上, 再按键盘上的 "h"按键, 将出现帮助文本, 对选项进行详细说明。

注意: 不建议手动编辑 make.settings 文件改动配置, 一切配置都需通过上面的图形界面进行

**FEATURE\_MQTT\_COMM\_ENABLED**

MQTT上云功能开关, 所谓MQTT上云是指搭载了EVS-SDK的嵌入式设备和物联网平台服务器之间使用 MQTT 协议进行连接和交互

**FEATURE\_COAP\_COMM\_ENABLED**

CoAP上云功能开关, 所谓CoAP上云是指搭载了EVS-SDK的嵌入式设备和物联网平台服务器之间使用 CoAP 协议进行连接和交互

**FEATURE\_HTTP\_COMM\_ENABLED**

HTTP/S上云功能开关, 所谓HTTP/S上云是指搭载了EVS-SDK的嵌入式设备和物联网平台服务器之间使用 HTTP 协议或 HTTPS 协议进行连接和交互

**FEATURE\_DYNAMIC\_REGISTER**

一型一密/动态注册功能开关, 所谓动态注册是指不需要为同个品类下的不同设备烧录不同的三元组, 只需烧录相同的productSecret, 每个设备在网络通信中动态注册自己

**FEATURE\_DEVICE\_MODEL\_GATEWAY**

高级版网关能力的开关, 配置进行高级版物模型相关的编程时, EVS-SDK是提供 linkkit\_xxx\_yyy() 风格的单品接口, 还是提供 linkkit\_gateway\_xxx\_yyy() 风格的网关接口

**FEATURE\_MQTT\_DIRECT**

MQTT直连功能开关, 所谓MQTT直连是指设备和物联网平台服务器之间使用 MQTT 协议进行连接, 而不会前置基于 HTTP 协议认证的交互过程

**FEATURE\_OTA\_ENABLED**

固件升级功能开关, 所谓固件升级是指设备从物联网平台服务器上下载用户在IoT控制台中上传的固件文件功能

**FEATURE\_DEVICE\_MODEL\_ENABLED**

高级版物模型能力的功能开关, 所谓高级版物模型能力是指设备可使用基于服务/属性/事件三要素的Alink协议和服务端通信

**典型产品的 make.settings 示例**

解压之后, 打开功能配置文件 make.settings, 根据需要编辑配置项, 使用不同的编译配置, 编译输出的SDK内容以及examples都有所不同

以下针对EVS-SDK的客户中, 较多出现的几种产品形态, 给出典型的配置文件, 并在注释中说明为什么这样配置

**不具有网关功能的WiFi模组**

这种场景下客户使用WiFi上行的MCU模组, 比如乐鑫ESP8266, 庆科MK3060等.

这种场景下,设备和物联网平台之间的连接只用于它自己和云端的通信, 不会用于代理给其它嵌入式设备做消息上报和指令下发或固件升级等.

**FEATURE\_MQTT\_COMM\_ENABLED**   = y         # 一般WiFi模组都有固定供电, 所以都采用MQTT的方式上云  
**FEATURE\_MQTT\_DIRECT**         = y         # MQTT直连效率更高, 该选项只在部分海外设备上才会关闭  
**FEATURE\_OTA\_ENABLED**         = y         # 一般WiFi模组的客户, 都会使用物联提供的固件升级服务  
**FEATURE\_DEVICE\_MODEL\_ENABLED** = y         # 一般WiFi模组片上资源充足, 可以容纳高级版, 所以打开  
**FEATURE\_DEVICE\_MODEL\_GATEWAY**  = n         # 如上述说明, 不具备高级版网关功能的场景, 当然关闭这个选项

* + 1. 基于Make的交叉编译示例

**交叉编译到嵌入式硬件平台**

对于嵌入式硬件平台的情况, 编译出目标平台的libiot\_sdk.a, 需要经历如下几个步骤:

* 在tools/board/目录下添加一个对应的配置文件, 文件名规范为config.XXX.YYY, 其中XXX部分就对应后面wrappers/os/XXX目录的HAL层代码
* 在配置文件中, 至少要指定:

交叉编译器 OVERRIDE\_CC 的路径

交叉链接器 OVERRIDE\_LD 的路径

静态库压缩器 OVERRIDE\_AR 的路径

编译选项 CONFIG\_ENV\_CFLAGS, 用于C文件的编译

链接选项 CONFIG\_ENV\_LDFLAGS, 用于可执行程序的链接

* 尝试编译SDK, 对可能出现的跨平台问题进行修正, 直到成功产生目标格式的libiot\_sdk.a
* 最后, 您需要以任何您喜欢的编译方式, 产生目标架构的libiot\_hal.a
* 若目标平台尚未被适配, 则libiot\_hal.a对应的源代码在EVS-SDK中并未包含, 需要您自行实现HAL\_\*()接口

下面以某款目前未官方适配的 arm-linux 目标平台为例, 演示如何编译出该平台上可用的libiot\_sdk.a

**安装交叉编译工具链**

仍以Ubuntu16.04开发环境为例

$ sudo apt-get install -y gcc-arm-linux-gnueabihf

$ arm-linux-gnueabihf-gcc --version

arm-linux-gnueabihf-gcc (Ubuntu/Linaro 4.8.4-2ubuntu1~14.04.1) 4.8.4

Copyright (C) 2013 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO

warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

添加配置文件

$ touch tools/board/config.arm-linux.demo

$ ls tools/board/

config.alios.mk3080 config.arm-linux.demo config.ubuntu.x86

编辑配置文件

在这一步, 需要设置编译选项和工具链, 以及跳过编译的目录

$ vim tools/board/config.arm-linux.demo

CONFIG\_ENV\_CFLAGS = \

-D\_PLATFORM\_IS\_LINUX\_ \

-Wall \

-DNO\_EXECUTABLES

CONFIG\_ENV\_LDFLAGS = \

-lpthread -lrt

OVERRIDE\_CC = arm-linux-gnueabihf-gcc

OVERRIDE\_AR = arm-linux-gnueabihf-ar

OVERRIDE\_LD = arm-linux-gnueabihf-ld

CONFIG\_wrappers :=

**注意, 上面的最后1行表示跳过对** wrappers **目录的编译, 以及** -DNO\_EXECUTABLES **表示不要产生可执行程序。在编译未被适配平台的库时在最初是必要的, 这样可以避免产生过多的错误**

选择配置文件

$ make reconfig

SELECT A CONFIGURATION:

1) config.alios.mk3080

2) config.arm-linux.demo

3) config.ubuntu.x86

#? 2

SELECTED CONFIGURATION:

VENDOR : arm-linux

MODEL : demo

**交叉编译产生库文件libiot\_sdk.a**

注: 本步骤不编译HAL, 只是为了验证配置文件中的交叉编译参数是否正确, 如果出现错误请对配置文件再次进行修改, 直到编译成功

$ make

BUILDING WITH EXISTING CONFIGURATION:

VENDOR : arm-linux

MODEL : demo

[CC] infra\_timer.o <= ...

[CC] infra\_json\_parser.o <= ...

[CC] infra\_preauth.o <= ...

获取交叉编译的产物, 包括静态库和头文件

$ ls -1 output/release/lib/

libiot\_sdk.a

libiot\_tls.a

这里, libiot\_sdk.a文件就是编译好的物联网套件SDK, 已经是ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV)格式, 也就是arm-linux格式的交叉编译格式了

另外, libiot\_tls.a是一个裁剪过的加解密库, 您可以选择使用它, 也可以选择使用平台自带的加解密库, 以减小最终固件的尺寸

$ ls -1 output/release/include/

dev\_model\_api.h

dev\_sign\_api.h

infra

mqtt\_api.h

这里, dev\_sign\_api.h就是使用SDK中”设备签名”功能需要包含的头文件, 类似mqtt\_api.h是使用SDK中”MQTT上云”功能需要的, infra下的头文件也请加入编译搜索路径

**开发未适配平台的HAL层**

对于实现平台抽象层接口 HAL\_XXX\_YYY() 的库 libiot\_hal.a, 不限制其编译和产生的方式

但是如果你愿意的话, 当然仍然可以借助物联网套件设备端EVS-SDK的编译系统来开发和产生它

复制一份HAL层实现代码

注: 在 wrappers/os 下需要创建一个与 tools/board/confg.XXX.YYY 中的 XXX 一样的目录用于存放HAL实现

由于目标平台为arm-linux，因此可以复制Ubuntu下面的HAL实现，下面是操作示例：

$ cd wrappers/os/

$ ls

freertos nos nucleus ubuntu

wrappers/os$ cp -rf ubuntu arm-linux

wrappers/os$ rm -f arm-linux/HAL\_UART\_linux.c

wrappers/os$ ls

arm-linux freertos nos nucleus ubuntu

wrappers/os$ tree -A arm-linux/

arm-linux/

+-- HAL\_AWSS\_linux.c

+-- HAL\_Crypt\_Linux.c

+-- HAL\_FS\_Linux.c

+-- HAL\_KV\_linux.c

+-- HAL\_OS\_linux.c

+-- HAL\_TCP\_linux.c

+-- HAL\_UDP\_linux.c

打开之前被关闭的编译开关

$ vim tools/board/config.arm-linux.demo

CONFIG\_ENV\_CFLAGS = \

-D\_PLATFORM\_IS\_LINUX\_ \

-Wall\

-DNO\_EXECUTABLES

CONFIG\_ENV\_LDFLAGS = \

-lpthread -lrt

OVERRIDE\_CC = arm-linux-gnueabihf-gcc

OVERRIDE\_AR = arm-linux-gnueabihf-ar

OVERRIDE\_LD = arm-linux-gnueabihf-ld

# CONFIG\_wrappers :=

可以看到在CONFIG\_wrappers :=这一行前添加了一个#符号, 代表这一行被注释掉了, wrappers将会进入编译过程

尝试交叉编译被复制的HAL层代码

$ make reconfig

SELECT A CONFIGURATION:

1) config.alios.mk3080

2) config.arm-linux.demo

3) config.ubuntu.x86

#? 2

SELECTED CONFIGURATION:

VENDOR : arm-linux

MODEL : demo

...

$ make

可以看到我们进展的十分顺利, 被复制的代码 wrappers/os/arm-linux/\*.c 直接编译成功了, 产生了 arm-linux 格式的 libiot\_hal.a

**交叉编译样例程序**

这样有了libiot\_hal.a, libiot\_tls.a, 以及libiot\_sdk.a, 已经可以尝试交叉编译样例的可执行程序, 并在目标嵌入式硬件开发板上运行一下试试了

方法是去掉 config.arm-linux.demo 里面的 -DNO\_EXECUTABLES开关, 使得\*/examples/目录下的样例源码被编译出来

注：**删掉**-DNO\_EXECUTABLES**开关时记得把上面一行**-Wall**后面的’\’符号也删掉**

修改后的config.arm-linux.demo内容如下所示：

$ vi tools/board/config.arm-linux.demo

CONFIG\_ENV\_CFLAGS = \

-D\_PLATFORM\_IS\_LINUX\_ \

-Wall

CONFIG\_ENV\_LDFLAGS = \

-lpthread -lrt

OVERRIDE\_CC = arm-linux-gnueabihf-gcc

OVERRIDE\_AR = arm-linux-gnueabihf-ar

OVERRIDE\_LD = arm-linux-gnueabihf-ld

# CONFIG\_wrappers :=

可以看到在 -DNO\_EXECUTABLES 开关从 CONFIG\_ENV\_CFLAGS 中去掉了, 例子可执行程序进入了编译范围

重新载入配置文件, 交叉编译可执行程序

$ make reconfig

$ make

如果有如下的编译输出, 则代表 mqtt-example 等一系列样例程序已经被成功的编译出来, 它们存放在 output/release/bin 目录下

[LD] dev-sign-example <= ...

[LD] mqtt-example <= ...

[LD] linkkit-example-solo <= ...

$ cd output/release/bin/

$ ls

dev-sign-example linkkit-example-solo mqtt-example

$ file \*

dev-sign-example: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked ...

linkkit-example-solo: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked ...

mqtt-example: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked ...

可以用file命令验证, 这些可执行程序确实是交叉编译到 arm-linux 架构上的

**尝试运行样例程序**

接下来, 您就可以把样例程序例如mqtt-example, 用SCP, TFTP或者其它方式, 拷贝下载到您的目标开发板上运行调试了

如果一切顺利, 样例程序和同样例程在 Ubuntu 上运行效果相同, 则证明 wrappers/os/arm-linux 部分的HAL层代码工作正常

如果样例程序运行起来, 和同样例程在 Ubuntu 上运行效果不同, 则需要再重点修改调试HAL实现

也就是指 wrappers/os/arm-linux 目录的HAL层代码, 因为这些代码是我们从 Ubuntu 主机部分复制的,可能并不完全适合 arm-linux

如此反复直到确保 libiot\_hal.a 的开发没问题为止

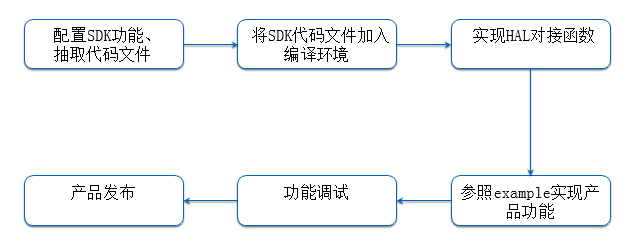
* + 1. 基于代码抽取时的移植说明

对于使用KEIL、IAR进行设备开发的用户来说，不能使用make的方式编译SDK。此时用户可以配置需要的SDK功能，使用SDK提供的抽取工具将相应的代码抽取出来，然后将源文件添加到开发工具中的项目后进行编译； 对于使用Linux作为开发环境的用户，也可以使用本方式将代码抽取出来之后进行交叉编译。

本文将以获取 MQTT上云 和 OTA固件升级 能力为例, 描述在 Windows 开发主机上的SDK移植过程

**过程说明**

设备端的整体开发过程如下所示

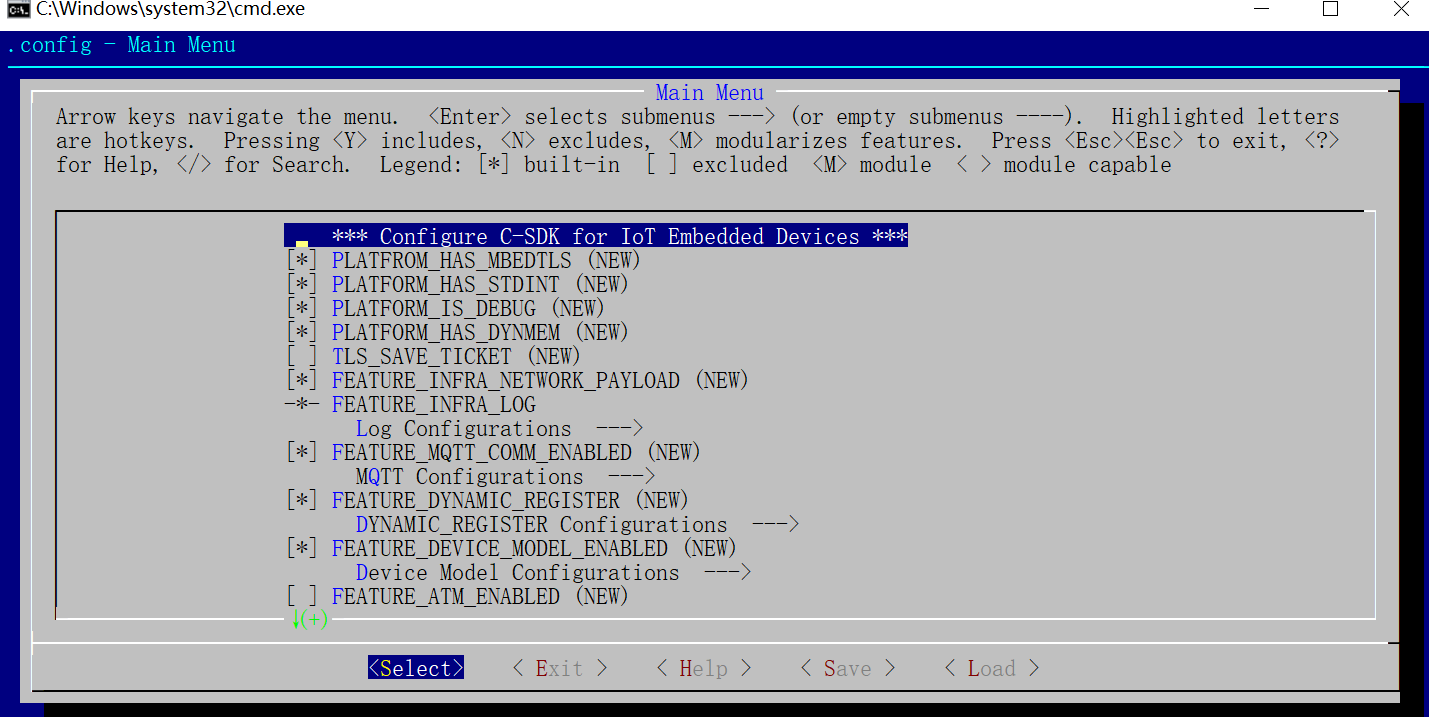
[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552462154505-3ae1610f-4a1d-46e4-9b7b-e8df8039bb75.png#align=left&display=inline&height=181&originHeight=247&originWidth=636&status=done&width=466)

以下是详细步骤讲解

**SDK功能配置**

SDK中有各种功能模块, 用户需要决定需要使用哪些功能, 在本例中假设用户需要”MQTT上云”和”OTA固件下载”的功能

在SDK的根目录双击 config.bat 脚本运行, 弹出如下的功能选择界面(相当于Ubuntu16.04 64位主机上的 make menuconfig 命令)



按下空格键可以选中或者失效某个功能, 使用小键盘的上下键来在不同功能之间切换

如果想知道每个选项的具体含义, 先用方向键将高亮光条移到那个选项上, 再按键盘上的”h”按键, 将出现帮助文本, 对选项进行详细说明

比如:

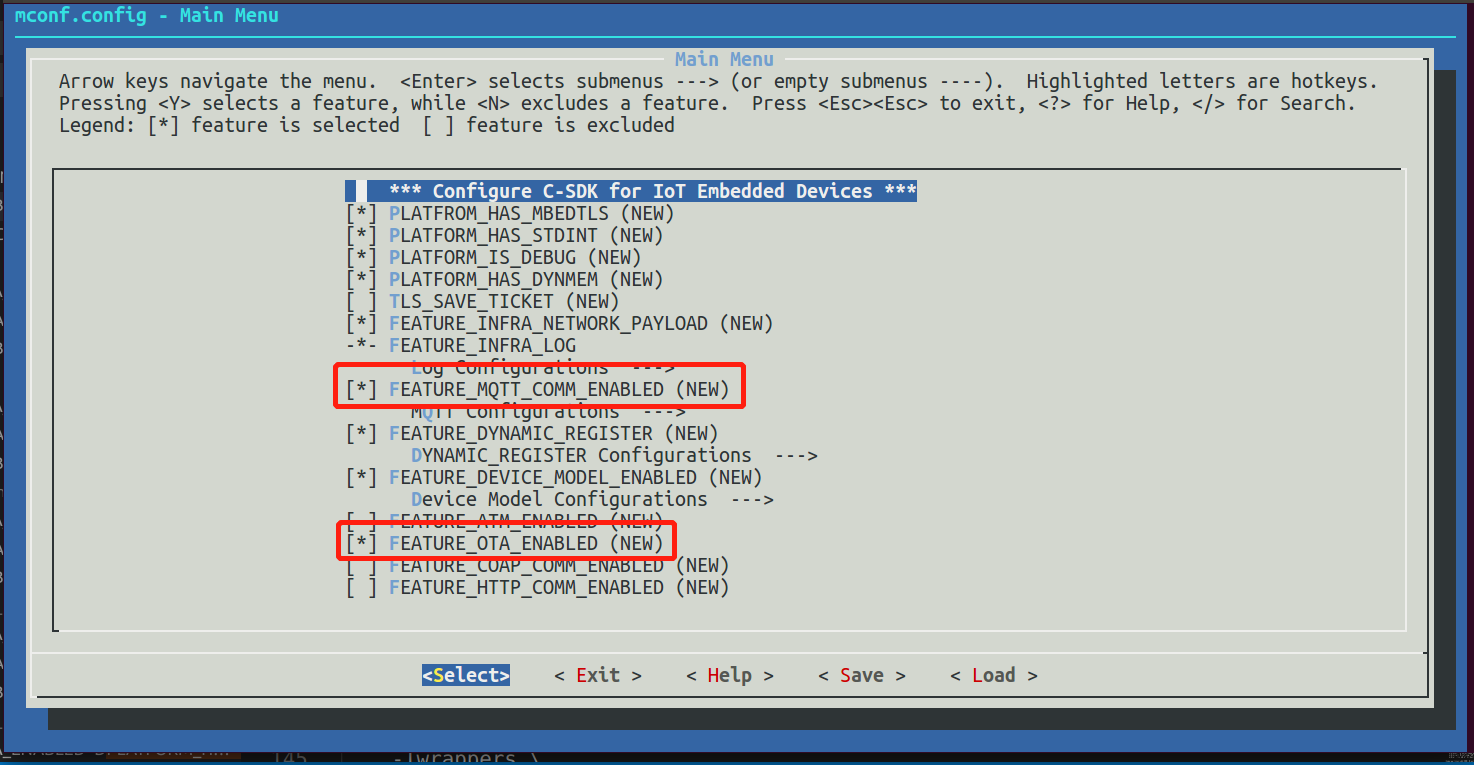
如果编译环境有自带标准头文件, 请使能选项 **PLATFORM\_HAS\_STDINT**

如果目标系统上支持 malloc 和 free 这样的动态内存管理能力, 请使能选项 **PLATFORM\_HAS\_DYNMEM**

在本例中, 我们需要”MQTT上云”和”OTA固件下载”的功能, 所以需要选中

**FEATURE\_MQTT\_COMM\_ENABLED**: 对应”MQTT上云”功能

**FEATURE\_OTA\_ENABLED**: 对应”OTA固件下载”功能



然后将光标移动到窗口底部的 <Exit> 上, 在随后弹出的保存对话框中选 <Yes> 存盘退出配置界面

**SDK代码抽取**

经过上面的步骤, SDK根目录下的 make.settings 功能配置文件的内容就会发生变化, 对应到用户所选择的功能，如下所示：

FEATURE\_MQTT\_COMM\_ENABLED=y

FEATURE\_MQTT\_DEFAULT\_IMPL=y

# FEATURE\_MQTT\_PRE\_AUTH is not set

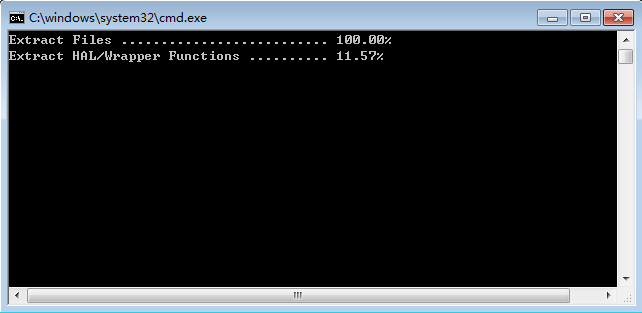
FEATURE\_MQTT\_DIRECT=y

...

FEATURE\_OTA\_ENABLED=y

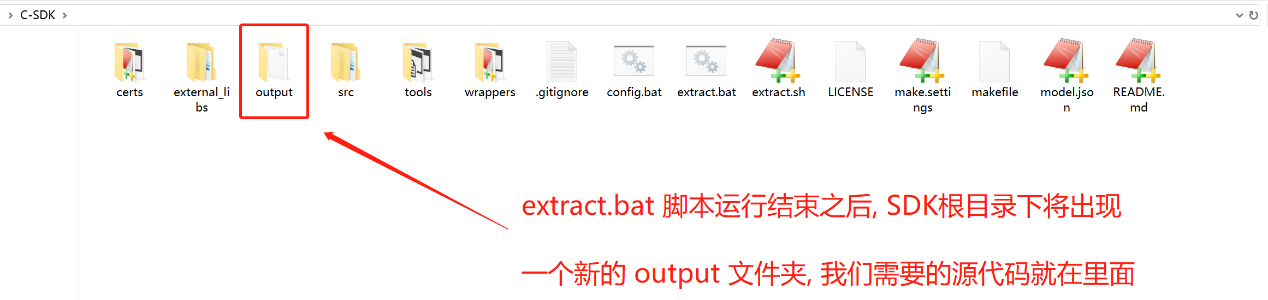
可以注意到其中对应所选功能点的开关 FEATURE\_MQTT\_COMM\_ENABLED 和 FEATURE\_OTA\_ENABLED 都已被设置为 y, 表示打开了

接下来运行代码抽取工具 extract.bat, 该脚本运行之后的界面如下所示：

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552462247790-43756050-7f74-4111-9ab2-2dfbcb03be53.png#align=left&display=inline&height=209&originHeight=313&originWidth=642&status=done&width=428)

**抽取后的代码说明**

如同 config.bat 脚本的输出是 make.settings 文件, extract.bat 脚本的输入是 make.settings, 输出的是 output 文件夹

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552462317956-b9f30f90-fc10-475c-94e9-826b0349d13b.png#align=left&display=inline&height=142&originHeight=549&originWidth=2322&status=done&width=600)

点击进去, 观察它的内容。output目录结构如下

output/

+-- eng

| +-- certs

| +-- dev\_model

| +-- dev\_sign

| +-- infra

| +-- mqtt

| +-- ota

| +-- wrappers

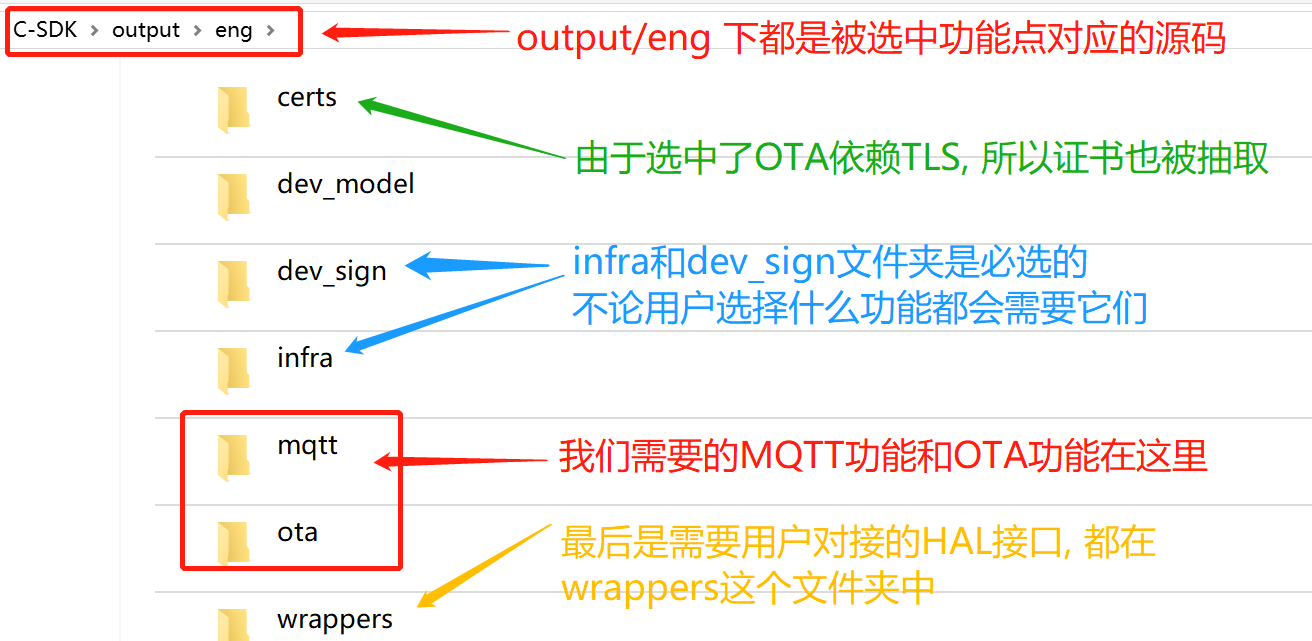
| +-- external\_libs

+-- examples

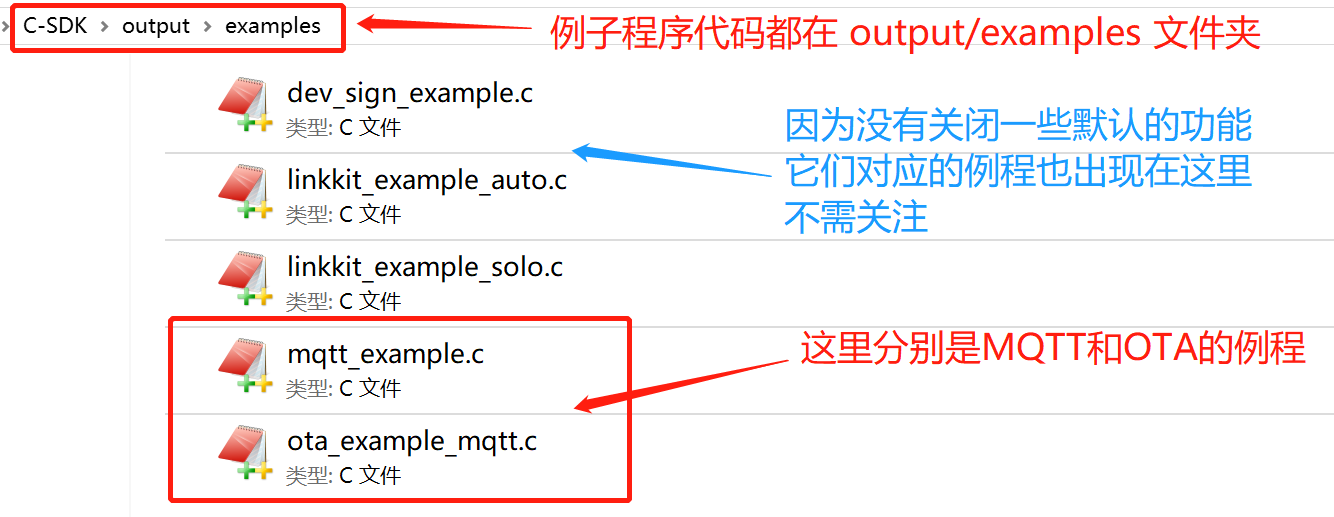
其中 output/eng 目录就对应的是SDK相关功能抽取出来的代码, output/example对应使用SDK中API接口的例子程序



在本例中, eng下面的目录如下所示：

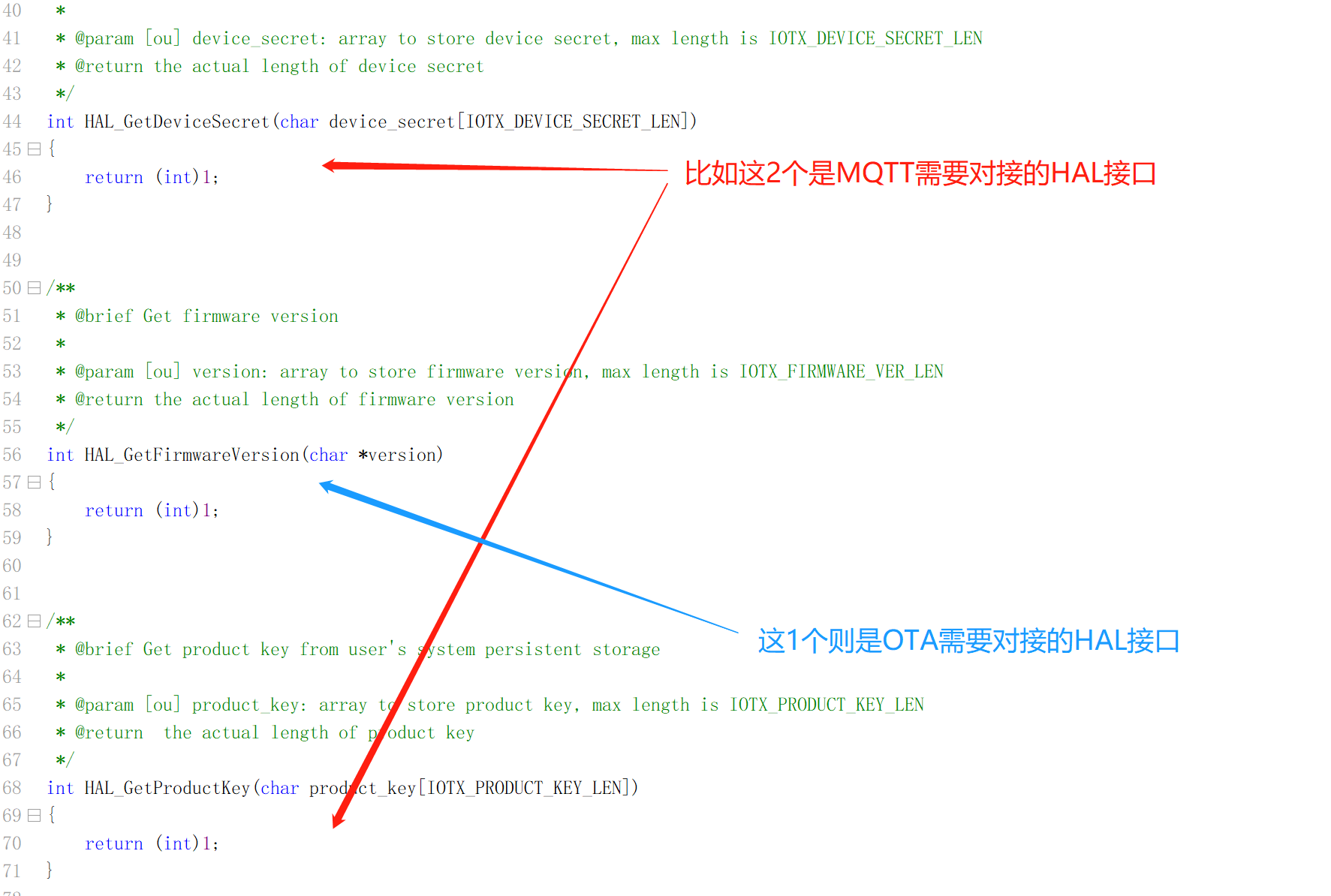
[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552462441620-69f4b0fb-5902-4f9f-b01c-bf6023908542.png#align=left&display=inline&height=293&originHeight=641&originWidth=1312&status=done&width=600)

这些功能点的例子程序都在 output/examples 文件夹

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552462509968-7acf1d53-cafb-4265-8d27-33d78f72e58b.png#align=left&display=inline&height=233&originHeight=517&originWidth=1334&status=done&width=600)

**对接HAL接口或者wrapper接口**

SDK运行时对外界所需要的依赖都以 HAL\_XXX或者 wrapper\_xxx进行定义,并放于文件output\eng\wrappers\wrapper.c中，用户需要对其中的函数进行实现。

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552462727250-2988e953-07c1-4403-b2fc-0d1428ddbeed.png#align=left&display=inline&height=353&originHeight=1194&originWidth=1773&status=done&width=524)

对每个 HAL\_XXX 接口都有详细的注释和文档, 用户可以参考SDK根目录下的wrappers子目录中的wrapper参考实现。

**其它**

抽取出来的代码使用make进行编译

用户也可能使用 GNU Make对抽取出来的代码进行编译, 因此在抽取出来的目录下仍然提供了一个示例的makefile, make之后将会生成以下内容：



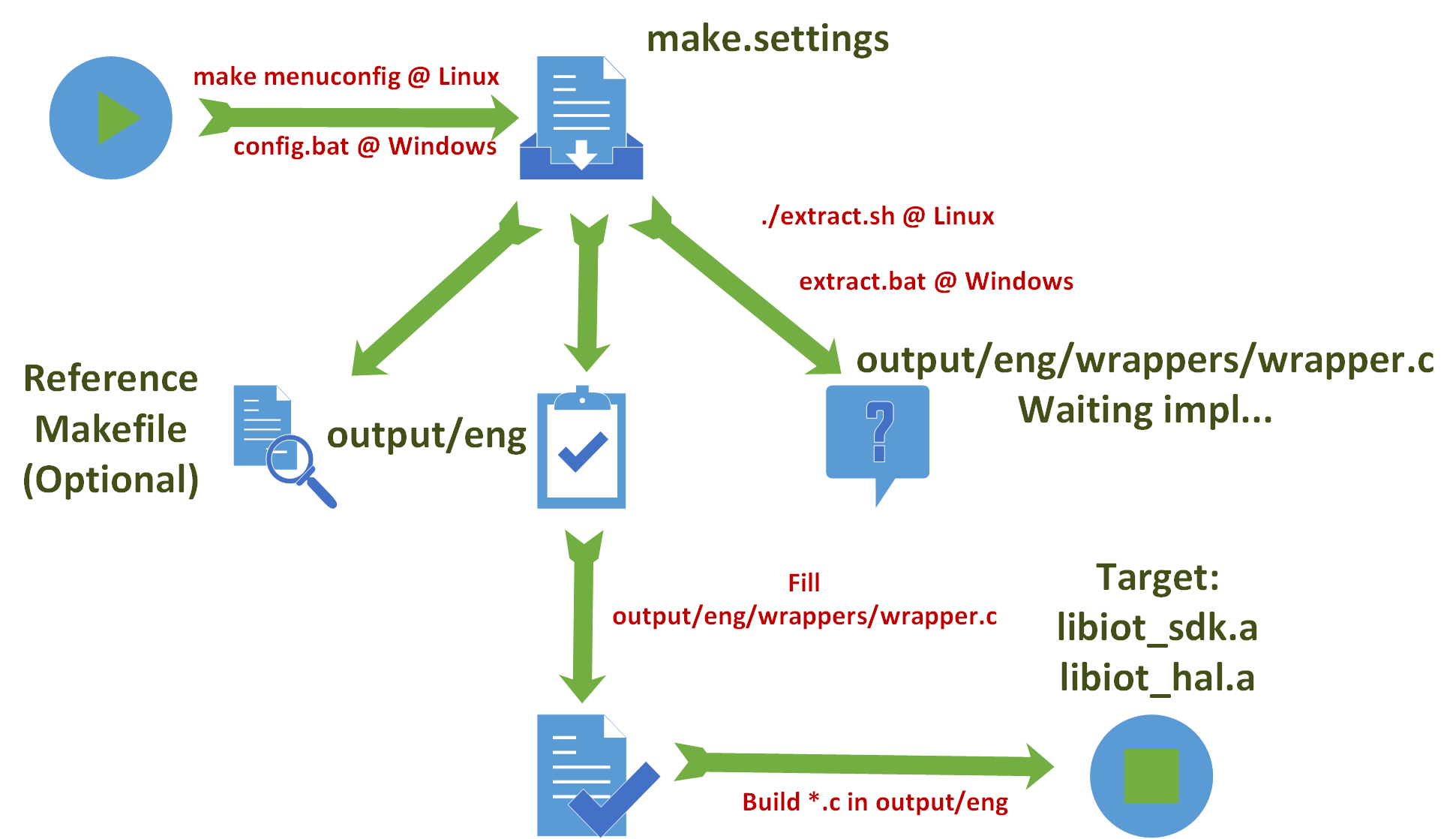
**使用C++编译器编译时错误的处理**

C++编译器在编译的时候判断比gcc判断更加严格，如果编译的时候出错，请编译时：

增加 -fpermissive -w, 去掉 -ansi -Wdeclaration-after-statement

**总结**

以图中红色文字表示用户执行的动作, 用绿色文字表示用户得到的产物, 则这个过程可用下图表示为

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552463211465-22401092-6555-4dfd-83f6-3b2e2088b3f1.png#align=left&display=inline&height=430&originHeight=1118&originWidth=1941&status=done&width=746)

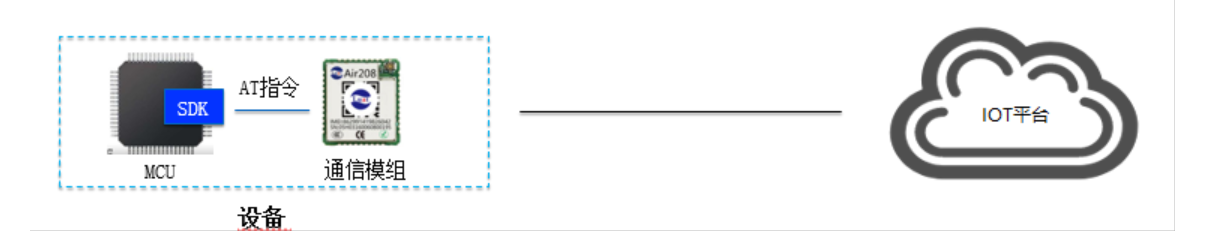
图中左上角是移植过程的开始, 到右下角是移植过程的结束, 以获得目标设备架构的二进制库 libiot\_sdk.a 和 libiot\_hal.a 为标志

到此为止整个SDK的抽取都已经讲解完毕, 您可以将 output/eng 下的所有目录加入自己的工程中编译和集成, 并参考 output/examples 下的例程开始调用SDK提供的API了

* 1. MCU上集成集成SDK
     1. MCU+支持TCP的模组

应用场景: 设备的硬件由一个MCU加上一个通信模组构成, 设备的应用逻辑运行在MCU上, 模组上支持了TCP但是并不支持MQTT, MCU通过模组提供的AT指令来控制模组何时连接云端服务以及收发数据。

本示例中：示例app + SDK + TCP模组驱动 一起消耗大概11KB内存。



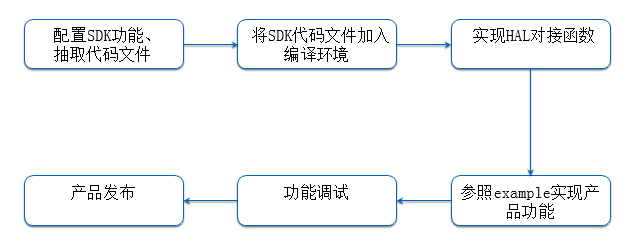
对于这样的场景, 设备厂商需要将EVS-SDK集成并运行在MCU上, 让EVS-SDK通过通信模组连接到物联网平台。

**文档目标**

下面的文档关注于讲解用户如何把SDK移植到MCU, 并与通信模组协作来与物联网平台通信. 为了简化移植过程, 下面的文档在MCU上以开发一个基础版产品作为案例进行讲解, 如果用户需要在MCU上使用SDK的其它功能, 可以在MCU上将基础版的example正确运行之后, 再重新配置SDK, 选中其它功能再进行产品功能开发。

**设备端开发过程**

设备端的开发过程如下所示：

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552465358224-8e1b5114-80f2-4cac-b48e-d83415f1994b.png)

**SDK配置与代码抽取**

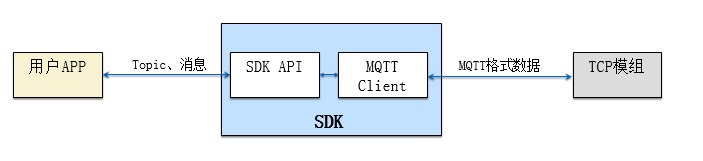
SDK中有各种功能模块, 用户需要决定：

需要使用哪些功能(SDK配置)

SDK提供了配置工具用于配置需要使能哪些功能, 每个功能的配置选项名称类似FEATURE\_MQTT\_XXX, 下面的章节中会讲解具体有哪些功能可供配置。

**SDK如何与外部模组进行数据交互**

SDK使用MQTT与物联网平台通信, 对于模组只支持TCP的情况, 意味着MCU上需要使能SDK自带的MQTT Client, 由MQTT Client将用户数据封装成MQTT协议之后通过通信模组上的TCP模块将数据发送到物联网平台，如下图所示。

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552465442143-464f0101-3de4-434f-8234-7f3995bb0406.png)

MQTT Client与模组协作的时候, 开发者需要编写一个TCP连接管理模块去实现如下功能：

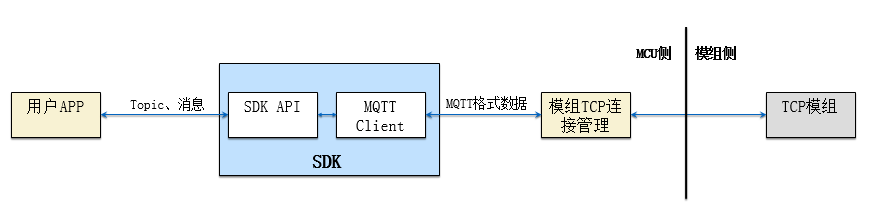
控制模组发起到物联网平台的TCP连接, 并记录模组返回的TCP连接ID。

当MQTT Client发送数据时, 将数据通过MQTT Client创建的模组TCP连接ID进行数据发送。

接收来自模组的字符流数据, 只有是从MQTT Client建立的TCP连接ID中的数据才能发送给MQTT Client。

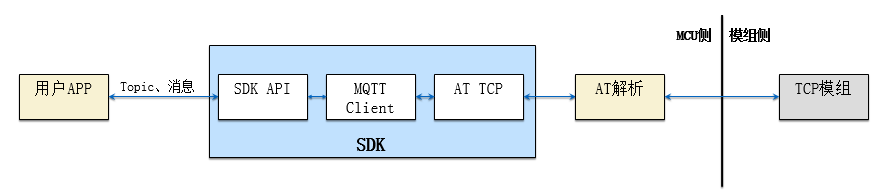
如果MQTT Client接收和处理数据的速度慢于模组发送数据给MCU的速度, 开发者还需要将数据进行缓存以避免数据丢失。

该TCP连接管理模组与SDK/TCP模组关系如下图所示：

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552465509963-841c410b-7390-4a5f-86f7-778dcf1de8a7.png)

EVS-SDK中包含了一个模组TCP连接和数据缓存的模块, 称为AT TCP, 如果开发者在MCU上尚未实现这样一个模组TCP连接和数据缓存的模块, 可以使用SDK提供的AT TCP。

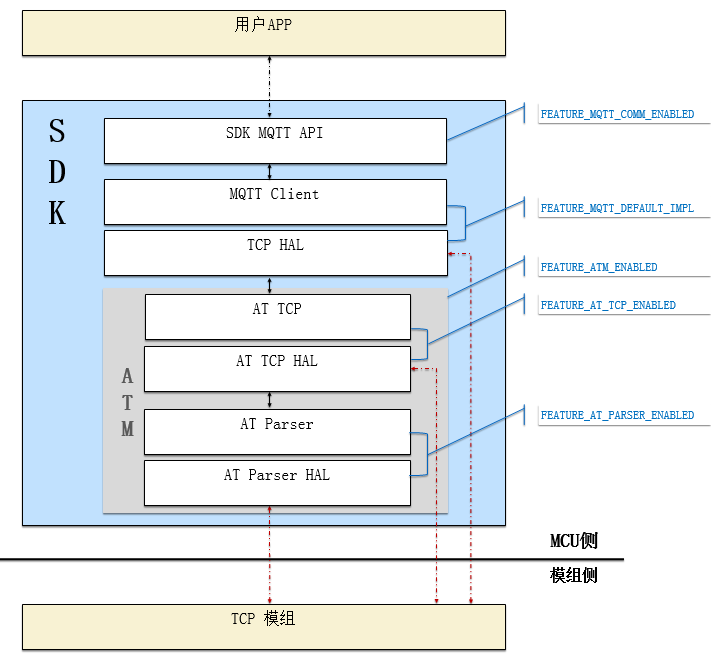
MCU与模组之间通常使用UART进行连接, 因此开发者需要开发代码对UART进行初始化, 通过UART接收来自模组的数据, 由于UART是一个字符一个字符的接收数据, 因此开发者还需要对收到的数据组装并判断AT指令是否承载TCP数据, 如果是才能将TCP数据发送给TCP连接管理模块. 这个模块与SDK/模组的关系如下图所示。

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552465555393-43f6228c-339a-4264-be6c-33bedde60967.png)

EVS-SDK中包含了一个AT解析的模块, 称为AT Parser, 如果开发者尚未实现这样的一个功能模块, 可以使能EVS-SDK中的AT Parser模块以减少开发工作量。

**SDK与模组对接结构说明**

下面是SDK与TCP模组的可能对接方式的图示, 以及相关的配置选项。

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552465591278-2ebb90ad-2ea9-4edf-86d2-cb8cf593c17f.png)

上图中SDK外部的蓝色配置选项表示该选项使能后将会使能的SDK功能模块, 每根红色虚线代表一种可能的SDK与模组的对接方式, HAL表示该模块与外部模块交互的函数定义。



**配置SDK**

SDK包含的功能较多, 为了节约对MCU RAM/Flash资源的消耗, 用户需要根据自己的产品功能定义需要SDK中的哪些功能。

**运行配置命令**

**Linux系统**

进入SDK的根目录下, 运行命令

make menuconfig

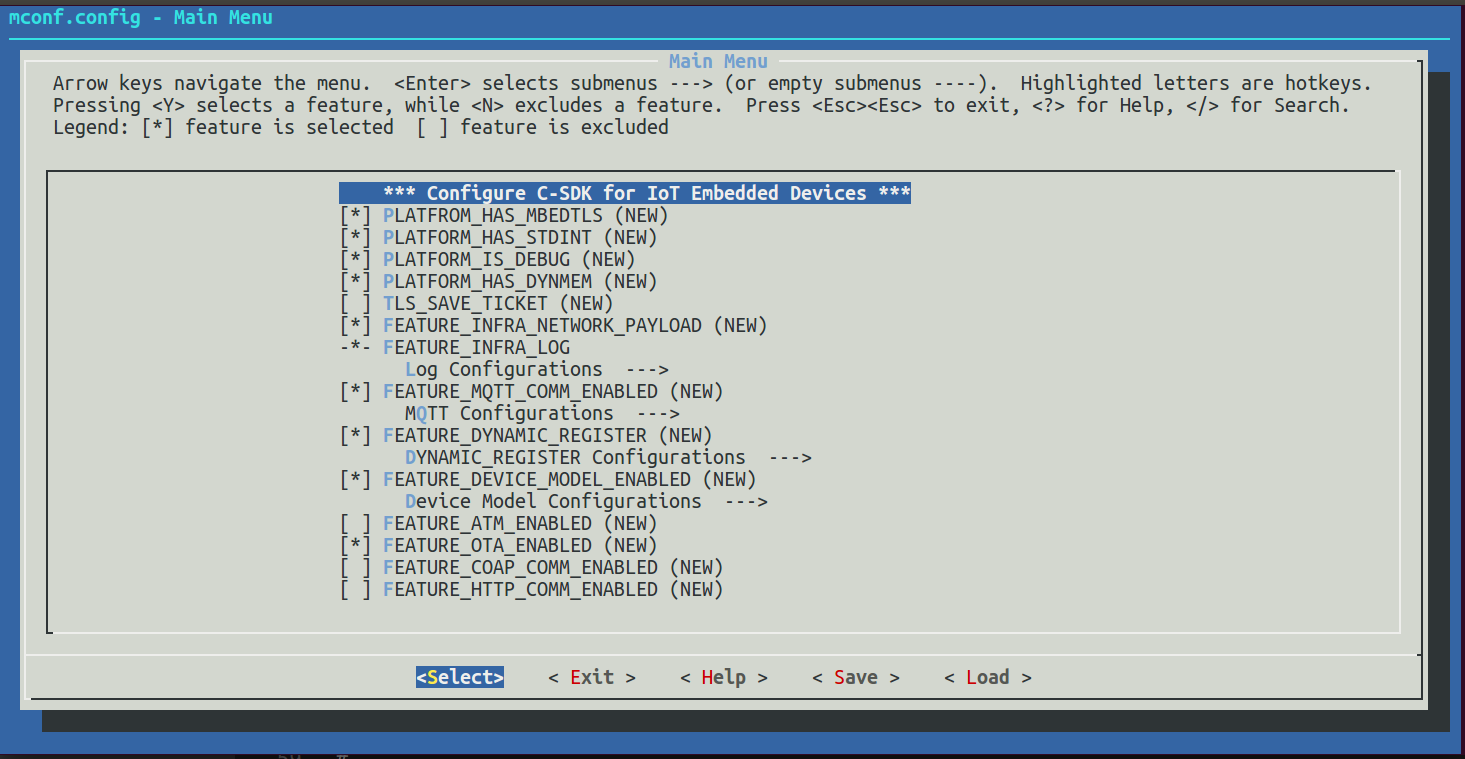
**Windows系统**

运行SDK根目录下的config.bat

config.bat

**使能需要的SDK功能**

运行上面的命令之后, 将会跳出下面的功能配置界面. 按下空格键可以选中或者失效某个功能, 使用小键盘的上下键来在不同功能之间切换. 如果想知道每个选项的具体含义, 先用方向键将高亮光条移到那个选项上, 再按键盘上的"h"按键, 将出现帮助文本, 说明选项是什么含义, 打开了和关闭了意味着什么。



如果编译环境有自带标准头文件<stdint.h>, 请使能选项。

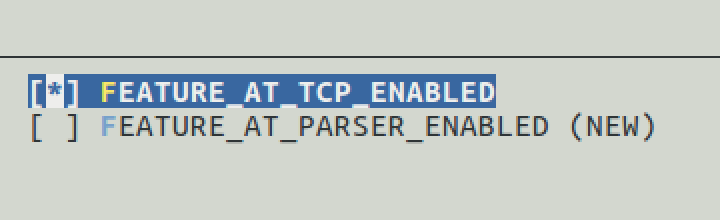
**PLATFORM\_HAS\_STDINT**

本场景中由于模组支持TCP但是不支持MQTT, 因此必须使能下面两项配置。

**FEATURE\_MQTT\_COMM\_ENABLED**, 使用物联SDK提供的MQTT API与云端通信。

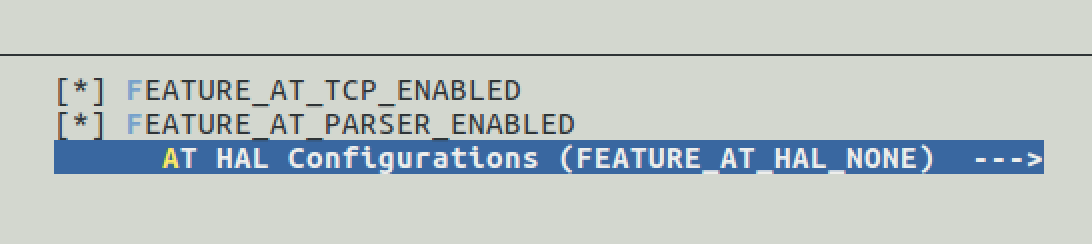
**FEATURE\_MQTT\_DEFAULT\_IMPL**, 使用物联SDK中自带的MQTT Client实现, 用户需要实现相关的TCP连接的创建/连接/数据收发过程。

**FEATURE\_ATM\_ENABLED** 如果希望使用物联提供的AT TCP或者AT Parser，请使能本选项。当本模块使能之后, 还会出现是否使能AT TCP的配置选项。

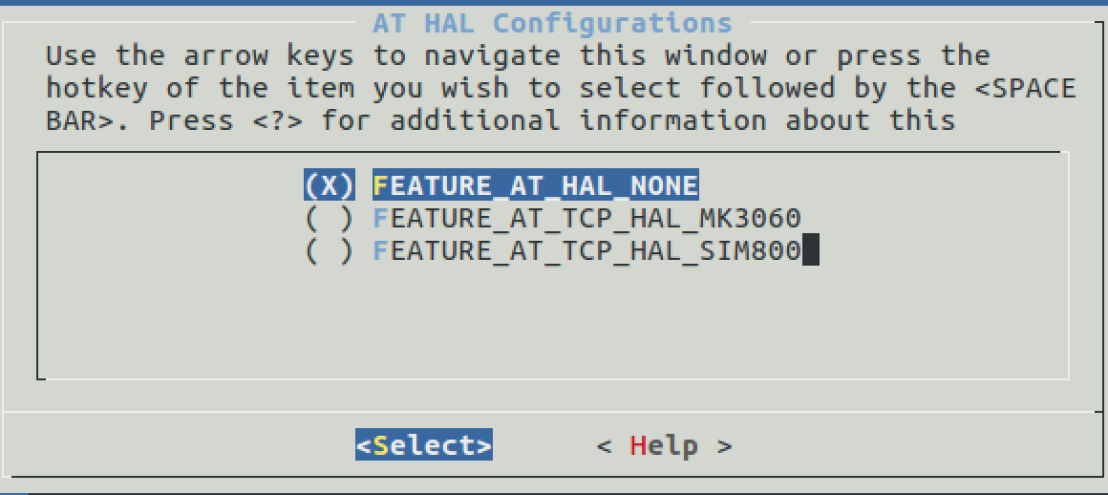
[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552465778586-b48d5ec8-ff29-4cb0-b99f-3167f7b21fb0.png)

开发者可以根据产品的实际情况选择是否使能ATM以及AT TCP. 如果开发者使能了ATM, 但是开发者没有用于AT收发/解析的框架, 可以选择使用at\_parser框架(非必须)。

**FEATURE\_AT\_PARSER\_ENABLED**

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552465806648-18b703b3-df2e-47f3-bdd5-243f190ce30c.png)

SDK基于at\_parser提供了已对接示例, 如果模组是sim800 2G模组或者mk3060 Wi-Fi模组, 可以进行进一步选择模组的型号, 可以让SDK将相应的HAL实现也包含在抽取的代码中. 如果不需要对接示例, 请忽略该步骤。

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552465841801-d807a468-86e5-4dfb-9ff1-6bf649c4a74f.png)

完整的配置开关说明表格如下, 但最终解释应以上面提到的"h"按键触发文本为准。



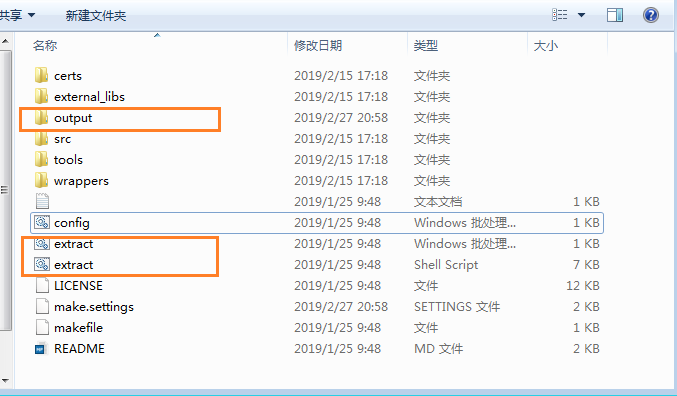




使能需要的SDK配置后, 保持配置并退出SDK配置工具

**抽取选中功能的源代码**

运行SDK根目录下的extract.bat([Linux下运行extract.sh](http://xn--Linuxextract-nx4so836dttxa.sh)), 客户选中的功能所对应的代码将会被放置到文件夹output, 如下图所示。

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1552466361204-0ad2eb3b-09c5-42a2-a1f3-02fb0ea0e407.png)

**实现HAL对接函数**

EVS-SDK被设计为可以在不同的操作系统上运行, 或者甚至在不支持操作系统的MCU上运行, 因此与系统相关的操作被定义成一些HAL函数, 需要客户进行实现. 另外, 由于不同的通信模组支持的AT指令集不一样, 所以与通信模组上TCP相关的操作也被定义成HAL函数需要设备开发者进行实现。

由于不同的用户使能的SDK的功能可能不一样, 因此需要对接的HAL函数会不一样, 设备开发者只需要实现位于文件output/eng/wrappers/wrapper.c中的HAL函数。下面对所有可能出现在文件wrapper.c的HAL函数进行讲解。

**MCU系统相关HAL**

**必须实现函数:**

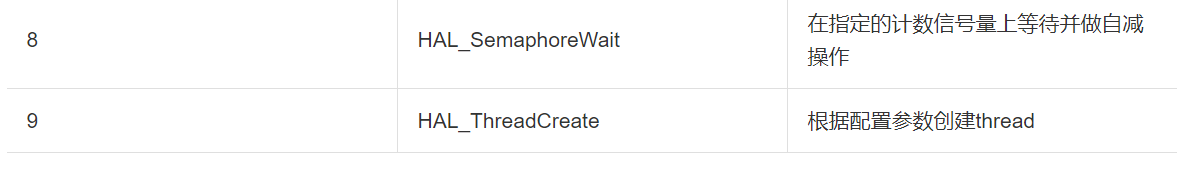


对以上函数若需了解更多细节, 请参考API接口文档。

**OS相关可选函数**

如果MCU没有运行OS, 或者SDK的MQTT API并没有在多个线程中被调用, 以下函数可以不用修改；在有OS场景下并且MQTT API被APP在多个线程中调用, 则需要用户对接以下函数:





对以上函数若需了解更多细节, 可参考API接口文档。

**TCP相关HAL**

如果用户未选择ATM, 用户适配时调用模组提供的TCP AT指令实现四个TCP HAL函数。下面是对这些函数的说明，也可API接口文档了解更多说明。



**AT TCP相关HAL**

如果用户选择使用ATM以及AT TCP, 并且未使能AT Parser, 用户需要实现下面表格中的HAL函数. 下面是对这些函数的说明:





**AT TCP HAL对接示例**

(1)HAL\_AT\_CONN\_Init

该函数完成HAL层数据初始化，下面的代码示例中, 创建了容量为LINK\_ID\_MAX的TCP连接数组, 用于对应模组上创建的TCP连接. 用户还需要在此处完成与模组必要的交互, 例如sim800模组对接时需要附着网络/获取ip地址等。

typedef struct link\_s {

int fd;

....

} link\_t;

static link\_t g\_link[LINK\_ID\_MAX];

int HAL\_AT\_CONN\_Init(void)

{

int link;

memset(g\_link, 0, sizeof(g\_link));

for (link = 0; link < LINK\_ID\_MAX; link++) {

g\_link[link].fd = -1;

}

...

inited = true;

return 0;

}

(2)HAL\_AT\_CONN\_Deinit

该函数完去初始化, 将HAL\_AT\_CONN\_Init()中分配的资源释放，下面的示例中只是简单的将inited变量设置为false。

int HAL\_AT\_CONN\_Deinit(void)

{

if (!inited) {

return 0;

}

inited = false;

return 0;

}

(3)HAL\_AT\_CONN\_Start

该函数用于建立TCP连接，下面的示例代码主要包括从g\_link数组中获得空闲的TCP连接元素/记录该连接的地址/端口等信息, 向模组发送拼接生成的建立TCP连接的AT指令. 用户需要将该函数实现修改为实际使用的模组提供的AT指令, 并对返回数据进行相应处理。

int HAL\_AT\_CONN\_Start(at\_conn\_t \*c)

{

int link\_id;

for (link\_id = 0; link\_id < LINK\_ID\_MAX; link\_id++) {

if (g\_link[link\_id].fd >= 0) {

continue;

} else {

g\_link[link\_id].fd = c->fd;

break;

}

}

...

/\* 拼接AT命令 \*/

snprintf(cmd, START\_CMD\_LEN, "%s=%d,%s,%s,%d",

START\_CMD, link\_id, start\_cmd\_type\_str[c->type],

c->addr, c->r\_port);

....

/\* 发送AT命令 \*/

at\_send\_wait\_reply(cmd, strlen(cmd), true, out, sizeof(out), NULL);

LOGD(TAG, "The AT response is: %s", out);

if (strstr(out, CMD\_FAIL\_RSP) != NULL) {

goto err;

}

return 0;

err:

// error handle

}

(4)HAL\_AT\_CONN\_Close

该函数用于关闭TCP连接，下面的示例代码主要包括记录向模组发送动态生成的AT指令, 然后删除linkid与fd的映射. 其中, fd\_to\_linkid()是fd向linkid转换函数。

int HAL\_AT\_CONN\_Close(int fd, int32\_t remote\_port)

{

int link\_id;

char cmd[STOP\_CMD\_LEN] = {0}, out[64];

link\_id = fd\_to\_linkid(fd);

if (link\_id < 0 || link\_id >= LINK\_ID\_MAX) {

LOGE(TAG, "No connection found for fd (%d) in %s", fd, \_\_func\_\_);

return -1;

}

snprintf(cmd, STOP\_CMD\_LEN - 1, "%s=%d", STOP\_CMD, link\_id);

LOGD(TAG, "%s %d - AT cmd to run: %s", \_\_func\_\_, \_\_LINE\_\_, cmd);

at\_send\_wait\_reply(cmd, strlen(cmd), true, out, sizeof(out), NULL);

LOGD(TAG, "The AT response is: %s", out);

if (strstr(out, CMD\_FAIL\_RSP) != NULL) {

LOGE(TAG, "%s %d failed", \_\_func\_\_, \_\_LINE\_\_);

goto err;

}

...

g\_link[link\_id].fd = -1;

}

(5)HAL\_AT\_CONN\_Send

该函数用于向模组发送数据, 主要工作是向模组发送拼接生成AT命令与数据. 下面的示例代码是向模组上指定的TCP socket发送数据, 用户需要将AT指令修改为实际连接模组对应的AT指令以及进行相应处理。

int HAL\_AT\_CONN\_Send(int fd,

uint8\_t \*data,

uint32\_t len,

char remote\_ip[16],

int32\_t remote\_port,

int32\_t timeout)

{

int link\_id;

char cmd[SEND\_CMD\_LEN] = {0}, out[128] = {0};

if (!data) {

return -1;

}

link\_id = fd\_to\_linkid(fd);

if (link\_id < 0 || link\_id >= LINK\_ID\_MAX) {

LOGE(TAG, "No connection found for fd (%d) in %s", fd, \_\_func\_\_);

return -1;

}

/\* AT+CIPSEND=id, \*/

snprintf(cmd, SEND\_CMD\_LEN - 1, "%s=%d,", SEND\_CMD, link\_id);

/\* [remote\_port,] \*/

if (remote\_port >= 0) {

snprintf(cmd + strlen(cmd), 7, "%d,", remote\_port);

}

at\_send\_data\_2stage((const char \*)cmd, (const char \*)data, len, out, sizeof(out));

}

(6)HAL\_AT\_CONN\_DomainToIp

该函数用于域名解析, 向模组发送动态生成的查询命令后, 根据已知格式解析回复。

int HAL\_AT\_CONN\_DomainToIp(char \*domain, char ip[16])

{

char cmd[DOMAIN\_CMD\_LEN] = {0}, out[256] = {0}, \*head, \*end;

snprintf(cmd, DOMAIN\_CMD\_LEN - 1, "%s=%s", DOMAIN\_CMD, domain);

/\* 发送查询命令 \*/

at\_send\_wait\_reply(cmd, strlen(cmd), true, out, sizeof(out), NULL);

LOGD(TAG, "The AT response is: %s", out);

if (strstr(out, AT\_RECV\_SUCCESS\_POSTFIX) == NULL) {

LOGE(TAG, "%s %d failed", \_\_func\_\_, \_\_LINE\_\_);

return -1;

}

/\* 根据已知格式解析回复 \*/

...

}

**调用接收函数**

用户在AT HAL层收到数据后需要调用API接口IOT\_ATM\_Input(见atm/at\_api.h), 将数据交付给上层。

void handle\_recv\_data()

{

struct at\_conn\_input param;

...

/\* 读取AT指令中数据长度信息 \*/

len = atoi(reader);

if (len > MAX\_DATA\_LEN) {

LOGE(TAG, "invalid input socket data len %d \r\n", len);

return;

}

/\*分配接收buffer, 用户也可以直接使用一个静态数组用于数据接收\*/

recvdata = (char \*)aos\_malloc(len);

if (!recvdata) {

LOGE(TAG, "Error: %s %d out of memory, len is %d. \r\n", \_\_func\_\_, \_\_LINE\_\_, len);

return;

}

/\* 读取数据 \*/

ret = at\_read(recvdata, len);

if (ret != len) {

LOGE(TAG, "at read error recv %d want %d!\n", ret, len);

goto err;

}

if (g\_link[link\_id].fd >= 0) {

param.fd = g\_link[link\_id].fd;

param.data = recvdata;

param.datalen = len;

param.remote\_ip = NULL;

param.remote\_port = 0;

/\* 向上层交付数据 \*/

if (IOT\_ATM\_Input(&param) != 0) {

at\_conn\_hal\_err(" %s socket %d get data len %d fail to post to at\_conn, drop it\n",

\_\_func\_\_, g\_link[link\_id].fd, len);

}

}

...

}

其中, struct at\_conn\_input定义见at\_wrapper.h。

struct at\_conn\_input {

int fd; /\* 数据上送需要操作的句柄 \*/

void \*data; /\* 接收到的数据(该部分内存由底层自行释放)\*/

uint32\_t datalen; /\* 接收到的数据长度 \*/

char \*remote\_ip; /\* 该数据的源地址, 为可选参数, 可以传入NULL(该部分内存由底层自行释放)\*/

uint16\_t remote\_port; /\* 该数据的源端口, 为可选参数, 可以传入0 \*/

};

**AT Parser相关HAL**

如果选择了at\_parser框架, 则需要对接以下四个UART HAL函数, 函数声明见at\_wrapper.h. 如果用户不使用at\_parser框架请忽略该步骤。



**产品相关HAL**

下面的HAL用于获取产品的身份认证信息, 设备厂商需要设计如何在设备上烧写设备身份信息, 并通过下面的HAL函数将其读出后提供给SDK。



对以上函数若需了解更多细节, 可参考API接口文档。

**代码集成**

如果设备商的开发环境使用makefile编译代码, 可以将SDK抽取出来的代码加入其编译环境进行编译. 如果设备商使用KEIL/IAR这样的开发工具, 可以将SDK抽取出来的代码文件加入到IDE的工程中进行编译。

下面是将抽取出来的output目录复制到Linux下, 并在output目录下创建的一个makefile的示例。

注: 配置时使能了ATM模块/使用AT TCP/AT Parser/并选择使用SIM800作为外接模组。

SDK\_PWD = $(shell pwd)/eng

SDK\_DIRS = $(SDK\_PWD)/dev\_sign $(SDK\_PWD)/atm $(SDK\_PWD)/infra $(SDK\_PWD)/mqtt $(SDK\_PWD)/wrappers

SDK\_SOURCES = $(foreach dir,$(SDK\_DIRS),$(wildcard $(dir)/\*.c))

SDK\_OBJS = $(patsubst %.c,%.o,$(SDK\_SOURCES))

SDK\_INC\_DIRS = $(foreach dir, $(SDK\_DIRS),-I$(dir) )

TARGET = testmqtt

all:eng/examples/mqtt\_example\_at.o $(SDK\_OBJS)

$(CC) -o $(TARGET) $(SDK\_OBJS) eng/examples/mqtt\_example\_at.o

clean:

rm -rf \*.o $(TARGET) $(SDK\_OBJS)

%.o:%.c

$(CC) -c $(SDK\_INC\_DIRS) $< -o $@

注:

上面的makefile仅供参考, 用户配置SDK时选用的功能不一样会导致目录出现差别, 用户需要将除了eng/examples外的目录加入编译系统/工具。

用户如果复制该makefile使用, 在复制/粘贴时all/clean/%.o:%.c下一行的命令语句可能出现多个空格/或者没有空格就直接是命令, 如果发现在命令前有空格需要将空格全部删除, 然后增加一个Tab键, 以避免make出错。

至此, SDK在MCU与模组之间的适配开发已结束, 用户可以进行产品业务功能的实现。

* 1. 模组上集成SDK
     1. 在支持TCP的广域网模组上集成SDK

EVS-SDK是物联网平台提供的用于将设备连接到物联网平台的设备端SDK，用于完成设备认证、数据通信等功能。将EVS-SDK集成到通信模组中，可以带来以下好处：

设备厂商在MCU上无需关心如何连接物联网平台，只是通过调用模组提供的AT指令就可以连接物联网平台，因此对MCU的资源消耗没有增加

物联将在认证合作伙伴页面露出通过认证的模组型号、购买链接、开发指导等文档，引导设备商以及服务提供商购买通过认证的通信模组连接物联网平台。

使用集成了SDK的模组开发设备的示意图如下所示：



设备商开发设备的流程为：

购买集成了物联网平台EVS-SDK的模组

在MCU上通过模组提供的AT指令连接物联网平台，以及从物联网平台收发数据

在物联网平台上部署云端服务，对设备进行管理

对于模组商而言，在模组上需要完成的工作包括：

将EVS-SDK正确的集成到模组上

提供相应的连接物联网平台物联网的AT指令供MCU调用

下面的文档只讲解如何将SDK的MQTT功能集成到模组上。对于模组商来说，集成SDK的功能越多，MCU侧设备厂商的开发功能越少，因此建议模组商尽可能多的集成SDK的功能，比如OTA、物模型等。

**在物联网平台上的操作**

为了验证模组上集成的SDK是否运行正确，需要将一个测试设备连接到物联网平台。用户需要在物联网平台上创建一个设备以获取到设备的身份信息。

登录物联网平台之后，创建一个交流或直流桩设备，在设备列表页面可以获取到产品的product\_key、device\_name以及device\_secret 等信息

**集成SDK的开发过程**

模组商在模组上集成SDK时，需要进行下面几个开发过程。

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1553178757608-75a270fc-3d77-4e46-8e9b-8db292920427.png#align=left&display=inline&height=43&originHeight=62&originWidth=904&status=done&width=626)

**SDK配置与代码抽取**

**配置SDK**

SDK包含的功能较多，下面讲解如何配置本场景中需要的功能。

**运行配置命令**

Linux系统

进入SDK的根目录下，运行命令

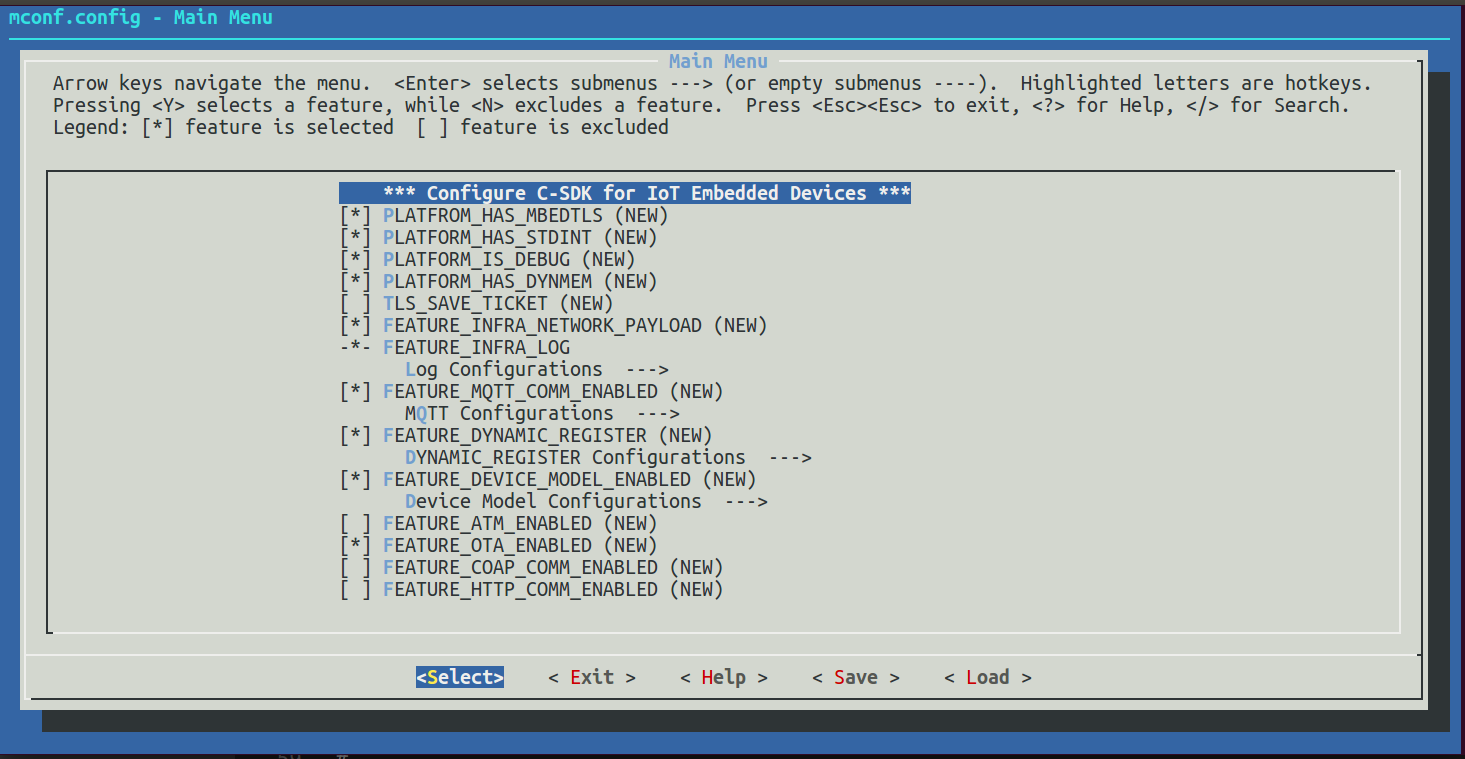
make menuconfig

Windows系统

运行SDK根目录下的config.bat

config.bat

上面的两种方式都会启动SDK的配置工具，界面如下所示：



**使能需要的SDK功能**

在功能配置界面，按下空格键可以选中或者失效某个功能，使用小键盘的上下键来在不同功能之间切换；如果想知道每个选项的具体含义，先用方向键将高亮光条移到那个选项上，再按键盘上的“h”按键，将出现帮助文本，说明选项是什么含义，以及打开了和关闭了意味着什么。

如果编译环境有自带标准头文件<stdint.h>，请使能选项。

**PLATFORM\_HAS\_STDINT**

由于模组支持TCP但是不支持MQTT，因此必须使能下面三项配置：

**FEATURE\_MQTT\_COMM\_ENABLED**，使用SDK提供的MQTT API与云端通信

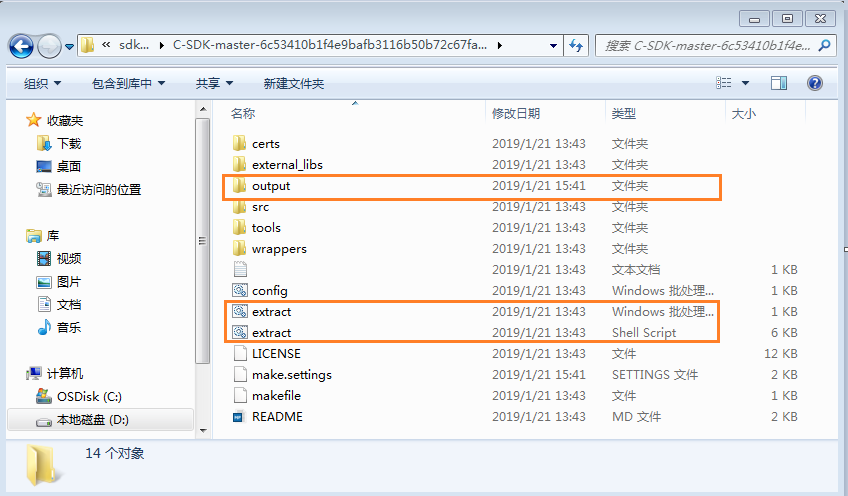
**FEATURE\_MQTT\_DEFAULT\_IMPL**，使用SDK中自带的MQTT Client实现，用户需要实现相关的TCP连接的创建、连接、数据收发过程

**FEATURE\_MQTT\_DIRECT**，设备端指定物联网平台端站点

其它功能均无需使能。

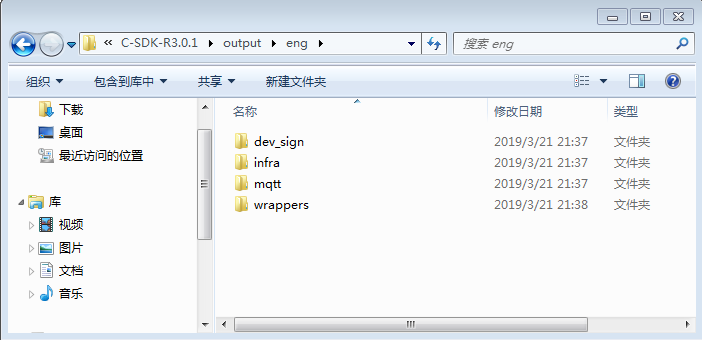
**抽取选中功能的源代码**

运行SDK根目录下的extract.sh（windows下运行extract.bat），客户选中的功能所对应的代码将会被放置到文件夹output。

[](https://cdn.nlark.com/yuque/0/2019/png/288961/1553178793024-1fb6ecbc-eb23-42a4-a786-169cbfcaa102.png#align=left&display=inline&height=286&originHeight=496&originWidth=848&status=done&width=489)

**将SDK代码文件加入客户编译环境**

客户将上一个步骤中得到的EVS-SDK的代码文件从output目录复制到自己的工程目录中，并修改自己的编译环境或者开发工具将这些代码文件集成到编译环境。

客户需要集成的文件包括eng目录下面的dev\_sign、infra、mqtt、wrappers目录下的代码文件。[](https://cdn.nlark.com/yuque/0/2019/png/288961/1553178808259-ecf52f6a-de17-449b-be36-55d724ac00db.png#align=left&display=inline&height=227&originHeight=340&originWidth=702&status=done&width=468)

**实现HAL对接函数**

EVS-SDK被设计为可以在不同的操作系统上运行，或者甚至在不支持操作系统的MCU上运行，因此与系统相关的操作被定义成一些HAL函数，需要客户进行实现；另外，由于不同的通信模组上的OS不同，所以与通信模组上TCP相关的操作也被定义成HAL函数需要客户进行实现。

所有HAL函数位于文件为output/eng/wrappers/wrapper.c中。

**系统相关HAL**

必须实现函数：



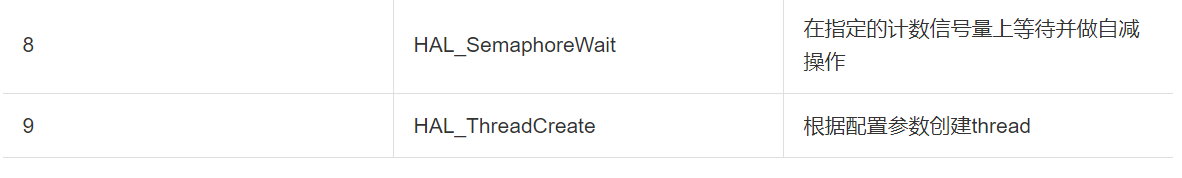
对以上函数若需了解更多细节, 可参考API接口文档

另外，在SDK的目录（非代码抽取目录）wrappers/os下有HAL的参考实现，用户可以查看是否有自己需要的OS参考实现，若未提供则用户需要自己进行实现

**可选实现函数**

如果模组没有运行OS，或者SDK的MQTT API并没有在多个线程中被调用，以下函数可以不用修改wrapper.c中相关的函数实现；在有OS场景下并且MQTT API被APP在多个线程中调用，则需要用户对接以下函数：





对以上函数接口若需了解更多细节，可以参考API接口文档

**TCP相关HAL**

MQTT基于TCP进行通信，模组商需要实现下面四个TCP HAL函数。下面是对这些函数的说明，也可以参考API接口文档了解更多说明。



**产品相关HAL**

下面的HAL用于获取产品的身份认证信息，设备厂商需要设计如何在设备上烧写设备身份信息，并通过下面的HAL函数将其读出后提供给SDK：



对以上函数若需了解更多细节，参考API接口文档

注：这几个参数在实际产品开发时应该由设备厂商通过AT指令告知模组，模组商调试时可以将自己创建的测试设备的ProductKey、DeviceName、DeviceSecret直接通过上面这几个函数返回。

如果模组不提供开发环境给用户进行二次开发，而是外接一个MCU并且产品的业务逻辑运行在MCU上，那么模组商还需要提供AT指令给MCU调用。

* 1. 高级系统集成SDK
     1. 目标系统为64位linux

**使用 make+gcc 编译SDK**

EVS-SDK对其HAL和TLS都已有提供的参考实现, 因此可以完整编译出所有的库和例子程序

选择平台配置

make reconfig

SELECT A CONFIGURATION:

1) config.alios.esp8266

2) config.alios.mk3080

3) config.ubuntu.x86

#? 3

SELECTED CONFIGURATION:

VENDOR : ubuntu

MODEL : x86

...

...

**编译**

make

**获取二进制库**

cd output/release/lib

ls

其中有三个主要产物, 它们都是64位架构的:

| 产物文件名 | 说明 |
| --- | --- |
| libiot\_hal.a | HAL接口层的参考实现, 提供了 HAL\_XXX() 接口 |
| libiot\_sdk.a | SDK的主库, 提供了 IOT\_XXX 接口和 linkkit\_xxx() 接口 |
| libiot\_tls.a | 裁剪过的 mbedtls, 提供了 mbedtls\_xxx() 接口, 支撑 libiot\_hal.a |

**获取可执行程序**

cd output/release/bin

ls

其中有两个主要产物, 它们都是64位架构的:

| 产物文件名 | 说明 |
| --- | --- |
| linkkit-example-solo | 高级版(旧版API)的例程, 可演示 linkkit\_xxx() 接口的使用 |
| mqtt-example | 基础版的例程, 可演示 IOT\_XXX() 接口的使用 |

* + 1. 目标系统为arm-linux

**安装交叉编译工具链**

sudo apt-get install -y gcc-arm-linux-gnueabihf

以如下命令和输出确认交叉编译工具链已安装好

arm-linux-gnueabihf-gcc --version

arm-linux-gnueabihf-gcc (Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609

Copyright (C) 2015 Free Software Foundation, Inc.

**创建平台配置文件**

vim tools/board/config.arm-linux.demo

CONFIG\_ENV\_CFLAGS = \

-D\_PLATFORM\_IS\_LINUX\_ \

-Wall \

-DNO\_EXECUTABLES \

CONFIG\_ENV\_LDFLAGS = \

-lpthread -lrt \

OVERRIDE\_CC = arm-linux-gnueabihf-gcc

OVERRIDE\_AR = arm-linux-gnueabihf-ar

OVERRIDE\_LD = arm-linux-gnueabihf-ld

CONFIG\_wrappers :=

或者写成

CONFIG\_ENV\_CFLAGS = \

-D\_PLATFORM\_IS\_LINUX\_ \

-Wall \

-DNO\_EXECUTABLES \

CONFIG\_ENV\_LDFLAGS = \

-lpthread -lrt \

CROSS\_PREFIX := arm-linux-gnueabihf-

CONFIG\_wrappers :=

**选择平台配置**

make reconfig

SELECT A CONFIGURATION:

1) config.alios.mk3080

2) config.arm-linux.demo

3) config.ubuntu.x86

#? 2

SELECTED CONFIGURATION:

VENDOR : arm-linux

MODEL : demo

**编译**

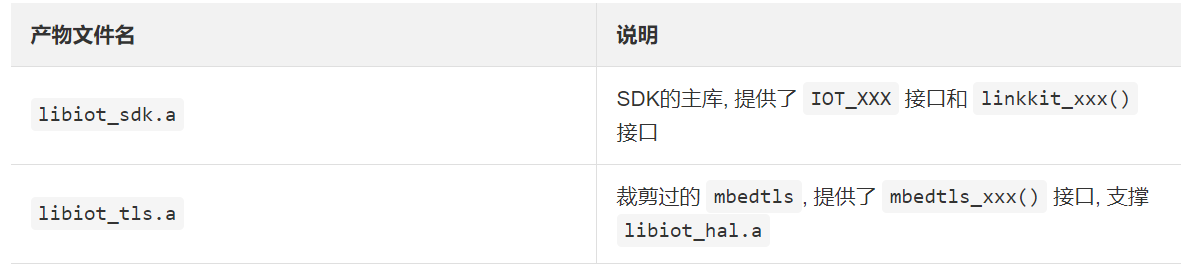
make

**获取二进制库**

cd output/release/lib

ls

其中有两个主要产物, **它们都是arm-linux架构的**:



1. 接口说明
   1. 功能列表

|  |  |  |  |
| --- | --- | --- | --- |
| **交互功能** | | **适用范围** | **备注** |
| 设备管理 | 固件升级 | 交流/直流 |  |
| 时钟同步 | 交流/直流 |  |
| 固件信息上报 | 交流/直流 |  |
| 设备配置与查询服务 | 交流/直流 |  |
| 设备数据查询服务 | 交流/直流 |  |
| 设备维护指令 | 交流/直流 |  |
| 充电枪电子锁控制 | 交流/直流 |  |
| 充电管理 | 计量计费模型管理 | 交流/直流 |  |
| 远程发起充电 | 交流/直流 |  |
| 充电设备发起充电 | 交流/直流 |  |
| 停止充电 | 交流/直流 |  |
| 交易记录管理 | 交流/直流 |  |
| 设备故障告警管理 | 交流/直流 |  |
| 预约充电 | 交流/直流 |  |
| 智能地锁控制 | 交流/直流 |  |
| 智能门锁控制 | 交流/直流 |  |
| 交流充电设备交互功能 | 交流充电设备属性上报 | 交流 |  |
| 有序充电策略 | 交流 |  |
| 交流充电枪状态变更事件 | 交流 |  |
| 充电前车辆信息上报 | 交流 |  |
| 直流充电设备交互功能 | 直流充电设备属性上报 | 直流 |  |
| 功率调节控制 | 直流 |  |
| 直流充电枪状态变更事件 | 直流 |  |

* 1. 功能接口说明
     1. 充电枪电子锁控制服务

SDK在收到物联平台下发的充电枪电子锁控制指令后，调用回调函数：int (\*cb)(evs\_service\_lockCtrl \*request, evs\_service\_feedback\_lockCtrl \*feedback);通过入参：request通知设备，设备通过出参：feedback将控制结果返回至SDK。

* + 1. 固件升级服务

SDK在收到物联平台下发的新固件消息通知时，调用回调函数：int (\*cb)(const char \*version);通知设备，设备在获取到新的固件信息后，通过判断当前设备状态决定是否进行升级，当需要升级时，通过调用函数：int evs\_linkkit\_fota(unsigned char \*buffer, int buffer\_length);进行固件获取。

* + 1. 时钟同步服务

设备在上电后，应主动通过调用函数：int evs\_linkkit\_time\_sync(void); 发出对时请求，SDK在收到平台下发的对时服务后，通过调用回调函数：int (\*cb)(const unsigned int);  获取物联平台下发的同步时间。

* + 1. 固件信息上报事件

设备在每次上电或固件信息发生变化时，通过调用函数：static int send\_event\_firware\_info(evs\_event\_fireware\_info \*data);将设备固件信息发送至物联平台。

* + 1. 设备配置信息更新及查询

设备在设备上电后，主动通过调用static int send\_event\_ask\_dev\_config(void);进行设备配置信息查询，当SDK收到物联平台下发的设备配置信息更新服务时通过回调函数：int (\*cb)(evs\_data\_dev\_config \*request, int \*feedback);通过入参：request将配置信息通知给设备，同时设备通过出参：feedback将配置更新结果返回至SDK。

当SDK收到物联平台下发的设备配置信息查询服务时通过回调函数：int (\*cb)(evs\_data\_dev\_config \*feedback);通知给设备，同时设备通过出参：feedback将配置信息返回至SDK。

* + 1. 设备日志查询服务

平台下发数据查询指令，并标明查询数据类型（交易记录、电表底值、日志、故障告警记录、BMS数据），SDK收到物联平台下发的信息后，通过调用回调函数：int (\*cb)(evs\_service\_query\_log \*request, evs\_service\_feedback\_query\_log \*feedback);通过参数：request，将信息透传给设备，设备收到指令后根据数据类型，通过出参：feedback，返回物联平台要查询的数据。

* + 1. 设备维护服务

设备在运行阶段，能够接受物联管理平台下发的维护（重启、检修、冻结、投运、停运、退运、恢复出厂设置）指令，SDK收到维护指令后，通过调用回调函数：int (\*cb)(evs\_service\_dev\_maintain \*request, evs\_service\_feedback\_dev\_maintain \*feedback);通过入参：request将维护指令透传给设备，设备根据维护指令进行设备的退运、停运、检修、投运等设备操作，并通过出参：feedback将维护结果返回至SDK。

* + 1. 计费模型管理

充电设备每次上线或计量计费模型失效时需要更新计量计费模型，设备通过函数：static int send\_event\_ask\_feeModel(evs\_event\_ask\_feeModel \*data);发送计量计费模型请求到平台。平台通过计量计费模型更新服务下发计量计费模型到充电设备。SDK在收到物联平台下发的计量计费模型时调用回调函数：int (\*cb)(evs\_service\_issue\_feeModel \*request, evs\_service\_feedback\_feeModel \*feedback); 通过入参：request将即将计费模型传递至设备，设备在更新计量计费模型后，通过出参：feedback将更新结果反馈至SDK，平台亦可以依据实际需要，主动将计量计费模型下发至充电设备。

* + 1. 启动充电服务

物联平台可以远程启动充电设备，SDK在充电设备在收到启动服务指令后，调用回调函数：int (\*cb)(evs\_service\_startCharge \*request, evs\_service\_feedback\_startCharge \*feedback); 通过入参：request将启动信息传递至设备，设备根据启动充电服务中的配置参数区分不同启动方式和充电模式，并将启动服务确认结果通过出参：feedback返回值SDK，设备在启动后将启动结果通过调用函数：static int send\_event\_start\_result(evs\_event\_startResult \*data); 发送至物联平台。

充电设备发起充电，充电设备通过调用函数：static int send\_event\_auth\_start(evs\_event\_startCharge \*data);向物联平台发送启动充电鉴权事件，物联平台在收到设备上送的鉴权指令后，将鉴权信息下发至设备，SDK在收到鉴权服务时，调用回调函数：int (\*cb)(evs\_service\_authCharge \*request, evs\_service\_feedback\_authCharge \*feedback);通过出参：request将鉴权结果传递至设备，同时设备通过出参：feedback将服务确认信息反馈至SDK。

* + 1. 停止充电功能

物联平台可以远程停止充电设备，SDK在充电设备在收到停止服务指令后，调用回调函数：int (\*cb)(evs\_service\_stopCharge \*request, evs\_service\_feedback\_stopCharge \*feedback);通过入参：request将停止信息传递至设备，设备根据停止充电服务中的配置参数进行设备的停止判断，并将停止服务确认结果通过出参：feedback返回值SDK，设备在停止后停止结果通过调用函数：static int send\_event\_stopCharge(evs\_event\_stopCharge \*data); 发送至物联平台。

* + 1. 交易记录管理

充电设备结束充电后，立刻在本地保存交易记录，并通过事件上送交易记录，平台接收到交易记录数据后进行校验，给充电设备回复“交易记录结果服务”，充电设备根据平台下发的校验情况确定是否需要再次上送本条交易记录。

接口函数int callback(evs\_service\_confirmTrade \*param, void \*result)

* + 1. 设备故障告警管理

故障、告警及单点遥信变位，通过事件立即上传至平台，故障和告警消失后同样需要立即上传平台。

接口函数static int send\_event\_alarm(evs\_event\_alarm \*data)

* + 1. 预约充电

平台可发起预约充电，由平台调用“预约充电”服务，充电设备收到预约充电请求指令后，返回服务调用结果，平台若10秒内没有收到充电设备的返回，则重发“预约充电服务”，最多重发2次。

接口函数Int callback(evs\_service\_rsvCharge \*parm,evs\_service\_feedback\_rsvCharge \*result)

* + 1. 智能地锁控制

SDK在充电结束后，将充电信息生成订单，并发送至物联管理平台，同时在设备无法接至物联管理平台时本地存储订单信息，并在连接恢复后再次上送订单至物联管理平台。

接口函数int callback(evs\_service\_groundLock\_ctrl \*param, evs\_service\_feedback\_groundLock\_ctrl \*result)

* + 1. 智能门锁控制

针对配置了智能门锁的充电设备，平台调用智能门锁控制服务远程控制门锁状态。当门锁设备发生状态变化时使用事件上报门锁状态。

接口函数int callback(evs\_service\_gateLock\_ctrl \*param, evs\_service\_feedback\_gateLock\_ctrl \*result)

* + 1. 交流充电设备属性上报

交流充电设备上报属性包括充电设备实时信息、充电中实时监测信息、非充电中实时监测信、输出电表监测信息，不同属性上送条件、频率可配置。

接口函数void evs\_send\_property(evs\_cmd\_property\_enum property\_type, void \*param)

* + 1. 有序充电策略

对于具有有序充电功能的充电设备，首次下发或策略变更时，平台调用“有序充电策略下发”，平台若10秒内没有收到充电设备调用返回，则重发“有序充电策略下发”，最多重发2次，函数int (\*cb)(evs\_service\_orderCharge \*request, evs\_service\_feedback\_orderCharge \*feedback)。

* + 1. 交流充电枪状态变更事件

充电设备运行过程中，交流流充电枪状态发生变更后需要发起直流充电枪状态变更事件，调用函数void evs\_send\_event(EVS\_CMD\_EVENT\_ACPILE\_CHANGE,& evs\_event\_pile\_stutus\_change)。

* + 1. 直流充电设备属性上报

直流充电设备上报充电中实时监测信息属性，调用函数void evs\_send\_property(EVS\_CMD\_PROPERTY\_DC\_WORK, & evs\_property\_dc\_work)；非充电中实时监测信息属性，调用函数void evs\_send\_property(EVS\_CMD\_PROPERTY\_DC\_NONWORK, & evs\_property\_dc\_nonWork)；输入电表监测信息、输出电表监测信息等属性，调用函数void evs\_send\_property(EVS\_CMD\_PROPERTY\_DCPILE, & evs\_property\_dcPile); 直流设备输出电表实时属性上报，调用函数void evs\_send\_property(EVS\_CMD\_PROPERTY\_DC\_OUTMETER, & evs\_property\_meter)；直流设备输入侧交流电表属性上报，调用函数void evs\_send\_property(EVS\_CMD\_PROPERTY\_DC\_INPUT\_METER,&evs\_property\_dc\_input\_meter)。

* + 1. 功率调节控制

平台调用该服务下发功率调节策略，设备应按该策略完成充电。调节策略下发后充电设备更新当前策略。回调函数int (\*cb)(evs\_service\_orderCharge \*request, evs\_service\_feedback\_orderCharge \*feedback)。

* + 1. 直流充电枪状态变更事件

充电设备运行过程中，直流充电枪状态发生变更后需要发起直流充电枪状态变更事件，调用函数void evs\_send\_event(EVS\_CMD\_EVET\_CAR\_INFO, &evs\_event\_car\_info)。

* + 1. 充电前车辆信息上报事件

有序充电设备，当充电枪插入车辆充电插座后，应在10s内获取并上报车辆信息，调用函数void evs\_send\_event(EVS\_CMD\_EVENT\_CAR\_INFO, & evs\_event\_car\_info)。

1. 版本变更说明

2020-07-06

1. 初次创建发布。

2020-07-09

1. 增加5.2.22充电前车辆信息上报事件；
2. 修正上一版本中的一些格式错误。

2020-08-07

1. 修正了目录和页码显示以及一些格式错误；
2. 修改了快速体验章节部分内容；
3. 完善平台操作部分内容，删减部分重复内容。