ALIBABA CLOUD

阿里云

IoT物联网操作系统
AliOS Things 文档

文档版本：20210427

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.硬件支持 08
1.1. SoC/MCU 08
1.2. Board 09
1.3. 传感器 11
2.API参考文档 16
2.1. 技术架构框图及目录结构 16
2.2. AliOS Things内核 18
2.2.1. 概览 18
2.2.2. 内存管理 19
2.2.3. 任务管理 22
2.2.4. 定时器 30
2.2.5. 信号量 37
2.2.6. 互斥信号量 44
2.2.7. 消息队列 47
2.2.8. 工作队列 50
2.2.9. 其他系统相关接口 53
2.3. 硬件抽象函数 56
2.3.1. GPIO 56
2.3.2. UART 63
2.3.3. SPI 70
2.3.4. I2C 74
2.3.5. FLASH 82
2.3.6. PWM 87
2.3.7. TIMER 92
2.3.8. WDG 97
2.3.9. RTC 99

2.3.10. ADC 103
2.3.11. DAC 107
2.3.12. RNG 111
2.4. 功能组件 113
2.4.1. 键值对存储模块（KV） 113
2.4.2. 命令行交互模块（CLI） 118
2.4.3. 虚拟文件系统（VFS） 123
2.4.4. OTA 158
2.4.5. 阿里云物联网平台连接 165
2.4.6. Socket组件文档 166
2.4.7. BLE host组件文档 184
2.4.8. BLE Mesh组件文档 204
2.4.9. Breeze-基于蓝牙低功耗协议栈的安全连云SDK 218
2.4.10. Netmgr组件文档 231
2.4.11. SAL组件文档 236
2.4.12. CoAP组件文档 258
2.4.13. AT组件文档 268
2.4.14. MAL组件文档 279
2.4.15. Network 297
2.4.15.1. Netmgr 297
2.4.15.2. Bluetooth 297
2.4.15.3. LoRaWAN 297
2.4.16. HTTP组件文档 297
2.4.17. MODBUS主协议 314
2.4.18. uData 327
2.4.19. uLog 334
3.开发工具和编译配置 341
3.1. 配置系统简介 341

3.2. 更多配置操作 342
3.3. 构建及编译命令详解 345
3.4. Keil/IAR 工程支持列表 347
3.5. 集成开发管理工具aos-cube 349
3.6. AliOS Studio图形化IDE插件 351
4.AliOS Things移植指南 366
4.1. 组件规范 366
4.1.1. 组件规范 366
4.2. BSP规范和移植指南 373
4.2.1. 板级支持目录规范 373
4.2.2. 芯片架构移植 384
4.2.3. 板级移植指导 387
4.2.4. k_config.h说明 404
4.2.5. 基于keil\iar的内核基础功能移植指导 406
5.远程调试运维指南 413
5.1. 概述 413
5.2. 功能开通 414
5.2.1. 开启设备的远程调试运维功能 414
5.2.2. 开通远程调试运维服务 420
5.3. 操作指南 424
5.3.1. 远程与设备做命令行交互 424
5.3.2. 远程查看分析设备日志 427
6.应用笔记 436
6.1. 维测模块及维测解析工具使用介绍 436
6.2. 使用线上的开发板做开发调试 444
6.3. 使用 Docker 开发 448
7.FAQ 454
7.1. 生成IAR/KEIL工程常见问题 454

7.2. aos-cube常见问题 ---------- 455
7.3. AliOS-Studio常见问题 ---------- 458

1.硬件支持

1.1. SoC/MCU

厂商	芯片系列	架构
ST	stm32l0xx	Cortex-M0+
ST	stm32l4xx	Cortex-M4
ST	stm32f4xx	Cortex-M4
ST	stm32f7xx	Cortex-M7
NXP	imx6.imx6sl	Cortex-A9
NXP	imx6.imx6dq	Cortex-A9
NXP	lpc540xx	Cortex-M4
NXP	lpc541xx	Cortex-M4
NXP	lpc546xx	Cortex-M4
NXP	KL2x	Cortex-M0+
NXP	KL8x	Cortex-M0+
NXP	KL4x	Cortex-M0+
NXP	i.MX RT1020	Cortex-M7
NXP	i.MX RT1050	Cortex-M7
Microchip	ATSAME54P20A	Cortex-M4
Espressif	esp32	Xtensa
Espressif	esp8266	Xtensa
MXCHIP	mk1101	Cortex-M3
MXCHIP	moc108	ARM968E-S
MXCHIP	MX1290	Cortex-M4
realtek	rtl8710bn	Cortex-M4
SiFive	freedom-e.e310	RISCV
SiliconLabs	efm32gxx	Cortex-M3
Rockchip	RK1108	Cortex-A7
C-SKY	CH2201	C-SKY

Cypress	cy8c6347	Cortex-M4
Nordic	nRF52 Series	Cortex-M4
GigaDevice	GD32F4xx	Cortex-M4
RDA	rda5981x	Cortex-M4
RDA	rda8955	MIPS-I
Allwinner	xr871	Cortex-M4
BEKEN	bk7231x	ARM968E-S
Renesas	RL78	Renesas
Renesas	RX65	Renesas
雄迈	xm510	Cortex-A5

1.2. Board

开发板	芯片	主频(MHz)	Flash(KB)	RAM(KB)	无线连接	
developerkit	STM32L496VGTx	80	1024	320	-	
starterkit	stm32l433	80	256	64	WiFi	
b_l475e	stm32l475	80	1024	128	WiFi	
stm32f429zi-nucleo	stm32f429zi	180	2048	256	WiFi	
stm32l432kc-nucleo	stm32l432kc	80	256	64	WiFi	
stm32l433rc-nucleo	stm32l433rc	80	256	64	WiFi	
stm32l476rg-nucleo	stm32l476rg	80	1024	128	WiFi	
stm32l496g-discovery	stm32l496g	80	1024	320	-	
stm32f769i-discovery	stm32f769i	180	2048	256	-	
eml3047	stm32l071kb	32	128	20	LoRa	
mk1101	mk1101				-	
mk3060	moc108	120	2048	256	WiFi	
mk3080	rtl8710bn	125	512	256	WiFi	

mk3165	stm32f411	100	256~512	128	-	
mk3166	stm32f412	100	512~1024	256	-	
mk3239	stm32f412	100	512~1024	256	-	
atsame54-xpro	ATSAME54P20A	120	1024	256	WiFi	
imx6sl	imx6sl	1000	4G eMMC	DDR3 512MB	-	
imx6dq	imx6dq	1000	16GB EMMC	DDR3 2GB	-	
lpcxpresso54102	lpc54102	100	512	104	WiFi	
lpcxpresso54608	lpc54608	220	512	200	WiFi	
frdmkl27z	mkl27z644	48	64	16	-	
evkmimxrt1020	mimxrt1021	500	64*1024	256*1024	-	
frdmkl81z	mkl81z7	72	128	96	-	
frdmkl26z	mkl81z7	48	256	32	-	
frdmkl27z	mkl27z644	48	64	16	-	
frdmkl28z	mkl28z7	96	512	128	-	
frdmkl43z	mkl43z4	48	256	32	-	
frdmkl82z	mkl82z7	72	128	96	-	
lpcxpresso54628	lpc54628	220	512	200	-	
lpcxpresso54018	lpc54018	100	512	104	-	
lpcxpresso54102	lpc54102	100	512	104	-	
lpcxpresso54114	lpc54114	100	8*1024		-	
lpcxpresso54608	lpc54608	180	128*1024	8*1024	-	
frdmkl28z	mkl28z7	96	512	128	-	
evkbimxrt1050	mimxrt1052	600	96	512	-	
frdmkl43z	mkl43z4	48	256	32	-	

lpcxpresso54114	lpc54114	100	256	192	-	
pca10040	nrf52832	64	512	64	BLE	
pca10056	nrf52840	64	1024	256	BLE	
amebaz_dev	rtl8710bn	125	512	256	WiFi	
cy8ckit-062	cy8c6347	150	1024	288	BLE	
cy8ckit-149	-		128	16	BLE	
uno-91h	rda5981x	160	32*1024	160	WiFi	
rda8955	rda8955		32*1024	32*1024	2G	
esp8266	esp8266	160	4028	50	WiFi	
esp32devkitc	esp32	240	448	520	WiFi	
gd32f4xx	GD32F450Z	200	3072	512	-	
bk7231devkitc	bk7231		256		WiFi	
bk7231udevkitc	bk7231u				WiFi+BLE	
cb2201	CH2201	48	256	80	WiFi	
rk1108	rk1108				-	
xr871evb	xr871	192	64	448	WiFi	
hifive1	freedom-e.e310	320	16*1024	16	-	
m100c	efm32gxx	32	128	16	-	
r5f565ne	RX65	120	2048	640	-	
m400	stm32l071kb	32	128	20	LoRa	
stm32l073rz-nucleo	STM32L073RZTx	32	192	20	Wifi	
r5f100lea	RL78	32	16~512	4~8	-	
xm510_evb	xm510				WiFi	
armhflinux	linux				WiFi	

1.3. 传感器

厂商	产品型号	类型	接口

ADI	ADXL345	加速度计	I2C
ADI	ADXL355	加速度计	I2C
ADI	ADXL372	加速度计	I2C
ADI	ADT7410	温度传感器	I2C
ADI	ADPD188GG	心跳传感器	I2C
AKM	AK9754	IR	I2C
AKM	AK09918	磁力计	I2C
AKM	AK09917	磁力计	I2C
AKM	AK09940	磁力计	I2C
AMS	TCS3400	RGB	I2C
AMS	CCS811	TVOC	I2C
AMS	ENS210	温湿度传感器	I2C
Bosch	BMM150	磁力计	I2C
Bosch	BMA421	加速度计	I2C
Bosch	BMA422	加速度计	I2C
Bosch	BMA455	加速度计	I2C
Bosch	BMA456	加速度计	I2C
Bosch	BMA280	加速度计	I2C
Bosch	BMA253	加速度计	I2C
Bosch	BMI160	加速度计、陀螺仪	I2C
Bosch	BMI120	加速度计、陀螺仪	I2C
Bosch	BMI260	加速度计、陀螺仪	I2C
Bosch	BMI055	加速度计、陀螺仪	I2C
Bosch	BMI088	加速度计、陀螺仪	I2C
Bosch	BMP380	气压计	I2C
Bosch	BMP280	气压计	I2C
Bosch	BMG160	陀螺仪	I2C
Bosch	BME280	温湿度传感器	I2C

Infineon	DSP310	气压计	I2C
liteon	LTR-303ALS	ALS	I2C
liteon	LTR-568ALS	ALS	I2C
liteon	LTR-507ALS	ALS/PS	I2C
liteon	LTR-559ALS	ALS/PS	I2C
liteon	LTR553	ALS/PS	I2C
liteon	LTR-659PS	PS	I2C
liteon	LTR-690PS	PS	I2C
liteon	LTR-706PS	PS	I2C
liteon	LTR-381RGB	RGB	I2C
liteon	LTR-390UV	UV	I2C
memsic	MMC3680KJ	磁力计	I2C
ROHM	BH1730	ALS	I2C
ROHM	BM1422	磁力计	I2C
ROHM	BM1383A	气压计	I2C
Sensirion	SCD30	CO2	I2C
Sensirion	SGPC3	TVOC	I2C
Sensirion	SHTC30	温湿度传感器	I2C
Sensirion	SHTC31	温湿度传感器	I2C
Sensirion	SHTC1	温湿度传感器	I2C
ST	vl53l0x	TOF	I2C
ST	LIS3MDL	磁力计	I2C
ST	LIS2MDL	磁力计	I2C
ST	AIS328DQ	加速度计	I2C
ST	H3LIS100DL	加速度计	I2C
ST	H3LIS331DL	加速度计	I2C
ST	LIS2DH12	加速度计	I2C
ST	LIS2HH12	加速度计	I2C
ST	LIS331HH	加速度计	I2C

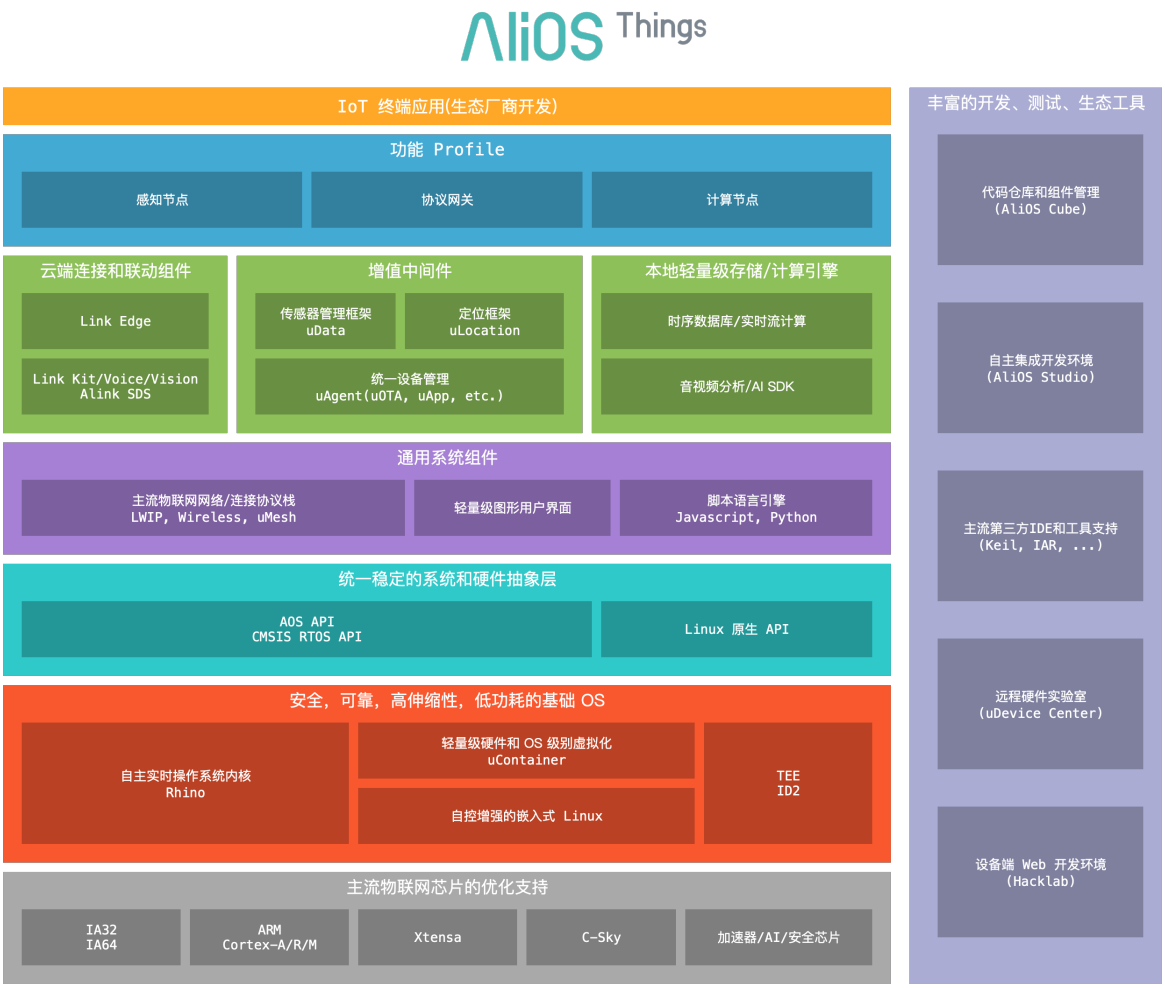
ST	LIS3DH	加速度计	I2C
ST	LIS2DW12	加速度计	I2C
ST	N2DM	加速度计	I2C
ST	LSM303AGR	加速度计、磁力计	I2C
ST	LSM6DS3TR	加速度计、陀螺仪	I2C
ST	LSM6DS3TR-C	加速度计、陀螺仪	I2C
ST	LSM6DSL	加速度计、陀螺仪	I2C
ST	LSM6DSM	加速度计、陀螺仪	I2C
ST	LSM6DSOQ	加速度计、陀螺仪	I2C
ST	LSM6DSR	加速度计、陀螺仪	I2C
ST	LPS22HB	气压计	I2C
ST	LPS33HB	气压计	I2C
ST	LPS35HB	气压计	I2C
ST	I3G4250D	陀螺仪	I2C
ST	L3GD20H	陀螺仪	I2C
ST	A3G4250D	陀螺仪	I2C
ST	HTS221	温湿度传感器	I2C
明皜科技	da213B	加速度计	I2C
明皜科技	da215	加速度计	I2C
明皜科技	da380B	加速度计	I2C
明皜科技	da217	加速度计	I2C
明皜科技	da270	加速度计	I2C
明皜科技	da312B	加速度计	I2C
深迪	ST480mc	磁力计	I2C
深迪	ST350mc	磁力计	I2C
深迪	SH200Q	加速度计、陀螺仪	I2C
深迪	SH200L	加速度计、陀螺仪	I2C
河北美泰	MSCO2MB	二氧化碳气体传感器	MODBUS
河北美泰	MSF1000MB	甲醛传感器	MODBUS

河北美泰	MSN1000MB	噪声传感器	MODBUS
河北美泰	MSP1000MB	大气压力传感器	MODBUS
河北美泰	MSPM25MB	空气质量传感器	MODBUS
河北美泰	MSTH1000	温湿度传感器	MODBUS
河北美泰	MSLT100MB	光照传感器	MODBUS
河北美泰	MLWS10	叶面湿度传感器	MODBUS
河北美泰	MSSM10	土壤水分传感器	MODBUS
河北美泰	MSEC10	土壤导电率传感器	MODBUS
河北美泰	MSPH10	土壤酸碱度传感器	MODBUS
昆仑	JHYL-W1	雨量计	MODBUS
昆仑	JHFX-W1	风向传感器	MODBUS
昆仑	JHFS-W1	风速传感器	MODBUS
昆仑	JWSK-VW1	温湿度变送器	MODBUS
昆仑	ZD-6W1H	光照传感器	MODBUS
昆仑	JQAW-6W1-SN	二氧化碳传感器	MODBUS
昆仑	JQYB-W1	大气压力传感器	MODBUS
昆仑	JTWS-AC	土壤温湿度传感器	MODBUS

2.API参考文档

2.1. 技术架构框图及目录结构

AliOS Things的架构可以适用于分层架构和组件化架构。



目录结构

AliOS Things core SDK包含以下目录：

```
AliOS-Things
├── application
│   ├── example       # 示例代码
│   └── profile       # 典型场景的应用方案
├── build             # 编译构建相关工具和脚本
├── components        # 功能组件
│   ├── dm            # 设备管理组件
│   │   ├── bootloader
│   │   ├── ota
│   │   ├── ulog
│   │   └── und
│   ├── linkkit       # 阿里云IoT连接套件
│   ├── network       # IP网络协议栈组件
│   │   ├── http
│   │   ├── lwip
│   │   └── netmgr
│   ├── security      # 安全类组件
│   │   └── mbedtls
│   └── utility       # 工具类组件
│       ├── cjson
│       └── yloop
├── core              # 内核及相关组件
├── document          # 说明文档
├── include           # 组件对外的头文件
├── platform          # 芯片平台支持和BSP
│   ├── arch          # 架构移植
│   ├── board         # 板级支持
│   └── mcu           # MCU, SoC 移植支持
└── projects          # 为不同开发环境提供的工程相关文件
```

增值类组件，可以通过uCube命令下载、安装、升级和卸载。 增值类组件，一般都安装到components目录。

```
components
├── bus             # 本地通讯协议
│   ├── canopen
│   ├── knx
│   ├── mbmaster
│   └── usb
├── dm          # 设备管理
│   └── uagent
├── fs          # 文件系统
│   ├── cramfs
│   ├── fatfs
│   ├── jffs2
│   ├── ramfs
│   ├── spiffs
│   ├── uffs
│   └── yaffs2
├── gui         # 人机交互界面
│   ├── freetype-2.5.3
│   └── littlevGL
├── language            # 脚本引擎
│   ├── jsengine
│   └── micropython
├── network             # IP网络协议栈
│   ├── coap
│   ├── httpdns
│   ├── libsrtp
│   ├── lwm2m
│   ├── mal
│   ├── rtp
│   ├── sal
│   ├── umesh2
│   └── websocket
├── peripherals         # 外设驱动
│   ├── iot_comm_module
│   │   ├── mal
│   │   └── sal
│   └── sensor
├── security        # 安全
│   └── linksecurity
├── service             # 应用组件
│   ├── uai
│   ├── udata
│   └── ulocation
├── utility  # 工具类
│   ├── at
│   ├── debug_tools
│   └── zlib
└── wireless        # 无线类
    ├── bluetooth
    └── lorawan
```

2.2. AliOS Things内核

2.2.1. 概览

本章节将介绍AliOS Things内核，目的是让用户了解内核的运转机制，学会如何使用内核接口。 了解内核的运转机制将帮助用户掌握如何使用内核接口，以及内核接口的使用场景和运行上下文，也可以帮助用户更快的定位编程中遇到的问题，协助用户使用内核来合理搭建高阶的软件架构，解决复杂的运用逻辑需求。

2.2.2. 内存管理

内存管理是指软件运行时系统内存资源的分配和使用的技术。其最主要的目的是如何高效，快速的分配，并且在适当的时候释放和回收内存资源。

API列表

aos_malloc()	从系统heap分配内存给用户
aos_zalloc()	从系统heap分配内存给用户，并且将分配的内存初始化为0
aos_calloc()	从系统heap分配内存给用户，并且将分配的内存初始化为0
aos_realloc()	重新调整之前调用 aos_malloc(aos_calloc、aos_zalloc) 所分配的内存块的大小
aos_free()	释放分配的内存

使用

添加该组件

内存管理是AliOS Things 默认添加的组件，开发者无需再手动添加。

包含头文件

```
#include <aos/kernel.h>
```

使用示例

```
char *buf = (char*)aos_malloc(100);
......
buf = aos_realloc(buf, 200);
......
aos_free(buf);
......
```

API 详情

内存管理的应用层API说明请参考include/aos/kernel.h。

aos_malloc()

从系统heap分配内存给用户。

函数原型

```
void *aos_malloc(unsigned int size);
```

输入参数

size	要分配内存块的字节数	100

返回参数

如果分配成功，返回所分配的内存区域起始地址指针；如果失败，返回NULL。

调用示例

```
char *ptr = NULL;
unsigned int size = 64;
ptr = (char*)aos_malloc(size);
if (NULL == ptr) {
  printf("aos_malloc failed\r\n");
  ...
}
```

aos_zalloc()

从系统heap分配内存给用户，并且将分配的内存初始化为0。

函数原型

```
void *aos_zalloc(unsigned int size);
```

输入参数

size	要分配内存块的字节数	64

返回参数

如果分配成功，返回所分配的内存区域起始地址指针；如果失败，返回NULL。

调用示例

```
char *ptr = NULL;
unsigned int size = 64;
ptr = (char*)aos_zalloc(size);
if (NULL == ptr) {
  printf("aos_zalloc failed\r\n");
}
```

aos_calloc()

从系统heap分配内存给用户，并且将分配的内存初始化为0。

函数原型

```
void *aos_calloc(unsigned int nitems, unsigned int size);
```

输入参数

nitems	要分配内存块个数	10

size	每个内存块的字节数	64

返回参数

如果分配成功，返回所分配的内存区域起始地址指针；如果失败，返回NULL。

调用示例

```
char *ptr = NULL;
unsigned int size = 64;
unsigned int n = 10;
ptr = (char*)aos_calloc(n, size);
if (NULL == ptr) {
  printf("aos_calloc failed\r\n");
}
```

aos_realloc()

重新调整之前调用 aos_malloc/aos_calloc/aos_zalloc 所分配的内存块的大小。当分配成功时，从原内存块拷贝数据到新分配的内存块，同时释放原内存块，返回新分配的内存块地址。若分配失败，则不释放原来的内存区域，并返回NULL。对于拷贝数据的大小，如果新分配的内存块比原内存块大，则只拷贝原内存块长度的数据到新分配的内存块；否则，只拷贝新分配内存块长度的数据。

函数原型

```
void *aos_realloc(void *mem, unsigned int size);
```

输入参数

mem	原内存块起始地址。若该指针为NULL，则直接分配内存	
size	本次操作期望新分配的内存区域大小。若该值为0，则释放mem指向的内存，不分配新内存块	64

返回参数

如果分配成功，返回新分配的内存区域起始地址指针；如果失败，返回NULL。

调用示例

```
char *ptr = NULL;
char *newptr = NULL;
unsigned int size = 64;
ptr = (char*)aos_malloc(size);
if (NULL == ptr) {
  printf("aos_malloc failed\r\n");
  ...
}
...
size = 100;
newptr = (char*)aos_realloc(ptr, size);
if (NULL == ptr) {
  printf("aos_realloc failed\r\n");
}
```

aos_free()

释放分配的内存。

函数原型

```
void aos_free(void *mem);
```

输入参数

mem	要释放的内存区域起始地址指针	

返回参数

无

调用示例

```
char *ptr = NULL;
unsigned int size = 64;
ptr = (char*)aos_malloc(size);
if (NULL == ptr) {
  printf("aos_malloc failed\r\n");
  ...
}
...
aos_free(ptr);
```

2.2.3. 任务管理

现代操作系统都建立在任务的基础上，任务是AliOS Things的一个基本执行环境，有的操作系统也称之为线程（thread）。多任务的运行环境提供了一个基本机制，从宏观上可以看作单个CPU执行单元上同时执行多个任务，从微观上看，CPU快速地切换任务来并发运行多个任务。AliOS Things实时操作系统支持多任务机制。

每个任务都具有上下文（context），上下文context是指当任务被调度执行的时候此任务能看见的CPU资源和系统资源，当发生任务切换的时候，任务的上下文被保存在任务控制块（ktask_t）中，这些上下文包括当前任务的CPU指令地址（PC指针），当前任务的栈空间，当前任务的CPU寄存器状态等。

任务管理功能的相关源码位于：/core/osal/aos/rhino.c 文件中(AliOS Things v3.1)；osal/aos/rhino.c（AliOS Things v3.0及以前）。

API列表

aos_task_new()	动态创建一个任务，任务句柄不返回，创建完后自动运行
aos_task_new_ext()	动态创建一个任务，传入任务句柄，并指定优先级，创建完后自动运行
aos_task_exit()	任务自动退出
aos_task_delete()	任务删除
aos_task_name()	返回当前任务名
aos_task_key_create()	申请一个任务私有数据区域
aos_task_key_delete()	释放一个任务私有数据区域
aos_task_setspecific()	设置指定任务私有数据区域的内容
aos_task_getspecific()	获取指定任务私有数据区域的内容

使用

添加该组件

任务管理是AliOS Things 默认添加的组件，开发者无需再手动添加。

包含头文件

```
#include <aos/kernel.h>
```

使用示例

示例说明：创建一个任务，在任务内创建任务私有数据区域，设置私有区域内容、获取私有区域内容，删除私有数据区域，最后任务退出。

```
/* 创建任务test_task，任务栈大小为1024字节 */
aos_task_new("test_task", test_task, NULL, 1024);
static void test_task(void *paras)
{
  int ret = -1;
  int var = 0;
  aos_task_key_t task_key;
  void *task_value = NULL;
  /* 创建任务私有数据区域*/
  ret = aos_task_key_create(&task_key);
  /* 打印任务名和任务私有数据区域索引值*/
  printf("%s task key %d: \r\n", aos_task_name(), task_key);
  var = 0x5a5a;
  /* 设置当前任务私有数据区域的某索引空闲块内容 */
  ret = aos_task_setspecific(task_key, &var);
  /* 获取当前任务私有数据区域的某索引数据块内容 */
  task_value = aos_task_getspecific(task_key);
  printf("%s task key 0x%x: \r\n", aos_task_name(), *(int*)task_value);
  /* 删除任务私有数据区域的空闲块索引 */
  aos_task_key_delete(task_key);
  /* 任务退出 */
  aos_task_exit(0);
}
```

API 详情

任务管理的应用层API说明请参考include/aos/kernel.h

aos_task_new()

动态创建一个任务，任务句柄不返回，创建完后自动运行。采用默认优先级AOS_DEFAULT_APP_PRI（32）。受宏RHINO_CONFIG_KOBJ_DYN_ALLOC开关控制。

函数原型

```
int aos_task_new(const char *name, void (*fn)(void *), void *arg, int stack_size);
```

输入参数

name	任务名	'"test_task"'
fn	任务处理函数	
arg	任务处理函数的参数	
stack_size	任务栈大小（单位：字节）	

返回参数

0表示成功，其他值表示失败。具体的返回值见本文档返回参数定义小节。

调用示例

```
static void test_task(void *paras)
{
  ......
}
aos_task_new("test_task", test_task, NULL, 1024);
```

aos_task_new_ext()

动态创建一个任务，传入任务句柄，并指定优先级，创建完后自动运行。

函数原型

```
int aos_task_new_ext(aos_task_t *task, const char *name, void (*fn)(void *), void *arg, int stack_size, int prio);
```

输入参数

task	任务句柄	
name	任务名	'"test_task"'
fn	任务处理函数	
arg	任务处理函数的参数	
stack_size	任务栈大小（单位：字节）	
prio	任务运行优先级（范围：0~RHINO_CONFIG_PRI_MAX-2;RHINO_CONFIG_PRI_MAX-1为idle任务优先级）	10

返回参数

0表示成功，其他值表示失败。具体的返回值见本文档返回参数定义小节。

调用示例

```
static void test_task(void *paras)
{
  ......
}
aos_task_t task_handle;
aos_task_new_ext(&task_handle, "test_task", test_task, NULL, 1024, 10);
```

aos_task_exit()

任务自动退出。

函数原型

```
void aos_task_exit(int code);
```

输入参数

code	任务退出码，暂时无用	

返回参数

无

调用示例

```
/* 任务处理函数 */
void task_entry(void *arg)
{
  ......
  aos_task_exit(0); /* 任务自动退出 */
}
```

aos_task_delete()

删除任务。

函数原型

```
int aos_task_delete(char *name);
```

输入参数

name	待删除的任务名	'"test_task"'

返回参数

0表示成功，其他值表示失败。具体的返回值见本文档返回参数定义小节。

调用示例

```
static void test_task(void *paras)
{
  ......
}
aos_task_new("test_task", test_task, NULL, 1024);
......
aos_task_delete("test_task");
```

aos_task_name()

获得当前任务名字。

函数原型

```
const char *aos_task_name(void);
```

输入参数

无

返回参数

任务名字符串。

调用示例

```
char *task_name;
task_name = aos_task_name();
printf("task name: %s\r\n", task_name);
```

aos_task_key_create()

申请一个任务私有数据区域。

函数原型

```
int aos_task_key_create(aos_task_key_t *key);
```

输入参数

key	任务私有数据区域的索引，新分配的索引保存在该参数中	

返回参数

0表示成功，失败返回-EINVAL。

调用示例

```
int ret = -1;
aos_task_key_t task_key;
ret = aos_task_key_create(&task_key);
```

aos_task_key_delete()

释放一个任务私有数据区域。

函数原型

```
void aos_task_key_delete(aos_task_key_t key);
```

输入参数

key	任务私有数据区域的索引，删除该索引指向的任务私有数据区域	

返回参数

无

调用示例

```
int ret = -1;
aos_task_key_t task_key;
ret = aos_task_key_create(&task_key);
......
aos_task_key_delete(task_key);
```

aos_task_setspecific()

设置指定任务私有数据区域的内容。

函数原型

```
int aos_task_setspecific(aos_task_key_t key, void *vp);
```

输入参数

key	任务私有数据区域的索引，设置该索引指向的任务私有数据区域	
vp	设置到任务私有数据区域的值	

返回参数

0表示成功，其他值表示失败。具体的返回值见本文档返回参数定义小节。

调用示例

```
int ret = -1;
int var = 0;
aos_task_key_t task_key;
ret = aos_task_key_create(&task_key);
var = 0x5a5a;
ret = aos_task_setspecific(task_key, &var);
```

aos_task_getspecific()

获取指定任务私有数据区域的内容。

函数原型

```
void *aos_task_getspecific(aos_task_key_t key);
```

输入参数

key	任务私有数据区域的索引，获取该索引指向的任务私有数据区域的内容	

返回参数

指定任务私有数据区域的内容。

调用示例

```
int ret = -1;
int var = 0;
aos_task_key_t task_key;
void *task_value = NULL;
ret = aos_task_key_create(&task_key);
var = 0x5a5a;
ret = aos_task_setspecific(task_key, &var);
……
task_value = aos_task_getspecific(task_key);
```

其他

返回参数定义

返回值定义在core/rhino/include/k_err.h文件中。该文件为内部文件，出错时可根据返回值查阅该文件确认出错原因。

```
typedef enum
{
  RHINO_SUCCESS = 0u,
  RHINO_SYS_FATAL_ERR,
  RHINO_SYS_SP_ERR,
  RHINO_RUNNING,
  RHINO_STOPPED,
  RHINO_INV_PARAM,
  RHINO_NULL_PTR,
  RHINO_INV_ALIGN,
  RHINO_KOBJ_TYPE_ERR,
  RHINO_KOBJ_DEL_ERR,
  RHINO_KOBJ_DOCKER_EXIST,
  RHINO_KOBJ_BLK,
  RHINO_KOBJ_SET_FULL,
  RHINO_NOTIFY_FUNC_EXIST,
  RHINO_MM_POOL_SIZE_ERR = 100u,
  RHINO_MM_ALLOC_SIZE_ERR,
  RHINO_MM_FREE_ADDR_ERR,
  RHINO_MM_CORRUPT_ERR,
  RHINO_DYN_MEM_PROC_ERR,
  RHINO_NO_MEM,
  RHINO_RINGBUF_FULL,
  RHINO_RINGBUF_EMPTY,
  RHINO_SCHED_DISABLE = 200u,
  RHINO_SCHED_ALREADY_ENABLED,
  RHINO_SCHED_LOCK_COUNT_OVF,
  RHINO_INV_SCHED_WAY,
  RHINO_TASK_INV_STACK_SIZE = 300u,
  RHINO_TASK_NOT_SUSPENDED,
  RHINO_TASK_DEL_NOT_ALLOWED,
  RHINO_TASK_SUSPEND_NOT_ALLOWED,
  RHINO_TASK_CANCELED,
  RHINO_SUSPENDED_COUNT_OVF,
  RHINO_BEYOND_MAX_PRI,
  RHINO_PRI_CHG_NOT_ALLOWED,
  RHINO_INV_TASK_STATE,
  RHINO_IDLE_TASK_EXIST,
  RHINO_NO_PEND_WAIT = 400u,
```

```
    RHINO_BLK_ABORT,
    RHINO_BLK_TIMEOUT,
    RHINO_BLK_DEL,
    RHINO_BLK_INV_STATE,
    RHINO_BLK_POOL_SIZE_ERR,
    RHINO_TIMER_STATE_INV = 500u,
    RHINO_NO_THIS_EVENT_OPT = 600u,
    RHINO_BUF_QUEUE_INV_SIZE = 700u,
    RHINO_BUF_QUEUE_SIZE_ZERO,
    RHINO_BUF_QUEUE_FULL,
    RHINO_BUF_QUEUE_MSG_SIZE_OVERFLOW,
    RHINO_QUEUE_FULL,
    RHINO_QUEUE_NOT_FULL,
    RHINO_SEM_OVF = 800u,
    RHINO_SEM_TASK_WAITING,
    RHINO_MUTEX_NOT_RELEASED_BY_OWNER = 900u,
    RHINO_MUTEX_OWNER_NESTED,
    RHINO_MUTEX_NESTED_OVF,
    RHINO_NOT_CALLED_BY_INTRPT = 1000u,
    RHINO_TRY_AGAIN,
    RHINO_WORKQUEUE_EXIST = 1100u,
    RHINO_WORKQUEUE_NOT_EXIST,
    RHINO_WORKQUEUE_WORK_EXIST,
    RHINO_WORKQUEUE_BUSY,
    RHINO_WORKQUEUE_WORK_RUNNING,
    RHINO_TASK_STACK_OVF = 1200u,
    RHINO_INTRPT_STACK_OVF,
    RHINO_STATE_ALIGN = INT_MAX /* keep enum 4 bytes at 32bit machine */
} kstat_t;
```

2.2.4. 定时器

tick一般是作为任务延迟调度的内部机制，其接口主要是系统内部使用。但是对于使用OS的应用软件有时也需要定时触发相关功能的接口，包括单次定时器和周期定时器。从用户层面来讲，不需要关注底层cpu的定时机制以及tick的调度，用户希望的定时器接口是可以创建和使能一个软件接口定时器，时间到了之后，用户的hook函数能被执行。而对于操作系统的定时器本身来讲，其也需要屏蔽底层定时模块的差异。因此，在软件层次上，对于定时器硬件相关的操作由tick模块完成，定时器（timer）模块基于tick作为最基本的时间调度单元，即最小时间周期，来推动自己时间轴的运行。 AliOS Things提供基本的软件定时器功能，包括定时器的创建、删除、启动，以及单次和周期定时器。

API 列表

aos_timer_new()	创建软件定时器
aos_timer_new_ext()	创建软件定时器
aos_timer_start()	启动软件定时器
aos_timer_stop()	停止软件定时器
aos_timer_change()	修改软件定时器的定时参数
aos_timer_free()	删除软件定时器

使用

添加该组件

软件定时器是AliOS Things 默认添加的组件，开发者无需再手动添加。

包含头文件

```
#include <aos/kernel.h>
```

使用示例

创建自动运行的周期执行定时器

```
aos_timer_t  g_timer;
int ret = -1;
...
static void timer_handler(void *arg1, void* arg2)
{
   printf("timer handler\r\n");
   ...
}
/*创建定时周期为200ms的周期执行的定时器，并自动运行*/
ret = aos_timer_new(&g_timer, timer_handler, NULL, 200, 1);
if (ret != 0) {
   printf("timer create failed\r\n");
     ...
  }
....
/*停止定时器*/
aos_timer_stop(&g_timer);
....
/*启动定时器*/
aos_timer_start(&g_timer);
....
/*停止定时器*/
aos_timer_stop(&g_timer);
/*释放定时器*/
aos_timer_free(&g_timer);
....
```

创建不自动运行的周期执行定时器，并在使用中改变定时周期。

```
aos_timer_t  g_timer;
int ret = -1;
...
static void timer_handler(void *arg1, void* arg2)
{
  printf("timer handler\r\n");
  ...
}
/*创建定时周期为200ms的周期执行的定时器，不自动运行*/
ret = aos_timer_new_ext(&g_timer, timer_handler, NULL, 200, 0, 0);
if (ret != 0) {
  printf("timer create failed\r\n");
    ...
}
....
/*需要手动启动定时器*/
aos_timer_start(&g_timer);
....
/*停止定时器*/
aos_timer_stop(&g_timer);
....
/*改变定时周期为1000ms, 注意：需要在定时器未启动状态是才能修改*/
aos_timer_change(&g_timer, 1000);
/*启动定时器*/
aos_timer_start(&g_timer);
....
/*停止定时器*/
aos_timer_stop(&g_timer);
/*释放定时器*/
aos_timer_free(&g_timer);
....
```

API详情

定时器的应用层API说明请参考include/aos/kernel.h

aos_timer_new()

创建一个软件定时器，创建后自动运行。

函数原型

```
int aos_timer_new(aos_timer_t *timer,void (*fn)(void *, void *),void *arg, int ms, int repeat);
```

输入参数

timer	软件定时器句柄	
fn	定时到期处理函数	
arg	定时到期处理函数的参数	
ms	定时器超时时间（单位ms），即间隔多少时间执行fn	1000

repeat	周期定时或单次定时（1：周期，0：单次）	1

返回参数

0表示成功，其他值表示失败。具体的返回值见本文档返回参数定义小节。

调用示例

```
static void timer_handler(void *arg1, void* arg2)
{
  /* 定时到期处理函数内不应调用能引起阻塞的函数 */
  ......
}
int ret = -1;
ret = aos_timer_new(&g_timer, timer_handler, NULL, 200, 1);
```

aos_timer_new_ext()

创建一个软件定时器，可通过入参决定创建后是否自动运行。

函数原型

```
int aos_timer_new_ext(aos_timer_t *timer, void (*fn)(void *, void *),void *arg, int ms, int repeat, unsigned char auto_run);
```

输入参数

timer	软件定时器句柄	
fn	定时到期处理函数	
arg	定时到期处理函数的参数	
ms	定时器超时时间（单位ms），即间隔多少时间执行fn	1000
repeat	周期定时或单次定时（1：周期，0：单次）	1
auto_run	1表示自动运行，0表示不自动运行，需要手动调用aos_timer_start才能启动	0

返回参数

0表示成功，其他值表示失败。具体的返回值见本文档返回参数定义小节。

调用示例

```
static void timer_handler(void *arg1, void* arg2)
{
  /* 定时到期处理函数内不应调用能引起阻塞的函数 */
  ......
}
int ret = -1;
ret = aos_timer_new_ext(&g_timer, timer_handler, NULL, 200, 1, 0);
```

aos_timer_stop()

停止软件定时器。

函数原型

```
int aos_timer_stop(aos_timer_t *timer);
```

输入参数

timer	软件定时器句柄	

返回参数

0表示成功，其他值表示失败。具体的返回值见本文档返回参数定义小节。

调用示例

```
int ret = -1;
ret = aos_timer_new(&g_timer, timer_handler, NULL, 200, 1);
......
/*停止定时器*/
aos_timer_stop(&g_timer);
```

aos_timer_free()

删除软件定时器。删除定时器前需调用aos_timer_stop停止定时器。

函数原型

```
void aos_timer_free(aos_timer_t *timer);
```

输入参数

timer	软件定时器句柄	

返回参数

无

调用示例

```
int ret = -1;
ret = aos_timer_new(&g_timer, timer_handler, NULL, 200, 1);
……
/*停止定时器*/
aos_timer_stop(&g_timer);
aos_timer_free(&g_timer);
```

aos_timer_start()

启动软件定时器。

函数原型

```
int aos_timer_start(aos_timer_t *timer);
```

输入参数

timer	软件定时器句柄	

返回参数

0表示成功，其他值表示失败。具体的返回值见本文档返回参数定义小节。

调用示例

```
int ret = -1;
ret = aos_timer_new_ext(&g_timer, timer_handler, NULL, 200, 1, 0);
aos_timer_start(&g_timer);
```

aos_timer_change()

修改软件定时器的定时参数。调用该函数前需先调用aos_timer_stop停止定时器。

函数原型

```
int aos_timer_change(aos_timer_t *timer, int ms);
```

输入参数

timer	软件定时器句柄	
ms	新的定时器超时时间（单位ms），即间隔多少时间执行定时器到期处理函数	1000

返回参数

0表示成功，其他值表示失败。具体的返回值见本文档返回参数定义小节。

调用示例

```
int ret = -1;
ret = aos_timer_new(&g_timer, timer_handler, NULL, 200, 1);
……
/*停止定时器*/
aos_timer_stop(&g_timer);
aos_timer_change(&g_timer, 1000);
aos_timer_start(&g_timer);
```

其他

返回参数定义

返回值定义在core/rhino/include/k_err.h文件中。该文件为内部文件，出错时可根据返回值查阅该文件确认出错原因。

```
typedef enum
{
  RHINO_SUCCESS = 0u,
  RHINO_SYS_FATAL_ERR,
  RHINO_SYS_SP_ERR,
  RHINO_RUNNING,
  RHINO_STOPPED,
  RHINO_INV_PARAM,
  RHINO_NULL_PTR,
  RHINO_INV_ALIGN,
  RHINO_KOBJ_TYPE_ERR,
  RHINO_KOBJ_DEL_ERR,
  RHINO_KOBJ_DOCKER_EXIST,
  RHINO_KOBJ_BLK,
  RHINO_KOBJ_SET_FULL,
  RHINO_NOTIFY_FUNC_EXIST,
  RHINO_MM_POOL_SIZE_ERR = 100u,
  RHINO_MM_ALLOC_SIZE_ERR,
  RHINO_MM_FREE_ADDR_ERR,
  RHINO_MM_CORRUPT_ERR,
  RHINO_DYN_MEM_PROC_ERR,
  RHINO_NO_MEM,
  RHINO_RINGBUF_FULL,
  RHINO_RINGBUF_EMPTY,
  RHINO_SCHED_DISABLE = 200u,
  RHINO_SCHED_ALREADY_ENABLED,
  RHINO_SCHED_LOCK_COUNT_OVF,
  RHINO_INV_SCHED_WAY,
  RHINO_TASK_INV_STACK_SIZE = 300u,
  RHINO_TASK_NOT_SUSPENDED,
  RHINO_TASK_DEL_NOT_ALLOWED,
  RHINO_TASK_SUSPEND_NOT_ALLOWED,
  RHINO_TASK_CANCELED,
  RHINO_SUSPENDED_COUNT_OVF,
  RHINO_BEYOND_MAX_PRI,
  RHINO_PRI_CHG_NOT_ALLOWED,
  RHINO_INV_TASK_STATE,
  RHINO_IDLE_TASK_EXIST,
  RHINO_NO_PEND_WAIT = 400u,
  RHINO_BLK_ABORT,
  RHINO_BLK_TIMEOUT,
```

```
    RHINO_BLK_DEL,
    RHINO_BLK_INV_STATE,
    RHINO_BLK_POOL_SIZE_ERR,
    RHINO_TIMER_STATE_INV = 500u,
    RHINO_NO_THIS_EVENT_OPT = 600u,
    RHINO_BUF_QUEUE_INV_SIZE = 700u,
    RHINO_BUF_QUEUE_SIZE_ZERO,
    RHINO_BUF_QUEUE_FULL,
    RHINO_BUF_QUEUE_MSG_SIZE_OVERFLOW,
    RHINO_QUEUE_FULL,
    RHINO_QUEUE_NOT_FULL,
    RHINO_SEM_OVF = 800u,
    RHINO_SEM_TASK_WAITING,
    RHINO_MUTEX_NOT_RELEASED_BY_OWNER = 900u,
    RHINO_MUTEX_OWNER_NESTED,
    RHINO_MUTEX_NESTED_OVF,
    RHINO_NOT_CALLED_BY_INTRPT = 1000u,
    RHINO_TRY_AGAIN,
    RHINO_WORKQUEUE_EXIST = 1100u,
    RHINO_WORKQUEUE_NOT_EXIST,
    RHINO_WORKQUEUE_WORK_EXIST,
    RHINO_WORKQUEUE_BUSY,
    RHINO_WORKQUEUE_WORK_RUNNING,
    RHINO_TASK_STACK_OVF = 1200u,
    RHINO_INTRPT_STACK_OVF,
    RHINO_STATE_ALIGN = INT_MAX /* keep enum 4 bytes at 32bit machine */
} kstat_t;
```

2.2.5. 信号量

对于多任务，甚至多核的操作系统，需要访问共同的系统资源。共享资源包括软件资源和硬件资源，软件共享资源主要是共享内存，包括共享变量、共享队列等等，硬件共享资源包括一些硬件设备的访问，例如：输入/输出设备。为了避免多个任务访问共享资源时相互影响甚至冲突，需要对共享资源进行保护，有下列几种处理方式：开关中断、信号量（semphore）、互斥量（mutex）。

开关中断：一般用于单核平台多任务之间的互斥，通过关闭任务的调度，从而达到单任务访问共享资源的目的。缺点是会影响中断响应时间。

信号量：多任务可以通过获取信号量来获取访问共享资源，可以配置信号量的数目，让多个任务同时获取信号量，当信号量无法获取时，相关任务会按照优先级排序等待信号量释放，并让出CPU资源；缺点是存在高低任务优先级反转的问题。

互斥量：任务也是通过获取mutex来获取访问共享资源的门禁，但是单次只有一个任务能获取到该互斥量。互斥量通过动态调整任务的优先级来解决高低优先级反转的问题。

本章节介绍AliOS Thngs上的信号量接口。

API 列表

aos_sem_new()	创建信号量对象
aos_sem_free()	删除信号量对象
aos_sem_wait()	请求一个信号量
aos_sem_signal()	释放一个信号量

aos_sem_is_valid()	判断信号量对象是否有效
aos_sem_signal_all()	释放信号量，并唤醒所有阻塞在该信号量上的任务

使用

添加该组件

信号量是AliOS Things 默认添加的组件，开发者无需再手动添加。

包含头文件

```
#include <aos/kernel.h>
```

使用示例

示例说明：当前任务创建一个信号量和子任务，并等待子任务释放信号量。

```
static aos_sem_t g_sem_taskexit_sync;
  unsigned int stack_size = 1024;
  int ret = -1;
  ....
  static void task1(void *arg)
  {
    ....
    /*释放信号量*/
    aos_sem_signal(&g_sem_taskexit_sync);
    ....
  }
  /*当前任务：创建信号量，信号量初始count为0*/
  ret = aos_sem_new(&g_sem_taskexit_sync, 0);
  if (ret != 0) {
    printf("sem create failed\r\n");
      ...
  }
  ....
  /*判断信号量是否可用*/
  ret = aos_sem_is_valid(&g_sem_taskexit_sync);
  if (ret == 0) {
    printf("sem is invalid\r\n");
      ...
   }
  /*创建新任务task1*/
  ret = aos_task_new("task1", task1, NULL, stack_size);
  if (ret != 0) {
    printf("timer create failed\r\n");
    ...
  }
  ....
  /*获取信号量，由于初始值为0，这里获取不到信号量，当前任务进入睡眠并发生切换。
   参数 AOS_WAIT_FOREVER 表示永久等待，知道获得信号量 */
  aos_sem_wait(&g_sem_taskexit_sync, AOS_WAIT_FOREVER);
  /*获取到信号量，当前任务继续执行下去*/
  printf("task1 exit!\r\n");
  ....
  /*删除信号量*/
  aos_sem_free(&g_sem_taskexit_sync);
```

API 详情

信号量的应用层API说明请参考include/aos/kernel.h

aos_sem_new()

创建信号量对象。

函数原型

```
int aos_sem_new(aos_sem_t *sem, int count);
```

输入参数

sem	信号量句柄，需要用户定义一个aos_sem_t结构体变量	
count	信号量初始个数	1

返回参数

0表示成功，其他值表示失败。具体的返回值见本文档返回参数定义小节。

调用示例

```
aos_sem_t sem_handle;
int status;
status = aos_sem_new(&sem_handle, 1);
```

aos_sem_free()

删除信号量对象。

函数原型

```
void aos_sem_free(aos_sem_t *sem);
```

输入参数

sem	信号量句柄	

返回参数

无

调用示例

```
aos_sem_t sem_handle;
int status;
status = aos_sem_new(&sem_handle, 1);
......
aos_sem_free(&sem_handle);
```

aos_sem_signal()

释放一个信号量。

函数原型

```
void aos_sem_signal(aos_sem_t *sem);
```

输入参数

sem	信号量句柄	

返回参数

无

调用示例

```
aos_sem_t sem_handle;
int status;
status = aos_sem_new(&sem_handle, 1);
......
aos_sem_signal(&sem_handle);
```

aos_sem_signal_all()

释放信号量，并唤醒所有阻塞在该信号量上的任务。

函数原型

```
void aos_sem_signal_all(aos_sem_t *sem);
```

输入参数

sem	信号量句柄	

返回参数

无

调用示例

```
aos_sem_t sem_handle;
int status;
status = aos_sem_new(&sem_handle, 1);
......
aos_sem_signal_all(&sem_handle);
```

aos_sem_wait()

请求一个信号量，若获取不到且超时时间不为0，则任务将被阻塞。

函数原型

```
int aos_sem_wait(aos_sem_t *sem, unsigned int timeout);
```

输入参数

sem	信号量句柄	
timeout	等待超时时间。0表示不超时，立即返回；AOS_WAIT_FOREVER表示永久等待；其他数值表示超时时间，单位ms	

返回参数

0表示成功，其他值表示失败。具体的返回值见本文档返回参数定义小节。

调用示例

```
aos_sem_t sem_handle;
int status;
status = aos_sem_new(&sem_handle, 1);
......
status = aos_sem_wait(&sem_handle, AOS_WAIT_FOREVER);
```

aos_sem_is_valid()

判断信号量对象是否有效。

函数原型

```
int aos_sem_is_valid(aos_sem_t *sem);
```

输入参数

sem	信号量句柄	

返回参数

1表示有效，0表示无效。

调用示例

```
extern aos_sem_t sem_handle;
int status;
/* 信号量句柄在别的文件定义，因此在使用前先判断一下是否有效 */
status = aos_sem_is_valid(&sem_handle);
if (status == 0) {
    printf("sem is invalid\r\n");
    ...
}
```

其他

返回参数定义

返回值定义在core/rhino/include/k_err.h文件中。该文件为内部文件，出错时可根据返回值查阅该文件确认出错原因。

```
typedef enum
{
  RHINO_SUCCESS = 0u,
  RHINO_SYS_FATAL_ERR,
  RHINO_SYS_SP_ERR,
  RHINO_RUNNING,
  RHINO_STOPPED,
  RHINO_INV_PARAM,
  RHINO_NULL_PTR,
  RHINO_INV_ALIGN,
  RHINO_KOBJ_TYPE_ERR,
  RHINO_KOBJ_DEL_ERR,
  RHINO_KOBJ_DOCKER_EXIST,
```

```
    RHINO_KOBJ_BLK,
    RHINO_KOBJ_SET_FULL,
    RHINO_NOTIFY_FUNC_EXIST,
    RHINO_MM_POOL_SIZE_ERR = 100u,
    RHINO_MM_ALLOC_SIZE_ERR,
    RHINO_MM_FREE_ADDR_ERR,
    RHINO_MM_CORRUPT_ERR,
    RHINO_DYN_MEM_PROC_ERR,
    RHINO_NO_MEM,
    RHINO_RINGBUF_FULL,
    RHINO_RINGBUF_EMPTY,
    RHINO_SCHED_DISABLE = 200u,
    RHINO_SCHED_ALREADY_ENABLED,
    RHINO_SCHED_LOCK_COUNT_OVF,
    RHINO_INV_SCHED_WAY,
    RHINO_TASK_INV_STACK_SIZE = 300u,
    RHINO_TASK_NOT_SUSPENDED,
    RHINO_TASK_DEL_NOT_ALLOWED,
    RHINO_TASK_SUSPEND_NOT_ALLOWED,
    RHINO_TASK_CANCELED,
    RHINO_SUSPENDED_COUNT_OVF,
    RHINO_BEYOND_MAX_PRI,
    RHINO_PRI_CHG_NOT_ALLOWED,
    RHINO_INV_TASK_STATE,
    RHINO_IDLE_TASK_EXIST,
    RHINO_NO_PEND_WAIT = 400u,
    RHINO_BLK_ABORT,
    RHINO_BLK_TIMEOUT,
    RHINO_BLK_DEL,
    RHINO_BLK_INV_STATE,
    RHINO_BLK_POOL_SIZE_ERR,
    RHINO_TIMER_STATE_INV = 500u,
    RHINO_NO_THIS_EVENT_OPT = 600u,
    RHINO_BUF_QUEUE_INV_SIZE = 700u,
    RHINO_BUF_QUEUE_SIZE_ZERO,
    RHINO_BUF_QUEUE_FULL,
    RHINO_BUF_QUEUE_MSG_SIZE_OVERFLOW,
    RHINO_QUEUE_FULL,
    RHINO_QUEUE_NOT_FULL,
    RHINO_SEM_OVF = 800u,
    RHINO_SEM_TASK_WAITING,
    RHINO_MUTEX_NOT_RELEASED_BY_OWNER = 900u,
    RHINO_MUTEX_OWNER_NESTED,
    RHINO_MUTEX_NESTED_OVF,
    RHINO_NOT_CALLED_BY_INTRPT = 1000u,
    RHINO_TRY_AGAIN,
    RHINO_WORKQUEUE_EXIST = 1100u,
    RHINO_WORKQUEUE_NOT_EXIST,
    RHINO_WORKQUEUE_WORK_EXIST,
    RHINO_WORKQUEUE_BUSY,
    RHINO_WORKQUEUE_WORK_RUNNING,
    RHINO_TASK_STACK_OVF = 1200u,
    RHINO_INTRPT_STACK_OVF,
    RHINO_STATE_ALIGN = INT_MAX /* keep enum 4 bytes at 32bit machine */
} kstat_t;
```

使用注意事项

1）在中断中禁止信号量获取检测 中断服务程序的执行不能被阻塞，因此不能在中断中调用请求信号量的接口。有些内核将这种判断处理交由上层软件进行判断和使用，AliOS Things的内核会在请求信号量时进行检测，如果是中断上下文，则直接返回失败。在中断上下文可以释放信号量。

2） 请求信号量时的非等待、永远等待、延时的区别 上层应用在获取信号量时，需要按照实际的需求来安排信号量获取策略。 `aos_sem_wait` 传入延时为0时，当获取不到信号量会立即返回并报失败；超时时间为AOS_WAIT_FOREVER时，会永久等待，直到获取到信号量，可能会造成该任务无法继续运行；其他值表示最大延迟的时间上限，达到上限时，即使未获取到信号量，任务也会被唤醒，并返回状态为超时。

3） 信号量优先级反转问题 优先级反转问题出现在高、低两个优先级任务同时使用信号量访问互斥资源时。当高优先级任务请求的信号量被低优先级任务已经占用时，高优先级任务会被阻塞。此时如果有一个中优先级任务，那么中优先级任务能抢占低优先级任务而得到CPU资源。这个时候出现了一种情况，由于资源被低优先级任务占用但由于低优先级任务得不到CPU资源而没有机会运行，最终导致高优先级任务得不到调度。

互斥信号量支持解决优先级反转问题，其方法是动态提高占用信号量的任务的运行优先级。

2.2.6. 互斥信号量

互斥信号量（mutex）的获取是完全互斥的，即同一时刻，mutex只能被一个任务获取。而信号量（sem）按照起始count的配置，存在多个任务获取同一信号量的情况，直到count减为0，则后续任务无法再获取信号量，当然sem的count初值设置为1，同样有互斥的效果。 Mutex的释放必须由占有该mutex的任务进行，其他任务进行释放，会直接返回失败。 为了解决优先级反转问题，高优先级的任务获取mutex时，如果该mutex被某低优先级的任务占用， 会动态提升该低优先级任务的优先级等于高优先级，并且将该优先级值依次传递给该低优先级任务依赖的互斥量关联的任务，以此递归下去。当某任务释放mutex时，会查找该任务的基础优先级，以及获取到的互斥量所阻塞的最高优先级的任务的优先级，取两者中高的优先级来重新设定此任务的优先级。总的原则就是，高优先级任务被mutex阻塞时，会将占用该mutex的低优先级任务临时提高；mutex释放时，相应任务的优先级需要恢复。

API列表

aos_mutex_new()	动态创建互斥信号量
aos_mutex_free()	删除互斥信号量
aos_mutex_lock()	请求互斥信号量
aos_mutex_unlock()	释放互斥信号量
aos_mutex_is_valid()	判断是否为有效的互斥信号量

API详情

aos_mutex_new()

函数原型：

int aos_mutex_new(aos_mutex_t *mutex)

定义描述

描述	动态创建互斥信号量
入参	mutex：互斥信号量结构体指针，需要用户定义一个aos_mutex_t结构体，并把该结构体指针传入

返回值	类型：int 返回成功或失败；返回0表示互斥信号量创建成功，非0表示失败。

aos_mutex_free()

函数原型：

void aos_mutex_free(aos_mutex_t *mutex)

定义描述

描述	删除互斥信号量
入参	mutex：互斥信号量结构体指针
返回值	无

aos_mutex_lock()

函数原型：

int aos_mutex_lock(aos_mutex_t *mutex, unsigned int timeout)

定义描述

描述	请求互斥信号量
入参	mutex：互斥信号量结构体指针
	timeout：等待超时时间，单位ms。0表示非阻塞请求，立即返回；AOS_WAIT_FOREVER表示永久等待；其他值则等待timeout ms，超时后返回。可通过返回值判断是否获得互斥信号量。
返回值	类型：int 返回成功或失败；返回0表示请求互斥信号量成功，非0表示失败。

aos_mutex_unlock()

函数原型：

int aos_mutex_unlock(aos_mutex_t *mutex)

定义描述

描述	释放互斥信号量
入参	mutex：互斥信号量结构体指针
返回值	类型：int 返回成功或失败；返回0表示释放互斥信号量成功，非0表示失败。

aos_mutex_is_valid()

函数原型：

int aos_mutex_is_valid(aos_mutex_t *mutex)

定义描述

描述	判断是否为有效的互斥信号量
入参	mutex：互斥信号量结构体指针
返回值	类型：int 返回有效或无效； 返回0表示mutex为无效互斥信号量，返回1表示mutex为有效的互斥信号量。

调用示例

```
#include <aos/aos.h>
aos_mutex_t test_mutex;
void task1_entry() {
  int ret = -1;
  while (1) {
    ret = aos_mutex_is_valid(&test_mutex);
    if (ret != 1) {
      return;
    }
    ret = aos_mutex_lock(&test_mutex, AOS_WAIT_FOREVER);
    if (ret != 0) {
      continue;
    }
    /*访问临界资源*/
    printf("task1 entry access critical zone\n");
    aos_mutex_unlock(&test_mutex);
  }
}
void task2_entry() {
  int ret = -1;
  while (1) {
    ret = aos_mutex_is_valid(&test_mutex);
    if (ret != 1) {
      return;
    }
    ret = aos_mutex_lock(&test_mutex, 1000);
    if (ret != 0) {
      continue;
    }
    /*访问临界资源*/
    printf("task2 entry access critical zone\n");
    aos_mutex_unlock(&test_mutex);
  }
}
int application_start(int argc, char *argv[])
{
  int ret = -1;
  ret = aos_mutex_new(&test_mutex);
  if (ret != 0) {
    return;
  }
  aos_task_new("task1", task1_entry, NULL, 512);
  aos_task_new("task2", task2_entry, NULL, 512);
}
```

使用注意事项

- 只有获得互斥信号量的任务才能释放互斥信号量。中断也不能释放互斥信号量。
- 互斥信号量的请求与释放需成对使用，避免只有请求没有释放，导致其他任务无法获得互斥信号量。

2.2.7. 消息队列

多任务系统中，任务间互相同步等待共享资源，我们一般会使用信号量，如果需要互斥，则使用互斥量。而任务间互相收发消息则可以使用消息队列。消息队列（queue）使用类似信号量的机制进行任务间的同步，并使用环形缓冲池（ring buffer）来进行消息的队列缓冲管理，以达到任务间收发消息的阻塞和通知管理。Queue的实现目的在于任务间互相收发消息。一般如果有信号量机制，用户就可以自己实现一套任务间的阻塞和通知收发功能，其本质在于接收方通过信号量的获取来开始接收消息，发送方通过信号量的释放来通知接收方处理。接收任务在无消息时被阻塞，消息到来时被唤醒处理。Queue就是基于这样一种类信号量机制来进行消息的收发。再加上ring buffer的缓冲机制来缓存任务间的消息队列，就组合成了本章的消息队列（queue）。其既包含消息的缓冲队列，又包含了消息的通知机制。消息队列模块整体受宏 `RHINO_CONFIG_BUF_QUEUE` 开关控制，对应的AOS API接口实现位于： `core/osal/aos/rhino.c` 中 `RHINO_CONFIG_BUF_QUEUE` 宏定义包含实现；对应的krhino内部实现位于： `core/rhino/k_buf_queue.c` 。

包含头文件

```
#include <aos/kernel.h>  //直接对应头文件
```

API列表

aos_queue_new()	创建一个队列，指定缓冲区大小，以及最大数据单元大小
aos_queue_free()	删除一个队列，并释放阻塞在其中的任务
aos_queue_send()	向queue内发送数据，并唤醒存在的第一个高优先级阻塞任务
aos_queue_recv()	从queue内收取数据，如没有数据则阻塞当前任务等待
aos_queue_is_valid()	判断一个队列queue是否有效
aos_queue_buf_ptr()	获取一个队列queue的缓冲区起点

API详情

aos_queue_new

创建一个队列，指定缓冲区大小，以及最大数据单元大小。

函数原型

```
int aos_queue_new(aos_queue_t *queue, void *buf, unsigned int size, int max_msg)
```

输入参数

aos_queue_t *queue	queue队列描述结构体指针；需要用户定义一个queue结构体	aos_queue_t g_queue; 传入&g_queue
void *buf	此queue队列的缓冲区起点	char buf[1000];传入buf
unsigned int size	此queue队列的缓冲区大小	1000

int max_msg	一次存入缓冲区的最大数据单元	50

返回参数

0	执行成功
其他	返回失败

调用示例

```
static aos_queue_t  g_queue;
static char  queue_buf[TEST_CONFIG_QUEUE_BUF_SIZE];
ret = aos_queue_new(&g_queue, queue_buf, 1000, 50);
```

aos_queue_free

删除一个队列，并释放阻塞在其中的任务。

函数原型

```
void aos_queue_free(aos_queue_t *queue)
```

输入参数

aos_queue_t *queue	删除一个队列，并释放阻塞在其中的任务	aos_queue_new创建传入的queue，如&g_queue

返回参数

无

调用示例

```
aos_queue_free(&g_queue);
```

aos_queue_send

向queue内发送数据，并唤醒存在的第一个高优先级阻塞任务。

函数原型

```
int aos_queue_send(aos_queue_t *queue, void *msg, unsigned int size)
```

输入参数

aos_queue_t *queue	queue队列描述结构体指针	aos_queue_new创建传入的queue，如&g_queue
void *msg	发送数据起始内存	用户定义的起始内存如:void* msg_send
unsigned int size	发送数据大小	用户定义的内存大小如:30

返回参数

0	执行成功
其他	返回失败

调用示例

```
char *msg_send = "hello,queue!";
ret = aos_queue_send(&g_queue, msg_send, strlen(msg_send));
```

aos_queue_recv

从queue内收取数据，如没有数据则阻塞当前任务等待。

函数原型

```
int aos_queue_recv(aos_queue_t *queue, unsigned int ms, void *msg, unsigned int *size)
```

输入参数

aos_queue_t *queue	queue队列描述结构体指针	aos_queue_new创建传入的queue，如&g_queue
unsigned int ms	传入0表示不超时，立即返回； AOS_WAIT_FOREVER表示永久等待； 其他数值表示超时时间，单位ms	参考参数描述
void *msg	出参；返回获取到的数据的内存指针	用户定义的表示地址变量如:void* msg_recv
unsigned int *size	出参；返回获取到的数据大小	用户定义的表示大小变量，如int size_recv ;

返回参数

0	执行成功
其他	返回失败

调用示例

```
char msg_recv[16] = {0};
unsigned int size_recv = 16;
memset(msg_recv, 0, size_recv);
ret = aos_queue_recv(&g_queue, 100, msg_recv, &size_recv);
```

aos_queue_is_valid

判断一个队列queue是否有效。

函数原型

```
int aos_queue_is_valid(aos_queue_t *queue)
```

输入参数

aos_queue_t *queue	判断一个队列queue是否有效	aos_queue_new创建传入的queue，如&g_queue

返回参数

0	queue无效
1	queue有效

调用示例

```
ret = aos_queue_is_valid(&g_queue);
```

aos_queue_buf_ptr

获取一个队列queue的缓冲区起点。

函数原型

```
void* aos_queue_buf_ptr(aos_queue_t *queue)
```

输入参数

aos_queue_t *queue	queue队列描述结构体指针	aos_queue_new创建传入的queue，如&g_queue

返回参数

NULL	获取失败
void*	返回队列queue的缓冲区起点

调用示例

```
void* bufstart;
bufstart = aos_queue_buf_ptr(&g_queue);
```

相关定义

相关宏定义

```
typedef struct
{
  void *hdl;
} aos_hdl_t;
typedef aos_hdl_t aos_queue_t;
```

2.2.8. 工作队列

在一个操作系统中，如果我们需要进行一项工作处理，往往需要创建一个任务来加入内核的调度队列。一个任务对应一个处理函数，如果要进行不同的事务处理，则需要创建多个不同的任务。任务作为cpu调度的基础单元，任务数量越大，则调度成本越高。工作队列（workqueue）机制简化了基本的任务创建和处理机制，一个workqueue对应一个实体task任务处理，workqueue下面可以挂接多个work实体，每一个work实体都能对应不同的处理接口。即用户只需要创建一个workqueue，则可以完成多个挂接不同处理函数的工作队列。其次，当某些实时性要求较高的任务中，需要进行较繁重钩子处理时，可以将其处理函数挂接在workqueue中，其执行过程将位于workqueue的上下文，而不会占用原有任务的处理资源。另外，workqueue还提供了work的延时处理机制，用户可以选择立即执行或是延时处理。由上可见，我们在需要创建大量实时性要求不高的任务时，可以使用workqueue来统一调度；或者将任务中实时性要求不高的部分处理延后到workqueue中处理。如果需要设置延后处理，则需要使用work机制。另外该机制不支持周期work的处理。 工作队列功能的相关源码位于： `core/rhino` 目录中。

包含头文件

```
#include <aos/kernel.h>
```

API列表

aos_workqueue_create()	创建一个工作队列，内部会创建一个任务关联workqueue
aos_work_init()	初始化一个work，暂不执行
aos_work_destroy()	删除一个work
aos_work_run()	运行一个work，使其在某worqueue内调度执行
aos_work_sched()	运行一个work，使其在默认工作队列g_workqueue_default内调度执行
aos_work_cancel()	取消一个work，使其从所在的工作队列中删除

API 详情

aos_workqueue_create()

函数原型：

int aos_workqueue_create(aos_workqueue_t *workqueue, int pri, int stack_size);

定义描述

描述	创建一个工作队列，内部会创建一个任务关联workqueue
入参	workqueue：工作队列描述结构体指针；需要用户定义一个workqueue结构体
	pri: 工作队列优先级，实际是关联任务优先级
	stack_size：任务栈大小（单位：字节）
返回值	类型：int 返回成功或失败

aos_work_init()

函数原型：

int aos_work_init(aos_work_t *work, void (*fn)(void *), void *arg, int dly);

定义描述

描述	初始化一个work，暂不执行
入参	work：work工作描述结构体指针；需要用户定义一个work结构体
	fn: work回调处理函数
	arg：work回调处理参数
	dly：延迟处理时间，单位ms，0表示不延迟
返回值	类型：int 返回成功或失败

aos_work_destroy()

函数原型：

void aos_work_destroy(aos_work_t *work);

定义描述

描述	删除一个work
入参	work：work工作描述结构体指针
返回值	无

aos_work_run()

函数原型：

int aos_work_run(aos_workqueue_t *workqueue, aos_work_t *work);

定义描述

描述	运行一个work，使其在某worqueue内调度执行
入参	workqueue：工作队列描述结构体指针
	work：需要执行的工作描述结构体指针
返回值	类型：int 返回成功或失败

aos_work_sched()

函数原型：

int aos_work_sched(aos_work_t *work);

定义描述

描述	运行一个work，使其在默认工作队列g_workqueue_default内调度执行

入参	work：需要执行的工作描述结构体指针
返回值	类型：int 返回成功或失败

aos_work_cancel()

函数原型：

int aos_work_cancel(aos_work_t *work);

定义描述

描述	取消一个work，使其从所在的工作队列中删除
入参	work：需要取消的工作描述结构体指针
返回值	类型：int 返回成功或失败

调用示例

创建一个工作队列，初始化一个work，把这个work加入到这个工作队列中，主线程等待，work执行，work执行完毕销毁

```
static aos_workqueue_t workqueue;
static aos_work_t     work;
static aos_sem_t      sync_sem;
static void workqueue_custom(void *arg)
{
  aos_msleep(1000);
  printf("workqueue custom");
  aos_sem_signal(&sync);
}
static void test_workqueue()
{
  int ret = 0;
  aos_sem_new(&sync_sem, 0);
  /* 创建一个工作队列workqueue，内部会创建一个任务关联该workqueue */
  aos_workqueue_create(&workqueue, 10, 1024);
  /* 初始化一个work，暂不执行，等待run */
  aos_work_init(&work, workqueue_custom, NULL, 100);
  /* 运行work，使其在某worqueue内调度执行 */
  aos_work_run(&workqueue, &work);
  /* sem等待，workqueue_custom得到执行 */
  aos_sem_wait(&sync_sem, AOS_WAIT_FOREVER);
  /* 释放sem */
  aos_sem_free(&sync_sem);
  /* 销毁work */
  aos_work_destroy(&work);
}
/* aos_work_cancel(aos_work_t *work) */
/* 删除work前，要确保work没有正在或将要被worqueue执行，否则会返回错误 */
/* 删除workqueue需要确保没有待处理或正在处理的wok，否则会返回错误 */
```

2.2.9. 其他系统相关接口

aos_reboot()	重启系统
aos_get_hz()	返回系统每秒tick频率
aos_version_get()	返回内核版本号，也是AliOS-Things的基础版本号
aos_now()	返回内核启动至今的ns数
aos_now_ms()	返回内核启动至今的ms数
aos_msleep()	将当前任务睡眠

API详情

aos_reboot()

函数原型：

void aos_reboot(void);

定义描述

描述	重启单板
参数	无
返回值	无

调用示例

`aos_reboot()` 接口会调用厂商提供的reboot接口，重启单板。

aos_get_hz()

函数原型：

int aos_get_hz(void);

定义描述

描述	返回系统每秒tick数
参数	无
返回值	类型：int

调用示例

`aos_get_hz()` 会返回返回系统每秒tick频率，这个数值由位于 `k_config.h` 中的 `RHINO_CONFIG_TICKS_PER_SECOND` 定义.例如：

```
#define RHINO_CONFIG_TICKS_PER_SECOND     100
```

表示每秒100个tick，即每个tick的值为10ms。

aos_version_get()

函数原型：

const char *aos_version_get(void);

定义描述

描述	返回内核版本号，也是AliOS-Things的基础版本号
参数	无
返回值	类型：char *;例如： AOS-R-2.0.0

调用示例

aos_version_get() 会返回内核版本号，这个值在 aos_common.h 中被如下定义：

```
const char *SYSINFO_KERNEL_VERSION="AOS-R-2.0.0";
```

即返回字符串常量"AOS-R-2.0.0"。

aos_now()

函数原型：

long long aos_now(void);

定义描述

描述	返回内核启动至今的ns数
参数	无
返回值	类型：long long (即64位)

调用示例

aos_now() 会返回内核启动至今的ns数，注意返回值为64位。

aos_now_ms()

函数原型：

long long aos_now_ms(void);

定义描述

描述	返回内核启动至今的ms数
参数	无
返回值	类型：long long (即64位)

调用示例

aos_now_ms() 会返回内核启动至今的ms数，注意返回值为64位。

aos_msleep()

函数原型：

void aos_msleep(int ms);

定义描述

描述	将当前任务睡眠，单位ms
参数	输入参数类型int，表示睡眠ms数
返回值	无

调用示例

`aos_msleep()` 会将当前任务睡眠并进行任务调度，如果参数传入为0，会返回无效参数。

2.3. 硬件抽象函数

特殊说明：此处为所有hal对外接口说明，对于具体单板不一定在硬件上或者软件适配时支持和对接了所有的hal接口。具体需要根据特定单板的hal对接情况来使用。

2.3.1. GPIO

对于不同底层驱动的GPIO操作实现，统一封装成本文所述hal接口。上层使用相关功能时，统一调用hal层接口，以保证app的通用性。 hal相关头文件位于目录：include/aos/hal。hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/。

API列表

hal_gpio_init	初始化指定GPIO管脚
hal_gpio_output_high	使指定GPIO输出高电平
hal_gpio_output_low	使指定GPIO输出低电平
hal_gpio_output_toggle	使指定GPIO输出翻转
hal_gpio_input_get	获取指定GPIO管脚的输入值
hal_gpio_enable_irq	使能指定GPIO的中断模式，挂载中断服务函数
hal_gpio_disable_irq	关闭指定GPIO的中断
hal_gpio_clear_irq	清除指定GPIO的中断状态
hal_gpio_finalize	关闭指定GPIO

API详情

请参考include/aos/hal/gpio.h

相关结数据结构

gpio_dev_t

```
typedef struct {
  uint8_t     port;   /* gpio逻辑端口号 */
  gpio_config_t config; /* gpio配置信息 */
  void      *priv;   /* 私有数据 */
} gpio_dev_t;
```

gpio_config_t

```
typedef enum {
  ANALOG_MODE,          /* 管脚用作功能引脚，如用于pwm输出，uart的输入引脚 */
  IRQ_MODE,           /* 中断模式，配置为中断源 */
  INPUT_PULL_UP,        /* 输入模式，内部包含一个上拉电阻 */
  INPUT_PULL_DOWN,      /* 输入模式，内部包含一个下拉电阻 */
  INPUT_HIGH_IMPEDANCE,   /* 输入模式，内部为高阻模式 */
  OUTPUT_PUSH_PULL,     /* 输出模式，普通模式 */
  OUTPUT_OPEN_DRAIN_NO_PULL, /* 输出模式，输出高电平时，内部为高阻状态 */
  OUTPUT_OPEN_DRAIN_NO_PULL, /* 输出模式，输出高电平时，被内部电阻拉高 */
} gpio_config_t;
```

gpio_irq_trigger_t

```
typedef enum {
  IRQ_TRIGGER_RISING_EDGE = 0x1, /* 上升沿触发 */
  IRQ_TRIGGER_FALLING_EDGE = 0x2, /* 下降沿触发 */
  IRQ_TRIGGER_BOTH_EDGES  = IRQ_TRIGGER_RISING_EDGE | IRQ_TRIGGER_FALLING_EDGE,                    /* 上升
沿下降沿均触发 */
} gpio_irq_trigger_t;
```

gpio_irq_handler_t

```
typedef void (*gpio_irq_handler_t)(void *arg);
```

hal_gpio_init

GPIO 初始化

函数原型

```
int32_t hal_gpio_init(gpio_dev_t *gpio);
```

参数

gpio_dev_t *gpio	入参	GPIO设备描述，定义需要初始化的GPIO管脚的相关特性	用户自定义该结构体

返回值

类型：int 返回成功或失败, 返回0表示GPIO初始化成功，非0表示失败

调用示例

```
#define GPIO_LED_IO 18
gpio_dev_t led;
led.port = GPIO_LED_IO;
/* set as output mode */
led.config = OUTPUT_PUSH_PULL;
ret = hal_gpio_init(&led);
```

hal_gpio_output_high

某GPIO输出高电平

函数原型

```
int32_t hal_gpio_output_high(gpio_dev_t *gpio)
```

参数

gpio_dev_t *gpio	入参	GPIO设备描述	使用hal_gpio_init初始化传入值，需要预设输出模式

返回值

类型：int 返回成功或失败, 返回0表示GPIO输出高电平成功，非0表示失败

调用示例

```
#define GPIO_IO_OUT 19
gpio_dev_t gpio_out;
gpio_out.port = GPIO_IO_OUT;
/* set as output mode */
gpio_out.config = OUTPUT_PUSH_PULL;
ret = hal_gpio_init(&gpio_out);
ret= hal_gpio_output_high(&gpio_out);
```

hal_gpio_output_low

某GPIO输出低电平

函数原型

```
int32_t hal_gpio_output_low(gpio_dev_t *gpio)
```

参数

gpio_dev_t *gpio	入参	GPIO设备描述	使用hal_gpio_init初始化传入值，需要预设输出模式

返回值

类型：int 返回成功或失败, 返回0表示GPIO输出低电平成功，非0表示失败

调用示例

```
#define GPIO_IO_OUT 19
gpio_dev_t gpio_out;
gpio_out.port = GPIO_IO_OUT;
/* set as output mode */
gpio_out.config = OUTPUT_PUSH_PULL;
ret = hal_gpio_init(&gpio_out);
ret= hal_gpio_output_low(&gpio_out);
```

hal_gpio_output_toggle

某GPIO输出翻转

函数原型

```
int32_t hal_gpio_output_toggle(gpio_dev_t* gpio)
```

参数

gpio_dev_t *gpio	入参	GPIO设备描述	使用hal_gpio_init初始化传入值，需要预设输出模式

返回值

类型：int 返回成功或失败，返回0表示GPIO翻转成功，非0表示失败。

调用示例

```
#define GPIO_IO_OUT 19
gpio_dev_t gpio_out;
gpio_out.port = GPIO_IO_OUT;
/* set as output mode */
gpio_out.config = OUTPUT_PUSH_PULL;
ret = hal_gpio_init(&gpio_out);
ret= hal_gpio_output_toggle(&gpio_out);
```

hal_gpio_input_get

获取某GPIO管脚输入值

函数原型

```
int32_t hal_gpio_input_get(gpio_dev_t *gpio, uint32_t *value)
```

参数

gpio_dev_t *gpio	入参	GPIO设备描述	使用hal_gpio_init初始化传入值，需要预设输入模式
uint32_t *value	出参	需要获取的管脚值存放地址	自定义uint32_t数据结构，传入地址

返回值

类型：int 返回成功或失败，返回0表示GPIO输入获取成功，非0表示失败。

调用示例

```
#define GPIO_IO_OUT 19
uint32_t pinval = 0;
gpio_dev_t gpio_out;
gpio_out.port = GPIO_IO_OUT;
/* set as input mode */
gpio_out.config = INPUT_PULL_UP;
ret = hal_gpio_init(&gpio_out);
ret= hal_gpio_input_get(&gpio_out,&pinval);
```

hal_gpio_enable_irq

使能指定GPIO的中断模式，挂载中断服务函数，需要预先调用hal_gpio_init，设置IRQ_MODE。

函数原型

```
int32_t hal_gpio_enable_irq(gpio_dev_t *gpio, gpio_irq_trigger_t trigger,gpio_irq_handler_t handler, void *arg)
```

参数

gpio_dev_t *gpio	入参	GPIO设备描述	使用hal_gpio_init初始化传入值
gpio_irq_trigger_t	入参	中断的触发模式，上升沿、下降沿还是都触发	直接使用gpio_irq_trigger_t枚举
gpio_irq_handler_t handler	入参	中断服务函数指针，中断触发后将执行指向的函数	
void *arg	入参	中断服务函数的入参	

返回值

类型：int 返回成功或失败，返回0使能中断成功，非0表示失败。

调用示例

```
#define GPIO_IO_INT 19
void gpio_irq_fun(void *arg)
{
}
gpio_dev_t gpio_int;
gpio_int.port = GPIO_IO_INT;
/* set as int mode */
gpio_int.config = IRQ_MODE;
ret = hal_gpio_init(&gpio_int);
/* int triggered int rising edge */
ret= hal_gpio_enable_irq(&gpio_int, IRQ_TRIGGER_RISING_EDGE,gpio_irq_fun,NULL);
```

hal_gpio_disable_irq

关闭指定GPIO的中断。

函数原型

```
int32_t hal_gpio_disable_irq(gpio_dev_t *gpio)
```

参数

gpio_dev_t *gpio	入参	GPIO设备描述	使用hal_gpio_init初始化传入值

返回值

类型：int 返回成功或失败，返回0表示中断去使能成功，非0表示失败。

调用示例

```
ret= hal_gpio_disable_irq(&gpio_int);
```

hal_gpio_clear_irq

清除指定GPIO的中断。

函数原型

```
int32_t hal_gpio_clear_irq(gpio_dev_t *gpio)
```

参数

gpio_dev_t *gpio	入参	GPIO设备描述	使用hal_gpio_init初始化传入值

返回值

类型：int 返回成功或失败，返回0表示清中断成功，非0表示失败。

调用示例

```
ret= hal_gpio_clear_irq(&gpio_int);
```

hal_gpio_finalize

关闭指定GPIO，及其中断。

函数原型

```
int32_t hal_gpio_finalize(gpio_dev_t *gpio)
```

参数

gpio_dev_t *gpio	入参	GPIO设备描述	使用hal_gpio_init初始化传入值

返回值

类型：int 返回成功或失败，返回0表示关闭成功，非0表示失败。

调用示例

```
ret= hal_gpio_finalize(&gpio_int);
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/gpio.h"
```

使用示例

GPIO作为输出

```
#include <aos/hal/gpio.h>
#define GPIO_LED_IO 18
/* define dev */
gpio_dev_t led;
int application_start(int argc, char *argv[])
{
  int ret = -1;
  /* gpio port config */
  led.port = GPIO_LED_IO;
  /* set as output mode */
  led.config = OUTPUT_PUSH_PULL;
  /* configure GPIO with the given settings */
  ret = hal_gpio_init(&led);
  if (ret != 0) {
    printf("gpio init error !\n");
  }
  /* output high */
  hal_gpio_output_high(&led);
  /* output low */
  hal_gpio_output_low(&led);
  /* toggle the LED every 1s */
  while(1) {
    /* toggle output */
    hal_gpio_output_toggle(&led);
    /* sleep 1000ms */
    aos_msleep(1000);
  };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

-

GPIO作为中断输入

```
#include <aos/hal/gpio.h>
#define GPIO_BUTTON_IO 5
/* define dev */
gpio_dev_t button1;
/* pressed flag */
int button1_pressed = 0;
void button1_handler(void *arg)
{
 button1_pressed = 1;
}
int application_start(int argc, char *argv[])
{
   int ret = -1;
 /* input pin config */
 button1.port = GPIO_BUTTON_IO;
 /* set as interrupt mode */
 button1.config = IRQ_MODE;
 /* configure GPIO with the given settings */
 ret = hal_gpio_init(&button1);
 if (ret != 0) {
    printf("gpio init error !\n");
 }
 /* gpio interrupt config */
 ret = hal_gpio_enable_irq(&button1, IRQ_TRIGGER_BOTH_EDGES,
                           button1_handler, NULL);
 if (ret != 0) {
    printf("gpio irq enable error !\n");
 }
   /* if button is pressed, print "button 1 is pressed !" */
   while(1) {
      if (button1_pressed == 1) {
       button1_pressed = 0;
         printf("button 1 is pressed !\n");
      }
      /* sleep 100ms */
      aos_msleep(100);
   };
}
当button被按下后，串口会打印"button 1 is pressed !"
```

移植说明

新建hal_gpio_xxmcu.c和hal_gpio_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_gpio_xxmcu.c中实现所需要的hal函数，hal_gpio_xxmcu.h中放置相关宏定义。 参考platform/mcu/stm32l4xx/src/STM32L496G-Discovery/hal/hal_gpio_stm32l4.c

2.3.2. UART

对于不同底层驱动的uart操作实现，统一封装成本文所述uart hal接口。 hal相关头文件位于目录：include/aos/hal。hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/ 注意：另系统启动后printf输出对接时，一般都使用了一个uart串口，实际使用时需要注意避免冲突。参见：utility/libc/newlib_stub.c 中_write_r实现

API列表

hal_uart_init	初始化指定UART
hal_uart_send	从指定的UART发送数据
hal_uart_recv	从指定的UART接收数据
hal_uart_recv_II	从指定的UART中断方式接收数据
hal_uart_finalize	关闭指定UART

API详情

请参考include/aos/hal/uart.h

相关结数据结构

uart_dev_t

```
typedef struct {
  uint8_t      port;  /**< uart port */
  uart_config_t  config; /**< uart config */
  void       *priv;  /**< priv data */
} uart_dev_t;
```

uart_config_t

```
typedef struct {
  uint32_t          baud_rate;    /**< Uart baud rate */
  hal_uart_data_width_t  data_width;    /**< Uart data width */
  hal_uart_parity_t     parity;      /**< Uart parity check mode */
  hal_uart_stop_bits_t   stop_bits;    /**< Uart stop bit mode */
  hal_uart_flow_control_t flow_control;  /**< Uart flow control mode */
  hal_uart_mode_t       mode;        /**< Uart send/receive mode */
} uart_config_t;
```

hal_uart_data_width_t

```
typedef enum {
  DATA_WIDTH_5BIT,
  DATA_WIDTH_6BIT,
  DATA_WIDTH_7BIT,
  DATA_WIDTH_8BIT,
  DATA_WIDTH_9BIT
} hal_uart_data_width_t;
```

hal_uart_parity_t

```
typedef enum {
  NO_PARITY,    /**< No parity check */
  ODD_PARITY,    /**< Odd parity check */
  EVEN_PARITY    /**< Even parity check */
} hal_uart_parity_t;
```

hal_uart_stop_bits_t

```
typedef enum {
  STOP_BITS_1,
  STOP_BITS_2
} hal_uart_stop_bits_t;
```

hal_uart_flow_control_t

```
typedef enum {
  FLOW_CONTROL_DISABLED, /**< Flow control disabled */
  FLOW_CONTROL_CTS,    /**< Clear to send, yet to send data */
  FLOW_CONTROL_RTS,    /**< Require to send, yet to receive data */
  FLOW_CONTROL_CTS_RTS   /**< Both CTS and RTS flow control */
} hal_uart_flow_control_t;
```

hal_uart_mode_t

```
typedef enum {
  MODE_TX,     /**< Uart in send mode */
  MODE_RX,     /**< Uart in receive mode */
  MODE_TX_RX    /**< Uart in send and receive mode */
} hal_uart_mode_t;
```

hal_uart_init

初始化指定UART

函数原型

```
int32_t hal_uart_init(uart_dev_t *uart)
```

参数

uart_dev_t *uart	入参	UART设备描述，定义需要初始化的UART参数	用户自定义一个uart_dev_t结构体

返回值

返回成功或失败, 返回0表示UART初始化成功，非0表示失败

调用示例

```
/* define dev */
#define UART1_PORT_NUM  1
uart_dev_t uart1;
/* uart port set */
uart1.port = UART1_PORT_NUM;
/* uart attr config */
uart1.config.baud_rate    = 115200;
uart1.config.data_width   = DATA_WIDTH_8BIT;
uart1.config.parity       = NO_PARITY;
uart1.config.stop_bits    = STOP_BITS_1;
uart1.config.flow_control = FLOW_CONTROL_DISABLED;
uart1.config.mode         = MODE_TX_RX;
ret = hal_uart_init(&uart1);
```

hal_uart_send

从指定的UART发送数据

函数原型

```
int32_t hal_uart_send(uart_dev_t *uart, const void *data, uint32_t size, uint32_t timeout)
```

参数

uart_dev_t *uart	入参	UART设备描述句柄	使用hal_uart_init传入值
const void *data	入参	指向要发送数据的数据指针	char pdata[10]
uint32_t size	入参	要发送的数据字节数	10
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败, 返回0表示UART数据发送成功，非0表示失败

调用示例

```
#define UART_BUF_SIZE  10
#define UART_TX_TIMEOUT 50
char uart_data_buf[UART_BUF_SIZE] = {0};
ret = hal_uart_send(&uart1, uart_data_buf, UART_BUF_SIZE, UART_TX_TIMEOUT);
```

hal_uart_recv

从指定的UART接收数据，一般是阻塞等待数据接收

函数原型

```
int32_t hal_uart_recv(uart_dev_t *uart, void *data, uint32_t expect_size,uint32_t timeout)
```

参数

uart_dev_t *uart	入参	UART设备描述句柄	使用hal_uart_init传入值
void *data	入参	指向接收缓冲区的数据指针	char pdata[10]
uint32_t expect_size	入参	期望接收的数据字节数	10
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败，返回0表示成功接收expect_size个数据，非0表示失败

调用示例

```
#define UART_BUF_SIZE  10
char uart_data_buf[UART_BUF_SIZE] = {0};
ret = hal_uart_recv(&uart1, uart_data_buf, UART_BUF_SIZE, HAL_WAIT_FOREVER);
```

hal_uart_recv_II

从指定的UART中断方式接收数据，与hal_uart_recv不同的是，其不需要阻塞等待消息；中断触发后会自动唤醒其继续处理字符信息。

函数原型

```
int32_t hal_uart_recv_II(uart_dev_t *uart, void *data, uint32_t expect_size,uint32_t *recv_size, uint32_t timeout)
```

参数

uart_dev_t *uart	入参	UART设备描述句柄	使用hal_uart_init传入值
void *data	入参	指向接收缓冲区的数据指针	char pdata[10]
uint32_t expect_size	入参	期望接收的数据字节数	10
uint32_t *recv_size	出参	实际接收数据字节数	用户自定义int类型，传入地址
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败，返回0表示成功接收expect_size个数据，非0表示失败

调用示例

```
#define UART_BUF_SIZE  10
uint32_t plen = 0;
char uart_data_buf[UART_BUF_SIZE] = {0};
ret = hal_uart_recv_II(&uart1, uart_data_buf, UART_BUF_SIZE,&plen, HAL_WAIT_FOREVER);
```

hal_uart_finalize

关闭指定UART

函数原型

```
int32_t hal_uart_finalize(uart_dev_t *uart)
```

参数

uart_dev_t *uart	入参	UART设备描述句柄	使用hal_uart_init传入值

返回值

类型：int 返回成功或失败, 返回0表示UART关闭成功，非0表示失败。

调用示例

```
ret = hal_uart_finalize(&uart1);
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/uart.h"
```

使用示例

```
#include <aos/hal/uart.h>
#define UART1_PORT_NUM  1
#define UART_BUF_SIZE   10
#define UART_TX_TIMEOUT 10
#define UART_RX_TIMEOUT 10
/* define dev */
uart_dev_t uart1;
/* data buffer */
char uart_data_buf[UART_BUF_SIZE];
int application_start(int argc, char *argv[])
{
  int count   = 0;
  int ret     = -1;
  int i       = 0;
  int rx_size = 0;
  /* uart port set */
  uart1.port = UART1_PORT_NUM;
  /* uart attr config */
  uart1.config.baud_rate    = 115200;
  uart1.config.data_width   = DATA_WIDTH_8BIT;
  uart1.config.parity       = NO_PARITY;
  uart1.config.stop_bits    = STOP_BITS_1;
  uart1.config.flow_control = FLOW_CONTROL_DISABLED;
  uart1.config.mode         = MODE_TX_RX;
  /* init uart1 with the given settings */
  ret = hal_uart_init(&uart1);
  if (ret != 0) {
    printf("uart1 init error !\n");
  }
  /* init the tx buffer */
  for (i = 0; i < UART_BUF_SIZE; i++) {
    uart_data_buf[i] = i + 1;
  }
  /* send 0,1,2,3,4,5,6,7,8,9 by uart1 */
  ret = hal_uart_send(&uart1, uart_data_buf, UART_BUF_SIZE, UART_TX_TIMEOUT);
  if (ret == 0) {
    printf("uart1 data send succeed !\n");
  }
  /* scan uart1 every 100ms, if data received send it back */
  while(1) {
    ret = hal_uart_recv_II(&uart1, uart_data_buf, UART_BUF_SIZE,
                  &rx_size, UART_RX_TIMEOUT);
    if ((ret == 0) && (rx_size == UART_BUF_SIZE)) {
      printf("uart1 data received succeed !\n");
      ret = hal_uart_send(&uart1, uart_data_buf, rx_size, UART_TX_TIMEOUT);
      if (ret == 0) {
        printf("uart1 data send succeed !\n");
      }
    }
    /* sleep 100ms */
    aos_msleep(100);
  };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_uart_xxmcu.c和hal_uart_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_uart_xxmcu.c中实现所需要的hal函数，hal_uart_xxmcu.h中放置相关宏定义。 参考platform/mcu/stm32l4xx/src/STM32L496G-Discovery/hal/hal_uart_stm32l4.c

2.3.3. SPI

对于不同底层驱动的spi操作实现，统一封装成本文所述spi hal接口。 hal相关头文件位于目录：include/aos/hal。hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/。

API列表

hal_spi_init	初始化指定SPI端口
hal_spi_send	从指定的SPI端口发送数据
hal_spi_recv	从指定的SPI端口接收数据
hal_spi_send_recv	从指定的SPI端口发送并接收数据
hal_spi_finalize	关闭指定SPI端口

API详情

请参考include/aos/hal/spi.h

相关宏定义

```
#define HAL_SPI_MODE_MASTER 1 /* spi communication is master mode */
#define HAL_SPI_MODE_SLAVE  2 /* spi communication is slave mode */
```

相关结数据结构

spi_dev_t

```
typedef struct {
  uint8_t    port;  /* spi port */
  spi_config_t config; /* spi config */
  void     *priv;  /* priv data */
} spi_dev_t;
```

spi_config_t

```
typedef struct {
  uint32_t mode; /* spi communication mode */
  uint32_t freq; /* communication frequency Hz */
} spi_config_t;
```

hal_spi_init

初始化指定SPI端口

函数原型

```
int32_t hal_spi_init(spi_dev_t *spi)
```

参数

spi_dev_t *spi	入参	SPI设备描述，定义需要初始化的SPI参数	用户自定义一个spi_dev_t结构体

返回值

返回成功或失败，返回0表示SPI初始化成功，非0表示失败

调用示例

```
#define SPI1_PORT_NUM  1
/* define dev */
spi_dev_t spi1;
/* spi port set */
spi1.port = SPI1_PORT_NUM;
/* spi attr config */
spi1.config.mode  = HAL_SPI_MODE_MASTER;
spi1.config.freq = 30000000;
ret = hal_spi_init(&spi1);
```

hal_spi_send

从指定的SP端口发送数据

函数原型

```
int32_t hal_spi_send(spi_dev_t *spi, const uint8_t *data, uint16_t size, uint32_t timeout)
```

参数

spi_dev_t *spi	入参	SPI设备描述	使用hal_spi_init初始化时传入值
const uint8_t *data	入参	指向要发送数据的数据指针	char spi_data_buf[10]
uint16_t size	入参	要发送的数据字节数	10
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败，返回0表示SPI数据发送成功，非0表示失败

调用示例

```
#define SPI_BUF_SIZE  10
#define SPI_TX_TIMEOUT 50
/* data buffer */
char spi_data_buf[SPI_BUF_SIZE];
ret = hal_spi_send(&spi1, spi_data_buf, SPI_BUF_SIZE, SPI_TX_TIMEOUT);
```

hal_spi_recv

从指定的SPI端口发送并接收数据

函数原型

```
int32_t hal_spi_recv(spi_dev_t *spi, uint8_t *data, uint16_t size, uint32_t timeout)
```

参数

spi_dev_t *spi	入参	SPI设备描述	使用hal_spi_init初始化时传入值
const uint8_t *data	入参	指向接收缓冲区的数据指针	char spi_data_buf[10]
uint16_t size	入参	期望接收的数据字节数	10
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败, 返回0表示成功接收size个数据，非0表示失败

调用示例

```
#define SPI_BUF_SIZE  10
#define SPI_RX_TIMEOUT 50
/* data buffer */
char spi_data_buf[SPI_BUF_SIZE];
ret =hal_spi_recv(&spi1, spi_data_buf, SPI_BUF_SIZE, SPI_RX_TIMEOUT);
```

hal_spi_send_recv

从指定的SPI端口发送并接收数据

函数原型

```
int32_t hal_spi_send_recv(spi_dev_t *spi, uint8_t *tx_data, uint8_t *rx_data, uint16_t size, uint32_t timeout)
```

参数

spi_dev_t *spi	入参	SPI设备描述	使用hal_spi_init初始化时传入值
uint8_t *tx_data	入参	指向发送缓冲区的数据指针	char spi_send_buf[10]

uint8_t *rx_data	入参	指向接收缓冲区的数据指针	char spi_recv_buf[10]
uint16_t size	入参	要发送和接收数据字节数	10
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败, 返回0表示成功发送和接收size个数据，非0表示失败

调用示例

```
#define SPI_BUF_SIZE  10
#define SPI_RX_TIMEOUT 50
/* data buffer */
char spi_send_buf[SPI_BUF_SIZE] = {0};
char spi_recv_buf[SPI_BUF_SIZE] = {0};
ret =hal_spi_send_recv(&spi1, spi_send_buf, spi_recv_buf,SPI_BUF_SIZE, SPI_RX_TIMEOUT);
```

hal_spi_finalize

关闭指定SPI端口

函数原型

```
int32_t hal_spi_finalize(spi_dev_t *spi)
```

参数

spi_dev_t *spi	入参	SPI设备描述	使用hal_spi_init初始化时传入值

返回值

类型：int 返回成功或失败, 返回0表示SPI关闭成功，非0表示失败。

调用示例

```
ret =hal_spi_finalize(&spi1);
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/spi.h"
```

使用示例

```
#include <aos/hal/spi.h>
#define SPI1_PORT_NUM  1
#define SPI_BUF_SIZE  10
#define SPI_TX_TIMEOUT 10
#define SPI_RX_TIMEOUT 10
/* define dev */
spi_dev_t spi1;
/* data buffer */
char spi_data_buf[SPI_BUF_SIZE];
int application_start(int argc, char *argv[])
{
  int count  = 0;
  int ret    = -1;
  int i      = 0;
  int rx_size = 0;
  /* spi port set */
  spi1.port = SPI1_PORT_NUM;
  /* spi attr config */
  spi1.config.mode  = HAL_SPI_MODE_MASTER;
  spi1.config.freq = 30000000;
  /* init spi1 with the given settings */
  ret = hal_spi_init(&spi1);
  if (ret != 0) {
    printf("spi1 init error !\n");
  }
  /* init the tx buffer */
  for (i = 0; i < SPI_BUF_SIZE; i++) {
    spi_data_buf[i] = i + 1;
  }
  /* send 0,1,2,3,4,5,6,7,8,9 by spi1 */
  ret = hal_spi_send(&spi1, spi_data_buf, SPI_BUF_SIZE, SPI_TX_TIMEOUT);
  if (ret == 0) {
    printf("spi1 data send succeed !\n");
  }
  /* scan spi every 100ms to get the data */
  while(1) {
    ret = hal_spi_recv(&spi1, spi_data_buf, SPI_BUF_SIZE, SPI_RX_TIMEOUT);
    if (ret == 0) {
      printf("spi1 data received succeed !\n");
    }
    /* sleep 100ms */
    aos_msleep(100);
  };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_spi_xxmcu.c和hal_spi_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_spi_xxmcu.c中实现所需要的hal函数，hal_spi_xxmcu.h中放置相关宏定义。 参考platform/mcu/stm32l4xx/src/STM32L496G-Discovery/hal/hal_spi_stm32l4.c

2.3.4. I2C

对于不同底层驱动的I2C操作实现，统一封装成本文所述hal I2C接口。 hal相关头文件位于目录：include/aos/hal。 hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/。

API列表

hal_i2c_init	初始化指定I2C端口
hal_i2c_master_send	master模式下从指定的I2C端口发送数据
hal_i2c_master_recv	master模式下从指定的I2C端口接收数据
hal_i2c_slave_send	slave模式下从指定的I2C端口发送数据
hal_i2c_slave_recv	slave模式下从指定的I2C端口接收数据
hal_i2c_mem_write	mem模式（读写I2C存储器）下从指定的I2C端口发送数据
hal_i2c_mem_read	mem模式（读写I2C存储器）下从指定的I2C端口接收数据
hal_i2c_finalize	关闭指定I2C端口

API详情

请参考include/aos/hal/i2c.h

相关宏定义

```
#define I2C_MODE_MASTER 1 /* i2c communication is master mode */
#define I2C_MODE_SLAVE  2 /* i2c communication is slave mode */
#define I2C_MEM_ADDR_SIZE_8BIT  1 /* i2c memory address size 8bit */
#define I2C_MEM_ADDR_SIZE_16BIT 2 /* i2c memory address size 16bit */
/*
 * Specifies one of the standard I2C bus bit rates for I2C communication
 */
#define I2C_BUS_BIT_RATES_100K  100000
#define I2C_BUS_BIT_RATES_400K  400000
#define I2C_BUS_BIT_RATES_3400K 3400000
#define I2C_HAL_ADDRESS_WIDTH_7BIT  0
#define I2C_HAL_ADDRESS_WIDTH_10BIT 1
```

相关结数据结构

i2c_dev_t

```
typedef struct {
  uint8_t    port;  /* i2c port */
  i2c_config_t config; /* i2c config */
  void      *priv;  /* priv data */
} i2c_dev_t;
```

i2c_config_t

```
typedef struct {
  uint32_t address_width;
  uint32_t freq;
  uint8_t  mode;
  uint16_t dev_addr;
} i2c_config_t;
```

hal_i2c_init

初始化指定I2C端口

函数原型

```
int32_t hal_i2c_init(i2c_dev_t *i2c)
```

参数

i2c_dev_t *i2c	入参	I2C设备描述，定义需要初始化的I2C参数	用户自定义一个i2c_dev_t结构体

返回值

返回成功或失败，返回0表示I2C初始化成功，非0表示失败

调用示例

```
#define I2C1_PORT_NUM  1
#define I2C2_PORT_NUM  2
#define I2C2_SLAVE_ADDR  0x50
/* define dev master */
i2c_dev_t i2c_dev_master;
i2c_dev_t i2c_dev_slave;
/* i2c port set */
i2c_dev_master.port = I2C1_PORT_NUM;
/* i2c attr config */
i2c_dev_master.config.mode       = I2C_MODE_MASTER;
i2c_dev_master.config.freq       = I2C_BUS_BIT_RATES_3400K;
i2c_dev_master.config.address_width = I2C_HAL_ADDRESS_WIDTH_7BIT;
i2c_dev_slave.port = I2C2_PORT_NUM;
/* i2c attr config */
i2c_dev_slave.config.mode       = I2C_MODE_SLAVE;
i2c_dev_slave.config.freq       = I2C_BUS_BIT_RATES_3400K;
i2c_dev_slave.config.address_width = I2C_HAL_ADDRESS_WIDTH_7BIT;
i2c_dev_slave.config.dev_addr   = I2C2_SLAVE_ADDR;
/* init master i2c with the given settings */
ret = hal_i2c_init(&i2c_dev_master);
/* init slave i2c with the given settings */
ret = hal_i2c_init(&i2c_dev_slave);
```

hal_i2c_master_send

master模式下从指定的I2C端口发送数据

函数原型

```
int32_t hal_i2c_master_send(i2c_dev_t *i2c, uint16_t dev_addr, const uint8_t *data, uint16_t size, uint32_t timeout)
```

参数

i2c_dev_t *i2c	入参	I2C设备描述	使用hal_i2c_init传入参数
uint16_t dev_addr	入参	目标设备地址	0x50
const uint8_t *data	入参	指向发送缓冲区的数据指针	char pdata_send[10]
uint16_t size	入参	要发送的数据字节数	10
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败，返回0表示I2C数据发送成功，非0表示失败

调用示例

```
char pdata_send[10] = {0};
#define I2C2_SLAVE_ADDR  0x50
ret = hal_i2c_master_send(&i2c_dev_master,I2C2_SLAVE_ADDR,pdata_send,10,50);
```

hal_i2c_master_recv

master模式下从指定的I2C端口接收数据

函数原型

```
int32_t hal_i2c_master_recv(i2c_dev_t *i2c, uint16_t dev_addr, uint8_t *data,uint16_t size, uint32_t timeout)
```

参数

i2c_dev_t *i2c	入参	I2C设备描述	使用hal_i2c_init传入参数
uint16_t dev_addr	入参	目标设备地址	0x50
uint8_t *data	入参	指向接收缓冲区的数据指针	char pdata_recv[10]
uint16_t size	入参	期望接收的数据字节数	10
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败，返回0表示成功接收size个数据，非0表示失败

调用示例

```
char pdata_recv[10] = {0};
#define I2C2_SLAVE_ADDR  0x50
ret = hal_i2c_master_recv(&i2c_dev_master,I2C2_SLAVE_ADDR,pdata_recv,10,50);
```

hal_i2c_slave_send

slave模式下从指定的I2C端口发送数据

函数原型

```
int32_t hal_i2c_slave_send(i2c_dev_t *i2c, const uint8_t *data, uint16_t size, uint32_t timeout)
```

参数

i2c_dev_t *i2c	入参	I2C设备描述	使用hal_i2c_init传入参数
const uint8_t *data	入参	指向发送缓冲区的数据指针	char pdata_send[10]
uint16_t size	入参	要发送的数据字节数	10
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败, 返回0表示成功发送size个数据，非0表示失败

调用示例

```
char pdata_send[10] = {0};
ret = hal_i2c_slave_send(&i2c_dev_slave,pdata_send,10,50);
```

hal_i2c_slave_recv

slave模式下从指定的I2C端口接收数据

函数原型

```
int32_t hal_i2c_slave_recv(i2c_dev_t *i2c, uint8_t *data, uint16_t size, uint32_t timeout)
```

参数

i2c_dev_t *i2c	入参	I2C设备描述	使用hal_i2c_init传入参数
uint8_t *data	入参	指向要接收数据的数据指针	char pdata_recv[10]
uint16_t size	入参	要接收的数据字节数	10
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败，返回0表示成功接收size个数据，非0表示失败

调用示例

```
char pdata_recv[10] = {0};
ret = hal_i2c_slave_recv(&i2c_dev_slave,pdata_recv,10,50);
```

hal_i2c_mem_write

向指定的设备内存写数据

函数原型

```
int32_t hal_i2c_mem_write(i2c_dev_t *i2c, uint16_t dev_addr, uint16_t mem_addr, uint16_t mem_addr_size, const uint8_t *data, uint16_t size, uint32_t timeout)
```

参数

i2c_dev_t *i2c	入参	I2C设备描述	使用hal_i2c_init传入参数
uint16_t dev_addr	入参	目标设备地址	0x50
uint16_t mem_addr	入参	内部内存地址	0x20
uint16_t mem_addr_size	入参	内部内存地址大小	1
const uint8_t *data	入参	指向要发送数据的数据指针	char pdata[10]
uint16_t size	入参	要发送的数据字节数	1
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败，返回0表示成功发送size个数据，非0表示失败

调用示例

```
char pdata[10] = {0};
ret = hal_i2c_mem_write(&i2c_dev_master,0x50,0x20,1,pdata,1,50);
```

hal_i2c_mem_read

从指定的设备内存读数据

函数原型

```
int32_t hal_i2c_mem_read(i2c_dev_t *i2c, uint16_t dev_addr, uint16_t mem_addr, uint16_t mem_addr_size, uint8_t *data, uint16_t size, uint32_t timeout)
```

参数

i2c_dev_t *i2c	入参	I2C设备描述	使用hal_i2c_init传入参数
uint16_t dev_addr	入参	目标设备地址	0x50
uint16_t mem_addr	入参	内部内存地址	0x20
uint16_t mem_addr_size	入参	内部内存地址大小	1
uint8_t *data	入参	指向接收缓冲区的数据指针	char pdata[10]
uint16_t size	入参	要接收的数据字节数	1
uint32_t timeout	入参	超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER	50

返回值

返回成功或失败, 返回0表示成功接收size个数据，非0表示失败

调用示例

```
char pdata[10] = {0};
ret = hal_i2c_mem_read(&i2c_dev_master,0x50,0x20,1,pdata,1,50);
```

hal_i2c_finalize

关闭指定I2C端口

函数原型

```
int32_t hal_i2c_finalize(i2c_dev_t *i2c)
```

参数

i2c_dev_t *i2c	入参	I2C设备描述	使用hal_i2c_init传入参数

返回值

类型：int 返回成功或失败, 返回0表示I2C关闭成功，非0表示失败。

调用示例

```
ret = hal_i2c_finalize(&i2c_dev_master);
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/i2c.h"
```

使用示例

```
#include <aos/hal/i2c.h>
#define I2C1_PORT_NUM  1
#define I2C_BUF_SIZE  10
#define I2C_TX_TIMEOUT 10
#define I2C_RX_TIMEOUT 10
#define I2C_DEV_ADDR     0x30f
#define I2C_DEV_ADDR_WIDTH 8
/* define dev */
i2c_dev_t i2c1;
/* data buffer */
char i2c_data_buf[I2C_BUF_SIZE];
int application_start(int argc, char *argv[])
{
  int count   = 0;
  int ret    = -1;
  int i     = 0;
  int rx_size = 0;
  /* i2c port set */
  i2c1.port = I2C1_PORT_NUM;
  /* i2c attr config */
  i2c1.config.mode       = I2C_MODE_MASTER;
  i2c1.config.freq       = 30000000;
  i2c1.config.address_width = I2C_DEV_ADDR_WIDTH;
  i2c1.config.dev_addr     = I2C_DEV_ADDR;
  /* init i2c1 with the given settings */
  ret = hal_i2c_init(&i2c1);
  if (ret != 0) {
    printf("i2c1 init error !\n");
  }
  /* init the tx buffer */
  for (i = 0; i < I2C_BUF_SIZE; i++) {
    i2c_data_buf[i] = i + 1;
  }
  /* send 0,1,2,3,4,5,6,7,8,9 by i2c1 */
  ret = hal_i2c_master_send(&i2c1, I2C_DEV_ADDR, i2c_data_buf,
                I2C_BUF_SIZE, I2C_TX_TIMEOUT);
  if (ret == 0) {
    printf("i2c1 data send succeed !\n");
  }
  ret = hal_i2c_master_recv(&i2c1, I2C_DEV_ADDR, i2c_data_buf,
                I2C_BUF_SIZE, I2C_RX_TIMEOUT);
  if (ret == 0) {
    printf("i2c1 data received succeed !\n");
  }
  while(1) {
    printf("AliOS Things is working !\n");
    /* sleep 1000ms */
    aos_msleep(1000);
  };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_i2c_xxmcu.c和hal_i2c_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_i2c_xxmcu.c中实现所需要的hal函数，hal_i2c_xxmcu.h中放置相关宏定义。 参考platform/mcu/stm32l4xx/src/STM32L496G-Discovery/hal/hal_i2c_stm32l4.c

2.3.5. FLASH

对于不同底层驱动的flash操作实现，统一封装成本文所述hal flash接口。 hal相关头文件位于目录：include/aos/hal。hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/。

API列表

hal_flash_info_get	获取指定区域的FLASH信息
hal_flash_erase	擦除FLASH的指定区域
hal_flash_write	写FLASH的指定区域
hal_flash_erase_write	先擦除再写FLASH的指定区域
hal_flash_read	读FLASH的指定区域
hal_flash_enable_secure	使能加密FLASH的指定区域
hal_flash_dis_secure	关闭加密FLASH的指定区域
hal_flash_addr2offset	将物理地址转换为分区号和偏移

API详情

相关宏定义

```
#define PAR_OPT_READ_POS （0）
#define PAR_OPT_WRITE_POS (1)
#define PAR_OPT_READ_MASK (0x1u << PAR_OPT_READ_POS)
#define PAR_OPT_WRITE_MASK (0x1u << PAR_OPT_WRITE_POS)
#define PAR_OPT_READ_DIS （0x0u << PAR_OPT_READ_POS)
#define PAR_OPT_READ_EN （0x1u << PAR_OPT_READ_POS)
#define PAR_OPT_WRITE_DIS (0x0u << PAR_OPT_WRITE_POS)
#define PAR_OPT_WRITE_EN （0x1u << PAR_OPT_WRITE_POS)
```

相关数据结构

hal_logic_partition_t

```
typedef struct {
  hal_flash_t partition_owner;
  const char *partition_description;
  uint32_t  partition_start_addr;
  uint32_t  partition_length;
  uint32_t  partition_options;
} hal_logic_partition_t;
```

hal_flash_t

```
typedef enum {
  HAL_FLASH_EMBEDDED,
  HAL_FLASH_SPI,
  HAL_FLASH_QSPI,
  HAL_FLASH_MAX,
  HAL_FLASH_NONE,
} hal_flash_t;
```

hal_partition_t

```
typedef enum {
  HAL_PARTITION_ERROR = -1,
  HAL_PARTITION_BOOTLOADER,
  HAL_PARTITION_APPLICATION,
  HAL_PARTITION_ATE,
  HAL_PARTITION_OTA_TEMP,
  HAL_PARTITION_RF_FIRMWARE,
  HAL_PARTITION_PARAMETER_1,
  HAL_PARTITION_PARAMETER_2,
  HAL_PARTITION_PARAMETER_3,
  HAL_PARTITION_PARAMETER_4,
  HAL_PARTITION_BT_FIRMWARE,
  HAL_PARTITION_SPIFFS,
  HAL_PARTITION_CUSTOM_1,
  HAL_PARTITION_CUSTOM_2,
  HAL_PARTITION_RECOVERY,
  HAL_PARTITION_RECOVERY_BACK_PARA,
  HAL_PARTITION_MAX,
  HAL_PARTITION_NONE,
} hal_partition_t;
```

hal_flash_info_get

获取指定区域的FLASH信息

函数原型

```
int32_t hal_flash_info_get(hal_partition_t in_partition, hal_logic_partition_t *partition)
```

参数

hal_partition_t in_partition	入参	FLASH分区号	HAL_PARTITION_APPLICATION
hal_logic_partition_t *partition	出参	分区信息	用户自定义一个hal_logic_partition_t结构体

返回值

成功则返回0，非0表示失败

调用示例

```
hal_logic_partition_t partition_info = {0};
ret = hal_flash_info_get(HAL_PARTITION_APPLICATION,&partition_info);
```

hal_flash_erase

擦除FLASH的指定区域

函数原型

```
int32_t hal_flash_erase(hal_partition_t pno, uint32_t off_set, uint32_t size)
```

参数

hal_partition_t pno	入参	FLASH分区号	HAL_PARTITION_APPLICATION
uint32_t off_set	入参	偏移量	0
uint32_t size	入参	要擦除的字节数	512

返回值

返回成功或失败, 返回0表示擦除成功，非0表示失败

调用示例

```
ret = hal_flash_erase(HAL_PARTITION_APPLICATION,0,512);
```

hal_flash_write

写FLASH的指定区域

函数原型

```
int32_t hal_flash_write(hal_partition_t pno, uint32_t *poff, const void *buf , uint32_t buf_size)
```

参数

hal_partition_t pno	入参	FLASH分区号	HAL_PARTITION_APPLICATION
uint32_t *poff	入参+出参	偏移量，写入后其值会刷新为写尾部	
const void *buf	入参	指向要写入数据的指针	char buf[512]
uint32_t buf_size	入参	要写入的字节数	512

返回值

返回成功或失败, 返回0表示写入成功，非0表示失败

调用示例

```
uint32_t off = 0;
char buf[512] = {0};
ret = hal_flash_write(HAL_PARTITION_APPLICATION,&off,buf,512);
```

hal_flash_erase_write

先擦除再写FLASH的指定区域

函数原型

```
int32_t hal_flash_erase_write(hal_partition_t in_partition, uint32_t *off_set, const void *in_buf, uint32_t in_buf_len);
```

参数

hal_partition_t in_partition	入参	FLASH分区号	HAL_PARTITION_APPLICATION
uint32_t *off_set	入参+出参	偏移量，写入后其值会更新为写尾部	偏移量
const void *in_buf	入参	指向要写入数据的指针	char buf[512]
uint32_t in_buf_len	入参	要擦除和写入的字节数	512

返回值

返回成功或失败, 返回0表示擦除写入成功，非0表示失败

调用示例

```
uint32_t off = 0;
char buf[512] = {0};
ret = hal_flash_erase_write(HAL_PARTITION_APPLICATION,&off,buf,512);
```

hal_flash_read

读取FLASH的指定区域

函数原型

```
int32_t hal_flash_read(hal_partition_t in_partition, uint32_t *off_set, void *out_buf, uint32_t in_buf_len);
```

参数

hal_partition_t in_partition	入参	FLASH分区号	HAL_PARTITION_APPLICATION
uint32_t *off_set	入参+出参	偏移量，读取后其值会更新为读取的尾部偏移位置	
void *out_buf	入参	数据缓冲区地址	char buf[512]

uint32_t in_buf_len	入参	要读取的字节数	512

返回值

返回成功或失败，返回0表示读取成功，非0表示失败

调用示例

```
uint32_t off = 0;
char buf[512] = {0};
ret = hal_flash_read(HAL_PARTITION_APPLICATION,&off,buf,512);
```

hal_flash_enable_secure

使能加密FLASH的指定区域

函数原型

```
int32_t hal_flash_enable_secure(hal_partition_t partition, uint32_t off_set, uint32_t size);
```

参数

hal_partition_t in_partition	入参	FLASH分区号	HAL_PARTITION_APPLICATION
uint32_t off_set	入参	偏移量	0
uint32_t size	入参	使能区域字节数	512

返回值

返回成功或失败，返回0表示使能成功，非0表示失败

调用示例

```
ret = hal_flash_enable_secure(HAL_PARTITION_APPLICATION,0,512);
```

hal_flash_dis_secure

关闭加密FLASH的指定区域

函数原型

```
int32_t hal_flash_dis_secure(hal_partition_t partition, uint32_t off_set, uint32_t size);
```

参数

hal_partition_t in_partition	入参	FLASH分区号	HAL_PARTITION_APPLICATION
uint32_t off_set	入参	偏移量	0
uint32_t size	入参	去使能加密区域字节数	512

返回值

返回成功或失败, 返回0表示关闭成功，非0表示失败

调用示例

```
ret = hal_flash_dis_secure(HAL_PARTITION_APPLICATION,0,512);
```

hal_flash_addr2offset

将物理地址转换为分区号和偏移。该偏移与hal_partitions分区表配置相关。

函数原型

```
int32_t hal_flash_addr2offset(hal_partition_t *in_partition, uint32_t *off_set, uint32_t addr);
```

参数

hal_partition_t *in_partition	出参	获取到的FLASH分区号	
uint32_t *off_set	出参	获取到的该地址在分区内偏移	
uint32_t addr	入参	输入需要查询地址	0x400000

返回值

返回成功或失败, 返回0表示转换成功，非0表示失败

调用示例

```
hal_partition_t in_partition;
uint32_t off_set=0;
ret = hal_flash_addr2offset(&in_partition,&off_set,0x400000);
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/flash.h"
```

2.3.6. PWM

对于不同底层驱动的pwm操作实现，统一封装成本文所述hal pwm接口。 hal相关头文件位于目录：include/aos/hal。hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/。

API列表

hal_pwm_init	初始化指定PWM
hal_pwm_start	开始输出指定PWM
hal_pwm_stop	停止输出指定PWM
hal_pwm_para_chg	修改指定PWM参数
hal_pwm_finalize	关闭指定PWM

API详情

请参考include/aos/hal/pwm.h

相关结数据结构

pwm_dev_t

```
typedef struct {
  uint8_t    port;  /* pwm port */
  pwm_config_t config; /* pwm config */
  void     *priv;  /* priv data */
} pwm_dev_t;
```

pwm_config_t

```
typedef struct {
  float   duty_cycle; /* the pwm duty_cycle */
  uint32_t freq;    /* the pwm freq */
} pwm_config_t;
```

hal_pwm_init

初始化指定PWM

函数原型

```
int32_t hal_pwm_init(pwm_dev_t *pwm);
```

参数

pwm_dev_t *pwm	入参	PWM设备描述，定义需要初始化的PWM参数	用户自定义一个pwm_dev_t结构体

返回值

返回成功或失败, 返回0表示PWM初始化成功，非0表示失败

调用示例

```
#define PWM1_PORT_NUM 1
/* define dev */
pwm_dev_t pwm1;
/* pwm port set */
pwm1.port = PWM1_PORT_NUM;
/* pwm attr config */
pwm1.config.duty_cycle = 0.5f; /* 1s */
pwm1.config.freq     = 300000; /* 1s */
/* init pwm1 with the given settings */
ret = hal_pwm_init(&pwm1);
```

hal_pwm_start

开始输出指定PWM

函数原型

```
int32_t hal_pwm_start(pwm_dev_t *pwm);
```

参数

pwm_dev_t *pwm	入参	PWM设备描述	使用hal_pwm_init时传入pwm_dev_t结构体

返回值

返回成功或失败, 返回0表示PWM开始输出成功，非0表示失败

调用示例

```
ret = hal_pwm_start(&pwm1);
```

hal_pwm_stop

停止输出指定PWM

函数原型

```
int32_t hal_pwm_stop(pwm_dev_t *pwm);
```

参数

pwm_dev_t *pwm	入参	PWM设备描述	使用hal_pwm_init时传入pwm_dev_t结构体

返回值

返回成功或失败, 返回0表示PWM停止输出成功，非0表示失败

调用示例

```
ret = hal_pwm_stop(&pwm1);
```

hal_pwm_para_chg

修改指定PWM参数

函数原型

```
int32_t hal_pwm_para_chg(pwm_dev_t *pwm, pwm_config_t para);
```

参数

pwm_dev_t *pwm	入参	PWM设备描述	使用hal_pwm_init时传入pwm_dev_t结构体
pwm_config_t para	入参	新配置参数	{0.25f,300000}

返回值

返回成功或失败，返回0表示PWM参数修改成功，非0表示失败

调用示例

```
pwm_config_t para = {0.25f,300000};
ret = hal_pwm_para_chg(&pwm1,para);
```

hal_pwm_finalize

关闭指定PWM

函数原型

```
int32_t hal_pwm_finalize(pwm_dev_t *pwm);
```

参数

pwm_dev_t *pwm	入参	PWM设备描述	使用hal_pwm_init时传入pwm_dev_t结构体

返回值

返回成功或失败，返回0表示PWM关闭成功，非0表示失败

调用示例

```
ret = hal_pwm_finalize(&pwm1);
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/pwm.h"
```

使用示例

```
#include <aos/hal/pwm.h>
#define PWM1_PORT_NUM 1
/* define dev */
pwm_dev_t pwm1;
int application_start(int argc, char *argv[])
{
  int ret = -1;
  pwm_config_t pwm_cfg;
  static int count = 0;
  /* pwm port set */
  pwm1.port = PWM1_PORT_NUM;
  /* pwm attr config */
  pwm1.config.duty_cycle = 0.5f; /* 1s */
  pwm1.config.freq     = 300000; /* 1s */
  /* init pwm1 with the given settings */
  ret = hal_pwm_init(&pwm1);
  if (ret != 0) {
    printf("pwm1 init error !\n");
  }
  /* start pwm1 */
  ret = hal_pwm_start(&pwm1);
  if (ret != 0) {
    printf("pwm1 start error !\n");
  }
  while(1) {
    /* change the duty cycle to 30% */
    if (count == 5) {
      memset(&pwm_cfg, 0, sizeof(pwm_config_t));
      pwm_cfg.duty_cycle = 0.3f;
      ret = hal_pwm_para_chg(&pwm1, pwm_cfg);
      if (ret != 0) {
        printf("pwm1 para change error !\n");
      }
    }
    /* stop and finalize pwm1 */
    if (count == 20) {
      hal_pwm_stop(&pwm1);
      hal_pwm_finalize(&pwm1);
    }
    /* sleep 1000ms */
    aos_msleep(1000);
    count++;
  };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_pwm_xxmcu.c和hal_pwm_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_pwm_xxmcu.c中实现所需要的hal函数，hal_pwm_xxmcu.h中放置相关宏定义。 参考platform/mcu/stm32l4xx/src/STM32L496G-Discovery/hal/hal_pwm_stm32l4.c

2.3.7. TIMER

对于不同底层驱动的timer操作实现，统一封装成本文所述hal timer接口。 hal相关头文件位于目录：include/aos/hal。hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/。

API列表

hal_timer_init	初始化指定TIMER
hal_timer_start	启动指定的TIMER
hal_timer_stop	停止指定的TIMER
hal_timer_para_chg	改变指定TIMER的参数
hal_timer_finalize	关闭指定TIMER

API详情

请参考include/aos/hal/timer.h

相关宏定义

```
#define TIMER_RELOAD_AUTO  1 /* timer reload automatic */
#define TIMER_RELOAD_MANU  2 /* timer reload manual */
```

相关结数据结构

timer_dev_t

```
typedef struct {
  int8_t      port;  /* timer port */
  timer_config_t config; /* timer config */
  void       *priv;  /* priv data */
} timer_dev_t;
```

timer_config_t

```
typedef struct {
  uint32_t     period; /* us */
  uint8_t      reload_mode;
  hal_timer_cb_t cb;
  void       *arg;
} timer_config_t;
```

hal_timer_cb_t

```
typedef void (*hal_timer_cb_t)(void *arg);
```

hal_timer_init

初始化指定TIMER

函数原型

```
int32_t hal_timer_init(timer_dev_t *tim);
```

参数

timer_dev_t *tim	入参	TIMER设备描述，定义需要初始化的TIMER参数	用户自定义一个timer_dev_t结构体

返回值

返回成功或失败，返回0表示TIMER初始化成功，非0表示失败

调用示例

```
#define TIMER1_PORT_NUM 1
/* define dev */
timer_dev_t timer1;
/* timer port set */
timer1.port = TIMER1_PORT_NUM;
/* timer attr config */
timer1.config.period      = 1000000; /* 1s */
timer1.config.reload_mode   = TIMER_RELOAD_AUTO;
timer1.config.cb          = timer_handler;
/* init timer1 with the given settings */
ret = hal_timer_init(&timer1);
```

hal_timer_start

启动指定的TIMER

函数原型

```
int32_t hal_timer_start(timer_dev_t *tim);
```

参数

timer_dev_t *tim	入参	TIMER设备描述	使用hal_timer_init时传入timer_dev_t结构体

返回值

返回成功或失败，返回0表示TIMER启动成功，非0表示失败

调用示例

```
ret = hal_timer_start(&timer1);
```

hal_timer_stop

停止指定的TIMER

函数原型

```
int32_t hal_timer_stop(timer_dev_t *tim);
```

参数

timer_dev_t *tim	入参	TIMER设备描述	使用hal_timer_init时传入timer_dev_t结构体

返回值

返回成功或失败，返回0表示TIMER停止成功，非0表示失败

调用示例

```
ret = hal_timer_stop(&timer1);
```

hal_timer_para_chg

改变指定TIMER的参数

函数原型

```
int32_t hal_timer_para_chg(timer_dev_t *tim, timer_config_t para);
```

参数

timer_dev_t *tim	入参	TIMER设备描述	使用hal_timer_init时传入timer_dev_t结构体
timer_config_t para	入参	新TIMER配置信息	{2000000}

返回值

返回成功或失败，返回0表示TIMER参数改变成功，非0表示失败

调用示例

```
timer_config_t timer_cfg;
memset(&timer_cfg, 0, sizeof(timer_config_t));
timer_cfg.period = 2000000;
ret = hal_timer_para_chg(&timer1, timer_cfg);
```

hal_timer_finalize

关闭指定TIMER

函数原型

```
int32_t hal_timer_finalize(timer_dev_t *tim);
```

参数

timer_dev_t *tim	入参	TIMER设备描述	使用hal_timer_init时传入timer_dev_t结构体

返回值

返回成功或失败，返回0表示TIMER关闭成功，非0表示失败

调用示例

```
ret = hal_timer_finalize(&timer1);
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/timer.h"
```

使用示例

```
#include <aos/hal/timer.h>
#define TIMER1_PORT_NUM 1
/* define dev */
timer_dev_t timer1;
void timer_handler(void *arg)
{
  static int timer_cnt = 0;
  printf("timer_handler: %d times !\n", timer_cnt++);
}
int application_start(int argc, char *argv[])
{
  int ret = -1;
  timer_config_t timer_cfg;
  static int count = 0;
  /* timer port set */
  timer1.port = TIMER1_PORT_NUM;
  /* timer attr config */
  timer1.config.period      = 1000000; /* 1s */
  timer1.config.reload_mode   = TIMER_RELOAD_AUTO;
  timer1.config.cb          = timer_handler;
  /* init timer1 with the given settings */
  ret = hal_timer_init(&timer1);
  if (ret != 0) {
    printf("timer1 init error !\n");
  }
  /* start timer1 */
  ret = hal_timer_start(&timer1);
  if (ret != 0) {
    printf("timer1 start error !\n");
  }
  while(1) {
    /* change the period to 2s */
    if (count == 5) {
      memset(&timer_cfg, 0, sizeof(timer_config_t));
      timer_cfg.period = 2000000;
      ret = hal_timer_para_chg(&timer1, timer_cfg);
      if (ret != 0) {
        printf("timer1 para change error !\n");
      }
    }
    /* stop and finalize timer1 */
    if (count == 20) {
      hal_timer_stop(&timer1);
      hal_timer_finalize(&timer1);
    }
    /* sleep 1000ms */
    aos_msleep(1000);
    count++;
  };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_timer_xxmcu.c和hal_timer_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_timer_xxmcu.c中实现所需要的hal函数，hal_timer_xxmcu.h中放置相关宏定义。 参考platform/mcu/stm32l4xx/src/STM32L496G-Discovery/hal/hal_timer_stm32l4.c

2.3.8. WDG

对于不同底层驱动的wdg操作实现，统一封装成本文所述hal wdg接口。 hal相关头文件位于目录：include/aos/hal。hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/。

API列表

hal_wdg_init	初始化指定看门狗
hal_wdg_reload	重载指定看门狗，喂狗
hal_wdg_finalize	关闭指定看门狗

API详情

请参考include/aos/hal/wdg.h

相关结数据结构

wdg_dev_t

```
typedef struct {
  uint8_t    port;  /* wdg port */
  wdg_config_t config; /* wdg config */
  void     *priv;  /* priv data */
} wdg_dev_t;
```

wdg_config_t

```
typedef struct {
  uint32_t timeout; /* Watchdag timeout ms */
} wdg_config_t;
```

hal_wdg_init

初始化指定看门狗

函数原型

```
int32_t hal_wdg_init(wdg_dev_t *wdg);
```

参数

wdg_dev_t *wdg	入参	看门狗设备描述	用户自定义一个wdg_dev_t结构体

返回值

返回成功或失败，返回0表示看门狗初始化成功，非0表示失败

调用示例

```
wdg_dev_t wdg1;
/* wdg port set */
wdg1.port = WDG1_PORT_NUM;
/* set reload time to 1000ms */
wdg1.config.timeout = 1000; /* 1000ms */
ret = hal_wdg_init(&wdg1);
```

hal_wdg_reload

重载指定看门狗，喂狗

函数原型

```
void hal_wdg_reload(wdg_dev_t *wdg);
```

参数

wdg_dev_t *wdg	入参	看门狗设备描述	使用hal_wdg_init时传入wdg_dev_t结构体

返回值

无。

调用示例

```
hal_wdg_reload(&wdg1);
```

hal_wdg_finalize

关闭指定看门狗

函数原型

```
int32_t hal_wdg_finalize(wdg_dev_t *wdg);
```

参数

wdg_dev_t *wdg	入参	看门狗设备描述	使用hal_wdg_init时传入wdg_dev_t结构体

返回值

返回成功或失败, 返回0表示看门狗关闭成功，非0表示失败

调用示例

```
ret = hal_wdg_finalize(&wdg1);
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/wdg.h"
```

使用示例

```
#include <aos/hal/wdg.h>
#define WDG1_PORT_NUM 1
/* define dev */
wdg_dev_t wdg1;
int application_start(int argc, char *argv[])
{
  int ret = -1;
  static int count = 0;
  /* wdg port set */
  wdg1.port = WDG1_PORT_NUM;
  /* set reload time to 1000ms */
  wdg1.config.timeout = 1000; /* 1000ms */
  /* init wdg1 with the given settings */
  ret = hal_wdg_init(&wdg1);
  if (ret != 0) {
    printf("wdg1 init error !\n");
  }
  while(1) {
   /* clear wdg about every 500ms */
    hal_wdg_reload(&wdg1);
    /* finalize wdg1 */
    if (count == 20) {
      hal_wdg_finalize(&wdg1);
    }
    /* sleep 500ms */
    aos_msleep(500);
    count++;
  };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_wdg_xxmcu.c和hal_wdg_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_wdg_xxmcu.c中实现所需要的hal函数，hal_wdg_xxmcu.h中放置相关宏定义。 参考platform/mcu/stm32l4xx/src/STM32L496G-Discovery/hal/hal_wdg_stm32l4.c

2.3.9. RTC

对于不同底层驱动的rtc操作实现，统一封装成本文所述hal rtc接口。 hal相关头文件位于目录：include/aos/hal。hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/。

API列表

hal_rtc_init	初始化指定RTC
hal_rtc_get_time	获取指定RTC时间
hal_rtc_set_time	设置指定RTC时间
hal_rtc_finalize	关闭指定RTC

API详情

请参考include/aos/hal/rtc.h

相关宏

```
#define HAL_RTC_FORMAT_DEC 1   /*  Use Binary-Coded */
#define HAL_RTC_FORMAT_BCD 2   /* Use Decimal-Coded */
```

相关数据结构

rtc_dev_t

```
typedef struct {
  uint8_t     port;     /* rtc port */
  rtc_config_t config; /* rtc config */
  void       *priv;     /* priv data */
} rtc_dev_t;
```

rtc_config_t

```
typedef struct {
  uint8_t  format; /* time formart DEC or BCD */
} rtc_config_t;
```

rtc_time_t

```
typedef struct {
  uint8_t sec;    /* DEC format:value range from 0 to 59, BCD format:value range from 0x00 to 0x59 */
  uint8_t min;    /* DEC format:value range from 0 to 59, BCD format:value range from 0x00 to 0x59 */
  uint8_t hr;     /* DEC format:value range from 0 to 23, BCD format:value range from 0x00 to 0x23 */
  uint8_t weekday; /* DEC format:value range from 1 to  7, BCD format:value range from 0x01 to 0x07 */
  uint8_t date;   /* DEC format:value range from 1 to 31, BCD format:value range from 0x01 to 0x31 */
  uint8_t month;  /* DEC format:value range from 1 to 12, BCD format:value range from 0x01 to 0x12 */
  uint8_t year;   /* DEC format:value range from 0 to 99, BCD format:value range from 0x00 to 0x99 */
} rtc_time_t;
```

hal_rtc_init

初始化指定RTC

函数原型

```
int hal_rtc_init(rtc_dev_t *rtc);
```

参数

rtc_dev_t *rtc	入参	初始化指定RTC	用户自定义一个rtc_dev_t结构体

返回值

返回成功或失败，返回0表示RTC初始化成功，非0表示失败

调用示例

```
#define RTC1_PORT_NUM 1
rtc_dev_t rtc1;
/* rtc port set */
rtc1.port = RTC1_PORT_NUM;
/* set to DEC format */
rtc1.config.format = HAL_RTC_FORMAT_DEC;
ret = hal_rtc_init(&rtc1);
```

hal_rtc_get_time

获取指定RTC时间

函数原型

```
int32_t hal_rtc_get_time(rtc_dev_t *rtc, rtc_time_t *time);
```

参数

rtc_dev_t *rtc	入参	RTC设备描述	使用hal_rtc_init时传入rtc_dev_t结构体
rtc_time_t *time	出参	要获取的时间	用户创建rtc_time_t结构体传入指针

返回值

返回成功或失败，返回0表示RTC时间获取成功，非0表示失败

调用示例

```
rtc_time_t  time_buf;
memset(&time_buf, 0, sizeof(rtc_time_t));
ret = hal_rtc_get_time(&rtc1, &time_buf);
```

hal_rtc_set_time

设置指定RTC时间

函数原型

```
int32_t hal_rtc_set_time(rtc_dev_t *rtc, const rtc_time_t *time);
```

参数

rtc_dev_t *rtc	入参	RTC设备描述	使用hal_rtc_init时传入rtc_dev_t结构体
rtc_time_t *time	入参	要设置的时间	用户创建rtc_time_t结构体传入指针

返回值

返回成功或失败，返回0表示RTC时间设定成功，非0表示失败

调用示例

```
rtc_time_t  time_buf;
time_buf.sec   = 0;
time_buf.min   = 0;
time_buf.hr    = 0;
time_buf.weekday = 2;
time_buf.date  = 1;
time_buf.month  = 1;
time_buf.year  = 2019;
/* set rtc1 time to 2019/1/1,00:00:00 */
ret = hal_rtc_set_time(&rtc1, &time_buf);
```

hal_rtc_finalize

关闭指定RTC

函数原型

```
int32_t hal_rtc_finalize(rtc_dev_t *rtc);
```

参数

rtc_dev_t *rtc	入参	RTC设备描述	使用hal_rtc_init时传入rtc_dev_t结构体

返回值

返回成功或失败，返回0表示RTC关闭成功，非0表示失败

调用示例

```
hal_rtc_finalize(&rtc1);
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/rtc.h"
```

使用示例

```
#include <aos/hal/rtc.h>
#define RTC1_PORT_NUM 1
/* define dev */
rtc_dev_t rtc1;
int application_start(int argc, char *argv[])
{
  int ret = -1;
  rtc_config_t rtc_cfg;
  rtc_time_t  time_buf;
  /* rtc port set */
  rtc1.port = RTC1_PORT_NUM;
  /* set to DEC format */
  rtc1.config.format = HAL_RTC_FORMAT_DEC;
  /* init rtc1 with the given settings */
  ret = hal_rtc_init(&rtc1);
  if (ret != 0) {
    printf("rtc1 init error !\n");
  }
  time_buf.sec   = 0;
  time_buf.min   = 0;
  time_buf.hr    = 0;
  time_buf.weekday = 2;
  time_buf.date   = 1;
  time_buf.month  = 1;
  time_buf.year   = 2019;
  /* set rtc1 time to 2019/1/1,00:00:00 */
  ret = hal_rtc_set_time(&rtc1, &time_buf);
  if (ret != 0) {
    printf("rtc1 set time error !\n");
  }
  memset(&time_buf, 0, sizeof(rtc_time_t));
  /* get rtc current time */
  ret = hal_rtc_get_time(&rtc1, &time_buf);
  if (ret != 0) {
    printf("rtc1 get time error !\n");
  }
  /* finalize rtc1 */
  hal_rtc_finalize(&rtc1);
  while(1) {
    /* sleep 500ms */
    aos_msleep(500);
  };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_rtc_xxmcu.c和hal_wdg_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_rtc_xxmcu.c中实现所需要的hal函数，hal_rtc_xxmcu.h中放置相关宏定义。 参考platform/mcu/stm32l4xx/src/STM32L496G-Discovery/hal/hal_rtc_stm32l4.c

2.3.10. ADC

对于不同底层驱动的adc操作实现，统一封装成本文所述hal adc接口。 hal相关头文件位于目录：include/aos/hal。 hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/。

API列表

hal_adc_init	初始化指定ADC
hal_adc_value_get	获取ADC采样值
hal_adc_finalize	关闭指定ADC

API详情

请参考include/aos/hal/rtc.h

相关结数据结构

adc_dev_t

```
typedef struct {
  uint8_t    port;  /* adc port */
  adc_config_t config; /* adc config */
  void     *priv;  /* priv data */
} adc_dev_t;
```

adc_config_t

```
typedef struct {
  uint32_t sampling_cycle;  /* sampling period in number of ADC clock cycles */
} adc_config_t;
```

hal_adc_init

初始化指定ADC

函数原型

```
int hal_adc_init(adc_dev_t *adc);
```

参数

adc_dev_t *adc	入参	ADC设备描述	用户自定义一个adc_dev_t结构体

返回值

返回成功或失败, 返回0表示ADC初始化成功，非0表示失败

调用示例

```
#define ADC1_PORT_NUM 1
/* define dev */
adc_dev_t adc1;
/* adc port set */
adc1.port = ADC1_PORT_NUM;
/* set sampling_cycle */
adc1.config.sampling_cycle = 100;
/* init adc1 with the given settings */
ret = hal_adc_init(&adc1);
```

hal_adc_value_get

获取ADC采样值

函数原型

```
int  hal_adc_value_get(adc_dev_t *adc, void *output, unsigned int timeout);
```

参数

adc_dev_t *adc	入参	ADC设备描述	使用hal_adc_init时传入adc_dev_t结构体
void *output	出参	数据缓冲区	int value; 传入&value
unsigned int timeout	入参	超时时间，单位tick	HAL_WAIT_FOREVER

返回值

返回成功或失败, 返回0表示ADC时间获取成功，非0表示失败

调用示例

```
int value = 0;
adc_dev_t adc1;
ret = hal_adc_value_get(&adc1, &value, HAL_WAIT_FOREVER);
```

hal_adc_finalize

关闭指定ADC

函数原型

```
int32_t hal_adc_finalize(adc_dev_t *adc);
```

参数

adc_dev_t *adc	入参	ADC设备描述	使用hal_adc_init时传入adc_dev_t结构体

返回值

返回成功或失败, 返回0表示ADC去初始化成功，非0表示失败

调用示例

```
ret = hal_adc_finalize(&adc1);
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/adc.h"
```

使用示例

```
#include <aos/hal/adc.h>
#define ADC1_PORT_NUM 1
/* define dev */
adc_dev_t adc1;
int application_start(int argc, char *argv[])
{
  int ret  = -1;
  int value = 0;
  adc_config_t adc_cfg;
  /* adc port set */
  adc1.port = ADC1_PORT_NUM;
  /* set sampling_cycle */
  adc1.config.sampling_cycle = 100;
  /* init adc1 with the given settings */
  ret = hal_adc_init(&adc1);
  if (ret != 0) {
    printf("adc1 init error !\n");
  }
  /* get adc value */
  ret = hal_adc_value_get(&adc1, &value, HAL_WAIT_FOREVER);
  if (ret != 0) {
    printf("adc1 vaule get error !\n");
  }
  /* finalize adc1 */
  hal_adc_finalize(&adc1);
  while(1) {
    /* sleep 500ms */
    aos_msleep(500);
  };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_adc_xxmcu.c和hal_adc_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_adc_xxmcu.c中实现所需要的hal函数，hal_adc_xxmcu.h中放置相关宏定义。 参考platform/mcu/stm32l4xx/src/STM32L496G-Discovery/hal/hal_adc_stm32l4.c

2.3.11. DAC

对于不同底层驱动的dac操作实现，统一封装成本文所述hal dac接口。 hal相关头文件位于目录：include/aos/hal。hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/。

API列表

hal_dac_init	初始化指定DAC
hal_dac_start	开始DAC输出
hal_dac_stop	停止DAC输出
hal_dac_set_value	设置DAC输出值
hal_dac_get_value	获取当前DAC输出值
hal_dac_finalize	关闭指定DAC

API详情

请参考include/aos/hal/dac.h

相关结数据结构

dac_dev_t

```
typedef struct {
  uint8_t port;  /* dac port */
  void  *priv;  /* priv data */
} dac_dev_t;
```

hal_dac_init

初始化指定DAC

函数原型

```
int32_t hal_dac_init(dac_dev_t *dac);
```

参数

dac_dev_t *dac	入参	DAC设备描述	用户自定义一个dac_dev_t结构体

返回值

返回成功或失败, 返回0表示DAC初始化成功，非0表示失败

调用示例

```
#define DAC1_PORT_NUM 1
/* define dev */
dac_dev_t dac1;
/* dac port set */
dac1.port = DAC1_PORT_NUM;
ret = hal_dac_init(&dac1);
```

hal_dac_start

开始DAC输出

函数原型

```
int32_t hal_dac_start(dac_dev_t *dac, uint32_t channel);
```

参数

dac_dev_t *dac	入参	DAC设备描述	使用hal_dac_init传入参数
uint32_t channel	入参	输出通道号	0

返回值

返回成功或失败, 返回0表示DAC开始输出成功，非0表示失败

调用示例

```
ret = hal_dac_start(&dac1, 0);
```

hal_dac_stop

停止DAC输出

函数原型

```
int32_t hal_dac_stop(dac_dev_t *dac, uint32_t channel);
```

参数

dac_dev_t *dac	入参	DAC设备描述	使用hal_dac_init传入参数
uint32_t channel	入参	输出通道号	0

返回值

返回成功或失败, 返回0表示DAC停止成功，非0表示失败

调用示例

```
ret = hal_dac_stop(&dac1, 0);
```

hal_dac_set_value

设置DAC输出值

函数原型

```
int32_t hal_dac_set_value(dac_dev_t *dac, uint32_t channel, uint32_t data);
```

参数

dac_dev_t *dac	入参	DAC设备描述	使用hal_dac_init传入参数
uint32_t channel	入参	输出通道号	0
uint32_t data	入参	输出值	10

返回值

返回成功或失败, 返回0表示DAC输出值设定成功，非0表示失败

调用示例

```
ret = hal_dac_set_value(&dac1, 0, 10);
```

hal_dac_get_value

获取当前DAC输出值

函数原型

```
int32_t hal_dac_get_value(dac_dev_t *dac, uint32_t channel);
```

参数

dac_dev_t *dac	入参	DAC设备描述	使用hal_dac_init传入参数
uint32_t channel	入参	输出通道号	0

返回值

返回DAC输出值，获取失败返回负数。

调用示例

```
int value;
value = hal_dac_get_value(&dac1, 0);
```

hal_dac_finalize

关闭指定DAC

函数原型

```
int32_t hal_dac_finalize(dac_dev_t *dac);
```

参数

adc_dev_t *adc	入参	ADC设备描述	使用hal_adc_init时传入adc_dev_t结构体

返回值

返回成功或失败，返回0表示DAC关闭成功，非0表示失败

调用示例

```
ret = hal_dac_finalize(&dac1);
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/dac.h"
```

使用示例

```
#include <aos/hal/dac.h>
#define DAC1_PORT_NUM 1
#define DAC1_CHANNEL_NUM 1
/* define dev */
dac_dev_t dac1;
int application_start(int argc, char *argv[])
{
  int ret  = -1;
  int value = 0;
  static int count = 0;
  /* dac port set */
  dac1.port = DAC1_PORT_NUM;
  /* init dac1 with the given settings */
  ret = hal_dac_init(&dac1);
  if (ret != 0) {
    printf("dac1 init error !\n");
  }
  value = 10;
  ret = hal_dac_set_value(&dac1, DAC1_CHANNEL_NUM, value);
  if (ret != 0) {
    printf("dac1 set value error !\n");
  }
  /* start dac output */
  ret = hal_dac_start(&dac1, DAC1_CHANNEL_NUM);
  if (ret != 0) {
    printf("dac1 start error !\n");
  }
  while(1) {
    if (count == 10) {
      /* finalize dac1 */
      hal_dac_finalize(&dac1);
    }
    /* sleep 500ms */
    aos_msleep(500);
    count++;
  };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_dac_xxmcu.c和hal_dac_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_dac_xxmcu.c中实现所需要的hal函数，hal_dac_xxmcu.h中放置相关宏定义。 参考platform/mcu/stm32l4xx/src/STM32L496G-Discovery/hal/hal_dac_stm32l4.c

2.3.12. RNG

对于不同底层驱动的rng操作实现，统一封装成本文所述hal rng接口。 hal相关头文件位于目录：include/aos/hal。hal相关实现位于具体的mcu目录下，如：platform/mcu/stm32f1xx/hal/。

API列表

hal_random_num_read	获取随机数

API详情

请参考include/aos/hal/rng.h

相关结数据结构

random_dev_t

```
typedef struct {
  uint8_t port; /* random device port */
  void  *priv; /* priv data */
} random_dev_t;
```

hal_random_num_read

获取随机数

函数原型

```
int32_t hal_random_num_read(random_dev_t random, void *buf, int32_t bytes);
```

参数

random_dev_t random	入参	RNG设备描述	用户自定义一个random_dev_t结构体
void *buf	出参	返回的数据指针	char buf[4]; 传入buf
int32_t bytes	入参	数据字节数	4

返回值

返回成功或失败, 返回0表示获取随机数成功，非0表示失败

调用示例

```
#define RNG1_PORT_NUM 1
/* define dev */
random_dev_t rng1;
int value = 0;
rng1.port = RNG1_PORT_NUM;
ret = hal_random_num_read(rng1, &value, sizeof(int));
```

使用

添加该组件

在相应的platform/mcu的mk内，添加对应hal文件的编译包含。

包含头文件

```
#include "aos/hal/rng.h"
```

使用示例

```
#include <aos/hal/rng.h>
#define RNG1_PORT_NUM 1
/* define dev */
random_dev_t rng1;
int application_start(int argc, char *argv[])
{
  int ret  = -1;
  int value = 0;
  rng1.port = RNG1_PORT_NUM;
  ret = hal_random_num_read(rng1, &value, sizeof(int));
  if (ret != 0) {
    printf("rng read error !\n");
  }
  while(1) {
    /* sleep 500ms */
    aos_msleep(500);
  };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_rng_xxmcu.c和hal_rng_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_rng_xxmcu.c中实现所需要的hal函数，hal_rng_xxmcu.h中放置相关宏定义

2.4. 功能组件

2.4.1. 键值对存储模块（KV）

对于嵌入式系统应用中，频繁使用的参数存储，过程变量存储等操作，AliOS-Things为用户提供了一种更加直观易于理解的基于键值对的存储方式，如报警温度=50度，可以通过定义一个键值对：`{"AlarmTemp":50}`来进行存储。获取时，也只需通过AlarmTemp这个关键字（键/KEY)即可快速获取和重新写入。KV将用户的指定片内flash扇区或eeprom虚拟为有一个存储空间，并且帮助用户管理键值对之间的映射关系，用户无需关心具体的值被写到哪里了，仅需要通过对键的操作即可完成数据的存取。

相对传统的读-修改-写的flash操作方式，KV服务将为用户提供近100倍的效率提升。

KV功能的相关源码位于：/core/kv/ 目录中。

API列表

aos_kv_set()	更新某个键的值，如果键不存在则创建一个新的键
aos_kv_get()	获取某个键的值
aos_kv_del()	删除键值对，删除后将不能访问也不能恢复
aos_kv_del_by_prefix()	以固定前缀删除键值对。如“Set_”
aos_kv_secure_set()	加密更新/创建某个键的值
aos_kv_secure_get()	获取某个加密的键的值

使用

添加该组件

kv系统是AliOS Things 默认添加的组件，开发者无需再手动添加。

包含头文件

```
#include <aos/aos.h>
#include "aos/kv.h"
```

使用示例

```
/* set wifi-ssid to aiot */
aos_kv_set("wifi-ssid", "aiot", 4, 1);
/* get wifi-ssid */
char buf[10] = {0};
int ret = 0;
int len = sizeof(buf);
ret = aos_kv_get("wifi-ssid", buf, &len);
```

API详情

KV的应用层API说明请参考core/kv/include/kv_api.h。

aos_kv_set()

更新某个键的值，如果键不存在则创建一个新的键。

函数原型

```
int aos_kv_set(const char *key, const void *value, int len, int sync);
```

输入参数

const char *key	需要操作的键的名称	“AlarmTemp”
const void *value	写入该键的值	50
int len	值的长度，字节为单位	1
int sync	同步写入标志，固定为1	1

返回参数

返回参数宏定义见core/kv/include/kv_api.h文件

调用示例

```
static char *g_key_1 = "key_1";
static char *g_val_1 = "val_1";
int ret = 0;
ret = aos_kv_set(g_key_1, g_val_1, strlen(g_val_1),1);
```

aos_kv_get()

获取某个键的值。

函数原型

```
int aos_kv_get(const char *key, void *buffer, int *buffer_len);
```

参数列表

const char *key	需要操作的键的名称	“AlarmTemp”
void *buffer	读取该键的值	50
int *buffer_len	返回实际值的长度，字节为单位	1

返回参数

返回参数宏定义见core/kv/include/kv_api.h文件。

调用示例

```
static char *g_key_1 = "key_1";
int ret = 0;
char buf[10] = {0};
int len = sizeof(buf);
ret = aos_kv_get(g_key_1,buf,&len);
```

aos_kv_del()

删除键值对，删除后将不能访问也不能恢复。

函数原型

```
int aos_kv_del(const char *key);
```

参数列表

const char *key	需要操作的键的名称	“AlarmTemp”

返回参数

返回参数宏定义见core/kv/include/kv_api.h文件。

调用示例

```
static char *g_key_1 = "key_1";
int ret = 0;
ret = aos_kv_del(g_key_1);
```

aos_kv_del_by_prefix()

以固定前缀删除键值对。如“Set_”。

函数原型

```
int aos_kv_del_by_prefix(const char *prefix);
```

参数列表

const char *prefix	需要删除的键的名称前缀	“Set_”

返回参数

返回参数宏定义见core/kv/include/kv_api.h文件。

调用示例

```
static char *prefix = "Special_";
int ret = 0;
ret = aos_kv_del_by_prefix(prefix);
```

aos_kv_secure_set()

以加密的方式更新/创建某个键的值，使用该api需要在menuconfig配置选项中开启kv安全存储。

函数原型

```
int aos_kv_secure_set(const char *key, const void *value, int len, int sync);
```

参数列表

const char *key	需要操作的键的名称	“AlarmTemp”
const void *value	写入该键的值	50
int len	值的长度，字节为单位	1
int sync	同步写入标志，固定为1	1

返回参数

返回参数宏定义见core/kv/include/kv_api.h文件。

调用示例

```
static char *g_key_1 = "key_1";
static char *g_val_1 = "val_1";
int ret = 0;
ret = aos_kv_secure_set(g_key_1, g_val_1, strlen(g_val_1),1);
```

aos_kv_secure_get()

获取某个加密的键的值，使用该api需要在menuconfig配置选项中开启kv安全存储。

函数原型

```
int aos_kv_secure_get(const char *key, void *buffer, int *buffer_len);
```

参数列表

const char *key	需要操作的加密键的名称	“AlarmTemp”
void *buffer	读取该加密键的值	50
int *buffer_len	返回实际值的长度，字节为单位	1

返回参数

返回参数宏定义见core/kv/include/kv_api.h文件。

调用示例

```
static char *g_key_1 = "key_1";
int ret = 0;
char buf[10] = {0};
int len = sizeof(buf);
ret = aos_kv_secure_get(g_key_1,buf,&len);
```

配置说明

在AliOS Things 源码里面输入 aos make menuconfig 命令即可进入到menuconfig配置界面中，依次选择 Kernel -> Key-Value Storage 即可进入到KV组件的配置界面：

配置项说明

```
--- Key-value Storage
(32)  The garbage collection task priority        # KV垃圾回收任务（GC）的优先级，默认32
(1024) The garbage collection task stack size (bytes)  # KV垃圾回收任务（GC））的栈大小，默认1024
(128)  The maximum length of key buffer (bytes)      # 一条KV的key存储大小，默认128(bytes)
(512)  The maximum length of value buffer (bytes)    # 一条KV的value存储大小，默认512(bytes)
[ ]  Key-value Storage Security Support          # 开启kv安全存储，默认不开启
```

移植说明

和KV相关的实现都在kv_adapt.c文件中，位于/core/kv/kv_adapt.c用户可以参考进行实现。

kv依赖AliOS Things的flash操作接口来实现kv存储功能。

其他

返回参数定义

返回参数定义在core/kv/include/kv_api.h文件中。

```
/* Key-value function return code description */
#define KV_OK           0    /* Successed */
#define KV_LOOP_CONTINUE    10000 /* Loop Continue */
#define KV_ERR_NO_SPACE    -10001 /* The space is out of range */
#define KV_ERR_INVALID_PARAM -10002 /* The parameter is invalid */
#define KV_ERR_MALLOC_FAILED -10003 /* The os memory malloc error */
#define KV_ERR_NOT_FOUND    -10004 /* Could not found the item */
#define KV_ERR_FLASH_READ   -10005 /* The flash read operation error */
#define KV_ERR_FLASH_WRITE  -10006 /* The flash write operation error */
#define KV_ERR_FLASH_ERASE  -10007 /* The flash erase operation error */
#define KV_ERR_OS_LOCK     -10008 /* The error related to os lock */
#define KV_ERR_OS_SEM      -10009 /* The error related to os semaphose */
#define KV_ERR_ENCRYPT     -10010 /* Data encryption error */
#define KV_ERR_DECRYPT     -10011 /* Data decryption error */
#define KV_ERR_NOT_SUPPORT  -10012 /* The function is not support yet */
```

2.4.2. 命令行交互模块（CLI）

在日常嵌入式开发中，用户经常会自行实现一套类似Linux Shell的交互工具来实现通过串口命令控制设备进入某种特定的状态，或执行某个特定的操作。如系统自检，模拟运行，或者进入手动模式进行设备点动。AliOS-Things原生实现了一套名为CLI（command-line interface）的命令行交互工具，在提供基本的系统交互命令的基础上，也支持用户自定义命令。对应的AOS API接口实现位于：`osal/aos/cli.c`中；对应的模块内部实现位于：`kernel/cli/`目录中。

包含头文件

```
#include "aos/cli.h"
```

API列表

aos_cli_init	初始化CLI模块(只需系统启动时初始化一次即可)
aos_cli_stop	停止并释放CLI模块系统资源
aos_cli_get_tag	获取特殊的CLI打印标识符
aos_cli_register_command	CLI命令绑定
aos_cli_unregister_command	CLI命令解绑
aos_cli_register_commands	CLI命令集绑定（多个CLI命令）
aos_cli_unregister_commands	CLI命令集解绑（多个CLI命令）

aos_cli_chg_passwd	更改CLI登陆密码（需要先打开CLI密码登陆功能）

API详情

aos_cli_init()

初始化CLI模块。

函数原型

```
int aos_cli_init(void)
```

返回参数

0	执行成功
其他	执行错误

调用示例

```
int ret = 0;
ret = aos_cli_init();
```

aos_cli_stop()

停止并释放CLI模块系统资源。

函数原型

```
int aos_cli_stop(void)
```

返回参数

0	执行成功
其他	执行错误

调用示例

```
int ret = 0;
ret = aos_cli_stop();
```

aos_cli_get_tag()

获取特殊的CLI打印标识符。

函数原型

```
const char *aos_cli_get_tag(void)
```

返回参数

const char *tag	特殊的CLI打印标识符
NULL	执行错误

调用示例

```
const char *tag = aos_cli_get_tag();
```

aos_cli_register_command()

CLI命令绑定。

函数原型

```
int aos_cli_register_command(const struct cli_command *cmd)
```

输入参数

const struct cli_command *cmd	需要执行绑定的CLI命令

返回参数

0	执行成功
其他	执行错误

调用示例

```
const struct cli_command command;
int ret = 0;
ret = aos_cli_register_command(&command);
```

aos_cli_unregister_command()

CLI命令解绑。

函数原型

```
int aos_cli_unregister_command(const struct cli_command *cmd)
```

输入参数

const struct cli_command *cmd	需要执行解绑的CLI命令

返回参数

0	执行成功
其他	执行错误

调用示例

```
int ret = 0;
ret = aos_cli_unregister_command(&command);
```

aos_cli_register_commands()

CLI命令集绑定。

函数原型

```
int aos_cli_register_commands(const struct cli_command *cmds, int num)
```

输入参数

const struct cli_command *cmds	需要执行绑定的CLI命令集
int num	命令集包含的CLI命令个数

返回参数

0	执行成功
其他	执行错误

调用示例

```
const struct cli_command command[3];
int ret = 0;
ret = aos_cli_register_commands(&command, 3);
```

aos_cli_unregister_commands()

CLI命令集解绑。

函数原型

```
int aos_cli_unregister_commands(const struct cli_command *cmds, int num)
```

输入参数

const struct cli_command *cmds	需要执行解绑的CLI命令集
int num	命令集包含的CLI命令个数

返回参数

0	执行成功
其他	执行错误

调用示例

```
// const struct cli_command command[3];
int ret = 0;
ret = aos_cli_register_commands(&command, 3);
```

aos_cli_chg_passwd

更改CLI登陆密码

函数原型

aos_cli_chg_passwd(char *old_passwd, char *new_passwd)

输入参数

char *old_passwd	旧密码（默认初始密码是aos）,密码长度限制为16ytes
char *new_passwd	新设置的密码，密码长度限制为16ytes

输出参数

0	密码修改成功
其他	密码修改失败

调用示例

```
int ret;
char old_passwd[16] = "aaa"
char new_passwd[16] = "bbb"
ret = aos_cli_chg_passwd(old_passwd, new_passwd);
```

配置文件

- cli_conf.h

文件位置：kernel/cli/include/cli_conf.h描述：CLI模块相关配置配置项

CLI_CONFIG_MINIMUM_MODE	0	CLI最小工作模式
CLI_CONFIG_INBUF_SIZE	256	CLI传入缓存大小
CLI_CONFIG_OUTBUF_SIZE	512	CLI传出缓存大小
CLI_CONFIG_MAX_COMMANDS	64	CLI可支持最大命令个数
CLI_CONFIG_MAX_ARG_NUM	16	CLI可支持最大命令参数个数
CLI_CONFIG_MAX_ONCECMD_NUM	4	CLI可支持最大(单次执行)命令个数
CLI_CONFIG_STACK_SIZE	2048	CLI任务栈大小
CLI_CONFIG_TASK_PRIORITY	60	CLI任务优先级
CLI_CONFIG_TELNET_SUPPORT	0	CLI TELNEL工作模式

移植说明

和CLI相关的移植实现都在cli_adapt.h/.c文件中，位于/kernel/cli/cli_adapt.c用户可以参考进行实现。需要移植的函数及说明：

cli_getchar()

从串口接收字符。

函数原型

```
int32_t cli_getchar(char *inbuf)
```

参数列表

char *inbuf	CLI 从串口接收的字符指针

返回参数

返回1表示CLI从串口接收到字符，0表示暂无字符输入。

cli_putstr()

通过串口或TELNET 输出字符串。

函数原型

```
int32_t cli_putstr(char *msg)
```

参数列表

char *msg	CLI 通过串口或TELNET 输出的字符串指针

返回参数

返回0表示字符串输出成功，其他则表示失败。

其他

命令结构定义

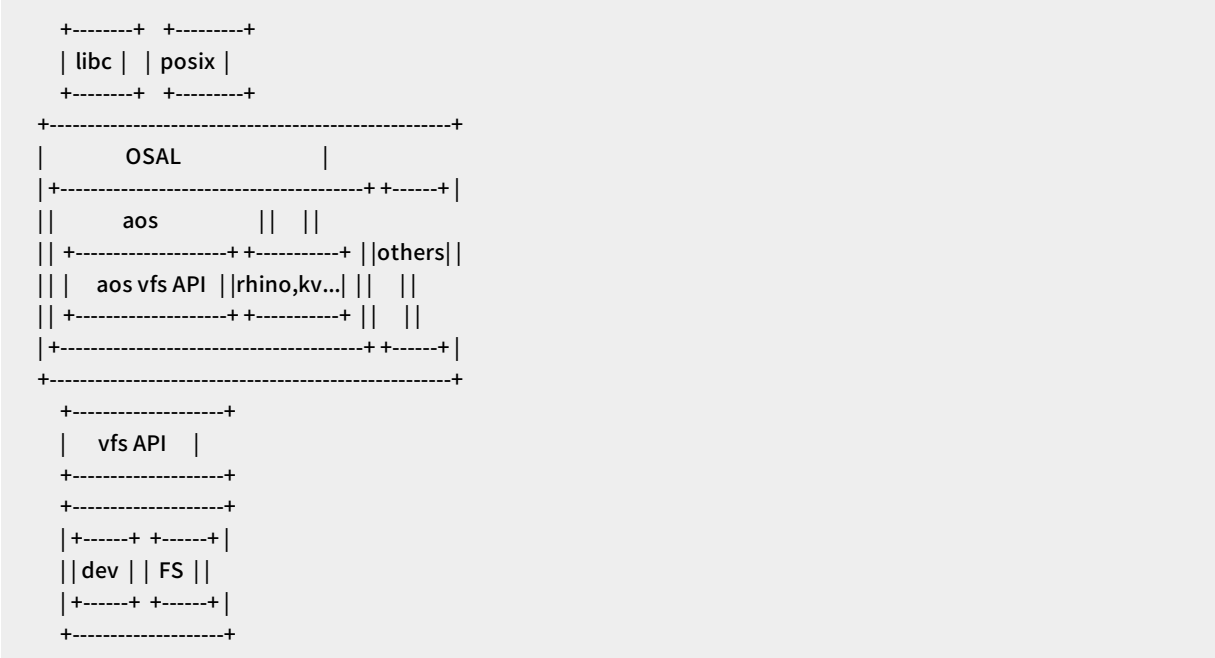
```
struct cli_command {
  const char *name;
  const char *help;
  void (*function)(char *outbuf, int len, int argc, char **argv);
};
```

2.4.3. 虚拟文件系统（VFS）

随着嵌入式设备功能的发展， 应用需要存储的数据越来越多，也越来越复杂。文件系统就是一种来应对这些繁琐复杂的数据管理方式。具体的文件系统（FATFS、SPIFFS等）都实现了一套数据的存储、分级组织、访问和获取等操作的抽象数据类型（Abstract data type），向用户提供了一种底层数据访问的机制，数据存储的基本单位为文件。

虚拟文件系统（Virtual File System）则提供了一种对具体文件系统类型的一个抽象，它可以将多个具体的文件系统接口统一起来，用户可通过虚拟文件系统屏蔽各个底层具体文件系统的操作接口、数据类型差异。用户也可通过注册接口将自己的文件系统挂载到VFS上进行操作访问。虚拟文件系统的风格与UNIX/Linux类似， 根目录及目录分隔符都用“/”来表示。

VFS对外提供aos_xxx接口，对应的头文件为 `aos/vfs.h` ，对应的接口实现位于： `core/osal/aos/vfs.c` 中。



严格来说，VFS是上图中vfs API这一层，对应的头文件是vfs_api.h，位于core/vfs/include/vfs_api.h，这些API仅供core内部使用。AliOS Things对应用开发提供统一的aos API，对于vfs提供aos vfs API。

API列表

名称	描述
aos_vfs_init	初始化VFS模块，申请系统资源
aos_open	打开文件
aos_close	关闭文件
aos_read	读取文件
aos_write	写入文件
aos_ioctl	执行特殊定义命令
aos_lseek	设置下次读取文件的位置
aos_sync	同步文件数据到存储设备
aos_stat	获取文件状态
aos_fstat	获取文件状态（POSIX）
aos_link	创建文件链接
aos_unlink	删除文件链接
aos_remove	删除文件或目录
aos_rename	重命名文件

aos_opendir	打开目录
aos_closedir	关闭目录
aos_readdir	读取目录
aos_mkdir	创建目录
aos_rmdir	删除目录
aos_rewinddir	重设读取目录的位置为开头
aos_telldir	获取目录的读取位置
aos_seekdir	设置下次读取目录的位置
aos_statfs	获取文件系统相关信息
aos_access	确定文件或目录的访问权限
aos_chdir	更改当前工作目录
aos_getcwd	获取当前工作目录
aos_pathconf	返回配置文件限制值
aos_fpathconf	返回配置文件限制值
aos_utime	设置文件访问或修改时间
aos_register_fs	挂载文件系统
aos_unregister_fs	卸载文件系统

使用

添加该组件

要使用上面 API列表中的API，需要同时添加vfs组件、osal_aos组件。vfs最终对接到文件系统或者设备驱动，所以应用要完整使用vfs，还需要添加文件系统（ramfs、spiffs、fatfs等）或者设备驱动组件。

例如，在某个在aos.mk文件中同时添加vfs、osal_aos和spiffs组件：

```
$(NAME)_COMPONENTS += vfs osal_aos spiffs
```

在AliOS Things中，由于组件vfs、osal_aos是基础组件，很多其他组件依赖这两个组件，如果选择了依赖vfs、或者osal_aos组件的其它组件，那么它(们)将被自动选中，无需额外再添加。

包含头文件

头文件所在位置：include/aos/vfs.h

```
#include "aos/vfs.h"
```

使用示例

这里以spiffs为例，演示应用中使用vfs API。作为示例，下面代码中注释了挂载spiffs的代码，用户可以添加真实的挂载文件系统的代码逻辑。

```
#include <stdio.h>
#include "aos/vfs.h"
// spiffs file operation structure
extern vfs_filesystem_ops_t spiffs_ops;
int fd;
int ret;
// mount spiffs firstly, add your mount logic here
// SPIFFS_mount(...);
// register spiffs filesystem
ret = aos_register_fs(path, &spiffs_ops, NULL);
if (ret < 0) {
  printf("register spiffs failed, errno %d\r\n", ret);
  return ret;
}
fd = aos_open("/data/a.txt", O_CREAT | O_RDWR);
if (fd < 0) {
  printf("open failed, errno %d\r\n", fd);
  return fd;
}
const char *str = "hello, world\n";
aos_write(fd, str, strlen(str));
aos_close(fd);
```

API详情

aos_vfs_init

函数原型

int aos_vfs_init(void);

详细说明

*aos_vfs_init()*初始化*VFS*系统。系统启动流程中组件初始化函数*aos_components_init()*函数调用了*vfs_init()*函数初始化*VFS*系统，和调用*aos_vfs_init()*效果相同。

输入参数

无

返回参数

返回值类型	返回值	描述
int	0	执行成功
负数	执行错误，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
int ret = 0;
ret = aos_vfs_init();
if (ret) {
  printf("aos_vfs_init() failed, errno %d\r\n", ret);
}
```

aos_open

函数原型

int aos_open(const char *path, int flags);

详细说明

*aos_open()*打开*path*所描述的文件或者设备，打开成功时返回系统分配的文件描述符*fd*，错误时返回对应的错误码。如果打开的文件不存在，但是参数*flags*指定了*O_CREAT*，那么就尝试创建文件。

打开文件时参数*flags*必须要包含*O_RDONLY*, *O_WRONLY*, *O_RDWR*三者之一。此外，打开文件是还可以指定文件创建标志位、文件状态标志位，这些标志位通过或'|'操作到参数flags。在这里常用的标志位有*O_CREAT*, *O_EXCL*, *O_TRUNC*, *O_APPEND*。其他在Linux操作系统中的类似标志位，在AliOS Things上可能不支持，但是指定了也不会有副作用。

O_RDONLY	以只读方式打开文件
O_WRONLY	以只写方式打开
O_RDWR	以可读可写方式打开
O_CREAT	文件不存在时创建文件
O_EXCL	在打开文件时，如果文件存在而且同时指定的标志位O_CREAT，则返回-EEXIST；如果文件不存在，则创建文件
O_TRUNC	打开文件时，如果文件的长度不为0，则将文件的长度截短为0，下次写入文件时，从文件开始出写入数据
O_APPEND	打开文件后，将文件的写入点设置为文件末尾，下次写入文件时，新写入的数据会追加到文件末尾

输入参数

参数类型	参数名称	参数描述
const char *	path	需打开的文件路径
int	flags	文件打开时的访问模式和标志

返回参数

返回值类型	返回值	描述
int	正数	文件描述符
负数	文件打开错误，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *path = /sdcard/test.txt;
int fd;
/**
 * open file in read/write mode, if file doesn't exist, create it.
 */
fd = aos_open(path, O_RDWR | O_CREAT);
if (fd < 0) {
   printf("aos_open(%s, 0x%x) failed, errno %d\r\n", path, fd);
}
```

aos_close

函数原型

int aos_close(int fd);

详细说明

关闭已打开的文件描述符*fd*。该描述符*fd*后续会被打开其他文件时复用。

输入参数

参数类型	参数名称	参数描述
int	fd	关闭文件的描述符

返回参数

返回值类型	返回值	描述
int	0	关闭成功
负数	关闭错误，返回错误码	

调用示例

```
#include <stdio.h>
#include “aos/vfs.h”
int fd;
int ret;
/**
 * open file in read/write mode, if file doesn't exist, create it.
 */
fd = aos_open("helloworld.txt", O_RDWR | O_CREAT);
if (fd < 0) {
  printf("aos_open(%s, 0x%x) failed, errno %d\r\n", path, fd);
  // do error handle
}
// do your file operations, such as read/write. etc.
// close the file
ret = aos_close(fd);
if (ret < 0) {
  printf("aos_close(%d) failed, errno %d\r\n", ret);
}
```

aos_read

函数原型

int aos_read(int fd, void *buf, size_t nbytes);

详细说明

*aos_read()*从打开的文件中读取若干字节到*buffer*。如果文件支持*seek*，那么*aos_read()*将从文件当前偏移(*offset*)处开始读取，读取成功后将文件的偏移向后增加成功读取到的字节数。

读取成功，返回读取到的字节个数；如果读取时文件偏移已经到达文件末尾，则返回0。其他错误情况，返回错误码。

输入参数

参数类型	参数名称	参数描述
int	fd	文件描述符
void *	buf	文件读取的数据缓存
size_t	nbytes	要读取的字节数

返回参数

返回值类型	返回值	描述
int	正数	成功读取到的字节数
0	已经读取到文件末尾，没有更多的数据可以读取	
负数	读取失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
#define BUF_SIZE (64)
int fd;
char buf[BUF_SIZE];
int ret;
/**
 * open file in read/write mode, if file doesn't exist, create it.
 */
fd = aos_open("helloworld.txt", O_RDONLY | O_CREAT);
if (fd < 0) {
   printf("aos_open(%s, 0x%x) failed, errno %d\r\n", path, fd);
   // do error handle
}
// fd is the open file descriptor
ret = aos_read(fd, buf, BUF_SIZE);
if (ret < 0) {
   printf("aos_read() failed, errno %d\r\n", ret);
} else if (ret == 0) {
   printf("end of the file\r\n");
} else {
   // handle the data read from file
}
```

aos_write

函数原型

int aos_write(int fd, const void *buf, size_t nbytes);

详细说明

*aos_write()*将*buffer*开始出的*nbytes*字节数据写入到已经打开的文件。如果文件支持*seek*，那么*aos_write()*将从文件当前偏移(*offset*)处开始写入，写入成功后将文件的偏移向后增加成功写入的字节数。

当以*O_APPEND*方式打开文件时，打开成功后，文件的偏移*offset*会被设置为文件末尾。

写入成功时，返回写入到文件的字节个数，实际写入的字节数有可能小于*nbytes*（文件空间不足）；写入失败时返回错误码（负数）。

输入参数

参数类型	参数名称	参数描述
int fd	fd	文件描述符
void *	buf	文件写入数据的缓存
size_t	nbytes	文件写入的长度

返回参数

返回值类型	返回值	描述
int	正数或0	成功写入文件的字节数
负数	读取失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *str="hello, aos_write";
int fd;
int ret;
/**
 * open file in read/write mode, if file doesn't exist, create it.
 */
fd = aos_open("helloworld.txt", O_WRONLY | O_CREAT | O_APPEND);
if (fd < 0) {
   printf("aos_open(%s, 0x%x) failed, errno %d\r\n", path, fd);
   // do error handle
}
// fd is the open file descriptor
ret = aos_write(fd, str, strlen(str));
if (ret < 0) {
   printf("aos_write() failed, errno %d\r\n", ret);
} else {
   printf("aos_write() writes %d char\r\n", ret);
}
```

aos_ioctl

函数原型

int aos_ioctl(int fd, int cmd, unsigned long arg);

详细说明

*aos_ioctl()*一般用于设置或者读取底层设备的参数。参数fd必须是已打开设备的描述符，*cmd*是设备相关的请求码，第三个参数*arg*是与*cmd*相关的参数。一般情况下，*aos_ioctl()*的输出参数通过第三个参数*arg*（将指针作为*arg*参数）返回，也有少数情况通过函数返回值返回。

输入参数

参数类型	参数名称	参数描述
int	fd	文件描述符（FD）
int	cmd	特殊命令标识
unsigned long	arg	命令参数

返回参数

返回值	返回值	描述
int	0	成功
正数	一些用法中，将返回值作为aos_ioctl()的输出参数	
负数	失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
#define CMD_SPEC 0
#define CMD_VAL  0
const char *dev = "/dev/i2c0";
int fd;
int ret;
/**
 * open file in read/write mode, if file doesn't exist, create it.
 */
fd = aos_open(dev, O_RDONLY);
if (fd < 0) {
   printf("aos_open(%s, 0x%x) failed, errno %d\r\n", path, fd);
   // do error handle
}
ret = aos_ioctl(fd, CMD_SPEC, CMD_VAL);
if (ret < 0) {
   printf("aos_ioctl() failed, errno %d\r\n", ret);
} else {
   // aos_ioctl() okay
}
```

aos_lseek

函数原型

off_t aos_lseek(int fd, off_t offset, int whence);

详细说明

*aos_lseek()*重新设置已打开文件的偏移，第二个参数*offset*是相对于第三个参数*whence*的偏移值，语义如下：***SEEK_SET***：文件偏移设置为从文件开始处*offset*字节处。 ***SEEK_CUR***：文件偏移设置为相对于当前文件位置的*offset*字节处。 ***SEEK_END***：文件偏移摄者为文件末尾的*offset*字节处。

输入参数

参数类型	参数名称	参数描述
int	fd	文件描述符
off_t	offset	文件读写位置需偏移长度
int	whence	文件读写位置偏移起始点

返回参数

返回值类型	返回值 int	描述
int	非负	成功，返回相对于文件开始处的偏移字节数，该值作为当前文件的偏移
负数	失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *file="helloworld.txt";
int fd;
int ret;
/**
 * open file in read/write mode, if file doesn't exist, create it.
 */
fd = aos_open(file, O_WRONLY);
if (fd < 0) {
   printf("aos_open(%s, 0x%x) failed, errno %d\r\n", path, fd);
   // do error handle
}
// set file offset to the end of the file.
ret = aos_lseek(fd, 0, SEEK_END);
if (ret < 0) {
   printf("aos_lseek() failed, errno %d\r\n", ret);
} else {
   printf("file %s size is %d Bytes\r\n", file, ret);
}
```

aos_sync

函数原型

int aos_sync(int fd);

详细说明

*aos_sync()*会将对文件系统元数据(*metadata*)的修改以及文件系统内部缓存中的数据写入底层文件系统中。

输入参数

参数类型	参数名称	参数描述
int	fd	文件描述符

返回参数

返回值类型	返回值	描述
int	0	数据同步成功
负数	数据同步失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *str="hello, aos_write";
int fd;
int ret;
/**
 * open file in read/write mode, if file doesn't exist, create it.
 */
fd = aos_open("helloworld.txt", O_WRONLY | O_CREAT | O_APPEND);
if (fd < 0) {
   printf("aos_open(%s, 0x%x) failed, errno %d\r\n", path, fd);
   // do error handle
}
// fd is the open file descriptor
ret = aos_write(fd, str, strlen(str));
if (ret < 0) {
   printf("aos_write() failed, errno %d\r\n", ret);
} else {
   printf("aos_write() writes %d char\r\n", ret);
}
// flush metadata and/or data into underlying file system.
ret = aos_sync(fd);
// other file operations
aos_close(fd);
```

aos_stat

函数原型

int aos_stat(const char *path, struct aos_stat *st);

详细说明

*aos_stat()*读取文件的属性信息，如文件大小、文件模式（权限、文件/文件夹）等信息到*st*指针指向的*buffer*。

输入参数

参数类型	参数名称	参数描述
const char *	path	文件路径
struct aos_stat *	st	文件状态信息结构体指针

返回参数

返回值类型	返回值	描述
int	0	获取文件信息成功
负数	获取文件信息失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *path = "/sdcard/test.txt";
struct aos_stat file_st;
int ret;
uint32_t size;
uint16_t mode;
/**
 * open file in read/write mode, if file doesn't exist, create it.
 */
fd = aos_open("helloworld.txt", O_WRONLY | O_CREAT | O_APPEND);
if (fd < 0) {
   printf("aos_open(%s, 0x%x) failed, errno %d\r\n", path, fd);
   // do error handle
}
ret = aos_fstat(fd, &file_st);
if (!ret) {
   size = file_st.st_size;
   mode = file_st.st_mode;
}
```

aos_fstat

函数原型

int aos_fstat(int fd, struct aos_stat *st);

详细说明

*aos_fstat()*函数的功能和*aos_stat()*相同，*aos_fstat()*读取文件的属性信息，如文件大小、文件模式（权限、文件/文件夹）等信息到*st*指针指向的*buffer*。区别在于，*aos_fstat()*函数使用文件描述符作为参数，而*aos_stat()*以文件的路径为参数。

输入参数

参数类型	参数名称	参数描述
int	fd	文件描述符(File Descriptor)
struct aos_stat *	st	文件状态信息结构体指针

返回参数

返回值类型	返回值	描述
int	0	获取文件信息成功
负数	获取文件信息失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *path = "/sdcard/test.txt";
struct aos_stat file_st;
int ret;
uint32_t size;
uint16_t mode;
ret = aos_stat(path, &file_st);
if (!ret) {
  size = file_st.st_size;
  mode = file_st.st_mode;
}
```

aos_link

函数原型

int aos_link(const char *oldpath, const char *newpath);

详细说明

*aos_link()*为文件创建一个新的链接文件。如果目标文件存在，则不会覆盖。使用老的文件名和新的文件名访问的是同一个文件，使用它们对文件的操作没有任何差异。

输入参数

参数类型	参数名称	参数描述
const char *	oldpath	文件链接的源文件名
const char *	newpath	文件链接的目标文件名

返回参数

返回值类型	返回值 int	描述
int	0	文件链接创建成功
负数	文件链接创建失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *oldname = "/sdcard/test1.txt";
const char *newname = "/sdcard/test2.txt";
int ret;
ret = aos_link(oldname, newname);
if (ret < 0) {
  printf("aos_link(%s, %s) failed, errno %d\r\n", oldname, newname, ret);
} else {
  printf("aos_link(%s, %s) success.\r\n", oldname, newname);
}
```

aos_unlink

函数原型

int aos_unlink(const char *path);

详细说明

*aos_unlink()*从文件系统中删除文件名。如果这个文件名是文件的最后一个链接，同时没有其他程序打开这个文件，那么*aos_unlink()*将会从文件系统中删除该文件，将该文件占用的空间释放出来。如果这个文件名是文件的最后一个链接，但是有其他程序打开了该文件，那么该文件将持续存在，直到最后一个打开该文件的程序关闭该文件。

输入参数

参数类型	参数名称	参数描述
const char *path	path	需删除的文件路径

返回参数

返回值类型	返回值	描述
int	0	文件名删除成功
负数	文件名删除失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *filename = "/sdcard/test1.txt";
int ret;
int ret = aos_unlink(filename);
if (ret < 0) {
  printf("aos_unlink(%s) failed, errno %d\r\n", filename, ret);
} else {
  printf("aos_unlink(%s) success.\r\n", filename);
}
```

aos_remove

函数原型

int aos_remove(const char *path);

详细说明

*aos_remove()*从文件系统中删除文件名，它既可以删除文件，也可以删除目录。

输入参数

参数类型	参数名称	参数描述
const char *	path	要被删除的文件或目录路径

返回参数

返回值类型	返回值	描述
int	0	文件或目录删除成功

负数	文件或目录删除失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *filename = "/sdcard/test1.txt";
int ret;
ret = aos_remove(filename);
if (ret < 0) {
  printf("aos_remove(%s) failed, errno %d\r\n", filename, ret);
} else {
  printf("aos_remove(%s) success.\r\n", filename);
}
```

aos_rename

函数原型

int aos_rename(const char *oldpath, const char *newpath);

详细说明

*aos_rename()*重命名一个文件，可以将文件从一个文件夹下通过重命名移动到另外一个文件夹下。

如果*newpath*指向的文件已经存在，*aos_rename()*将会自动替换它。

如果*oldpath*和*newpath*是指向同一个文件的连接文件，那么*aos_rename()*什么都不做，直接返回成功。

如果*newpath*指向的文件已经存在，但是*aos_rename()*由于某种原因失败了，系统会保证存在一份名为*newpath*的实例。

*oldpath*可以是一个文件夹，在这种情形下，*newpath*必须要么不存在，要么是一个空文件夹。

如果*oldpath*是一个链接，那么*aos_rename()*将重命名该链接；如果newpath是一个已经存在的链接，那么它将会被替换。

输入参数

参数类型	参数名称	参数描述
const char *	oldpath	原文件路径名
const char *	newpath	新文件路径名

返回参数

返回值类型	返回值	描述
int	0	文件重命名成功
负数	文件重命名失败，返回错误码	

调用示例

```
#include <stdio.h>
#include “aos/vfs.h”
const char *oldpath = "/sdcard/origin.txt";
const char *newpath = "/sdcard/new.txt";
int ret;
ret = aos_rename(oldpath, newpath);
if (ret < 0) {
  printf("aos_rename(%s, %s) failed, errno %d\r\n", oldpath, newpath, ret);
} else {
  printf("aos_rename(%s, %s) success\r\n", oldpath, newpath);
}
```

aos_opendir

函数原型

aos_dir_t *aos_opendir(const char *path);

详细说明

*aos_opendir(*打开目录名*path*对应的目录流，并返回目录流指针。打开目录成功后，流位置指向目录的第一个成员。

输入参数

参数类型	参数名称	参数描述
const char *	path	要打开的目录路径

返回参数

返回值类型	返回值	描述
aos_dir_t*	非NULL	打开目录成功
NULL	打开目录失败	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char dirpath = "/sdcard/dir";
aos_dir_t *dp;
dp = aos_opendir(dirpath);
if (NULL == dp) {
  printf("aos_opendir(%s) failed\r\n", dirpath);
} else {
  printf("aos_opendir(%s) success\r\n", dirpath);
}
```

aos_closedir

函数原型

int aos_closedir(aos_dir_t *dirp);

详细说明

*aos_closedir()*关闭目录流。调用*aos_closedir()*后，目录流指针不再可用。

输入参数

参数类型	参数名称	参数描述
aos_dir_t *	dirp	要关闭的目录流指针

返回参数

返回值类型	返回值	描述
int	0	目录关闭成功
负数	目录关闭失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char dirpath = "/sdcard/dir";
aos_dir_t *dirp;
dirp = aos_opendir(dirpath);
if (NULL == dirp) {
  printf("aos_opendir(%s) failed\r\n", dirpath);
  // do error handle.
} else {
  printf("aos_opendir(%s) success\r\n", dirpath);
}
// do dir operations
ret = aos_closedir(dirp);
if (ret < 0) {
  printf("aos_closedir() failed, errno %d \r\n", ret);
} else {
  printf("aos_closedir() success\r\n");
}
```

aos_readdir

函数原型

aos_dirent_t *aos_readdir(aos_dir_t *dirp);

详细说明

*aos_readdir()*返回一个指向*aos_dirent_t*的指针，该指针关联的目录流*dirp*的成员指针指向下一个成员。当读取到目录的末尾或者读取失败，返回*NULL*。

输入参数

参数类型	参数名称	参数描述
aos_dir_t *	dirp	需读取的目录流指针

返回参数

返回值类型	返回值 aos_dirent_t *	描述

aos_dirent_t *	非NULL	读取目录成功，返回aos_dirent_t类型的指针
NULL	读取到目录的末尾或者读取失败	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char dirpath = "/sdcard/dir";
aos_dir_t *dirp;
dirp = aos_opendir(dirpath);
if (NULL == dirp) {
  printf("aos_opendir(%s) failed\r\n", dirpath);
  // do error handle.
} else {
  printf("aos_opendir(%s) success\r\n", dirpath);
}
aos_dirent_t *dirent;
while (dirent = aos_readdir(dirp)) {
  printf("dir name: %s\r\n" dirent->d_name);
}
ret = aos_closedir(dirp);
if (ret < 0) {
  printf("aos_closedir() failed, errno %d \r\n", ret);
} else {
  printf("aos_closedir() success\r\n");
}
```

aos_mkdir

函数原型

int aos_mkdir(const char *path);

详细说明

*aos_mkdir()*创建名为*path*的目录。如果已经存在*path*目录，则创建失败。

输入参数

参数类型	参数名称	参数描述
const char *	path	需创建的目录路径

返回参数

返回值类型	返回值	描述
int	0	目录创建成功
负数	目录创建失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *path = "/sdcard/dir";
int ret;
ret = aos_mkdir(path);
if (ret < 0) {
  printf("aos_mkdir(%s) failed, errno %d\r\n", path, ret);
} else {
  printf("aos_mkdir(%s) success\r\n", path);
}
```

aos_rmdir

函数原型

int aos_rmdir(const char *path);

详细说明

*aos_rmdir()*删除一个目录。删除目录时，目录必须为空。

输入参数

参数类型	参数名称	参数描述
const char *	path	需删除的目录路径

返回参数

返回值	返回值 int	描述
int	0	目录删除成功
负数	目录删除失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *path = "/sdcard/dir";
int ret;
ret = aos_mkdir(path);
if (ret < 0) {
  printf("aos_mkdir(%s) failed, errno %d\r\n", path, ret);
} else {
  printf("aos_mkdir(%s) success\r\n", path);
}
// do any operations
ret = aos_rmdir(path);
if (ret < 0) {
  printf("aos_rmdir(%s) failed, errno %d\r\n", path, ret);
} else {
  printf("aos_rmdir(%s) success.", path);
}
```

aos_rewinddir

函数原型

void aos_rewinddir(aos_dir_t *dirp);

详细说明

*aos_rewinddir()*将目录流dirp的位置重置为目录的开始处。

输入参数

参数类型	参数名称	参数描述
aos_dir_t *	dir	要重置的目录流指针

返回参数

无

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char dirpath = "/sdcard/dir";
aos_dir_t *dirp;
dirp = aos_opendir(dirpath);
if (NULL == dirp) {
  printf("aos_opendir(%s) failed\r\n", dirpath);
  // do error handle.
} else {
  printf("aos_opendir(%s) success\r\n", dirpath);
}
aos_dirent_t *dirent;
while (dirent = aos_readdir(dirp)) {
  printf("dir name: %s\r\n" dirent->d_name);
}
// reset the position of the dirp
aos_rewinddir(dirp);
```

aos_telldir

函数原型

int aos_telldir(aos_dir_t *dirp);

详细说明

*aos_telldir()*返回与目录流dirp相关联的目录流的当前位置。

输入参数

参数类型	参数名称	参数描述
aos_dir_t *	dirp	要获取位置的目录流指针

返回参数

返回值类型	返回值	描述
int	非负数	获取成功，目录流当前的位置

负数	获取失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char dirpath = "/sdcard/dir";
aos_dir_t *dirp;
dirp = aos_opendir(dirpath);
if (dirp) {
  aos_dirent_t *dirent;
  int loc;
  while (dirent = aos_readdir(dirp)) {
    printf("dir name: %s\r\n" dirent->d_name);
    loc = aos_telldir(dirp);
    printf("loc %d\r\n", loc);
  }
  closedir(dirp);
}
```

aos_seekdir

函数原型

void aos_seekdir(aos_dir_t *dir, long loc);

详细说明

*aos_seekdir()*设置目录流的位置，下次调用*aos_readdir()*时目录流将从设置的位置开始读取目录。参数*loc*必须是之前通过调用*aos_readdir()*返回的值。

输入参数

参数名称	参数描述
aos_dir_t *dir	需设置读取位置的目录流指针
long loc	读取位置

返回参

无

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char dirpath = "/sdcard/dir";
aos_dir_t *dirp;
// open dir
dirp = aos_opendir(dirpath);
if (dirp) {
  aos_dirent_t *dirent;
  int loc;
  int seek_loc -1;
  // traverse the dir and record the location of "hello.txt"
  while (dirent = aos_readdir(dirp)) {
    printf("dir name: %s\r\n" dirent->d_name);
    loc = aos_telldir(dirp);
    printf("loc %d\r\n", loc);
    if (strcmp(dirent->d_name, "hello.txt") == 0) {
      seek_loc = loc;
    }
  }
  if (seek_loc != -1) {
    // seekdir, set the position of the dir to "hello.txt"
    aos_seekdir(dirp, (long)seek_loc);
    // traverse the dir from the position of "hello.txt"
    while (dirent = aos_readdir(dirp)) {
      printf("dir name: %s\r\n" dirent->d_name);
      loc = aos_telldir(dirp);
      printf("loc %d\r\n", loc);
      if (strcmp(dirent->d_name, "hello.txt") == 0) {
        seek_loc = loc;
      }
    }
  }
  closedir(dirp);
}
```

aos_statfs

函数原型

int aos_statfs(const char *path, struct aos_statfs *buf);

详细说明

*aos_statfs()*返回挂载的文件系统的信息。*path*是挂载的文件系统路径或者文件系统挂载路径下的任何文件路径，*buf*是指向*struct aos_statfs*结构体指针，用于获取文件系统的信息。

输入参数

参数类型	参数名称	参数描述
const char *	path	挂载的文件系统路径或者文件系统挂载路径下的任何文件路径
struct aos_statfs *	buf	文件系统信息结构缓存

返回参数

返回值	返回值 int	描述
int	0	文件系统信息获取成功
负数	文件系统信息获取失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *path = "/sdcard/a.txt";
struct aos_statfs statfs_buf;
int ret;
ret = aos_statfs(path, &statfs_buf);
if (!ret) {
  // print fs total size, free size
  printf("fs size %d Bytes, free %d Bytes\r\n",
      statfs_buf.f_bsize, statfs_buf.f_bfree);
}
```

aos_access

函数原型

int aos_access(const char *path, int amode);

详细说明

*aos_access(*检查当前程序是否可以访问*path*文件。*amode*指定访问权限，它的值可以是*F_OK*，或者*R_OK*、*W_OK*和*X_OK*的或“|”组合。*F_OK*用于测试文件是否存在，*R_OK*、*W_OK*和*X_OK*分别测试文件是否具有读、写、执行权限。

输入参数

参数类型	参数名称	参数描述
const char *	path	文件目录路径
int	amode	需查询的访问权限

返回参数

返回值类型	返回值	描述
int	0	允许该权限访问
负数	禁止该权限访问，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *path = "/sdcard/test.txt";
int ret = aos_access(path, F_OK);
if (!ret) {
  printf("file %s exists\r\n", path);
}
```

aos_chdir

函数原型

int aos_chdir(const char *path);

详细说明

*aos_chdir()*改变当前程序的工作目录到path路径。

要支持改变工作目录，*VFS*需要使能宏*CURRENT_WORKING_DIRECTORY_ENABLE*。

输入参数

参数类型	参数名称	参数描述
const char *	path	要变更的目标路径

返回参数

返回值类型	返回值	描述
int	0	当前工作目录变更成功
负数	当前工作目录变更失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *path = "/sdcard/dir";
int ret = aos_chdir(path);
if (ret) {
   printf("aos_chdir(%) failed, errno %d\r\n", path, ret);
} else {
   printf("aos_chdir(%) success\r\n", path);
}
```

aos_getcwd

函数原型

char *aos_getcwd(char *buf, size_t size);

详细说明

*aos_getcwd()*返回当当前程序的绝对工作目录字符串指针，同时如果*buf*参数不为空，也将当前工作目录字符串复制到*buf*中。参数*size*是*buf*的的字节数。如果当前的工作目录字符串长度超过*size*字节（包括字符串末尾的'\0'），那么返回*NULL*。

输入参数

参数类型	参数名称	参数描述
char *	buf	获取当前工作目录的缓存
size_t	size	缓存的长度

返回参数

返回值类型	返回值	描述
char *	非NULL	返回当前工作目录的指针
NULL	获取当前工作目录失败	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
char *cwd = NULL;
char buf[32];
int size = 32;
cwd = aos_getcwd(buf, size);
if (NULL != cwd) {
  printf("current work dir is %s\r\n", cwd);
}
```

aos_pathconf

函数原型

long aos_pathconf(const char *path, int name);

详细说明

*aos_pathconf()*函数返回配置文件的限制值，是与文件或目录相关联的运行时限制。*path*参数是限制值的路径，*name*是想得到限制值的名称，*name*的取值主要有以下几个取值：

限制名	说明	name参数
FILESIZEBITS	在指定目录中允许的普通文件最大长度所需的最少位数	_PC_FILESIZEBITS
LINK_MAX	文件链接数的最大值	_PC_LINK_MAX
MAX_CANON	终端规范输入队列的最大字节数	_PC_MAX_CANON
MAX_INPUT	终端输入队列可用空间的字节数	_PC_MAX_INPUT
NAME_MAX	文件名的最大字节数	_PC_NAME_MAX
PATH_MAX	相对路径名的最大字节数，包括null	_PC_PATH_MAX
PIPE_BUF	能原子的写到管道的最大字节数	_PC_PIPE_BUF
SYMLINK_MAX	符号链接中的字节数	_PC_SYMLINK_MAX

输入参数

参数名称	参数名称	参数描述
const char *	path	需获取配置信息的文件路径
int	name	文件配置类型名

返回参数

返回值类型	返回值	描述
long	正数	成功，返回配置文件的限制值
	负数	失败，返回错误码

返回文件配置类型值。

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *path = "/sdcard/test.txt";
long value = aos_pathconf(path, _PC_NAME_MAX);
if (value >= 0) {
  printf("name max is %d\r\n", value);
}
```

aos_fpathconf

函数原型

long aos_fpathconf(int fd, int name);

详细说明

*aos_fpathconf()*和*aos_pathconf()*类似，返回配置文件的限制值，是与文件或目录相关联的运行时限制。*aos_fpathconf()*的第一个参数是已打开文件的描述符*fd*，*name*是想得到限制值的名称，*name*的取值主要有以下几个取值：

限制名	说明	name参数
FILESIZEBITS	在指定目录中允许的普通文件最大长度所需的最少位数	_PC_FILESIZEBITS
LINK_MAX	文件链接数的最大值	_PC_LINK_MAX
MAX_CANON	终端规范输入队列的最大字节数	_PC_MAX_CANON
MAX_INPUT	终端输入队列可用空间的字节数	_PC_MAX_INPUT
NAME_MAX	文件名的最大字节数	_PC_NAME_MAX
PATH_MAX	相对路径名的最大字节数，包括null	_PC_PATH_MAX
PIPE_BUF	能原子的写到管道的最大字节数	_PC_PIPE_BUF
SYMLINK_MAX	符号链接中的字节数	_PC_SYMLINK_MAX

输入参数

参数类型	参数名称	参数描述
int	fd	需获取配置信息的文件描述符
int	name	文件配置类型名

返回参数

返回值类型	返回值	描述
long	正数	成功，返回配置文件的限制值
负数	失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
int name = PATH_MAX;
long value = aos_fpathconf(fd, name);
#include <stdio.h>
#include "aos/vfs.h"
const char *path = "/sdcard/test.txt";
int fd;
fd = aos_open(path, O_RDONLY);
if (fd > 0) {
  long value = aos_pathconf(path, _PC_NAME_MAX);
  if (value >= 0) {
    printf("name max is %d\r\n", value);
  }
}
aos_close(fd);
```

aos_utime

函数原型

int aos_utime(const char *path, const struct aos_utimbuf *times);

详细说明

*aos_utime()*通过参数*times*的*actime*和*modtime*成员改变文件的访问时间和修改时间。

```
struct aos_utimbuf {
  time_t actime; /**< time of last access */
  time_t modtime; /**< time of last modification */
};
```

输入参数

参数名称		参数描述
const char *	path	要设置访问修改时间的文件路径
const struct aos_utimbuf *	times	时间信息指针

返回参数

返回值类型	返回值	描述
int	0	时间修改成功

	负数	时间修改失败，返回错误码

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
struct aos_utimbuf time;
const char *path = "/sdcard/test.txt";
time.actime = 100;
time.modtime = 1000;
int ret = aos_utime(path, &time);
if (ret < 0) {
   printf("modify %s time failed, errno %d\r\n", path, ret);
}
```

aos_register_fs

函数原型

int aos_register_fs(const char *path, fs_ops_t *ops, void *arg);

详细说明

*aos_register_fs()*向系统注册文件系统，*path*新注册文件系统要挂载的点，*ops*是文件系统的操作结构体指针，*arg*是向文件系统传递的挂载参数。

输入参数

参数名称	参数名称	参数描述
const char *	path	文件系统挂载路径
fs_ops_t *	ops	文件系统的操作结构体指针
void *	arg	文件系统挂载参数

返回参数

返回值	返回值	描述
int	0	文件系统挂载成功
负数	文件系统挂载失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *path = "/spiffs";
static vfs_filesystem_ops_t spiffs_ops = {
  .open     = &spiffs_vfs_open,
  .close    = &spiffs_vfs_close,
  .read     = &spiffs_vfs_read,
  .write    = &spiffs_vfs_write,
  .lseek    = &spiffs_vfs_lseek,
  .sync     = &spiffs_vfs_sync,
  .stat     = &spiffs_vfs_stat,
  .unlink   = &spiffs_vfs_unlink,
  .rename   = &spiffs_vfs_rename,
  .opendir  = &spiffs_vfs_opendir,
  .readdir  = &spiffs_vfs_readdir,
  .closedir = &spiffs_vfs_closedir,
  .mkdir    = NULL,
  .ioctl    = NULL
};
int ret;
// mount spiffs firstly, add your mount logic here
// SPIFFS_mount(...);
// register spiffs filesystem
ret = aos_register_fs(path, &spiffs_ops, NULL);
if (ret < 0) {
  printf("register spiffs failed, errno %d\r\n", ret);
}
```

aos_unregister_fs

函数原型

int aos_unregister_fs(const char *path);

详细说明

*aos_unregister_fs()*将文件系统从对应的挂载点卸载掉，*path*是文件系统的挂载点路径。

输入参数

参数类型	参数名称	参数描述
const char *	path	要卸载的文件系统挂载路径

返回参数

返回值类型	返回值	描述
int	0	文件系统卸载成功
负数	文件系统卸载失败，返回错误码	

调用示例

```
#include <stdio.h>
#include "aos/vfs.h"
const char *path = "/spiffs";
int ret;
ret = aos_unregister_fs(path);
if (ret < 0) {
   printf("unregister fs %s failed, errno %d\r\n", path, ret);
}
```

标准宏和结构体说明

VFS_DEVICE_NODES

宏VFS_DEVICE_NODES用于配置VFS最大设备节点数。

```
#ifndef VFS_CONFIG_DEVICE_NODES
#define VFS_DEVICE_NODES 25
#else
#define VFS_DEVICE_NODES VFS_CONFIG_DEVICE_NODES
#endif
```

VFS_FD_OFFSET

VFS_FD_OFFSET用于配置VFS文件描述符初始偏移量。

```
#ifndef VFS_CONFIG_FD_OFFSET
#define VFS_FD_OFFSET 512
#else
#define VFS_FD_OFFSET VFS_CONFIG_FD_OFFSET
#endif
```

VFS_PATH_MAX

宏VFS_PATH_MAX用于配置VFS路径最大长度。

```
#ifndef VFS_CONFIG_PATH_MAX
#define VFS_PATH_MAX 256
#else
#define VFS_PATH_MAX VFS_CONFIG_PATH_MAX
#endif
```

VFS_MAX_FILE_NUM

宏VFS_MAX_FILE_NUM用于配置VFS支持最大文件打开数。

```
#ifndef VFS_CONFIG_MAX_FILE_NUM
#define VFS_MAX_FILE_NUM (VFS_DEVICE_NODES * 2)
#else
#define VFS_MAX_FILE_NUM VFS_CONFIG_MAX_FILE_NUM
#endif
```

CURRENT_WORKING_DIRECTORY_ENABLE

宏CURRENT_WORKING_DIRECTORY_ENABLE用于使能VFS记录当前程序的工作目录。

```
#ifndef VFS_CONFIG_CURRENT_DIRECTORY_ENABLE
#define CURRENT_WORKING_DIRECTORY_ENABLE 0
#else
#define CURRENT_WORKING_DIRECTORY_ENABLE VFS_CONFIG_CURRENT_DIRECTORY_ENABLE
#endif
```

struct aos_utimbuf

结构体aos_utimbuf用于获取文件的访问时间和修改时间。该结构体定义在include/aos/vfs.h中。

```
struct aos_utimbuf {
  time_t actime;  /* time of last access */
  time_t modtime; /* time of last modification */
};
```

struct aos_statfs

结构体aos_statfs用于获取文件系统的信息。该结构体定义在include/aos/vfs.h中。

```
struct aos_statfs {
  long f_type;   /* fs type */
  long f_bsize;  /* optimized transport block size */
  long f_blocks; /* total blocks */
  long f_bfree;  /* available blocks */
  long f_bavail; /* number of blocks that non-super users can acquire */
  long f_files;  /* total number of file nodes */
  long f_ffree;  /* available file nodes */
  long f_fsid;   /* fs id */
  long f_namelen; /* max file name length */
};
```

struct aos_stat

结构体aos_stat用于获取文件的信息。该结构体定义在include/aos/vfs.h中。

```
struct aos_stat {
  uint16_t st_mode;   /* mode of file */
  uint32_t st_size;   /* bytes of file */
  time_t   st_actime; /* time of last access */
  time_t   st_modtime; /* time of last modification */
};
```

aos_dirent_t

结构体aos_dirent_t用于描述目录文件。aos_readdir()成功返回后，返回一个指向aos_dirent_t的结构体指针，通过该指针访问到文件的类型、文件名。该结构体定义在include/aos/vfs.h中。

```
typedef struct {
  int32_t d_ino;   /* file number */
  uint8_t d_type;  /* type of file */
  char    d_name[]; /* file name */
} aos_dirent_t;
```

aos_dir_t

结构体aos_dir_t用于描述目录，aos_opendir()成功返回后，返回一个指向aos_dir_t的结构体指针。该结构体定义在include/aos/vfs.h中。

```
typedef struct {
  int32_t dd_vfs_fd;
  int32_t dd_rsv;
} aos_dir_t;
```

inode_ops_t

联合体inode_ops_t统一对inode的操作方法，一个inode要么是设备文件，要么是文件系统，将他们定义成一个联合体，可以节约内存。该结构体定义在include/aos/vfs.h中。

```
union inode_ops_t {
  const file_ops_t *i_ops;  /* char driver operations */
  const fs_ops_t   *i_fops; /* FS operations */
};
```

inode_t

结构体inode_t用于描述一个inode节点。一个文件、文件夹、设备在虚拟文件系统中被抽象成一个inode。该结构体定义在include/aos/vfs.h中。

```
typedef struct {
  union inode_ops_t  ops;    /* inode operations */
  void          *i_arg;  /* per inode private data */
  char          *i_name;  /* name of inode */
  int           i_flags; /* flags for inode */
  uint8_t       type;    /* type for inode */
  uint8_t       refs;    /* refs for inode */
} inode_t;
```

file_t

结构体file_t是虚拟文件系统的文件的抽象。该结构体定义在include/aos/vfs.h中。

```
typedef struct {
  inode_t *node;  /* node for file */
  void    *f_arg;  /* f_arg for file */
  size_t   offset; /* offset for file */
} file_t;
```

file_ops_t

定义file_ops_t结构。该结构体定义在include/aos/vfs.h中。

```
typedef const struct file_ops file_ops_t;
```

struct file_ops

结构体file_ops集成对文件操作的方法。该结构体定义在include/aos/vfs.h中。

```
struct file_ops {
  int    (*open)(inode_t *node, file_t *fp);
  int    (*close)(file_t *fp);
  ssize_t (*read)(file_t *fp, void *buf, size_t nbytes);
  ssize_t (*write)(file_t *fp, const void *buf, size_t nbytes);
  int    (*ioctl)(file_t *fp, int cmd, unsigned long arg);
  int    (*poll)(file_t *fp, int flag, poll_notify_t notify, void *fd, void *arg);
};
```

fs_ops_t

定义fs_ops_t结构。该结构体定义在include/aos/vfs.h中。

```
typedef const struct fs_ops   fs_ops_t;
```

struct fs_ops

结构体fs_ops集成对文件系统的操作方法。该结构体定义在include/aos/vfs.h中。

```
struct fs_ops {
  int       (*open)(file_t *fp, const char *path, int flags);
  int       (*close)(file_t *fp);
  ssize_t     (*read)(file_t *fp, char *buf, size_t len);
  ssize_t     (*write)(file_t *fp, const char *buf, size_t len);
  off_t      (*lseek)(file_t *fp, off_t off, int whence);
  int       (*sync)(file_t *fp);
  int       (*stat)(file_t *fp, const char *path, struct aos_stat *st);
  int       (*fstat)(file_t *fp, struct aos_stat *st);
  int       (*link)(file_t *fp, const char *path1, const char *path2);
  int       (*unlink)(file_t *fp, const char *path);
  int       (*remove)(file_t *fp, const char *path);
  int       (*rename)(file_t *fp, const char *oldpath, const char *newpath);
  aos_dir_t   *(*opendir)(file_t *fp, const char *path);
  aos_dirent_t *(*readdir)(file_t *fp, aos_dir_t *dir);
  int       (*closedir)(file_t *fp, aos_dir_t *dir);
  int       (*mkdir)(file_t *fp, const char *path);
  int       (*rmdir)(file_t *fp, const char *path);
  void       (*rewinddir)(file_t *fp, aos_dir_t *dir);
  long       (*telldir)(file_t *fp, aos_dir_t *dir);
  void       (*seekdir)(file_t *fp, aos_dir_t *dir, long loc);
  int       (*ioctl)(file_t *fp, int cmd, unsigned long arg);
  int       (*statfs)(file_t *fp, const char *path, struct aos_statfs *suf);
  int       (*access)(file_t *fp, const char *path, int amode);
  long       (*pathconf)(file_t *fp, const char *path, int name);
  long       (*fpathconf)(file_t *fp, int name);
  int       (*utime)(file_t *fp, const char *path, const struct aos_utimbuf *times);
};
```

配置说明

配置文件为./core/vfs/Config.in。进入menuconfig界面，如下所示：

```
-*- Kernel Core (rhino)
[ ] config micro kernel ----
-*- Initialize Function
[ ] Power Management ----
[ ] C++ Support
-*- Newlib (C-library) adaptation layer
[ ] Command-Line Interface ----
[ ] Coredump debug Support ----
-*- Key-value Storage --->
-*- Virtual File System --->
-*- AOS API Support --->
[ ] POSIX API Support
[ ] CMSIS API Support
```

再进入Virtual File System，配置以下参数：

```
--- Virtual File System
(25) The maximum number of VFS device nodes
(512) The default offset of VFS file descriptor
(256) The maximum length of device path (bytes)
(50) The maximum number of VFS files
[ ] Current Directory Recording Support
```

配置项

宏定义	默认值	描述
VFS_CONFIG_DEVICE_NODES	25	VFS最大设备节点数
VFS_CONFIG_FD_OFFSET	512	VFS文件描述符初始偏移量
VFS_CONFIG_PATH_MAX	256	VFS路径最大长度
VFS_CONFIG_MAX_FILE_NUM	50	VFS支持最大文件打开数
CURRENT_WORKING_DIRECTORY_ENABLE	n	使能VFS记录当前程序的工作目录

移植说明

代码目录结构

头文件vfs.h的内容是aos vfs对外的数据结构和API。

```
include/aos/vfs.h
```

vfs.c是aos vfs对外API的实现文件。

```
core/osal/aos/vfs.c
```

目录core/vfs/里面是vfs的核心文件，包括头文件和C文件。这些头文件是vfs内部使用或者core内部使用的头文件，应用开发不应当访问这些头文件。

```
core/vfs/
├── aos.mk
├── Config.in
├── include
│   ├── vfs_adapt.h
│   ├── vfs_api.h
│   ├── vfs_conf.h
│   └── vfs_types.h
├── vfs_adapt.c
├── vfs.c
├── vfs_file.c
├── vfs_file.h
├── vfs_inode.c
└── vfs_inode.h
```

2.4.4. OTA

OTA（over the air)已成为物联网设备的刚需功能, AliOS Things OTA有完备的升级方案，对各种升级场景都有很好的支持；除一般的整包升级外，现有的高阶能力有：压缩升级、差分升级及安全升级；支持的升级通道：http、https、BLE、3G/4G，NB等；复杂场景支持：网关及子设备升级，连接型模组升级非连接主设备的间接升级；完备的配套工具：差分工具、本地签名工具、ymodem辅助升级工具，多固件打包工具等；

API 列表

ota_service_init()	初始化OTA模块
ota_service_start()	固件下载，完成OTA固件的主要流程
ota_hal_init()	OTA hal层初始化，主要完成ota flash 下载分区的初始化等操作
ota_hal_write()	写固件到ota 存储区
ota_hal_read()	读取ota 存储区固件数据
ota_hal_boot()	设置保存bootload相关的固件及升级状态信息
ota_hal_rollback()	检测OTA升级是否升级成功
ota_hal_version()	获取固件版本号

使用

添加该组件

aos.mk 中引入：

```
$(NAME)_COMPONENTS += ota
```

Config.in 中引入：

```
config ENABLE_OTA
  bool "Enable OTA"
  default y
  help
    enable feature OTA.
```

包含头文件

```
#include "ota/ota_agent.h"
```

使用示例

ota服务接口使用请参考linkkit app或ota app

API 详情

OTA的API说明请参考include/dm/ota/ota_hal.h

ota_service_init()

初始化ota服务

函数原型

```
int ota_service_init(ota_service_t *ctx);
```

输入参数

ota_service_t *ctx	ota参数集	"devicename" , "devicesercet", "productname","productsecret"等

返回参数

返回参数宏定义见include/dm/ota/ota_agent.h文件

调用示例

```
ota_service_t ctx = {0};
char product_key[21] = "a1RIsMLz2BJ";
char product_secret[65] = "fSAF0hle6xL0oRWd";
char device_name[33] = "example1";
char device_secret[65] = "RDXf67itLqZCwdMCRrw0N5FHbv5D7jrE";
memset(&ctx, 0, sizeof(ota_service_t));
strncpy(ctx.pk, product_key, sizeof(ctx.pk) - 1);
strncpy(ctx.dn, device_name, sizeof(ctx.dn) - 1);
strncpy(ctx.ds, device_secret, sizeof(ctx.ds) - 1);
strncpy(ctx.ps, product_secret, sizeof(ctx.ps) - 1);
ctx.dev_type = 1;
ota_service_init(&ctx);
```

ota_service_start()

固件下载，完成OTA固件的主要流程

函数原型

```
int ota_service_start(ota_service_t *ctx);
```

输入参数

ota_service_t *ctx	ota参数集	"devicename" , "devicesercet", "productname","productsecret"等

返回参数

返回参数宏定义见include/dm/ota/ota_agent.h文件

调用示例

```
ota_service_t ctx = {0};
char product_key[21] = "a1RIsMLz2BJ";
char product_secret[65] = "fSAF0hle6xL0oRWd";
char device_name[33] = "example1";
char device_secret[65] = "RDXf67itLqZCwdMCRrw0N5FHbv5D7jrE";
memset(&ctx, 0, sizeof(ota_service_t));
strncpy(ctx.pk, product_key, sizeof(ctx.pk) - 1);
strncpy(ctx.dn, device_name, sizeof(ctx.dn) - 1);
strncpy(ctx.ds, device_secret, sizeof(ctx.ds) - 1);
strncpy(ctx.ps, product_secret, sizeof(ctx.ps) - 1);
ctx.dev_type = 1;
ota_service_start(&ctx);
```

ota_hal_init()

OTA hal层初始化，主要完成ota 下载分区flash擦除，待下载固件size判断等操作,此函数为弱符号函数，用户可自定义

函数原型

```
int ota_hal_init(ota_boot_param_t *parm);
```

输入参数

ota_boot_param_t *parm	boot 参数集	

返回参数

返回参数宏定义见include/dm/ota/ota_agent.h文件

调用示例

```
int ret = 0;
ota_boot_param_t param;
param.len = 512000;  /*500k*/
ret = ota_hal_init(&param);
```

ota_hal_write()

向ota 存储区写固件数据,此函数为弱符号函数，用户可自定义

函数原型

```
int ota_hal_write(unsigned int *off_set, char *in_buf, unsigned int in_buf_len);
```

输入参数

unsigned int *off_set	相对ota 存储flash起始地址偏移量	0
char *in_buf	保存固件数据	0xab,0xcd,0xef,0xgh
unsigned int in_buf_len	写入数据长度，单位为字节	4

返回参数

返回参数宏定义见include/dm/ota/ota_agent.h文件

调用示例

```
int ret = 0;
unsigned int offset = 0;
char buf[4] = {0xab, 0xcd, 0xef, ox12};
ret = ota_hal_write(&offset, buf, sizeof(buf));
```

ota_hal_read()

从ota 存储区读取固件数据,此函数为弱符号函数，用户可自定义

函数原型

```
int ota_hal_read(unsigned int *off_set, char *out_buf, unsigned int out_buf_len);
```

输入参数

unsigned int *off_set	相对ota 存储flash起始地址偏移量	0
char *out_buf	读取固件数据	0xab\0xcd\0xef
unsigned int out_buf_len	读取的长度，单位为字节	3

返回参数

返回参数宏定义见include/dm/ota/ota_agent.h文件

调用示例

```
int ret = 0;
unsigned int offset = 0;
char buf[4] = {0};
ret = ota_hal_read(&offset, buf, sizeof(buf));
```

ota_hal_boot()

保存bootload相关的固件及升级状态信息,此函数为弱符号函数，用户可自定义

函数原型

```
int ota_hal_boot(ota_boot_param_t *parm);
```

输入参数

ota_boot_param_t *parm	boot参数集	

返回参数

返回参数宏定义见include/dm/ota/ota_agent.h文件

调用示例

```
/* OTA upgrade magic <--> upg_flag */
#define OTA_UPGRADE_CUST   0x8778 /* upgrade user customize image */
#define OTA_UPGRADE_ALL    0x9669 /* upgrade all image: kernel+framework+app */
#define OTA_UPGRADE_XZ     0xA55A /* upgrade xz compressed image */
#define OTA_UPGRADE_DIFF   0xB44B /* upgrade diff compressed image */
int ret = 0;
ota_boot_param_t ota_info;
ota_info.len = 512000;  /*image size = 500k*/
ota_info.upg_flag = OTA_UPGRADE_ALL;
ret = ota_hal_boot(&ota_info);
```

ota_hal_rollback()

检测OTA升级是否升级成功,此函数为弱符号函数，用户可自定义

函数原型

```
int ota_hal_rollback(void);
```

输入参数

NULL

返回参数

返回参数宏定义见include/dm/ota/ota_agent.h文件

调用示例

```
int ret = 0;
ret = ota_hal_rollback();
```

ota_hal_version()

获取固件版本号,此函数为弱符号函数，用户可自定义

函数原型

```
const char *ota_hal_version(unsigned char dev_type, char *dn);
```

输入参数

unsigned char dev_type	设备类型	0(主设备)
char *dn	设备名称	"light"

返回参数

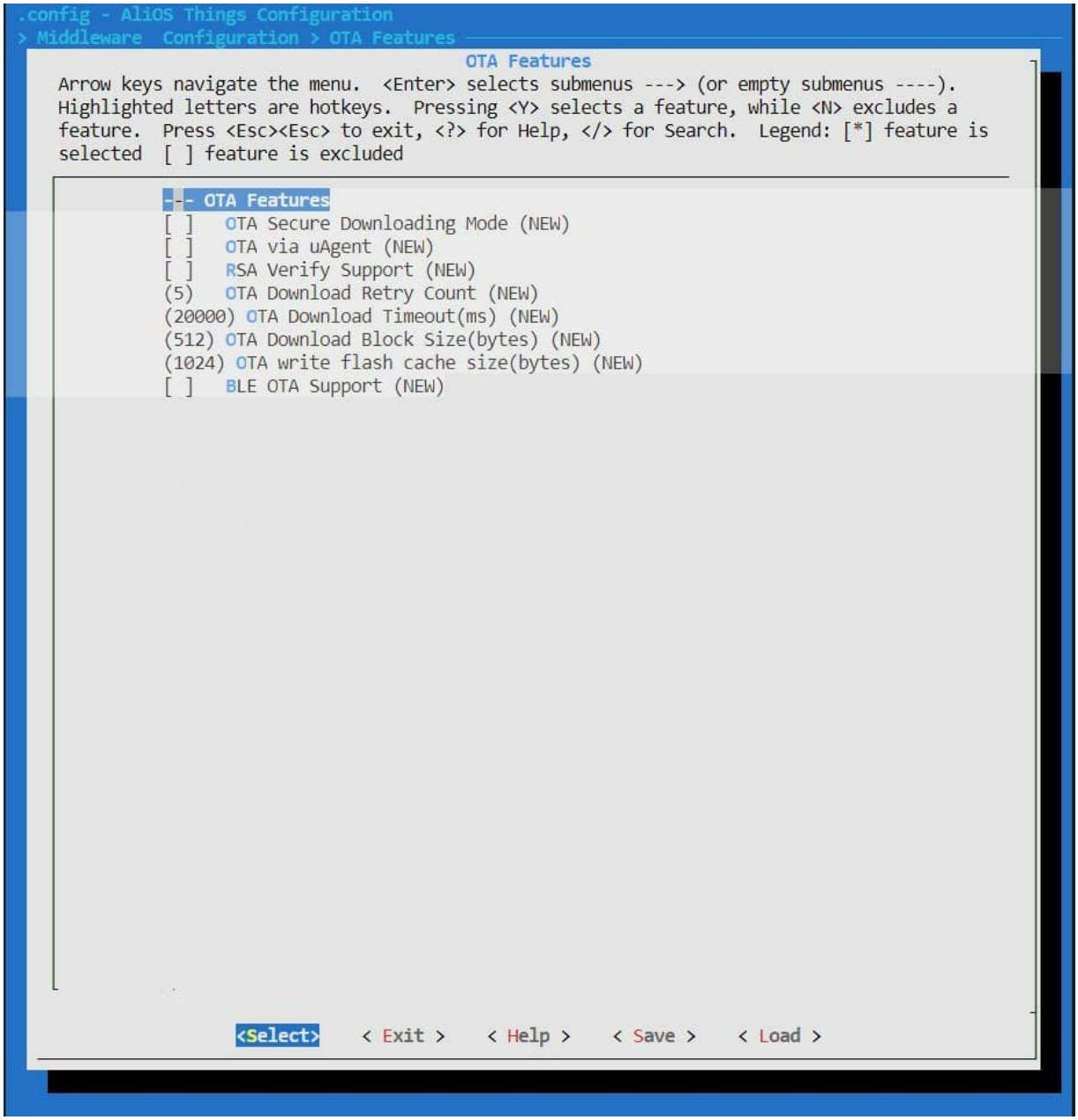
返回设备的版本号，如“app-1.0.0”

调用示例

```
const char *ver = NULL;
unsigned char dev_type = 0;
char *dev_name = "light"
ver = ota_hal_version(dev_type, dev_name);
```

配置说明

在AliOS Things 源码里面输入 `aos make menuconfig` 命令即可进入到menuconfig配置界面中，依次选择 `Middleware` -> `OTA Features` 即可进入到OTA组件的配置界面：



配置项说明

```
--- OTA Features
[ ] OTA Secure Downloading Mode (NEW)      #是否支持https 下载方式升级
[ ] OTA via uAgent (NEW)                   #是否支持uAgent方式升级
[ ] RSA Verify Support (NEW)               #是否支持安全升级
(5) OTA Download Retry Count (NEW)         #固件下载失败的重试次数
(20000) OTA Download Timeout(ms) (NEW)     # 固件下载超时时间
(512) OTA Download Block Size(bytes) (NEW) # OTA下载固件每次读取数据大小
(1024) OTA write flash cache size(bytes) (NEW) # OTA写flash缓存大小，默认是1k，可根据设备资源自定义size
[ ] BLE OTA Support (NEW)                  # 开启BLE OTA功
```

其他

返回参数定义

返回参数定义在include/dm/ota/ota_agent.h文件中。

```
/* OTA status code */
typedef enum {
    OTA_FINISH          = 4,       /*OTA finish status*/
    OTA_DOWNLOAD        = 3,  /*OTA download status*/
    OTA_TRANSPORT       = 2,    /*OTA transport status*/
    OTA_INIT            = 1,        /*OTA init status*/
    OTA_SUCCESS         = 0,
    OTA_INIT_FAIL       = -1,         /*OTA init failed.*/
    OTA_TRANSPORT_INT_FAIL   = -2,  /*OTA transport init failed.*/
    OTA_TRANSPORT_PAR_FAIL   = -3,  /*OTA transport parse failed.*/
    OTA_TRANSPORT_VER_FAIL   = -4,  /*OTA transport verion is too old.*/
    OTA_DOWNLOAD_INIT_FAIL   = -5,  /*OTA download init failed.*/
    OTA_DOWNLOAD_HEAD_FAIL   = -6,  /*OTA download header failed.*/
    OTA_DOWNLOAD_CON_FAIL    = -7,  /*OTA download connect failed.*/
    OTA_DOWNLOAD_REQ_FAIL    = -8,  /*OTA download request failed.*/
    OTA_DOWNLOAD_RECV_FAIL   = -9,  /*OTA download receive failed.*/
    OTA_VERIFY_MD5_FAIL     = -10, /*OTA verfiy MD5 failed.*/
    OTA_VERIFY_SHA2_FAIL    = -11, /*OTA verfiy SH256 failed.*/
    OTA_VERIFY_RSA_FAIL     = -12, /*OTA verfiy RSA failed.*/
    OTA_VERIFY_IMAGE_FAIL    = -13, /*OTA verfiy image failed.*/
    OTA_UPGRADE_WRITE_FAIL   = -14, /*OTA upgrade write failed.*/
    OTA_UPGRADE_PARAM_FAIL   = -15, /*OTA upgrade parameter failed.*/
    OTA_UPGRADE_FW_SIZE_FAIL = -16, /*OTA upgrade FW too big.*/
    OTA_UPGRADE_SET_BOOT_FAIL = -17, /*OTA upgrade set boot failed.*/
    OTA_CUSTOM_CALLBAK_FAIL  = -18, /*OTA custom callback failed.*/
    OTA_MCU_INIT_FAIL       = -19, /*OTA MCU init failed.*/
    OTA_MCU_VERSION_FAIL     = -20, /*OTA MCU version failed.*/
    OTA_MCU_NOT_READY       = -21, /*OTA MCU not ready.*/
    OTA_MCU_REBOOT_FAIL      = -22, /*OTA MCU fail to reboot.*/
    OTA_MCU_HEADER_FAIL      = -23, /*OTA MCU header error.*/
    OTA_MCU_UPGRADE_FAIL     = -24, /*OTA MCU upgrade fail.*/
    OTA_INVALID_PARAMETER    = -25, /*OTA INVALID PARAMETER.*/
} OTA_ERRNO_E;
```

2.4.5. 阿里云物联网平台连接

linkkit app是提供给使用C语言开发产品的设备厂商，设备商可以使用该SDK将产品接入到阿里云IoT平台，从而通过阿里云IoT平台对设备进行远程管理的例子程序.

设备商可以使用该例程开发诸如灯/插座之类的智能单品，也可以使用该例程开发具有子设备接入能力的网关，比如Zigbee网关。

2. 功能列表

2.1 云端连接模块

设备可使用MQTT连接阿里云IoT平台

设备可使用CoAP连接阿里云IoT平台

设备可使用HTTP/S连接阿里云IoT平台

设备可使用HTTP2连接阿里云IoT平台

MQTT连接服务器站点可配置

云端服务器站点动态选择

上报SDK版本号支持

iTLS支持

智能单品一型一密/动态注册四元组

2.2 设备物模型

设备属性/事件/服务支持

TLS静态集成/动态下拉支持

2.3 服务提供模块

设备OTA

WiFi配网支持

WiFi/以太网设备绑定token生成以及通告

设备reset支持

2.4 子设备管理框架

添加/删除子设备

禁用/解禁子设备

子设备物模型支持,包括子设备属性/服务/事件的代理

2.5 设备本地控制

设备本地上线通告

设备本地属性/事件数据订阅

设备本地控制命令接收与处理

子设备本地控制

3. 文档说明

- 设备接入Link Kit SDK

2.4.6. Socket组件文档

Socket 通常也称作"套接字"，是支持 TCP/IP 协议的网络通信应用的基本操作单元，可以用来实现网间不同虚拟机或不同计算机之间的通信。使用TCP/IP协议的应用程序通过在客户端和服务器各自创建一个 Socket ，然后通过操作各自的 Socket 就可以完成客户端和服务器的连接以及数据传输的任务了。

Socket 的本质是编程接口(API)，是对 TCP/IP 的封装。使开发者不需要面对复杂的 TCP/IP 协议族，只需要调用几个较简单的 Socket API 就可以完成网络通信了。

AliOS-Things提供了一整套类BSD Socket API的接口。

Socket API

使用socket api的应用需要包含如下头文件:

```
#include “network/network.h"
```

名称	作用
socket	创建一个 socket 套接字
bind	将端口号和 IP 地址绑定带指定套接字上
listen	开始监听
accept	接受连接请求
connect	建立连接
send	面向连接的发送数据（TCP）
recv	面向连接的接收数据（TCPtcp）
sendto	无连接的发送数据（UDP）
recvfrom	无连接的接收数据（UDP）
select	查询它的可读性、可写性及错误状态信息
closesocket	关闭 socket
shutdown	按设置关闭套接字
gethostbyname	通过域名获取主机的 IP 地址等信息
getsockname	获取本地主机的信息
getpeername	获取连接的远程主机的信息
ioctlsocket	设置套接字控制模式
setsockopt	设置socket属性
getsockopt	获取socket属性

TCP/UDP

要学用套接字编程，一定要了解 TCP/UDP 协议。TCP/UDP 协议工作在 TPC/IP 协议栈的传输层，如下图所示：

TCP（Transmission Control Protocol 传输控制协议）是一种面向连接的协议，使用该协议时，可以保证客户端和服务端的连接是可靠和安全的。使用 TCP 协议进行通信之前，通信双方必须先建立连接，然后再进行数据传输，通信结束后终止连接。

优点：能保证可靠性、稳定性。

适用场景：TCP适合用于端到端的通信，适用于对可靠性要求较高的服务。

基于 TCP 的 socket 编程流程如下图所示：

UDP（User Datagram Protocol 用户数据报协议）是一种非面向连接的协议，它不能保证网络连接的可靠性。 客户端发送数据之前并不会去服务器建立连接，而是直接将数据打包发送出去。当服务器接收数据时它也不向发送方提供确认信息，如果出现丢失包或重份包的情况，也不会向发送方发出差错报文。

优点：控制选项少，无须建立连接，从而使得数据传输过程中的延迟小、数据传输效率高。

适用场景：UDP适合对可靠性不高，或网络质量有保障，或对实时性要求较高的应用程序。

基于 UDP 的 socket 编程流程如下图所示：

使用

输入aos make menuconfig命令，进入如下界面：

选择TCP/IP Selection：

选择ENABLE AOS TCP/IP：

退出保存配置，就选择了LwIP模块。

API 详解

函数原型

socket

使用 socket 通信之前，通信双方都需要各自建立一个 socket。我们通过调用 socket 函数来创建一个 socket 套接字：

```
int socket(int domain, int type, int protocol)
```

函数参数

参数	描述
domain	协议域
type	类型
protocol	传输协议
返回	——
> = 0	成功，返回一个代表套接字描述符的整数
< 0	失败

domain 参数支持下列参数：

```
AF_INET   Ipv4
AF_INET6  Ipv6
AF_PACKET  PACKET
AF_UNSPEC 未指定
```

type 参数支持下列参数：

```
SOCK_DGRAM  长度固定的、无连接的不可靠的报文传递（UDP）
SOCK_RAM   IP 协议的数据报接口
SOCK_STREAM 有序、可靠、双向的面向连接字节流（TCP）
```

protocol 参数：

通常是 0，表示按给定的 domain 和 type 选择默认传输协议。在 AF_INET 通信域中套接字类型 SOCK_STREAM 的默认传输协议是 TCP。在 AF_INET 通信域中套接字类型 SOCK_DGRAM 的默认传输协议是 TCP。

当对同一 domian 和 type 支持多个协议时，可以使用 protocol 参数选择一个特定协议。

函数返回

返回一个 socket 描述符，它唯一标识一个 socket。这个 socket 描述符 跟文件描述符 一样，后续的操作都有用到它，比如，把它作为参数，通过它来进行一些读写操作等。

函数原型

bind

bind 函数用来将套接字与计算机上的一个端口号相绑定，进而在该端口监听服务请求，该函数的函数原型如下：

```
int bind(int s, const struct sockaddr *addr, socklen_t namelen)
```

函数参数

参数	描述
s	要绑定的 socket描述符
addr	一个指向含有本机 IP 地址和端口号等信息的 sockaddr 结构的指针
namelen	sockaddr 结构的长度
返回	——
0	成功
< 0	失败

sockaddr 结构体定义如下：

```
struct sockaddr {
 u8_t sa_len;
 u8_t sa_family;
 char sa_data[14];
};
```

在 IPv4 因特网域（AF_INET）中，我们使用 sockaddr_in 结构体来代替 sockaddr 结构体：

```
struct sockaddr_in {
 u8_t        sin_len;
 sa_family_t    sin_family;
 in_port_t      sin_port;
 struct in_addr  sin_addr;
 char        sin_zero[SIN_ZERO_LEN];
};
```

其中，

- sin_family 一般固定写 AF_INET；
- sin_port 为套接字的端口号；
- sin_addr 为套接字的 IP 地址，
- sin_zero 通常全为 0，主要功能是为了与 sockaddr 结构在长度上保持一致。这样指向 sockaddr_in 的指针和指向 sockaddr 的指针可以互相转换。

一般情况下，可以将 sin_port 设为 0，这样系统会随机选择一个未被占用的端口号。同样，sin_addr 设为 INADDR_ANY，系统会自动填入本机的 IP 地址。

注意事项

当调用 bind 函数时，不要将端口号设为小于 1024 的值，因为 1-1024 为系统的保留端口号，我们可以选择大于 1024 的任何一个未被占用的端口号。

函数原型

listen

listen 函数用来将套接字设为监听模式，并在套接字指定的端口上开始监听，以便对到达的服务请求进行处理。该函数的函数原型如下：

```
int listen(int s, int backlog)
```

函数参数

参数	描述
s	绑定后的 socket描述符
backlog	连接请求队列可以容纳的最大数目
返回	——
0	成功
< 0	失败

函数原型

accept

accept 函数用来从完全建立的连接的队列中接受一个连接，该函数的函数原型如下：

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen)
```

函数参数

参数	描述
s	被监听的 socket 描述符
addr	一个指向 scokaddr_in 结构的指针，存放提出连接请求的主机 IP 地址和端口号等信息
addrlen	一个指向 socklen_t 的指针，用来存放 sockaddr_in 结构的长度
返回	——
> = 0	成功，返回新创建的套接字描述符
< 0	失败

服务端接受连接后，accept 函数会返回一个新的 socket 描述符，线程可以使用这个新的描述符同客户端传输数据。

函数原型

connect

connect 函数用来与服务器建立一个 TCP 连接，该函数的函数原型如下：

```
int connect(int s, const struct sockaddr *name, socklen_t namelen)
```

函数参数

参数	描述
s	socket 描述符
name	指向 sockaddr 结构的指针，存放要连接的服务器的 IP 地址和端口号等信息
namelen	sockaddr 结构体的长度
返回	——
> = 0	成功，返回新创建的套接字描述符
< 0	失败

函数原型

send

send 函数用来在面向连接的数据流 socket 模式下发送数据，该函数的函数原型如下：

```
int send(int s, const void *dataptr, size_t size, int flags)
```

函数参数

参数	描述
s	socket 描述符
dataptr	指向所要发送的数据区的指针
size	要发送的字节数
flags	控制选项，通常为 0
返回	——
>0	成功，返回发送的数据的长度
<=0	失败

如果返回值小于 size的话，你需要再次发送剩下的数据。

函数原型

recv

recv 函数用来在面向连接的数据流 socket 模式下接收数据，该函数的函数原型如下：

```
int recv(int s, void *mem, size_t len, int flags)
```

函数参数

参数	描述
s	socket 描述符
mem	指向存储数据的内存缓存区的指针
len	缓冲区的长度
flags	控制选项，通常为 0
返回	——
> 0	成功，返回接收的数据的长度
= 0	目标地址已传输完并关闭连接
< 0	失败

函数原型

sendto

sendto 函数用来在无连接的数据报 socket 模式下发送数据，该函数的函数原型如下：

```
int sendto(int s, const void *dataptr, size_t size, int flags, const struct sockaddr *to, socklen_t tolen)
```

函数参数

参数	描述
s	socket 描述符
dataptr	指向所要发送的数据区的指针
size	要发送的字节数
flags	控制选项，通常为 0
to	指向 sockaddr 结构体的指针，存放目的主机的 IP 和端口号
tolen	sockaddr 结构体的长度
返回	——
> 0	成功，返回发送的数据的长度
< = 0	失败

函数原型

recvfrom

recvfrom函数用来在无连接的数据报 socket 模式下接收数据，该函数的函数原型如下：

```
int recvfrom(int s, void*mem, size_t size, int flags, struct sockaddr *from, socklen_t *fromlen)
```

函数参数

参数	描述
s	socket 描述符
mem	指向存储数据的内存缓存区的指针
size	缓冲区的长度
flags	控制选项，通常为 0
from	指向 sockaddr 结构体的指针，存放源主机的 IP 和端口号
fromlen	指向 sockaddr 结构体的长度的指针
返回	——
> 0	成功，返回接收的数据的长度
= 0	目标地址已传输完并关闭连接
< 0	失败

函数原型

select

select 函数用来查询一个或者多个socket的可读性、可写性及错误状态信息，该函数的函数原型如下：

```
select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
        struct timeval *timeout);
```

函数参数

参数	描述
maxfdp1	最大的文件描述符
readset	读文件描述符
writeset	写文件描述符
exceptset	异常的文件描述符
timeout	超时时间
返回	——
> 0	成功，返回发送的数据的长度
< = 0	失败

函数原型

closesocket

closesocket 在传输完数据之后关闭 socket 并释放资源的函数，该函数的函数原型如下：

```
int closesocket(int s)
```

函数参数

参数	描述
s	socket 描述符
返回	——
0	成功
< 0	失败

函数原型

shutdown

shutdown 允许进行单向的关闭操作，或是全部禁止掉，该函数的函数原型如下：

```
int shutdown(int s, int how)
```

函数参数

参数	描述
s	socket 描述符
how	控制选项
返回	——
0	成功
< 0	失败

how 参数支持下列参数：

```
SHUT_RD   关闭接收信道
SHUT_WR   关闭发送信道
SHUT_RDWR  将发送和接收信道全部关闭
```

函数返回

返回 0 表示成功

函数原型

gethostbyname

此函数可以通过域名来获取主机的 IP 地址等信息，该函数的函数原型如下：

```
struct hostent* gethostbyname(const char*name)
```

函数参数

参数	描述
name	主机域名
返回	——
> 0	成功，返回一个 hostent 结构体指针
< 0	失败

name 可以是具体域名，如：“www.aliyun.com”，也可以是 IP 地址，如：“192.168.8.6”

hostent 结构体定义如下：

```
struct hostent {
  char *h_name;    /* 主机正式域名 */
  char **h_aliases;  /* 主机的别名数组 */
  int   h_addrtype; /* 协议类型，对于 TCP/IP 为 AF_INET  */
  int   h_length;   /* 协议的字节长度，对于 IPv4 为 4 个字节 */
  char **h_addr_list; /* 地址的列表*/
#define h_addr h_addr_list[0] /* 保持向后兼容 */
};
```

函数原型

getsockname

此函数可以获取本地主机的信息，该函数的函数原型如下：

```
int getsockname(int s, struct sockaddr *name, socklen_t *namelen)
```

函数参数

参数	描述
s	socket 描述符
name	sockaddr 结构体指针，用来存储得到的主机信息
namelen	指向 sockaddr 结构体的长度的指针
返回	——
0	成功
< 0	失败

函数原型

getpeername

此函数可以得到与本地主机连接的远程主机的信息，该函数的函数原型如下：

```
int getpeername(int s, struct sockaddr *name, socklen_t *namelen)
```

函数参数

参数	描述
s	socket 描述符
name	sockaddr 结构体指针，用来存储得到的主机信息
namelen	指向 sockaddr 结构体的长度的指针
返回	——
0	成功
< 0	失败

函数原型

ioctlsocket

设置套接字控制模式，该函数的函数原型如下：

```
int ioctlsocket(int s, long cmd, void *argp)
```

函数参数

参数	描述
s	socket 描述符
cmd	套接字操作命令
argp	操作命令所带参数
返回	——
0	成功
< 0	失败

cmd 参数支持下列参数：

```
FIONBIO 开启或关闭套接字的非阻塞模式，arg 参数为 1 开启非阻塞，为 0 关闭非阻塞。
```

函数原型

getsockopt

设置套接字控制模式，该函数的函数原型如下：

```
int getsockopt (int s, int level, int optname, void *optval, socklen_t *optlen)
```

函数参数

参数	描述
s	socket 描述符

参数	描述
level	选项定义的层次；目前支持SOL_SOCKET, SOL_PACKET, IPPROTO_IP和IPPROTO_TCP
optname	需设置的选项
optval	指向option属性的指针
optlen	指向option属性长度的指针
返回	——
0	成功
< 0	失败

optname参数支持下列参数：

```
/*
 * SOL_SOCKET options
 */
/*
 * Option flags per-socket. These must match the SOF_ flags in ip.h (checked in init.c)
 */
#define SO_REUSEADDR  0x0004 /* Allow local address reuse */
#define SO_KEEPALIVE  0x0008 /* keep connections alive */
#define SO_BROADCAST  0x0020 /* permit to send and to receive broadcast messages (see IP_SOF_BROADCAST op
tion) */
#define SO_TCPSACK    0x0040 /* Allow TCP SACK (Selective acknowledgment) */
/*
 * Additional options, not kept in so_options.
 */
#define SO_DEBUG      0x0001 /* Unimplemented: turn on debugging info recording */
#define SO_ACCEPTCONN 0x0002 /* socket has had listen() */
#define SO_DONTROUTE  0x0010 /* Unimplemented: just use interface addresses */
#define SO_USELOOPBACK 0x0080 /* Unimplemented: bypass hardware when possible */
#define SO_LINGER     0x0100 /* linger on close if data present */
#define SO_DONTLINGER ((int)(~SO_LINGER))
#define SO_OOBINLINE  0x0200 /* Unimplemented: leave received OOB data in line */
#define SO_REUSEPORT  0x0400 /* Unimplemented: allow local address & port reuse */
#define SO_SNDBUF     0x1001 /* Unimplemented: send buffer size */
#define SO_RCVBUF     0x1002 /* receive buffer size */
#define SO_SNDLOWAT   0x1003 /* Unimplemented: send low-water mark */
#define SO_RCVLOWAT   0x1004 /* Unimplemented: receive low-water mark */
#define SO_SNDTIMEO   0x1005 /* send timeout */
#define SO_RCVTIMEO   0x1006 /* receive timeout */
#define SO_ERROR      0x1007 /* get error status and clear */
#define SO_TYPE       0x1008 /* get socket type */
#define SO_CONTIMEO   0x1009 /* Unimplemented: connect timeout */
#define SO_NO_CHECK   0x100a /* don't create UDP checksum */
#define SO_BIO        0x100b /* set socket into blocking mode */
#define SO_NONBLOCK   0x100c /* set/get blocking mode via optval param */
#define SO_NBIO       0x100d /* set socket into NON-blocking mode */
/*
 * SOL_PACKET options
 */
#define PACKET_RECV_OUTPUT          3
/*
 * Options for level IPPROTO_IP
 */
#define IP_TOS          1
#define IP_TTL          2
/*
 * Options for level IPPROTO_TCP
 */
#define TCP_NODELAY   0x01   /* don't delay send to coalesce packets */
#define TCP_KEEPALIVE 0x02   /* send KEEPALIVE probes when idle for pcb->keep_idle milliseconds */
#define TCP_KEEPIDLE  0x03   /* set pcb->keep_idle  - Same as TCP_KEEPALIVE, but use seconds for get/setsockopt
*/
#define TCP_KEEPINTVL 0x04   /* set pcb->keep_intvl - Use seconds for get/setsockopt */
#define TCP_KEEPCNT   0x05   /* set pcb->keep_cnt   - Use number of probes sent for get/setsockopt */
```

函数原型

setsockopt

设置套接字控制模式，该函数的函数原型如下：

```
int setsockopt (int s, int level, int optname, const void *optval, socklen_t optlen)
```

函数参数

参数	描述
s	socket 描述符
level	选项定义的层次；目前支持SOL_SOCKET, SOL_PACKET, IPPROTO_IP和IPPROTO_TCP
optname	需设置的选项
optval	指向option属性的指针
optlen	option属性的长度
返回	——
0	成功
< 0	失败

optname参数支持下列参数：

```
/*
 * SOL_SOCKET options
 */
/*
 * Option flags per-socket. These must match the SOF_ flags in ip.h (checked in init.c)
 */
#define SO_REUSEADDR  0x0004 /* Allow local address reuse */
#define SO_KEEPALIVE  0x0008 /* keep connections alive */
#define SO_BROADCAST  0x0020 /* permit to send and to receive broadcast messages (see IP_SOF_BROADCAST op
tion) */
#define SO_TCPSACK    0x0040 /* Allow TCP SACK (Selective acknowledgment) */
/*
 * Additional options, not kept in so_options.
 */
#define SO_DEBUG      0x0001 /* Unimplemented: turn on debugging info recording */
#define SO_ACCEPTCONN 0x0002 /* socket has had listen() */
#define SO_DONTROUTE  0x0010 /* Unimplemented: just use interface addresses */
#define SO_USELOOPBACK 0x0080 /* Unimplemented: bypass hardware when possible */
#define SO_LINGER     0x0100 /* linger on close if data present */
#define SO_DONTLINGER ((int)(~SO_LINGER))
#define SO_OOBINLINE  0x0200 /* Unimplemented: leave received OOB data in line */
#define SO_REUSEPORT  0x0400 /* Unimplemented: allow local address & port reuse */
#define SO_SNDBUF     0x1001 /* Unimplemented: send buffer size */
#define SO_RCVBUF     0x1002 /* receive buffer size */
#define SO_SNDLOWAT   0x1003 /* Unimplemented: send low-water mark */
#define SO_RCVLOWAT   0x1004 /* Unimplemented: receive low-water mark */
#define SO_SNDTIMEO   0x1005 /* send timeout */
#define SO_RCVTIMEO   0x1006 /* receive timeout */
#define SO_ERROR      0x1007 /* get error status and clear */
#define SO_TYPE       0x1008 /* get socket type */
#define SO_CONTIMEO   0x1009 /* Unimplemented: connect timeout */
#define SO_NO_CHECK   0x100a /* don't create UDP checksum */
#define SO_BIO        0x100b /* set socket into blocking mode */
#define SO_NONBLOCK   0x100c /* set/get blocking mode via optval param */
#define SO_NBIO       0x100d /* set socket into NON-blocking mode */
/*
 * SOL_PACKET options
 */
#define PACKET_RECV_OUTPUT          3
/*
 * Options for level IPPROTO_IP
 */
#define IP_TOS         1
#define IP_TTL         2
/*
 * Options for level IPPROTO_TCP
 */
#define TCP_NODELAY   0x01   /* don't delay send to coalesce packets */
#define TCP_KEEPALIVE 0x02   /* send KEEPALIVE probes when idle for pcb->keep_idle milliseconds */
#define TCP_KEEPIDLE  0x03   /* set pcb->keep_idle  - Same as TCP_KEEPALIVE, but use seconds for get/setsockopt
*/
#define TCP_KEEPINTVL 0x04   /* set pcb->keep_intvl - Use seconds for get/setsockopt */
#define TCP_KEEPCNT   0x05   /* set pcb->keep_cnt   - Use number of probes sent for get/setsockopt */
```

注意事项

在网络中都采用大端字节序，但是不同的嵌入式系统，其字节序不一定都是大端格式，相反小端字节序倒是很常见，比如 STM32。我们在设置 IP 和端口号时，要根据自己的平台特点进行必要的字节序转换。

下面给出套接字字节转换函数的列表：

```
htons() —— "Host to Network Short" 主机字节顺序转换为网络字节顺序
htonl() —— "Host to Network Long" 主机字节顺序转换为网络字节顺序
ntohs() —— "Network to Host Short" 网络字节顺序转换为主机字节顺序
ntohl() —— "Network to Host Long" 网络字节顺序转换为主机字节顺序
```

对于一个“192.168.2.1”这种字符串形式的 IP 地址，我们如何将其正确的转换为网络字节序呢？

可以使用 inet_addr（“192.168.2.1”），结果直接就是网络字节序了；

我们也可以使用 inet_ntoa()（“ntoa”代表“Network to ASCII”）函数将一个长整形的 IP 地址转换为一个字符串。

Socket编程范例

TCP Client

```
#include "network/network.h"
#define SERVER_PORT 6666
#define BUFFER_SIZE 1024
char send_buf[BUFFER_SIZE] = "client hello";
char recv_buf[BUFFER_SIZE] = {0};
int tcp_client(void)
{
  int sockfd = -1;
  struct sockaddr_in server;
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0) {
    printf("sock create failed\n");
    return -1;
  }
  memset(&server, 0, sizeof(struct sockaddr_in));
  server.sin_family = AF_INET;
  server.sin_port = htons(SERVER_PORT);
  server.sin_addr.s_addr = inet_addr( "127.0.0.1" );
  if(connect(sockfd, (struct sockaddr*)&server, sizeof(server)) < 0) {
    printf("sock connect failed\n");
    close(sockfd);
    return -1;
  }
  send(sockfd, send_buf, sizeof(send_buf), 0);
  printf("send data:%s\n", send_buf);
  recv(sockfd, recv_buf, sizeof(recv_buf ), 0);
  printf("recv data:%s\n", recv_buf);
  close(sockfd);
  return 0;
}
```

TCP Server

```
#include "network/network.h"
#define SERVER_PORT 6666
#define BUFFER_SIZE 1024
#define SOMAXCONN 2
```

```
char recv_buf[BUFFER_SIZE];
int tcp_server(void)
{
  int sockfd = -1;
  int acceptfd = -1;
  struct sockaddr_in server;
  struct sockaddr_in peerServer;
  int recv_bytes;
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0) {
    printf("sock create failed\n");
    return -1;
  }
  memset(&server, 0, sizeof(struct sockaddr_in));
  server.sin_family = AF_INET;
  server.sin_port = htons(SERVER_PORT);
  server.sin_addr.s_addr = htonl(INADDR_ANY);
  if(bind(sockfd, (struct sockaddr*)&server, sizeof(server)) < 0) {
    printf("sock bind failed\n");
    close(sockfd);
    return -1;
  }
  if(listen(sockfd, SOMAXCONN) < 0) {
    close(sockfd);
    printf("sock listen failed\n");
    return -1;
  }
  socklen_t len = sizeof(peerServer);
  acceptfd = accept(sockfd, (struct sockaddr*)&peerServer, &len);
  if (acceptfd < 0) {
    printf("sock accept failed\n");
    close(sockfd);
    return -1;
  }
  while(1) {
    memset(recv_buf, 0, sizeof(recv_buf));
    if(recv_bytes = recv(acceptfd, recv_buf, sizeof(recv_buf), 0) < 0)
    {
      if((EINTR == errno) || (EAGAIN == errno) || (EWOULDBLOCK == errno))
      {
        continue;
      }
      printf("Recv data failed\n");
      close(sockfd);
      close(acceptfd);
      return -1;
    }
    printf("recv data:%s len:%d\n", recv_buf, recv_bytes);
    send(acceptfd,recv_buf, recv_bytes, 0);
  }
  close(sockfd);
  close(acceptfd);
  return 0;
}
```

UDP Client

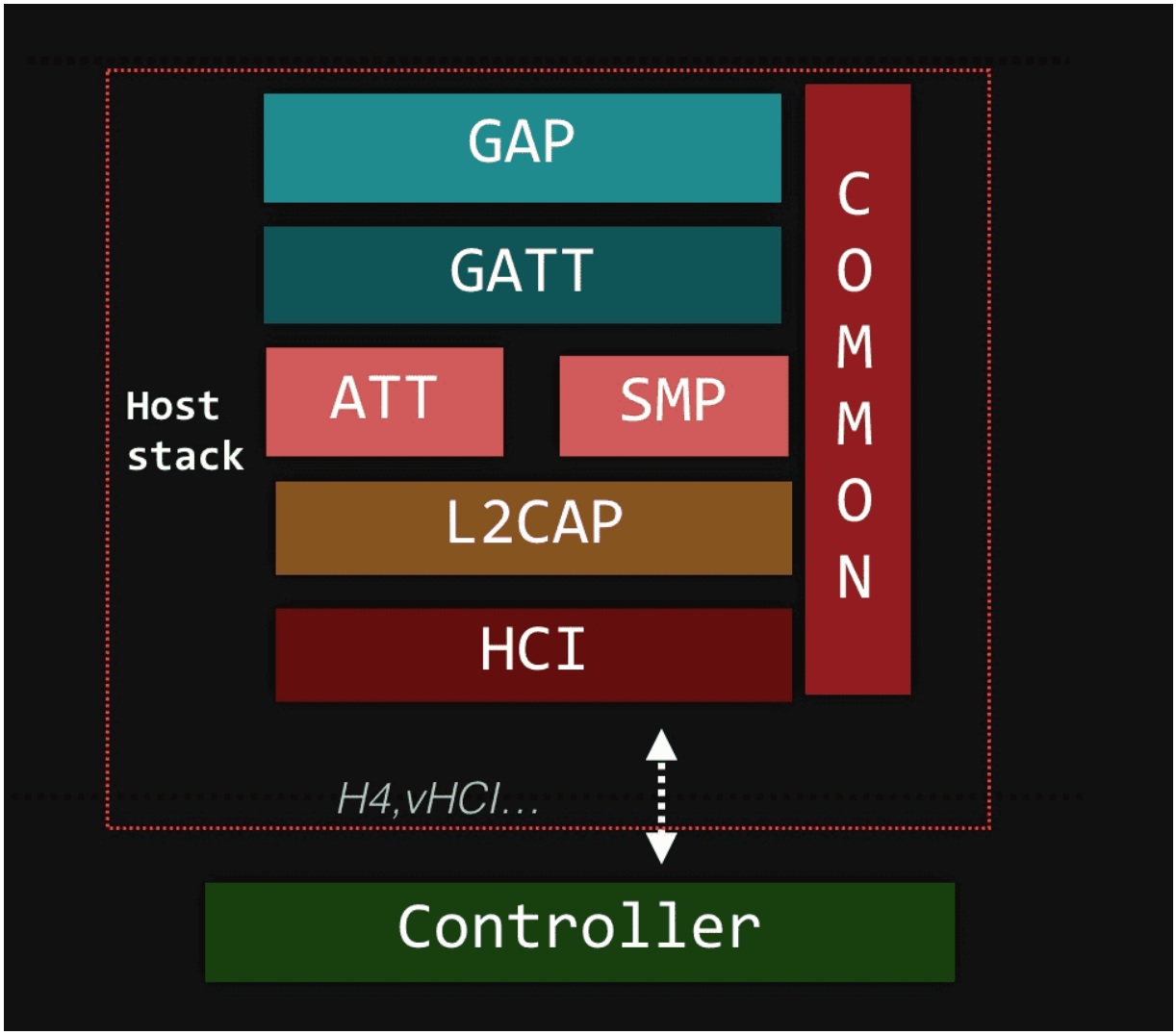
```
#include "network/network.h"
#define SERVER_PORT 8000
#define BUFFER_SIZE 1024
char send_buffer[BUFFER_SIZE] = "client hello";
int udp_client(void)
{
  struct sockaddr_in server_addr;
  int sockfd;
  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family = AF_INET;
  server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
  server_addr.sin_port = htons(SERVER_PORT);
  sockfd = socket(AF_INET, SOCK_DGRAM, 0);
  if(sockfd < 0)
  {
    printf("Create socket failed\n");
    close(sockfd);
    return -1;
  }
  if(sendto(sockfd, send_buffer, BUFFER_SIZE, 0, (struct sockaddr*)&server_addr,sizeof(server_addr)) < 0)
  {
    printf("Send data failed\n");
    close(sockfd);
    return -1;
  }
  close(sockfd);
  return 0;
}
```

UDP Server

```
#include "netowrk/network.h"
#define SERVER_PORT 8000
#define BUFFER_SIZE 1024
char recv_buffer[BUFFER_SIZE];
int udp_server(void)
{
  struct sockaddr_in server_addr;
  int sockfd;
  struct sockaddr_in client_addr;
  socklen_t fromlen;
  int recv_bytes;
  bzero(&server_addr, sizeof(server_addr));
  server_addr.sin_family = AF_INET;
  server_addr.sin_port = htons(SERVER_PORT);
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  sockfd = socket(AF_INET, SOCK_DGRAM, 0);
  if(sockfd < 0)
  {
    printf("Create socket failed\n");
    return -1;
  }
  if(bind(sockfd,(struct sockaddr*)&server_addr,sizeof(server_addr)) < 0)
  {
    printf("Server bind failed\n");
    close(sockfd);
    return -1;
  }
  fromlen = sizeof(client_addr);
  memset(recv_buffer, 0, BUFFER_SIZE);
  if((recv_bytes = recvfrom(sockfd, recv_buffer, BUFFER_SIZE, 0,(struct sockaddr*)&client_addr, &fromlen)) < 0)
  {
    if((EINTR == errno) || (EAGAIN == errno) || (EWOULDBLOCK == errno))
    {
      continue;
    }
    printf("Receive Data Failed\n");
    close(sockfd);
    return -1;
  }
  printf("Recv data:%s len:%d", recv_buffer, recv_bytes);
  close(sockfd);
  return 0;
}
```

2.4.7. BLE host组件文档

AliOS Things提供支持符合Bluetooth 4.0/4.2/5.0核心协议规范的BLE host软件协议栈，功能框图如下图红色部分：



主要支持功能有：

- Generic Access Profile(GAP)多角色支持
  - Peripheral&Central
  - Observer&Broadcaster
- Generic Attribute Profile(GATT)多连接支持
  - GATT client
  - GATT server
- Security Manager(SM)支持
  - Legacy Pairing
  - 多安全等级设定Security Level 1, 2, 3, 4
  - 安全连接Security Connection
  - LE Privacy(RPA地址生成)
- HCI接口支持
  - 标准HCI接口，支持host-only，host通过HCI和controller对接(如MCU+controller)
  - 虚拟HCI接口，支持host+controller，适合SOC的硬件平台(如nrf52832)

API列表

bt_enable	BLE协议栈初始化
bt_le_adv_start	BLE 广播开始
bt_le_adv_stop	BLE 广播停止
bt_le_scan_start	BLE scan开始
bt_le_scan_stop	BLE scan停止
bt_gatt_service_register	GATT服务注册接口
bt_gatt_service_unregister	GATT服务注销
bt_gatt_foreach_attr	GATT在指定handle范围内服务搜索
bt_gatt_attr_next	GATT获取下一个attribute接口
bt_gatt_attr_read	GATT读取attribute接口
bt_gatt_attr_read_service	GATT读取attribute包含服务接口
bt_gatt_attr_read_included	GATT读取包含属性接口
bt_gatt_attr_read_chrc	GATT读取包含characteristic接口
bt_gatt_attr_read_ccc	GATT读Client Configuration Characteristic接口
bt_gatt_attr_write_ccc	GATT写Client Configuration Characteristic接口
bt_gatt_attr_read_cep	GATT读写CEP attribute
bt_gatt_attr_read_cud	GATT读写CUD attribute
bt_gatt_attr_read_cpf	GATT读写CPF attribute
bt_gatt_notify	GATT发送notify接口
bt_gatt_indicate	GATT发送indicate接口
bt_gatt_discover	GATT client发现服务接口
bt_gatt_read	GATT client读GATT server属性
bt_gatt_write	GATT client写GATT server属性
bt_gatt_write_without_response	GATT client写GATT server，不需回复
bt_gatt_subscribe	GATT client订阅notification
bt_gatt_unsubscribe	GATT client取消订阅notification
bt_conn_security	设置security安全等级
bt_conn_enc_key_size	获取加密key大小

bt_conn_cb_register	注册连接相关回调函数，用以返回连接各状态
bt_conn_auth_cb_register	注册认证相关函数回调函数
bt_conn_auth_passkey_entry	passkey回复
bt_conn_auth_cancel	取消认证配对
bt_conn_auth_passkey_confirm	passkey匹配回复函数
bt_conn_auth_pairing_confirm	pairing请求回复函数
bt_conn_auth_pincode_entry	PIN code回复函数

使用

添加该组件

aos.mk中引入

```
$(NAME)_COMPONENTS += bt_host
```

包含头文件

对外头文件代码位于include/wireless/bluetooth/bluetooth,在应用的文件中添加头文件

```
#include <bluetooth/bluetooth.h>
#include <bluetooth/conn.h>
```

使用示例

- 蓝牙广播示例，参考 `application/example/example_legacy/bluetooth/bleadv` ,演示了如何开启协议栈，并广播特定数据的示例。
- 蓝牙做GATT server，参考 `application/example/example_legacy/bluetooth/bleperipheral` ，演示了GATT注册服务（DIS, HRS, BAS），并在服务发现之后推送模拟的应用数据至GATT client端。

API详情

bt_enable

BLE模块初始化，包含协议栈初始化和HCI driver初始化。

函数原型

```
int bt_enable(bt_ready_cb_t cb)
```

输入参数

cb	bt_ready_cb_t	初始化结束回调函数。

返回参数

`0`：成功， `其他值` ：失败。

调用示例

```
static void bt_ready(int err)
{
  if (err) {
     printf("BLE not ready (err %d)", err);
     return;
   }
 //协议栈异步初始化之后的回调函数,可以在这里增加用户逻辑，如开启广播等。
 //...
}
void ble_sample(void)
{
   int err = bt_enable(bt_ready);
   if (err) {
     printf("Bluetooth init failed (err %d)\n", err);
     return;
   }
}
```

bt_le_adv_start

开启蓝牙BLE广播。

函数原型

```
int bt_le_adv_start(const struct bt_le_adv_param *param,
            const struct bt_data *ad, size_t ad_len,
            const struct bt_data *sd, size_t sd_len)
```

输入参数

param	struct bt_le_adv_param*	广播参数如广播interval、广播地址等。
ad	struct bt_data*	广播数据。
ad_len	size_t	广播数据长度。
sd	struct bt_data*	Scan response 数据。
sd_len	size_t	Scan response 数据长度。

返回参数

`0`：成功，`其他值`：失败。

调用示例

```
static void bt_ready(int err)
{
  if (err) {
    printf("Bluetooth init failed (err %d)\n", err);
    return;
  }
  /*开启蓝牙广播具体步骤*/
  /* 1.广播参数设置，包含interval,地址等*/
  bt_addr_t addr = {.val = { 0x80, 0x81, 0x82, 0x83, 0x84, 0x05 }};
  struct bt_le_adv_param adv_param = {
    .options = 0, \
    .interval_min = BT_GAP_ADV_FAST_INT_MIN_2, \
    .interval_max = BT_GAP_ADV_FAST_INT_MAX_2, \
    .own_addr = &addr, \
  };
 /* 2.广播payload填充*/
  const struct bt_data adv_data[] = {
    BT_DATA(BT_DATA_FLAGS, data, 1),
    BT_DATA(BT_DATA_NAME_COMPLETE, adv_name, strlen(adv_name))
  };
 /* 3.填充scan response内容*/
 static const struct bt_data sd[] = {
  BT_DATA(BT_DATA_NAME_COMPLETE, CONFIG_BT_DEVICE_NAME,
      sizeof(CONFIG_BT_DEVICE_NAME) - 1),
 };
  /*4. 开启广播*/
  err = bt_le_adv_start(&adv_param, adv_data, ARRAY_SIZE(adv_data),
            sd, ARRAY_SIZE(sd));
  if (err) {
    printf("Advertising failed to start (err %d)\n", err);
    return;
  }
  printf("Advertising successfully started\n");
}
```

bt_le_adv_stop

停止蓝牙BLE广播。

函数原型

```
int bt_le_adv_stop(void)
```

输入参数

无

返回参数

`0`：成功，`其他值`：失败。

调用示例

```
int err = bt_le_adv_stop();
if (err) {
  printf("Advertising failed to stop (err %d)\n", err);
  return;
}
```

bt_le_scan_start

开启蓝牙scan。

函数原型

```
int bt_le_scan_start(const struct bt_le_scan_param *param, bt_le_scan_cb_t cb)
```

输入参数

param	struct bt_le_scan_param*	BLE扫描参数:如广播类型, duplicate filtering, scan interval 和 scan windows等。
cb	bt_le_scan_cb_t	收到scan到的数据回调函数。

返回参数

0: 成功， 其他值 : 失败。

调用示例

```
static void device_found(const bt_addr_le_t *addr, s8_t rssi, u8_t type, struct net_buf_simple *ad)
{
  char addr_str[BT_ADDR_LE_STR_LEN];
  /* 处理发现的广播数据类型 */
  if (type != BT_LE_ADV_IND && type != BT_LE_ADV_DIRECT_IND) {
    return;
  }
  if(rssi < -70){
    return;
  }
  /*这里可以对scan到的数据包作进一步解析*/
}
static void bt_scan_example(int err)
{
  int err = bt_le_scan_start(BT_LE_SCAN_ACTIVE, device_found);
  if(err){
    printf("failed to start scan(%d)\n");
  }
  printf("Bluetooth start scan\n");
}
```

bt_le_scan_stop

停止BLE scan接口。

函数原型

```
int bt_le_scan_stop(void)
```

输入参数

无

返回参数

0：成功，其他值：失败。

bt_gatt_service_register

GATT server端注册GATT服务接口。

函数原型

```
int bt_gatt_service_register(struct bt_gatt_service *svc)
```

输入参数

svc	struct bt_gatt_service	GATT service 结构体。

返回参数

0：成功，其他值：失败。

调用示例

```
/* Heart Rate 服务定义 */
static struct bt_gatt_attr attrs[] = {
  BT_GATT_PRIMARY_SERVICE(BT_UUID_HRS),
  BT_GATT_CHARACTERISTIC(BT_UUID_HRS_MEASUREMENT, BT_GATT_CHRC_NOTIFY),
  BT_GATT_DESCRIPTOR(BT_UUID_HRS_MEASUREMENT, BT_GATT_PERM_READ, NULL,
        NULL, NULL),
  BT_GATT_CCC(hrmc_ccc_cfg, hrmc_ccc_cfg_changed),
  BT_GATT_CHARACTERISTIC(BT_UUID_HRS_BODY_SENSOR, BT_GATT_CHRC_READ),
  BT_GATT_DESCRIPTOR(BT_UUID_HRS_BODY_SENSOR, BT_GATT_PERM_READ,
        read_blsc, NULL, NULL),
  BT_GATT_CHARACTERISTIC(BT_UUID_HRS_CONTROL_POINT, BT_GATT_CHRC_WRITE),
  BT_GATT_DESCRIPTOR(BT_UUID_HRS_CONTROL_POINT, BT_GATT_PERM_READ, NULL,
        NULL, NULL),
};
static struct bt_gatt_service hrs_svc = BT_GATT_SERVICE(attrs);
void hrs_init(u8_t blsc)
{
  hrs_blsc = blsc;
  /*注册定义的HRS服务*/
  bt_gatt_service_register(&hrs_svc);
}
```

bt_gatt_service_unregister

GATT server端注销服务接口。

函数原型

```
int bt_gatt_service_unregister(struct bt_gatt_service *svc)
```

输入参数

svc	struct bt_gatt_service	GATT service 结构体。

返回参数

`0`：成功，`其他值`：失败。

bt_gatt_foreach_attr

通过指定一定范围handle获取GATT server属性。

函数原型

```
void bt_gatt_foreach_attr(u16_t start_handle, u16_t end_handle,
                          bt_gatt_attr_func_t func, void *user_data)
```

输入参数

start_handle	u16_t	开始handle。
end_hanlde	u16_t	结束handle。
func	bt_gatt_attr_func_t	回调函数。
user_data	void*	用户数据。

返回参数

无。

bt_gatt_attr_next

获取GATT server下一个属性。

函数原型

```
struct bt_gatt_attr *bt_gatt_attr_next(const struct bt_gatt_attr *attr)
```

输入参数

attr	struct bt_gatt_service	GATT service 结构体。

返回参数

`下一个属性结构体指针：`成功，`NULL：`失败。

bt_gatt_attr_read

读取GATT属性接口。

函数原型

```
ssize_t bt_gatt_attr_read(struct bt_conn *conn, const struct bt_gatt_attr *attr,
                          void *buf, u16_t buf_len, u16_t offset,
                          const void *value, u16_t value_len)
```

输入参数

conn	struct bt_conn *	BLE连接。
attr	struct bt_gatt_attr *	要读取的GATT属性。
buf	void*	存储地址,read的内容会存在此buffer中。
buf_len	u16_t	存储buffer长度。
offset	u16_t	开始的偏移。
value	void*	属性值。
value_len	u16_t	属性长度。

返回参数

`读取到的属性长度：` 成功， `负值：` 失败。

bt_gatt_attr_read_included

读取GATT server包含的服务并存储，当attribute的user_data是 `bt_gatt_include` 时使用。

函数原型

```
ssize_t bt_gatt_attr_read_included(struct bt_conn *conn,
                                   const struct bt_gatt_attr *attr,
                                   void *buf, u16_t len, u16_t offset)
```

输入参数

conn	struct bt_conn *	BLE连接。
attr	struct bt_gatt_attr *	要读取的属性。
buf	void*	存储buffer空间。
len	u16_t	存储buffer的大小。
offset	u16_t	开始偏移。

返回参数

`读取到的属性长度：` 成功， `负值：错误码。`

bt_gatt_attr_read_chrc

读取GATT server包含的服务，当attribute的user_data是 `bt_gatt_chrc` 时使用。

函数原型

```
ssize_t bt_gatt_attr_read_chrc(struct bt_conn *conn,
                               const struct bt_gatt_attr *attr, void *buf,
                               u16_t len, u16_t offset)
```

输入参数

conn	struct bt_conn *	BLE连接。
attr	struct bt_gatt_attr *	要读取的属性。
buf	void*	读取到的包含character存储的buffer空间。
len	u16_t	存储buffer的大小。
offset	u16_t	开始偏移。

返回参数

`读取到的属性长度：` 成功， `负值：` 失败。

bt_gatt_attr_read_ccc

读取GATT server包含的服务，当attribute的user_data是_ `bt_gatt_ccc` 时使用。

函数原型

```
ssize_t bt_gatt_attr_read_ccc(struct bt_conn *conn,
                              const struct bt_gatt_attr *attr, void *buf,
                              u16_t len, u16_t offset)
```

输入参数

conn	struct bt_conn *	BLE连接。
attr	struct bt_gatt_attr *	要读取的属性。
buf	void*	存储读取到CCC的buffer空间。
len	u16_t	存储buffer的大小。
offset	u16_t	开始偏移。

返回参数

`读取到的属性长度：` 成功， `负值：` 失败。

bt_gatt_attr_write_ccc

写Client Characteristic Configuration (CCC)属性。

函数原型

```
ssize_t bt_gatt_attr_write_ccc(struct bt_conn *conn,
                               const struct bt_gatt_attr *attr, const void *buf,
                               u16_t len, u16_t offset, u8_t flags)
```

输入参数

conn	struct bt_conn *	BLE连接。
attr	struct bt_gatt_attr *	要读取的属性。
buf	void*	存储buffer空间。
len	u16_t	存储buffer的大小。
offset	u16_t	开始偏移。
flags	u8_t	写入标志

返回参数

`读取到的属性长度：` 成功， `负值：` 失败。

bt_gatt_attr_read_cep

获取Characteristic Extended Properties (CEP)属性。

函数原型

```
ssize_t bt_gatt_attr_read_cep(struct bt_conn *conn,
                              const struct bt_gatt_attr *attr, void *buf,
                              u16_t len, u16_t offset)
```

输入参数

conn	struct bt_conn *	BLE连接。
attr	struct bt_gatt_attr *	要读取的属性。
buf	void*	读取到的CEP存储buffer空间。
len	u16_t	存储buffer的大小。
offset	u16_t	开始偏移。

返回参数

`读取到的属性长度：` 成功， `负值：` 失败。

bt_gatt_attr_read_cud

读取Characteristic User Description （CUD）属性并存储结。

函数原型

```
ssize_t bt_gatt_attr_read_cud(struct bt_conn *conn,
                              const struct bt_gatt_attr *attr, void *buf,
                              u16_t len, u16_t offset)
```

输入参数

conn	struct bt_conn *	BLE连接。
attr	struct bt_gatt_attr *	要读取的属性。
buf	void*	获取到的CUD存储到buffer。
len	u16_t	存储buffer的大小。
offset	u16_t	开始偏移。

返回参数

`读取到的属性长度：` 成功， `负值：` 失败。

bt_gatt_attr_read_cpf

获取Characteristic Presentation Format （CPF）属性并存储结果。

函数原型

```
ssize_t bt_gatt_attr_read_cpf(struct bt_conn *conn,
                              const struct bt_gatt_attr *attr, void *buf,
                              u16_t len, u16_t offset)
```

输入参数

conn	struct bt_conn *	BLE连接。
attr	struct bt_gatt_attr *	要读取的属性。
buf	void*	获取到的CPF存储到buffer地址。
len	u16_t	存储buffer的大小。
offset	u16_t	开始偏移。

返回参数

`读取到的属性长度：` 成功， `负值：` 失败。

bt_gatt_notify

GATT server发送notify API。如果连接参数为NULL，将通知所有使能ccc的peer，否则指定connection发送notify。

函数原型

```
int bt_gatt_notify(struct bt_conn *conn, const struct bt_gatt_attr *attr,
                   const void *data, u16_t len)
```

输入参数

conn	struct bt_conn *	BLE连接。
attr	struct bt_gatt_attr *	GATT属性。
data	void*	要发送的notify数据。
len	u16_t	要发送的notify数据长度。

返回参数

`0`：成功，`其它负值`：失败。

调用事例

```
/* HRS服务定义 */
static struct bt_gatt_attr attrs[] = {
  BT_GATT_PRIMARY_SERVICE(BT_UUID_HRS),
  BT_GATT_CHARACTERISTIC(BT_UUID_HRS_MEASUREMENT, BT_GATT_CHRC_NOTIFY),
  BT_GATT_DESCRIPTOR(BT_UUID_HRS_MEASUREMENT, BT_GATT_PERM_READ, NULL,
        NULL, NULL),
  BT_GATT_CCC(hrmc_ccc_cfg, hrmc_ccc_cfg_changed),
  BT_GATT_CHARACTERISTIC(BT_UUID_HRS_BODY_SENSOR, BT_GATT_CHRC_READ),
  BT_GATT_DESCRIPTOR(BT_UUID_HRS_BODY_SENSOR, BT_GATT_PERM_READ,
        read_blsc, NULL, NULL),
  BT_GATT_CHARACTERISTIC(BT_UUID_HRS_CONTROL_POINT, BT_GATT_CHRC_WRITE),
  BT_GATT_DESCRIPTOR(BT_UUID_HRS_CONTROL_POINT, BT_GATT_PERM_READ, NULL,
        NULL, NULL),
};
void hrs_notify(void)
{
  static u8_t hrm[2];
  hrm[0] = 0x06;
  hrm[1] = heartrate;
  /*调用bt_gatt_notify接口*/
  bt_gatt_notify(NULL, &attrs[2], &hrm, sizeof(hrm));
}
```

bt_gatt_indicate

GATT server 发送Indicate 给GATT client的API。

函数原型

```
int bt_gatt_indicate(struct bt_conn *conn,
            struct bt_gatt_indicate_params *params)
```

输入参数

conn	struct bt_conn *	BLE 连接。
params	struct bt_gatt_indicate_params *	GATT indicate结构体，数据段填充需要的indication数据。

返回参数

0: 成功， 其它值 ：失败。

bt_gatt_discover

发现GATT service, 或者发现GATT character,通过指定params中的参数类型

函数原型

```
int bt_gatt_discover(struct bt_conn *conn,
                     struct bt_gatt_discover_params *params)
```

输入参数

conn	struct bt_conn *	BLE连接。
params	struct bt_gatt_discover_params *	GATT discover结构体，包含发现类型，服务或者characteristic的UUID数据。

返回参数

0: 成功， 其它值 ：失败。

调用事例

```
static void connected(struct bt_conn *conn, uint8_t err)
{
  default_conn = bt_conn_ref(conn);
  //发现GATT服务
  uuid.val = 0xffa0;
  static struct bt_gatt_discover_params discov_param;
  discov_param.uuid = &uuid.uuid;
  discov_param.func = discover_func;
  discov_param.start_handle = 0x0001;
  discov_param.end_handle = 0xffff;
   /*指定BT_GATT_DISCOVER_PRIMARY为发现gatt serice,需要指定start_handle和end_handle*/
  discov_param.type = BT_GATT_DISCOVER_PRIMARY;
  int err = bt_gatt_discover(default_conn, &discov_param);
  if (err) {
    printf("Discover failed (err %d)\n", err);
    return;
  }
}
static u8_t discover_char_func(struct bt_conn *conn,
        const struct bt_gatt_attr *attr,
        struct bt_gatt_discover_params *param)
{
    uuid.val = 0xffa1;
    static struct bt_gatt_discover_params discov_param;
    discov_param.uuid = &uuid.uuid;
    discov_param.start_handle = attr->handle + 1;
   /*指定BT_GATT_DISCOVER_CHARACTERISTIC为发现character*/
    discov_param.type = BT_GATT_DISCOVER_CHARACTERISTIC;
    int err = bt_gatt_discover(conn, &discov_param);
    if (err) {
      printk("Char Discovery failed (err %d)\n", err);
    }
}
```

bt_gatt_read

GATT client读GATT server服务的API。

函数原型

```
int bt_gatt_read(struct bt_conn *conn, struct bt_gatt_read_params *params)
```

输入参数

conn	BTstruct bt_conn *	BLE连接。
params	struct bt_gatt_read_params *	GATT read服务的结构体，包括读的句柄，回调函数等。

返回参数

`0`：成功，`其它负值`：失败。

bt_gatt_write

GATT client调用此接口向GATT server写服务，GATT server需要write response,并通过回调告知。

函数原型

```
int bt_gatt_write(struct bt_conn *conn, struct bt_gatt_write_params *params)
```

输入参数

conn	struct bt_conn *	BLE连接。
params	struct bt_gatt_write_params *	BLE gatt write服务的结构体，包括句柄，写的数据，长度等。

返回参数

`0`：成功，`其它负值`：失败。

调用事例

```
static void write_cb(struct bt_conn *conn, u8_t err,
        struct bt_gatt_write_params *params)
{
  //write之后的回调函数
}
void gatt_write_example(void)
{
      static struct bt_gatt_write_params write_params;
      write_params.handle =  m_ctx.write_hdl;//write的handle，从character服务发现获取
      write_params.func = write_cb;//写成功之后的回调函数
      write_params.offset = 0;
      write_params.data = m_ctx.data;//write的数据
      write_params.length = m_ctx.mtu;//write的数据长度
      int ret = bt_gatt_write(default_conn , &write_params); //default_conn是ble建立连接的
      if(0 != ret){
        printf("gatt write failed(%d)\n", ret)
    return;
      }
}
```

bt_gatt_write_without_response

GATT client调用此接口向GATT server写服务，相比于bt_gatt_write无需response 。

函数原型

```
int bt_gatt_write_without_response(struct bt_conn *conn, u16_t handle,
                   const void *data, u16_t length,
                   bool sign)
```

输入参数

conn	struct bt_conn *	BT连接。
handle	u16_t	属性句柄。

data	void*	属性值。
len	u16_t	属性值长度。
sign	bool	是否签名。

返回参数

`0`：成功，`其它负值`：失败。

bt_gatt_subscribe

GATT client订阅GATT server的Indication/Notification,当GATT服务端Indication/Notification有对应的消息推送时，对应的订阅类型的回调函数会有产生。

函数原型

```
int bt_gatt_subscribe(struct bt_conn *conn,
                      struct bt_gatt_subscribe_params *params)
```

输入参数

conn	struct bt_le_adv_param*	BLE连接。
params	struct bt_gatt_subscribe_params *	订阅参数，如订阅类型，对应的回调函数等。

返回参数

`0`：成功，`其它负值`：失败。

bt_gatt_unsubscribe

GATT client取消订阅GATT server的Indication/Notification。

函数原型

```
int bt_gatt_unsubscribe(struct bt_conn *conn,
                        struct bt_gatt_subscribe_params *params)
```

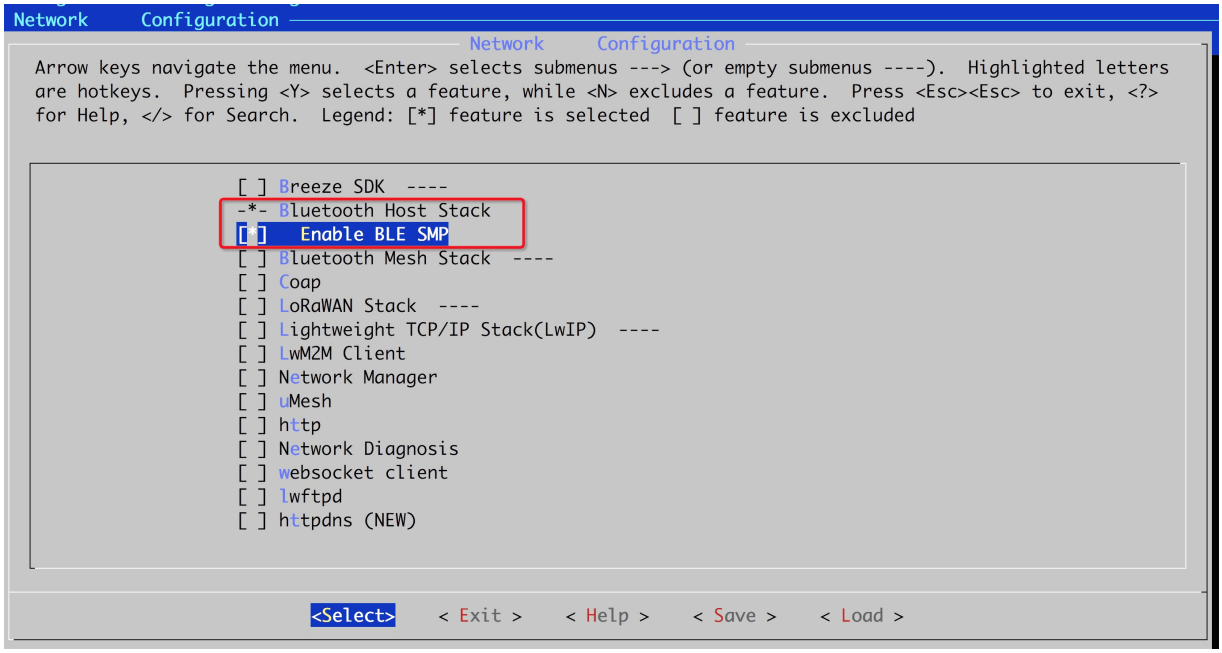
输入参数

conn	struct bt_le_adv_param*	BLE连接。
params	struct bt_gatt_subscribe_params *	取消订阅参数。如订阅类型，对应的回调函数等。

返回参数

`0`：成功，`其它负值`：失败。

配置说明



配置项说明

- ENABLE BLE SMP使能蓝牙加密模块，在需要配对认证情况下需要使能。
- 开启SMP会增加代码和内存使用大小，可以配合设置安全等级，用户的IO capability来决定开启或者关闭。

标准宏和结构体说明

bt_ready_cb_t

蓝牙协议栈初始化成功回调函数

```
typedef void (*bt_ready_cb_t)(int err);
```

bt_data

蓝牙广播数据格式字段，在bt_le_adv_start会将这些字段广播出去。

```
struct bt_data {
  u8_t type;
  u8_t data_len;
  const u8_t *data;
};
```

BT_DATA

bt_data类型的宏定义。

```
#define BT_DATA(_type, _data, _data_len)
  {
    .type = (_type),
    .data_len = (_data_len),
    .data = (const u8_t *)(_data),
  }
```

BT_LE_ADV_PARAM

广播参数的宏定义。

```
#define BT_LE_ADV_PARAM(_options, _int_min, _int_max) \
    (&(struct bt_le_adv_param) { \
      .options = (_options), \
      .interval_min = (_int_min), \
      .interval_max = (_int_max), \
    })
```

BT_LE_SCAN_PARAM

LE扫描的参数宏。

```
#define BT_LE_SCAN_PARAM(_type, _filter, _interval, _window) \
    (&(struct bt_le_scan_param) { \
      .type = (_type), \
      .filter_dup = (_filter), \
      .interval = (_interval), \
      .window = (_window), \
    })
```

广播选项

广播配置选项， `BT_LE_ADV_OPT_CONNECTABLE` 为可连接广播包；`BT_LE_ADV_OPT_ONE_TIME` 位在开启之后，从连接状态断开后需要重新开启。

```
enum {
  BT_LE_ADV_OPT_NONE = 0,
  BT_LE_ADV_OPT_CONNECTABLE = BIT(0),
  BT_LE_ADV_OPT_ONE_TIME = BIT(1),
};
```

struct bt_le_adv_param

广播参数结构体,依次是广播选项，广播的最小间隔，最大间隔，非连接广播包NRPA地址。

```
struct bt_le_adv_param {
  u8_t  options;
  u16_t interval_min;
  u16_t interval_max;
  const bt_addr_t *own_addr;
};
```

struct bt_le_scan_param

BLE广播的选项配置。

- type，主动扫描或者被动扫描。
- filter_dup，相同地址过滤是否开启。
- interval，scan的interval，0.625ms*interval。
- window，扫描的窗口大小,真实时间大小为0.625ms*window。

```
struct bt_le_scan_param {
  u8_t  type;
  u8_t  filter_dup;
  u16_t interval;
  u16_t window;
};
```

2.4.8. BLE Mesh组件文档

蓝牙Mesh网络是一种多对多通信网络，节点设备之间可以相互通信，具有覆盖范围大、功耗低、安全性高、入网灵活、支持大规模节点设备等特点。 蓝牙SIG组织于2017年7月发布了蓝牙Mesh V1.0标准，AliOS Things BLE Mesh协议栈是基于该标准的实现。下面的架构图，展示了AliOS Things中提供的蓝牙相关软件产品，以及BLE Mesh协议栈在产品架构中的位置。

AliOS Things BLE产品架构图

Ble mesh组件源代码位于 `component/wireless/bluetooth/blemesh` 下，头文件位于 `include/wireless/bluetooth/blemesh` 下。

API列表

bt_mesh_init	蓝牙Mesh协议栈初始化
bt_mesh_reset	蓝牙Mesh节点状态重置
bt_mesh_prov_enable	节点Provision使能设置（ADV或者GATT）
bt_mesh_prov_disable	节点Provision禁止设置（ADV或者GATT）

bt_mesh_model_msg_init	BLE model消息初始化
bt_mesh_model_send	发送一条应用层消息
bt_mesh_model_publish	发送model的发布消息
bt_mesh_get_shell_cmd_list	获取BLE mesh相关的命令

使用

组件使能需要在需要引用blemesh组件的aos.mk中添加：

```
$(NAME)_COMPONENTS := bt_mesh
```

包含头文件

应用需要在源文件中添加ble mesh的头文件

```
#include <blemesh.h>
```

使用示例

1.蓝牙协议初始化

```
int ret;
ret = bt_enable(bt_ready);
if (ret) {
   printk("Bluetooth init failed (err %d)\n", ret);
 }
//然后在协议栈初始化成功回调函数中执行ble mesh初始化的流程
```

2.Mesh composition data准备

该步骤准备provision过程中所需的composition data，包括company ID、节点包含的elements信息。本示例中，element包括Configuration Client、Configuration Server、Sensor Client三个Model。

```
static struct bt_mesh_model root_models[] = {
    /* BLE mesh强制配置server model */
    BT_MESH_MODEL_CFG_SRV(&cfg_srv),
    BT_MESH_MODEL_CFG_CLI(&cfg_cli),
    BT_MESH_MODEL(BT_MESH_MODEL_ID_SENSOR_CLI, temp_cli_op,
            &temp_cli_pub, NULL),
};
static struct bt_mesh_elem elements[] = {
    BT_MESH_ELEM(0, root_models, BT_MESH_MODEL_NONE),
};
/* 配置一个Provisioning Node的 composition data */
static const struct bt_mesh_comp comp = {
    .cid = CID_INTEL,
    .elem = elements,
    .elem_count = ARRAY_SIZE(elements),
};
```

3.Mesh provision数据准备

Mesh provision时所需的参数，包括设备uuid、OOB输出/入方式、provision完成的回调处理函数等。该示例代码中，指定设备uuid为0xdddd，OOB以数字形式输出，并指定了provision完成后的回调。

```
static const uint8_t dev_uuid[16] = { 0xdd, 0xdd };
static const struct bt_mesh_prov prov = {
    .uuid = dev_uuid,
    .output_size = 4,
    .output_actions = BT_MESH_DISPLAY_NUMBER,
    .output_number = output_number,
    .complete = prov_complete,
};
```

4. 初始化mesh协议栈

```
int ret;
ret = bt_mesh_init(&prov, &comp);
if (ret) {
 printk("Initializing mesh failed (err %d)\n", ret);
 return;
 }
```

5. 使能provision

该步骤使能节点的provision功能，使设备可以被provisioner设备发现和provision。provision类型支持PB-ADV（BT_MESH_PROV_GATT）和PB-GATT（BT_MESH_PROV_ADV）。通过以上几个步骤后，设备还需要主动调用bt_mesh_prov_enable来具备被provisioner发现和配置的功能，传入参数指定参数PB-GATT或者PB-ADV方式，或者都指定。用户可以通过Bluez meshctl、nRF Mesh手机APP等工具对该设备进行provision操作。

```
bt_mesh_prov_enable(BT_MESH_PROV_GATT | BT_MESH_PROV_ADV);
```

API详情

bt_mesh_init

初始化BLE mesh协议栈，成功之后Node需要调用bt_mesh_prov_enable()来使能unprovisioned adv。

函数原型

```
int bt_mesh_init(const struct bt_mesh_prov *prov, const struct bt_mesh_comp *comp)
```

输入参数

prov	const struct bt_mesh_prov *	provision配置信息，包含uuid、OOB认证方式和认证信息长度、provision相关回调等
comp	const struct bt_mesh_comp *	节点支持的element、model等配置

返回参数

0 :成功。

负值:错误, 见**错误编码定义部分。**

调用示例：

```
/*定义Node上的各个element*/
static struct bt_mesh_model root_models[] = {
  BT_MESH_MODEL_CFG_SRV(&cfg_srv),
  BT_MESH_MODEL_CFG_CLI(&cfg_cli),
  BT_MESH_MODEL_HEALTH_SRV(&health_srv, &health_pub),
};
static struct bt_mesh_elem elements[] = {
  BT_MESH_ELEM(0, root_models, vnd_models),
};
static const struct bt_mesh_comp comp = {
  .cid = CID_INTEL,
  .elem = elements,
  .elem_count = ARRAY_SIZE(elements),
};
/*定义provisioning的能力和属性*/
static const struct bt_mesh_prov prov = {
  .uuid = dev_uuid,
  .output_size = 4,
  .output_actions = BT_MESH_DISPLAY_NUMBER,
  .output_number = output_number,
  .complete = prov_complete,
  .reset = prov_reset,
};
/*初始化调用BLE mesh协议栈*/
int ret;
ret = bt_mesh_init(&prov, &comp);
if (ret) {
  printk("Initializing mesh failed (err %d)\n", ret);
  return;
}
```

bt_mesh_reset

蓝牙Mesh节点状态重置。调用该接口后，设备需要重新provision才能加入mesh网络。此外，调用该接口后，设备不会自动广播unprovision beacon，要调用bt_mesh_prov_enable接口重新使能相应bearer的广播。

函数原型

```
void bt_mesh_reset(void)
```

输入参数

无

返回参数

无

调用示例

```
static int cmd_reset(int argc, char *argv[])
{
  bt_mesh_reset();
  printk("Local node reset complete\n");
  return 0;
}
```

bt_mesh_prov_enable

Provision使能设置,调用此接口设备才能进入unprovisioned广播态，参数可以指定provision方式（PB-ADV或者PB-GATT）

函数原型

```
int bt_mesh_prov_enable(bt_mesh_prov_bearer_t bearers)
```

输入参数

bearer	bt_mesh_prov_bearer_t	Bearer设置，ADV或者GATT。

返回参数

0 :成功。负值:错误, 具体错误见错误编码定义部分。

bt_mesh_prov_disable

关闭Provision能力（PB-ADV或者PB-GATT）。

函数原型

```
int bt_mesh_prov_disable(bt_mesh_prov_bearer_t bearers)
```

输入参数

bearer	bt_mesh_prov_bearer_t	Bearer provision方式，ADV或者GATT，或者全部。

返回参数

0 :成功。

负值:错误编码, 见**错误编码定义**部分。

调用示例:

```
err = bt_mesh_prov_disable(bearer);
if (err) {
  printk("Failed to disable %s (err %d)\n", bearer2str(bearer), err);
} else {
  printk("%s disabled\n", bearer2str(bearer));
}
```

bt_mesh_model_msg_init

model消息初始化。该接口初始化消息buf，并填充opcode头信息。

函数原型

```
void bt_mesh_model_msg_init(struct net_buf_simple *msg, u32_t opcode)
```

输入参数

msg	struct net_buf_simple *	用于发送消息的net_buf
opcode	u32_t	消息对应的opcode码

返回参数

无

调用示例

```
static void gen_onoff_get(struct bt_mesh_model *model,
                          struct bt_mesh_msg_ctx *ctx,
                          struct net_buf_simple *buf)
{
  struct net_buf_simple *msg = NET_BUF_SIMPLE(2 + 1 + 4);
  printk("onoff get: addr 0x%04x onoff 0x%02x\n",
      model->elem->addr, g_onoff_state.current);
  bt_mesh_model_msg_init(&msg, BT_MESH_MODEL_OP_2(0x82, 0x04));
  net_buf_simple_add_u8(&msg, g_onoff_state.current);
  if (bt_mesh_model_send(model, ctx, &msg, NULL, NULL)) {
    printk("Unable to send On Off Status response\n");
  }
}
```

bt_mesh_model_send

发送一条应用层消息。

函数原型

```
int bt_mesh_model_send(struct bt_mesh_model *model,
          struct bt_mesh_msg_ctx *ctx,
          struct net_buf_simple *msg,
          const struct bt_mesh_send_cb *cb, void *cb_data)
```

输入参数

model	struct bt_mesh_model *	消息对应的model
ctx	struct bt_mesh_msg_ctx *	消息的上下文信息，包括key索引、对端地址、TTL等
msg	struct net_buf_simple *	需要发送的消息
cb	const struct bt_mesh_send_cb *	消息发送完毕的回调。该参数可选

cb_data	void*	回调函数的用户参数

返回参数

0：成功。

负值：错误编码，见**错误编码定义**部分。

调用示例

可参考bt_mesh_model_msg_init部分示例。

bt_mesh_model_publish

发送model的发布消息。调用该接口前，用户确保需要model的bt_mesh_model_pub.msg中包含有效的消息。该接口仅用于非周期性的消息发布。

函数原型

```
int bt_mesh_model_publish(struct bt_mesh_model *model)
```

输入参数

model	struct bt_mesh_model *	消息对应的model

返回参数

0：成功 。

负值:错误, 编码见**错误编码定义**部分。

调用示例:

```
bt_mesh_model_msg_init(msg, BT_MESH_MODEL_OP_2(0x82, 0x04));
net_buf_simple_add_u8(msg, g_onoff_state.current);
err = bt_mesh_model_publish(model);
if (err) {
  printk("bt_mesh_model_publish err %d\n", err);
}
```

bt_mesh_get_shell_cmd_list

获取ble mesh相关的命令，此部分命令列表可以参考通过shell 命令部分。

函数原型

```
struct mesh_shell_cmd *bt_mesh_get_shell_cmd_list()
```

输入参数

无

返回参数

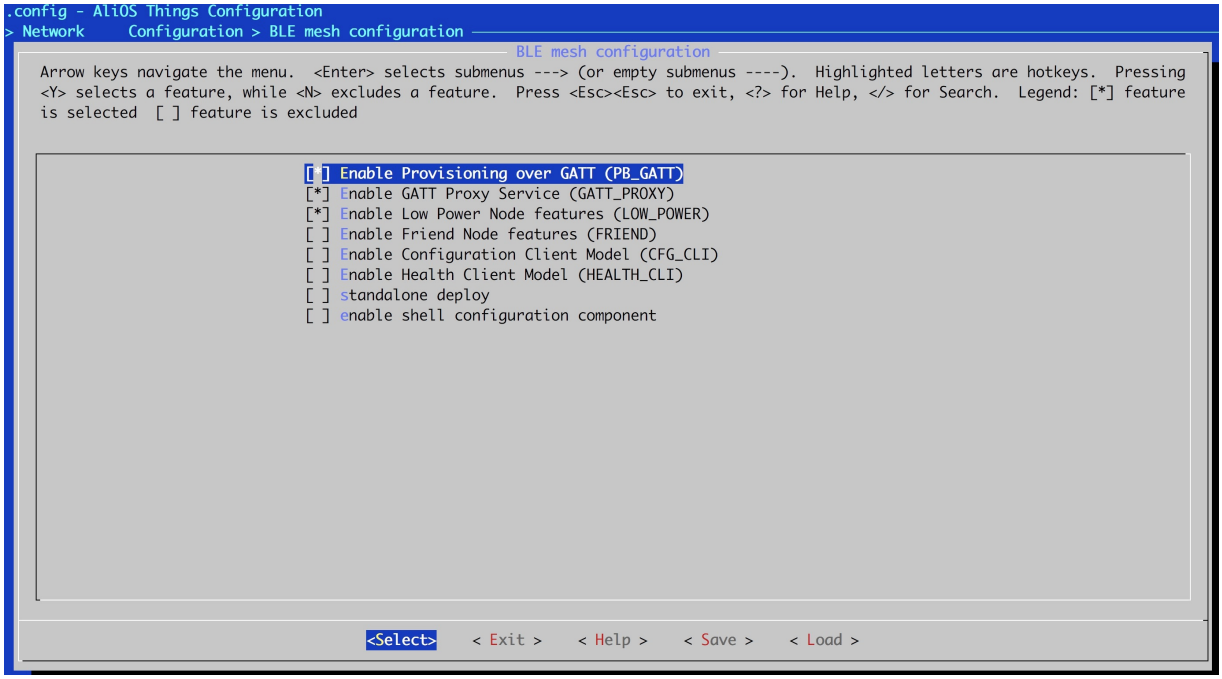
struct mesh_shell_cmd的命令列表。

调用示例：

```
mesh_cmds = bt_mesh_get_shell_cmd_list();
if (mesh_cmds) {
  p = mesh_cmds;
  while (p->cmd_name != NULL) {
    if (strcmp(p->cmd_name, cmd_str) != 0) {
      p++;
      continue;
    }
    if (p->cb) {
      no_match = 0;
      p->cb(argc - 1, &(argv[1]));//这里会调用注册的回调函数
    }
    break;
  }
}
```

配置说明

在ble mesh配置选项中选择后可以继续根据配置使能相对应的功能和配置。



配置项说明

- PB-GATT：通过GATT链路进行provisioning。
- GATT Proxy：支持GATT代理服务，一般用在GATT client和mesh网络之间。
- LOW-Power：Low power功能。
- FRIEND:friend功能。
- CFG_CLI:配置客户端，基本model。
- HEALTH_CLI:基本model health。
- standalone deploy:单独编译ble mesh组件，不包含依赖的ble host。
- enable shell configuration component：使能shell命令的功能。

标准宏和结构体说明

struct bt_mesh_prov

bt mesh属性和能力的结构体。

```
static struct bt_mesh_prov prov = {
    .uuid = dev_uuid,
    .link_open = link_open,
    .link_close = link_close,
    .complete = prov_complete,
    .reset = prov_reset,
    //.static_val = NULL,
    //.static_val_len = 0,
    .output_size = 6,
    .output_actions = (BT_MESH_DISPLAY_NUMBER | BT_MESH_DISPLAY_STRING),
    .output_number = output_number,
    .output_string = output_string,
    .input_size = 6,
    .input_actions = (BT_MESH_ENTER_NUMBER | BT_MESH_ENTER_STRING),
    .input = input,
};
```

struct bt_mesh_provisioner

BLE mesh provisioner实例，初始化调用。如：

```
static struct bt_mesh_provisioner provisioner = {
    .prov_uuid          = 0,
    .prov_unicast_addr    = PROVISIONER_UNICAST_ADDR,
    .prov_start_address   = NODE_START_UNICAST_ADDR,
    .prov_attention       = 0,
    .prov_algorithm       = 0,
    .prov_pub_key_oob     = 1,
    .prov_pub_key_oob_cb  = provisioner_pub_key_oob,
    .prov_static_oob_val  = default_static_oob,
    .prov_static_oob_len  = 4,
    .prov_input_num       = provisioner_input_num,
    .prov_output_num      = provisioner_output_num,
    .flags              = 0,
    .iv_index           = 0,
    .prov_link_open       = provisioner_link_open,
    .prov_link_close      = provisioner_link_close,
    .prov_complete        = provisioner_complete,
};
ret = bt_mesh_init(&prov, &comp, &provisioner);
if (ret) {
    printk("Mesh initialization failed (err %d)\n", ret);
}
```

struct bt_mesh_model_pub

model publication的定义。

```
struct bt_mesh_model_pub {
  /** 上下文的model，协议栈初始化需要设置. */
  struct bt_mesh_model *mod;
  u16_t addr;      /**< Publish 地址. */
  u16_t key;       /**< Publish AppKey 索引. */
  u8_t  ttl;       /**< Publish 的TTL. */
  u8_t  retransmit;  /**< 重试次数. */
  u8_t  period;     /**< Publish 时间长度. */
  u8_t  period_div:4, /**< Public 时间的除数因子. */
      cred:1,     /**< Friendship Credentials Flag. */
      count:3;    /**< 剩余重传次数. */
  u32_t period_start; /**< 当前时间段的开始时间. */
  struct net_buf_simple *msg;
  int (*update)(struct bt_mesh_model *mod);
  /** Publish 消息的定时器. */
  struct k_delayed_work timer;
};
```

struct bt_mesh_elem

bt mesh的结构体抽象。

```
struct bt_mesh_elem {
  /* 单播地址，在provisioning的时候设置 */
  u16_t addr;
  /* 地址的描述符  */
  const u16_t loc;
  const u8_t model_count;
  const u8_t vnd_model_count;
  struct bt_mesh_model * const models;
  struct bt_mesh_model * const vnd_models;
};
```

struct bt_mesh_msg_ctx

mesh发送消息的结构体。

```
struct bt_mesh_msg_ctx {
  /** 发送消息目的NetKey的索引*/
  u16_t net_idx;
  /** 要加密的AppKey索引. */
  u16_t app_idx;
  /** 目的地址*/
  u16_t addr;
  /** 收到的TTL */
  u8_t  recv_ttl:7;
  /** 当使用ACK的时候强制发送 */
  u8_t  send_rel:1;
  /** 发送TTL. */
  u8_t  send_ttl;
};
```

struct bt_mesh_model_op

mesh model的opcode结构体,包含opcode,handler。

```
struct bt_mesh_model_op {
  /* ble mesh的消息的OpCode  */
  const u32_t  opcode;
  /* 最小的消息长度 */
  const size_t min_len;
  /* 对应opcode的回调函数 */
  void (*const func)(struct bt_mesh_model *model,
        struct bt_mesh_msg_ctx *ctx,
        struct net_buf_simple *buf);
};
```

struct bt_mesh_model

bt mesh model的实例。

```
struct bt_mesh_model {
  union {
    const u16_t id;
    struct {
      u16_t company;
      u16_t id;
    } vnd;
  };
  /* Model属于的Element */
  struct bt_mesh_elem *elem;
  /* Model的Publication 上下文*/
  struct bt_mesh_model_pub * const pub;
  /* AppKey列表 */
  u16_t keys[CONFIG_BT_MESH_MODEL_KEY_COUNT];
  /* Subscription 列表 (组播或者虚拟地址) */
  u16_t groups[CONFIG_BT_MESH_MODEL_GROUP_COUNT];
  const struct bt_mesh_model_op * const op;
  /* Model自定义用户地址 */
  void *user_data;
};
```

struct mesh_shell_cmd

BLE mesh 命令函结构体定义。

```
struct mesh_shell_cmd {
    const char *cmd_name;
    shell_cmd_function_t cb;
    const char *help;
    const char *desc;
};
```

BT_MESH_ELEM (*loc,* mods, _vnd_mods)

BLE mesh element的抽象。可以使用此宏定义一个element, 配合bt_mesh_model, bt_mesh_elem使用如下。

```
static struct bt_mesh_model root_models[] = {
  BT_MESH_MODEL_CFG_SRV(&cfg_srv),
  BT_MESH_MODEL_CFG_CLI(&cfg_cli),
  BT_MESH_MODEL_HEALTH_SRV(&health_srv, &health_pub),
  BT_MESH_MODEL_HEALTH_CLI(&health_cli),
};
static struct bt_mesh_elem elements[] = {
  BT_MESH_ELEM(0, root_models, BT_MESH_MODEL_NONE),
};
```

值定义

_mods	model的数组
_vnd_mods	vendor model的数组

BT_MESH_MODEL(*id,* op, *pub,* user_data)

BT SIG bt mesh model的定义。

op	model opcode的回调函数
pub	model publish的参数
user	model的用户参数

BT_MESH_MODEL_CFG_CLI

定义一个config client的model。

```
#define BT_MESH_MODEL_CFG_CLI(cli_data)                 \
    BT_MESH_MODEL(BT_MESH_MODEL_ID_CFG_CLI,             \
        bt_mesh_cfg_cli_op, NULL, cli_data)
```

BT_MESH_MODEL_CFG_SRV

定义一个config server的model。

```
#define BT_MESH_MODEL_CFG_SRV(srv_data)                 \
    BT_MESH_MODEL(BT_MESH_MODEL_ID_CFG_SRV,             \
        bt_mesh_cfg_srv_op, NULL, srv_data
```

BT_MESH_MODEL_HEALTH_CLI

定义一个health client 的model。

```
#define BT_MESH_MODEL_HEALTH_CLI(cli_data)              \
    BT_MESH_MODEL(BT_MESH_MODEL_ID_HEALTH_CLI,          \
        bt_mesh_health_cli_op, NULL, cli_data)
```

BT_MESH_MODEL_HEALTH_SRV

定义一个health server的model。

```
#define BT_MESH_MODEL_HEALTH_SRV(srv, pub)                  \
    BT_MESH_MODEL(BT_MESH_MODEL_ID_HEALTH_SRV,              \
          bt_mesh_health_srv_op, pub, srv)
```

enum bt_mesh_output_action_t

BLE mesh output能力,现阶段只处理BT_MESH_DISPLAY_STRING和BT_MESH_DISPLAY_NUMBER。

```
BT_MESH_NO_OUTPUT
BT_MESH_BLINK
BT_MESH_BEEP
BT_MESH_VIBRATE
BT_MESH_DISPLAY_NUMBER
BT_MESH_DISPLAY_STRING
```

enum bt_mesh_input_action_t

BLE mesh 输入能力，现阶段只处理BT_MESH_ENTER_STRING和BT_MESH_ENTER_NUMBER。

```
BT_MESH_NO_INPUT
BT_MESH_PUSH
BT_MESH_TWIST
BT_MESH_ENTER_NUMBER
BT_MESH_ENTER_STRING
```

错误码定义

使用了标准的POSIX错误码，仅列出常用的错误码，更多错误码请参考：
components/wireless/bluetooth/blemesh/util/include/errno.h

```
#define EPERM 1      /*操作不允许*/
#define ENOENT 2     /*文件/路径不存在*/
#define ESRCH 3      /*进程不存在*/
#define EINTR 4      /*中断的系统调用*/
#define EIO 5        /*I/O错误*/
#define ENXIO 6      /*设备/地址不存在*/
#define E2BIG 7      /*参数列表过长*/
#define ENOEXEC 8    /*执行格式错误*/
#define EBADF 9      /*错误文件编号*/
#define ECHILD 10    /*子进程不存在*/
#define EAGAIN 11    /*重试*/
#define ENOMEM 12    /*内存不足*/
#define EACCES 13    /*无权限*/
#define EFAULT 14    /*地址错误*/
#define ENOTBLK 15   /*需要块设备*/
#define EBUSY 16     /*设备或资源忙*/
#define EEXIST 17    /*文件已存在*/
#define EXDEV 18     /*跨设备链路*/
#define ENODEV 19    /*设备不存在*/
#define ENOTDIR 20   /*路径不存在
#define EISDIR 21    /*是路径*/
#define EINVAL 22    /*无效参数*/
#define ENFILE 23    /*文件表溢出*/
#define EMFILE 24    /*打开的文件过多*/
#define ENOTTY 25    /*非打字机*/
#define ETXTBSY 26   /*文本文件忙*/
#define EFBIG 27     /*文件太大*/
#define ENOSPC 28    /*设备无空间*/
#define ESPIPE 29    /*非法查询*/
#define EROFS 30     /*只读文件系统*/
//....POSIX err code
```

enum bt_mesh_prov_bearer_t

provision bearer的两种类型。

```
BT_MESH_PROV_ADV
BT_MESH_PROV_GATT
```

enum bt_mesh_prov_oob_info_t

OOB消息,对应蓝牙mesh协议的OOB消息，16比特，BLE mesh spec定义(Mesh Profile, V1.0, Page 119):

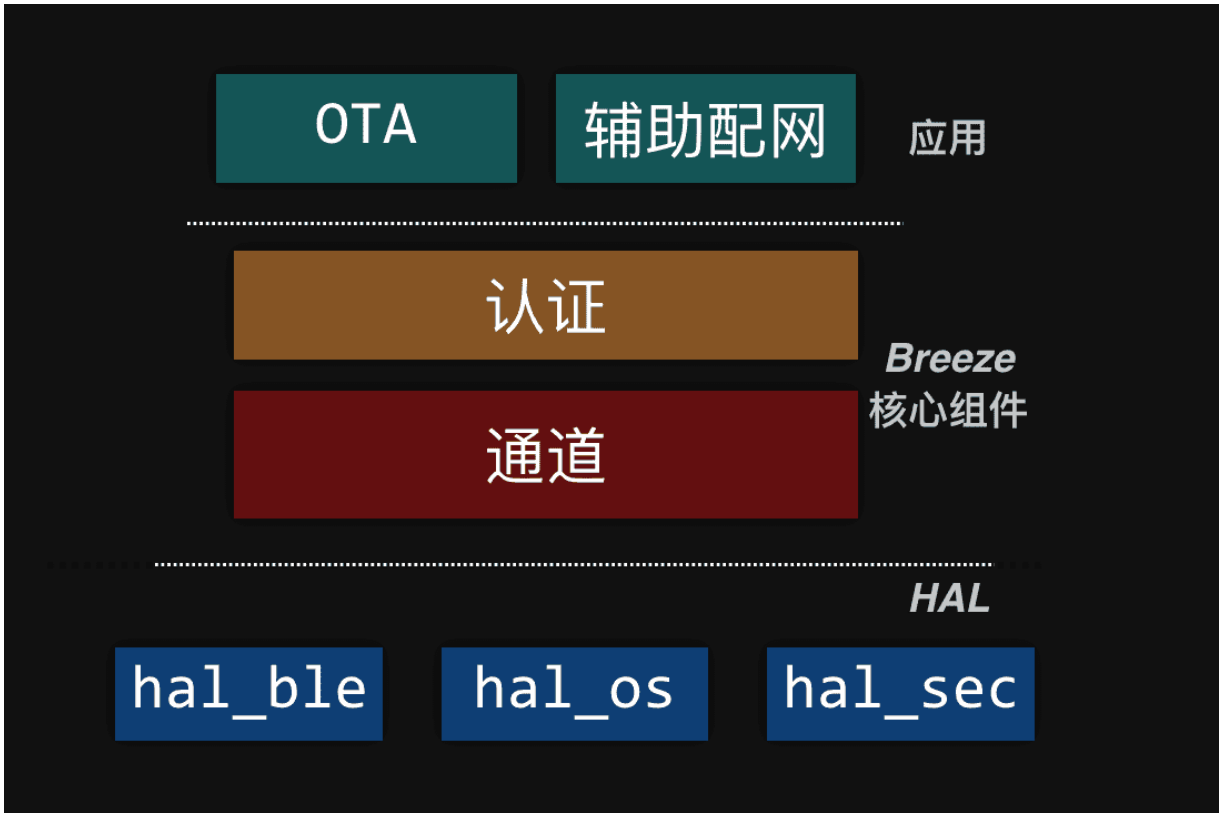
0	Other
1	Electronic / URI
2	2D machine-readable code
3	Bar code
4	Near Field Communication (NFC)
5	Number

6	String
7-10	Reserved for Future Use
11	on box
12	in box
13	On piece of paper
14	Inside manual
15	On device

```
BT_MESH_PROV_OOB_OTHER
BT_MESH_PROV_OOB_URI
BT_MESH_PROV_OOB_2D_CODE
BT_MESH_PROV_OOB_BAR_CODE
BT_MESH_PROV_OOB_NFC
BT_MESH_PROV_OOB_NUMBER
BT_MESH_PROV_OOB_STRING
BT_MESH_PROV_OOB_ON_BOX
BT_MESH_PROV_OOB_IN_BOX
BT_MESH_PROV_OOB_ON_PAPER
BT_MESH_PROV_OOB_IN_MANUAL
BT_MESH_PROV_OOB_ON_DEV
```

2.4.9. Breeze-基于蓝牙低功耗协议栈的安全连云SDK

阿里云IoT针对蓝牙设备，提供了一套基于蓝牙BLE链接轻量级安全Breeze方案，可以实现APP-设备-云的完整链路。本文主要介绍在设备端上，如何让硬件具备蓝牙BLE能力，并与手机蓝牙连接后通过一套SDK建立安全通道，将数据推送给云端。除此之外，通道也具备作为蓝牙辅助配网，和OTA升级的能力。



Breeze组件源代码位于 componet/wireless/ble/breeze 下，头文件位于 include/wireless/bluetooth/breeze 下。

API列表

breeze_start	启动breeze SDK服务。用户使用此接口初始化和启动breeze服务
breeze_end	停止breeze服务，用户调用此接口停止breeze服务
breeze_post	推送设备端状态数据至移动端，使用BLE indicate方式
breeze_post_fast	推送设备端状态数据至移动端，区别在于使用BLE notification方式
breeze_post_ext	设备端上报带有cmd字段的数据至移动端
breeze_append_adv_data	广播内容增加用户自定义数据
breeze_restart_advertising	SDK重启蓝牙广播
breeze_awss_init	该接口对蓝牙配网SDK进行初始化。在用户业务逻辑初始化阶段调用
breeze_awss_end	停止蓝牙配网服务

使用

使能Breeze SDK

在组件aos.mk添加breeze组件：

```
$(NAME)_COMPONENTS := breeze
```

包含头文件

应用需要在应用源文件添加breeze部分对外头文件：

```
#include <breeze.h>
```

使用示例

使用SDK API

可参考事例breezeapp, 路径在 `application/example/example_legacy/bluetooth/breezeapp`

1. 准备设备信息，包含设备Product ID, device Secret, device name, product key, product secret。

```
/*
* 1.定义五元组
*/
#define PRODUCT_ID    577245
#define DEVICE_SECRET  "************************"
#define DEVICE_NAME   "12345678901234567890"
#define PRODUCT_KEY   "**********"
#define PRODUCT_SECRET "***********"
```

2.SDK初始化，包括对结构体的初始化，回调函数注册等。

```
static void alink_work(void *arg)
{
  bool          ret;
  uint32_t        err_code;
  struct device_config init_bzlink;
  uint8_t         bd_addr[BD_ADDR_LEN] = { 0 };
//如果使能OTA，注册相关回调函数
#ifdef CONFIG_AIS_OTA
  ota_breeze_service_manage_t ota_module;
#endif
  (void)arg;
  //初始化结构体
  memset(&init_bzlink, 0, sizeof(struct device_config));
  init_bzlink.product_id      = PRODUCT_ID;
  init_bzlink.status_changed_cb = dev_status_changed_handler;
  init_bzlink.set_cb        = set_dev_status_handler;
  init_bzlink.get_cb        = get_dev_status_handler;
  init_bzlink.apinfo_cb       = apinfo_handler;
  init_bzlink.product_secret_len = strlen(PRODUCT_SECRET);
  memcpy(init_bzlink.product_secret, PRODUCT_SECRET,
  init_bzlink.product_secret_len);
  init_bzlink.product_key_len = strlen(PRODUCT_KEY);
  memcpy(init_bzlink.product_key, PRODUCT_KEY, init_bzlink.product_key_len);
  init_bzlink.device_key_len = strlen(DEVICE_NAME);
  memcpy(init_bzlink.device_key, DEVICE_NAME, init_bzlink.device_key_len);
#ifndef CONFIG_MODEL_SECURITY
```

```
#ifndef CONFIG_MODEL_SECRET
    init_bzlink.secret_len = strlen(DEVICE_SECRET);
    memcpy(init_bzlink.secret, DEVICE_SECRET, init_bzlink.secret_len);
#else
    init_bzlink.secret_len = 0;
#endif
#ifdef CONFIG_AIS_OTA
    ota_module.is_ota_enable = true;
    ota_module.verison.fw_ver_len = strlen(SOFTWARE_VERSION);
    if(ota_module.verison.fw_ver_len > sizeof(ota_module.verison.fw_ver)) {
        printf("breeze version too long");
        return;
    }
    memcpy(ota_module.verison.fw_ver, SOFTWARE_VERSION, ota_module.verison.fw_ver_len);
    ota_module.get_dat_cb = NULL;
    ota_breeze_service_init(&ota_module);
    init_bzlink.ota_cb = ota_module.get_dat_cb;
#else
    init_bzlink.ota_cb = ota_handler;
#endif
    //开启breeze SDK
    ret = breeze_start(&init_bzlink);
    if (ret != 0) {
        printf("breeze_start failed.\r\n");
    } else {
        printf("breeze_start succeed.\r\n");
    }
}
int application_start(int argc, char **argv)
{
    alink_work(NULL);
    return 0;
}
```

3.整个通道建立以及对应事件后会通过回调来驱动

- 连接状态回调，会打印连接状态的状态：蓝牙连接，断开，认证，发送数据完成等。

```
static bool ble_connected = false;
static void dev_status_changed_handler(breeze_event_t event)
{
  switch (event) {
    case CONNECTED:
      ble_connected = true;
      printf("dev_status_changed(): Connected.\n");
      break;
    case DISCONNECTED:
      ble_connected = false;
      printf("dev_status_changed(): Disconnected.\n");
      break;
    case AUTHENTICATED:
      printf("dev_status_changed(): Authenticated.\n");
      break;
    case TX_DONE:
      printf("dev_status_changed(): Tx-done.\n");
      break;
    default:
      break;
  }
}
```

- 手机下发数据回调函数,与get_dev_status_handler区别在于不需要单独使用0x03格式命令来回复给手机端。

```
static void set_dev_status_handler(uint8_t *buffer, uint32_t length)
{
  printf("%s command (len: %u) received.\r\n", __func__, length);
}
```

- 手机下发数据回调函数，需要回复0x03格式命令并填充数据。

```
static void get_dev_status_handler(uint8_t *buffer, uint32_t length)
{
  /* echo the receiving data */
  uint8_t cmd = 0x03;
  breeze_post_ext(cmd, buffer, length);
}
```

- 辅助配网信息回调，如果有对应配网信息会通过此回调通知

```
static void apinfo_handler(breeze_apinfo_t *ap)
{
  //解析配网信息并处理辅助配网逻辑
}
```

- OTA事件数据回调,在OTA中需要处理此事件

```
static void ota_handler(breeze_otainfo_t *ota)
{
  if(ota != NULL){
    if(ota->type == OTA_CMD){
      printf("RECV OTA CMD\n");
    } else if(ota->type == OTA_EVT){
      printf("RECV OTA EVT (%d)\n", ota->cmd_evt.m_evt.evt);
    } else{
      printf("unknown ota info\r\n");
    }
  }
}
```

API详情

breeze_start

启动breeze SDK服务。用户使用此接口初始化和启动breeze服务。

函数原型

```
int breeze_start(struct device_config *dev_conf)
```

输入参数

dev_conf	device_config	初始化Breeze SDK的信息，包含设备信息，回调函数等

返回参数

0-成功；-1-失败

调用示例

参考使用SDK API部分示例。

breeze_end

函数原型

```
int breeze_end(void)
```

输入参数

无

返回参数

0-成功；-1-失败

调用示例

```
int ret = breeze_end();
if (ret != 0){
  printf("Breeze SDK deinit failed\n")
}
```

breeze_post

推送设备端状态数据至移动端，使用BLE indication方式。

函数原型

```
uint32_t breeze_post(uint8_t *buffer, uint32_t length)
```

输入参数

buffer	uint8_t*	待发送数据指针
length	uint32_t	待发送数据长度

返回参数

0-成功；其他错误值-失败

调用示例

```
//发送数据
uint8_t data = {0x01, 0x02, 0x03, 0x04, 0x05};
breeze_post(date, sizeof(data));
//然后在异步事件中判断是否有TX_DONE事件回调，有则证明发送成功
static void dev_status_changed_handler(breeze_event_t event)
{
  switch (event) {
    //其他事件...
    case TX_DONE:
      printf("dev_status_changed(): Tx-done.\n");
      break;
    default:
      break;
  }
}
```

breeze_post_fast

和breeze_post类似，推送设备端状态数据至移动端，区别在于使用BLE notification方式，因此无法和breeze_post一样在dev_status_changed_handler中检测TX_DONE事件。

函数原型

```
uint32_t breeze_post_fast(uint8_t *buffer, uint32_t length)
```

输入参数

buffer	uint8_t*	待发送数据指针

length	uint32_t	待发送数据长度

返回参数

0-成功；其他错误值-失败。

调用示例

```
//发送数据
uint8_t data = {0x01, 0x02, 0x03, 0x04, 0x05};
breeze_post_fast(date, sizeof(data));
```

breeze_post_ext

设备端上报带有cmd字段的数据至移动端，常用的cmd命令字段值有0x03，表示对get_cb的回应消息，另外OTA模块有自定义的cmd字段，同样可以和breeze_post一样在dev_status_changed_handler中检测TX_DONE事件

函数原型

```
uint32_t breeze_post_ext(uint8_t cmd, uint8_t *buffer, uint32_t length)
```

输入参数

buffer	uint8_t*	待发送数据指针
length	uint32_t	待发送数据长度
cmd	uint8_t*	推送给移动端的cmd类型

返回参数

0-成功；其他错误值-失败

调用示例：

```
uint8_t data = {0x01, 0x02, 0x03, 0x04, 0x05};
uint8_t cmd = 0x03;
breeze_post_ext(cmd ,date, sizeof(data));
//然后在异步事件中判断是否有TX_DONE事件回调，有则证明发送成功
static void dev_status_changed_handler(breeze_event_t event)
{
  switch (event) {
    //其他事件...
    case TX_DONE:
      printf("dev_status_changed(): Tx-done.\n");
      break;
    default:
      break;
  }
}
```

breeze_append_adv_data

广播内容增加用户自定义数据，调用后需要调用breeze_restart_advertsing。

函数原型

```
void breeze_append_adv_data(uint8_t *data, uint32_t len)
```

输入参数

buffer	uint8_t*	待增加数据指针
length	uint32_t	待增加数据长度

返回参数

无

调用示例：

```
static bool ble_connected = False;
static void continue_adv_work(void *arg)
{
  static uint8_t user_adv[] = {0x55, 0xaa};
  user_adv[0]++;
  user_adv[1]++;
  breeze_append_adv_data(user_adv, sizeof(user_adv) / sizeof(user_adv[0]));
  if (!ble_connected) breeze_restart_advertising();
  aos_post_delayed_action(2000, continue_adv_work, NULL);
}
#endif
```

breeze_restart_advertising

SDK重启蓝牙广播，其广播内容是默认状态。

函数原型

```
void breeze_restart_advertising()
```

输入参数

无

返回参数

无

调用示例：

```
static void event_handler(ali_event_t *p_event)
 //.....
    case BZ_EVENT_DISCONNECTED:
       core_reset();
       notify_status(DISCONNECTED);
#if BZ_ENABLE_OTA
       m_disc_evt.type = OTA_EVT;
       m_disc_evt.cmd_evt.m_evt.evt = ALI_OTA_ON_DISCONNECTED;
       m_disc_evt.cmd_evt.m_evt.d = 0;
       b_notify_upper = true;
       if(g_disconnect_by_ota == true){
       } else
#endif
       {
#ifndef BLE_ADV_ASYN
  //在应用层或者接口层直接调用
          breeze_restart_advertising();
#endif
       }
       break;
```

breeze_awss_init

该接口对蓝牙配网SDK进行初始化。在用户业务逻辑初始化阶段调用。

函数原型

```
void breeze_awss_init(apinfo_ready_cb cb, breeze_dev_info_t *info)
```

输入参数

cb	apinfo_ready_cb	设备完成WiFi信息（SSID、密码）后的回调函数，由用户定义/提供，并由SDK完成调用
info	breeze_dev_info_t	为设备信息，包括ProductID，Product Key，Product Secret，Device Name，Device Secret等字段，由用户提供

返回参数

无

调用示例：

```
//配网回调函数
static void apinfo_ready_handler(breeze_apinfo_t *ap)
{
  if (!ap)
    return;
  memcpy(&apinfo, ap, sizeof(apinfo));
  //这里拿到配网信息(password，ssid等)之后的流程
}
//需要蓝牙配网的五元组信息
breeze_dev_info_t dinfo = {
    .product_id = PRODUCT_ID,
    .product_key = PRODUCT_KEY,
    .product_secret = PRODUCT_SECRET,
    .device_name = DEVICE_NAME,
    .device_secret = DEVICE_SECRET
  };
breeze_awss_init(apinfo_ready_handler, &dinfo);
```

breeze_awss_end

停止蓝牙配网服务。

函数原型

```
void breeze_awss_end()
```

输入参数

无

返回参数

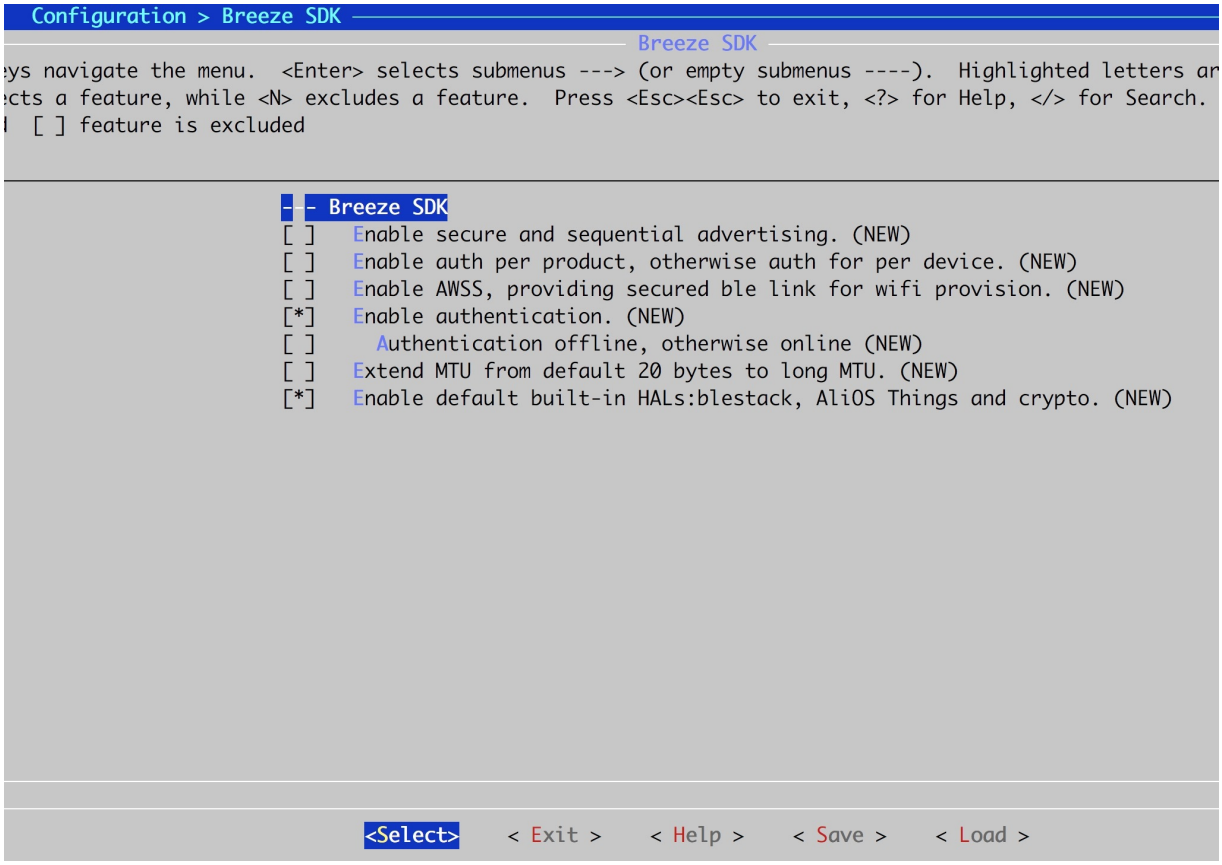
无

调用示例：

无

配置说明

在选择后可以继续根据配置使能相对应的功能和配置。



配置项说明

- Enable secure and sequencial adv: 连续安全广播，默认关闭。
- Enable auth per product: 使能设备一型一密安全，默认一机一密。
- Enable AWSS：使能蓝牙辅助配网功能，SDK默认关闭。
- Enable authentication:使能SDK认证功能，默认打开
- Authentication offline, 离线认证使能，在第一次连接后会存储密钥下次连接后不需要再次通过云端鉴权认证。默认在线认证，每次连接手机端需要和云端再次交互认证设备。
- Extend MTU：扩展SDK的MTU，根据蓝牙ATT层的MTU动态决定MTU。
- Enable default built-in components：默认选择适配自带蓝牙协议栈和安全部分，客户在对接到第三方ble stack， OS, 安全组件需要关闭此选项。

标准宏和结构体说明

struct device_config

breeze SDK初始化结构体，SDK初始化必须填充此结构体。

```
struct device_config
{
  uint8_t     bd_addr[BD_ADDR_LEN];
  char        model[STR_MODEL_LEN];
  uint32_t     product_id;
  char        product_key[STR_PROD_KEY_LEN];
  uint8_t     product_key_len;//product key的长度
  char        device_key[STR_DEV_KEY_LEN]; //设备的device name，字符串
  uint8_t     device_key_len;//设备的device name长度
  char        secret[STR_SEC_LEN];//设备的密钥，设备唯一，字符串
  uint8_t     secret_len;//设备的密钥长度
  char        product_secret[STR_PROD_SEC_LEN];
  uint8_t     product_secret_len;
  dev_status_changed_cb status_changed_cb;
  set_dev_status_cb    set_cb;
  get_dev_status_cb    get_cb;
  apinfo_ready_cb     apinfo_cb;
  ota_dev_cb        ota_cb;
};
```

其各个参数说明如下：

- bd_addr 蓝牙地址
- model 根据PID的得到的model信息，小端格式,对应广播字段的设备ID
- product_id product ID信息，由云端生成
- product_key 同一类型设备的product key
- product_key_len product key的长度
- device_key 设备的device name
- device_key_len 设备device name的长度
- secret 设备的密钥，设备唯一
- secret_len 设备密钥的长度
- product_secret 同一类型的设备的product secret
- product_secret_len product secret的长度，同一类型设备唯一

列出结构体中有关回调函数：

dev_status_changed_cb

SDK状态回调函数，包含连接，认证，断开，数据发送完成等。 **回调原型**

```
typedef void (*dev_status_changed_cb)(breeze_event_t event)
```

set_dev_status_cb

回调函数原型

```
typedef void (*set_dev_status_cb)(uint8_t *buffer, uint32_t length)
```

get_dev_status_cb

回调函数原型

```
typedef void (*get_dev_status_cb)(uint8_t *buffer, uint32_t length)
```

apinfo_ready_cb

蓝牙辅助配网事件回调接口。 **回调函数原型**

```
typedef void (*apinfo_ready_cb)(breeze_apinfo_t *ap);
```

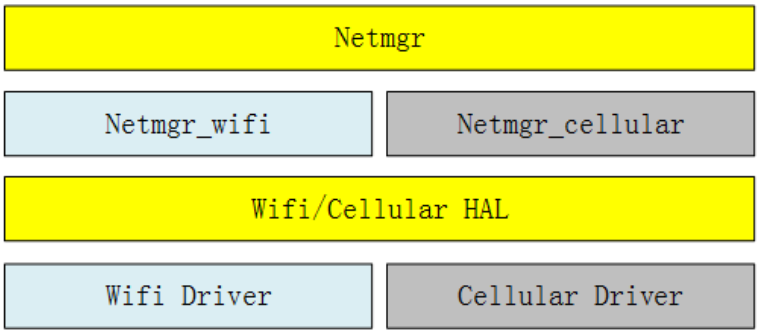
ota_dev_cb

OTA事件和数据回调接口。 **回调函数原型**

```
typedef void (*ota_dev_cb)(breeze_otainfo_t *otainfo);
```

2.4.10. Netmgr组件文档

Netmgr是AliOS Things的网络管理模块，主要管理设备连网操作。目前支持两种网络：蜂窝（Cellular）网络和Wi-Fi网络。



蜂窝网络

使用者通过调用相关API，完成入网及获取IP的功能。

Wi-Fi网络

使用者通过调用相关的API，完成连接指定Wi-Fi，获取IP的功能，或者启动Wi-Fi配网功能。

API

netmgr的主要API如下表所示：

API	API说明	Wi-Fi	Cellular
netmgr_init	初始化	Y	Y
netmgr_deinit	反初始化	Y	Y
netmgr_start	启动	Y	Y
netmgr_connect	连接指定网络	Y	N

netmgr_stats	获取网络参数	Y	Y

事件监听

使用aos_register_event_filter函数通过注册回调函数来监听Cellular/Wi-Fi的获取IP事件。

```
int aos_register_event_filter(uint16_t type, aos_event_cb cb, void *priv);
```

其中type事件类型(type)以及获取IP的事件码(code)信息如下：

```
/* WiFi event */
#define EV_WIFI          0x0002
#define CODE_WIFI_ON_GOT_IP    5
/* cellular event */
#define EV_CELLULAR          0x0005
#define CODE_CELLULAR_ON_GOT_IP    5
```

回调函数的定义如下，当有事件触发的时候，可以从event里读出当前的事件类型(type)以及对应的事件码(code)

```
typedef void (*aos_event_cb)(input_event_t *event, void *private_data);
```

使用

添加组件

输入aos make menuconfig进入如下界面：

选择Network:

选择Config Network Interface Types来选择网络类型：

选择Wi-Fi/Cellular，保存退出。

包含头文件

```
#include "netmgr.h"
```

使用示例

启动Wi-Fi配网

```
static void wifi_service_event(input_event_t *event, void *priv_data)
{
  if (event->type != EV_WIFI) {
    return;
  }
  if (event->code != CODE_WIFI_ON_GOT_IP) {
    return;
  }
  // add application start logic here
}
void start_netmgr(void) {
  netmgr_init();
  aos_register_event_filter(EV_WIFI, wifi_service_event, NULL);
  netmgr_start(true);
}
```

直接连接W-iFi

```
void start_netmgr(void)
{
  netmgr_init();
  netmgr_connect("Test_WiFi", "123456", 10000);
}
```

蜂窝网络连接

```
void start_netmgr(void)
{
  netmgr_init();
  netmgr_start(false);
}
```

API详情

netmgr_init

初始化netmgr模块

函数原型

```
int32_t netmgr_init(void);
```

参数列表

参数名称	参数描述	参数示例
无	无	无

返回参数

0，成功

小于0， 失败

备注

在蜂窝网络场景下，初始化变量，及注册网络相关事件处理函数。网络相关事件具体包括：建立连接，断开连接，或得到IP。

在Wi-Fi网络场景下，初始化变量，注册网络相关事件处理函数，读取存储在flash上的SSID和Password，注册Wi-Fi配网方法。

netmgr_deinit

反初始化netmgr

函数原型

```
void netmgr_deinit(void);
```

参数列表

参数名称	参数描述	参数示例
无	无	无

返回参数

无

备注

释放相关申请资源。

netmgr_start

启动netmgr

函数原型

```
int32_t netmgr_start(bool autoconfig);
```

参数列表

参数名称	参数描述	参数示例
autoconfig	是否自动发起Wi-Fi配网	无

返回参数

0，成功

小于0， 失败

备注

在蜂窝网场景下，该函数可以不做调用。

在Wi-Fi场景下，当传输参数autoconfig是false，检查flash是否存有合法SSID和Password。如果有，则开始连接，否则不做任何操作。当传入参数autoconfig是true，检查flash是否存有合法SSID和Password。如果有，则开始连接，否则开始W-iFi配网流程。

netmgr_connect

连接网络

函数原型

```
int32_t netmgr_connect(const char *ssid, const uint8_t *password, uint32_t timeout);
```

参数列表

参数名称	参数描述	参数示例
ssid	Wi-Fi SSID	aliyun
password	密码	12345678
timeout	超时时间	100

返回参数

0，成功

-1， 入参错误

-2，连接超时

备注

该API只在Wi-Fi网络场景下使用。根据传入的SSID和Password，在指定的时间内(timeout)连接Wi-Fi。

netmgr_stats

获取网络统计信息，现在主要是IP地址。

函数原型

```
void netmgr_stats(int32_t interface, netmgr_stats_t *stats);
```

参数列表

参数名称	参数描述	参数示例
interface	网卡名字	eth1
stats	网卡统计信息	123456

返回参数

无

备注

```
#define IP_STR_SIZE 32
typedef struct netmgr_stats_s {
   bool ip_available;
   char ip[IP_STR_SIZE];
} netmgr_stats_t;
```

命令行CLI

启动网络连接

```
netmgr start
```

Wi-Fi场景会连接存储在flash中的SSID/PASSWORD，如果没有存储有效SSID/PASSWORD，则启动Wi-Fi配网流程。

清除存储的SSID/PASSWORD

```
netmgr clear
```

获取网络参数

```
netmgr stats
```

连接指定SSID/PASSWORD

```
netmgr connect SSID PASSWORD
```

2.4.11. SAL组件文档

SAL，是Socket Adapter Layer的简称。AliOS Things中SAL套件是针对MCU+外部通信模组的方式，提供标准Socket接口服务。SAL套件提供AT命令到Socket标准接口的转换。借助SAL套件，用户不用感知底层通信方式和介质（如WiFi、2G、4G等模组），可以使用标准Socket接口进行应用开发，使上层应用具有更好的可移植性。AliOS Things中SAL组件的架构图如下。

其中，组件包括：

- **SAL Core**：由AliOS Things提供，SAL核心组件（上图蓝色）。主要包括Socket连接管理、数据缓存、协议转换等功能，对上提供标准Socket接口服务，对下提供统一的HAL(Hardware Adapter Layer)接口规范（可以对接到不同厂商的AT模组）。
- **SAL Driver**：驱动，部分（如sim800、M5310）由AliOS Things提供（上图绿色），其他由用户自己提供（上图红色）。SAL驱动模块基于具体型号的通信模组提供的AT命令，实现SAL规范的HAL接口功能。

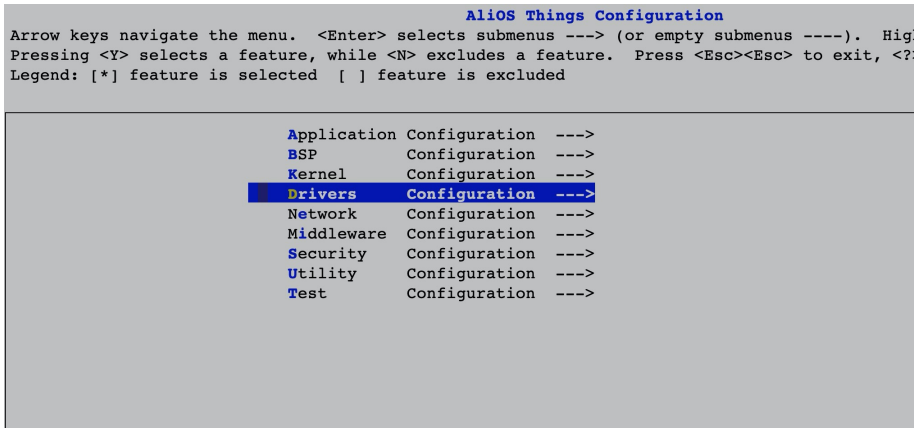
API列表

名称	说明
socket	创建套接字，返回文件描述符。
connect	与远端服务器建立一个连接。
select	查询一个或者多个socket的可读性、可写性及错误状态信息。
gethostbyname	域名解析，获取主机域名对应的IP，不可重入。
getaddrinfo	域名解析，获取主机域名对应的IP，可重入。
freeaddrinfo	释放addrinfo结构体，一般与getaddrinfo配合使用。
send	向远端发送数据。
recv	接收远端发送的数据。
sendto	无连接模式下的发送数据。

recvfrom	无连接模式下的接收数据。
close	关闭socket，释放相关资源。
sal_init	SAL模块初始化，包括初始化底层驱动模块。
sal_add_dev	配置驱动参数，例如串口通信串口号、波特率等，并添加设备。

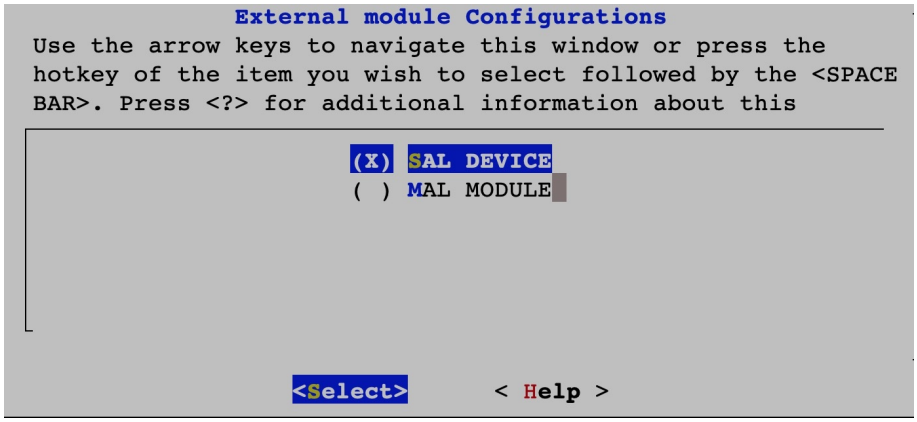
使用

添加该组件

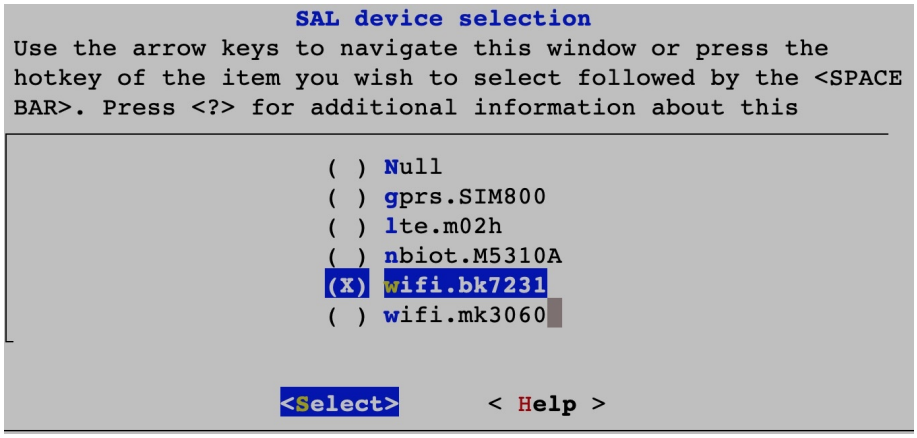


在Drivers里面选择"External module enable"

在External module Configurations里选择"SAL DEVICE"。



在SAL device selection里选择具体的外接模组。例如，使用WiFi模组bk7231，则选择wifi.bk7231。



头文件

对外头文件代码位于 `include/network/sal` ，包括

```
sal/sal_arch.h
sal/sal_def.h
sal/sal_ipaddr.h
sal/sal_sockets.h
```

使用时只需包含

```
#include <network/network.h>
```

使用示例

SAL提供标准socket的API，编程方式也按照通用的socket编程。例如，与远端建立TCP连接并发送数据：

```
/* 域名解析 */
if ((rc = getaddrinfo(server_domain, servname, &hints, &addrInfoList)) != 0) {
    LOGE(TAG, "getaddrinfo error: %d, errno = %d", rc, errno);
    return;
}
for (cur = addrInfoList; cur != NULL; cur = cur->ai_next) {
    if (cur->ai_family != AF_INET) {
        LOGE(TAG, "Socket type error");
        continue;
    }
     /* 创建socket */
    fd = socket(cur->ai_family, cur->ai_socktype, cur->ai_protocol);
    if (fd < 0) {
        LOGE(TAG, "Failed to create socket, errno = %d", errno);
        continue;
    }
    /* 与远端连接 */
    if (connect(fd, cur->ai_addr, cur->ai_addrlen) == 0) {
        break;
    } else {
        LOGE(TAG, "Failed to connect addr, errno = %d", errno);
    }
    close(fd);
}
 /* 向远端发送数据 */
if (send(fd, tcp_payload, strlen(tcp_payload), 0) <= 0) {
    LOGE(TAG, "Failed to send data, errno = %d", errno);
    goto ret;
}
```

更详细的例子请参考 application/example/example_legacy/sal_app

API 详情

socket

原型

```
int socket(int domain, int type, int protocol);
```

接口说明

使用 SAL socket 通信之前，使用该函数创建套接字，返回文件描述符。

参数说明

参数	数据类型	方向	说明
domain	int	输入	创建的套接字指定协议集，目前支持AF_INET
type	int	输入	socket类型，目前支持SOCK_STREAM和SOCK_DGRAM

参数	数据类型	方向	说明
protocol	int	输入	实际使用的传输协议，默认为0

返回值说明

值	说明
非负值	成功，即文件描述符，后续socket操作使用该值
负值	失败

接口示例

```
/* 创建UDPsocket */
int fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd < 0){
  LOGE(TAG, "Failed to create socket, errno = %d", errno);
  return;
} else {
  LOGD(TAG, "UDP socket create OK");
}
```

connect

原型

```
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

接口说明

用来与远端服务器建立一个连接。

参数说明

参数	数据类型	方向	说明
s	int	输入	socket文件描述符
name	struct sockaddr *	输入	指向 sockaddr 结构的指针，存放要连接的服务器的 IP 地址和端口号等信息
namelen	int	输入	sockaddr 结构体的长度

返回值说明

值	说明
0	连接成功
非0	失败

接口示例

```
/* 与远端建立连接 */
if (connect(fd, ai_addr, ai_addrlen) == 0) {
  break;
} else {
  LOGE(TAG, "Failed to connect addr, errno = %d", errno);
}
```

select

原型

```
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
      struct timeval *timeout);
```

接口说明

用来查询一个或者多个socket的可读性、可写性及错误状态信息。

参数说明

参数	数据类型	方向	说明
maxfdp1	int	输入	最大文件描述符值加1
readset	fd_set *	输入	（可选）指向一组等待可读性检查的套接口
writeset	fd_set *	输入	（可选）指向一组等待可写性检查的套接口
exceptset	fd_set *	输入	（可选）指向一组等待错误检查的套接口
timeout	struct timeval *	输入	最长等待时间，阻塞操作则为NULL

返回值说明

值	说明
正值	有读、写、错误事件
0	超时
负值	出错，具体错误通过errno获取

接口示例

```
FD_ZERO(&sets);
FD_SET(fd, &sets);
/* 对write set进行等待 */
ret = select(fd + 1, NULL, &sets, NULL, &timeout);
if (ret > 0) {
    if (0 == FD_ISSET(fd, &sets)) {
        ret = 0;
        continue;
    }
   /* 发送数据 */
   send(fd, buf + len_sent, len - len_sent, 0);
} else if (0 == ret) {
    break;
}
```

gethostbyname

原型

```
struct hostent *gethostbyname(const char *name);
```

接口说明

域名解析，获取主机域名对应的IP。

参数说明

参数	数据类型	方向	说明
name	const char *	输入	主机域名

返回值说明

值	说明
非NULL	成功
NULL	失败

其中，hostent结构体定义见标准宏和结构体说明。

接口示例

```
struct sockaddr_in server_addr;
struct hostent *host;
/* 域名解析 */
if ((host = gethostbyname(host_addr)) == NULL) {
    LOGE("Gethostname  error,  %s\n ", strerror(errno));
    return -1;
}
bzero(&server_addr, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port   = htons(port);
server_addr.sin_addr   = *((struct in_addr *)host->h_addr);
/* 连接远端 */
connect(sockfd, (struct sockaddr *)(&server_addr), sizeof(struct sockaddr);
```

getaddrinfo

原型

```
int getaddrinfo(const char *nodename, const char *servname,
        const struct addrinfo *hints, struct addrinfo **res);
```

接口说明

域名解析，获取主机域名对应的IP。

参数说明

参数	数据类型	方向	说明
nodename	const char *	输入	主机域名
servname	const char *	输入	（可选）端口号字符传，

hints	const struct addrinfo *	输入	addrinfo结构体的指针，在这个结构中填入关于期望返回的信息类型的暗示，例如
			hints.ai_family = AF_UNSPEC;
			hints.ai_socktype = SOCK_STREAM;

res	struct addrinfo **	输出	返回解析地址

返回值说明

值	说明
0	成功
非0	失败

接口示例

```
struct addrinfo  hints;
hints.ai_family   = AF_INET; // only IPv4
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
sprintf(service, "%u", port);
/* 域名解析 */
if ((rc = getaddrinfo(host, service, &hints, &addrInfoList)) != 0) {
    return (uintptr_t)-1;
}
for (cur = addrInfoList; cur != NULL; cur = cur->ai_next) {
  if (cur->ai_family != AF_INET) {
      rc = -1;
      continue;
  }
  /* 创建socket */
  fd = socket(cur->ai_family, cur->ai_socktype, cur->ai_protocol);
  if (fd < 0) {
   rc = -1;
   continue;
  }
  /* 连接 */
  if (connect(fd, cur->ai_addr, cur->ai_addrlen) == 0) {
      rc = fd;
      break;
  }
  close(fd);
}
```

freeaddrinfo

原型

```
void freeaddrinfo(struct addrinfo *ai);
```

接口说明

释放addrinfo结构体，一般与getaddrinfo配合使用。

参数说明

参数	数据类型	方向	说明
ai	struct addrinfo *	输入	addrinfo结构体指针

返回值说明

无

接口示例

```
/* 域名解析 */
getaddrinfo(host, service, &hints, &addrInfoList)
/* 释放addrinfo结构体 */
freeaddrinfo(addrInfoList);
```

send

原型

```
int send(int s, const void *data, size_t size, int flags);
```

接口说明

向远端发送数据。

参数说明

参数	数据类型	方向	说明
s	int	输入	socket 文件描述符
data	const void *	输入	发送数据缓存指针
size	size_t	输入	发送数据字节数
flag	int	输入	控制选项，通常为 0

返回值说明

值	说明
正值	成功，发送长度
非正值	失败

接口示例

```
/* 向已连接的远端发送数据 */
if (send(fd, tcp_payload, strlen(tcp_payload), 0) <= 0) {
  LOGE(TAG, "Failed to send data, errno = %d", errno);
  goto ret;
} else {
  LOGD(TAG, "TCP socket send to server OK");
}
```

recv

原型

```
int recv(int s, void *mem, size_t len, int flags);
```

接口说明

接收远端发送的数据。

参数说明

参数	数据类型	方向	说明
s	int	输入	socket 文件描述符
mem	const void *	输出	接收数据缓存指针
len	size_t	输入	接收缓存大小

参数	数据类型	方向	说明
flags	int	输入	控制选项，通常为 0

返回值说明

值	说明
正值	成功，接收长度
非正值	失败

接口示例

```
while (total_received < HTTP_BUFF_SIZE) {
  /* 从远端接收数据 */
  bytes_received =
    recv(sockfd, buffer, (HTTP_BUFF_SIZE - total_received), 0);
  if (bytes_received == -1) {
    return -1;
  } else if (bytes_received == 0) {
    return 0;
  } else {
   buffer = buffer + bytes_received;
  }
  total_received += bytes_received;
}
```

sendto

原型

```
int sendto(int s, const void *data, size_t size, int flags, const struct sockaddr *to, socklen_t tolen);
```

接口说明

无连接的数据报 socket 模式下发送数据。

参数说明

参数	数据类型	方向	说明
s	int	输入	socket 文件描述符
data	const void *	输入	发送数据缓存指针
size	size_t	输入	发送数据字节数
flags	int	输入	控制选项，通常为 0
to	const struct sockaddr *	输入	指向 sockaddr 结构体的指针，存放目的主机的 IP 和端口号

参数	数据类型	方向	说明
tolen	socklen_t	输入	sockaddr 结构体的长度

返回值说明

值	说明
正值	成功，发送长度
非正值	失败

接口示例

```
/* 向远端发送UDP数据 */
ret = sendto(fd, udp_payload, strlen(udp_payload), 0, (struct sockaddr*)&addr, sizeof(addr));
if (ret < 0){
  LOGE(TAG, "udp sendto failed, errno = %d", errno);
  close(fd);
  return;
} else {
  LOGD(TAG, "UDP socket sendto OK");
}
```

recvfrom

原型

```
int recvfrom(int s, void *mem, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen
```

接口说明

无连接的数据报 socket 模式下接收数据。

参数说明

参数	数据类型	方向	说明
s	int	输入	socket 文件描述符
mem	const void *	输出	接收数据缓存指针
len	size_t	输入	接收缓存大小
flags	int	输入	控制选项，通常为 0
from	struct sockaddr *	输出	指向 sockaddr 结构体的指针，存放源主机的 IP 和端口号
fromlen	socklen_t *	输出	指向 sockaddr 结构体的长度的指针

返回值说明

值	说明
正值	成功，接收长度
非正值	失败

接口示例

```
/* 从远端接收UDP数据 */
ret = recvfrom(fd, recvbuf, sizeof(recvbuf), 0, (struct sockaddr*)&addr, &recvaddrlen);
if (ret < 0) {
  LOGE(TAG, "udp sendto failed, errno = %d", errno);
  close(fd);
  return;
} else {
  LOGD(TAG, "%d bytes data received.", ret);
}
```

close

原型

```
int sal_close(int s);
```

接口说明

关闭socket，释放相关资源。

参数说明

参数	数据类型	方向	说明
s	int	输入	socket 文件描述符

返回值说明

值	说明
0	成功
负值	失败

接口示例

```
/* 创建socket */
fd = socket(AF_INET, SOCK_DGRAM, 0);
/* 释放socket */
close(fd);
```

sal_init

原型

```
int sal_init(void);
```

接口说明

SAL模块初始化，包括初始化底层驱动模块。

参数说明

无

返回值说明

值	说明
0	成功
负值	失败

接口示例

```
sal_device_config_t data = {0};
/*
 * 设置外接模组连接参数：
 * UART配置:
 *   - 115200
 *   - 8n1
 *   - no flow control
 *   - tx/rx mode
 */
data.uart_dev.port = 1;
data.uart_dev.config.baud_rate = 115200;
data.uart_dev.config.data_width = DATA_WIDTH_8BIT;
data.uart_dev.config.parity = NO_PARITY;
data.uart_dev.config.stop_bits  = STOP_BITS_1;
data.uart_dev.config.flow_control = FLOW_CONTROL_DISABLED;
data.uart_dev.config.mode = MODE_TX_RX;
/* 配置驱动参数 */
if (sal_add_dev("bk7231", &data) != 0) {
   LOG("Failed to add SAL device!");
   return -1;
}
/* 初始化SAL core */
sal_init();
```

sal_add_dev

原型

```
int sal_add_dev(char* driver_name, void* data);
```

接口说明

配置驱动参数，例如串口通信串口号、波特率等。

参数说明

参数	数据类型	方向	说明
driver_name	char *	输入	驱动名称

参数	数据类型	方向	说明
data	void*	输入	配置参数指针

返回值说明

值	说明
0	成功
负值	失败

接口示例

见sal_init示例

配置说明

SAL可配置项包括：

- 是否定义WITH_SAL宏，默认为是；
- 是否使用AOS HAL，默认为是；
- 是否开启SAL debug打印，默认为否；
- 配置SAL接收缓存大小，默认为32；

移植说明

SAL模块需要实现两类HAL，一类为模组连接操作HAL；另一类为OS基础HAL。

模组连接操作HAL

该类HAL头文件为

```
#include "hal_sal.h"
```

HAL函数以函数指针的方式，挂载在sal_opt_s结构体中。

add_dev

原型

```
int (*add_dev)(void* data);
```

接口说明

配置参数，添加设备。

参数说明

参数	数据类型	方向	说明
data	void*	输入	配置参数指针

返回值说明

值	说明
0	成功
负值	失败

init

原型

```
int (*init)(void);
```

接口说明

初始化底层驱动

参数说明

无

返回值说明

值	说明
0	成功
负值	失败

start

原型

```
int (*start)(sal_conn_t *c);
```

接口说明

通过模组与远端建立socket连接。

参数说明

参数	数据类型	方向	说明
c	sal_conn_t*	输入	连接参数指针

返回值说明

值	说明
0	成功
负值	失败

send_data

原型

```
int (*send_data)(int fd, uint8_t *data, uint32_t len,
        char remote_ip[16], int32_t remote_port, int32_t timeout);
```

接口说明

通过模组向远端已建立socket连接发送数据。

参数说明

参数	数据类型	方向	说明
fd	int	输入	socket文件描述符
data	uint8_t*	输入	发送缓存指针
len	uint32_t	输入	发送长度
remote_ip	char []	输入	远端IP地址
remote_port	int32_t	输入	远端端口
timeout	int32_t	输入	超时时间，毫秒

返回值说明

值	说明
0	成功
负值	失败

domain_to_ip

原型

```
int (*domain_to_ip)(char *domain, char ip[16]);
```

接口说明

域名解析

参数说明

参数	数据类型	方向	说明
domain	char *	输入	主机域名
ip	char []	输出	IP地址

返回值说明

值	说明
0	成功
负值	失败

finish

原型

```
int (*finish)(int fd, int32_t remote_port);
```

接口说明

关闭socket连接

参数说明

参数	数据类型	方向	说明
fd	int	输入	socket文件描述符
remote_port	int32_t	输入	远端port号

返回值说明

值	说明
0	成功
负值	失败

deinit

原型

```
int (*deinit)(void);
```

接口说明

驱动模块去初始化，释放相关资源。

参数说明

无

返回值说明

值	说明
0	成功
负值	失败

register_netconn_data_input_cb

原型

```
int (*register_netconn_data_input_cb)(netconn_data_input_cb_t cb);
```

接口说明

注册数据接收回调函数。

参数说明

无

返回值说明

值	说明
0	成功
负值	失败

其中，

netconn_data_input_cb_t

定义如下

```
int (*netconn_data_input_cb_t)(int fd, void *data, size_t len, char remote_ip[16], uint16_t remote_port);
```

接口说明

接收数据回调函数。

参数说明

参数	数据类型	方向	说明
fd	int	输入	socket文件描述符
data	void *	输入	接收数据缓存指针
len	size_t	输入	数据长度
remote_ip	char []	输入	数据源IP
remote_port	uint16_t	输入	远端port

返回值说明

值	说明
0	成功
负值	失败

OS基础HAL

接口	描述
sal_malloc	malloc, 入参： s：需要分配的内存大小 返回值：非NULL，分配的内存地址 NULL，分配内存失败

sal_free	free，入参： p: 需要释放的内存指针 返回值：无
sal_msleep	sleep，入参： ms: 毫秒数 返回值：无
sal_sem_new	创建信号量，入参： sem：填充创建信号的地址 count：初始信号量值 返回值：0成功，-1错误
sal_sem_free	销毁信号量，入参： sem：信号量地址 返回值：无
sal_sem_signal	释放信号量，入参： sem: 信号量地址 返回值：无
sal_sem_valid	检查信号量是否合法：入参： sem: 信号量地址 返回值：1合法，0非法
sal_arch_sem_wait	等待信号量： sem：信号量地址 timeout：超时时间 返回值：(~0)为超时，其他等待时间
sal_mbox_new	创建mbox: mb：填充创建mbox地址 size：mbox大小 返回值：0成功，-1错误
sal_mbox_free	销毁mbox: mb：mbox地址 返回值：无

sal_mbox_post	向mbox发送数据： mb：mbox地址 msg：数据地址 返回值：空
sal_mbox_trypost	向mbox发送数据，并返回是否成功： mb：mbox地址 msg：数据地址 返回值：0成功，-1失败
sal_mbox_valid	检查mbox是否合法： mb：mbox地址 返回值：1合法，0非法
sal_arch_mbox_fetch	从mbox获取数据： mb：mbox地址 msg：用于填充数据地址 timeout：超时时间 返回：(~0)为超时，其他等待时间
sal_arch_mbox_tryfetch	从mbox获取数据，并返回是否成功： mb：mbox地址 msg：用于填充数据地址 返回：0成功，-1失败
sal_mutex_new	创建锁： mutex：用于填充创建锁地址 返回：0成功，-1失败
sal_mutex_lock	上锁： mutex：锁地址 返回：无
sal_mutex_unlock	解锁： mutex：锁地址 返回：无

sal_mutex_free	销毁锁： mutex：锁地址 返回：无
sal_mutex_valid	判断锁是否合法： mutex：锁地址 返回值：1合法，0非法

标准宏和结构体说明

hostent结构体定义

```
struct hostent {
  char  *h_name;    /* 主机正式域名 */
  char **h_aliases;  /* 主机的别名数组 */
  int   h_addrtype; /* 协议类型，对于 TCP/IP 为 AF_INET  */
  int   h_length;   /* 协议的字节长度，对于 IPv4 为 4 个字节 */
  char **h_addr_list; /* 地址的列表*/
#define h_addr h_addr_list[0] /* 保持向后兼容 */
};
```

sal_op_t结构体定义

```
typedef struct sal_op_s {
  struct sal_op_s * next; /* 下一个sal_op_s */
  char *version; /* 版本信息 */
  char *name; /* 外接模组名称 */
  /* 添加模组 */
  int (*add_dev)(void*);
  /* 模组初始化 */
  int (*init)(void);
  /* 创建TCP/UDP连接 */
  int (*start)(sal_conn_t *c);
  /* 向远端发送数据 */
  int (*send_data)(int fd, uint8_t *data, uint32_t len,
        char remote_ip[16], int32_t remote_port, int32_t timeout);
  /* 主动接收远端数据 */
  int (*recv_data)(int fd, uint8_t *data, uint32_t len,
        char remote_ip[16], int32_t remote_port);
  /* 域名解析 */
  int (*domain_to_ip)(char *domain, char ip[16]);
  /* 关闭远端连接 */
  int (*finish)(int fd, int32_t remote_port);
  /* SAL去初始化 */
  int (*deinit)(void);
   /* 数据接收回调 */
  int (*register_netconn_data_input_cb)(netconn_data_input_cb_t cb);
} sal_op_t;
```

sal_conn_t结构体定义

```
typedef struct {
  int fd;           /* 所用socket文件描述符 */
  CONN_TYPE type;       /* 连接类型，如tcp_client、udp_client */
  char *addr;          /* 远端地址 */
  int32_t r_port;       /* 远端端口号 */
  int32_t l_port;       /* 本地端口号，如果不使用，设置为-1 */
  uint32_t tcp_keep_alive; /* tcp连接保活时间 */
} sal_conn_t;
```

2.4.12. CoAP组件文档

CoAP协议（RFC 7252)使用RESTful服务框架，通过统一的资源标识、统一的操作接口、可自描述的消息，实现物联网设备端与服务端所需交互。AliOS Things中CoAP组件的结构图如下。

其包括：

- **CoAP Core**：CoAP核心模块，主要包括上下文处理、会话处理、DTLS适配、IO适配。 该模块实现CoAP主流程，对外提供标准的CoAP协议行为。
- **Adpater**：连云适配模块，主要包括连接上下文、认证、消息处理。该模块对上提供连接云平台所需API。用户可以基于该API实现数据上报的应用。

API列表

名称	说明
IOT_CoAP_Init	初始化CoAP上下文
IOT_CoAP_Deinit	去初始化CoAP上下文。
IOT_CoAP_DeviceNameAuth	与服务端对设备三元组信息进行认证。
IOT_CoAP_Yield	处理远端数据与发送响应超时。
IOT_CoAP_SendMessage	向远端topic发送数据。
IOT_CoAP_GetMessagePayload	提取消息体。
IOT_CoAP_GetMessageCode	提取回复消息码。

使用

添加该组件

在使用CoAP的组件或应用对应的aos.mk中添加

```
$(NAME)_COMPONENTS += libcoap
```

在使用CoAP的组件或应用对应的Config.in中添加

```
select AOS_COMP_LIBCOAP if !AOS_CREATE_PROJECT
```

头文件

对外头文件代码位于 `include/network/coap` ，包括

```
coap.h
```

使用时只需包含

```
#include <coap.h>
```

使用示例

```
/*初始化CoAP上下文*/
p_ctx = IOT_CoAP_Init(&config);
if (NULL != p_ctx) {
  /*与云端进行认证*/
  if (IOT_CoAP_DeviceNameAuth(p_ctx) == IOTX_SUCCESS) {
    do {
      /*接收数据，并处理超时*/
      IOT_CoAP_Yield(p_ctx);
    } while (m_coap_client_running);
  } else {
    LOGE(TAG, "CoAP authentication failed");
  }
  /*去初始化*/
  IOT_CoAP_Deinit(&p_ctx);
}
```

更详细的例子请参考 application/example/coap_demo

API 详情

IOT_CoAP_Init

原型

```
iotx_coap_context_t *IOT_CoAP_Init(iotx_coap_config_t *p_config);
```

接口说明

根据配置参数，初始化CoAP上下文，创建DTLS Session。

参数说明

参数	数据类型	方向	说明
p_config	iotx_coap_config_t *	输入	CoAP配置参数

其中，iotx_coap_context_t为void，数据类型iotx_coap_config_t定义见标准宏和结构体说明。

返回值说明

值	说明
非NULL	成功，CoAP上下文句柄指针
NULL	失败

接口示例

```
iotx_coap_config_t    config;
iotx_coap_context_t *p_ctx = NULL;
/* 设置参数 */
memset(&config, 0, sizeof(iotx_coap_config_t));
config.p_url = url;
config.p_devinfo = (iotx_coap_device_info_t *)&deviceinfo;
config.wait_time_ms = 3000;
/* 根据参数，初始化上下文 */
p_ctx = IOT_CoAP_Init(&config);
```

IOT_CoAP_Deinit

原型

```
void IOT_CoAP_Deinit(iotx_coap_context_t **p_context);
```

接口说明

去初始化CoAP上下文

参数说明

参数	数据类型	方向	说明
p_context	iotx_coap_context_t **	输入	CoAP上下文句柄指针的地址

返回值说明

无

接口示例

```
iotx_coap_config_t    config;
iotx_coap_context_t *p_ctx = NULL;
/* 根据参数，初始化上下文 */
p_ctx = IOT_CoAP_Init(&config);
/* 去初始化 */
IOT_CoAP_Deinit(&p_ctx);
```

IOT_CoAP_DeviceNameAuth

原型

```
int IOT_CoAP_DeviceNameAuth(iotx_coap_context_t *p_context);
```

接口说明

与服务端对设备三元组信息进行认证。

参数说明

参数	数据类型	方向	说明
p_context	iotx_coap_context_t *	输入	CoAP上下文句柄指针

返回值说明

值	说明
0	成功
负值	失败，参考iotx_ret_code_t定义

其中，iotx_ret_code_t定义见标准宏和结构体说明。

接口示例

```
iotx_coap_config_t    config;
iotx_coap_context_t *p_ctx = NULL;
/* 根据参数，初始化上下文 */
p_ctx = IOT_CoAP_Init(&config);
/* 认证设备信息 */
IOT_CoAP_DeviceNameAuth(p_ctx);
```

IOT_CoAP_Yield

原型

```
int IOT_CoAP_Yield(iotx_coap_context_t *p_context);
```

接口说明

处理远端数据与发送响应超时。

参数说明

参数	数据类型	方向	说明
p_context	iotx_coap_context_t *	输入	CoAP上下文句柄指针

返回值说明

值	说明
非负	成功
负值	失败

接口示例

```
if (IOT_CoAP_DeviceNameAuth(p_ctx) == IOTX_SUCCESS) {
  do {
    /* 处理远端数据与请求超时 */
    IOT_CoAP_Yield(p_ctx);
  } while (m_coap_client_running);
}
```

IOT_CoAP_SendMessage

原型

```
int IOT_CoAP_SendMessage(iotx_coap_context_t *p_context, char *p_path, iotx_message_t *p_message);
```

接口说明

向远端topic发送数据

参数说明

参数	数据类型	方向	说明
p_context	iotx_coap_context_t *	输入	CoAP上下文句柄指针
p_path	char *	输入	topic路径
p_message	iotx_message_t *	输入	需要发生的消息

返回值说明

值	说明
0	成功
负值	失败，参考iotx_ret_code_t定义

接口示例

```
iotx_message_t message;
/* 设置发送topic*/
snprintf(path, IOTX_URI_MAX_LEN, "/topic/%s/%s/user/update", IOTX_PRODUCT_KEY, IOTX_DEVICE_NAME);
/* 配置发送消息 */
memset(&message, 0, sizeof(iotx_message_t));
message.p_payload = (unsigned char *)"{\"name\":\"hello world\"}";
message.payload_len = strlen("{\"name\":\"hello world\"}");
message.resp_callback = iotx_response_handler;
message.msg_type = IOTX_MESSAGE_CON;
message.content_type = IOTX_CONTENT_TYPE_JSON;
/* 发送消息 */
IOT_CoAP_SendMessage(p_ctx, path, &message);
```

IOT_CoAP_GetMessagePayload

原型

```
int IOT_CoAP_GetMessagePayload(void *p_message, unsigned char **pp_payload, int *p_len);
```

接口说明

提取消息体

参数说明

参数	数据类型	方向	说明
p_message	void *	输入	消息指针
pp_payload	unsigned char **	输出	存储消息体指针的地址
p_len	int *	输出	消息体长度

返回值说明

值	说明
0	成功
负值	失败，参考iotx_ret_code_t定义

接口示例

```
/* 响应消息处理回调 */
static void iotx_response_handler(void *arg, void *p_response)
{
  int       len     = 0;
  unsigned char *p_payload = NULL;
  /* 提取消息体内容 */
  IOT_CoAP_GetMessagePayload(p_response, &p_payload, &len);
}
```

IOT_CoAP_GetMessageCode

原型

```
int IOT_CoAP_GetMessageCode(void *p_message, iotx_coap_resp_code_t *p_resp_code);
```

接口说明

提取回复消息码

参数说明

参数	数据类型	方向	说明
p_message	void *	输入	消息指针
p_resp_code	iotx_coap_resp_code_t *	输出	消息码

其中，iotx_coap_resp_code_t定义见标准宏和结构体说明。

返回值说明

值	说明
0	成功
负值	失败，参考iotx_ret_code_t定义

```
/* 响应消息处理回调 */
static void iotx_response_handler(void *arg, void *p_response)
{
  iotx_coap_resp_code_t resp_code;
  /* 提取消息体码 */
  IOT_CoAP_GetMessageCode(p_response, &resp_code);
}
```

配置说明

CoAP可配置项包括：

- 是否使用DTLS，默认为使用DTLS
- 是否使用阿里云IOT平台PSK认证方式，默认使用阿里云IOT平台PSK认证方式
- 是否开启Debug输出，默认开启Debug输出
- 是否定义BUILD_AOS，默认为定义BUILD_AOS宏

移植说明

CoAP模块需要实现连接wrapper、系统wrapper、认证wrapper。

连接wrapper

接口	描述
ssize_t coap_network_send(coap_socket_t *sock, const struct coap_session_t *session, const uint8_t *data, size_t datalen)	数据发送 sock：CoAP socket上下文 session：CoAP session data：需要发送数据地址 datalen：数据长度 返回值：>0发送长度，-1失败
ssize_t coap_network_read(coap_socket_t *sock, struct coap_packet_t *packet)	数据接收 sock：CoAP socket上下文 packet：接收数据结构体 返回值：>0接收长度，-1失败
int coap_socket_connect_udp(coap_socket_t *sock, const coap_address_t *local_if, const coap_address_t *server, int default_port, coap_address_t *local_addr, coap_address_t *remote_addr);	建立UDP与远端的连接 sock：CoAP socket上下文 local_if：本地地址 server：远端地址 default_port：默认端口号 local_addr：实际使用本地地址 remote_addr：实际远端的地址 返回值：0发送长度，-1失败
void coap_socket_close(coap_socket_t *sock)	关闭连接 client：HTTP上下文 返回值：HTTPC_RESULT

int coap_dtls_send(coap_session_t *coap_session, const uint8_t *data, size_t data_len);	发送dtls数据 c_session：coap session data：需要发送的数据地址 data_len：数据长度 返回值：>0发送长度，-1失败
int coap_dtls_receive(coap_session_t *coap_session, const uint8_t *data, size_t data_len);	接收dtls数据 c_session：coap session data：接收缓存地址 data_len：缓存大小 返回值：>0接收长度，-1失败
void *coap_dtls_new_client_session(coap_session_t *coap_session);	创建客户端dtls session c_session：coap session 返回值：非空 dlts session，NULL失败
void coap_dtls_free_session(coap_session_t *coap_session);	释放dtls session 返回值：无

系统wrapper

接口	描述
void *coap_malloc_type(coap_memory_tag_t type, size_t size);	分配空间 type：类型 size：分配大小 返回值：分配地址，NULL失败
void coap_free_type(coap_memory_tag_t type, void *p);	释放空间type：类型 p：内存地址 返回值：无
void *coap_realloc(void *mem, unsigned int size)	重分配mem：内存地址 size：重新分配的大小 返回值：分配地址，NULL失败

认证wrapper

接口	描述

void * coap_wrapper_cjson_parse(const char *src);	解析json src：原字符串地址 返回：解析后json地址，NULL失败
void * coap_wrapper_cjson_object_item(void *root, const char *key)	获取key对应的json节点 root：json地址 key：key值字段 返回值：json节点地址，NULL失败
char * coap_wrapper_cjson_value_string(void *node)	获取json节点字符串 node：json节点地址 返回值：字符串地址，NULL失败
int coap_wrapper_cjson_value_int(void *node)	获取json节点整型值 node：json节点地址 返回值：整型值
void coap_wrapper_cjson_release(void *root)	释放json root：json地址
void coap_wrapper_hmac_md5(const char *msg, int msg_len, char *digest, const char *key, int key_len);	对字符串md5加密 msg：字符串地址 msg_len：字符串长度 digest：生成的摘要 key：密钥 key_len：密钥长度 返回值：无
void * coap_wrapper_aes128_init(const unsigned char *key, const unsigned char *iv, bool is_encrypt)	aes128初始化 key：密钥地址 IV：IV地址 is_encrypt：是否加密 返回值：aes上下文
int coap_wrapper_aes128_destroy(void *aes)	销毁aes上下文： aes：aes上下文地址 返回值：0成功；1失败

int coap_wrapper_aes128_cbc_decrypt(void *aes, const void *src, unsigned int blockNum,void *dst);	aes128解密 aes：aes上下文地址 src：源地址 blockNum：块数量 dst：目标地址 返回值：0成功；1失败
int coap_wrapper_aes128_cbc_encrypt(void *aes, const void *src, unsigned int blockNum, void *dst);	aes128加密 aes：aes上下文地址 src：源地址 blockNum：块数量 dst：目标地址 返回值：0成功；1失败
void coap_wrapper_sha256(const unsigned char *input, unsigned int ilen, unsigned char output[32]);	sha256加密 input：源地址 ilen：输入长度 output:目标地址 返回值：无

标准宏和结构体说明

iotx_coap_config_t结构体定义

```
typedef struct {
  char            *p_url;      /* 连接地址URL */
  int             wait_time_ms; /* 连接等待时间，单位毫秒 */
  iotx_coap_device_info_t *p_devinfo;    /* 设备四元组信息 */
  iotx_event_handle_t     event_handle; /* 事件回调 */
} iotx_coap_config_t;
```

iotx_ret_code_t枚举定义

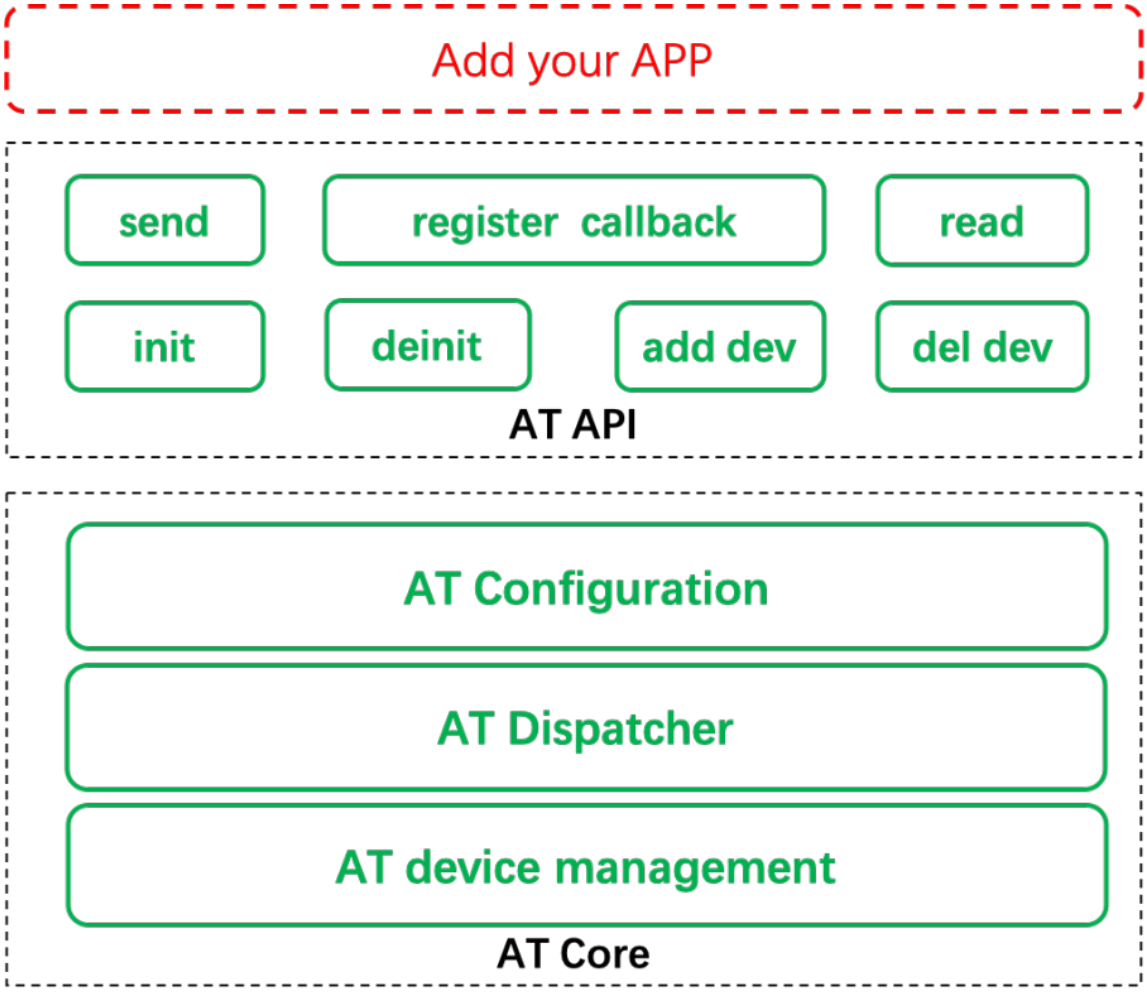
```
typedef enum {
  IOTX_ERR_RECV_MSG_TIMEOUT = -9, /* 接收消息超时 */
  IOTX_ERR_SEND_MSG_FAILED = -8, /* 发送消息失败 */
  IOTX_ERR_MSG_TOO_LOOG    = -7, /* 消息体太长 */
  IOTX_ERR_URI_TOO_LOOG    = -6, /* topic URI太长 */
  IOTX_ERR_NOT_AUTHED      = -5, /* 设备未认证 */
  IOTX_ERR_AUTH_FAILED     = -4, /* 设备认证失败 */
  IOTX_ERR_BUFF_TOO_SHORT  = -3, /* 缓存太小 */
  IOTX_ERR_NO_MEM          = -2, /* 内存分配失败 */
  IOTX_ERR_INVALID_PARAM   = -1, /* 参数非法 */
  IOTX_SUCCESS             = 0, /* 成功 */
} iotx_ret_code_t;
```

iotx_coap_resp_code_t枚举定义

```
typedef enum {
  IOTX_COAP_RESP_CODE_CONTENT     = 0x45,   /* 对应2.05, 内容 */
  IOTX_COAP_RESP_CODE_BAD_REQUEST  = 0x80,   /* 对应4.00, 非法请求 */
  IOTX_COAP_RESP_CODE_UNAUTHORIZED  = 0x81,   /* 对应4.01, token非法或过期 */
  IOTX_COAP_RESP_CODE_NOT_FOUND    = 0x84,   /* 对应4.04, Path或URI不存在 */
  IOTX_COAP_RESP_CODE_URL_TOO_LONG  = 0x8E,   /* 对应4.14, URI太长 */
  IOTX_COAP_RESP_CODE_INTERNAL_SERVER_ERROR = 0xA0,/* 对应5.00, 服务器内部错误 */
} iotx_coap_resp_code_t;
```

2.4.13. AT组件文档

AT组件提供了MCU与外接通信模组间基于AT命令的基础通信能力，是物联网应用场景中不可或缺的模块之一。基于该组件，用户可以只关注AT指令的语义，无需关心AT指令的收发过程，简化外接模组的AT通信驱动的开发。下图是AliOS Things中AT组件的结构示意图。其包括：



- **AT Core**：AT组件核心模块，主要包括AT配置处理、AT接收分发，AT设备管理。该模块实现AT接收、发处理、添加AT设备主流程。
- **AT API**：AT组件API，主要包括发送、注册回调、数据读取、初始化、去初始化、添加设备、删除设备。该模块封装了AT核心模块的功能，简化了用户的调用方式。

API列表

名称	说明
at_init	初始化AT组件
at_deinit	去初始化AT组件
at_add_dev	根据配置添加AT设备
at_delete_dev	根据设备描述符删除AT设备
at_send_wait_reply	在用户task中，使用该接口发送数据，并等待回复
at_send_no_reply	使用该接口发送数据，但不等待回复
at_read	在回调函数里，从底层读取AT字符流
at_register_callback	注册AT接收处理回调函数
at_yield	单task情况下，从底层接收数据

使用

添加该组件

在使用at的组件或应用对应的aos.mk中添加

```
$(NAME)_COMPONENTS += at
```

在使用at的组件或应用对应的Config.in中添加

```
select AOS_COMP_AT if !AOS_CREATE_PROJECT
```

头文件

对外头文件代码位于 `include/utility/at` ，包括

```
at.h
```

使用时只需包含

```
#include "at/at.h"
```

使用示例

```
/* 初始化 */
  at_init();
  /**
   * 配置AT设备参数
   */
  uart_dev.port          = 1;
  uart_dev.config.baud_rate   = 9600;
  uart_dev.config.data_width  = DATA_WIDTH_8BIT;
  uart_dev.config.parity     = NO_PARITY;
  uart_dev.config.stop_bits   = STOP_BITS_1;
  uart_dev.config.flow_control = FLOW_CONTROL_DISABLED;
  uart_dev.config.mode       = MODE_TX_RX;
  at_config.type                  = AT_DEV_UART;
  at_config.port                  = uart_dev.port;
  at_config.dev_cfg               = &uart_dev;
  at_config.send_wait_prompt          = 0;
  /* 添加AT设备，成功后返回设备fd */
  if ((at_dev_fd = at_add_dev(&at_config)) < 0) {
    LOGE(TAG, "AT parser device add failed!\n");
    return -1;
  }
  /* 注册感兴趣前缀对应的回调处理 */
  at_register_callback(at_dev_fd, "+NSONMI:", NULL, NULL, 0,
                m5310a_socket_data_indicator_handler, NULL);
  /* 发送AT test命令 */
  memset(cmd, 0, sizeof(cmd));
  snprintf(cmd, sizeof(cmd) - 1, "%s", "AT");
  /* 发送并等待回复 */
  at_send_wait_reply(at_dev_fd, cmd, strlen(cmd), true, NULL, 0,
              rsp, M5310A_DEFAULT_RSP_LEN, NULL);
  if (strstr(rsp, "OK") == NULL) {
    LOGE(TAG, "%s %d failed rsp %s\r\n", __func__, __LINE__, rsp);
    return -1;
  }
```

更详细的例子请参考 application/example/example_legacy/at_app

API 详情

at_init

原型

```
int at_init(void);
```

接口说明

初始化AT组件。

参数说明

无

返回值说明

值	说明
0	成功
负值	失败

接口示例

```
/* 初始化，先于其他API前调用 */
at_init();
```

at_deinit

原型

```
int at_deinit(void);
```

接口说明

去初始化AT组件。

参数说明

无

返回值说明

值	说明
0	成功
负值	失败

接口示例

```
/* 去初始化 */
at_deinit();
```

at_add_dev

原型

```
int at_add_dev(at_config_t *config);
```

接口说明

根据配置添加AT设备。

参数说明

参数	数据类型	方向	说明
config	at_config_t *	输入	AT设备配置，at_config_t定义见标准宏和结构体定义

返回值说明

值	说明
非负值	成功，AT设备文件描述符
负值	失败

接口示例

```
/**
  * 配置AT设备参数
  */
 uart_dev.port          = 1;
 uart_dev.config.baud_rate   = 9600;
 uart_dev.config.data_width  = DATA_WIDTH_8BIT;
 uart_dev.config.parity     = NO_PARITY;
 uart_dev.config.stop_bits   = STOP_BITS_1;
 uart_dev.config.flow_control = FLOW_CONTROL_DISABLED;
 uart_dev.config.mode       = MODE_TX_RX;
 at_config.type                  = AT_DEV_UART;
 at_config.port                  = uart_dev.port;
 at_config.dev_cfg               = &uart_dev;
 at_config.send_wait_prompt          = 0;
 /* 添加AT设备，成功后返回设备fd */
 if ((at_dev_fd = at_add_dev(&at_config)) < 0) {
   LOGE(TAG, "AT parser device add failed!\n");
   return -1;
 }
```

at_delete_dev

原型

```
int at_delete_dev(int fd);
```

接口说明

根据设备描述符删除AT设备。

参数说明

参数	数据类型	方向	说明
fd	int	输入	AT设备文件描述符

返回值说明

值	说明
0	成功
负值	失败

接口示例

```
/* 添加AT设备，成功后返回设备fd */
if ((at_dev_fd = at_add_dev(&at_config)) < 0) {
    LOGE(TAG, "AT parser device add failed!\n");
    return -1;
}
/* 删除fd对应的AT设备 */
if ((at_delete_dev(at_dev_fd)) < 0) {
    LOGE(TAG, "AT parser device delete failed!\n");
    return -1;
}
```

at_send_wait_reply

原型

```
int at_send_wait_reply(int fd, const char *cmd, int cmdlen, bool delimiter,
                       const char *data, int datalen,
                       char *replybuf, int bufsize,
                       const at_reply_config_t *atcmdconfig);
```

接口说明

在用户task中，使用该接口发送数据，并等待回复。

参数说明

参数	数据类型	方向	说明
fd	int	输入	AT设备文件描述符
cmd	const char *	输入	准备发送的命令字符串
cmdlen	int	输入	命令字符串长度
delimiter	bool	输入	是否添加分割符号
const char *	const char *	输入	准备发送的数据字符串
datalen	int	输入	数据长度
replybuf	char *	输出	回复缓存指针，由调用者提供
bufsize	int	输入	回复缓存大小
atcmdconfig	const at_reply_config_t *	输入	回复前缀、成功后缀、失败后缀等配置

返回值说明

值	说明
0	成功
负值	失败

接口示例

```
/* 发送AT test命令 */
  memset(cmd, 0, sizeof(cmd));
  snprintf(cmd, sizeof(cmd) - 1, "%s", "AT");
  /* 发送并等待回复 */
  at_send_wait_reply(at_dev_fd, cmd, strlen(cmd), true, NULL, 0,
             rsp, M5310A_DEFAULT_RSP_LEN, NULL);
  /* 检查是否包含期望字符 */
  if (strstr(rsp, "OK") == NULL) {
    LOGE(TAG, "%s %d failed rsp %s\r\n", __func__, __LINE__, rsp);
    return -1;
  }
```

at_send_no_reply

原型

```
int at_send_no_reply(int fd, const char *data, int datalen, bool delimiter);
```

接口说明

使用该接口发送数据，但不等待回复。

参数说明

参数	数据类型	方向	说明
fd	int	输入	AT设备文件描述符
const char *	const char *	输入	准备发送的数据字符串
datalen	int	输入	数据长度

返回值说明

值	说明
0	成功
负值	失败

接口示例

```
/* 准备AT test命令 */
memset(cmd, 0, sizeof(cmd));
snprintf(cmd, sizeof(cmd) - 1, "%s", "AT");
/* 发送cmd，无等待 */
at_send_no_reply(at_dev_fd, cmd, strlen(cmd), true);
```

at_read

原型

```
int at_read(int fd, char *outbuf, int readsize);
```

接口说明

在回调函数里，从底层读取AT字符流。

参数说明

参数	数据类型	方向	说明
fd	int	输入	AT设备文件描述符
char *	char *	输出	缓存指针
readsize	int	输入	读取长度

返回值说明

值	说明
非负	成功，读取长度
负值	失败

接口示例

```
while (data != ',') {
  /* 从对应fd读取数据，一般只在回调函数里调用*/
  ret = at_read(at_dev_fd, &data, 1);
  if (ret != 1) {
    break;
  }
  /* 解析读出的数据*/
  if(data >= '0' && data <= '9') {
    fd = fd * 10 + (data - '0');
  }
 }
```

at_register_callback

原型

```
int at_register_callback(int fd, const char *prefix, const char *postfix, char *recvbuf,
                         int bufsize, at_recv_cb cb, void *arg);
```

接口说明

注册AT接收处理回调函数

参数说明

参数	数据类型	方向	说明
fd	int	输入	AT设备文件描述符
const char *	const char *	输入	感兴趣的前缀，必须
postfix	const char *	输入	感兴趣的后缀，可选

参数	数据类型	方向	说明
recvbuf	char *	输入	用于缓存前缀后缀之间的数据
bufsize	int	输入	缓存大小
cb	at_recv_cb	输入	回调处理函数
arg	void *	输入	回调处理函数参数

返回值说明

值	说明
0	成功
负值	失败

接口示例

```
/* 注册多个前缀对应的回调函数 */
at_register_callback(at_dev_fd, "+NSONMI:", NULL, NULL, 0,
                     m5310a_socket_data_indicator_handler, NULL);
at_register_callback(at_dev_fd, "\r\nCONNECT OK", NULL, NULL, 0,
                     m5310a_socket_connect_indicator_handler, NULL);
at_register_callback(at_dev_fd, "+NSOCLI:", NULL, NULL, 0,
                     m5310a_socket_close_indicator_handler, NULL);
```

at_yield

原型

```
int at_yield(int fd, char *replybuf, int bufsize, const at_reply_config_t *atcmdconfig,
        int timeout_ms);
```

接口说明

单task情况下，从底层接收数据。

参数说明

参数	数据类型	方向	说明
fd	int	输入	AT设备文件描述符
char *	char *	输出	接收缓存
bufsize	int	输入	缓存大小
atcmdconfig	const at_reply_config_t *	输入	接收前缀、成功后缀、失败后缀配置
timeout_ms	int	输入	超时时间

返回值说明

值	说明
0	成功
负值	失败

接口示例

```
/* 单task时, 调用该函数从底层缓存中收取数据并处理回调 */
at_yield(at_dev_fd, NULL, 0, NULL, 200);
```

配置说明

AT可配置项包括：

- 是否定义AOS_ATCMD，默认为定义AOS_ATCMD
- 最大AT设备数，默认是1
- AT发送后等待超时时间，默认为是5000ms
- AT读取超时时间，默认为1000ms
- AT分段发送时等待提示符超时时间，默认为200ms

移植说明

AT模块需要实现AT设备操作wrapper、系统wrapper。

AT设备操作wrapper

接口	描述
int (*init)(void *dev)	初始化设备 dev：设备参数 返回值：0成功，-1失败
int (*recv)(void *dev, void *data, uint32_t expect_size, uint32_t *recv_size, uint32_t timeout)	数据接收 dev：设备参数 data：数据地址 expect_size：期望长度 recv_size：接收长度 timeout：接收超时时长 返回值：0成功，-1失败

int (*send)(void *dev, void *data, uint32_t size, uint32_t timeout)	数据发送 dev：设备参数 data：数据地址 size：长度 timeout：超时时长 返回值：>0发送长度，-1失败
int (*deinit)(void *dev)	去初始化 dev：设备参数 返回值：0成功，-1失败

系统wrapper

锁函数

函数名	说明
void *atpsr_mutex_new(void)	创建锁
void atpsr_mutex_lock(void *mutex)	上锁
void atpsr_mutex_unlock(void *mutex)	开锁
void atpsr_mutex_free(void *mutex)	删除锁

信号量函数

函数名	说明
void *atpsr_sem_new(void)	创建信号量
void atpsr_sem_signal(void *sem)	释放信号量
int atpsr_sem_wait(void *sem, uint32_t timeout_ms)	等待信号量
void atpsr_sem_free(void *sem)	删除信号量

任务函数

函数名	说明
int atpsr_task_new_ext(void *task, char *name, void (*fn)(void *), void *arg, int stack_size, int prio)	创建任务
void atpsr_sleep_ms(const unsigned int millisec)	休眠任务

内存函数

函数名	说明
void *atpsr_malloc(uint32_t size)	分配内存
void atpsr_free(void *ptr)	释放内存

标准宏和结构体说明

结构体at_config_t定义

```
typedef struct {
  uint8_t       port;          // 端口号
  at_dev_type_t    type;          // AT设备类型，例如UART
  void          *dev_cfg;       // AT设备底层参数，例如hal/uart.h中的uart_config_t
  at_reply_config_t reply_cfg;       // AT接收前缀、后缀
  char          *send_delimiter;    // AT发送分割符
  uint32_t       timeout_ms;       // AT发送或接收超时
  uint8_t       send_wait_prompt;   // AT发送数据时是否等待提示符
  uint32_t       prompt_timeout_ms;  // AT发送数据时等待提示符超时时间
  uint8_t       send_data_no_wait;  // AT发送数据后是否等待回复
  int          recv_task_priority; // AT接收task优先级
  int          recv_task_stacksize; // AT接收task栈大小
} at_config_t;
```

结构体at_reply_config_t定义

```
typedef struct {
  char *reply_prefix;          // 回复前缀
  char *reply_success_postfix;     // 成功后缀
  char *reply_fail_postfix;       // 失败后缀
} at_reply_config_t;
```

结构体at_dev_ops_t定义

```
typedef struct {
  at_dev_type_t type;
  int (*init)(void *dev);
  int (*recv)(void *dev, void *data, uint32_t expect_size,
        uint32_t *recv_size, uint32_t timeout);
  int (*send)(void *dev, void *data, uint32_t size,
        uint32_t timeout);
  int (*deinit)(void *dev);
} at_dev_ops_t;
```

2.4.14. MAL组件文档

MAL，是MQTT Adapter Layer的简称。MAL组件将外部扩展模组中提供的MQTT协议，通过AT或其他命令的方式，为用户转换为AliOS Things系统中提供的统一MQTT协议API，提高用户应用程序的可移植性和硬件无关性。下图是AliOS Things中MAL套件的架构示意图。

其中，组件包括：

- **MAL Core**：由AliOS Things提供，MAL核心组件。主要包括MQTT连接管理、数据缓存、协议转换等功能，对上提供MQTT API接口服务，对下提供统一的HAL接口规范（可以对接到不同厂商的AT模组）。

- **MAL Driver**：驱动，部分由AliOS Things提供，其他由用户自己提供。MAL驱动模块基于具体型号的通信模组提供的AT命令，实现MAL规范的HAL接口功能。

API包括：

- **MQTT API**：这一层接口提供MQTT标准接口，如subscribe、unsubscribe、publish等。
- **MAL HAL API**：这一层接口定义MAL核心模块与不同厂商模组驱动之间的统一界面。这一层HAL的对接是模组驱动接入中的主要工作。

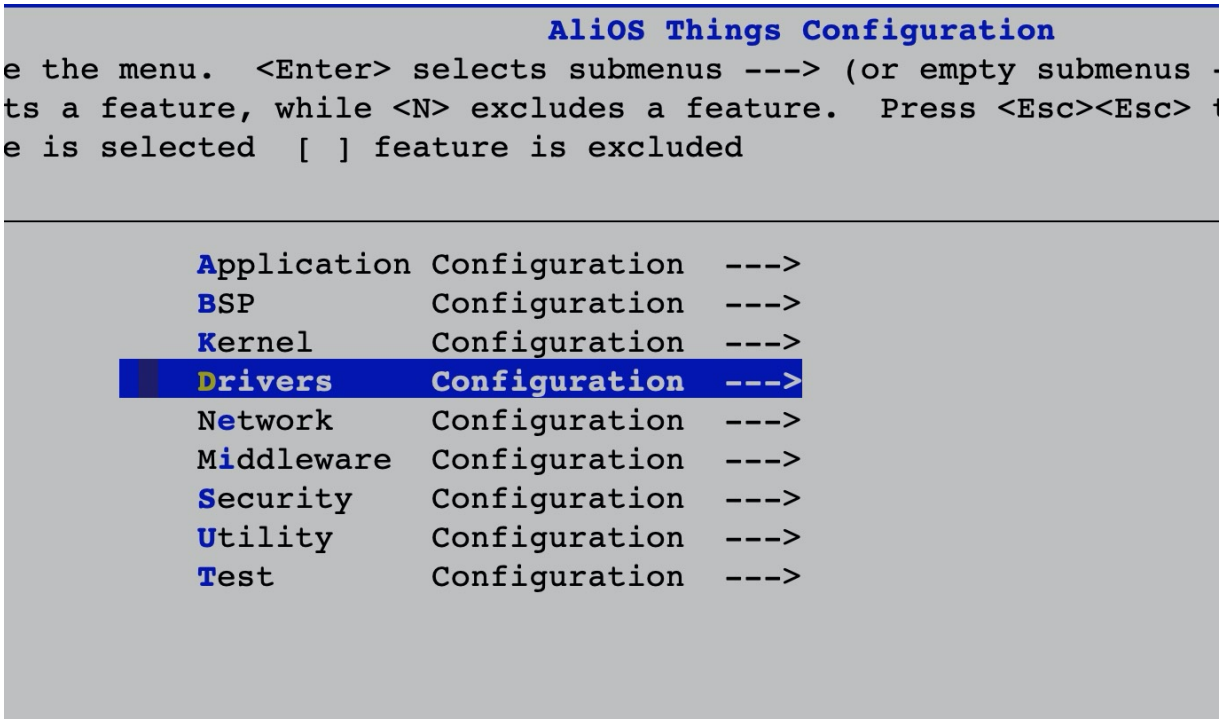
API列表

名称	说明
IOT_MQTT_Construct	云端建立MQTT连接
IOT_MQTT_Destroy	销毁指定MQTT连接并释放资源
IOT_MQTT_CheckStateNormal	获取当前MQTT连接状态
IOT_MQTT_Subscribe	向云端订阅指定的MQTT Topic
IOT_MQTT_Subscribe_Sync	向云端订阅指定的MQTT Topic，该接口为同步接口
IOT_MQTT_Unsubscribe	向云端取消订阅指定的topic
IOT_MQTT_Publish	向指定topic推送消息
IOT_MQTT_Publish_Simple	向指定topic推送消息，MQTT句柄可为NULL
mal_init	MAL模块初始化，包括初始化底层驱动模块。
mal_add_dev	配置驱动参数，例如串口通信串口号、波特率等。

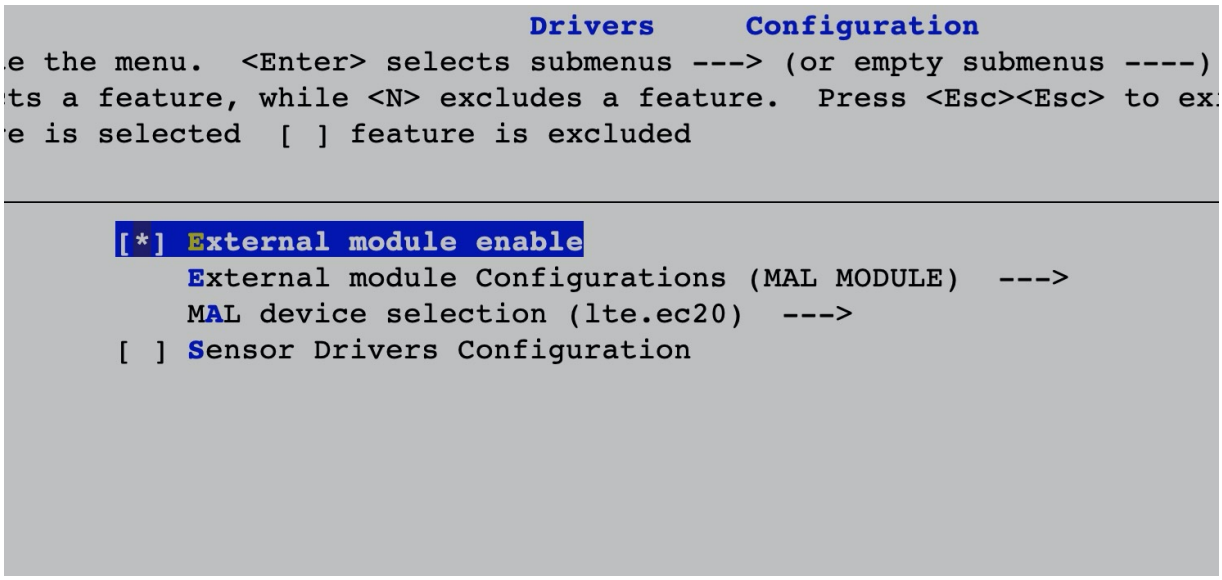
使用

添加该组件

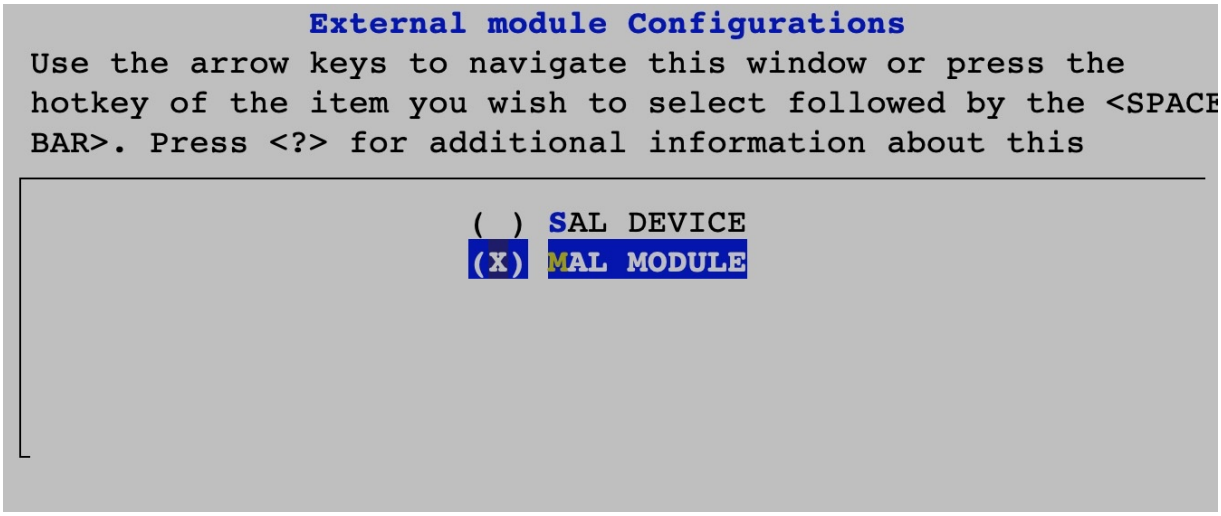
步骤1，aos make menuconfig，选择Drivers



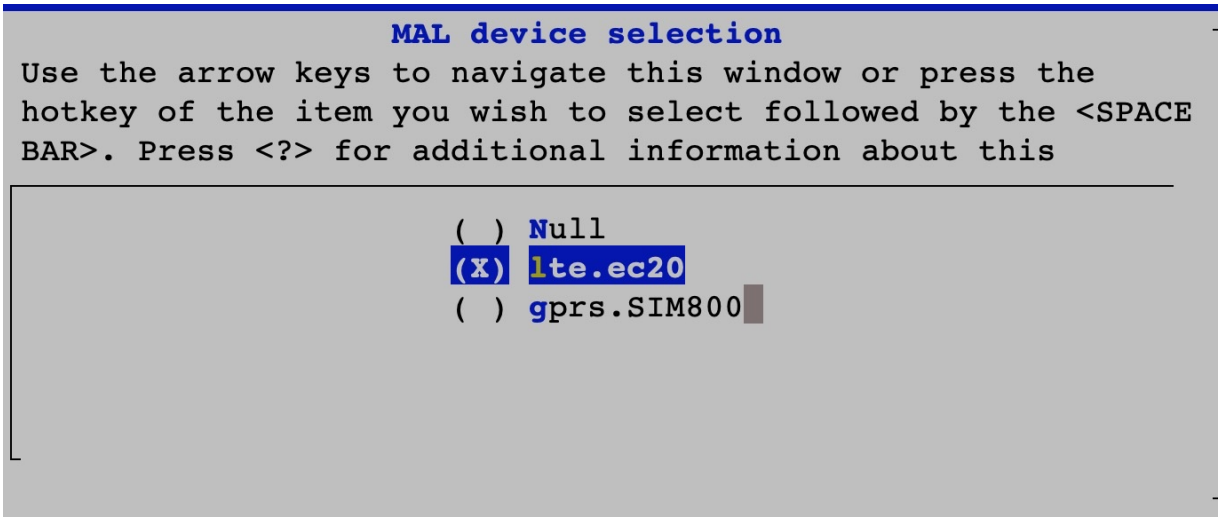
步骤2，选择External module enable



步骤3，选择Exernal module Configurations，并选择MAL MODULE



步骤4，选择MAL device selection，并选择相应设备对接。



头文件

对外头文件代码位于 `include/network/mal` ，包括

```
mal.h
hal_mal.h
```

使用时只需包含mal.h，hal_mal.h在对接时使用

```
#include "mal/mal.h"
```

使用示例

```
pclient = IOT_MQTT_Construct(&mqtt_params);
 if (NULL == pclient) {
    EXAMPLE_TRACE("MQTT construct failed");
    return -1;
 }
res = IOT_MQTT_Subscribe_Sync(pclient, topic, IOTX_MQTT_QOS0, example_message_arrive, NULL);
if (res < 0) {
   EXAMPLE_TRACE("subscribe failed");
   HAL_Free(topic);
   return -1;
}
 while (1) {
   int rc = IOT_MQTT_Publish(pclient, TOPIC_UPDATE, &topic_msg);
   if (rc < 0) {
     EXAMPLE_TRACE("IOT_MQTT_Publish fail, ret=%d", rc);
   }
   IOT_MQTT_Yield(pclient, 200);
}
```

更详细的例子请参考 application/example/example_legacy/mal_app/

API 详情

IOT_MQTT_Construct

原型

```
void *IOT_MQTT_Construct(iotx_mqtt_param_t *pInitParams)
```

接口说明

与云端建立MQTT连接, 入参 pInitParams 为 NULL 时将会使用默认参数建连。

参数说明

参数	数据类型	方向	说明
pInitParams	iotx_mqtt_param_t *	输入	MQTT初始化参数,填写NULL将以默认参数建连

返回值说明

值	说明
NULL	失败
非NULL	MQTT句柄

接口示例

```
iotx_mqtt_param_t     mqtt_params;
iotx_sign_mqtt_t      sign_mqtt;
/* 生成签名 */
gen_aliyun_mqtt_sign(&sign_mqtt);
/* 配置MQTT连接参数 */
mqtt_params.host = sign_mqtt.hostname;
mqtt_params.port = sign_mqtt.port;
mqtt_params.client_id = sign_mqtt.clientid;
mqtt_params.username = sign_mqtt.username;
mqtt_params.password = sign_mqtt.password;
mqtt_params.request_timeout_ms = 2000;
mqtt_params.clean_session = 0;
mqtt_params.keepalive_interval_ms = 60000;
mqtt_params.write_buf_size = 1024;
mqtt_params.read_buf_size = 1024;
mqtt_params.handle_event.h_fp = example_event_handle;
/* 与云端建立MQTT连接 */
pclient = IOT_MQTT_Construct(&mqtt_params);
if (NULL == pclient) {
  return -1;
}
```

IOT_MQTT_Destroy

原型

```
int IOT_MQTT_Destroy(void **phandle);
```

接口说明

销毁指定MQTT连接并释放资源

参数说明

参数	数据类型	方向	说明
phandle	void **	输入	MQTT句柄,可为NULL

返回值说明

值	说明
0	成功
< 0	失败

接口示例

```
/* 与云端建立MQTT连接 * /
pclient = IOT_MQTT_Construct(&mqtt_params);
/* 销毁指定MQTT连接并释放资源 * /
IOT_MQTT_Destroy(&pclient);
```

IOT_MQTT_CheckStateNormal

原型

```
int IOT_MQTT_CheckStateNormal(void *handle);
```

接口说明

获取当前MQTT连接状态

参数说明

参数	数据类型	方向	说明
handle	void *	输入	MQTT句柄,可为NULL

返回值说明

值	说明
0	未连接
1	已连接

接口示例

```
/* 获取当前MQTT连接状态 */
int state = IOT_MQTT_CheckStateNormal(pclient);
if (1 == state) {
  LOG("MQTT connected");
} else {
  LOG("MQTT disconnected");
}
```

IOT_MQTT_Yield

原型

```
int IOT_MQTT_Yield(void *handle, int timeout_ms);
```

接口说明

用于接收网络报文并将消息分发到用户的回调函数中

参数说明

参数	数据类型	方向	说明
handle	void *	输入	MQTT句柄,可为NULL
timeout_ms	int	输入	尝试接收报文的超时时间

返回值说明

值	说明
0	成功

接口示例

```
while (1) {
    /* 周期上报消息 */
   if (0 == loop_cnt % 20) {
     example_publish(pclient);
   }
   /* 接收网络报文并将消息分发到用户的回调函数 * /
   IOT_MQTT_Yield(pclient, 200);
   loop_cnt += 1;
}
```

IOT_MQTT_Subscribe

原型

```
int IOT_MQTT_Subscribe(void *handle,
                       const char *topic_filter,
                       iotx_mqtt_qos_t qos,
                       iotx_mqtt_event_handle_func_fpt topic_handle_func,
                       void *pcontext);
```

接口说明

向云端订阅指定的MQTT Topic

参数说明

参数	数据类型	方向	说明
handle	void *	输入	MQTT句柄,可为NULL
topic_filter	const char *	输入	需要订阅的topic
qos	iotx_mqtt_qos_t	输入	采用的QoS策略
topic_handle_func	iotx_mqtt_event_handle_func_fpt	输入	用于接收MQTT消息的回调函数
pcontext	void *	输入	用户Context, 会通过回调函数送回

返回值说明

值	说明
0	成功
< 0	失败

接口示例

```
const char *fmt = "/%s/%s/user/get";
char *topic = NULL;
int topic_len = 0;
topic_len = strlen(fmt) + strlen(DEMO_PRODUCT_KEY) + strlen(DEMO_DEVICE_NAME) + 1;
topic = HAL_Malloc(topic_len);
memset(topic, 0, topic_len);
/* 拼接主题 */
HAL_Snprintf(topic, topic_len, fmt, DEMO_PRODUCT_KEY, DEMO_DEVICE_NAME);
/* 订阅主题 */
res = IOT_MQTT_Subscribe_Sync(handle, topic, IOTX_MQTT_QOS0, example_message_arrive, NULL);
if (res < 0) {
  EXAMPLE_TRACE("subscribe failed");
  HAL_Free(topic);
  return -1;
}
HAL_Free(topic);
```

IOT_MQTT_Subscribe_Sync

原型

```
int IOT_MQTT_Subscribe_Sync(void *handle,
                            const char *topic_filter,
                            iotx_mqtt_qos_t qos,
                            iotx_mqtt_event_handle_func_fpt topic_handle_func,
                            void *pcontext,
                            int timeout_ms);
```

接口说明

向云端订阅指定的MQTT Topic, 该接口为同步接口

参数说明

参数	数据类型	方向	说明
handle	void *	输入	MQTT句柄,可为NULL
topic_filter	const char *	输入	需要订阅的topic
qos	iotx_mqtt_qos_t	输入	采用的QoS策略
topic_handle_func	iotx_mqtt_event_handle_func_fpt	输入	用于接收MQTT消息的回调函数
pcontext	void *	输入	用户Context, 会通过回调函数送回
timeout_ms	int	输入	该同步接口的超时时间

返回值说明

值	说明
0	成功

值	说明
< 0	失败

接口示例

```
const char *fmt = "/%s/%s/user/get";
char *topic = NULL;
int topic_len = 0;
topic_len = strlen(fmt) + strlen(DEMO_PRODUCT_KEY) + strlen(DEMO_DEVICE_NAME) + 1;
topic = HAL_Malloc(topic_len);
memset(topic, 0, topic_len);
/* 拼接主题 */
HAL_Snprintf(topic, topic_len, fmt, DEMO_PRODUCT_KEY, DEMO_DEVICE_NAME);
/* 同步订阅，5000毫秒超时 */
res = IOT_MQTT_Subscribe_Sync(handle, topic, IOTX_MQTT_QOS0, example_message_arrive, NULL,5000);
if (res < 0) {
  EXAMPLE_TRACE("subscribe failed");
  HAL_Free(topic);
  return -1;
}
HAL_Free(topic);
```

IOT_MQTT_Unsubscribe

原型

```
int IOT_MQTT_Unsubscribe(void *handle, const char *topic_filter);
```

接口说明

向云端取消订阅指定的topic

参数说明

参数	数据类型	方向	说明
handle	void *	输入	MQTT句柄,可为NULL
topic_filter	const char *	输入	需要取消订阅的topic

返回值说明

值	说明
0	成功
< 0	失败

接口示例

```
/* 订阅主题 */
IOT_MQTT_Subscribe(pclient, TOPIC_GET, IOTX_MQTT_QOS1, _demo_message_arrive, NULL);
/* 取消订阅 */
IOT_MQTT_Unsubscribe(pclient, TOPIC_GET);
```

IOT_MQTT_Publish

原型

```
int IOT_MQTT_Publish(void *handle, const char *topic_name, iotx_mqtt_topic_info_pt topic_msg);
```

接口说明

向指定topic推送消息

参数说明

参数	数据类型	方向	说明
handle	void *	输入	MQTT句柄,可为NULL
topic_name	const char *	输入	接收此推送消息的目标topic
topic_msg	iotx_mqtt_topic_info_pt	输入	需要推送的消息

返回值说明

值	说明
> 0	成功(消息是QoS1时, 返回值就是这个上报报文的MQTT消息ID, 对应协议里的 `messageId`)
0	成功(消息是QoS0时)
< 0	失败

接口示例

```
iotx_mqtt_topic_info_t topic_msg;
/* 组装消息 */
memset(&topic_msg, 0x0, sizeof(iotx_mqtt_topic_info_t));
topic_msg.qos = IOTX_MQTT_QOS1;
topic_msg.retain = 0;
topic_msg.dup = 0;
topic_msg.payload = (void *)ptopic_info->payload;
topic_msg.payload_len = ptopic_info->payload_len;
/* 向主题发布消息 */
int rc = IOT_MQTT_Publish(pclient, TOPIC_UPDATE, &topic_msg);
if (rc < 0) {
    EXAMPLE_TRACE("IOT_MQTT_Publish fail, ret=%d", rc);
}
```

IOT_MQTT_Publish_Simple

原型

```
int IOT_MQTT_Publish_Simple(void *handle, const char *topic_name, int qos, void *data, int len)
```

接口说明

向指定topic推送消息

参数说明

参数	数据类型	方向	说明
handle	void *	输入	MQTT句柄,可为NULL
topic_name	const char *	输入	接收此推送消息的目标topic
qos	int	输入	采用的QoS策略
data	void *	输入	需要发送的数据
len	int	输入	数据长度

返回值说明

值	说明
> 0	成功(消息是QoS1时，返回值就是这个上报报文的MQTT消息ID，对应协议里的 `messageId`)
0	成功(消息是QoS0时)
< 0	失败

接口示例

```
const char    *fmt = "/%s/%s/user/get";
char        *topic = NULL;
int         topic_len = 0;
char        *payload = "{\"message\":\"hello!\"}";
topic_len = strlen(fmt) + strlen(DEMO_PRODUCT_KEY) + strlen(DEMO_DEVICE_NAME) + 1;
topic = HAL_Malloc(topic_len);
/* 拼接topic字符串 */
HAL_Snprintf(topic, topic_len, fmt, DEMO_PRODUCT_KEY, DEMO_DEVICE_NAME);
/* 向指定topic推送消息 */
IOT_MQTT_Publish_Simple(0, topic, IOTX_MQTT_QOS0, payload, strlen(payload));
HAL_Free(topic);
```

mal_init

原型

```
int mal_init(void);
```

接口说明

MAL模块初始化，包括初始化底层驱动模块。

参数说明

无

返回值说明

值	说明
0	成功
负值	失败

接口示例

```
mal_device_config_t data = {0};
data.uart_dev.port = 1;
data.uart_dev.config.baud_rate = 115200;
data.uart_dev.config.data_width = DATA_WIDTH_8BIT;
data.uart_dev.config.parity = NO_PARITY;
data.uart_dev.config.stop_bits  = STOP_BITS_1;
data.uart_dev.config.flow_control = FLOW_CONTROL_DISABLED;
data.uart_dev.config.mode = MODE_TX_RX;
/* 配置驱动参数 */
if (mal_add_dev("sim800", &data) != 0) {
  LOG("Failed to add MAL device!");
  return -1;
}
/* 初始化MAL core */
mal_init();
```

mal_add_dev

原型

```
int mal_add_dev(char* driver_name, void* data);
```

接口说明

配置驱动参数，例如串口通信串口号、波特率等。

参数说明

参数	数据类型	方向	说明
driver_name	char *	输入	驱动名称
data	void*	输入	配置参数指针

返回值说明

值	说明
0	成功
负值	失败

接口示例

见mal_init示例

配置说明

MAL可配置项包括：

- 最大topic长度，默认128
- 最大消息长度，默认256
- 是否定义WITH_MAL宏，默认定义WITH_MAL宏
- 是否定义NO_TCPIP，默认定义NO_TCPIP宏

移植说明

MAL模块需要实现AT连接HAL、系统HAL。

AT连接HAL

MAL AT连接HAL函数定义见mal_op_t。

接口	描述
int (*add_dev)(void* data)	添加设备 data：设备参数 返回值：0成功，-1失败
int (*init)(iotx_mqtt_param_t *pInitParams)	初始化模组，使其处理准备连接状态 pInitParams：连接状态参数 返回值：0成功，-1失败
int (*connect)(char *proKey, char *devName, char *devSecret)	与远端建立MQTT连接 proKey：product key devName：Device Name devSecret：Device Secret 返回值：0成功，-1失败
int (*subscribe)(const char *topic, int qos, unsigned int *mqtt_packet_id, int *mqtt_status, int timeout_ms)	订阅主题 topic：主题字符串地址 qos：QoS等级 mqtt_packet_id：消息包ID mqtt_status：MQTT消息状态 timeout_ms：超时时间 返回值：0成功，-1失败

int (*unsubscribe)(const char *topic, unsigned int *mqtt_packet_id, int *mqtt_status)	取消订阅主题 topic：主题字符串地址 mqtt_packet_id：数据包ID mqtt_status：MQTT消息状态 返回值：0成功，-1失败
int (*publish)(const char *topic, int qos, const char *message, unsigned int msg_len)	发布消息 topic：主题字符串地址 qos：QoS等级 message：消息地址 msg_len：消息长度 返回值：0成功，-1失败
int (*conn_state)(void)	查询连接状态 返回值参考iotx_mc_state_t
int (*disconn)(void)	断开连接 返回值：0成功，-1失败
int (*deinit)(void)	去初始化 返回值：0成功，-1失败
int (*register_mqtt_data_input_cb)(mqtt_data_input_cb_t cb)	注册MQTT数据接收回调 cb：回调函数 返回值：0成功，-1失败

系统wrapper

锁函数

函数名	说明
void *HAL_MutexCreate(void)	创建锁
void HAL_MutexLock(void *mutex)	上锁
void HAL_MutexUnlock(void *mutex)	开锁
void HAL_MutexDestroy(void *mutex)	删除锁

信号量函数

函数名	说明
void *HAL_SemaphoreCreate(void)	创建信号量
void HAL_SemaphorePost(void *sem)	释放信号量
int HAL_SemaphoreWait(void *sem, uint32_t timeout_ms)	等待信号量
void HAL_SemaphoreDestroy(void *sem)	删除信号量

时间函数

函数名	说明
void HAL_SleepMs(uint32_t ms)	睡眠函数
uint64_t HAL_UptimeMs(void)	获取系统时间

内存函数

函数名	说明
void *HAL_Malloc(uint32_t size)	分配内存
void HAL_Free(void *ptr)	释放内存

打印函数

函数名	说明
nt HAL_Snprintf(char *str, const int len, const char *fmt, ...)	格式化打印函数

标准宏和结构体说明

结构体iotx_mqtt_param_t定义

```
typedef struct {
    uint16_t             port;
    const char           *host;
    const char           *client_id;
    const char           *username;
    const char           *password;
    const char           *pub_key;
    const char           *customize_info;
    uint8_t              clean_session;
    uint32_t             request_timeout_ms;
    uint32_t             keepalive_interval_ms;
    uint32_t             write_buf_size;
    uint32_t             read_buf_size;
    iotx_mqtt_event_handle_t    handle_event;
} iotx_mqtt_param_t, *iotx_mqtt_param_pt;
```

- `port` ：云端服务器端口

- host ：云端服务器地址
- client_id ：MQTT客户端ID
- username ：登录MQTT服务器用户名
- password ：登录MQTT服务器密码
- pub_key ：MQTT连接加密方式及密钥
- clean_session ：选择是否使用MQTT协议的clean session特性
- request_timeout_ms ：MQTT消息发送的超时时间
- keepalive_interval_ms ：MQTT心跳超时时间
- write_buf_size ：MQTT消息发送buffer最大长度
- read_buf_size ：MQTT消息接收buffer最大长度
- handle_event ：用户回调函数, 用与接收MQTT模块的事件信息
- customize_info ：用户自定义上报信息，是以逗号为分隔符kv字符串，如用户的厂商信息，模组信息自定义字符串为"pid=123456,mid=abcd";

pInitParams**结构体的成员配置为0或NULL时将使用内部默认参数**

结构体mal_op_t定义

```
typedef struct mal_op_s {
  struct mal_op_s * next;  //<! Next mal_op_t structure
  char *version; //<! Reserved for furture use.
  char *name; //<! Drvier name
  /**
   * Add mal device .
   *
   * @param[in]  data - device parameters
   *
   * @return  0 - success, -1 - failure
   */
  int (*add_dev)(void* data);
  /**
   * Module low level init so that it's ready to setup mqtt connection.
   *
   * @param[in]  pInitParams - connect parameters which are used to setup
   *                 the MQTT connection.
   * @return  0 - success, -1 - failure
   */
  int (*init)(iotx_mqtt_param_t *pInitParams);
  /**
   * Start a socket connection via module.
   *
   * @param[in]  proKey   - product key.
   * @param[in]  devName   - device name.
   * @param[in]  devSecret - device secret.
   *
   * @note: If the module does not accept the triple, ingore these parameters
   *
   * @return  0 - success, -1 - failure
   */
  int (*connect)(char *proKey, char *devName, char *devSecret);
  /**
   * Subscribe a topic via the MQTT connection
   *
   * @param[in]  topic       - MQTT topic.
```

```
 * @param[in] qos        - QoS used.
 * @param[out] mqtt_packet_id - MQTT packet ID.
 * @param[out] mqtt_status   - MQTT subscribe status.
 * @param[out] timeout_ms    - time out in milliseconds.
 *
 * @return  0 - success, -1 - failure
 */
int (*subscribe)(const char *topic, int qos, unsigned int *mqtt_packet_id, int *mqtt_status, int timeout_ms);
/**
 * Unsubscribe a topic via the MQTT connection
 *
 * @param[in]  topic       - MQTT topic.
 * @param[out] mqtt_packet_id - MQTT packet ID.
 * @param[out] mqtt_status   - MQTT unsubscribe status.
 *
 * @return  0 - success, -1 - failure
 */
int (*unsubscribe)(const char *topic, unsigned int *mqtt_packet_id, int *mqtt_status);
/**
 * Publish MQTT message to the topic via module.
 *
 * @param[in]  topic - MQTT topic
 * @param[in]  qos   - quality of service used
 * @param[in]  message - message to be published
 * @param[in]  msg_len - message length
 *
 * @return  0 - success, -1 - failure
 */
int (*publish)(const char *topic, int qos, const char *message, unsigned int msg_len);
/**
 * Query AT MQTT connection status
 *
 * @return  MQTT status value.
 *       Refer to definition of iotx_mc_state_t in mal.h.
 */
int (*conn_state)(void);
/**
 * Disconnect the MQTT connection via module.
 *
 * @return  0 - success, -1 - failure
 */
int (*disconn)(void);
/**
 * Destroy MAL or exit low level state if necessary.
 *
 * @return  0 - success, -1 - failure
 */
int (*deinit)(void);
/**
 * Register mqtt data input function
 * Input data from mqtt module.
 * This callback should be called when mqtt data is received from the module
 * @param[in]  topic    - topic of the received message.
 * @param[in]  topic_len - length of topic.
 * @param[in]  message   - received message.
 * @param[in]  meg_len   - length of received message.
 *
 * @return  0 - success, -1 - failure
```

```
    @return 0 - success, -1 - failure
    */
    int (*register_mqtt_data_input_cb)(mqtt_data_input_cb_t cb);
} mal_op_t;
```

枚举iotx_mc_state_t定义

```
/* State of MQTT client */
typedef enum {
    IOTX_MC_STATE_INVALID = 0,              /* MQTT in invalid state */
    IOTX_MC_STATE_INITIALIZED = 1,          /* MQTT in initializing state */
    IOTX_MC_STATE_CONNECTED = 2,            /* MQTT in connected state */
    IOTX_MC_STATE_DISCONNECTED = 3,         /* MQTT in disconnected state */
    IOTX_MC_STATE_DISCONNECTED_RECONNECTING = 4,  /* MQTT in reconnecting state */
    IOTX_MC_STATE_CONNECT_BLOCK = 5         /* MQTT in connecting state when using async protocol stack */
} iotx_mc_state_t;
```

2.4.15. Network

2.4.15.1. Netmgr

2.4.15.2. Bluetooth

2.4.15.3. LoRaWAN

2.4.16. HTTP组件文档

HTTP (HyperText Transfer Protocol)是一款用于传输超文本的应用层协议。AliOS Things的网络协议栈包含了HTTP组件，提供HTTP客户端标准能力。开发者可以通过组件API，在设备端快速实现与HTTP服务端的数据交互。

下图是AliOS Things中HTTP组件的结构示意图。

其包括：

- **HTTP Core**：HTTP核心模块，主要包括连接建立、请求创建、请求发送、回复接收、回复解析。 该模块实现HTTP主流程，对外提供标准的HTTP协议行为。
- **Method API**：请求方法接口模块，主要包括get、post、put、delete等基本请求方法接口。该模块封装了HTTP核心模块的发送请求、接收响应的过程，以简化用户的调用。

API列表

名称	说明
httpclient_prepare	分配HTTP请求头部缓存和响应缓存
httpclient_unprepare	释放HTTP请求头部缓存和响应缓存
httpclient_reset	重置HTTP请求头部缓存和响应缓存
httpclient_get	发起GET请求，并接收响应
httpclient_head	发起HEAD请求，并接收响应

httpclient_post	发起POST请求，并接收响应
httpclient_put	发起PUT请求，并接收响应
httpclient_delete	发起DELETE请求，并接收响应
httpclient_conn	发起HTTP连接
httpclient_send	发送HTTP请求
httpclient_recv	接收HTTP响应
httpclient_clse	关闭HTTP连接
httpclient_set_custom_header	设置HTTP用户定义头部
httpclient_get_response_code	获取HTTP响应码
httpclient_get_response_header_value	获取HTTP响应头部字段值
httpclient_formdata_addtext	添加文本form data
httpclient_formdata_addfile	添加文件form data

注：该列表为新API，deprecated API不在本文介绍范围内。

使用

添加该组件

在使用http的组件或应用对应的aos.mk添加

```
$(NAME)_COMPONENTS += http
```

在使用http的组件或应用对应的Config.in添加

```
select AOS_COMP_HTTP if !AOS_CREATE_PROJECT
select CONFIG_NEW_HTTP_API
```

注：选择CONFIG_NEW_HTTP_API使用新API。如果为兼容以前的代码，请使用deprecated API（见http/http.h）。

头文件

对外头文件代码位于 `include/network/http` ，包括

```
http/http.h
http/httpclient.h
```

使用时需包含

```
#include <httpclient.h>
```

如果为兼容以前的代码，使用deprecated API时需要包含：

```
#include <http.h>
```

使用示例

```
httpclient_t client = {0};
 httpclient_data_t client_data = {0};
 int ret;
 char * customer_header = "Accept: */*\r\n";
 memset(req_buf, 0, sizeof(req_buf));
 client_data.header_buf = req_buf;
 client_data.header_buf_len = sizeof(req_buf);
 memset(rsp_buf, 0, sizeof(rsp_buf));
 client_data.response_buf = rsp_buf;
 client_data.response_buf_len = sizeof(rsp_buf);
 httpclient_set_custom_header(&client, customer_header);
 ret = httpclient_get(&client, url, &client_data);
 if( ret >= 0 ) {
   LOGI(TAG, "Data received: %s", client_data.response_buf);
 }
```

更详细的例子请参考 application/example/example_legacy/httpclient_app

API 详情

httpclient_prepare

原型

```
HTTPC_RESULT httpclient_prepare(httpclient_data_t *client_data, int header_size, int resp_size);
```

接口说明

分配HTTP请求头部缓存和响应缓存。

参数说明

参数	数据类型	方向	说明
client_data	httpclient_data_t *	输入	用户数据结构体
header_size	int	输入	头部大小
resp_size	int	输入	回复大小

httpclient_data_t定义见标准宏和结构体说明。

返回值说明

值	说明
0	成功
非0	失败，失败原因见HTTPC_RESULT定义

接口示例

```
httpclient_data_t client_data = {0};
/* 分配空间，并将地址挂在client_data结构体 */
ret = httpclient_prepare(&client_data, REQ_BUF_SIZE, RSP_BUF_SIZE);
if (ret != 0) {
  LOGE(TAG, "httpclient prepare buffer failed");
}
```

httpclient_unprepare

原型

```
HTTPC_RESULT httpclient_unprepare(httpclient_data_t *client_data);
```

接口说明

释放HTTP请求头部缓存和响应缓存。

参数说明

参数	数据类型	方向	说明
client_data	httpclient_data_t *	输入	用户数据结构体

返回值说明

值	说明
0	成功
非0	失败，失败原因见HTTPC_RESULT定义

接口示例

```
httpclient_data_t client_data = {0};
/* 分配空间，并将地址挂在client_data结构体 */
ret = httpclient_prepare(&client_data, REQ_BUF_SIZE, RSP_BUF_SIZE);
if (ret != 0) {
  LOGE(TAG, "httpclient prepare buffer failed");
}
/* 释放client_data中的缓存空间 */
httpclient_unprepare(&client_data);
```

httpclient_reset

原型

```
void httpclient_reset(httpclient_data_t *client_data);
```

接口说明

重置HTTP请求头部缓存和响应缓存。

参数说明

参数	数据类型	方向	说明
client_data	httpclient_data_t *	输入	用户数据结构体

返回值说明

无

接口示例

```
httpclient_data_t client_data = {0};
/* 分配空间，并将地址挂在client_data结构体 */
ret = httpclient_prepare(&client_data, REQ_BUF_SIZE, RSP_BUF_SIZE);
if (ret != 0) {
  LOGE(TAG, "httpclient prepare buffer failed");
}
/* 重置client_data中的缓存空间 */
httpclient_reset(&client_data);
```

httpclient_get

原型

```
HTTPC_RESULT httpclient_get(httpclient_t *client, char *url, httpclient_data_t *client_data);
```

接口说明

根据连接参数，向URL发送GET请求，并等待回复。

参数说明

参数	数据类型	方向	说明
client	httpclient_t *	输入	HTTP client上下文，包含配置参数，如服务端口号、服务端证书等
url	char *	输入	URL地址
client_data	httpclient_data_t *	输出	回复数据，由调用者分配空间

返回值说明

值	说明
0	成功
非0	失败，失败原因见HTTPC_RESULT定义

接口示例

```
httpclient_t client = {0};
httpclient_data_t client_data = {0};
/* 分配空间，并将地址挂在client_data结构体 */
ret = httpclient_prepare(&client_data, REQ_BUF_SIZE, RSP_BUF_SIZE);
if (ret != 0) {
  LOGE(TAG, "httpclient prepare buffer failed");
}
/* 向URL发送GET请求，并等待回复 */
ret = httpclient_get(&client, url, &client_data);
if( ret >= 0 ) {
  LOGI(TAG, "Data received: %s", client_data.response_buf);
}
```

httpclient_post

原型

```
HTTPC_RESULT httpclient_post(httpclient_t *client, char *url, httpclient_data_t *client_data);
```

接口说明

根据连接参数，向URL发送POST请求，并等待回复

参数说明

参数	数据类型	方向	说明
client	httpclient_t *	输入	HTTP client上下文，包含配置参数，如服务端口号、服务端证书等
url	char *	输入	URL地址
client_data	httpclient_data_t *	输出	回复数据，由调用者分配空间

返回值说明

值	说明
0	成功
非0	失败，失败原因见HTTPC_RESULT定义

接口示例

```
httpclient_t client = {0};
httpclient_data_t client_data = {0};
/* 分配空间，并将地址挂在client_data结构体 */
ret = httpclient_prepare(&client_data, REQ_BUF_SIZE, RSP_BUF_SIZE);
if (ret != 0) {
  LOGE(TAG, "httpclient prepare buffer failed");
}
/* 向URL发送POST请求，并等待回复 */
ret = httpclient_post(&client, url, &client_data);
if( ret >= 0 ) {
  LOGI(TAG, "Data received: %s", client_data.response_buf);
}
```

httpclient_put

原型

```
HTTPC_RESULT httpclient_put(httpclient_t *client, char *url, httpclient_data_t *client_data);
```

接口说明

根据连接参数，向URL发送PUT请求，并等待回复

参数说明

参数	数据类型	方向	说明
client	httpclient_t *	输入	HTTP client上下文，包含配置参数，如服务端口号、服务端证书等
url	char *	输入	URL地址
client_data	httpclient_data_t *	输出	回复数据，由调用者分配空间

返回值说明

值	说明
0	成功
非0	失败，失败原因见HTTPC_RESULT定义

接口示例

```
httpclient_t client = {0};
httpclient_data_t client_data = {0};
/* 分配空间，并将地址挂在client_data结构体 */
ret = httpclient_prepare(&client_data, REQ_BUF_SIZE, RSP_BUF_SIZE);
if (ret != 0) {
  LOGE(TAG, "httpclient prepare buffer failed");
}
/* 向URL发送PUT请求，并等待回复 */
ret = httpclient_put(&client, url, &client_data);
if( ret >= 0 ) {
  LOGI(TAG, "Data received: %s", client_data.response_buf);
}
```

httpclient_delete

原型

```
HTTPC_RESULT httpclient_delete(httpclient_t *client, char *url, httpclient_data_t *client_data);
```

接口说明

根据连接参数，向URL发送delete请求，并等待回复

参数说明

参数	数据类型	方向	说明
client	httpclient_t *	输入	HTTP client上下文，包括配置参数，如服务端口号、服务端证书
url	char *	输入	URL地址
client_data	httpclient_data_t *	输出	回复数据，由调用者分配空间

返回值说明

值	说明
0	成功
非0	失败，失败原因见HTTPC_RESULT定义

接口示例

```
httpclient_t client = {0};
httpclient_data_t client_data = {0};
/* 分配空间，并将地址挂在client_data结构体 */
ret = httpclient_prepare(&client_data, REQ_BUF_SIZE, RSP_BUF_SIZE);
if (ret != 0) {
  LOGE(TAG, "httpclient prepare buffer failed");
}
/* 向URL发送DELETE请求，并等待回复 */
ret = httpclient_delete(&client, url, &client_data);
if( ret >= 0 ) {
  LOGI(TAG, "Data received: %s", client_data.response_buf);
}
```

httpclient_get_response_code

原型

```
int httpclient_get_response_code(httpclient_t *client);
```

接口说明

获取上一次请求服务端回复码

参数说明

参数	数据类型	方向	说明
client	httpclient_t *	输入	HTTP client上下文，包括配置参数，如服务端口号、服务端证书

返回值说明

值	说明
整型	服务端回复码

接口示例

```
/* 向URL发送GET请求，并等待回复 */
ret = httpclient_get(&client, url, &client_data);
if( ret >= 0 ) {
  LOGI(TAG, "Data received: %s", client_data.response_buf);
}
/* 获取回复码 */
code = httpclient_get_response_code(&client);
```

httpclient_get_response_header_value

原型

```
int httpclient_get_response_header_value(char *header_buf, char *name, int *val_pos, int *val_len);
```

接口说明

根据名称获取回复头部特定值

参数说明

参数	数据类型	方向	说明
header_buf	char *	输入	头部缓存
name	char *	输入	头部名称
val_pos	int *	输出	头部开始位置
val_len	int *	输出	头部长度

返回值说明

值	说明
0	成功
非0	错误

接口示例

```
/* 向URL发送GET请求，并等待回复 */
ret = httpclient_get(&client, url, &client_data);
if( ret >= 0 ) {
  LOGI(TAG, "Data received: %s", client_data.response_buf);
}
/* 获取回复头部Content-lenght字段值 */
 if(0 == httpclient_get_response_header_value(client_data.header_buf, "Content-Length", (int *)&val_pos, (int *)&
val_len)) {
  sscanf(client_data.header_buf + val_pos, "%d", &total_len);
 }
```

httpclient_set_custom_header

原型

```
void httpclient_set_custom_header(httpclient_t *client, char *header);
```

接口说明

设置HTTP请求报文定制头部

参数说明

参数	数据类型	方向	说明
client	httpclient_t *	输入	HTTP client上下文，包含配置参数，如服务端口号、服务端证书等
header	char *	输入	定制头部指针,必须为有\r\n结束符的字符串

返回值说明

无

接口示例

```
char * customer_header = "Accept: */*\r\n";
/* 设置请求头部 */
httpclient_set_custom_header(&client, customer_header);
/* 向URL发送GET请求，并等待回复 */
ret = httpclient_get(&client, url, &client_data);
if( ret >= 0 ) {
  LOGI(TAG, "Data received: %s", client_data.response_buf);
}
```

httpclient_formdata_addtext

原型

```
int httpclient_formdata_addtext(httpclient_data_t* client_data, char* content_disposition, char* content_type, char* name, char* data, int data_len);
```

接口说明

添加文本表单数据

参数说明

参数	数据类型	方向	说明
client_data	httpclient_data_t*	输入	待添加的HTTP数据结构体地址
content_disposition	char *	输入	待添加的内容地址
content_type	char *	输入	内容类型的地址
name	char *	输入	名称的地址
data	char *	输入	表单数据地址
data_len	int	输入	表单数据长度

返回值说明

值	说明
0	成功
-1	错误

接口示例

```
char * upload_text = "upload text example";
/* 添加文本表单 */
httpclient_formdata_addtext(&client_data, "form-data", "application/octet-stream","uploadText", upload_text, strlen(upload_text));
/* 上传表单数据 */
httpc_post(&client, url, &client_data);
```

httpclient_formdata_addfile

原型

```
int httpc_formdata_addfile(httpclient_data_t* client_data, char* content_disposition, char* name, char* content_type, char* file_path);
```

接口说明

添加文件表单数据

参数说明

参数	数据类型	方向	说明
client_data	httpclient_data_t*	输入	待添加的HTTP数据结构体地址
content_disposition	char *	输入	待添加的内容地址
name	char *	输入	名称的地址
content_type	char *	输入	内容类型的地址
file_path	char *	输入	文件路径

返回值说明

值	说明
0	成功
-1	错误

接口示例

```
/* 添加文件表单 */
httpclient_formdata_addfile(&client_data, "form-data", "uploadFile", "application/octet-stream", src_path);
/* 上传表单数据 */
httpc_post(&client, url, &client_data);
```

httpclient_send

原型

```
HTTPC_RESULT httpclient_send(httpclient_t *client, const char *url, int method, httpclient_data_t *client_data);
```

接口说明

向服务端发送请求

参数说明

参数	数据类型	方向	说明

参数	数据类型	方向	说明
client	httpclient_t *	输入	HTTP client上下文，包含配置参数，如服务端口号、服务端证书等
url	const char *	输入	请求地址
method	int	输入	请求方法
client_data	const char *	输入	用户数据结构体，包含发送头部缓存

返回值说明

值	说明
0	成功
非0	失败，失败原因见HTTPC_RESULT定义

接口示例

```
httpclient_t client = { 0 };
httpclient_data_t client_data = {0};
/* 准备client_data缓存空间 */
/* 建立HTTP连接 */
...
/* 向远端发送GET请求 */
ret = httpclient_send(&client, url, HTTP_GET, &client_data);
if(HTTP_SUCCESS != ret) {
    LOGE(TAG, "http send request failed");
    return -1;
}
```

httpclient_recv

原型

```
HTTPC_RESULT httpclient_recv(httpclient_t *client, httpclient_data_t *client_data);
```

接口说明

接收服务端回复

参数说明

参数	数据类型	方向	说明
client	httpclient_t *	输入	HTTP client上下文，包含配置参数，如服务端口号、服务端证书等
client_data	httpclient_data_t *	输出	用户数据结构体，包含接收回复缓存,该空间由用户分配

返回值说明

值	说明
0	成功
非0	失败，失败原因见HTTPC_RESULT定义

接口示例

```
while (total_len == 0 || recv_total_len < total_len) {
    /* 从远端接收数据 */
    ret = httpclient_recv(&client, &client_data);
    if (ret == HTTP_SUCCESS || ret == HTTP_EAGAIN) {
      recv_len = client_data.content_block_len;
      /* 处理缓存中的数据 */
    } else {
      recv_len = 0;
    }
    if (ret < 0) {
      break;
    }
}
```

httpclient_conn

原型

```
HTTPC_RESULT httpclient_conn(httpclient_t *client, const char *url);
```

接口说明

建立HTTP连接。

参数说明

参数	数据类型	方向	说明
client	httpclient_t *	输入	HTTP client上下文，包含配置参数，如服务端口号、服务端证书等
url	const char *	输入	HTTP/HTPS地址

返回值说明

值	说明
0	成功
非0	失败，失败原因见HTTPC_RESULT定义

接口示例

```
/* 根据URL与远端建立HTTP/HTTPS连接 */
ret = httpclient_conn(&client, url);
if(HTTP_SUCCESS != ret ) {
  LOGE(TAG, "http connect failed");
  return -1;
}
```

httpclient_clse

原型

```
void httpclient_clse(httpclient_t *client);
```

接口说明

关闭http连接

参数说明

参数	数据类型	方向	说明
client	httpclient_t *	输入	HTTP client上下文，包含配置参数，如服务端口号、服务端证书等

返回值说明

无

接口示例

```
/* 根据URL与远端建立HTTP/HTTPS连接 */
ret = httpclient_conn(&client, url);
/* 关闭HTTP连接 */
httpclient_clse(&client);
```

配置说明

HTTP可配置项包括：

- 是否支持HTTP安全通道，默认为否

移植说明

HTTP模块需要实现连接wrapper

接口	描述
int http_tcp_conn_wrapper(httpclient_t *client, const char *host)	建立TCP连接 client：HTTP上下文 host：远端服务地址 返回值：0成功，-1失败

int http_tcp_conn_wrapper(httpclient_t *client, const char *host)	关闭TCP连接 client：HTTP上下文 返回值：0成功，-1失败
int http_tcp_send_wrapper(httpclient_t *client, const char *data, int length)	向TCP远端发送数据 client：HTTP上下文 data：数据地址 length：长度 返回值：>0发送长度，-1失败
int http_tcp_recv_wrapper(httpclient_t *client, char *buf, int buflen, int timeout_ms, int *p_read_len)	接收TCP数据client：HTTP上下文 buf：接收缓存 buflen：缓存大小 timeout_ms：超时时间 p_read_len：接收长度 返回值：HTTPC_RESULT
int http_ssl_conn_wrapper(httpclient_t *client, const char *host)	建立SSL连接 client：HTTP上下文 host：远端服务地址 返回值：0成功，-1失败
int http_ssl_close_wrapper(httpclient_t *client)	关闭SSL连接 client：HTTP上下文 返回值：0成功，-1失败
int http_ssl_send_wrapper(httpclient_t *client, const char *data, size_t length)	向SSL远端发送数据 client：HTTP上下文 data：数据地址 length：长度 返回值：>0发送长度，-1失败

int http_ssl_recv_wrapper(httpclient_t *client, char *buf, int buflen, int timeout_ms, int *p_read_len);	接收SSL数据 client：HTTP上下文 buf：接收缓存 buflen：缓存大小 timeout_ms：超时时间 p_read_len：接收长度 返回值：HTTPC_RESULT

标准宏和结构体说明

HTTPC_RESULT枚举定义

```
/** @brief  http error code */
typedef enum {
  HTTP_EAGAIN  = 1, /**< 更多的数据  */
  HTTP_SUCCESS = 0, /**< 操作成功   */
  HTTP_ENOBUFS = -1, /**< 缓存错误   */
  HTTP_EARG    = -2, /**< 参数错误   */
  HTTP_ENOTSUPP = -3, /**< 不支持    */
  HTTP_EDNS    = -4, /**< DNS解析错误 */
  HTTP_ECONN   = -5, /**< 连接失败   */
  HTTP_ESEND   = -6, /**< 发送数据失败 */
  HTTP_ECLSD   = -7, /**< 连接关闭   */
  HTTP_ERECV   = -8, /**< 接收失败   */
  HTTP_EPARSE  = -9, /**< url解析是吧 */
  HTTP_EPROTO  = -10, /**< 协议错误   */
  HTTP_EUNKOWN = -11, /**< 未知错误   */
  HTTP_ETIMEOUT = -12, /**< 超时     */
} HTTPC_RESULT;
```

httpclient_t结构体定义

```
typedef struct {
  int socket;              /**< 套接字ID     */
  int remote_port;         /**< 远端port     */
  int response_code;       /**< 响应码       */
  char *header;            /**< HTTP请求头部  */
  char *auth_user;         /**< 用户名       */
  char *auth_password;     /**< 密码        */
  bool is_http;            /**< 是否是HTTP连接 */
#if CONFIG_HTTP_SECURE
  const char *server_cert;    /**< 服务端证书地址  */
  const char *client_cert;    /**< 客户端证书地址  */
  const char *client_pk;      /**< 客户端私钥     */
  int server_cert_len;        /**< 服务端证书长度  */
  int client_cert_len;        /**< 客户端证书长度  */
  int client_pk_len;          /**< 客户端私钥长度  */
  void *ssl;              /**< ssl上下文地址   */
#endif
} httpclient_t;
```

httpclient_data_t结构体定义

```
typedef struct {
  bool is_more;          /* 指示是否需要接收更多数据 */
  bool is_chunked;       /* 指示收到的数据是否分块(chunked) */
  int retrieve_len;      /* 需要获取的总长度 */
  int response_content_len;  /* 回复内容的长度*/
  int content_block_len;     /* 单块长度 */
  int post_buf_len;      /* post数据缓存大小 */
  int response_buf_len;      /* 回复主体缓存大小 */
  int header_buf_len;        /* 回复头部缓存大小 */
  char *post_content_type;   /* post数据类型 */
  char *post_buf;        /* post数据缓存地址 */
  char *response_buf;        /* 回复主体缓存地址*/
  char *header_buf;      /* 回复头部缓存地址 */
  bool  is_redirected;       /* 是否是重定向URL */
  char* redirect_url;        /* 重定向URL */
} httpclient_data_t;
```

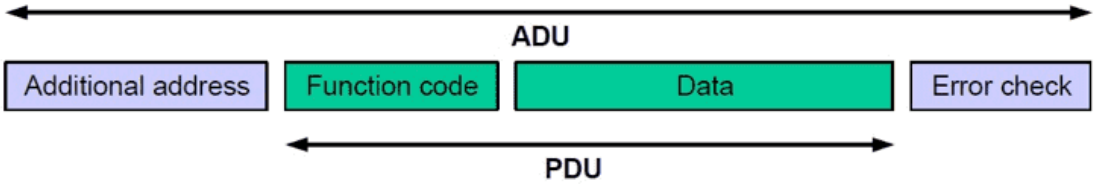
HTTP_REQUEST_TYPE枚举定义

```
typedef enum {
  HTTP_GET,
  HTTP_POST,
  HTTP_PUT,
  HTTP_DELETE,
  HTTP_HEAD
} HTTP_REQUEST_TYPE;
```

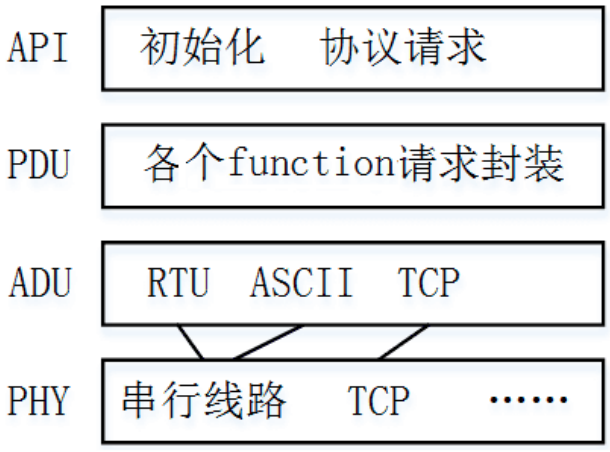
2.4.17. MODBUS主协议

MODBUS协议是一种主从式协议，即主站发起通讯，从站响应。主设备可以与一个或多个从设备通信。

当MODBUS主设备想要从一台从设备请求数据的时候，这个主设备会发送一条包含该从设备地址、请求数据以及一个用于检测错误的校验码的数据帧。网络上的所有其它设备都可以看到这一条信息，但是只有指定地址的设备才会做出反应。数据帧结构如下：



AliOS Things的MODBUS主协议是针对物联网RTOS系统特点而专门设计的。协议在实现时分为四层，整体架构如下：



API层主要用于提供用户接口。PDU层用于封装数据包PDU部分，ADU层用于封装数据包ADU部分，PHY用于通信链路初始化及数据发送与接收。

API列表

mbmaster_rtu_init()	初始化基于RTU的MODBUS主协议通信
mbmaster_rtu_uninit()	释放基于RTU的modbus主协议通信
mbmaster_read_coils()	读线圈，功能码0x01
mbmaster_read_discrete_inputs()	读离散输入，功能码0x02
mbmaster_read_holding_registers()	读保持寄存器，功能码0x03
mbmaster_read_input_registers()	读输入寄存器，功能码0x04
mbmaster_write_single_coil()	写单个线圈，功能码0x05
mbmaster_write_single_register()	写单个寄存器，功能码0x06
mbmaster_write_multiple_coils()	写多个线圈，功能码0x0F
mbmaster_write_multiple_registers()	写多个寄存器，功能码0x10

使用

添加该组件

在调用该组件的其它组件aos.mk中增加

```
$(NAME)_COMPONENTS += mbmaster
```

或者在应用层代码中包含头文件

```
#include "mbmaster.h"
```

包含头文件

```
#include "mbmaster.h"
```

使用示例

```
#include "modbus/mbmaster.h"
/* 定义串口通讯参数 */
#define SERIAL_PORT      2        /* 用于MODBUS通讯的uart端口号 */
#define SERIAL_BAUD_RATE   9600 /* 波特率 */
/* 定义从设备参数 */
#define DEVICE1_SLAVE_ADDR  0x1  /* 从设备地址 */
/* 定义请求参数 */
#define DEVICE1_REG1_ADDR  0x0   /* 待读寄存器地址 */
#define RECV_LEN_MAX        20    /* 数据缓存长度, 必须大于等于 (REQ_REGISTER_NUMBER * 2) */
#define REQ_REGISTER_NUMBER 2    /* 读寄存器的个数 */
void mb_main(void)
{
  uint8_t    buf[RECV_LEN_MAX];
  uint8_t    len;
  mb_status_t status;
  uint16_t   simulator1 = 0, simulator2 = 0;
  uint16_t   data_write = 0, data_resp = 0;
  uint16_t  *register_buf;
  /* 定义一个指针，调用mbmaster_rtu_init函数进行初始化 */
  mb_handler_t *mb_handler;
  /**
   * 初始化通信端口。用户需根据实际使用的串口参数进行配置
   */
  status = mbmaster_rtu_init(&mb_handler, SERIAL_PORT, SERIAL_BAUD_RATE, MB_PAR_NONE);
  if (status != MB_SUCCESS) {
    LOGE(MODBUSM_APP, "mbmaster init error");
    return;
  }
  /* 该用例每隔2秒循环一次，首先发起一个写请求，然后再发起一个读请求。*/
  while (1) {
    /**
     * 发起写寄存器请求。
     * data_resp是从设备返回的写结果，如果写成功，则该值等于写入值data_write。
     */
    status = mbmaster_write_single_register(mb_handler, DEVICE1_SLAVE_ADDR, DEVICE1_REG1_ADDR,
                            data_write, NULL, &data_resp, NULL, AOS_WAIT_FOREVER);
    if (status == MB_SUCCESS) {
      if (data_write != data_resp) {
        LOGE(MODBUSM_APP, "write single register error");
      } else {
        LOGI(MODBUSM_APP, "write single register ok");
      }
    } else {
      LOGE(MODBUSM_APP, "write single register error");
    }
    data_write++; /* 生成一个新的待写值 */
    aos_msleep(1000);
    memset(buf, 0, RECV_LEN_MAX);
    /**
     * 发起一个读寄存器请求。
     * 缓存长度必须大于等于 (REQ_REGISTER_NUMBER * 2)
     */
    status = mbmaster_read_holding_registers(mb_handler, DEVICE1_SLAVE_ADDR, DEVICE1_REG1_ADDR,
                             REQ_REGISTER_NUMBER, buf, &len, AOS_WAIT_FOREVER);
    if (status == MB_SUCCESS) {
```

```
        /*寄存器长度为16位 */
        register_buf = buf;
        simulator1 = register_buf[0];
        simulator2 = register_buf[1];
        LOGI(MODBUSM_APP, "read holding register simulator1: %d,simulator2: %d", simulator1, simulator2);
      } else {
        LOGE(MODBUSM_APP, "read holding register error");
      }
      aos_msleep(1000);
    }
}
```

API详情

modbus的应用层API说明请参考include/bus/modbus/mbmaster.h。

aos_mbmaster_rtu_init()

初始化基于RTU的MODBUS主协议通信。

函数原型

```
mb_status_t mbmaster_rtu_init(mb_handler_t **handler, uint8_t port, uint32_t baud_rate, mb_parity_t parity);
```

输入参数

handler	modbus句柄，用户定义一个mb_handler_t类型指针，将该指针的地址传入，调用成功后该指针指向新分配的句柄结构体	
port	用于通信的串口端口号	2
baud_rate	串口波特率	9600
parity	串行通信校验方式，MB_PAR_NONE不校验，MB_PAR_ODD奇校验，MB_PAR_EVEN偶校验	MB_PAR_NONE

返回参数

返回参数宏定义见include/bus/modbus/mbmaster.h文件

调用示例

```
mb_status_t status;
mb_handler_t *mb_handler;
status = mbmaster_rtu_init(&mb_handler, SERIAL_PORT, SERIAL_BAUD_RATE, MB_PAR_NONE);
if (status != MB_SUCCESS) {
  LOGE(MODBUSM_APP, "mbmaster init error");
  return;
}
```

aos_mbmaster_rtu_uninit()

释放基于RTU的modbus主协议通信。

函数原型

```
mb_status_t mbmaster_rtu_uninit(mb_handler_t *req_handler);
```

输入参数

req_handler	modbus句柄	

返回参数

返回参数宏定义见include/bus/modbus/mbmaster.h文件

调用示例

```
mb_status_t status;
mb_handler_t *mb_handler;
status = mbmaster_rtu_init(&mb_handler, SERIAL_PORT, SERIAL_BAUD_RATE, MB_PAR_NONE);
if (status != MB_SUCCESS) {
  LOGE(MODBUSM_APP, "mbmaster init error");
  return;
}
......
mbmaster_rtu_uninit(&mb_handler);
```

mbmaster_read_coils()

读线圈，功能码0x01。线圈是位码形式，每个线圈长度为1 bit。

函数原型

```
mb_status_t mbmaster_read_coils(mb_handler_t *req_handler, uint8_t slave_addr, uint16_t start_addr,
                                uint16_t quantity, uint8_t *respond_buf, uint8_t *respond_count,
                                uint32_t timeout);
```

输入参数

req_handler	modbus句柄	
slave_addr	从设备地址	0x01
start_addr	待读线圈起始地址	0x0
quantity	待读线圈数量	10
respond_buf	用于接收从设备响应数据的缓存，该缓存长度必须大于等于quantity/8字节（若不能整除需再增加1字节	
respond_count	从设备实际返回的数据长度，单位为字节）	
timeout	请求超时时间，单位ms	100

返回参数

返回参数宏定义见include/bus/modbus/mbmaster.h文件

调用示例

```
#define DEVICE1_SLAVE_ADDR  0x1
#define DEVICE1_COILS_ADDR  0x0
#define RECV_LEN_MAX      20
#define REQ_BIT_NUMBER     13
mb_status_t status;
mb_handler_t *mb_handler;
uint8_t    buf[RECV_LEN_MAX];
uint8_t    len;
status = mbmaster_rtu_init(&mb_handler, 2, 9600, MB_PAR_NONE);
if (status != MB_SUCCESS) {
  LOGE(MODBUSM_APP, "mbmaster init error");
  return;
}
status = mbmaster_read_coils(mb_handler, DEVICE1_SLAVE_ADDR, DEVICE1_COILS_ADDR,
                    REQ_BIT_NUMBER, buf, &len, AOS_WAIT_FOREVER);
```

mbmaster_read_discrete_inputs()

读离散输入，功能码0x02。离散输入是位码形式，每个离散输入长度为1 bit。

函数原型

```
mb_status_t mbmaster_read_discrete_inputs(mb_handler_t *req_handler, uint8_t slave_addr, uint16_t start_ad
dr,
                        uint16_t quantity, uint8_t *respond_buf, uint8_t *respond_count,
                        uint32_t timeout);
```

输入参数

req_handler	modbus句柄	
slave_addr	从设备地址	0x01
start_addr	待读离散输入的起始地址	0x0
quantity	待读离散输入的数量	10
respond_buf	用于接收从设备响应数据的缓存，该缓存长度必须大于等于quantity/8 字节（若不能整除需再增加1字节）	
respond_count	从设备实际返回的数据长度，单位为字节	
timeout	请求超时时间，单位ms	100

返回参数

返回参数宏定义见include/bus/modbus/mbmaster.h文件

调用示例

```
#define DEVICE1_SLAVE_ADDR  0x1
#define DEVICE1_COILS_ADDR  0x0
#define RECV_LEN_MAX      20
#define REQ_BIT_NUMBER     13
mb_status_t status;
mb_handler_t *mb_handler;
uint8_t    buf[RECV_LEN_MAX];
uint8_t    len;
status = mbmaster_rtu_init(&mb_handler, 2, 9600, MB_PAR_NONE);
if (status != MB_SUCCESS) {
  LOGE(MODBUSM_APP, "mbmaster init error");
  return;
}
status = mbmaster_read_discrete_inputs(mb_handler, DEVICE1_SLAVE_ADDR, DEVICE1_COILS_ADDR,
                          REQ_BIT_NUMBER, buf, &len, AOS_WAIT_FOREVER);
```

mbmaster_read_holding_registers()

读保持寄存器，功能码0x03。每个寄存器长度为16位。

函数原型

```
mb_status_t mbmaster_read_holding_registers(mb_handler_t *req_handler, uint8_t slave_addr,
                          uint16_t start_addr, uint16_t quantity,
                          uint8_t *respond_buf, uint8_t *respond_count, uint32_t timeout);
```

输入参数

req_handler	modbus句柄	
slave_addr	从设备地址	0x01
start_addr	待读寄存器的起始地址	0x0
quantity	待读寄存器的数量	2
respond_buf	用于接收从设备响应数据的缓存，该缓存长度必须大于等于quantity *2 字节	
respondcount	从设备实际返回的数据长度，单位为字节	
timeout	请求超时时间，单位ms	100

返回参数

返回参数宏定义见include/bus/modbus/mbmaster.h文件

调用示例

```
#define DEVICE1_SLAVE_ADDR  0x1
#define DEVICE1_REG1_ADDR   0x0
#define RECV_LEN_MAX      20
#define REQ_REGISTER_NUMBER 2
mb_status_t status;
mb_handler_t *mb_handler;
uint8_t    buf[RECV_LEN_MAX];
uint8_t    len;
status = mbmaster_rtu_init(&mb_handler, 2, 9600, MB_PAR_NONE);
if (status != MB_SUCCESS) {
  LOGE(MODBUSM_APP, "mbmaster init error");
  return;
}
status = mbmaster_read_holding_registers(mb_handler, DEVICE1_SLAVE_ADDR, DEVICE1_REG1_ADDR,
                                         REQ_REGISTER_NUMBER, buf, &len, AOS_WAIT_FOREVER);
```

mbmaster_read_input_registers()

读输入寄存器，功能码0x04。每个寄存器长度为16位。

函数原型

```
mb_status_t mbmaster_read_input_registers(mb_handler_t *req_handler, uint8_t slave_addr,
                                          uint16_t start_addr, uint16_t quantity,
                                          uint8_t *respond_buf, uint8_t *respond_count, uint32_t timeout);
```

输入参数

req_handler	modbus句柄	
slave_addr	从设备地址	0x01
start_addr	待读寄存器的起始地址	0x0
quantity	待读寄存器的数量	2
respond_buf	用于接收从设备响应数据的缓存，该缓存长度必须大于等于quantity *2 字节	
respondcount	从设备实际返回的数据长度，单位为字节	
timeout	请求超时时间，单位ms	100

返回参数

返回参数宏定义见include/bus/modbus/mbmaster.h文件

调用示例

```
#define DEVICE1_SLAVE_ADDR  0x1
#define DEVICE1_REG1_ADDR   0x0
#define RECV_LEN_MAX      20
#define REQ_REGISTER_NUMBER 2
mb_status_t status;
mb_handler_t *mb_handler;
uint8_t    buf[RECV_LEN_MAX];
uint8_t    len;
status = mbmaster_rtu_init(&mb_handler, 2, 9600, MB_PAR_NONE);
if (status != MB_SUCCESS) {
  LOGE(MODBUSM_APP, "mbmaster init error");
  return;
}
status = mbmaster_read_input_registers(mb_handler, DEVICE1_SLAVE_ADDR, DEVICE1_REG1_ADDR,
                                       REQ_REGISTER_NUMBER, buf, &len, AOS_WAIT_FOREVER);
```

mbmaster_write_single_coil()

写单个线圈，功能码0x05。线圈是位码形式，每个线圈长度为1 bit。

函数原型

```
mb_status_t mbmaster_write_single_coil(mb_handler_t *req_handler, uint8_t slave_addr, uint16_t coil_addr,
                                       uint16_t coil_value, uint16_t *resp_addr, uint16_t *resp_value,
                                       uint8_t *exception_code, uint32_t timeout);
```

输入参数

req_handler	modbus句柄	
slave_addr	从设备地址	0x01
coil_addr	待写线圈地址	0x0
coil_value	待写的值，0x0000表示off即写0，0xFF00表示ON即写1	0xFF00
resp_addr	从设备返回的实际写的地址，写成功时等于coil_addr	
resp_value	从设备返回的实际写的值，写成功时等于coil_value	
exception_code	异常码，当写异常时有效	
timeout	请求超时时间，单位ms	100

返回参数

返回参数宏定义见include/bus/modbus/mbmaster.h文件

调用示例

```
#define DEVICE1_SLAVE_ADDR  0x1
#define DEVICE1_COILS_ADDR  0x0
#define RECV_LEN_MAX      20
#define REQ_BIT_NUMBER     13
mb_status_t status;
mb_handler_t *mb_handler;
uint8_t    buf[RECV_LEN_MAX];
uint8_t    len;
uint16_t   data_resp = 0;
status = mbmaster_rtu_init(&mb_handler, 2, 9600, MB_PAR_NONE);
if (status != MB_SUCCESS) {
  LOGE(MODBUSM_APP, "mbmaster init error");
  return;
}
status = mbmaster_write_single_coil(mb_handler, DEVICE1_SLAVE_ADDR, DEVICE1_COILS_ADDR ,
                        0xFF00, NULL, &data_resp, NULL, AOS_WAIT_FOREVER);
```

mbmaster_write_single_register()

写单个寄存器，功能码0x06。寄存器长度16位。

函数原型

```
mb_status_t mbmaster_write_single_register(mb_handler_t *req_handler, uint8_t slave_addr, uint16_t register_
addr,
                        uint16_t register_value, uint16_t *resp_addr, uint16_t *resp_value,
                        uint8_t *exception_code, uint32_t timeout);
```

输入参数

req_handler	modbus句柄	
slave_addr	从设备地址	0x01
register_addr	待写寄存器地址	0x0
register_value	待写的值	0x03
resp_addr	从设备返回的实际写的地址，写成功时等于register_addr	
resp_value	从设备返回的实际写的值，写成功时等于register_value	
exception_code	异常码，当写异常时有效	
timeout	请求超时时间，单位ms	100

返回参数

返回参数宏定义见include/bus/modbus/mbmaster.h文件

调用示例

```
#define DEVICE1_SLAVE_ADDR  0x1
#define DEVICE1_REG1_ADDR   0x0
#define RECV_LEN_MAX      20
#define REQ_REGISTER_NUMBER 2
mb_status_t status;
mb_handler_t *mb_handler;
uint8_t    buf[RECV_LEN_MAX];
uint8_t    len;
uint16_t   data_resp = 0;
status = mbmaster_rtu_init(&mb_handler, 2, 9600, MB_PAR_NONE);
if (status != MB_SUCCESS) {
  LOGE(MODBUSM_APP, "mbmaster init error");
  return;
}
status = mbmaster_write_single_register(mb_handler, DEVICE1_SLAVE_ADDR, DEVICE1_REG1_ADDR,
                          0x03, NULL, &data_resp, NULL, AOS_WAIT_FOREVER);
```

mbmaster_write_multiple_coils()

写多个线圈，功能码0x0F。线圈是位码形式，每个线圈长度为1 bit。

函数原型

```
mb_status_t mbmaster_write_multiple_coils(mb_handler_t *req_handler, uint8_t slave_addr, uint16_t start_add
r,
                          uint16_t quantity, uint8_t *outputs_buf, uint16_t *resp_addr,
                          uint16_t *resp_quantity, uint8_t *exception_code, uint32_t timeout);
```

输入参数

req_handler	modbus句柄	
slave_addr	从设备地址	0x01
start_addr	待写线圈起始地址	0x0
quantity	写的数量	10
outputs_buf	输出缓存，里面保存了待写的值，位码形式1位对应1个线圈	
resp_addr	从设备返回的实际写的地址，写成功时等于start_addr	
resp_quantity	从设备返回的实际写的数量，写成功时等于quantity	
exception_code	异常码，当写异常时有效	
timeout	请求超时时间，单位ms	100

返回参数

返回参数宏定义见include/bus/modbus/mbmaster.h文件

调用示例

```
#define DEVICE1_SLAVE_ADDR  0x1
#define DEVICE1_COILS_ADDR  0x0
#define RECV_LEN_MAX      20
#define REQ_BIT_NUMBER     13
mb_status_t status;
mb_handler_t *mb_handler;
uint8_t    buf[RECV_LEN_MAX];
uint8_t    len;
status = mbmaster_rtu_init(&mb_handler, 2, 9600, MB_PAR_NONE);
if (status != MB_SUCCESS) {
  LOGE(MODBUSM_APP, "mbmaster init error");
  return;
}
buf[0] = 0x55;
buf[1] = 0x3c;
status = mbmaster_write_multiple_coils(mb_handler, DEVICE1_SLAVE_ADDR, DEVICE1_COILS_ADDR, REQ_BIT_NU
MBER,
                                       buf, NULL, NULL, NULL, AOS_WAIT_FOREVER);
```

mbmaster_write_multiple_registers()

写多个寄存器，功能码0x10。寄存器长度为16位。

函数原型

```
mb_status_t mbmaster_write_multiple_registers(mb_handler_t *req_handler, uint8_t slave_addr, uint16_t start_
addr,
                                              uint16_t quantity, uint8_t *outputs_buf, uint16_t *resp_addr,
                                              uint16_t *resp_quantity, uint8_t *exception_code, uint32_t timeout);
```

输入参数

req_handler	modbus句柄	
slave_addr	从设备地址	0x01
start_addr	待写寄存器的起始地址	0x0
quantity	写的数量	10
outputs_buf	输出缓存，里面保存了待写的值	
resp_addr	从设备返回的实际写的地址，写成功时等于start_addr	
resp_quantity	从设备返回的实际写的数量，写成功时等于quantity	
exception_code	异常码，当写异常时有效	
timeout	请求超时时间，单位ms	100

返回参数

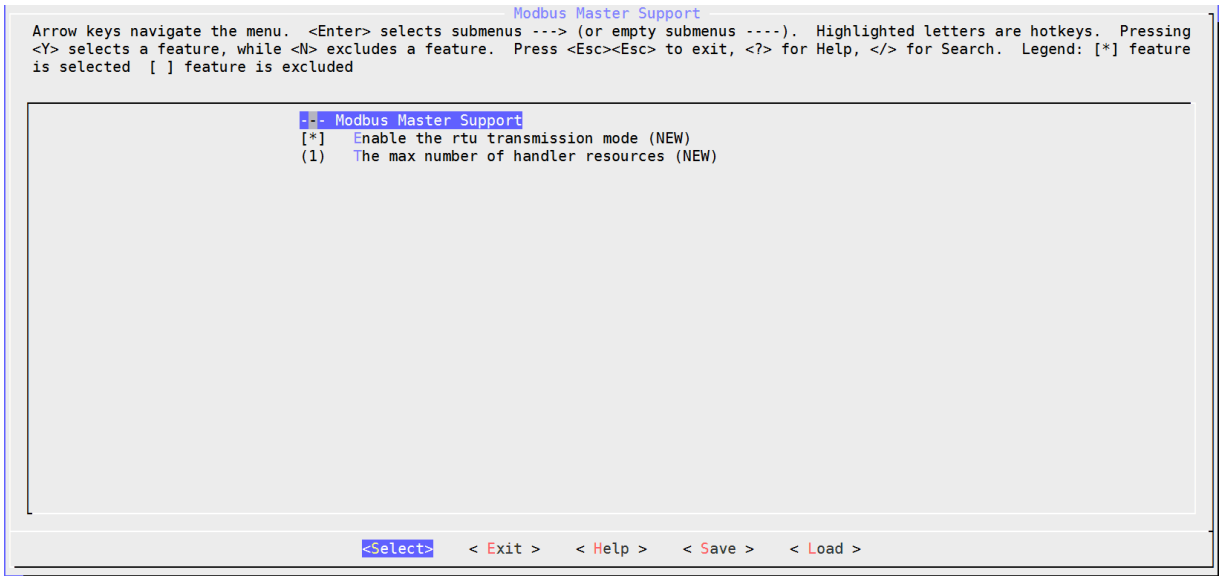
返回参数宏定义见include/bus/modbus/mbmaster.h文件

调用示例

```
#define DEVICE1_SLAVE_ADDR  0x1
#define DEVICE1_REG1_ADDR  0x0
#define RECV_LEN_MAX     20
#define REQ_REGISTER_NUMBER 2
mb_status_t status;
mb_handler_t *mb_handler;
uint8_t    register_buf[RECV_LEN_MAX];
uint8_t    len;
status = mbmaster_rtu_init(&mb_handler, 2, 9600, MB_PAR_NONE);
if (status != MB_SUCCESS) {
  LOGE(MODBUSM_APP, "mbmaster init error");
  return;
}
register_buf[0] = 0x03;
register_buf[1] = 0x04;
status = mbmaster_write_multiple_registers(mb_handler, DEVICE1_SLAVE_ADDR, DEVICE1_REG1_ADDR, REQ_REGI
STER_NUMBER,
        register_buf, NULL, NULL, NULL, AOS_WAIT_FOREVER);
```

配置说明

在AliOS Things 源码里面输入 `aos make menuconfig` 命令即可进入到menuconfig配置界面中，依次选择 `Middleware` -> `Modbus Master Support` 即可进入到MODBUS主协议组件的配置界面：



配置项说明

```
--- Key-value Storage
--- Modbus Master Support
[*] Enable the rtu transmission mode (NEW)      #使能RTU传输，默认使能
(1) The max number of handler resources (NEW) #modbus句柄结构体的数量，默认为1
```

移植说明

MODBUS主协议RTU方式基于串口HAL层接口，完成串口适配后即可使用。

其他

返回参数定义

返回参数定义在include/bus/modbus/mbmaster.h文件中。

```
typedef enum mb_status {
  MB_SUCCESS = 0u,
  MB_MUTEX_ERROR,        /* 请求互斥信号量失败，MODBUS协议栈内部通过互斥信号量保护临界资源 */
  MB_INVALID_SLAVE_ADDR, /* 无效的从设备地址 */
  MB_INVALID_PARAM,      /* 无效的参数 */
  MB_RESPOND_EXCEPTION,  /* 远程设备响应异常 */
  MB_RESPOND_LENGTH_ERR, /* 远程设备响应的数据帧长度错误 */
  MB_RESPOND_FRAME_ERR,  /* 远程设备响应的帧错误，比如校验和错误 */
  MB_RESPOND_TIMEOUT,    /* 请求超时 */
  MB_CANNOT_GET_HANDLER, /*分配句柄失败，可能是配置的资源不足引起的 */
  MB_SLAVE_NO_RESPOND,   /* 从设备无响应 */
  MB_FRAME_SEND_ERR,     /* 帧发送错误 */
  MB_SERIAL_INIT_FAILED,    /* 串行通信链路初始化失败 */
  MB_SERIAL_UNINIT_FAILED,  /* 串行通信链路逆初始化失败 */
  MB_FUNCTION_CODE_NOT_SUPPORT, /* 不支持的功能码 */
} mb_status_t;
```

使用注意事项

在调用 mbmaster_read_coils()、mbmaster_read_discrete_inputs()、mbmaster_read_holding_registers()、mbmaster_read_input_registers 函数时，需确保传入的用户缓存足够大，若为读寄存器类型则至少为2*quantity字节，若为读位码类型，则至少为quantity/8字节（若无法整除还需再增加1）。

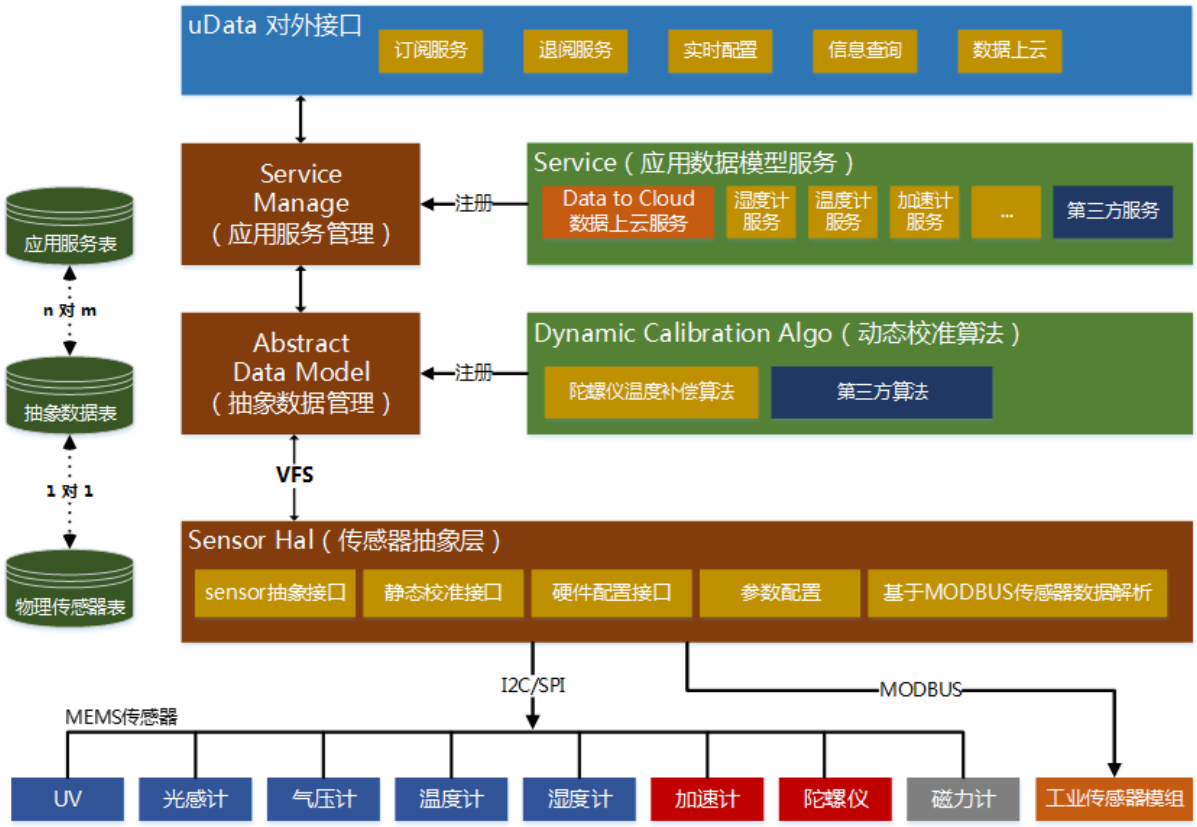
2.4.18. uData

uData 框架设计之初的思想是基于传统sensorhub概念基础之上的，结合IoT的业务场景和AliOS Things物联网操作系统的特点设计而成的一个面对IoT的感知设备处理框架。uData主要分kernel和framework两层，kernel层主要是负责传感器驱动和相关的静态校准，包括轴向校准等；framework层主要是负责应用服务管理、动态校准管理和对外模块接口等。

功能特性

- 数据上云
- 用户数据订阅
- 用户数据退订

udata架构图：



模块依赖

- device.sensor
- cjson (optional)
- uData为传感器管理架构，依赖于底层sensor模块；而uData的service初始化参数可采用数组或json格式来配置，后者依赖cjson模块。

使用指南

developerkit板上已集成了多款传感器，建议开发者采用该单板进行体验和调试。

包含头文件

```
#include "sensor/sensor.h"
#include "udata/udata.h"
```

API列表

udata_init	uData模块初始化(只需在APP中初始化一次即可)
udata_subscribe	uData服务订阅
udata_unsubscribe	uData服务退订
udata_report_publish	订阅的uData服务数据获取

API详情

int udata_init(void)

返回参数

0	执行成功
其他	执行错误

调用示例

```
int ret = 0;
ret = udata_init();
```

int udata_subscribe(udata_type_e type)

输入参数

udata_type_e type	订阅的服务类型

返回参数

0	执行成功
其他	执行错误

调用示例

```
/* 订阅温度服务 */
int ret = 0;
ret = udata_subscribe(UDATA_SERVICE_TEMP);
```

int udata_unsubscribe(udata_type_e type)

输入参数

udata_type_e type	退订的服务类型

返回参数

0	执行成功
其他	执行错误

调用示例

```
/* 退订温度服务 */
int ret = 0;
ret = udata_unsubscribe(UDATA_SERVICE_TEMP);
```

int udata_report_publish(udata_type_e type, void *pdata)

输入参数

udata_type_e type	需要读取的服务类型
void *pdata	服务类型数据需要存放的地址

返回参数

0	执行成功
其他	执行错误

调用示例

```
temperature_data_t temp;
int ret = 0;
ret = int udata_report_publish(UDATA_SERVICE_TEMP, &temp);
```

配置文件

uData支持json格式的参数配置，可参考以下配置(udata_config.json):

```
"udata_config_desc": [
  {
    "service_config": [
      {
        "type": "UDATA_SERVICE_ACC",   # service类型
        "task": "shared",          # service任务配置, 简单算法可采用共享任务--'shared'；复杂算法可独立创建任务--'exclu
sive';
        "devices": [              # service依赖的物理传感器
          {
            "tag": "TAG_DEV_ACC",   # 传感器tag
            "index": 0,          # 传感器索引
            "interval": 1000      # 传感器数据采样周期, 单位：ms
          }
        ]
      }
    ],
    "data_to_cloud": [
      {
        "dtc_enable": "off",        # 数据上云默认关闭
        "data_type": "UDATA_FLOAT",    # 上传的数据类型，UDATA_UINT32, UDATA_INT32, UDATA_FLOAT三种类型
        "coefficient": 1000,         # 上传数据缩小的比例系数
        "dtc_interval": 4000,        # 传感器数据周期, 单位：ms，建议与数据采样周期成比例关系
        "name": "Accelerometer",      # 上传数据名称；对于结构体类型数据，该处为结构体名称
        "sub_para": [
          {
            "name": "x"          # 对于结构体类型数据，该处为数据元素子名称
          },
          {
            "name": "y"
          },
          {
            "name": "z"
          }
        ]
      }
    ]
  }
]
```

或使用C语言格式的配置(udata_config.c):

```
/*sensor tag -- sensor index -- sampling period（ms） */
udata_tag_para_st g_dev_acc[] = {
{TAG_DEV_ACC, 0, 1000},
};
/*service type -- 是否独立任务运行 -- 依赖的sensor个数 -- sensor配置数组 */
udata_service_para_t g_service_para[] = {
{UDATA_SERVICE_ACC, false, UDATA_ARRAY_NUM_CALC(g_dev_acc),&g_dev_acc[0]},
};
```

配置方式选择如下所示:

上云配置

对于想通过该单板连接阿里云的用户，可在link develop平台上创建设备,设备属性如下所示：

自定义功能 添加功能

功能类型	功能名称	标识符	数据类型	数据定义	操作
属性	上云	dtc_config	text	数据长度：32	编辑 删除
属性	压力sensor	Barometer	int32	取值范围：0 ~ 110000	编辑 删除
属性	温度	CurrentTemperature	float	取值范围：-100 ~ 100	编辑 删除
属性	相对湿度	CurrentHumidity	float	取值范围：0 ~ 100	编辑 删除
属性	光照强度	LightLux	int32	取值范围：0 ~ 100000	编辑 删除
属性	接近光	Proximity	int32	取值范围：0 ~ 100	编辑 删除
属性	加速度计	Accelerometer	struct		编辑 删除
属性	磁力计	Magnetometer	struct		编辑 删除
属性	陀螺仪	Gyroscope	struct		编辑 删除

属性 加速度传感器 Accelerometer 复合型 - 编辑 删除

参数名称	参数标识	数据类型	参数属性
x轴加速度	x	浮点型(单精度)	取值范围：-16 ~ 16 单位：grav / 重力加速度
y轴加速度	y	浮点型(单精度)	取值范围：-16 ~ 16 单位：grav / 重力加速度
z轴加速度	z	浮点型(单精度)	取值范围：-16 ~ 16 单位：grav / 重力加速度

属性 磁场强度传感器 Magnetometer 复合型 - 编辑 删除

参数名称	参数标识	数据类型	参数属性
x轴磁场强度	x_gs	浮点型(单精度)	取值范围：-1000.0 ~ 1000.0 单位：/
y轴磁场强度	y_gs	浮点型(单精度)	取值范围：-1000.0 ~ 1000.0 单位：/
z轴磁场强度	z_gs	浮点型(单精度)	取值范围：-1000.0 ~ 1000.0 单位：/

属性 陀螺仪传感器 Gyroscope 复合型 - 编辑 删除

参数名称	参数标识	数据类型	参数属性
x轴角加速度	x_dps	浮点型(单精度)	取值范围：-2000.0 ~ 2000.0 单位：/
y轴角加速度	y_dps	浮点型(单精度)	取值范围：-2000.0 ~ 2000.0 单位：/
z轴角加速度	z_dps	浮点型(单精度)	取值范围：-2000 ~ 2000 单位：/

在文件 `app\example\udataapp\mqtt\mqtt_example.c` 中修改上述设备的三元组。

编译运行

本地调试

通过以下命令编译工程，可进行本地调试：

```
aos create project -b developerit -t udataapp udataapp
cd udataapp
aos make
```

运行成功后，可在本地串口看到以下打印输出：

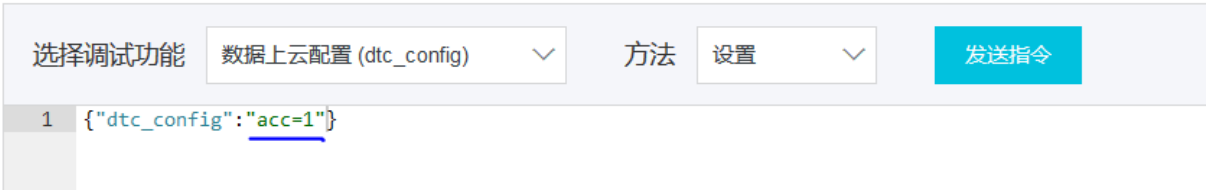
```
uData_application:::::::::timestamp = (250370)
[252370]<V> enter udata_std_service_process  6

uData_application:::::::::::::::type = (6)
uData_application:::::::::::::::data = (293)
uData_application:::::::::timestamp = (252370)
uData_application:::::::::::::::type = (8)
uData_application:::::::::::::::data = (401)
uData_application:::::::::timestamp = (252370)
[254370]<V> enter udata_std_service_process  6

uData_application:::::::::::::::type = (6)
uData_application:::::::::::::::data = (293)
uData_application:::::::::timestamp = (254370)
uData_application:::::::::::::::type = (8)
uData_application:::::::::::::::data = (395)
uData_application:::::::::timestamp = (254370)
[256370]<V> enter udata_std_service_process  6
```

数据上云

通过以下命令编译工程，可进行本地调试：

```
aos make dtc=mqtt
```

配网命令： `netmgr connect ssid passwd` 通过以下命令可以开启加速度传感器上云功能：




```
加速度：acc=1
磁力计：mag=1
陀螺仪：gyro=1
环境光：als=1
接近光：ps=1
温度计：temp=1
湿度计：humi=1
气压计：baro=1
```

运行成功后，可在云端可以看到以下数据上传：

2.4.19. uLog

软件产品的开发，测试和长期运维，离不开日志系统。uLog在AliOS Things中承担各个组件的日志记录和输出，主要特性如下： A. 遵循至简模式（默认）和syslog协议，日志记录包含下列要素：

- 日志生成时间
- 日志优先级
- 日志产生源（子系统标识）
- 日志Tag（模块名及在模块中的行号）
- 日志内容

B. 日志的优先级可以按需选择静态编译包含或者动态控制（支持API CLI，服务端运维通道（3.1版本后）），方便在调试，测试，以及发布的产品现场Trouble-Shooting 时push出较多日志（低优先级日志准出），产品release时push较少日志避免影响系统功能（高优先级日志准出）

C. 日志抛出方式多样式，可以进行动态切换通道，也可以通道组合，每个通道的日志优先级独立可控：

- 默认输出（嵌入式产品一般默认定向到串口）
- syslog udp发送至监听设备
- 具备文件回滚方式记录至文件系统
- 经由云端通道上行至阿里云物联网运维服务控制台（AliOS Things3.1后）

D. 异步日志功能 uLog可选异步日志功能，则负责抛出日志控制的逻辑以低于其他任务（仅比idle任务高1个级别）在后台运行，避免影响应用任务运行。

API列表

ulog()	记录日志通用方法
LOG()	以Emergency级别记录日志的简要接口
LOGF()	以Fatal级别记录日志的简要接口

LOGE()	以Error级别记录日志的简要接口
LOGW()	以Warning级别记录日志的简要接口
LOGI()	以Info级别记录日志的简要接口
LOGD()	以Debug级别记录日志的简要接口
aos_set_log_level()	获取某个加密的键的值
ulog_man()	获取某个加密的键的值
aos_get_ulog_list()	获取某个加密的键的值

使用

添加该组件

ulog是AliOS Things 默认添加的组件，开发者无需再手动添加

包含头文件

```
#include "ulog/ulog.h"
```

使用示例

```
LOGI("APP", "You are No. %d developer to use AliOS Things", user_number);
```

API详情

uLog的应用层API说明请参考include/dm/ulog/ulog.h。

ulog()

通用的记录日志方式，一般被下述简要日志记录API替代

函数原型

```
int ulog(const unsigned char s, const char *mod, const char *f, const unsigned long l, const char *fmt, ...)
```

输入参数

const unsigned char s	记录日志级别	LOG_ERR, 其它选值参见见include/ulog/ulog.h文件
const char *mod	记录日志产生时的模块名	"app"
const char *f	记录日志产生文件名，或者用户自己选择其他字符	__FILE__
const unsigned long l	记录日志产生在文件名的行数，或者用户自己选择使用其他32位数据	__LINE__

const char *fmt	可变参数格式化，结尾不用加回车换行，抛出日志时会自动添加	"Day %d"
...	可变参数	54

返回参数

-1	一般入参出错
其它>0的值	调用成功

调用示例

```
ulog(LOG_INFO, "APP", __FILE__, __LINE__, "Helloworld");
```

LOG()

简要的以Emergency级别记录日志方式，由于mod固定，一般用的地方较少

函数原型

```
LOG(...)
```

输入参数

...	可变参数，类似于printf，结尾不用加回车换行，抛出日志时会自动添加	"task exit"

返回参数

和ulog()返回参数一致

调用示例

```
LOG("Tom is %d years old", tom_age);
```

LOGF()

简要的以Fatal级别记录日志方式，一般用在严重错误，影响到产品正常且不可恢复的场景

函数原型

```
LOGF(mod, ...)
```

输入参数

const char *mod	记录日志产生时的模块名	"app"
...	可变参数，类似于printf，结尾不用加回车换行，抛出日志时会自动添加	"task exit"

返回参数

和ulog()返回参数一致

调用示例

```
LOGF("APP", "App Task Exit as unknown reason");
```

LOGE()

简要的以Error级别记录日志方式，一般用在产生一般错误，影响产品部分功能的场景

函数原型

```
LOGE(mod, ...)
```

输入参数

const char *mod	记录日志产生时的模块名	"app"
...	可变参数，类似于printf，结尾不用加回车换行，抛出日志时会自动添加	"task exit"

返回参数

和ulog()返回参数一致

调用示例

```
LOGE("BLE", "Network Disconnect");
```

LOGW()

简要的以Warning级别记录日志方式，一般用在产生可能会带来影响产品功能的场景

函数原型

```
LOGW(mod, ...)
```

输入参数

const char *mod	记录日志产生时的模块名	"app"
...	可变参数，类似于printf，结尾不用加回车换行，抛出日志时会自动添加	"task exit"

返回参数

和ulog()返回参数一致

调用示例

```
LOGW("APP", "Unknown Telegram 0x%x recvd", telegram_type);
```

LOGI()

简要的以Info级别记录日志方式，一般用在普通日志的场景

函数原型

```
LOGI(mod, ...)
```

输入参数

const char *mod	记录日志产生时的模块名	"app"
...	可变参数，类似于printf，结尾不用加回车换行，抛出日志时会自动添加	"task exit"

返回参数

和ulog()返回参数一致

调用示例

```
LOGI("APP", "Recv Operation command from host");
```

LOGD()

简要的以Debug级别记录日志方式，一般用在记录仅在研发时需要的日志。如果编译系统不含'DEBUG'开关编译，则此类日志不会被编译出固件

函数原型

```
LOGD(mod, ...)
```

输入参数

const char *mod	记录日志产生时的模块名	"app"
...	可变参数，类似于printf，结尾不用加回车换行，抛出日志时会自动添加	"task exit"

返回参数

和ulog()返回参数一致

调用示例

```
LOGD("APP", "Driver Mount Sucessfully");
```

aos_set_log_level()

修改默认定向日志输出级别

函数原型

```
int aos_set_log_level(aos_log_level_t log_level)
```

输入参数

aos_log_level_t log_level	准出日志级别	LOG_ERR

返回参数

0	设置成功
-22	由于入参不合法造成设置失败

调用示例

```
aos_set_log_level(LOG_ERR);
```

ulog_man()

一套简易的服务机制进行ulog功能控制，完成诸如使能通过udp 发送至remote syslog 监听工具，修改监听工具IP，暂停或继续日志记录在文件系统的功能。这些功能的控制通过该字符串控制，比如ulog_man("tcpip on=1")则通知ulog使能udp输出日志功能 。

函数原型

```
int ulog_man(const char *cmd_str)
```

输入参数

const char *cmd_str	控制命令字符	"tcpip on=1"

返回参数

0	设置成功
负值	由于入参不合法造成设置失败

调用示例

```
ulog_man("tcpip on=1");
```

aos_get_ulog_list()

获取文件系统记录日志文件列表

函数原型

```
int aos_get_ulog_list(char *buf, const unsigned short len)
```

输入参数

char *buf	用以记录结果的buffer	buffer
len	buffer长度	256

返回参数

0	设置成功
负值	由于入参不合法造成设置失败

调用示例

```
char buf[256];
memset(buf, 0, sizeof(buf));
aos_get_ulog_list(buf, sizeof(buf));
```

配置说明

在AliOS Things 源码里面输入 `aos make menuconfig` 命令即可进入到menuconfig配置界面中，依次选择 `Middlware Configurations` -> `uLog Configuration` 即可进入到uLog组件的配置界面：

配置项说明

```
--- uLog Configuration
(6)  Level Stop Filter of Default Direction(UART for RTOS)        # 默认定向日志过滤优先级，默认6
(256) Max Length of Log Text  # 日志内容长度限制，默认256
[]  Support syslog Time Format when Log      # 使用syslog时间格式记录日志，默认不开启
[] Enable Rich Details for Log                  #记录更多细节如日志产生文件及行号，默认不开启
[] Switch on Pop Out Log to Cloud             #日志抛出到云端运维控制台，默认不开启
[] Switch on Pop Out Log Into File System      #日志抛出到文件系统，默认不开启
[] Switch on Pop Out Log via Syslog UDP        #日志抛出到udp syslog服务器，默认不开启
[] Using Async Mode to Log on Default Direct  #对默认定向日志抛出仍使用异步机制，默认不开启
```

其它

syslog日志优先级

优先级遵循syslog协议定义，其中 `LOG_NONE` 用于控制日志输出，值越低相应优先级越高。

```
#define LOG_EMERG   0 /* system is unusable */
#define LOG_ALERT   1 /* action must be taken immediately */
#define LOG_CRIT    2 /* critical conditions */
#define LOG_ERR     3 /* error conditions */
#define LOG_WARNING 4 /* warning conditions */
#define LOG_NOTICE  5 /* normal, but significant, condition */
#define LOG_INFO    6 /* informational message */
#define LOG_DEBUG   7 /* debug-level message */
#define LOG_NONE    8 /* used in stop filter, all log will pop out */
```

当前设备主要使用其中的Alert，Critical，Error，Warning，Info，Debug这6个优先级（对应API参见API介绍--客户使用API）

3.开发工具和编译配置

3.1. 配置系统简介

本文档仅适用于AliOS Things 3.1及其后续版本，3.1之前版本请忽略。

自AliOS Things 2.1开始，引入了基于Menuconfig的图形配置系统，组件配置文件（Config.in）是配置系统工作的基础。本文将对系统中配置文件及其组织方式进行简要介绍。

更多关于配置文件编写和语法介绍请参考Linux官方文档。

在系统范围内，配置文件划分为两类：

- 顶层配置文件：配置文件入口，包含组件配置文件和系统配置变量
- 组件配置文件：描述组件配置能力，定义与组件相关的配置选项

顶层配置文件

顶层配置文件，在用户创建工程时动态生成，放置于工程目录下：$APPDIR/Config.in。它仅包含了当前工程所依赖组件的配置文件。 用户修改了应用代码，并且引入了新的组件或者删除以前引入的组件，代码编译时会先自动更新顶级配置文件，增加或删除相应组件的配置文件。 一个典型的顶层配置文件采用如下格式：

```
APP配置项
系统配置项
source 必选组件配置文件
if 选择条件
source 可选组件配置文件
endif
```

组件配置文件

组件配置文件是构成组件的基本要素，系统通过组件配置文件描述组件的配置能力。自AliOS Things2.1起，开发组件配置文件将与添加组件代码、添加组件Makefile一样，成为组件开发的必要步骤。通常情况下，每个组件都应该具有：

- 一个唯一的组件ID
- 根据组件配置能力提供具体的配置选项

配置项命名规范

- 配置项全部使用大写字母，各域之间使用“_”分隔；
- 组件ID定义：AOS_组件类型_组件名称。类型取值范围：APP，COMP，BOARD，MCU，ARCH。例如：AOS_BOARD_DEVELOPERKIT，AOS_APP_HELLOWORLD，AOS_COMP_RHINO。
- 组件内部配置项推荐的定义方式：组件名称_CONFIG_XXX。例如：RHINO_CONFIG_SEM。

与传统配置系统的差异点

- 顶层配置文件在工程目录下，$APPDIR/Config.in。
- 组件配置文件中的差异点：
- 组件默认为使能，且不允许改为禁止。
- 不使用select使能其它组件，而是在组件Makefile里面选择依赖组件。
- 可以配置可选组件的选择条件，配合组件Makefile完成选择依赖组件。

一个典型的组件配置文件采用如下格式：

```
关键字 组件ID
  类型
  默认值
关键字 组件配置项
  类型 “提示信息”
  默认值
  帮助信息
```

例如：

```
# components/network/http
# 组件唯一ID，默认为使能，不允许改为n
config AOS_COMP_HTTP
  bool
  default y
# 组件内部的配置项
menu "HTTP Client Configuration"
config HTTP_CONFIG_SERVER_NAME_SIZE
  int "The size of server name"
  default 64
  help
    The size of server name in bytes
config HTTP_CONFIG_SECURE
  bool "Support HTTP Secure"
  default n
  help
    set to y if use HTTPS
    default n
if HTTP_CONFIG_SECURE
# 配置可选组件mbedtls、itls的条件HTTP_CONFIG_SECURE_TLS、HTTP_CONFIG_SECURE_ITLS
# 组件Makefile根据此处的配置，引入相应的组件
choice
  prompt "Security Selection"
  default HTTP_CONFIG_SECURE_TLS
  help
   HTTPS over MbedTLS or iTLS
  config HTTP_CONFIG_SECURE_TLS
    bool "HTTPS over MbedTLS"
  config HTTP_CONFIG_SECURE_ITLS
    bool "HTTPS over iTLS"
endchoice
endif
endmenu
```

关于不同类型组件配置文件开发请参考：板级移植指导。

使用menuconfig配置

请参见更多配置操作。

3.2. 更多配置操作

本文档仅适用于AliOS Things3.1及其后续版本，3.1之前版本请忽略。

预定义配置是由系统预设，或者用户创建的配置内容，可以在工程中复用并作为工程配置的起点。系统使用统一的SDK目录 `AOS_SDK_PATH` 下 `build/configs` 存放预定义配置文件，用户可以加载已有配置，也可以将自定义配置保存至该目录。当用户自行创建配置文件时应当遵守以下规则：

- 文件名必须以“_defconfig”结尾，例如： `profile-linkkit-mk3060_defconfig`
- 文件名不能包含字符“@”，如需表达App@Board组合请使用其他字符代替，例如： `app-board_defconfig`

加载预定义配置

```
# 检查有效配置列表
aos make list-defconfig
# 加载预定义配置
aos make <profile-name_defconfig>
```

保存预定义配置

```
# 情形一：保存所有配置选项，包括未配置选项
# <AOS_SDK_PATH> 为AliOS Things的源代码目录
cp .config <AOS_SDK_PATH>/build/configs/<config_name>_defconfig
# 情形二：仅保存最少配置选项，不包括默认选项
aos make savedefconfig
cp .defconfig <AOS_SDK_PATH>/build/configs/<config_name>_defconfig
```

使用menuconfig修改配置

通过menuconfig，用户可以方便地调整系统和组件配置，生成符合项目需要的配置文件，具体操作步骤如下：

1）执行 `aos make menuconfig` 启动图形配置菜单：

如上图所示，menuconfig基本操作包括：

- 上下箭头键选择菜单，左右箭头键选择功能按钮
- 当选择“Select”按钮时，回车键进入高亮菜单
- 按空格键“选中/取消”菜单列表中的配置选项
- 选择“Exit”按钮返回上级菜单/退出配置界面，连续按两次Esc键退出配置界面
- 选择“Save”按钮保存当前配置
- 选择“Load”按钮加载已保存配置

2）修改配置，如：修改固件版本号

从主菜单选择“Application Configuration”，选择“Firmware Version”单选菜单，修改固件版本号，如图：

3）保存配置并退出：选择“Save”按钮，保存当前配置为“./.config”：

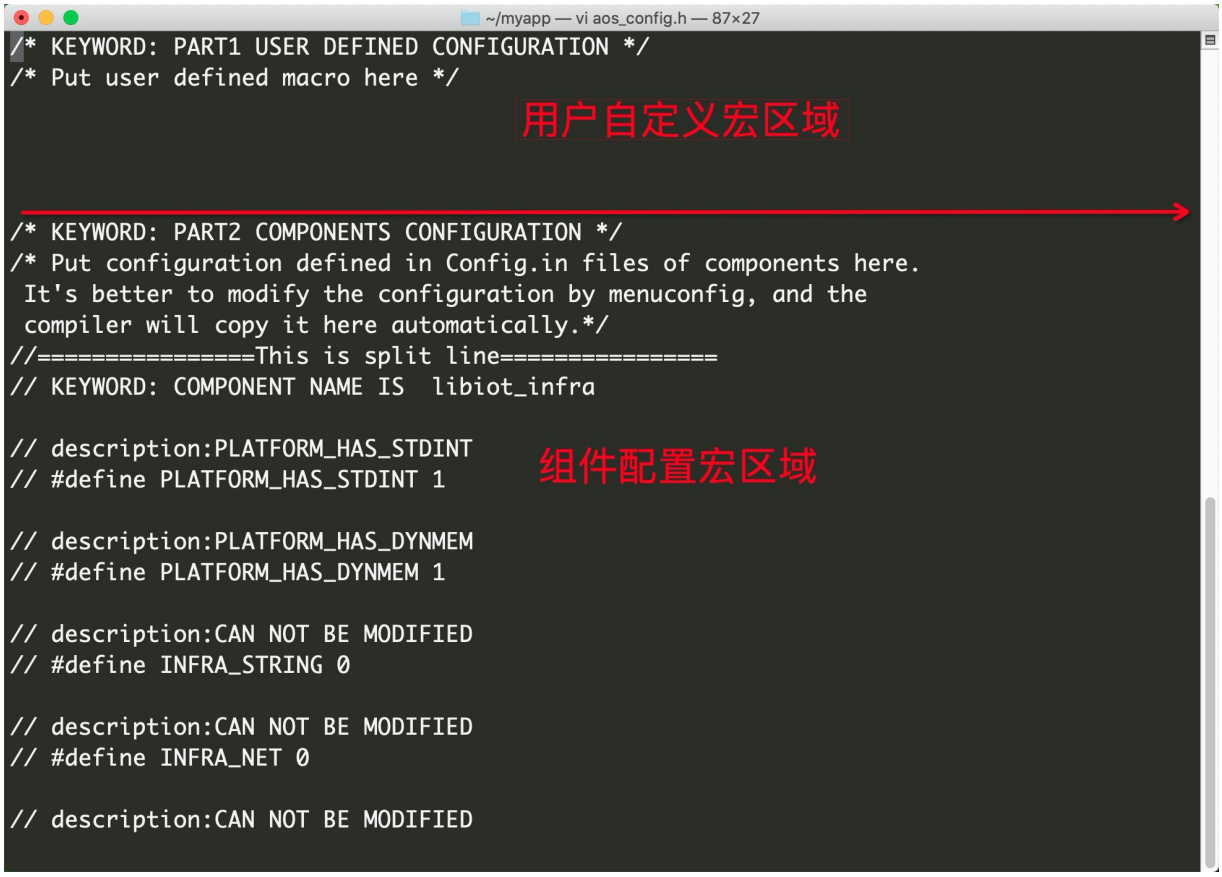
保存并退出配置界面，完成配置。构建系统将自动把.config的配置信息同步到aos_config.h文件中。

使用aos_config.h修改配置

aos_config.h分为2部分：用户自定义的宏和组件的配置所对应的宏。

初始情况下，用户自定义的宏区域不包含任何宏，由用户添加。

组件配置的宏是和.config里面的配置一一对应的，两者是靠编译系统来实现自动同步的。



用户可直接修改aos_config.h里的配置项，如修改固件版本信息： 1） 打开aos_config.h，修改固件版本号后保存退出。

2）编译的时候，构建系统将自动把aos_config.h里修改的配置自动同步到.config

```
aos make
```

3）除了编译的时候，会同步aos_config.h和.config的配置信息，使用 `aos make menuconfig` 时也会自动同步。

```
aos make menuconfig
```

首先构建系统将自动检测aos_config.h是否有修改，若有，则将配置信息同步到.config文件中； 然后打开menuconfig图形化界面，修改配置； 最后保存配置信息至.config文件后，构建系统又将.config的配置信息自动同步到aos_config.h文件中。

增加组件

用户使用 `aos create project` 创建了工程以后，构建系统会自动选择当前App和board所依赖的组件。 后续的开发中，用户还可以通过两种方式增加新的组件。 1）在应用代码里面包含相应组件的头文件，如增加cli组件。

```
// helloworld.c
#include <aos/kernel.h>
#include <aos/cli.h>  // 增加组件头文件
// other code
......
```

2）修改工程目录下的aos.mk，在 `$(NAME)_COMPONENTS_CUSTOMIZED +=` 这一行加上组件名。

```
# 通过源代码里面#include <组件头文件> 引入的组件，自动存放在此处。用户不要修改。
$(NAME)_COMPONENTS_CUSTOMIZED :=
# 用户想要通过修改aos.mk新增的组件，请在此处修改。
$(NAME)_COMPONENTS_CUSTOMIZED += cli
# 将上述组件传递给构建系统。用户不要修改。
$(NAME)_COMPONENTS += $($(NAME)_COMPONENTS_CUSTOMIZED)
```

3）增加了新的组件以后，调用aos make或者aos make menuconfig等命令后，新的组件配置将自动出现在aos_config.h中的组件配置宏区域。

清除工程和配置

当需要清除当前工程和全部配置文件时，恢复到初始状态，执行命令：

```
aos make distclean
```

视频演示

3.3. 构建及编译命令详解

本文档仅适用于AliOS Things 3.1及其后续版本，3.1之前版本请忽略。

本文将详细介绍AliOS Things 3.1支持的构建命令及其用途，帮助开发者快速掌握高级命令行构建功能。

AliOS Things 3.1代码可视化裁剪页面地址：https://aliosthings.iot.aliyun.com/aos/download。

关于工程配置请参考更多配置操作。

AliOS Things 2.1开始全面支持menuconfig配置系统，并将工程配置做为构建的必要步骤之一。与工程配置相关的命令包括：

- 根据用户指定的App、Board创建工程，并生成默认配置：

```
aos create project -b board -t templateapp -d destdir appname
```

其中-b 指定板子名称，必选。 -t 指定App模板，可选；若不指定，则使用最精简的templateapp作为模板。 -d 指定工程目录，可选；若不指定，则使用当前目录。

- 打开menuconfig菜单，允许用户在交互状态下自定义工程配置：

```
aos make menuconfig
```

- 列出当前可以用的预定义配置：

```
aos make list-defconfig
```

- 为当前工程加载预定义配置：

```
aos make <configname>_defconfig
```

更多关于预定义配置的说明请参考更多配置操作。

- 清除工程编译结果，恢复配置文件至初始状态（“aos make clean”仅清除编译结果）

```
aos make distclean
```

编译相关命令和参数

在完成工程配置后，编译命令简化为：

```
aos make
```

通过附加命令行参数的方式可以控制编译行为，实现高级编译功能：

```
aos make [VAR=value]
```

有效命令行参数包括：

- **MBINS=[app|kernel]**：分别生成App、Kernel镜像，详情请参见多bin特性。
- **BUILD_TYPE=[debug|release|release_log]**：控制构建类型：
- debug：编译时不优化（-O0）并启用日志
- release：编译时优化（-Os），不启用日志
- release_log：默认值，编译时优化，并启用日志
- **IDE=[keil|iar]**：生成指定工程并执行编译，仅支持Windows环境，详情请参见自动生成Keil MDK工程、自动生成IAR工程
- **COMPILER=[armcc|iar|rvct]**：使用指定编译器执行编译，仅支持Windows环境
- **VERBOSE=1**：编译时输出详细命令
- 更多命令行参数请查看 `aos make help`

创建组件相关命令和参数

在AliOS Things目录下，基于组件脚手架创建一个新的组件，命令如下：

```
aos create component -t <component_type> <component_name>
```

其中 component_type 表示组件类型，目前支持的类型有：

```
bus:    本地通讯协议
dm:    设备管理
fs:    文件系统
generals:  通用类
gui:    人机交互界面
language:  脚本引擎
network:   IP网络协议栈
peripherals: 外设驱动
security:  安全
service:  应用组件
utility:  工具类
wireless:  无线类
```

component_name 表示组件名称。

如果组件内部带示例APP，可使用 `aos open` 命令打开，调试组件功能。

```
aos open <component_name> -b <board_name>
```

其中 component_name 表示组件名称。 board_name 表示board名称。

其他命令

AliOS Things 3.1构建系统还支持以下命令：

- 查看帮助

```
aos make help
```

- 清除构建生成文件

```
# 清除构建生成文件，不包括配置文件
aos make clean
# 连同配置文件一同清除
aos make distclean
```

- 仅生成Keil、IAR工程，不执行编译

```
# 生成Keil工程
aos make export-keil
# 生成IAR工程
aos make export-iar
```

- 将aos_config.h的配置项同步到.config文件中

```
aos make update
```

如果用户安装了graphviz，执行该命令也会生成组件依赖文件comp_deps_topo.png。 如果没有安装graphviz，会提示 `sh: 1: dot: not found` 信息，但是不影响上述的同步操作。

更多命令和参数请以 `aos make help` 为准。

3.4. Keil/IAR 工程支持列表

在web页面上选且只选了一个board和一个app组件时，才可以选择用Keil或者IAR的形式下载！页面地址：https://aliosthings.iot.aliyun.com/aos/download 。

目前AliOS Things可支持以下board的keil/IAR工程文件输出：

1	Cortex-M3	mvs_ap80xx	ap80a0	helloworld	✔	
2	Cortex-M4	stm32l475	b_l475e	helloworld	✔	✔
3	Cortex-M4	stm32l4xx_cube	developerkit	helloworld	✔	✔
4	Cortex-M0	es8p508x	es8p5088fllq	helloworld	✔	
5	Cortex-M0	fm33a0xx	fm33a0xx-discovery	helloworld	✔	
6	Cortex-M0	hc32l136	hc32l136k8ta	helloworld	✔	
7	Cortex-M3	hk32f103	hk32f103rb_evb	helloworld	✔	
8	Cortex-M0	hr8p2xx	hr8p287fjlt	helloworld	✔	
9	Cortex-M0	hr8p2xx	hr8p296fllt	helloworld	✔	
10	Cortex-M4	lpc54114impl	lpcxpresso54114	helloworld	✔	✔
11	Cortex-M4	lpc54628impl	lpcxpresso54628	helloworld	✔	✔
12	Cortex-M3	sscp131	ssc1667	helloworld	✔	
13	Cortex-M4	stm32l4xx_cube	starterkit	helloworld	✔	✔
14	Cortex-M4	stm32f4xx_cube	stm32f401re-nucleo	helloworld	✔	✔
15	Cortex-M4	stm32f4xx_cube	stm32f411re-nucleo	helloworld	✔	✔
16	Cortex-M4	stm32f4xx_cube	stm32f412zg-nucleo	helloworld	✔	✔
17	Cortex-M4	stm32f4xx_cube	stm32f429zi-nucleo	helloworld	✔	✔
18	Cortex-M4	stm32l4xx_cube	stm32l432kc-nucleo	helloworld	✔	✔
19	Cortex-M4	stm32l4xx_cube	stm32l476rg-nucleo	helloworld	✔	✔
20	Cortex-M4	stm32l4xx_cube	stm32l476rg-nucleo	blink	✔	✔

21	Cortex-M4	stm32l4xx_cube	stm32l496g-discovery	helloworld	✔	✔

3.5. 集成开发管理工具aos-cube

AliOS Things uCube是AliOS Things基于命令行的开发管理工具（命令简写为aos），主要功能包括：工程配置与编译、Image下载调试、应用开发框架、脚手架、组件安装管理、设备管理、代码审查、OTA工具功能以及与IDE集成功能。

uCube 基于 Python（支持Py 2.7和3）语言开发，需要有Python开发环境。如开发环境中尚未安装aos-cube，请通过 `pip install -U aos-cube` 进行安装（需确保先安装了python和pip工具）。

使用

工具改善计划

从0.3.11之后的版本开始，安装aos-cube工具后，首次运行aos-cube命令时会有如下提示：

```
***Attention***:
============================================================
  In order to improve the user experience of this tool,
  we want to collect some of your system information
  (e.g. OS version, IP, shell used, aos-cube functionalities
  most frequently used, etc.). All collected information
  will be only be used in the user-experience improve plan,
  and will be carefully and restrictly secured.
============================================================
Do you want to participate in the activity?
Please type 'Y[es]' or 'N[o]': Y
```

该提示旨在提示用户参与工具改善计划，用户可自由选择，用户选择Yes后会有部分信息收集用于工具使用的统计和后续改善。**建议用户选择Yes参与该计划**。

查看版本和帮助

使用 `aos --version` 查看aos-cube版本。 使用 `aos -h` 查看aos-cube支持的命令。 使用 `aos <cmd> -h` 查看命令的格式、选项和参数说明。 使用 `aos <cmd> <subcmd> -h` 查看子命令（如有）的格式、选项和参数说明。

环境变量设置

用户拿到AliOS Things SDK后，开发前首先需设置“AOS_SDK_PATH”环境变量（指向aos源码根目录）。

创建应用

通过 `aos create project -b <board_name> -d <project_dir> -t <example_templete> <projectname>` 创建应用工程，开始应用开发。

配置

切换到应用工程目录，执行 `aos make menuconfig` 可进行可视化配置。同时，配置也可以通过修改应用工程中的aos_config.h文件进行。

更新工程元数据

当开发中修改了代码和配置，可以通过 `aos make update` 进行工程元数据的更新。

工程构建

完成应用开发后，可以通过构建命令 `aos make` 进行编译。关于使用 `aos make` 执行构建请参考使用命令行工具开发。

烧录与调试

构建完成后，连接开发板到开发主机，首先将编译生成的镜像上传到开发板，然后启动调试器：

```
aos upload helloworld@developerkit
aos debug helloworld@developerkit
```

AliOS Things 2.1及其后续版本支持简化的烧录和调试命令行（需要aos-cube 0.3.x）：

```
aos upload
aos debug
```

注意，命令行烧录和调试功能目前仅支持stm系列、esp32/esp8266等部分硬件平台。

串口监控

通过连接串口，查看应用输出和执行过程：

```
$ aos monitor <serial port> <baudrate>
```

SAL/MAL驱动开发

如果是进行SAL/MAL外设驱动开发，可以借助脚手架自动生成驱动框架，然后在对应的框架中完成驱动开发。

- 生成SAL驱动框架：`aos create saldriver -t <gprs|wifi|lte|nbiot|eth|other>`
- 生成MAL驱动框架：`aos create maldriver -t <gprs|wifi|lte|nbiot|eth|other>`

组件开发与管理

从3.1版本开始，AliOS Things支持组件话开发和管理。官方认证的组件发布在云端服务器，用户可以通过aos-cube命令进行组件的安装和管理。

组件查询

- 本地组件查询：`aos list comp`
- 云端组件查询：`aos list comp -r -a`

组件安装

- 从云端安装最新版本组件：`aos install comp <name>`
- 从云端安装特定版本组件：`aos install comp <name=version>`
- 多个组件安装：`aos install comp <name1> <name2> <name3>`
- 从本地文件安装组件：`aos install comp -L <zip_or_rpm_file>` 。本地组件文件，可以是通过 `aos pack` 命令生成的ZIP文件，也可以是从AliOS Things组件官网下载的RPM文件。

组件卸载

- 单个组件卸载：`aos remove comp <name>`
- 多组件卸载：`aos remove comp <name> [name2] [name3]`

组件升级

- 查询可升级组件：`aos upgrade comp -c`
- 组件升级：`aos upgrade comp <name1> <name2>`

组件开发

- 使用脚手架生成组件框架：`aos create component -t <bus|dm|fs|gui|language|linkkit|network|peripherals|security|service|utility|wireless|generals> <compname>`
- 根据需要增加源码和头文件，并修改对应的aos.mk文件。规范详见《组件规范》。
- 组件打包与分享：`aos pack <comp_dir>`
- 打开组件示例工程：`aos open -b <board> <comp_name>`

命令速查表

3.6. AliOS Studio图形化IDE插件

介绍

`AliOS Studio` 是一套基于vscode的开发环境，支持**windows**、**linux**、**macOS**。 `AliOS Studio` 有以下功能：

- 极佳开发体验、简单操作界面
- 支持AliOS Things应用开发
- 代码补全、索引、提示等
- 编译/下载/调试 AliOS Things
- 适配多种开发板
- 串口工具、TSL转换工具等

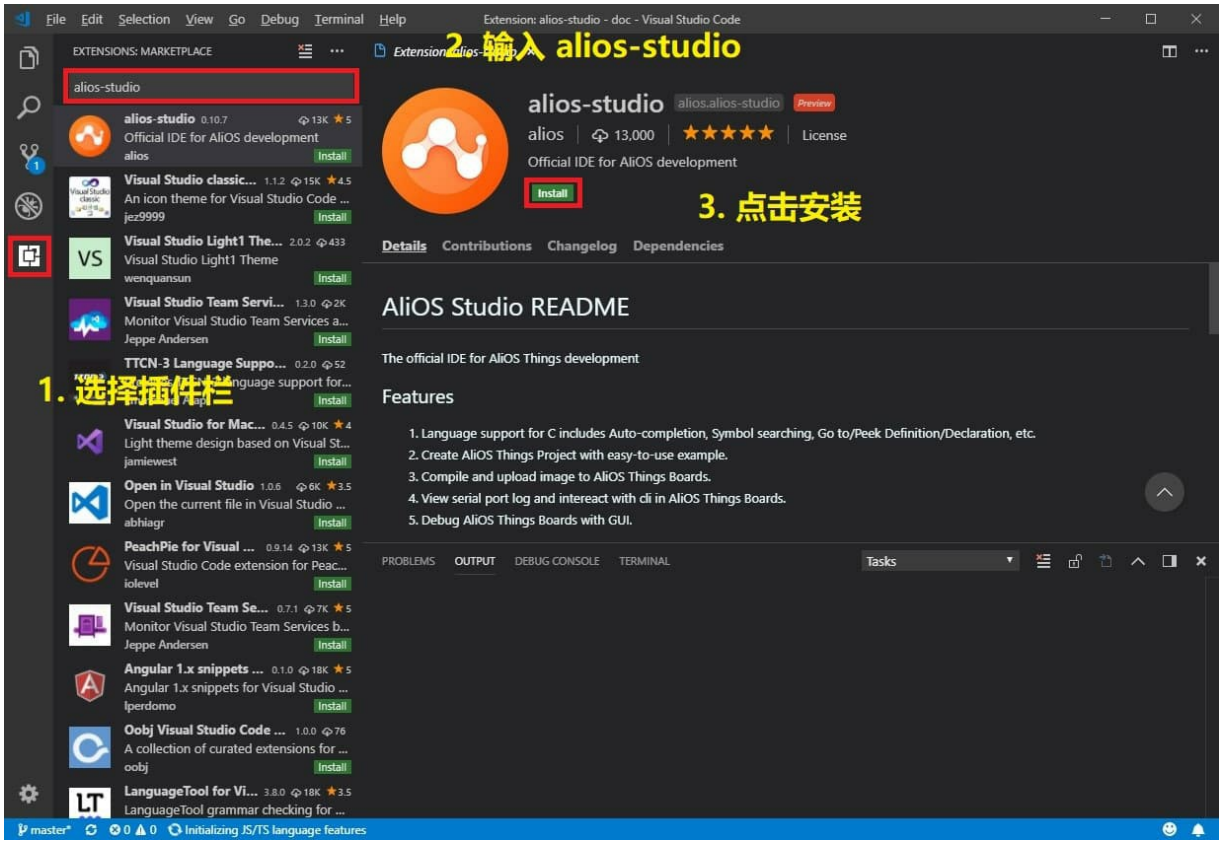
安装

下载并安装 Visual Studio Code

访问 https://code.visualstudio.com/ 下载并安装vscode。

安装 AliOS Studio 插件

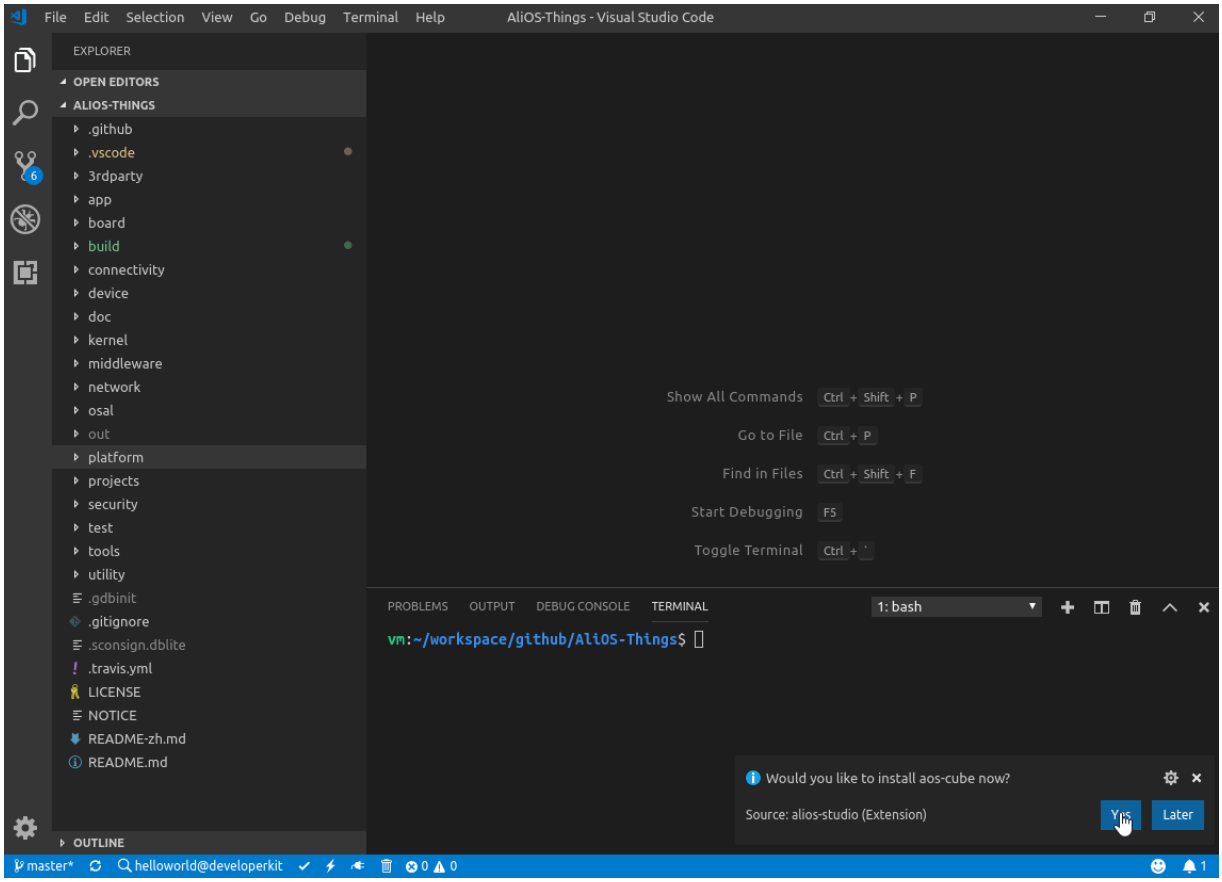
打开vscode，按照下图所示安装 `AliOS Studio` 插件：



安装 aos-cube

AliOS Studio 依赖 aos-cube，如果想要手动安装 aos-cube 的话，请参考 开发工具: aos-cube，同时 AliOS Studio 也支持一键安装 aos-cube，如下图所示：

使用AliOS Studio一键安装功能首先需要安装python2.7和pip。



`AliOS Studio` 一键安装的 `aos-cube` 是安装在虚拟python环境里面的(virualenv)，在vscode的终端里面能够正常使用 `aos-cube` ，其他终端无法正常使用 `aos-cube` 。

使用

AliOS-Studio 工具栏

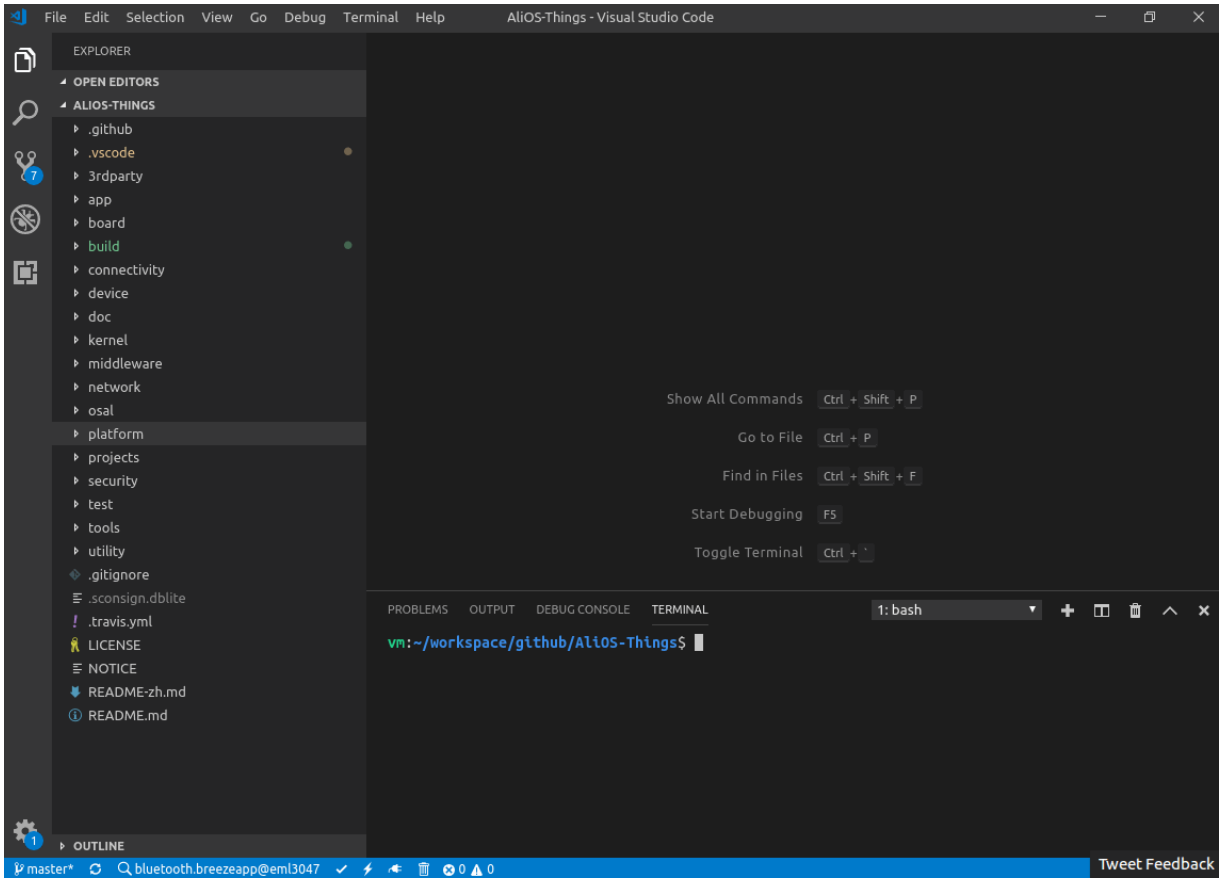
`AliOS Studio` 的主要功能都集中在vscode**下方工具栏**中，小图标从左至右功能分别是 `创建应用工程` `编译` `烧录` `串口工具` `清除` 。

注： 当用vscode打开了AliOS Things源码或者应用工程时，才会显示全部的工具图标。

左侧的 `helloworld@developerkit` 是编译目标，格式遵循 `应用名字@目标板名字` 的规则，点击它可以依次选择应用和目标板。

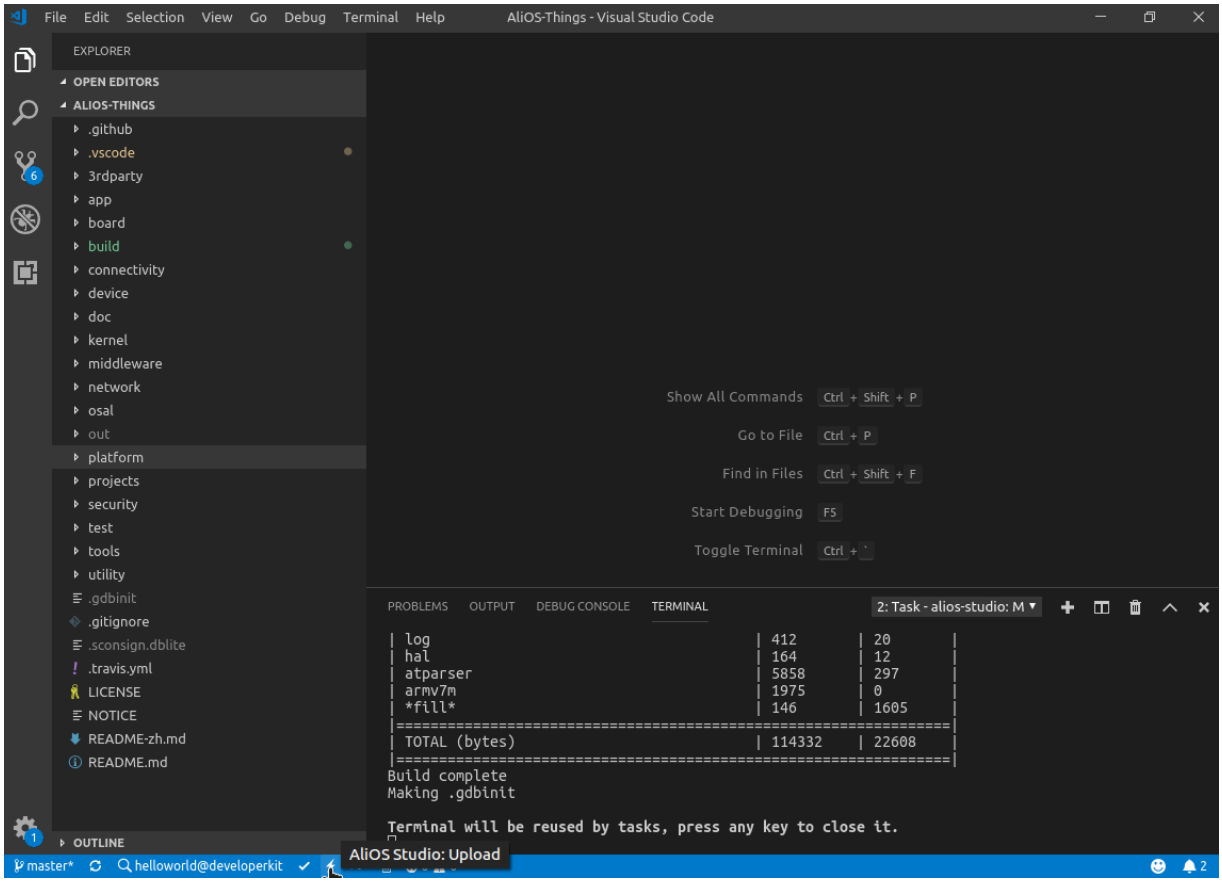
编译 - Build

点击 `编译目标` 选择应用和目标板，点击编译图标进行编译：



烧录 - Upload

1. 通过 USB Micro 线缆连接好开发板和电脑
2. 点击下方工具栏闪电图标完成固件烧录：



这里可以看到目前支持烧录（upload）的开发板，如果想要自己添加开发板支持，请参考：

- https://github.com/alibaba/AliOS-Things/tree/master/build/site_scons
- 让你的开发板支持AliOS Studio烧录

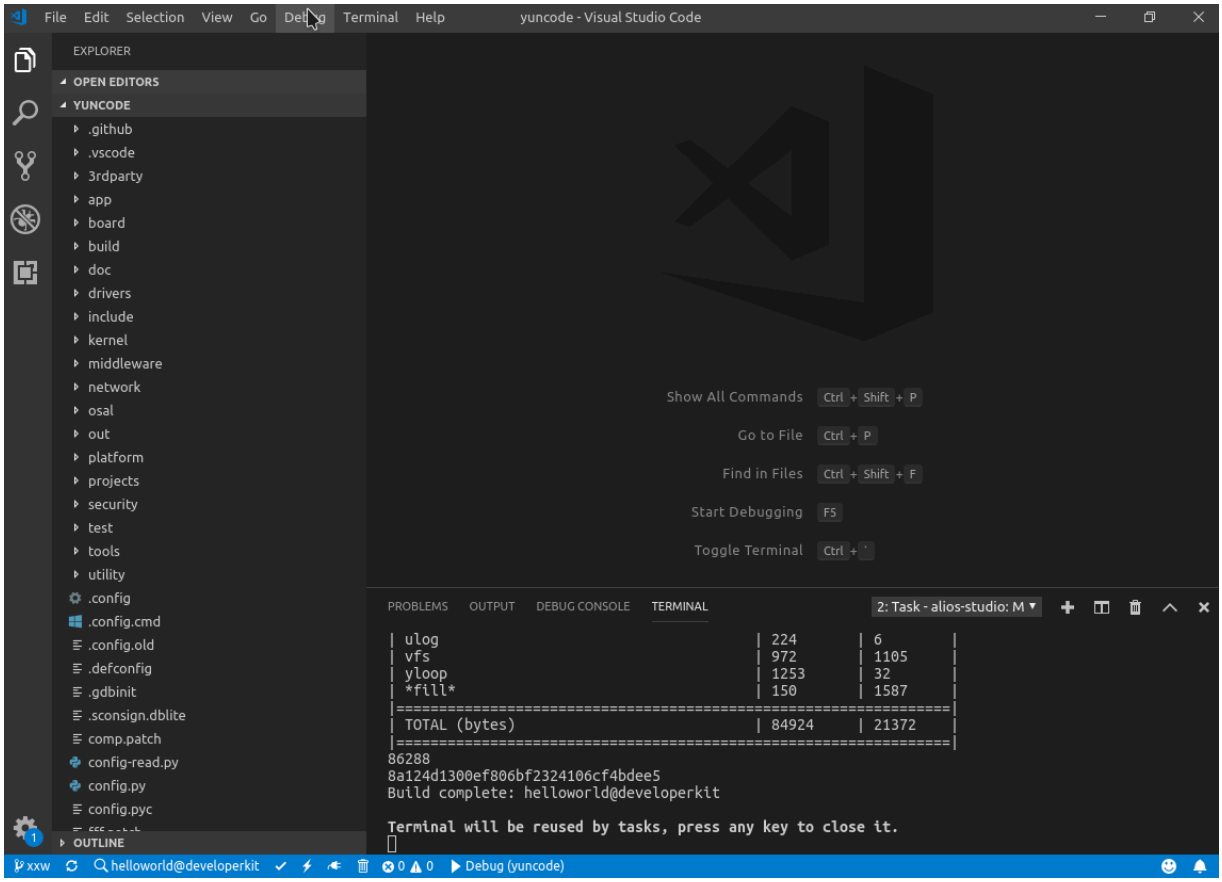
串口监控 - Monitor

1. 通过 USB Micro 线缆连接好开发板和电脑
2. 点击下方工具栏插头图标打开串口。第一次连接会提示填写串口设备名和波特率，再次点击可以看到串口输出，同时也可以在这里输入命令进行交互。

这里如果打开串口出错，请注意你的用户是否有串口访问权限。

调试 - Debug

按 `F5` 或者点击菜单栏 `Debug` > `Start Debugging` 进入调试模式：



这里可以看到目前支持调试（debug）的开发板，如果想要自己添加开发板支持，请参考：

- https://github.com/alibaba/AliOS-Things/tree/master/build/site_scons。
- 让你的开发板支持AliOS Studio烧录

参考视频：使用 AliOS Studio 开始 AliOS Things 调试。

设置优化等级

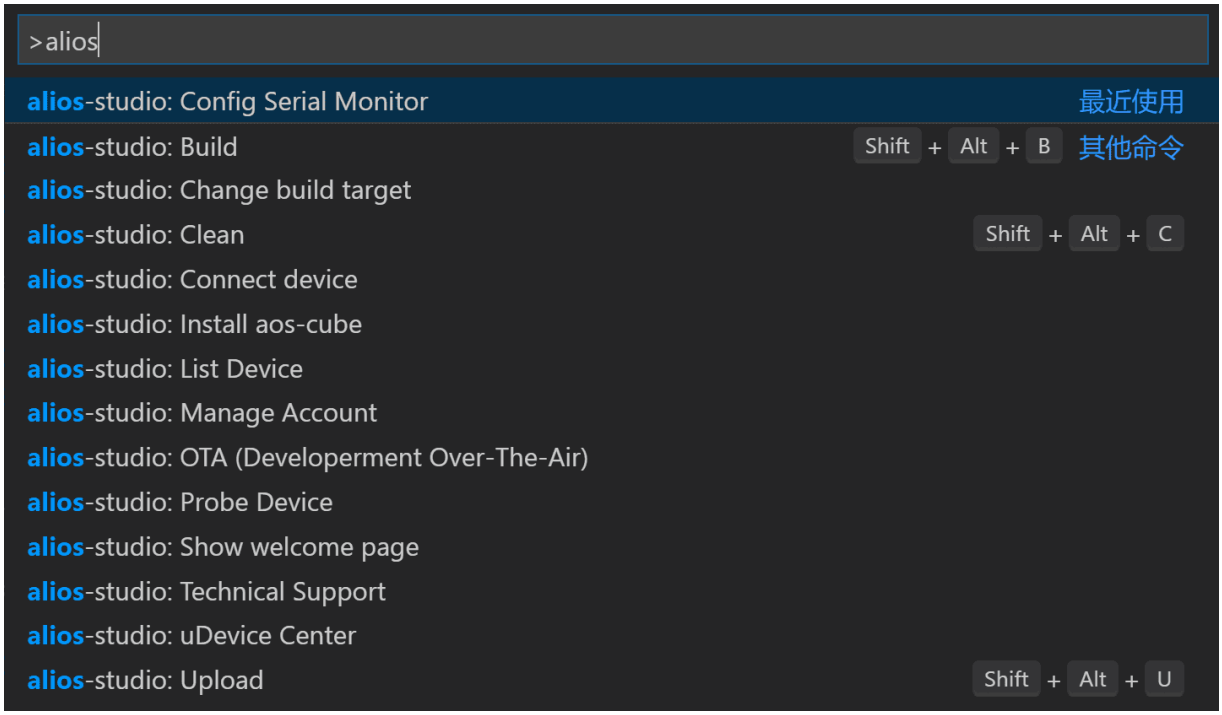
使用调试功能，最好设置优化等级为 `-Og` 或者 `-O0` ，否则会出现函数跳转异常、单步调试异常、变量optimize-out等问题。设置优化等级：

- AliOS Things 2.1版本以前：手动更改 `build/aos_toolchain_arm-none-eabi.mk` 中的 `COMPILER_SPECIFIC_OPTIMIZED_CFLAGS` 变量为-Og 或者 -O0。
- AliOS Things 2.1版本及以后：使用命令 `aos make BUILD_TYPE=debug` 即可。你也可以参考配置项：task.json中的说明，更改默认的Build选项。

更多说明

AliOS Studio 命令列表

按 `Ctrl-Shift-P` 打开vscode的命令面板，输入 `alios-studio` 可以看到 `AliOS Studio` 支持的命令：



命令说明：

Build	编译： aos make app@board	
Change build target	改变编译目标：app和board	
Clean	清除： aos make clean	
Config Serial Monitor		
Connect device	打开串口： aos monitor	
Install aos-cube	参考：AliOS Studio一键安装aos-cube	-
List Device	列出所有串口	-
Manage Account	管理阿里云账号	-
Probe Device	-	-
Technical Support	打开钉钉	-
Upload	上传固件到开发板： aos upload app@board	
OTA(Developerment Over-The-Air)	一键OTA功能	-
Show welcome page	显示welcome页面	-

AliOS Studio 快捷键

默认快捷键：

shift+alt+b	alios-studio.build	编译
shift+alt+c	alios-studio.clean	清除
shift+alt+u	alios-studio.upload	上传固件

也可以在 keybindings.json 中自定义自己喜欢的按键组合：

```
[
  {
   "command": "alios-studio.build",
   "key": "shift+alt+b"
  },
  {
   "command": "alios-studio.clean",
   "key": "shift+alt+c"
  },
  {
   "command": "alios-studio.upload",
   "key": "shift+alt+u"
  }
]
```

配置文件说明

在AliOS Things源码或者应用工程中，都有 .vscode/ 目录，该目录下面都有3个json文件，这些json文件分别配置不一样的功能：

- launch.json - 设置调试参数
- settings.json - AliOS Studio配置选项
- tasks.json - 设置tasks参数（包括编译、烧录、串口监控、清除等tasks）

AliOS-Things 2.1版本以后，新增加了一个 .TAGS.AOS.DB 文件，该文件是符号表数据库。

launch.json

AliOS Studio依赖C/C++插件提供的调试能力，使用 launch.json 来配置调试参数，launch.json的详细配置说明请参考：vscode-cpptools/launch.md。

每次更改编译目标(app@board)的时候，都会同步更新launch.json。

launch.json 中的关键配置项如下如所示：

```
{
  "version": "0.2.0",
  "configurations": [
    {
      ......
      "program": "${workspaceRoot}/out/helloworld@cy8ckit-149/binary/helloworld@cy8ckit-149.elf",
      "miDebuggerServerAddress": "localhost:4242",
      "setupCommands": [
        ......
        {
          "text": "target remote localhost:4242"
        }
        ......
      ],
      "osx": {
        "miDebuggerPath": "arm-none-eabi-gdb"
      },
      "linux": {
        "miDebuggerPath": "arm-none-eabi-gdb"
      },
      "windows": {
        "miDebuggerPath": "arm-none-eabi-gdb.exe"
      }
    }
  ]
}
```

配置项说明

program	gcc编译出来的elf文件，位于 `out/app@board/binary/app@board.elf`
miDebuggerServerAddress 和setupCommands	配置gdb的连接端口，不同的gdb server使用不同的端口
miDebuggerPath	gdb执行文件路径

settings.json

一般情况下无需更改settings.json的内容，AliOS Studio会根据配置自动更新。

```
{
  "aliosStudio.inner.yosBin": "aos",
  "aliosStudio.hardware.board": "developerkit",
  "aliosStudio.name": "helloworld",
  "aliosStudio.aosVersion": "2.1.0",
  "C_Cpp.default.browse.databaseFilename": "${workspaceRoot}/.vscode/.TAGS.AOS.DB"
}
```

该配置项为AliOS Things 2.1.0版本中的配置。

配置项说明

`yosBin`	-

hardware.board	编译的目标开发板
name	编译的目标应用
aosVersion	AliOS Things 2.1.0版本及以后新增该配置选项，标志当前代码的版本号。
C_Cpp.default.browse.databaseFilename	配置符号表数据库保存路径

tasks.json

vscode 的 tasks.json 官方说明请参考https://code.visualstudio.com/Docs/editor/tasks。task的属性请参考：https://code.visualstudio.com/Docs/editor/tasks#_custom-tasks。

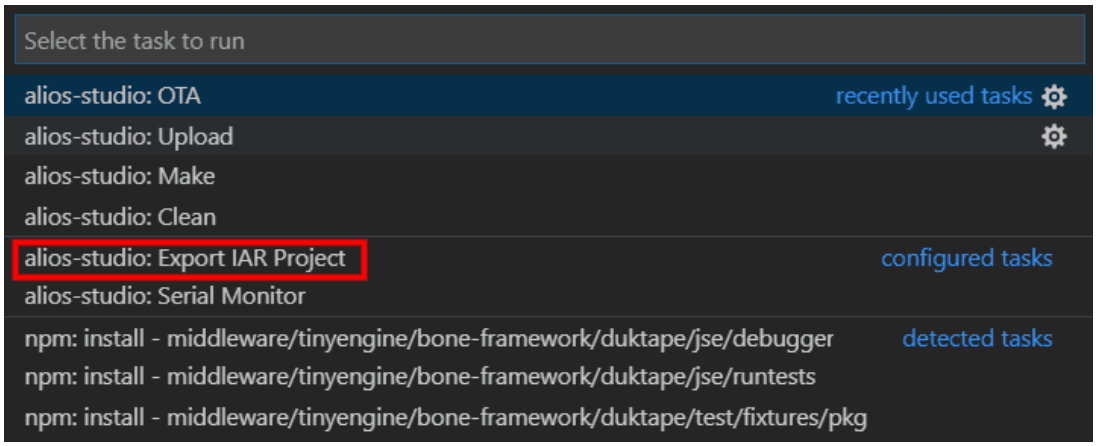
tasks.json 用来描述当前支持哪些tasks，比如点击工具栏的编译按钮(

)实际上就是执行 tasks.json 中的 alios-studio: Make 任务。

tasks.json的任务说明

alios-studio: Make	编译代码	
alios-studio: Upload	上传代码到开发板	
alios-studio: Serial Monitor	启动串口工具	
alios-studio: Clean	清除代码目标文件	
alios-studio: OTA	一键OTA功能	

当然，你也可以在tasks.json中添加自己的任务，然后依次点击vscode菜单栏的Terminal > Run Task... ，即可看到你配置的导出IAR工程的task:

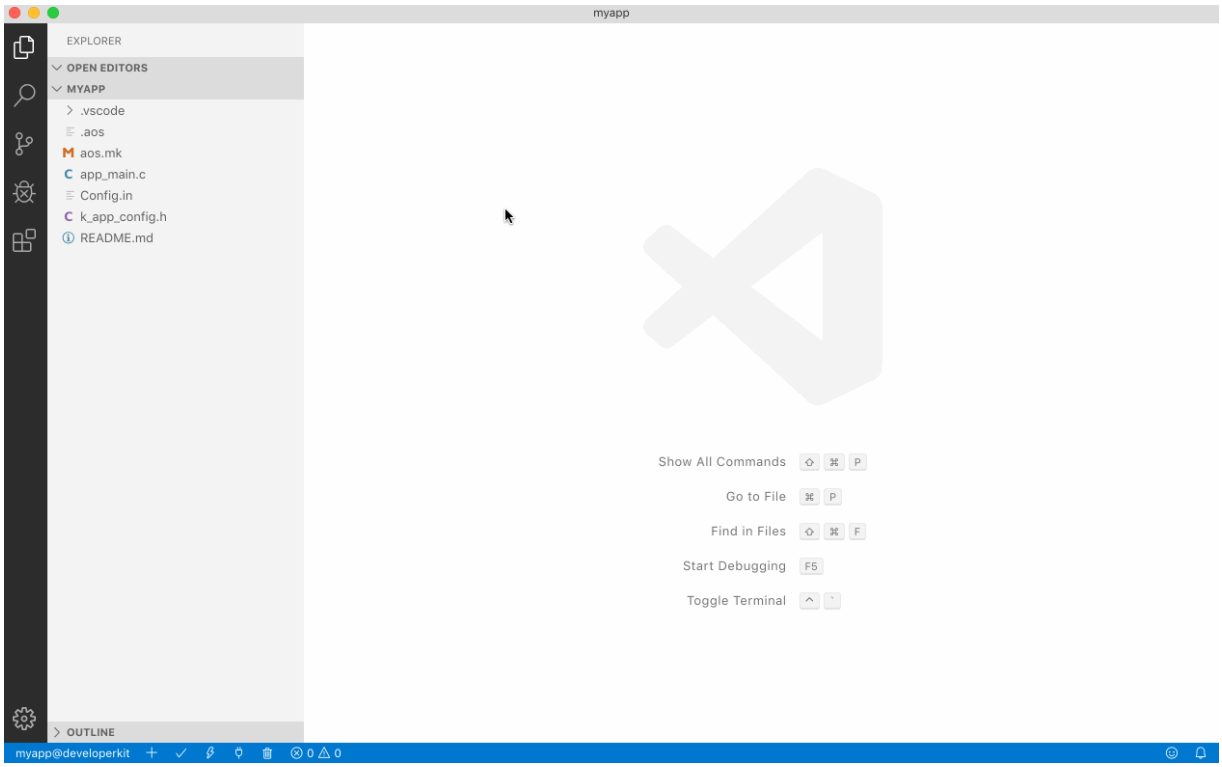


更多的自定义task可以参考 附录添加自定义task。

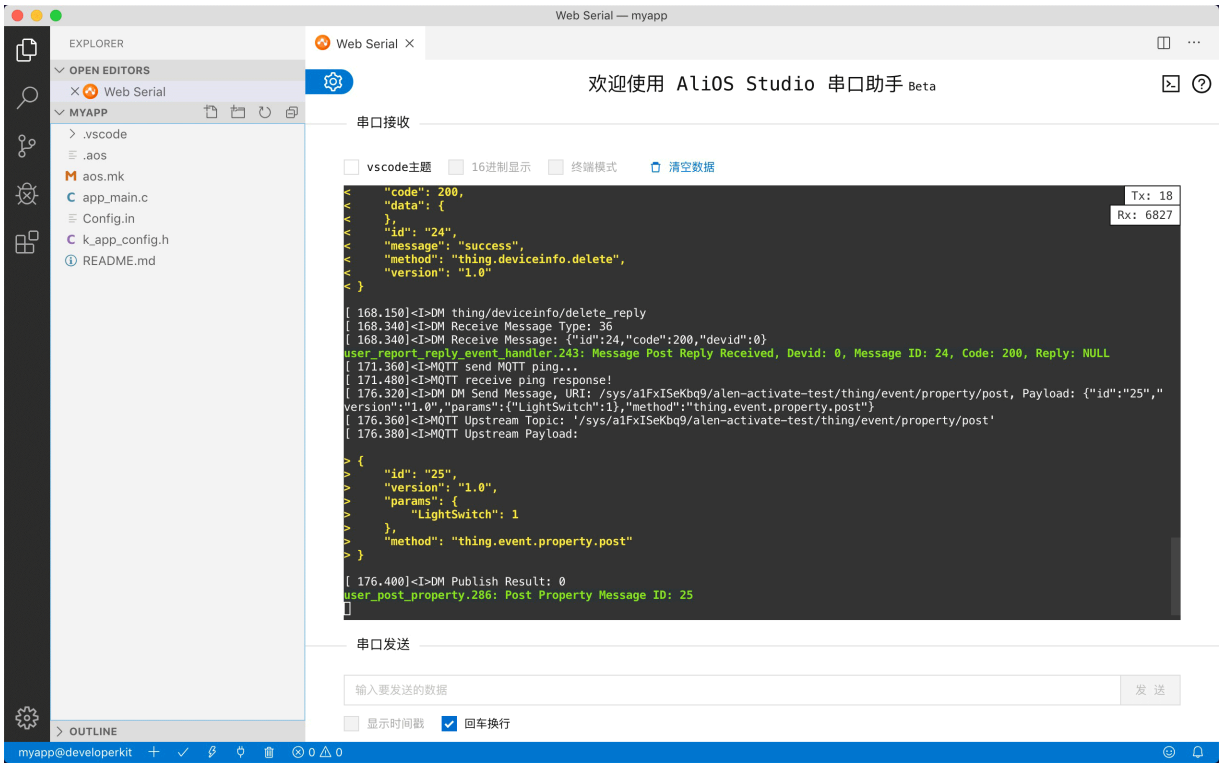
其他功能

可视化串口助手

AliOS Studio 提供了友好界面的串口助手：



AliOS Studio串口助手也支持多种vscode主题：



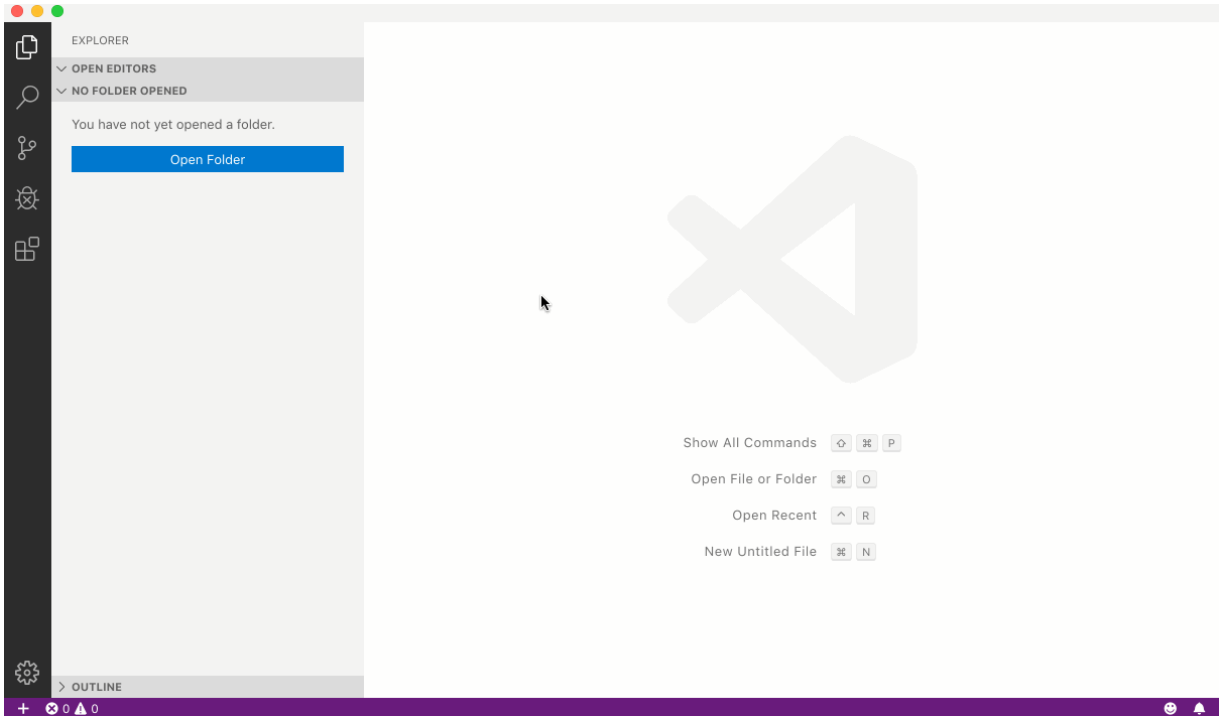
如何复制串口数据

使用快捷键复制串口数据：MacOS用户使用 `cmd+c` ，Windows和Linux用户使用 `ctr+insert` 。

AliOS Things 3.0 应用开发

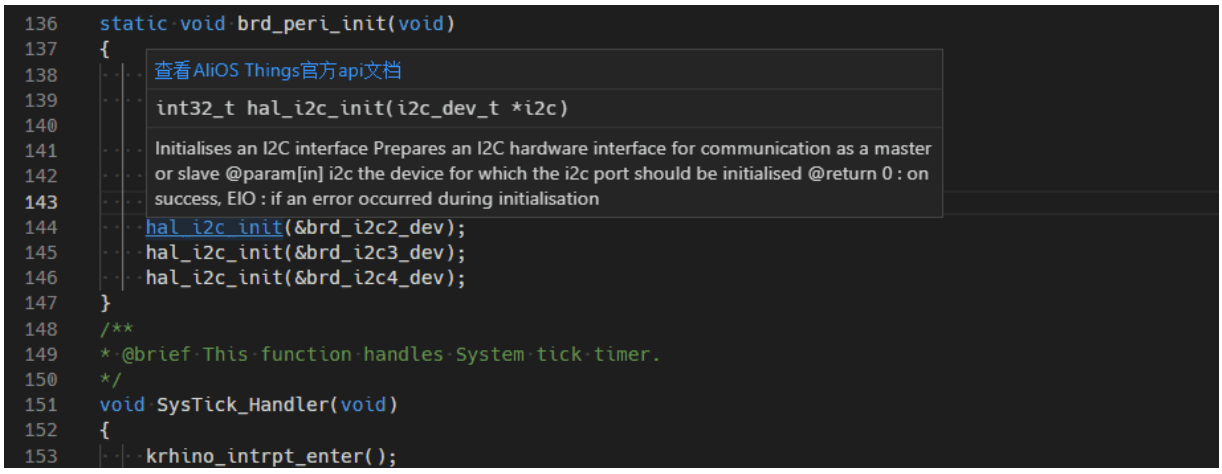
AliOS Things 3.0版本于9月27日在云栖大会正式发布，在新版本中带来了全新的应用开发框架，帮助用户快速构建自己的应用。使用户可以更专注于自身应用的开发。开发者可以在AliOS Studio中快速的创建应用工程：

要求 AliOS Things >= 3.0.0 和 aos-cube >= 0.3.7。



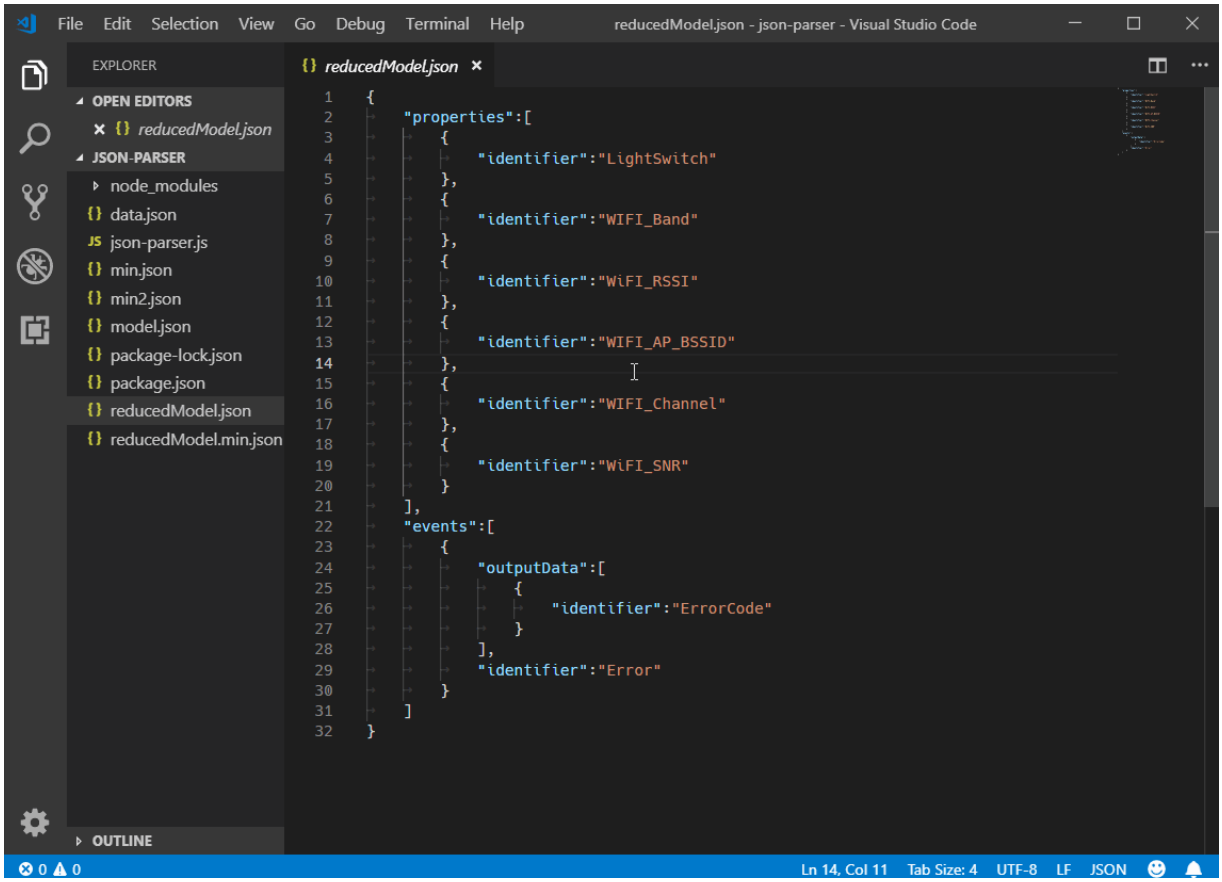
鼠标移到AliOS Things的API上会显示API说明链接

为了方便开发者尽快熟悉AliOS Things API，当鼠标移到AliOS Things的API上就会显示 查看AliOSThings 官方API文档 ：



转换TSL json文件为C代码文件

物的模型(TSL) 是阿里云IOT平台很重要的一个概念，是一个数据模型，它是物理空间中的实体，如传感器、车载装置、楼宇、工厂等在云端的数字化表示。 AliOS Studio 提供了一个高效的方法可以快速的把TSL json文件转换为C代码文件，右键json文件，然后选中 Convert TSL json to C string 即可转换：



附录

添加自定义task

添加task - 导出IAR/MDK工程：

```
{
 "label": "alios-studio: Export IAR Project",
 "type": "shell",
 "command": "aos",
 "args": [
  "make",
  "IDE=iar"
 ],
 "presentation": {
  "focus": true
 }
}
```

添加task - 多线程编译：

```
{
 "label": "alios-studio: Parallel Build",
 "type": "shell",
 "command": "aos",
 "args": [
  "make",
  "JOBS=8"
 ],
 "presentation": {
  "focus": true
 }
}
```

添加task - 编译debug类型固件：

该固件配合调试功能。

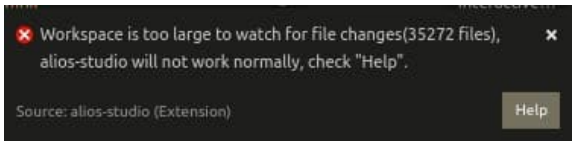
```
{
 "label": "alios-studio: Build Debug",
 "type": "shell",
 "command": "aos",
 "args": [
  "make",
  "BUILD_TYPE=debug"
 ],
 "presentation": {
  "focus": true
 }
}
```

常见问题

Visual Studio Code is unable to watch for file changes in this large workspace

针对Linux系统，windows和mac不会出现这种情况。

该错误在 linux系统上比较常见，主要是因为linux系统最大可监听文件数有限制。linux系统默认系统可监听文件数为8192个， `AliOS-Things` 的源码比较大，文件数远远大于8192个，此时vscode无法监听所有的文件改动，导致 AliOS Studio 插件会工作不正常，报如下错误：



解决办法： 此时需要设置**linux系统最大可监听文件数**。

使用如下命令查看当前可监听文件数：

```
cat /proc/sys/fs/inotify/max_user_watches
```

编辑文件： `/etc/sysctl.conf` ，然后增加如下行：

```
fs.inotify.max_user_watches=524288
```

使用如下指令生效：

```
sudo sysctl -p
```

Arch Linux 用户请参考此链接.

更多细节请参考："Visual Studio Code is unable to watch for file changes in this large workspace" (error ENOSPC)。

Workspace is too large to watch for file changes

和上面的问题一样：Visual Studio Code is unable to watch for file changes in this large workspace

SyntaxError: .vscode\launch.json: Unexpected token / in JSON at position 4378

请不要在 `.vscode/tasks.json` 或者 `.vscode/launch.json` 中添加注释。

调试模式，提示gdb is not signed

试试换个toolchain，或者删除这个toolchain，让aos-cube自己下载toolchain。

4.AliOS Things移植指南

4.1. 组件规范

4.1.1. 组件规范

组件是构成OS的基本单元，本文档通过约定组件定义、组件构成、组件操作、以及组件版本管理等内容，帮助开发者统一认识，同时也为OS构建系统和uCube等工具开发提供规范，为形成统一的组件生态打下基础。

本文适用于组件开发人员，组件管理工具和基础设施。

2 组件定义

2.1 什么是组件

从系统角度看，除了构建脚本和辅助工具外，一切都是组件；根据组件的应用范围，可以将组件划分为以下三类：

- BSP组件：物理上包含board、arch、mcu三个组件的集合；
- 系统组件：一组独立功能的集合，等同于其他系统上库（Library）的概念；
- 应用组件：直接向用户提供服务的App或Example。

2.2 组件划分粒度

原则上，应该以尽量细的粒度划分组件。影响组件粒度的因素包括：

- 独立性：组件功能应该相对独立，可以单独对外提供接口和服务；
- 耦合性：如果组件必须依赖其他组件，依靠自身始终无法对外提供服务，则考虑合并为一个组件；
- 相关性：如果一组组件共同完成一项功能，且没有被其他组件依赖，未来也没有被依赖的可能，则考虑合并为一个组件。

2.3 组件依赖关系

组件的依赖关系分为两种：必选依赖和可选依赖。

- 必选依赖：是指组件A在完成某个功能时，必须引入组件B，一起配合。例如：HTTP组件，完成访问HTTP服务器的功能，必须引入TCP/IP组件。
- 可选依赖：是指组件A在完成某个功能时，可以引入组件C，也可以引入组件D。例如：HTTP组件，要访问HTTPS服务器，可以引入mbedtls组件，也可以引入openssl组件，来完成访问服务器中加解密的功能。

3 组件构成

组件是OS功能和构建的基本单元。为了使OS能被灵活地配置，适应多种硬件平台。组件本身应当：

- 可配置：具有高度可配置的能力，依赖关系被清晰地定义和管理；
- 可发布：具有独立的版本管理和迭代能力；
- 文档全：完善的说明文档，描述组件功能和特性。

除了源代码之外，为了支持上述能力，组件应该包含：

- Config.in：负责提供组件配置选项；
- aos.mk：负责提供构建规则，解析配置文件中定义的选项，定义各选项之间的依赖关系；
- README.md：说明文档的基本功能，组件依赖和使用方法。

以上文件将作为生成组件元数据的基础，组件基础设施和工具通过组件元数据完成组件管理。

一个简单的组件示例如下：

```
mycomp
├──── aos.mk       # 负责提供构建规则
├──── Config.in    # 负责提供组件配置
├──── doc          # 组件使用手册、API说明等文档
├──── example      # 测试组件功能的示例APP
│   ├──── aos.mk
│   ├──── app.config
│   ├──── app_main.c
│   ├──── Config.in
│   ├──── main.c
│   └──── README.md
├──── inc          # 组件的内部头文件
├──── mycomp       # 组件的对外头文件
│   └──── mycomp.h
├──── README.md    # 组件的简要说明
└──── src          # 组件的源代码
    └──── mycomp.c
```

注意：

针对仅组件内部引用的头文件，是放在inc目录下；

针对组件对外公开API和数据结构的头文件，放在与组件同名的目录下。组件发布时，会将该目录复制到AliOS Things的顶层目录下的include中。

3.1 配置文件Config.in

组件需要提供配置文件Config.in，以便通过OS构建系统完成组件的配置工作。通常情况下，每个组件都应该具有：

- 一个唯一的配置ID，用于指示组件是否被使能；
- 根据组件配置能力提供具体的配置选项。

文件基本语法参考官方文档《kconfig-language.txt》以及网友整理的《Linux源码Kconfig文件语法分析》。

3.1.1 配置项命名规范

- 配置项全部使用大写字母，各域之间使用“_”分隔；
- 组件ID定义：AOS_组件类型_组件名称。类型取值范围：APP，COMP，BOARD，MCU，ARCH。例如：AOS_BOARD_DEVELOPERKIT，AOS_APP_HELLOWORLD，AOS_COMP_RHINO。
- 组件内部配置项推荐的定义方式：组件名称_CONFIG_XXX。例如：RHINO_CONFIG_SEM。

3.1.2 组件依赖的规范

- 组件默认为使能，且不允许改为禁止；否则有可能无法引入Config.in中指定的其它配置信息。
- 不在Config.in中使用select使能其它组件，而是在组件Makefile文件aos.mk里面选择依赖组件。
- 可以配置可选组件的选择条件，配合组件Makefile完成选择依赖组件。

3.1.3 组件配置示例

```
# components/network/http
# 组件唯一ID，默认为使能，不允许改为n
config AOS_COMP_HTTP
  bool
  default y
# 组件内部的配置项
menu "HTTP Client Configuration"
config HTTP_CONFIG_SERVER_NAME_SIZE
  int "The size of server name"
  default 64
  help
    The size of server name in bytes
config HTTP_CONFIG_SECURE
  bool "Support HTTP Secure"
  default n
  help
    set to y if use HTTPS
    default n
if HTTP_CONFIG_SECURE
# 配置可选组件mbedtls、itls的条件HTTP_CONFIG_SECURE_TLS、HTTP_CONFIG_SECURE_ITLS
# 组件Makefile根据此处的配置，引入相应的组件
choice
  prompt "Security Selection"
  default HTTP_CONFIG_SECURE_TLS
  help
   HTTPS over MbedTLS or iTLS
  config HTTP_CONFIG_SECURE_TLS
    bool "HTTPS over MbedTLS"
  config HTTP_CONFIG_SECURE_ITLS
    bool "HTTPS over iTLS"
endchoice
endif
endmenu
```

3.2 组件Makefile aos.mk

为了方便统一管理和标识组件，组件Makefile将统一命名为“aos.mk”。

3.2.1 通用组件配置信息

为了支持组件化编译和发布，Makefile需要提供以下信息：

- 必选字段：
- NAME：组件名称，是相应Config.in里组件ID中的组件名称的小写版
- $(NAME)_MBINS_TYPE：MBINS类型定义，支持三种类型：kernel，app，share
- $(NAME)_VERSION：组件版本（新增）
- $(NAME)_SUMMARY：单行描述信息
- 按需填写字段：
- $(NAME)_COMPONENTS：组件依赖关系（允许指定依赖组件的版本）
- $(NAME)_COMPONENTS-$(bool型配置选项)：组件可选依赖关系
- $(NAME)_SOURCES：组件源文件
- $(NAME)_INCLUDES：编译组件需要依赖的头文件

注意：参考Linux kernel处理bool类型配置选项的方式，允许组件以`$(NAME)_COMPONENTS-$(bool类型配置选项) += 组件名称`的方式按需为变量赋值。该方式同样适用于$(NAME)_SOURCES，$(NAME)_INCLUDES等。

- 其他可选字段：
- $(NAME)_LICENSE：许可证信息，默认值：Apache 2
- $(NAME)_VENDER：组件提供者，默认值：Alibaba
- $(NAME)_URL：主页地址，默认值：无
- $(NAME)_DESCRIPTION：多行描述信息，默认值：无
- $(NAME)_PREBUILT_LIBRARY：默认链接的.a，比如厂商的驱动.a等
- $(NAME)_LINK_FILES：armcc编译的时候才使用，为了解决COMPILER=armcc时，链接.a符号不全的问题

3.2.2 通用全局变量

全局变量用来定义和影响全局构建行为，根据需要定义以下变量：

- GLOBAL_INCLUDES：全局可见的头文件搜索路径，用于对外提供头文件
- GLOBAL_CFLAGS：全局可见编译选项，建议使用组件配置文件Config.in管理
- GLOBAL_CXXFLAGS：全局C++编译器选项
- GLOBAL_LDFLAGS：全局可见链接选项
- GLOBAL_DEFINES：全局可见宏定义，建议使用组件配置文件Config.in管理
- GLOBAL_ASMFLAGS：全局汇编选项
- ……

注意：使用GLOBAL_INCLUDES添加头文件路径保持 最小够用 的原则，尽量不包含其它组件的头文件路径。

3.2.3 BSP相关变量

BSP组件在物理上由board，arch，mcu三部分组成，指定board即确定了相应的arch和mcu。为了支持这一特殊依赖关系，board组件Makefile中需要提供以下信息：

- 必选字段：
- HOST_ARCH：CPU框架类型
- HOST_MCU_FAMILY：MCU名称，对应platform/mcu/*
- 其他可选字段：
- SUPPORT_MBINS：是否支持多bin编译，例如：“yes”-支持，“no”-不支持
- HOST_MCU_NAME：用来区分mcu系列下的具体某MCU名
- ENABLE_VFP：用来区分链接的库是否包含浮点数，使用umesh,a, mbmaster.a, activation,a时需要使用
- 系统自定义字段：系统或组件定义的其他字段和变量

3.2.4 在Makefile中支持KConfig选项

KConfig选项最终以两种方式影响系统：

- 全局生效的Makefile变量
- 全局生效的C语言宏定义

在Makefile中直接引用全局变量，例如：

```
ifeq ($(HTTP_CONFIG_SECURE),)
...
endif
```

在源代码中引用宏定义。例如：

```
#if HTTP_CONFIG_SECURE
...
#else
...
#endif
// 或者
#if !HTTP_CONFIG_SECURE
...
#else
...
#endif
```

3.2.5组件Makefile示例

```
## 必须定义：名称、多bin类型、版本、摘要
NAME := http
$(NAME)_MBINS_TYPE := kernel
$(NAME)_VERSION := 1.0.1
$(NAME)_SUMMARY := http client component
## 按需定义：
# 固定源码
$(NAME)_INCLUDES += include
$(NAME)_SOURCES := src/http_client.c \
           src/http_string.c \
           src/http_parser.c \
           src/http_upload.c \
           wrappers/http_aos_wrapper.c
# 固定导出头文件
GLOBAL_INCLUDES += ../../../include/network/http
# 必选依赖的组件
# $(NAME)_COMPONENTS += lwip
# 可选依赖的组件
$(NAME)_COMPONENTS-$(CONFIG_HTTP_SECURE_TLS) += mbedtls
$(NAME)_COMPONENTS-$(CONFIG_HTTP_SECURE_ITLS) += itls
# 不要使用以下方式引入可选依赖的组件
# ifeq (y,$(CONFIG_HTTP_SECURE_TLS))
# $(NAME)_COMPONENTS += mbedtls
# endif
# 根据配置选项控制的源码，例如
# $(NAME)_SOURCES-$(AOS_XXX) += yyy.c
# 注意：所有通过-D定义的选项都可以转化为KConfig配置，不需要在Makefile中重复定义，例如：
# GLOBAL_CFLAGS += -DXXX
# GLOBAL_DEFINES += XXX
# 根据配置选项处理其他逻辑，例如
# ifeq ($(AOS_ZZZ),value)
# ...
# endif
```

3.3 组件编写规范

为了使组件在不同硬件间可以快速复用，对于直接操作硬件设备的组件，必须满足以下规范：

1）使用OS提供的HAL API控制硬件，而不直接调用芯片厂商提供的底层API。

2）操作硬件设备的端口，如GPIO、I2C等，在组件API的输入参数中指定，而不能直接在组件内部指定。

a、对于使用GPIO类控制硬件的组件，GPIO端口必须以GPIO编号方式指定，如0表示PA0，1表示PA1等。具体编号参见各MCU下的映射关系表。

b、对于使用复合类端口控制硬件的组件，如I2C、UART、SPI等，复合类端口必须以相应的结构体句柄方式指定，如i2c_dev_t等。

组件编写示例和调用示例如下：

```
// GPIO类组件实现示例：
#include "aos_P9813.h"
gpio_dev_t clk;
gpio_dev_t din;
void P9813_init(int clk_pin_num,int din_pin_num)
{
  din.port = clk_pin_num;
  din.config = OUTPUT_PUSH_PULL;
  hal_gpio_init(&din);
  clk.port = din_pin_num;
  clk.config = OUTPUT_PUSH_PULL;
  hal_gpio_init(&clk);
}
// GPIO类组件调用示例：
P9813_init(24,25);
```

```
// 复合类组件实现示例：
#include "aos_AT24C.h"
void AT24C_init(i2c_dev_t *pi2c_dev)
{
  hal_i2c_init(pi2c_dev);
  ........
}
// 复合类组件调用示例：
i2c_dev_t i2c_dev;
i2c_dev.port = PORT_I2C_1;
i2c_dev.config.address_width = I2C_HAL_ADDRESS_WIDTH_7BIT;
i2c_dev.config.freq = I2C_BUS_BIT_RATES_400K;
i2c_dev.config.mode = I2C_MODE_MASTER;
AT24C_init(&i2c_dev);
```

3.4 组件预定义配置

组件预定义配置是由组件提供者预设，主要用来配置其依赖组件的参数。

仅App组件和board组件包含预定义配置。App组件的预定义配置为app.config，放置于该App根目录下；board组件的预定义配置为board.config，放置于该board根目录下。

例如，OS内核rhino组件默认采用单核CPU，即RHINO_CONFIG_CPU_NUM的默认值为1。使用双核CPU的board，可以在其相应的board.config中加入 `RHINO_CONFIG_CPU_NUM=2` 。

又如，LwIP组件默认不使能telnet功能，即LWIP_CONFIG_TELNETD_ENABLED的默认值为n。对于需要使用telnet功能的app，可以在其app.config中加入 `LWIP_CONFIG_TELNETD_ENABLED=y` 。

对于默认已配置的宏，可以通过以下方式禁止： `# <宏名称> is not set` 。

例如mbedtls组件的MBEDTLS_CONFIG_DTLS默认值为y，在app.config或者board.config中加入 `# MBEDTLS_CONFIG_DTLS is not set` ，可将该宏取消定义。

组件预定义配置的优先级是 app.config > board.config > 组件的默认配置。

用户创建工程时，将采用以下流程生成工程配置的数据文件.config。

4 组件管理

4.1 组件元数据

组件元数据是执行组件管理的基础，主要用来描述组件基本信息、系统中包含的组件索引等等。这些数据将用来构建组件仓库，支持uCube完成组件操作。

4.2 组件依赖定义

组件依赖定义放在组件Makefile中。配置文件中可以定义可选依赖的组件的条件，不能直接选择组件。例如：

```
# 组件的配置文件Config.in
# 定义可选依赖组件的条件
config HTTP_CONFIG_SECURE_TLS
  bool "HTTPS over MbedTLS"
  default n
  help
    HTTPS over MbedTLS
# 组件Makefile aos.mk
# 必选依赖的组件
$(NAME)_COMPONENTS += lwip
# 可选依赖的组件
$(NAME)_COMPONENTS-$(CONFIG_HTTP_SECURE_TLS) += mbedtls
```

在现有依赖基础上，引入版本信息，允许依赖特定版本的组件，依赖指定格式如下：

```
$(NAME)_COMPONENTS := 组件名(操作符+版本号)
例如:
$(NAME)_COMPONENTS := rhino(>=2.0.0)
```

操作符：=、<、>、<=、>=

版本号：参见组件版本号定义

注意：

- 使用半角括号“()”包含版本信息；
- “组件名(操作符+版本号)”中间不能出现空格。

4.2 组件相关操作

基于组件化，uCube工具可以针对组件执行如下操作：

- 安装
- 升级
- 搜索、查询
- 删除
- ...

uCube（也叫aos-cube）的详细操作可参考《集成开发管理工具aos-cube》。

5 组件版本管理

5.1 组件版本号

版本格式：主版本号.次版本号.修订号.[bugfix]，版本号递增规则如下：

- 主版本号：当做了不兼容的 API 修改；
- 次版本号：当做了向下兼容的功能性新增；
- 修订号：当做了向下兼容的问题修正；
- bugfix：发布后做了紧急bug修复。

5.2 版本发布

除了随OS统一发布外，组件可以独立发布新版本，而不需要重新发布OS和其他组件。关于版本发布细则参考日常发布流程。

4.2. BSP规范和移植指南

4.2.1. 板级支持目录规范

本文主要介绍AliOS Things新增单板支持，相关board/platform目录、文件的部署规范，以及相关接口定义和使用的标准。相关代码合入AliOS Things代码仓库，都需要遵循此文档规范。

代码提交和回流过程中，都会按照本规范进行代码检查，不符合的代码不予入库，请务必遵循。

AliOS Things参考版本：AOS-R-3.1.0

参考示例规范单板：

board: aaboard_demo ;

mcu: aamcu_demo;

arch: Cortex-M4

说明：

如果使用其他版本，请参考：AliOS Things板级支持目录规范

和历史版本的差异，主要包括：

a、 原根目录board目录归纳到platform下，对应platform/board目录；

b、 board下面重新梳理了几个典型单板的初始化代码样例,部分board及其hal的抽象定义和实现，主要是将主任务的入口移动到app里面实现，由app来决定board的初始化内容，而非默认在board内全初始化；相关初始化代码和流程与对应的app关联修改，原有board实现放在board_legacy目录下；

c、 对应上述新规范board目录，相应的app实现在application/example目录下；原有的app实现移入application/example/example_legacy目录。

d、板级board初始化代码按照模块统一梳理。

1、目录结构规范

下述为新增一个单板支持，必须关注的几个目录项：

目录名	介绍
app/example	通用用户运行实例，如helloworld实例，可直接使用，无特殊情况不修改
platform/board	用户需要适配、可配置board级代码，系统启动相关代码
platform/arch	该CPU架构内核调度适配接口，可直接使用

platform/mcu	该MCU通用SDK以及对应的hal适配层

1.1、example新增规范

原则上不建议新增example，除非目前的example不能满足功能需求。app/example下为通用运行实例，如果新增example，需要具有通用性，而不是为了某个特殊，或者临时性的修改。

1.1.1、目录功能

针对上层用户需要运行的具体实例出发，抽象出具体代表某个功能的实例工程，如基本的定时输出功能：helloworld_demo,

上云通道实例功能：mqttapp。app/example目录下，为已经抽象的通用运行实例，原则上用户不需要修改，直接使用现行实例。

其中app/example/example_legacy下为3.1版本之前规范的app示例，其不包括maintask.c主任务实现，其相关主任务的板级实现统一在跳转app前做全初始化，即不同的app，底层初始化的模块是一样的；

app/example目录非example_legacy下为3.1版本要求的规范app示例，其包括maintask.c主任务实现，对板级初始化做了模块区分，以期望实现不同app按照具体需求来初始化不同的board模块。

其中，app/example/example_legacy对应platform/board/board_legacy目录的单板；

其他app/example对应platform/board下非board_legacy实现。

另，platform/board/board_legacy原有单板只能支持老的example_legacy实现；新的platform/board单板，兼容app/example下所有app。

1.1.2、命名规范

运行实例取名简洁、直观，需要和具体运行的功能对应。

1.1.3、目录结构规范

文件部署规范如下，以helloworld_demo示例，

```
helloworld_demo
|-- maintask.c      # main task entry "aos_maintask"
|-- appdemo.c       # helloworld source code, including app entry ”application_start”
|-- Config.in       # menuconfig config file
|-- aos.mk          # aos build system file(for make)
|-- k_app_config.h  # aos app config file, has higher priority than k_config.h
|-- README.md
```

1.1.4、函数命名规范

主函数入口统一使用aos_maintask函数。用户可以在该接口内添加板级初始化代码。

运行实例入口统一使用application_start函数。用户可以在该接口内添加具体实现。

1.1.5、mk编写规范（aos.mk）

```
NAME := helloworld_demo            #example名，和目录统一
$(NAME)_MBINS_TYPE := app          #在多bin情况下，归属kernel还是app
$(NAME)_VERSION := 1.0.0           #menuconfig组件版本号
$(NAME)_SUMMARY := Hello World     #描述
$(NAME)_SOURCES     +=             #example.c文件
$(NAME)_COMPONENTS  += osal_aos    #依赖其他组件名
GLOBAL_INCLUDES     +=             #全局头文件
GLOBAL_DEFINES      +=             #全局宏定义
```

1.1.6、config.in编写规范

以helloworld_demo为例：

```
config AOS_APP_HELLOWORLD_DEMO       # 定义组件配置选项
  bool "Helloworld Demo"          # 配置选项类型，双引号定义该选项显示名称
  help
    Hello World               # 配置选项帮助
if AOS_APP_HELLOWORLD
# Configurations for app helloworld_demo     # 如有必要，定义更多组件内配置选项
endif
```

组件配置选项命名规范：使用前缀“AOS_APP_” + 组件NAME

将新增example加入系统配置菜单：

如果新增example在“app/exampe”目录下，编辑“app/exampe/Config.in”

如果新增example在“app/profile”目录下，编辑“app/profile/Config.in”

例如：

```
source "app/example/helloworld_demo/Config.in"  # 引用example配置文件
if AOS_APP_HELLOWORLD_DEMO              # 如果example组件被启用
  config AOS_BUILD_APP
    default "helloworld_demo"         # 为AOS_BUILD_APP赋值，与example目录一致
endif
```

注意：AOS_BUILD_APP默认值必须与配置命令行（AliOS Thing v3.1及以后，aos create project -b board -t app myapp

；3.1以前版本，aos make app@board -c config ）输入的app保持一致。

1.2、board新增规范

1.2.1、目录功能

board下统一放置板级相关启动、配置、初始化代码，以及板级外设驱动。

其中platform/board/board_legacy下为3.1版本之前规范的board示例，其对应app/example/example_legacy下app实现；在跳转到app入口application_start前完成了board的全部初始化；

platform/board下非board_legacy目录为3.1版本要求的规范board示例，其对应app/example目录下非example_legacy实现；其进入main函数后，完成最基本的初始化，并创建主任务后，直接跳转到app内主任务入口aos_maintask，并最终跳转到application_start入口。

platform/board/board_legacy原有单板只能支持老的example_legacy实现；新的platform/board单板，兼容app/example下所有app。

1.2.2、命名规范

board取名需要使用标准通用名，能方便检索获取到相关单板信息。

1.2.3、目录结构规范

board目录下文件结构部署和命名需要遵循下面布局规则，以aaboard_demo单板为例

```
Dir\File                Description                          Necessary for kernel run
|-- drivers             # board peripheral driver                          N
|-- config
|  |-- board.h          # board config file, define for user, such as uart port num     Y
|  |-- k_config.c       # user's kernel hook and mm memory region define           Y
|  |-- k_config.h       # kernel config file .h                            Y
|  |-- partition_conf.c    # board flash config file                          N
|-- startup
|  |-- board.c          # board_init implement                             Y
|  |-- startup.c        # main entry file                              Y
|  |-- startup_gcc.s    # board startup assember for gcc                       Y
|  |-- startup_iar.s    # board startup assember for iar                       Y
|  |-- startup_keil.s   # board startup assember for keil                      Y
|-- aaboard_demo.icf    # linkscript file for iar                          Y
|-- aaboard_demo.ld     # linkscript file for gcc                          Y
|-- aaboard_demo.sct    # linkscript file for sct                          Y
|-- aos.mk              # board makefile                               Y
|-- Config.in           # menuconfig component config                          Y
|-- ucube.py            # aos build system file                            N
|-- README.md                                                    Y
注：gcc、keil、iar任意支持一种即可，无需全部支持，即startup_xx.s和链接文件只需实现一套即可
```

1.2.4、函数命名规范

文件	函数名
k_config.c	实现样例单板aaboard_demo该文件内所有对接接口
partition_conf.c	统一分区初始化接口：flash_partition_init
board.c	统一单板初始化接口：board_init
startup.c	无特殊情况统一C程序主入口为main； 内部调用单板初始化board_init；内部调用krhino接口初始化内核； 内部创建主任务入口sys_init。（具体见初始化流程规范）

1.2.5、mk编写规范（aos.mk）

```
NAME := board_aaboard_demo          #board_+单板名
$(NAME)_MBINS_TYPE := kernel        #在多bin情况下，归属kernel还是app
$(NAME)_VERSION   := 1.0.1          #组件版本号
$(NAME)_SUMMARY   :=                #描述
MODULE        := 1062               #固定
HOST_ARCH     := Cortex-M4          #CPU arch
HOST_MCU_FAMILY   := mcu_aamcu_demo   #归属MCU系列，需要对应platform/mcu下aos.mk组件
SUPPORT_MBINS     := no             #是否支持app/kernel的bin分离
HOST_MCU_NAME     := aamcu1_demo      # MCU子系列类型
ENABLE_VFP        := 1              #是否支持浮点数
$(NAME)_SOURCES     +=              #board组件包含.c文件
$(NAME)_COMPONENTS   += $(HOST_MCU_FAMILY)  #依赖其他组件名
GLOBAL_INCLUDES     +=              #头文件
GLOBAL_CFLAGS       +=              #c文件编译选项
GLOBAL_ASMFLAGS     +=              #汇编编译选项
GLOBAL_LDFLAGS      +=              #链接选项
GLOBAL_DEFINES      +=              #用户自定义宏
注意：
（1）、其中HOST_MCU_FAMILY的定义需要对应platform/mcu某子目录下aos.mk中的组件名NAME，一般是mcu_+“mcu名”。HOST_MCU_NAME表示具体的mcu子系列。
（2）、用户可以通过GLOBAL_DEFINES定义宏，如GLOBAL_DEFINES += CONFIG_AOS_CLI_BOARD或者GLOBAL_DEFINES += CONFIG_AOS_KV_BLK_BITS=14。当然也可以直接在编译选项 GLOBAL_CFLAGS使用-D定义。
（3）、原则上在3.1版本中，mk只用来增加定义的组件components，其他配置和宏定义统一在Config.in定义。具体参考组件化指导文档。
```

1.2.6、config.in编写规范

以aaboard_demo为例：

```
config AOS_BOARD_AABOARD_DEMO   # 定义组件配置选项
  bool "AABOARD_DEMO"      # 配置选项类型，双引号定义该选项显示名称
  help
    ...              # 配置选项帮助
if AOS_BOARD_AABOARD_DEMO
# Configurations for board aaboard_demo
# "BSP SUPPORT FEATURE"      # 硬件支持的能力
config BSP_SUPPORT_UART
  bool
  default y
...
endif
```

board组件配置选项命名规范：使用前缀“AOS_BOARD_” + 组件NAME。

1.3、arch子目录新增规范

目前arch下已经适配了主要的CPU架构，原则上只需要直接使用。如果需要新增，需要关注下述章节。

1.3.1、目录功能

arch目录下主要是基本的CPU架构相关的porting，主要包括开关中断实现、任务切换、中断上下文切换等功能。

1.3.2、命名规范

必须使用业界通用CPU架构名，从名字可以清晰了解是哪种CPU、哪种processs系列。

1.3.3、目录结构规范

新增CPU架构规范，以ARM体系为例。

三级和四级目录按照具体情况可选。对于三级目录，如果此架构只会使用gcc，则不需要分此目录；如果二级目录可以区分不同的处理器系列或架构类型，则按照具体情况不需要添加。

一级目录 CPU arch	二级目录 Process arch	三级目录(具体情况可选) Compiler Type	四级目录(具体情况可选) Process series
ARM	armv5	armcc/gcc/iccarm	
	armv6m	armcc/gcc/iccarm	m0
	armv7m	armcc/gcc/iccarm	m3
			m4
			m7
	armv7a	armcc/gcc/iccarm	a5
			a7
			a9

1.3.4、函数命名规范

CPU arch统一对接下述接口

CPU Porting接口	说明
cpu_intrpt_save	关中断
cpu_intrpt_restore	开中断
cpu_intrpt_switch	中断退出切换（在中断处理函数尾部使用，需要确保被打断的上下文正确保存，中断退出后，回到当前最高优先级任务）
cpu_task_switch	任务切换（需要保存老任务上下文、获取最高优先级任务、恢复新任务上下文）
cpu_first_task_start	进入第一个任务调度
cpu_task_stack_init	任务栈初始化
cpu_cur_get	获取当前核号

1.3.5、mk编写规范（aos.mk）

没有例外情况，统一在二级Process arch目录添加对应的编译mk文件。

arch mk添加规范如下（以armv7m为例）：

```
NAME := arch_armv7m                #arch_+架构名
$(NAME)_MBINS_TYPE := kernel       #多bin情况下，归属kernel还是app
$(NAME)_VERSION  := 1.0.2          #menuconfig版本号
$(NAME)_SUMMARY  := arch for armv7m   #描述
$(NAME)_SOURCES    +=              #组件包含.c文件
GLOBAL_INCLUDES    +=              #包含头文件
ifeq ($(COMPILER),armcc)           #区分编译器
ifeq ($(HOST_ARCH),Cortex-M4)      #区分Process series
```

1.3.6、config.in编写规范

以armv7m为例：

```
config AOS_ARCH_ARMV7M       # 定义组件配置选项
  bool                # 配置选项类型
  help
   arch for armv7m        # 配置选项帮助
if AOS_ARCH_ARMV7M
# Configurations for arch armv7m # 如有必要，定义更多组件内配置选项
endif
```

Arch组件配置选项命名规范：使用前缀“AOS_ARCH_” + 组件NAME

1.4、mcu子目录新增规范

1.4.1、目录功能

Mcu目录存放其原始SDK驱动文件，以及hal驱动对接层。

其中的SDK文件原则上直接使用厂商的驱动包，除了License或bug修复等，原则上不做修改。板级相关的配置代码统一放入board目录。

1.4.2、命名规范

Mcu命令需要使用业界通用名，能直观方便检索到相关信息为准。

1.4.3、目录结构规范

```
Dir\File              Description                 Necessary for kernel run
|-- drivers          # board peripheral driver                  Y
|-- hal            # hal API layer, hal uart is necessary          Y
|-- aos.mk          # mcu makefile                          Y
|-- Config.in       # menuconfig component config                 Y
|-- README.md                                           Y
```

1.4.4、函数命名规范

统一按照样例aamcu_demo/hal下列出的各模块hal API实现。

1.4.5、mk编写规范（aos.mk）

mcu的mk文件，其描述了当前mcu组件需要的编译文件和编译选项。

如果该系列MCU能实现一个通用mk文件则使用一个即可；如果该MCU体系下存在多种MCU子系列，那么需要添加子mcu的mk文件，在其中放置不同的属性定义。aos.mk作为主mk，主要放置公共的属性配置，并使用HOST_MCU_NAME来分别引用对应的子mcu。

示例：

```
aamcu_demo                  #mcu主目录
  |-- aos.mk              # 该mcu主mk
  |-- aamcu1_demo.mk          # aamcu1_demo
  |-- aamcu2_demo.mk          # aamcu2_demo
```

在对应board如aaboard_demo的aos.mk文件引用此mcu模块名时，使用格式：

示例：

```
HOST_MCU_FAMILY  := mcu_aamcu_demo
HOST_MCU_NAME    := aamcu1_demo
```

在mcu的主aos.mk中需要分别对子mcu进行引用，使用格式：

```
include $($(NAME)_LOCATION)/$(HOST_MCU_NAME).mk
```

aos.mk其他包含项：

```
NAME := mcu_aamcu_demo          #主MCU名，一般是mcu_+当前mcu目录名
$(NAME)_MBINS_TYPE  := kernel     #多bin情况下，归属kernel还是app
$(NAME)_VERSION    := 1.0.2      #menuconfig组件版本号
$(NAME)_SUMMARY    := driver & sdk   #描述
$(NAME)_SOURCES    +=          #MCU组件包含.c文件
$(NAME)_COMPONENTS  +=          #依赖其他组件名
GLOBAL_INCLUDES    +=          #头文件
GLOBAL_CFLAGS     +=          #c文件编译选项
GLOBAL_ASMFLAGS    +=          #汇编编译选项
GLOBAL_LDFLAGS     +=          #链接选项
GLOBAL_DEFINES     +=          #用户自定义宏
```

如，需要指定该MCU的CPU类型，则如下定义，将对应CPU调度代码加入编译体系：

$(NAME)_COMPONENTS += arch_armv7m

同样的，在3.1版本中，mk建议只定义组件化的模块components,其他配置和选项需要在Config.in中定义。

1.4.6、config.in编写规范

以aamcu_demo为例：

```
config AOS_MCU_AAMCU_DEMO       # 定义组件配置选项
  bool                  # 配置选项类型
  select ...              # 依赖其他组件
  help
   driver & sdk for platform/mcu aamcu_demo    # 配置选项帮助
if AOS_MCU_AAMCU_DEMO
# Configurations for mcu aamcu_demo  # 如有必要，定义更多组件内配置选项
endif
```

mcu组件配置选项命名规范：使用前缀“AOS_MCU_” + 组件NAME.

2、接口定义使用规范

2.1、内核接口使用规范

规范：对于纯内核系统，或者内核本身、bsp相关代码使用krhino接口；对于上层连接协议栈、app统一使用aos接口。

2.2、HAL定义规范

规范：hal相关接口的命名和声明统一参照目录aamcu_demo/hal下提供的样例demo实现。

特殊说明：

Flash相关hal接口实现，对OTA功能有影响，包括不限于：

相关hal函数返回值需要规范，统一为正确返回0，错误返回负值；

相关含有出参的接口，如off_set需要正确赋值；

具体细节以OTA相关实现要求为准。

3、初始化流程规范

系统从复位启动到main函数入口的流程一般使用该单板通用的汇编程序来实现，进入main函数后需要遵循下面规范。下面规范一方面为了使用接口统一，另一方面避免发生一些已知的流程问题。

main函数位置规范：

按照1.2

章节board目录结构描述，统一放在startup.c中实现，并统一跳转到app中的主任务入口aos_maintask，由其跳转到app入口application_start。

单板驱动初始化规范：

在初始化流程中，对于板级的初始化，按照模块功能划分，统一了初始化的接口实现，参考aaboard_demo/config/board_api.h，具体实现在board.c中。

主要包括：

```
void board_basic_init(void)      # 基础的堆、时钟初始化
void board_stduart_init(void)    # 标准输出
void board_tick_init(void)
void board_flash_init(void)
void board_network_init(void)
void board_gpio_init(void)
void board_wdg_init(void)
void board_ota_init(void)
void board_dma_init(void)
```

用户需要按照具体运行的app需求，来实现对应的模块初始化代码；并且在app主任务中将上述接口封装在board_init里面统一调用，不同的app中board_init封装的board模块不同。

对于board_basic_init函数，需要在进入main函数内，内核启动前调用；

其他的模块初始化按需封装在board_init内，在主函数入口aos_maintask调用。

board_basic_init此接口调用时，内核尚未初始化，此初始化阶段不能激活中断处理，否则会触发中断调度。

内核初始化调用：

统一使用krhino接口，如krhino_init和krhino_start。

主任务创建规范：

内核初始化本身只会创建内部任务，如idle/timer任务；初始化流程中需要创建主任务，供用户app运行。统一通过krhino接口如krhino_task_dyn_create来创建主任务；主任务的入口统一为**aos_maintask**。

接口使用限制说明：

krhino_init前不调用malloc、printf函数。原因是此类库函数被内核重定向，会调用内核接口aos_malloc，依赖内核的初始化。

系统初始化流程规范如下：

主任务会在krhino_start开始调度后进入，如果不创建主任务，则系统会默认进入OS自身创建的其他任务运行，比如idle任务。

主任务入口aos_maintask实现：

在aos_maintask中，统一调用board_init实现板级初始化，内部按照需求来添加board的具体组件；

如果需要初始化相关中间件和协议栈模块，使用aos_components_init接口；

最后，在非多bin的情况下，统一调用application_start进入上层app入口；多bin情况下，由aos_components_init内部分发处理。

（1）、系统初始化示例：

参考代码（platform/board/aaboard_demo/startup/startup.c）：

```
int main(void)
{
  /*irq initialized is approved here.But irq triggering is forbidden, which will enter CPU scheduling.
  Put them in sys_init which will be called after aos_start.
  Irq for task schedule should be enabled here, such as PendSV for cortex-M4.
  */
  /* board basic init: Base CLK, heap, define in board\aaboard_demo\startup\board.c */
  board_basic_init();
  /* kernel init, malloc can use after this! */
  krhino_init();
  /* main task to run */
  krhino_task_dyn_create(&g_main_task, "main_task", 0, AOS_MAIN_TASK_PRI, 0,
                AOS_MAIN_TASK_STACK_SIZE, (task_entry_t)aos_maintask, 1);
  /* kernel start schedule! */
  krhino_start();
  /* never run here */
  return 0;
}
```

（2）、主任务初始化示例：

参考代码（application/example/helloworld_demo/maintask.c）：

```
void board_init(void)
{
  board_tick_init();
  board_stduart_init();
  board_dma_init();
  board_gpio_init();
}
void aos_maintask(void* arg)
{
  board_init();
  /*内核参数初始化，一般使用默认*/
  board_kinit_init(&kinit);
  /*中间件模块初始化，按宏开关*/
  aos_components_init(&kinit);
  /*跳转到app用户入口*/
#ifndef AOS_BINS
  application_start(kinit.argc, kinit.argv); /* jump to app entry */
#endif
}
```

（3）、用户app入口示例：

参考代码（application/example/helloworld_demo/appdemo.c）：

```
int application_start(int argc, char *argv[])
{
  int count = 0;
  printf("nano entry here!\r\n");
  while(1) {
    printf("hello world! count %d \r\n", count++);
    aos_msleep(1000);
  };
}
```

4、内核认证

AliOS

Things提供了基本的内核测试用例集，用于内核移植后的测试验证，所有移植的平台都需要运行该测试样例，确保内核功能的正确性。

内核测试集目录：test/testcase/certificate/certificate_test

在上面目录下提供了两个测试文件rhino_test.c和aos_test.c。其中rhino_test.c针对于纯内核的移植，aos_test.c针对于至少包含AOS API层的移植。

目前主要的认证项都会通过aos层，如果只关注rhino_test.c相关纯内核的验证，需要做以下修改：

- 修改rhino_test.c配置项，如：

```
/*以下字符定义可任取名字，不能为空*/
#define SYSINFO_ARCH     "unknown"
#define SYSINFO_MCU      "unknown"
#define SYSINFO_DEVICE_NAME "unknown"
#define SYSINFO_APP_VERSION "3.1.0"
/*kv不属于纯krhino模块，需要关闭*/
#define TEST_CONFIG_KV_ENABLED          (0)
```

- 将rhino_test.c和cut.c/cut.h加入编译体系

可以将test/testcase/certificate/certificate_test目录下此三个直接拷贝到对应mcu下，新建一个test目录并加入到makefile；其他IDE直接添加编译文件。

- 在主任务中调用test_certificate执行测试用例认证直到用例通过即可。

上面属于纯krhino内核的测试方式，如果带aos接口层的测试请参考，

AliOS Things Kernel

测试指南：https://github.com/alibaba/AliOS-Things/wiki/Manual-API

5、代码合入整体原则

5.1、公共代码修改

公共代码原则上避免修改，以影响其他单板。通用文件修改后，需要确认不影响其他工程的编译和运行。如果影响公共代码，需要清晰说明：是修复bug、增加新特性、或是改进功能，并介绍如何完成的。

公共代码范围：目前除新增board目录、新增mcu目录，其他目录或者文件都视为公共文件，包括app/example目录。修改后，都可能影响其他单板。

5.2、编译链接选项限制

在编译选项中，严禁加-w选项关闭编译告警，以控制上传代码质量。

5.3、License准则

- 原创为主尊重版权，请作者标明自己的 copyright 信息，并签署CLA
- 禁止合入这些 license 的代码: AGPL, CPAL, OSL 等严格开源许可证
- 谨慎处理 GPL/LGPL 等强制开源许可证的代码，考虑以下替代方案

（1）将 GPL/LGPL 或类似许可证源码编译为二进制程序，作为独立软件使用

（2）将 LGPL 或类似许可证源码编译为动态连接库，并以动态连接方式调用

（3）将 LGPL

或类似许可证源码编译为静态链接库与应用程序相结合发布，但同时提供整个应用程序(含

LGPL 静态连接库)的目标代码和 LGPL 库源码

- 允许使用的 license: BSD, MIT, Apache License Version 2.0, Zlib, CDDL 等宽松开源许可证

5.4、CI验证通过

- Build: 代码 autobuild, PV build 通过
- Test: PV 测试通过(目前暂无此步骤)
- Agreement: CLA 已签署

4.2.2. 芯片架构移植

芯片架构的相关代码位于 `platform/arch` 中。

arch目录下已经实现了基本通用的CPU架构的porting，如果新增单板的CPU架构在此列表中，则跳过本章节进入新增MCU描述。

下面是支持的arch列表：

ARM	ARM9

	Cortex-M0/M0+
	Cortex-M3
	Cortex-M4
	Cortex-M7
	Cortex-A5
	Cortex-A7
	Cortex-A9
Xtensa	lx6
	lx106
C-SKY	cskyv2-l
Renesas	rl78
	rx600
MIPS	mips32
	mips-I
Linux	
RISC-V	risc_v32I

2、新增CPU适配点

对于系统中已经支持的CPU架构，可以直接使用对应的platform/arch模块，如果需要新增CPU架构支持，需要适配下面几个接口，其对所有CPU架构通用：

cpu_intrpt_save	关中断
cpu_intrpt_restore	开中断
cpu_intrpt_switch	中断退出切换（在中断处理函数尾部使用，需要确保被打断的上下文正确保存，中断退出后，回到当前最高优先级任务）
cpu_task_switch	任务切换（需要保存老任务上下文、获取最高优先级任务、恢复新任务上下文）
cpu_first_task_start	进入第一个任务调度
cpu_task_stack_init	任务栈初始化
cpu_cur_get	获取当前核号

涉及到新CPU架构移植，可联系相关支持人员。

3、CPU arch目录规范

规范：新增CPU架构规范，以ARM体系为例

ARM	armv5	armcc/gcc/iccarm	
	armv6m	armcc/gcc/iccarm	m0
	armv7m	armcc/gcc/iccarm	m3
			m4
			m7
	armv7a	armcc/gcc/iccarm	a5
			a7
			a9

三级和四级目录按照具体情况可选。对于三级目录，如果此架构的演化只会使用一套编译体系即gcc，则不需要分此目录；如果二级目录可以区分不同的处理器系列或架构类型，则按照具体情况不需要添加。

规范：arch需要包含下面几个文件

k_types.h	基本数据类型定义，无特殊情况可沿用
port.h	CPU接口头文件
port_c.c	CPU适配.c文件，主要是cpu_task_stack_init适配
port_s.S	CPU适配.s文件，相关汇编实现

章节2.2.2列出的CPU的适配接口按照实际情况在port_c.c 或者port_s.S中实现。其他CPU架构相关的文件需要放在arch目录下，可以按照实际需求安放。

3.1、arch mk文件编写

规范： 没有例外情况，统一在二级Process arch目录添加对应的编译mk文件。

arch mk添加规范如下（以armv7m为例）：

```
NAME := arch_armv7m                #arch_+架构名
$(NAME)_MBINS_TYPE := kernel       #多bin情况下，归属kernel还是app
$(NAME)_VERSION  := 1.0.2          #menuconfig版本号
$(NAME)_SUMMARY  := arch for armv7m  #描述
$(NAME)_SOURCES    +=              #组件包含.c文件
GLOBAL_INCLUDES    +=              #包含头文件
ifeq ($(COMPILER),armcc)           #区分编译器
ifeq ($(HOST_ARCH),Cortex-M4)      #区分Process series
```

3.2、config.in文件编写

arch Config.in添加规范如下（以armv7m为例）：

```
config AOS_ARCH_ARMV7M       # 定义组件配置选项
  bool                # 配置选项类型
  help
    arch for armv7m         # 配置选项帮助
if AOS_ARCH_ARMV7M
# Configurations for arch armv7m # 如有必要，定义更多组件内配置选项
endif
```

Arch组件配置选项命名规范：使用前缀“AOS_ARCH_” + 组件NAME。 原则上在3.1版本中，mk只用来增加定义的组件components，其他配置和宏定义统一在Config.in定义。具体参考组件化指导文档。

4.2.3. 板级移植指导

本文主要介绍如何将AliOS Things内核移植到一个新开发板上。主要内容包括目录结构介绍、内核移植说明、hal抽象层移植点。通过本文的介绍可以完成内核基本功能、内核功能认证相关的工作。本文不包含具体中间件和协议栈适配相关内容。

AliOS Things参考版本：AOS-R-3.1.0

参考示例单板：

board: aaboard_demo;

mcu: aamcu_demo;

arch: Cortex-M4

说明：

如果使用其他版本，请参考：AliOS Things内核驱动移植指导

和历史版本差异，主要包括：

a、 原根目录board目录归纳到platform下，对应platform/board目录；

b、 board下面重新梳理了几个典型单板的初始化代码样例，部分board及其hal的抽象定义和实现，主要将主任务的入口移动到app里面实现，由app来决定board的初始化内容，而非默认在board内全初始化；相关初始化代码和流程与对应的app关联修改，原有board实现放在board_legacy目录下；

c、 对应上述新规范board目录，相应的app实现在application/example目录下；原有的app实现移入application/example/example_legacy目录。

d、板级board初始化代码按照模块统一梳理。

1、基本介绍

1.1 目录结构介绍

下述为新增一个单板支持，必须关注的几个目录项：

目录名	介绍
app/example	通用用户运行实例，如helloworld实例，可直接使用，无特殊情况不修改
platform/board	用户需要适配、可配置board级代码，系统启动相关代码
platform/arch	该CPU架构内核调度适配接口，可直接使用
platform/mcu	该MCU通用SDK以及对应的hal适配层

注意：platform/arch下已经适配了目前主流的CPU架构。其他目录结构，如build目录存放了通用的编译体系，用户一般情况下不需要修改；kernel目录下为内核代码，对于移植来说不需要修改。

1.2 编译环境相关介绍

1.2.1 编译环境安装

使用AliOS Things的编译体系需要安装python和aos cube插件。

环境搭建参考github链接：https://github.com/alibaba/AliOS-Things/wiki/AliOS-Things-Environment-Setup

上述链接详细描述了不同的PC环境下，编译环境的搭建指导。

1.2.2 编译命令

构建步骤

AliOS Things 2.1以后引入了全新的,基于menuconfig的配置系统，构建过程更新为“先配置，再构建”两个步骤：

```
通过图形界面配置，然后构建
$ aos make menuconfig
$ aos make
配置时指定app、board，先生成最简配置，然后构建
$ aos make appname@boardname -c config && aos make
清除构建目录，不清除配置
$ aos make clean
清除构建目录和配置
$ aos make distclean
```

AliOS Things 3.1 版本后提供了新的构建编译方式，并且兼容之前版本的编译方式，命令形式如下，具体参看编译指导文档：

```
$ aos create project -b pca10040 -d ../blemesh_app -t blemesh_tmall test_blemesh_tmall
$ cd ../blemesh_app/blemesh_tmall
$ aos make
```

1.2.3 编译生成文件

编译后，会自动在主目录下生成out目录，其中主要关注下面几个文件。

生成文件	介绍
config.mk	编译工程所有的配置选项
binary/helloworld@aaboard_demo.bin	可执行bin文件
binary/helloworld@aaboard_demo.elf	可执行elf文件
binary/helloworld@aaboard_demo.map	生成map文件

2、移植指导及规范

按照上述章节描述，对于移植工作主要涉及四个模块的内容，可以概述为移植四要素：CPU、MCU、board以及example。在移植中由于存在依赖关系，实际会依次按照这四个模块来进行适配。

移植一个新单板，需要先考虑其属于哪种CPU架构，来适配对应的CPU调度接口；第二步添加该MCU通用的设备驱动；然后在board模块下添加相应的板级、外设驱动程序；最后按照需要编写该单板需要运行的example实例。

如果移植依赖的某个要素已经存在，则可直接使用，比如要新增一个stm32f429zi单板，其基于Cortex-M4的CPU架构，而Platform/arch已经有相关实现，则可直接使用。

以下章节按照移植顺序，说明新增一个单板涉及到的所有模块适配工作。

2.1新增CPU架构

涉及目录：platform/arch

2.1.1 已支持CPU架构

arch目录下已经实现了基本通用的CPU架构的porting，如果新增单板的CPU架构在此列表中，则跳过本章节进入新增MCU描述。

下面是支持的arch列表：

CPU Arch	Processor series
ARM	ARM9
	Cortex-M0/M0+
	Cortex-M3
	Cortex-M4
	Cortex-M7
	Cortex-A5
	Cortex-A7
	Cortex-A9
Xtensa	lx6
	lx106
C-SKY	cskyv2-l
Renesas	rl78
	rx600
MIPS	mips32
	mips-I
Linux	
RISC-V	risc_v32I

2.1.2 新增CPU适配点

对于系统中已经支持的CPU架构，可以直接使用对应的platform/arch模块，如果需要新增CPU架构支持，需要适配下面几个接口，其对所有CPU架构通用：

CPU Porting接口	说明
cpu_intrpt_save	关中断
cpu_intrpt_restore	开中断

cpu_intrpt_switch	中断退出切换（在中断处理函数尾部使用，需要确保被打断的上下文正确保存，中断退出后，回到当前最高优先级任务）
cpu_task_switch	任务切换（需要保存老任务上下文、获取最高优先级任务、恢复新任务上下文）
cpu_first_task_start	进入第一个任务调度
cpu_task_stack_init	任务栈初始化
cpu_cur_get	获取当前核号

涉及到新CPU架构移植，可联系相关支持人员。

2.1.3 CPU arch目录规范

规范：新增CPU架构规范，以ARM体系为例

一级目录 CPU arch	二级目录 Process arch	三级目录(具体情况可选) Compiler Type	四级目录(具体情况可选) Process series
ARM	armv5	armcc/gcc/iccarm	
	armv6m	armcc/gcc/iccarm	m0
	armv7m	armcc/gcc/iccarm	m3
			m4
			m7
	armv7a	armcc/gcc/iccarm	a5
			a7
			a9

三级和四级目录按照具体情况可选。对于三级目录，如果此架构的演化只会使用一套编译体系即gcc，则不需要分此目录；如果二级目录可以区分不同的处理器系列或架构类型，则按照具体情况不需要添加。

规范：arch需要包含下面几个文件

arch文件	介绍
k_types.h	基本数据类型定义，无特殊情况可沿用
port.h	CPU接口头文件
port_c.c	CPU适配.c文件，主要是cpu_task_stack_init适配
port_s.S	CPU适配.s文件，相关汇编实现

章节2.2.2列出的CPU的适配接口按照实际情况在port_c.c

或者port_s.S中实现。其他CPU架构相关的文件需要放在arch目录下，可以按照实际需求安放。

2.1.4 arch mk文件编写

规范： 没有例外情况，统一在二级Process arch目录添加对应的编译mk文件。

arch mk添加规范如下（以armv7m为例）：

```
NAME := arch_armv7m                #arch_+架构名
$(NAME)_MBINS_TYPE := kernel       #多bin情况下，归属kernel还是app
$(NAME)_VERSION  := 1.0.2          #menuconfig版本号
$(NAME)_SUMMARY  := arch for armv7m  #描述
$(NAME)_SOURCES    +=              #组件包含.c文件
GLOBAL_INCLUDES    +=              #包含头文件
ifeq ($(COMPILER),armcc)           #区分编译器
ifeq ($(HOST_ARCH),Cortex-M4)      #区分Process series
```

2.1.5 config.in文件编写

arch Config.in添加规范如下（以armv7m为例）：

```
config AOS_ARCH_ARMV7M       # 定义组件配置选项
  bool                # 配置选项类型
  help
   arch for armv7m        # 配置选项帮助
if AOS_ARCH_ARMV7M
# Configurations for arch armv7m # 如有必要，定义更多组件内配置选项
endif
```

Arch组件配置选项命名规范：使用前缀“AOS_ARCH_” + 组件NAME。

原则上在3.1版本中，mk只用来增加定义的组件components，其他配置和宏定义统一在Config.in定义。具体参考组件化指导文档

2.2新增mcu

涉及目录：platform/mcu。

mcu目录存放其原始SDK驱动文件，以及hal驱动对接层。

外设驱动以及用户对于单板的配置代码不放入此目录，以便该SDK能支持该MCU下所有系列单板。

2.2.1 mcu目录规范

主要目录结构：

```
Dir\File                Description              Necessary for kernel run
|-- drivers          # board peripheral driver                Y
|-- hal              # hal API layer, hal uart is necessary         Y
|-- aos.mk           # mcu makefile                           Y
|-- Config.in        # menuconfig component config              Y
|-- ucube.py         # aos build system file(for scons)           N
|-- README.md                                                 Y
```

2.2.2 mcu mk文件编写

mcu的mk文件，其描述了当前mcu组件需要的编译文件和编译选项。

如果该系列MCU能实现一个通用mk文件则使用一个即可；如果该MCU体系下存在多种MCU子系列，那么需要添加子mcu的mk文件，在其中放置不同的属性定义。aos.mk作为主mk，主要放置公共的属性配置，并使用HOST_MCU_NAME来分别引用对应的子mcu。不同的mcu子系列主要是由于其链接的驱动文件或者编译选项等不同，需要通过不同的mk来区分实现。

示例：

```
aamcu_demo                  #mcu主目录
  |-- aos.mk              # 该mcu主mk
  |-- aamcu1_demo.mk          # aamcu1_demo
  |-- aamcu2_demo.mk          # aamcu2_demo
```

在对应board如aaboard_demo的aos.mk文件引用此mcu模块名时，使用格式：

示例：

```
HOST_MCU_FAMILY  := mcu_aamcu_demo
HOST_MCU_NAME    := aamcu1_demo
```

在mcu的主aos.mk中需要分别对子mcu进行引用，使用格式：

```
include $($(NAME)_LOCATION)/$(HOST_MCU_NAME).mk
```

aos.mk其他必须包含项：

```
NAME := mcu_aamcu_demo          #主MCU名，一般是mcu_+当前mcu目录名
$(NAME)_MBINS_TYPE  := kernel     #多bin情况下，归属kernel还是app
$(NAME)_VERSION    := 1.0.2      #menuconfig组件版本号
$(NAME)_SUMMARY    := driver & sdk   #描述
$(NAME)_SOURCES    +=          #MCU组件包含.c文件
$(NAME)_COMPONENTS  +=          #依赖其他组件名
GLOBAL_INCLUDES    +=          #头文件
GLOBAL_CFLAGS     +=          #c文件编译选项
GLOBAL_ASMFLAGS    +=          #汇编编译选项
GLOBAL_LDFLAGS     +=          #链接选项
GLOBAL_DEFINES     +=          #用户自定义宏
```

2.2.3 关联对应CPU

每个mcu都需要关联对应的CPU，通过在mk中增加引用cpu arch模块来进行关联，并最终通过在board下aos.mk定义的HOST_ARCH来确定具体型号。

例如对于Cortex-M4：

```
$(NAME)_COMPONENTS += arch_armv7m
```

2.2.4 config.in文件编写

mcu Config.in添加规范如下（以aamcu_demo为例）：

```
config AOS_MCU_AAMCU_DEMO       # 定义组件配置选项
  bool                  # 配置选项类型
  select AOS_ARCH_ARMV7M       # 依赖特定arch
  select ...              # 依赖其他组件
  help
   driver & sdk for platform/mcu aamcu_demo   # 配置选项帮助
if AOS_MCU_AAMCU_DEMO
# Configurations for mcu aamcu_demo # 如有必要，定义更多组件内配置选项
endif
```

mcu组件配置选项命名规范：使用前缀“AOS_MCU_” + 组件NAME。

同样，原则上在3.1版本中，mk只用来增加定义的组件components，其他配置和宏定义统一在Config.in定义。具体参考组件化指导文档。

2.3新增board

涉及目录：board

board中主要存放外设驱动，板级初始化、以及用户对该board的驱动适配文件。

其中platform/board/board_legacy下为3.1版本之前规范的board示例，其对应app/example/example_legacy下app实现；在跳转到app入口application_start前完成了board的全部初始化；

platform/board下非board_legacy目录为3.1版本要求的规范board示例，其对应app/example目录下非example_legacy实现；其进入main函数后，完成最基本的初始化，并创建主任务后，直接跳转到app内主任务入口aos_maintask，并最终跳转到application_start入口。

platform/board/board_legacy原有单板只能支持老的example_legacy实现；新的platform/board单板，兼容app/example下所有app。

2.3.1 board目录规范

board取名需要使用官方通用名，能方便检索到相关信息

board目录下文件结构部署和命名需要遵循下面布局规则，以aaboard_demo单板为例：

```
Dir\File                Description                          Necessary for kernel run
|-- drivers             # board peripheral driver                        N
|-- config
|  |-- board.h          # board config file, define for user, such as uart port num     Y
|  |-- k_config.c       # user's kernel hook and mm memory region define            Y
|  |-- k_config.h       # kernel config file .h                          Y
|  |-- partition_conf.c    # board flash config file                          N
|-- startup
|  |-- board.c          # board_init implement                           Y
|  |-- startup.c        # main entry file                          Y
|  |-- startup_gcc.s       # board startup assember for gcc                    Y
|  |-- startup_iar.s       # board startup assember for iar                    Y
|  |-- startup_keil.s      # board startup assember for keil                   Y
|-- aaboard_demo.icf       # linkscript file for iar                        Y
|-- aaboard_demo.ld        # linkscript file for gcc                        Y
|-- aaboard_demo.sct       # linkscript file for sct                        Y
|-- aos.mk              # board makefile                             Y
|-- Config.in           # menuconfig component config                       Y
|-- ucube.py            # aos build system file(for scons)                  N
|-- README.md                                                  Y
```

board相关初始化使用的函数名需规范统一，参照如下：

文件	函数名
k_config.c	实现样例单板aaboard_demo该文件内所有对接接口
partition_conf.c	统一分区初始化接口：flash_partition_init
board.c	统一单板初始化接口：board_init
startup.c	无特殊情况统一C程序主入口为main； 内部调用单板初始化board_init；内部调用krhino接口初始化内核； 内部创建主任务入口sys_init。（具体见初始化流程规范）

2.3.2 board mk文件编写

以下列出mk文件中需要修改的关键点：

```
NAME := board_aaboard_demo          #board_+单板名
$(NAME)_MBINS_TYPE := kernel        #在多bin情况下，归属kernel还是app
$(NAME)_VERSION  := 1.0.1           #组件版本号
$(NAME)_SUMMARY  :=                 #描述
MODULE        := 1062               #固定
HOST_ARCH     := Cortex-M4          #CPU arch
HOST_MCU_FAMILY  := mcu_aamcu_demo  #归属MCU系列，需要对应platform/mcu下aos.mk组件
SUPPORT_MBINS    := no              #是否支持app\kernel的bin分离
HOST_MCU_NAME    := aamcu1_demo     #MCU子系列类型
ENABLE_VFP       := 1               #是否支持浮点数
$(NAME)_SOURCES    +=               #board组件包含.c文件
$(NAME)_COMPONENTS  +=              #依赖其他组件名
GLOBAL_INCLUDES    +=               #头文件
GLOBAL_CFLAGS      +=               #c文件编译选项
GLOBAL_ASMFLAGS    +=               #汇编编译选项
GLOBAL_LDFLAGS     +=               #链接选项
GLOBAL_DEFINES     +=               #用户自定义宏
注意：
（1）、其中HOST_MCU_FAMILY的定义需要对应platform/mcu某子目录下aos.mk中的组件名NAME，一般是mcu_+“mcu名”。HOST_MCU_NAME表示具体的mcu子系列。
（2）、用户可以通过GLOBAL_DEFINES定义宏，如GLOBAL_DEFINES += CONFIG_AOS_CLI_BOARD或者GLOBAL_DEFINES += CONFIG_AOS_KV_BLK_BITS=14。当然也可以直接在编译选项 GLOBAL_CFLAGS使用-D定义。
（3）、原则上在3.1版本中，mk只用来增加定义的组件components，其他配置和宏定义统一在Config.in定义。具体参考组件化指导文档
```

2.3.3 关联对应MCU

每个board需要关联其从属的MCU，通过在其mk中添加HOST_MCU_FAMILY定义来指定。同时，如果存在子MCU，则还需要设置具体的HOST_MCU_NAME。

例如对于aaboard_demo单板，其要关联MCU是aamcu_demo系列下的aamcu1_demo，在aaboard_demo目录下的aos.mk设置如下：

```
HOST_MCU_FAMILY  := mcu_aamcu_demo
HOST_MCU_NAME    := aamcu1_demo
```

2.3.4 config.in文件编写

board Config.in添加规范如下（以aaboard_demo为例）：

```
config AOS_BOARD_AABOARD_DEMO   # 定义组件配置选项
  bool "AABOARD_DEMO"      # 配置选项类型，双引号定义该选项显示名称
  select AOS_MCU_AAMCU_DEMO   # 依赖特定mcu
  select ...            # 依赖其他组件
  help
   ...              # 配置选项帮助
if AOS_BOARD_AABOARD_DEMO
# Configurations for board aaboard_demo
# "BSP SUPPORT FEATURE"      # 硬件支持的能力
config BSP_SUPPORT_UART
  bool
  default y
...
endif
```

board组件配置选项命名规范：使用前缀“AOS_BOARD_” + 组件NAME

2.4新增example

涉及目录：app/example

example目录主要存放用户实际需要运行的程序，包括主任务入口aos_maintask，以及用户app统一入口application_start。

原则上不建议新增example，除非目前的example不能满足功能需求。app/example下为通用运行实例，如果新增example，需要具有通用性,而不是为了某个特殊，或者临时性的修改。

其中app/example/example_legacy下为3.1版本之前规范的app示例，其不包括maintask.c主任务实现，其相关主任务的板级实现统一在跳转app前做全初始化，即不同的app，底层初始化的模块是一样的；

app/example目录非example_legacy下为3.1版本要求的规范app示例，其包括maintask.c主任务实现，对板级初始化做了模块区分，以期望实现不同app按照具体需求来初始化不同的board模块。

其中，app/example/example_legacy对应platform/board/board_legacy目录的单板；

其他app/example对应platform/board下非board_legacy实现。

另，platform/board/board_legacy原有单板只能支持老的example_legacy实现；新的platform/board单板，兼容app/example下所有app。

2.4.1 example目录规范

以helloworld目录为例：

```
helloworld_demo
|-- maintask.c      # main task entry "aos_maintask"
|-- appdemo.c       # helloworld source code, including app entry ” application_start”
|-- Config.in       # menuconfig config file
|-- aos.mk          # aos build system file(for make)
|-- k_app_config.h    # aos app config file, has higher priority than k_config.h
|-- ucube.py        # aos build system file(for scons)
|-- README.md
```

2.4.2 example mk文件编写

```
NAME := helloworld_demo            #example名，和目录统一
$(NAME)_MBINS_TYPE := app          #在多bin情况下，归属kernel还是app
$(NAME)_VERSION := 1.0.0           #menuconfig组件版本号
$(NAME)_SUMMARY := Hello World     #描述
$(NAME)_SOURCES     +=             #example.c文件
$(NAME)_COMPONENTS  +=             #依赖其他组件名
GLOBAL_INCLUDES     +=             #全局头文件
GLOBAL_DEFINES      +=             #全局宏定义
```

2.4.3 config.in文件编写

example Config.in文件编写规范（以helloworld为例）：

```
config AOS_APP_HELLOWORLD_DEMO       # 定义组件配置选项
  bool "Helloworld Demo"             # 配置选项类型，双引号定义该选项显示名称
  select AOS_COMP_OSAL_AOS           # 依赖其他组件
  help
    Hello World                      # 配置选项帮助
if AOS_APP_HELLOWORLD
# Configurations for app helloworld_demo     # 如有必要，定义更多组件内配置选项
endif
```

组件配置选项命名规范：使用前缀“AOS_APP_” + 组件NAME

将新增example加入系统配置菜单：

如果新增example在“app/exampe”目录下，编辑“app/exampe/Config.in”

如果新增example在“app/profile”目录下，编辑“app/profile/Config.in”

例如：

```
source "app/example/helloworld_demo/Config.in"  # 引用example配置文件
if AOS_APP_HELLOWORLD_DEMO                      # 如果example组件被启用
  config AOS_BUILD_APP
    default "helloworld_demo"          # 为AOS_BUILD_APP赋值，与example目录一致
endif
```

注意：AOS_BUILD_APP默认值必须与配置命令行（aos make app@board -cconfig）输入的app保持一致.

3、代码适配修改点说明

上一章节按照目录结构来说明适配新单板关注的目录结构，此章节针对特定移植点来说明具体关键特性的实现。对于一项移植工作，建立完相应的目录结构，并完成或者核对相关适配点的修改后，才能完成适配工作。此章节还可以对适配工作后的核对工作提供参考。

以下具体列出新单板适配的关键特性点。

3.1 CPU arch

第一项关键移植特性点就是CPU的架构支持。对于系统中已经支持的CPU架构，可以直接使用对应的platform/arch模块，如果需要新增CPU架构支持，参考章节2.1。

3.2 系统tick

Tick相关的需要有两处修改：

3.2.1 tick中断挂接

在tick中断处理接口内部需要调用krhino_tick_proc，并且在处理前后需要加入krhino_intrpt_enter和krhino_intrpt_exit。krhino_intrpt_exit中会使用cpu_intrpt_switch发起新的任务调度。

样例：

```
krhino_intrpt_enter();
krhino_tick_proc();
krhino_intrpt_exit();
```

修改位置：

参考board/aaboard_demo/startup/board.c中SysTick_Handler实现。

可按照实际情况在对应驱动代码中直接修改。

3.2.2 tick频率配置

需要将tick中断的频率配置给相应的寄存器。AliOS Things

在k_config.h中有相关RHINO_CONFIG_TICKS_PER_SECOND的设定。

修改位置：

参考board/aaboard_demo/startup/board.c中board_tick_init的实现，通过RHINO_CONFIG_TICKS_PER_SECOND配置时钟频率。

可按照实际情况对应驱动代码中修改。

3.3 基本串口打印

系统需要支持基本的串口打印功能，库函数_write_r已经对接到hal_uart_send接口，因此对接相应的hal接口即可。

修改位置：

参考platform/mcu/aamcu_demo/hal/hal_uart.c

3.4 内核可配置项（k_config.h）

k_config.h文件中包含了所有内核裁剪配置，包括模块裁剪、内存裁剪。可以根据不同的模块需求，以及内存大小来进行修改裁剪。k_config.h统一规范放在对应board的config目录下，参考：board/aaboard_demo/config/k_config.h

3.4.1 内核模块裁剪

k_config.h中定义了一系列内核相关的宏定义，主要包括模块的使能、栈的大小以及任务的优先级等等的配置。

对于内核模块的裁剪来说，需要运行的上层app直接影响了具体哪些内核模块需要打开。有一个简单有效的方式来选取合适的k_config.h作为基础参考版：

每个board的目录下，都放有一个ucube.py文件，内部通过linux_only_targets定义了该单板支持的运行实例，如helloworld、udataapp、linkkitapp。如果新增单板需要支持的运行实例和已有单板类似，则可直接拷贝其k_config.h过来作为基础版本。

3.4.2 内核内存裁剪

对于内存裁剪，不同的CPU由于需要保存的栈上下文有区别，所以在不同的平台上会有区别。基本考虑点是任务的上下文大小，任务内部的处理需要的大致栈大小。这里给出参考的较小任务栈配置，用户需要按照自身资源的情况来调整。

```
RHINO_CONFIG_TIMER_TASK_STACK_SIZE 128
RHINO_CONFIG_K_DYN_TASK_STACK    128
RHINO_CONFIG_IDLE_TASK_STACK_SIZE  100
RHINO_CONFIG_CPU_USAGE_TASK_STACK  100
需要运行上层协议栈时，
打开RHINO_CONFIG_WORKQUEUE项配置栈大小，内核运行时，此项不需要打开：
#define RHINO_CONFIG_WORKQUEUE       1
#define RHINO_CONFIG_WORKQUEUE_STACK_SIZE 512
```

可以使用krhino_task_stack_min_free接口来获取某任务的空闲栈大小。如果系统支持了cli，可以使用tasklist命令来输出所有的任务栈信息。下图

“MinFreesize”，

表示该任务运行到目前为止未使用的栈空间，单位都是cpu_stack_t（4字节）。

3.5 内核堆配置（k_config.c）

k_config.c

中定义g_mm_region结构体来内核的堆空间。内核初始化时，会自动调用该内存空间。

如果要使用内存申请功能，则需要打开RHINO_CONFIG_MM_TLF宏，来使能k_mm模块，并且配置对应的堆空间。

堆空间定义有三种方式：链接脚本定义、汇编定义、数组定义。**推荐方式：链接脚本定义**。

其基本原则是要预留一个内存空间作为堆使用，并将其交给g_mm_region管理。

参考文件board/aaboard_demo/config/k_config.c关于堆空间的说明。

3.5.1 链接脚本定义（建议方式）

链接脚本中定义堆空间：

```
PROVIDE (heap_start = __stack);          //end of stack
__heap_limit = ORIGIN(RAM) + LENGTH(RAM);
PROVIDE (heap_end = __heap_limit);
PROVIDE (heap_len = heap_end - heap_start);
```

堆的起点heap_start定义为栈的结尾，堆的结尾heap_end定义为RAM的结尾，这样剩余RAM的空间都交给OS管理。

对应的krhino的堆空间初始化为：

```
k_mm_region_t g_mm_region[] = {{(uint8_t *) &heap_start, (size_t) &heap_len}};
```

注意：这段内存分配给堆使用，并不是表示内存都耗尽了，而是将其交给OS管理，用户通过malloc出来的内存都是从其中申请。

3.5.2 汇编定义

汇编中定义堆空间：

```
heap_len    EQU   0x200
AREA   HEAP, NOINIT, READWRITE, ALIGN=3
heap_start
Heap_Mem     SPACE  heap_len
heap_end
```

此方式并没有将剩余RAM的空间都直接交给OS管理，需要用户自己来调整大小。

对应的krhino的堆空间初始化为：

```
k_mm_region_t g_mm_region[] = {{(uint8_t *) &heap_start, (size_t) & heap_len }};
```

3.5.3 数组定义

直接定义一个数组：

```
#define HEAP_BUFFER_SIZE 1024*30
uint8_t g_heap_buf[HEAP_BUFFER_SIZE];
```

此方式也没有将剩余RAM的空间都直接交给OS管理，需要用户自己来调整大小。

对应的krhino的堆空间初始化为：

```
k_mm_region_t g_mm_region[] = {{g_heap_buf, HEAP_BUFFER_SIZE}};
```

3.6 系统初始化

3.6.1 初始化相关规范流程

系统从复位启动到main函数入口的流程使用该board通用的启动汇编来实现，此处关注进入main函数内的系统启动流程。

主要包括：基础堆、时钟初始化，内核模块初始化krhino_init，创建主任务krhino_task_dyn_create，建立用户app入口。

，内核启动krhino_start。

主任务会在krhino_start开始调度后进入，如果不创建主任务，则系统会默认进入OS自身创建的其他任务运行，比如idle任务。

注意事项：

(1)、内核启动之前首先调用board_basic_init，初始化系统运行必要的heap堆和必要的时钟；**由于此接口调用时，内核尚未初始化，此初始化阶段不能激活中断处理，否则会触发中断调度；**

(2)、内核初始化统一使用krhino接口，如krhino_init和krhino_start；

(3)、krhino_init前不调用malloc、printf函数。原因是此类库函数被内核重定向，会调用内核接口aos_malloc，依赖内核的初始化；

(4)、内核初始化本身只会创建内部任务，如idle/timer任务；初始化流程中需要创建主任务，供用户app运行。统一通过krhino接口如krhino_task_dyn_create来创建主任务；主任务的入口统一为**aos_maintask;**

(5)、在aos_maintask中，统一调用board_init实现板级初始化，内部按照需求来添加board的具体组件；如果需要初始化相关中间件和协议栈模块，使用aos_components_init接口；最后，在非多bin的情况下，统一调用application_start进入上层app入口；多bin情况下，由aos_components_init内部分发处理。

3.6.2 系统初始化示例

（1）、系统初始化示例：

参考代码（platform/board/aaboard_demo/startup/startup.c）：

```
int main(void)
{
  /*irq initialized is approved here.But irq triggering is forbidden, which will enter CPU scheduling.
  Put them in sys_init which will be called after aos_start.
  Irq for task schedule should be enabled here, such as PendSV for cortex-M4.
  */
  /* board basic init: Base CLK, heap, define in board\aaboard_demo\startup\board.c */
  board_basic_init();
  /* kernel init, malloc can use after this! */
  krhino_init();
  /* main task to run */
  krhino_task_dyn_create(&g_main_task, "main_task", 0, AOS_MAIN_TASK_PRI, 0,
                AOS_MAIN_TASK_STACK_SIZE, (task_entry_t)aos_maintask, 1);
  /* kernel start schedule! */
  krhino_start();
  /* never run here */
  return 0;
}
```

（2）、主任务初始化示例：

参考代码（application/example/helloworld_demo/maintask.c）：

```
void board_init(void)
{
  board_tick_init();
  board_stduart_init();
  board_dma_init();
  board_gpio_init();
}
void aos_maintask(void* arg)
{
  board_init();
  /*内核参数初始化，一般使用默认*/
  board_kinit_init(&kinit);
  /*中间件模块初始化，按宏开关*/
  aos_components_init(&kinit);
  /*跳转到app用户入口*/
#ifndef AOS_BINS
  application_start(kinit.argc, kinit.argv); /* jump to app entry */
#endif
}
```

（3）、用户app入口示例：

参考代码（application/example/helloworld_demo/appdemo.c）：

```
int application_start(int argc, char *argv[])
{
  int count = 0;
  printf("nano entry here!\r\n");
  while(1) {
    printf("hello world! count %d \r\n", count++);
    aos_msleep(1000);
  };
}
```

4、hal实现说明

4.1 目录结构

hal的具体实现统一放在platform/mcu目录下，参考platform/mcu/aamcu_demo/hal下命令方式；

hal的对外头文件统一在include/aos/hal目录下，内部已经定义了目前所需要的hal对外接口；

代码中引用hal头文件，统一使用：

```
#include "aos/hal/*.h"
```

4.2 接口说明

hal相关接口说明以及具体参数描述不在本文讨论中，具体参考下面文中章节**硬件抽象函数**：

https://help.aliyun.com

4.3 pin脚映射（推荐不强制）

4.3.1 GPIO、LED、KEY映射

统一定义HAL_GPIO_**来定义gpio口，包含group和group内偏移定义，如

```
#define HAL_GPIO_16 ((uint8_t)16) /* represent GPIOB pin 0 */
```

其中该mcu每gpio group有16个pin脚，则HAL_GPIO_16代码groupB的pin0

定义完整的LED和KEY的管脚宏定义，如：

```
//LED
#define LED1  21  //或使用HAL_GPIO_21，表示GPIOB-PIN5
#define LED2  5   //或使用HAL_GPIO_5，表示GPIOA-PIN5
#define LED3  0xff //不支持统一使用0xff，访问做合法性判断
```

```
//KEY
#define KEY1  45  //或使用HAL_GPIO_45，表示GPIOC-PIN13
#define KEY2  67  //或使用HAL_GPIO_67，表示GPIOE-PIN3
#define KEY3  0xff //不支持统一使用0xff，访问做合法性判断
```

4.3.2 复合设备映射

对于uart、i2c、spi等多管脚多配置参数驱动设备，统一在相关Board实现中，采用下述管脚参数映射规则，便于统一管理和查找，描述以uart为例：

（1）、board.h中定义uart功能性逻辑抽象宏定义，

参考platform/board/aaboard_demo/config/board.h：

```
typedef enum{
  PORT_UART_STD = 0,
  PORT_UART_AT  = 1,
  PORT_UART_SIZE,
  PORT_UART_INVALID = 255,
  PORT_UART_DEMO = PORT_UART_AT,
}PORT_UART_TYPE;
```

如上，定义uart标准输出的逻辑port为PORT_UART_STD，AT通道的uart逻辑port为PORT_UART_AT。

后续通过hal接口访问某个设备，都通过逻辑port定义访问，而不是直接使用厂商的物理定义，以实现统一化。

（2）、在mcu/hal下头文件内定义UART的map结构体，指明逻辑设备和物理设备的**对应关系、私有参数、管脚定义**三个元素：

参考**platform/mcu/aamcu_demo/hal/include/hal_uart.h**：

```
typedef struct{
  PORT_UART_TYPE uartFuncP;       //uart logic port
  void*       uartPhyP;       //uart physical device
  uartAttribute  attr;          //uart private args
#if (HAL_VERSION >= 30100)        //used after 3.1 version，defined in board aos.mk
  gpio_uart_pin_config_t *pin_conf;  //pins define
  uint8_t           pin_cnt;   //pins used in uart
#endif
}UART_MAPPING;
```

其中HAL_VERSION >= 30100这个定义主要是为了兼容老版本；也可通过加入needmap魔术字来区分，避免老的单板未赋值相关结构体内容而访问非法数据，比如参考I2C_MAPPING；

attr表示内部私有特定参数，其中共有通用参数都已经在hal的对外接口中体现。

（3）、定义某Board的uart映射数组：

参考**platform/board/aaboard_demo/startup/board.c**：

```
static gpio_uart_pin_config_t usart1_pin_conf[] = {
  {UART_TX, HAL_GPIO_9},
  {UART_RX, HAL_GPIO_10}
};
UART_MAPPING UART_MAPPING_TABLE[] =
{
 {PORT_UART_STD, USART2, {UART_OVERSAMPLING_16, 64}, usart2_pin_conf,
  sizeof(usart2_pin_conf)/sizeof(usart2_pin_conf[0])},
 {PORT_UART_AT, USART1, {UART_OVERSAMPLING_16, 64}, usart1_pin_conf,
  sizeof(usart1_pin_conf)/sizeof(usart1_pin_conf[0])}
};
```

通过上述逻辑port对应的数据结构体，来具体调用相关的hal接口初始化赋值。

后续调用hal uart相关接口时，将**逻辑port**作为指示某uart设备的标识。如，系统中默认将PORT_UART_STD作为标准输出：

```
void board_stduart_init(void)
{
 uart_0.port = PORT_UART_STD;
 uart_0.config.baud_rate = STDIO_UART_BUADRATE;
 uart_0.config.data_width = DATA_WIDTH_8BIT;
 uart_0.config.flow_control = FLOW_CONTROL_DISABLED;
 uart_0.config.mode = MODE_TX_RX;
 uart_0.config.parity = NO_PARITY;
 uart_0.config.stop_bits = STOP_BITS_1;
 hal_uart_init(&uart_0);
}
```

特殊说明：

Flash相关hal接口实现，对OTA功能有影响，包括不限于：

相关hal函数返回值需要规范，统一为正确返回0，错误返回负值；

相关含有出参的接口，如off_set需要正确赋值；

具体细节以OTA相关实现要求为准。

5、内核测试认证指导

AliOS

Things提供了基本的内核测试用例集，用于内核移植后的测试验证，所有移植的平台都需要运行该测试样例，确保内核功能的正确性。

内核测试集目录：test/testcase/certificate_test

在上面目录下提供了两个测试文件rhino_test.c和aos_test.c。其中rhino_test.c针对于纯内核rhino的测试，aos_test.c针对于至少包含AOS API层的移植。

目前主要的认证项都会通过aos层，如果只关注rhino_test.c相关纯内核的验证，需要做以下修改：

- 修改rhino_test.c配置项，如：

```
/*以下字符定义可任取名字，不能为空*/
#define SYSINFO_ARCH     "unknown"
#define SYSINFO_MCU      "unknown"
#define SYSINFO_DEVICE_NAME "unknown"
#define SYSINFO_APP_VERSION "3.1.0"
/*kv不属于纯krhino模块，需要关闭*/
#define TEST_CONFIG_KV_ENABLED          (0)
```

- 将rhino_test.c和cut.c/cut.h加入编译体系

可以将test/testcase/certificate_test目录下此三个直接拷贝到对应mcu下，新建一个test目录并加入到makefile；其他IDE直接添加编译文件。

- 在主任务中调用test_certificate执行测试用例认证直到用例通过即可。

上面属于纯rhino内核的测试方式，如果带aos接口层的测试请参考，

AliOS Things Kernel

测试指南：https://github.com/alibaba/AliOS-Things/wiki/Manual-API

6、代码合入整体原则

6.1、公共代码修改

公共代码原则上避免修改，以影响其他单板。通用文件修改后，需要确认不影响其他工程的编译和运行。如果影响公共代码，需要清晰说明：是修复bug、增加新特性、或是改进功能，并介绍如何完成的。

公共代码范围：目前除新增board目录、新增mcu目录，其他目录或者文件都视为公共文件，包括app/example目录。修改后，都可能影响其他单板。

6.2、编译链接选项限制

在编译选项中，严禁加-w选项关闭编译告警，以控制上传代码质量。

6.3、License准则

- 原创为主尊重版权，请作者的标明自己的 copyright 信息，并签署CLA
- 禁止合入这些 license 的代码: AGPL, CPAL, OSL 等严格开源许可证

- 谨慎处理 GPL/LGPL 等强制开源许可证的代码，考虑以下替代方案

（1）将 GPL/LGPL 或类似许可证源码编译为二进制程序，作为独立软件使用

（2）将 LGPL 或类似许可证源码编译为动态连接库，并以动态连接方式调用

（3）将 LGPL

或类似许可证源码编译为静态链接库与应用程序相结合发布，但同时提供整个应用程序(含

LGPL 静态连接库)的目标代码和 LGPL 库源码

- 允许使用的 license: BSD, MIT, Apache License Version 2.0, Zlib, CDDL 等宽松开源许可证

6.4、CI验证通过

- Build: 代码 autobuild, PV build 通过
- Test: PV 测试通过(目前暂无此步骤)
- Agreement: CLA 已签署

4.2.4. k_config.h说明

AliOS-Things的内核（称为Rhino）可以通过宏进行功能配置。完整的配置宏可以在"k_default_config.h"文件中看到，里面的宏可分为两类——开关类与数值设置类。开关类负责打开或关闭一个内核模块，数值设置用于设定一些参数。"k_default_config.h"文件位于Rhino内核代码中，其对每个可配置宏都进行了默认值的设置。

针对每一个AliOS-Things支持的单板，还配套有一个"k_config.h"。其用于设定本单板环境下特定的内核Rhino配置，这些宏配置值通常与"k_default_config.h"值不同。这些"k_config.h"位于\board***目录下，***为具体单板名称。

AliOS-Things内部组件都是通过#include "k_api.h"来使用这些配置宏的，"k_api.h"中固定包含顺序：

```
#include "k_config.h"
#include "k_default_config.h"
```

所以，单板特定的宏设置优先于默认设置。以信号量功能开关举例：

```
//"k_config.h"中这么描述：
#ifndef RHINO_CONFIG_SEM
#define RHINO_CONFIG_SEM            1
#endif
//"k_default_config.h"中这么描述
#ifndef RHINO_CONFIG_SEM
#define RHINO_CONFIG_SEM            0
#endif
```

最终RHINO_CONFIG_SEM生效值为1，即信号量功能打卡。对于"k_config.h"中未出现的Rhino内核配置项，则"k_default_config.h" 中默认值生效。

常用配置选项说明

- **RHINO_CONFIG_SEM**

信号量模块的开关，"0"表示关闭 / "1"表示打开。主要对应"k_sem.h"中的功能。

- **RHINO_CONFIG_TASK_SEM**

任务信号量模块的开关，"0"表示关闭 / "1"表示打开。主要对应"k_task_sem.h"中的功能。对比信号量用于同步或互斥场景，任务信号量只用于同步，提供更高效方便的方式。

- **RHINO_CONFIG_QUEUE**

队列模块的开关，"0"表示关闭 / "1"表示打开。主要对应"k_queue.h"中的功能。

- **RHINO_CONFIG_BUF_QUEUE**

缓存队列模块的开关，"0"表示关闭 / "1"表示打开。主要对应"k_buf_queue.h"中的功能。

- **RHINO_CONFIG_PWRMGMT**

功耗管理模块的开关，"0"表示关闭 / "1"表示打开。用于开启低功耗功能（该功能需要厂商BSP配合OS一同完成）。

- **RHINO_CONFIG_TIMER**

timer模块的开关，"0"表示关闭 / "1"表示打开。主要对应"k_timer.h"中的功能。

- **RHINO_CONFIG_TIMER_TASK_PRI**

timer模块打开时，定时器超时回调都在内核创建的定时器任务上下文中执行。该任务优先级通过上述宏配置。timer任务优先级与用户的回调实际工作有关，通常优先级设定的较高。

- **RHINO_CONFIG_TIMER_TASK_STACK_SIZE**

timer模块打开时，定时器超时回调都在内核创建的定时器任务上下文中执行。该任务栈大小通过上述宏配置，单位是4字节（比如宏值define成256，表示任务栈实际为1024字节大小）。timer任务栈大小与用户的回调实际工作有关，初始可以设定大一点，运行时通过cli的tasklist命令查看，若timer任务（名称为"timer_task"）栈最小空闲值较大，则可以减小该宏以节省内存。

- **RHINO_CONFIG_SCHED_RR**

任务round robin调度方式开关，"0"表示关闭 / "1"表示打开。Rhino为实时调度内核，即高优先级任务永远优先于低优先级任务执行。而对于相同优先级的任务，则有两种调度策略：

1. RR，即相同优先级任务分享时间片，每个任务执行到一定时间后自动让出CPU，供下一个同优先级任务执行；
2. FIFO，即相同优先级任务先进入ready状态的先执行，只有本任务发生阻塞（比如sleep或者等待信号量等）后，才能轮到相同优先级下一个任务执行；

RHINO_CONFIG_SCHED_RR为0和1分别对应RR与FIFO方式。

- **RHINO_CONFIG_TIME_SLICE_DEFAULT**

当任务round robin调度方式打开时（即RHINO_CONFIG_SCHED_RR为1），每个任务的时间片可在创建时指定，若创建时填写0则为该宏的默认值。单位毫秒。

- **RHINO_CONFIG_TICKS_PER_SECOND**

配置每秒系统的tick数，比如100表示每10ms到来一个系统tick，1000表示每1ms都有个tick。系统tick是内核计时的基础单位。超时时间（如sleep，sem_take等）与定时器控制，内部都已tick为计数基础。因此该宏值越高，表示计时精度越高，但系统处理tick中断本身的消耗也就越大。

- **RHINO_CONFIG_SYSTEM_STATS**

内核系统统计开关，"0"表示关闭 / "1"表示打开。打开后完成以下统计：

1. 统计全系统的 最长关中断时间、最长关任务调度时间与任务切换次数；
2. 针对每个任务，统计该任务最长关中断时间与最长关任务调度时间；
3. 针对每个任务，统计该任务执行次数、总耗时以及CPU占用率；

- **RHINO_CONFIG_MM_TLF**

堆管理算法开关，"0"表示关闭 / "1"表示打开。打开后Rhino接管malloc, free等C库的内存管理，并提供"k_mm.h"中的功能。

- **RHINO_CONFIG_MM_BLK**

小内存块优化算法开关，"0"表示关闭 / "1"表示打开。RHINO_CONFIG_MM_TLF打开后，Rhino使用TLF算法管理内存，该算法更加健壮但会消耗一定的内存。针对小内存快（比如小于32字节），可以通过RHINO_CONFIG_MM_BLK宏开启BLK算法优化，提高内存利用率。

- **RHINO_CONFIG_MM_TLF_BLK_SIZE**

小内存块优化空间大小，单位为字节。RHINO_CONFIG_MM_BLK打开后，需要配置RHINO_CONFIG_MM_TLF_BLK_SIZE来决定堆中多少内存划分给BLK算法。

- **RHINO_CONFIG_MM_DEBUG**

缓存队列模块的开关，"0"表示关闭 / "1"表示打开。主要对应"k_mm_debug.h"中的功能。打开后，当用户申请内存不足，或者rhino检测到内存越界时，都会有详细的打印。CLI中也有"dumpsys mm_info"命令可以查看详细内容。

4.2.5. 基于keil\iar的内核基础功能移植指导

目标示例开发板：STM32F103ZET6，Cotex-m3架构

移植目标：基本任务运行，tick时钟实现任务周期睡眠，基本串口打印

版本：基于AOS-R-2.1.0版本，其他版本可能存在具体的目录和规范不一样，整体流程相同。

1.1、使用ST CubeMX自动生成AliOS Things工程

ST CubeMX工程中集成了AliOS Things的组件功能，用户可以在CubeMX中通过组件选择的方式生成对应的keil或iar工程。具体操作指导参考网址：https://github.com/alibaba/AliOS-Things/wiki/Generate-Keil-IAR-Project-via-ST-CubeMX.zh

1.2、使用IDE命令自动生成keil\iar工程介绍

AliOS Things支持在aos make 编译的时候使用IDE=keil 或 IDE=iar命令来自动生成对应的keil\iar工程。此功能的前提是已经基于该单板建立了gcc的编译体系。具体参考：https://github.com/alibaba/AliOS-Things/wiki/Auto-generate-keil-iar-project 如，通过命令aos make rhinorun@stm32f103ze IDE=keil可以在project目录下，生成如下的内核keil工程：

生成的工程相关的编译文件、编译选项等都已经配置好，可以直接编译和使用；如果存在较少的编译问题，直接按照提示解决具体问题即可。

2、手动建立kei\iar示例指导

上面提到的自动化工具，参考相应的指导说明操作即可；本章节主要提供手动建立基本的内核keil\iar工程的示例指导。按照实际工程搭建的过程来说明。

2.1、基本代码准备

单板相关的所有代码放入AliOS Things系统时，都需要遵从《AliOS Things板级支持目录规范》，包括app\board\mcu等目录结构内的文件和目录布局，具体细节见wiki：https://github.com/alibaba/AliOS-Things/wiki/AliOS-Things%E6%9D%BF%E7%BA%A7%E6%94%AF%E6%8C%81%E7%9B%AE%E5%BD%95%E8%A7%84%E8%8C%83本文档侧重于介绍keil\iar工程相关的工程建立，涉及到具体的系统代码实现细节可参考《AliOS Things内核驱动移植指导》：https://github.com/alibaba/AliOS-Things/wiki/AliOS-Things%E5%86%85%E6%A0%B8%E9%A9%B1%E5%8A%A8%E7%A7%BB%E6%A4%8D%E6%8C%87%E5%AF%BC

2.1.1、STM32CubeF1系列Drivers驱动程序

下载安装STM32CubeMX，选择生成STM32F103ZET6驱动程序，或者官网直接下载STM32CubeF1软件包。驱动包下载更新地址：http://www.st.com/content/st_com/en.html选择搜索STM32CubeMX或者STM32CubeF1关键字。

STM32CubeMX下载后直接安装使用。下面是通过STM32CubeMX生成驱动源码的基本流程参考。

- STM32CubeMX选择“New Project”，MCU选择STM32F103ZE系列，参考开发板为STM32F103ZET6，选择对应的MCU名；

- 设置Pinout查看对应的芯片手册，选择USART1作为打印串口，并配置相关时钟，对应的PA9\PA10分别为发送和接收接口。
- 设置系统频率：

生成code，keil工程选择MDK-ARM V5，iar工程选择EWARM:

点击确定后，相关keil工程的所有文件则生成在指定目录，如：

生成工程时，选择EWARM:

在生成目录下，会产生iar的工程文件，其他文件和Keil生成相同：

Drivers下面为该stm32f1xx系列的通用驱动；MDK-ARM中为keil工程文件；EWARM下为iar工程文件；Src和Inc为stm32f103ze单板的系统初始化程序。生成代码后，相关代码需要拷贝到AliOS Things工程。相关SDK驱动统一放在：platform\mcu\stm32f1xx\drivers下；Keil工程统一放在：projects\Keil\rhinorun@stm32f103ze\keil_project目录下；iar工程统一放在：projects\IAR\rhinorun@stm32f103ze\iar_project目录下。其中rhinorun表示需要运行的实例；stm32f103ze为单板名。其他文件待放入board目录下。

2.1.2、rhino内核源码

源码路径：kernel\rhino

2.1.3、Cortex-m3相关代码

实现代码路径：Cortex-m3实现：platform\arch\arm\armv7m\armcc\m3arch公共文件：platform\arch\arm\armv7m\common

2.1.4、Mcu相关代码

2.1.4.1、mcu下drivers目录

在platform\mcu下建立stm32f1xx代表该MCU体系的目录，下面包含drivers和hal目录。按照2.1.1章节描述，将CubeMX生成的sdk放入platform\mcu\stm32f1xx\drivers目录下,不做任何修改。

2.1.4.2、mcu下hal目录

对于基本的打印功能，首先需要实现uart的hal接口，在mcu\stm32f1xx下面建立hal目录，专门存放hal相关的实现，建立hal_uart_stm32f1.c文件，实现相关的hal_uart_init以及hal_uart_send接口。Hal层代码具体的实现参考：https://github.com/alibaba/AliOS-Things/wiki/AliOS-Things-HAL-Porting-Guide#1%E7%A1%AC%E4%BB%B6%E6%8A%BD%E8%B1%A1%E5%B1%82%E7%A7%BB%E6%A4%8D

2.1.5、Board相关系统启动初始化

在board下面建立stm32f103ze单板目录，并按照如下目录结构规范建立相应的目录文件：

```
Dir\File                Description                    Necessary for kernel run
|-- drivers             # board peripheral driver                    N
|-- config
|  |-- board.h          # board config file, such as uart port num          Y
|  |-- k_config.c       # user's kernel hook and mm memory region define      Y
|  |-- k_config.h       # kernel config file .h                    Y
|  |-- partition_conf.c    # board flash config file                    N
|-- startup
|  |-- board.c          # board_init implement                     Y
|  |-- startup.c        # main entry file                   Y
|  |-- startup_gcc.s      # board startup assember for gcc              Y
|  |-- startup_iar.s      # board startup assember for iar              Y
|  |-- startup_keil.s     # board startup assember for keil             Y
|-- stm32f103ze.icf       # linkscript file for iar                  Y
|-- stm32f103ze.ld        # linkscript file for gcc                  Y
|-- stm32f103ze.sct       # linkscript file for sct                  Y
|-- aos.mk              # board makefile                    Y
|-- Config.in           # menuconfig component config                 Y
```

当然对于只建立keil\iar工程来说，aos.mk、gcc相关的文件都是不必要的。

2.1.5.1、Board下drivers目录

此处drivers目录下驱动的相关实现都来源于CubeMX生成的keil\iar工程中Src和Inc目录：生成的Src目录：

生成的Inc目录：

将main.c修改为stm32f1xx_main.c，main.h修改为stm32f1xx_main.h，注释掉内部生成的系统默认main函数。后续将会在board\stm32f103ze\startup\startup.c中建立OS系统级的main函数。所有文件统一放在board\stm32f103ze\drivers下作为板级驱动。修改后文件列表：

2.1.5.2、Board下config目录

```
|-- config
|  |-- board.h
|  |-- k_config.c
|  |-- k_config.h
|  |-- partition_conf.c
```

其中，board.h为具体的设备逻辑port定义；k_config.c内为内核相关的配置接口实现；k_config.h为内核的配置头文件；partition_conf.c为flash分区配置。此部分代码可以直接参考其他board单板实现，进行相应的参数修改。

2.1.5.3、Board下startup目录

```
|-- startup
|  |-- board.c
|  |-- startup.c
|  |-- startup_gcc.s
|  |-- startup_iar.s
|  |-- startup_keil.s
```

其中，board.c中实现板级的初始化board_init，可参考原有main.c中的main函数初始化相关硬件；startup.c为OS级的启动代码，基本可以实现通用，可直接参考其他实现；关于keil的启动汇编代码startup_keil.s，在生成keil工程时已经自动生成；iar的启动汇编startup_iar.s在生成iar工程时产生。

2.1.5.4、单板初始化示例代码(startup.c)

（1）、系统初始化示例：

```
int main(void)
{
  /*irq initialized is approved here.But irq triggering is forbidden, which will enter CPU scheduling.
  Put them in sys_init which will be called after aos_start.
  Irq for task schedule should be enabled here, such as PendSV for cortex-M4.
  */
  board_init();  //including aos_heap_set(); flash_partition_init();
  /*kernel init, malloc can use after this!*/
  krhino_init();
  /*main task to run */
  krhino_task_dyn_create(&g_main_task, "main_task", 0, OS_MAIN_TASK_PRI, 0, OS_MAIN_TASK_STACK, (task_e
ntry_t)sys_init, 1);
  /*kernel start schedule!*/
  krhino_start();
  /*never run here*/
  return 0;
}
```

（2）、主任务初始化示例：

```
static void sys_init(void)
{
  /* user code start*/
  /*insert driver to enable irq for example: starting to run tick time.
   drivers to trigger irq is forbidden before aos_start, which will start core schedule.
  */
  /*user_trigger_irq();*/ //for example
  /*aos components init including middleware and protocol and so on !*/
  aos_components_init(&kinit);
  #ifndef AOS_BINS
  application_start(kinit.argc, kinit.argv); /* jump to app/example entry */
  #endif
}
```

（3）、用户app入口示例（参考test\develop\rhinorun\rhinorun.c）：

```
int application_start(int argc, char *argv[])
{
  int count = 0;
  printf("nano entry here!\r\n");
  while(1) {
    printf("hello world! count %d \r\n", count++);
    aos_msleep(1000);
  };
}
```

2.2、基本内核代码适配

目标建立一个基本的延时打印任务，需要的代码修改包括：

· 基本的任务处理和调度在2.1.2和2.1.3代码中已经提供；

· stm32f1xx_it.c中修改tick时钟：

SysTick_Handler中断处理调用krhino处理函数krhino_tick_proc；

HAL_InitTick设置每秒Tick数时，使用宏RHINO_CONFIG_TICKS_PER_SECOND；

· board_init.c基本驱动初始化：系统时钟、uart以及tick等的初始化。

· rhinorun.c中实现用户app入口函数：application_start，内部实现循环睡眠打印。

· hal_uart*.c打印接口：见章节2.1.4.2中hal介绍。

· main函数初始化：章节2.1.5.4已介绍

2.3、keil\iar工程配置

2.3.1、内核工程依赖源文件

下述为新增一个单板支持，必须关注的几个目录项：

App\example	通用用户运行实例，如helloworld实例，可直接使用，无特殊情况不修改
test\develop	用户自定义特殊运行实例，满足某一特定场景时添加，和上面example同用途
board	用户需要适配、可配置board级代码，系统启动相关代码
Platform\arch	该CPU架构内核调度适配接口，可直接使用
Platform\mcu	该MCU通用SDK以及对应的hal适配层
Rhino	内核源码
utility\libc	C库适配接口：newlib_stub.c(gcc)、armcc_libc.c(armcc)、iar_libc.c(icc)

原则上基本内核工程使用上述组件，即可组成完整的闭环系统；但是由于具体接口实现方式，可能会引入其他关联依赖模块，主要包括：

kernel\debug模块	platform\arch中异常处理代码需要使用调试接口；系统资源紧张时，可以将arch下面的panic*.c和该debug模块整体从工程中删除。
Osal模块	C库适配接口中实现依赖aos API接口
Ulog	系统记录日志功能，如果只使用printf，该模块可删除。
Kernel\init	系统初始化后，如果调用中间件和协议层初始化接口aos_components_init，即会引入。可删除。

上表中都是被引入的依赖模块，如果在系统资源紧张的情况下，用户可以按照自己的需求，去掉依赖关系，并将上述模块不加入编译体系。例如去掉aos_components_init的引用，即不会引入Kernel\init。

2.3.2、keil工程建立示例

2.3.2.1、keil工程配置

基于生成的Keil工程路径，配置需要编译的源文件，需要包含的头文件，需要使用的编译选项，以及需要的链接选项等。章节2.1.1中已经将CubeMX生成的Keil工程拷贝到AliOS Things的project目录下

打开该keil工程，开始相关的配置操作。主要配置项包括：·加入编译.c.S源文件：右击工程，选择“Manage Project Items”；或通过工具栏快捷键选择：

“Manage Project Items” 中建立相关groups名，一般按照子目录或者组件取名，命名直观；并包含该目录需要编译的.c以及汇编.s.S文件；

参考2.3.1内核工程依赖源文件的描述，对于目前基本的kernel工程，建立如下Groups分组：

rhinorun	test/develop/rhinorun
board_stm32f103ze	board/stm32f103ze
arch_armv7m	platform/arch/arm/armv7m中m3相关
mcu_stm32f1xx	platform/mcu/stm32f1xx下drivers和hal目录；其中drivers按照具体厂商需要包含的实际驱动文件为准；hal下面优先包含hal_uart*.c，其他hal按需增删。
rhino	kernel/rhino下所有文件
newlib_stub	utility/libc/compilers/armlibc
debug	kernel/debug
osal	osal/aos基本的common.c和rhino.c
ulog	middleware/uagent/ulog
kernel_init	kernel/init

添加对应的文件后，如下图所示：

·右击工程，选择“Options for Target”，内部集成了所有相关的配置项；或通过工具栏快捷键选择，如下图。用户也可以设置具体“Groups”或者具体某个文件的配置项；建议没有特殊需求，统一使用project项目的总配置。

首先要确认“Device”选项卡中，Device型号选择正确：

“C/C++”和asm选项卡设置编译选项、建立编译依赖头文件；对于AliOS Things工程需要的编译选项，可以参照已有board的实现来添加。对于引用的头文件，在上一章节中所有包含目录下的头文件目录，理论上都需要增加进入” Include Paths”。

·选择工程“Options for Target”，“Linker”选项，用户可以自设置链接sct文件，如果不设置则采用系统默认链接文件；在” Misc controls”中设置对应的链接选项。

其他选项采用默认配置，也可按需调整。工程建立后编译工程到编译OK。

2.3.2.2、keil单板调试

·右击工程，选择“Options for Target”，“Debug”选择ST-Linker Debugger。

· Load 烧录bin文件:通过菜单栏Flash->download，或者工具栏快捷键完成代码烧写。

· Debug调试代码：通过菜单栏通过Debug->Start/Stop Debug Sesion，或者工具栏快捷键打开调试界面。

· 连接串口工具，查看打印Logapplication_start中每周期打印一次串口，并将当前任务睡眠，运行OK后如下图所示：

2.3.3、iar工程建立示例

2.3.3.1、iar工程配置

基于生成的iar工程路径，配置需要编译的源文件，需要包含的头文件，需要使用的编译选项，以及需要的链接选项等。章节2.1.1中已经将CubeMX生成的iar工程拷贝到AliOS Things的project目录下

打开该iar工程，开始相关的配置操作。主要配置项包括：· 加入编译.c.S源文件，依然按照先建立Group，再建立Files的方式：

右击workspace区域，弹出菜单中选择add，可以添加Group；选中对应的Group右击，选择Add Files，可以选择属于该Group的源文件。参考2.3.1内核工程依赖源文件的描述，对于目前基本的kernel工程，同keil工程，也建立2.3.2.1中描述的Groups分组；添加对应的文件后，如下图所示：

· 右击工程名，弹出菜单选择“Options”，打开整体工程配置项；用户也可以选中某个Group或者文件右击，单独给其修改配置。

打开”Options”窗口后，首先要确认设备Device类型是否正确：

“C/C++ Compiler”和”Assembler”选项卡设置编译选项、建立编译依赖头文件；对于AliOS Things工程需要的编译选项，可以参照已有board的实现来添加。对于引用的头文件，在所有包含目录下的头文件目录，理论上都需要增加进入”Additional include directories”。在“Defined symbols”选项中增加预编译宏定义；” Extra Options”标签中增加编译选项定义。

· 选择选项卡“Linker”，在标签“Config”设置链接脚本；在”Extra Options”标签中增加链接选项定义。

其他选项采用默认配置，也可按需调整。工程建立后编译，可能存在具体的编译问题，解决后直到工程编译完成。

2.3.3.2、iar单板调试

· 右击工程，选择“Options”，弹出配置窗口后，打开选项卡“Debugger”，点击标签“Setup”，” Driver”选择ST-LINK。

· 选择工具栏“Download and Debug”快捷键，进行烧录调试。

· 连接串口工具，查看打印Log

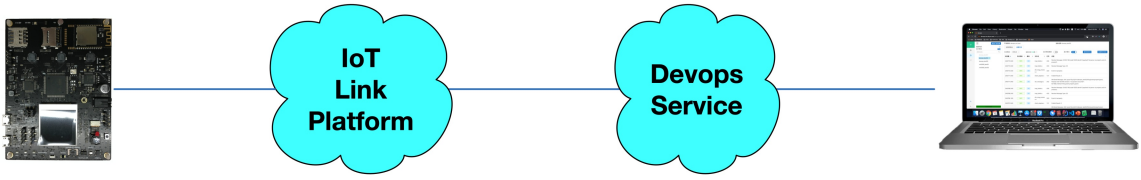
5.远程调试运维指南

AliOS Things内置了uAgent模块来支持远程调试运维。同时，我们在云端提供了调试运维控制台，来让开发者通过网页远程触达设备。

无论对于我们开发中的设备，还是已经量产的设备，远程调试运维功能都能为我们快速解决问题提供便利。AliOS Things通过uAgent模块来为设备提供远程调试运维功能。结合阿里云IoT的调试运维服务控制台，开发者可以很方便地通过网页远程调试运维设备，快速诊断解决遇到的问题。

How Does It Work?

运程调试运维服务的架构图如下图所示。对于连接到阿里云IoT物联网平台的AliOS Things设备，其搭载的uAgent模块可以与云端远程调试运维服务通信交互，从而让开发/运维人员可以在自己电脑上远程操作运维设备。



AliOS Things设备 IoT LP平台 远程调试运维服务 开发/运维人员

远程调试运维功能开通与使用步骤

1. 开启设备的远程调试运维功能
2. 开通远程调试运维服务
3. 对设备做远程调试运维操作，如：远程与设备做命令行交互远程查看分析设备日志

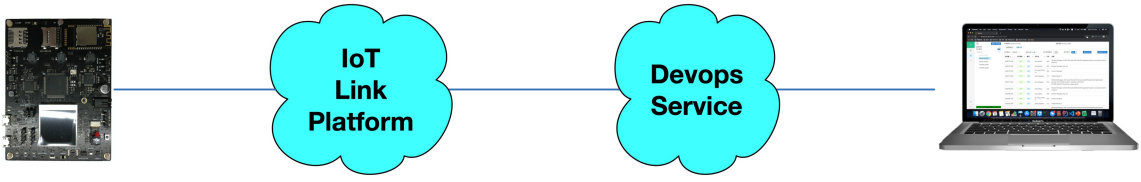
5.1. 概述

AliOS Things内置了uAgent模块来支持远程调试运维。同时，我们在云端提供了调试运维控制台，来让开发者通过网页远程触达设备。

无论对于我们开发中的设备，还是已经量产的设备，远程调试运维功能都能为我们快速解决问题提供便利。AliOS Things通过uAgent模块来为设备提供远程调试运维功能。结合阿里云IoT的调试运维服务控制台，开发者可以很方便地通过网页远程调试运维设备，快速诊断解决遇到的问题。

How does It Work?

运程调试运维服务的架构图如下图所示。对于连接到阿里云IoT物联网平台的AliOS Things设备，其搭载的uAgent模块可以与云端远程调试运维服务通信交互，从而让开发/运维人员可以在自己电脑上远程操作运维设备。



AliOS Things设备 IoT LP平台 远程调试运维服务 开发/运维人员

服务开通与使用步骤

1. 开启设备的远程调试运维功能
2. 开通远程调试运维服务

3. 对设备做远程调试运维操作，如：远程与设备做命令行交互远程查看分析设备日志

5.2. 功能开通

5.2.1. 开启设备的远程调试运维功能

本文档讲述如在设备端开启远程调试运维功能。

前置条件

- 请使用AliOS Things 3.1.0及以上版本
- 需要开通阿里云物联网服务，并通过物联网控制台创建产品和添加设备得到一个有效的设备身份证书（四元组）

创建一个连到阿里云物联网平台的应用工程（Optional）

Note：如果您已经有创建好了您的连云的应用，请忽略此步骤。

本步骤帮助您基于linkkit_demo模板创建一个在mk3072开发板上运行的连接阿里云的应用工程。如果您手上没有mk3072开发板，也可以用其它您有的AliOS Things开发板，只需在创建工程的时候选择相应的开发板就行。

步骤1. 创建应用工程

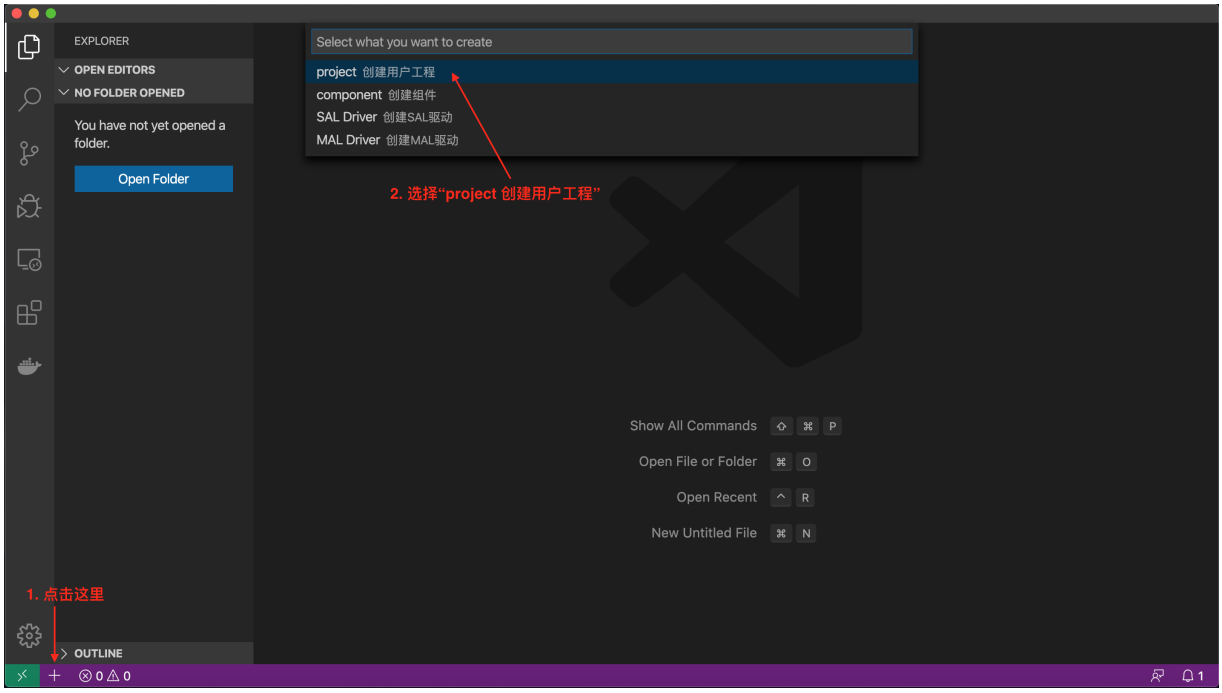
方式A. 基于命令行创建（适用于习惯命令行的开发者） 。参考文档使用命令行开发

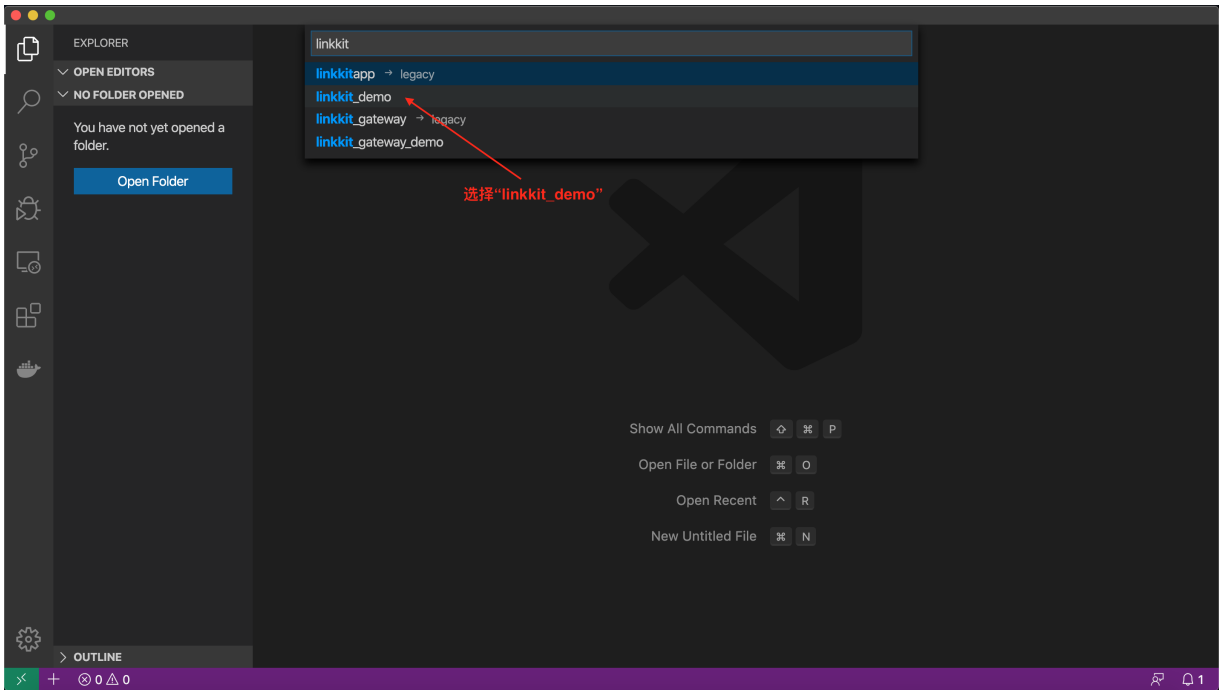
打开命Teriminal令行；输入如下命令：

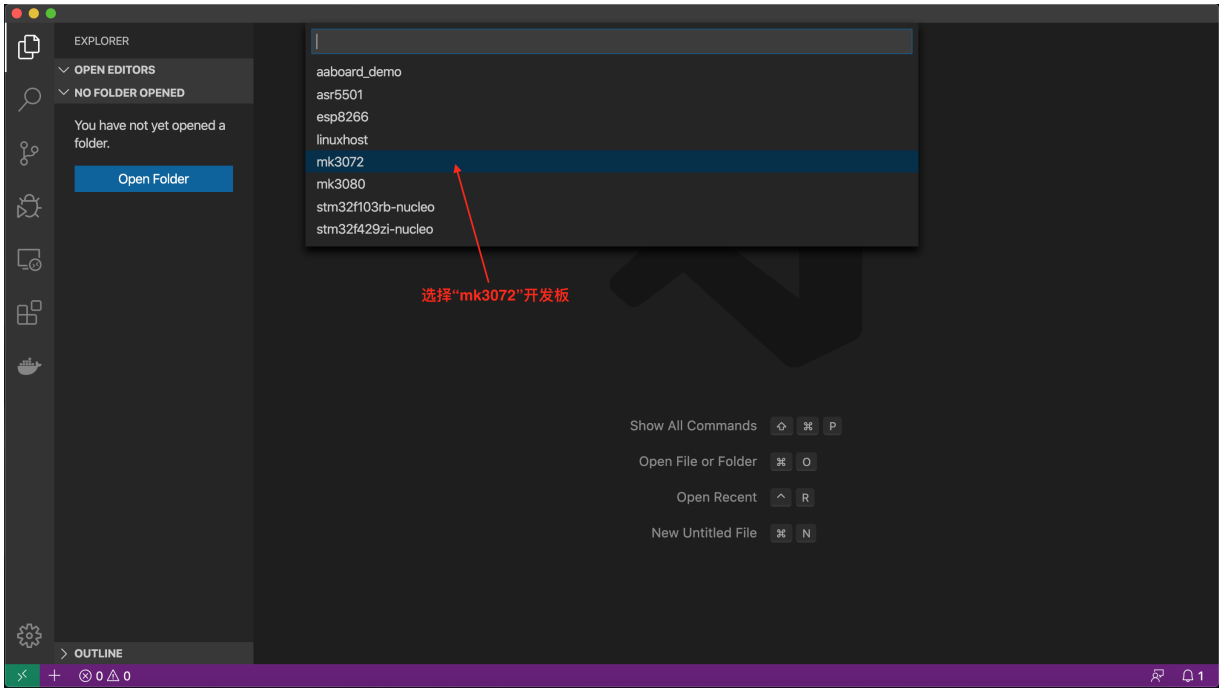
```
aos create project -d . -b mk3072 -t linkkit_demo myapp
```

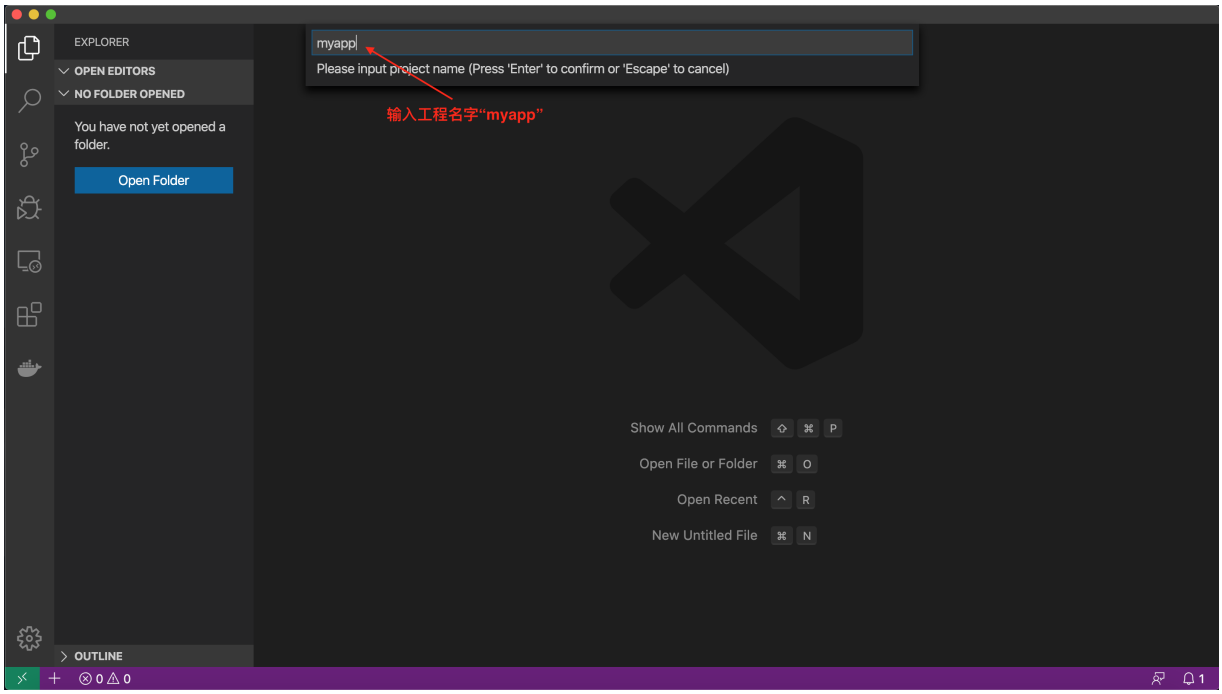
方式B. 基于AliOS Studio IDE创建（适用于习惯IDE的开发者）。参考文档使用 AliOS Studio IDE 开发

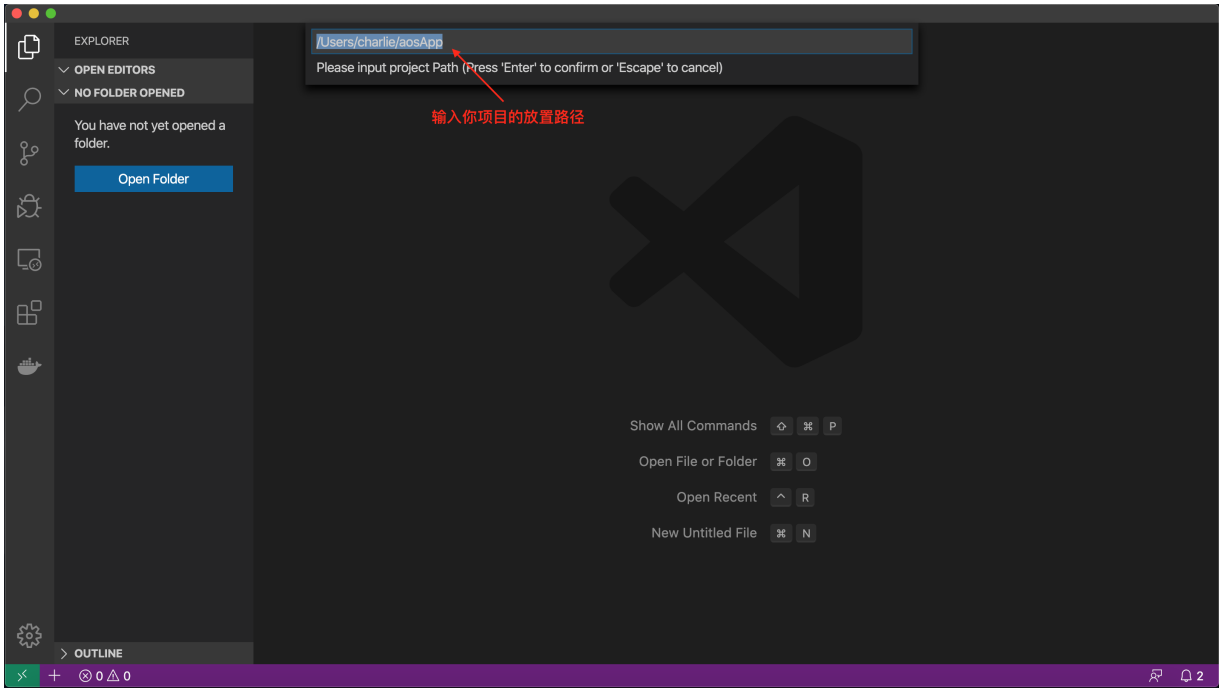
请按下面一下图片所示的方法创建您的应用工程。

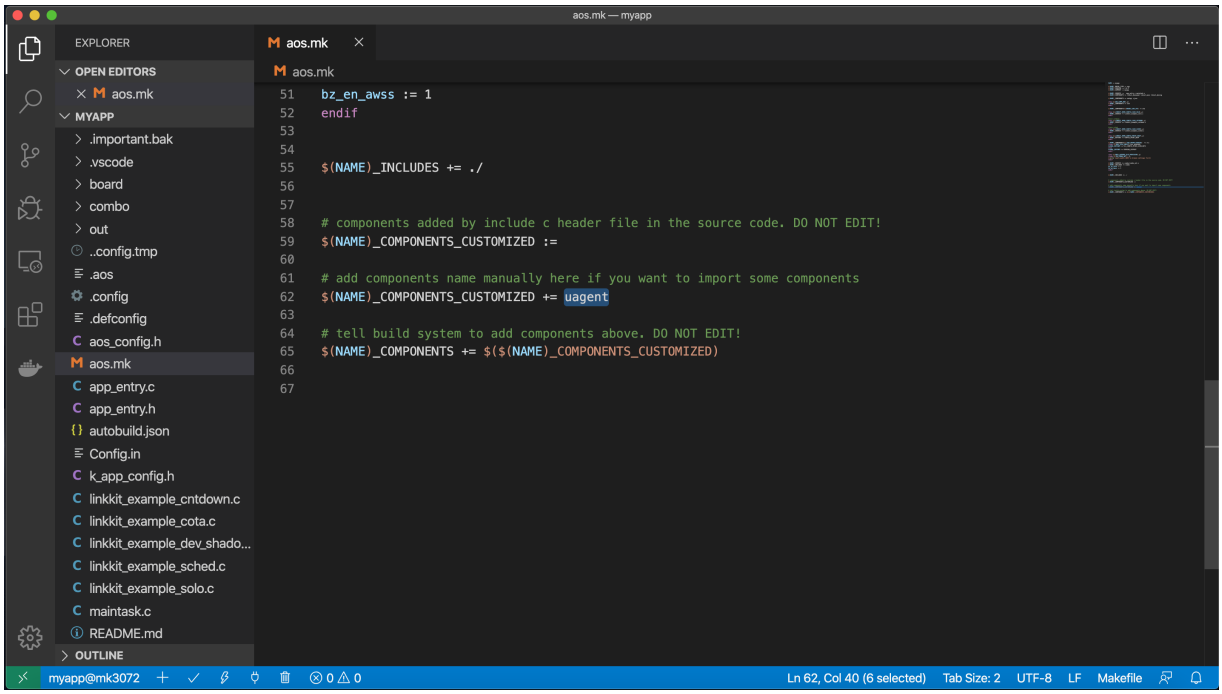

EXPLORER
OPEN EDITORS
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
Select what you want to create
project 创建用户工程
component 创建组件
SAL Driver 创建SAL驱动
MAL Driver 创建MAL驱动
2. 选择“project 创建用户工程”
Show All Commands
Open File or Folder
Open Recent
New Untitled File
1. 点击这里
OUTLINE


EXPLORER
OPEN EDITORS
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
linkkit
linkkitapp → legacy
linkkit_demo
linkkit_gateway → legacy
linkkit_gateway_demo
选择“linkkit_demo”
Show All Commands
Open File or Folder
Open Recent
New Untitled File
OUTLINE


EXPLORER
OPEN EDITORS
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
aaboard_demo
asr5501
esp8266
linuxhost
mk3072
mk3080
stm32f103rb-nucleo
stm32f429zi-nucleo
选择“mk3072”开发板
Show All Commands
Open File or Folder
Open Recent
New Untitled File
OUTLINE


EXPLORER
OPEN EDITORS
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
myapp
Please input project name (Press 'Enter' to confirm or 'Escape' to cancel)
输入工程名字“myapp”
Show All Commands
Open File or Folder
Open Recent
New Untitled File
OUTLINE





步骤2. 设置四元组信息

编辑您应用工程下的linkkit_example_solo.c文件，将其中的PRODUCT_KEY/PRODUCT_SECRET/DEVICE_NAME/DEVICE_SECRET信息修改为您阿里云账号下的有效设备身份四元组。

使能远程调试运维组件(uAgent)

步骤1. 在您的应用目录下，找到并打开aos.mk

```
vim path/to/myapp/aos.mk
```

步骤2. 按如下diff所示，在aos.mk里面增加对uagent组件的依赖

```
diff --git a/aos.mk b/aos.mk
index 1e5d10f..6a67ee2 100644
--- a/aos.mk
+++ b/aos.mk
@@ -59,7 +59,7 @@ $(NAME)_INCLUDES += ./
 $(NAME)_COMPONENTS_CUSTOMIZED :=
 # add components name manually here if you want to import some components
-$(NAME)_COMPONENTS_CUSTOMIZED +=
+$(NAME)_COMPONENTS_CUSTOMIZED += uagent
 # tell build system to add components above. DO NOT EDIT!
 $(NAME)_COMPONENTS += $($(NAME)_COMPONENTS_CUSTOMIZED)
```

步骤3. 重新编译固件，烧录到设备上

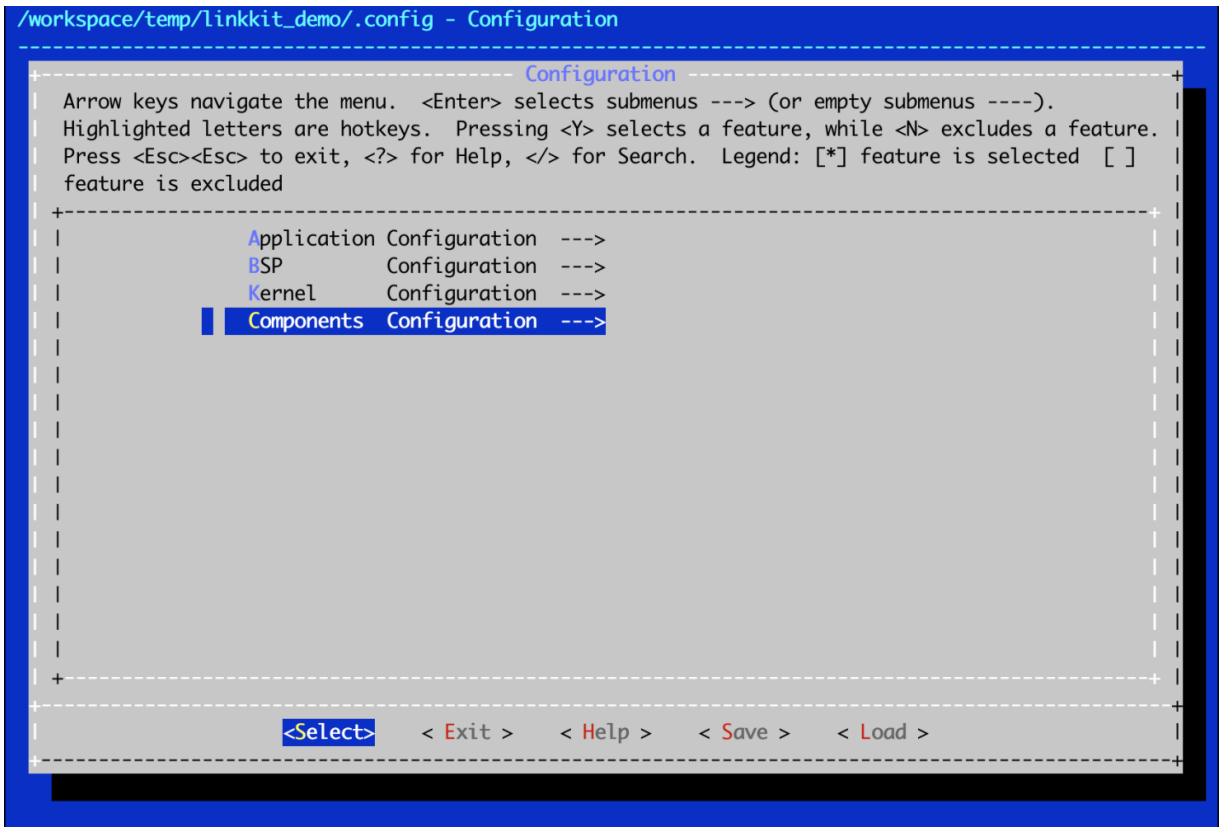
```
aos make clean && aos make
```

使能远程日志查看功能（按需使能）

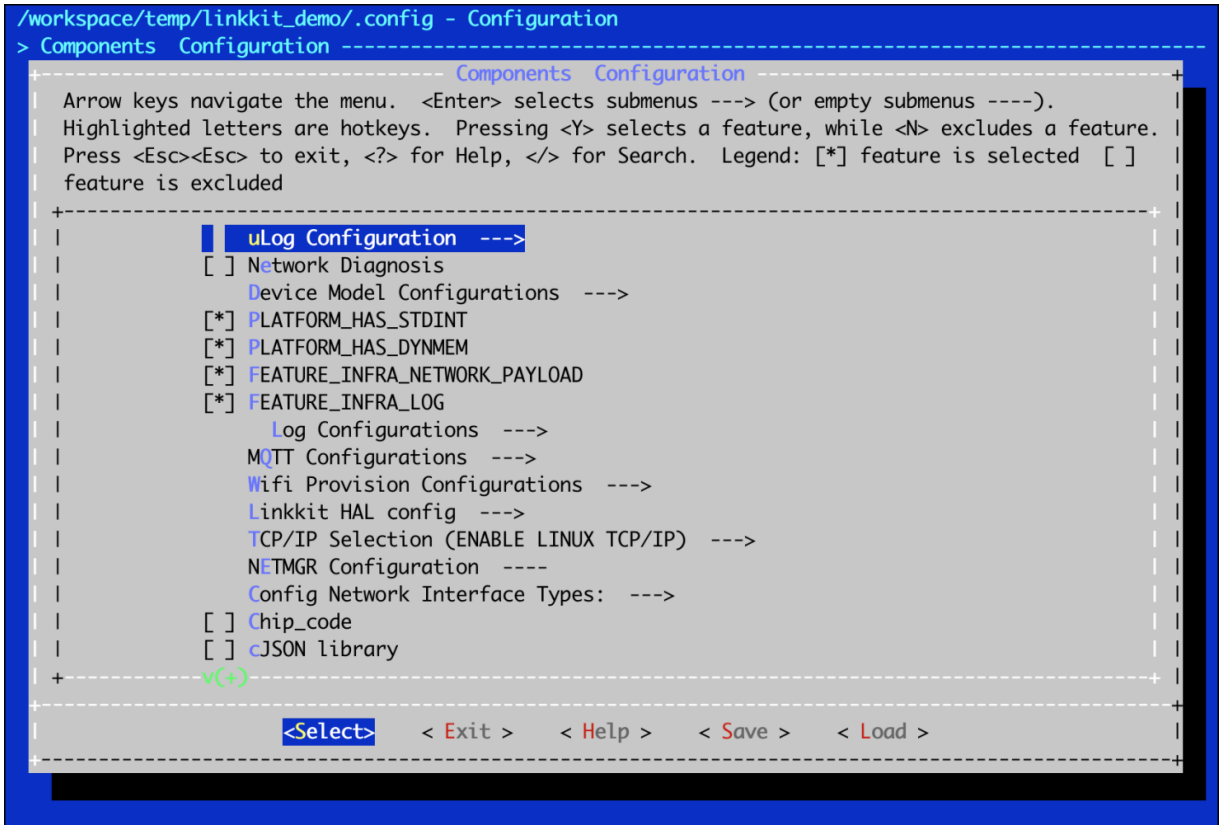
步骤1. 在您的应用目录下，输入"aos make menuconfig"命令，进入组件配置界面

```
aos make menuconfig
```

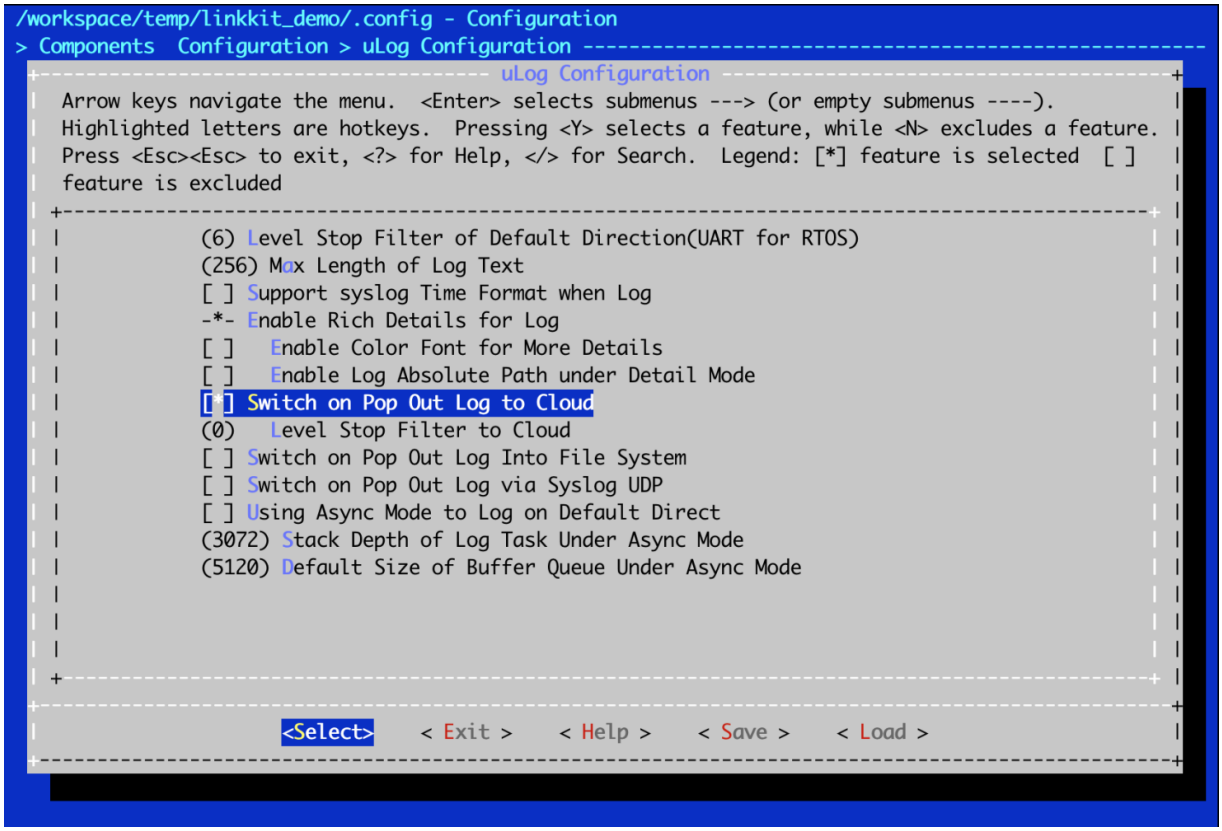
步骤2. 选择Components目录，按enter进入



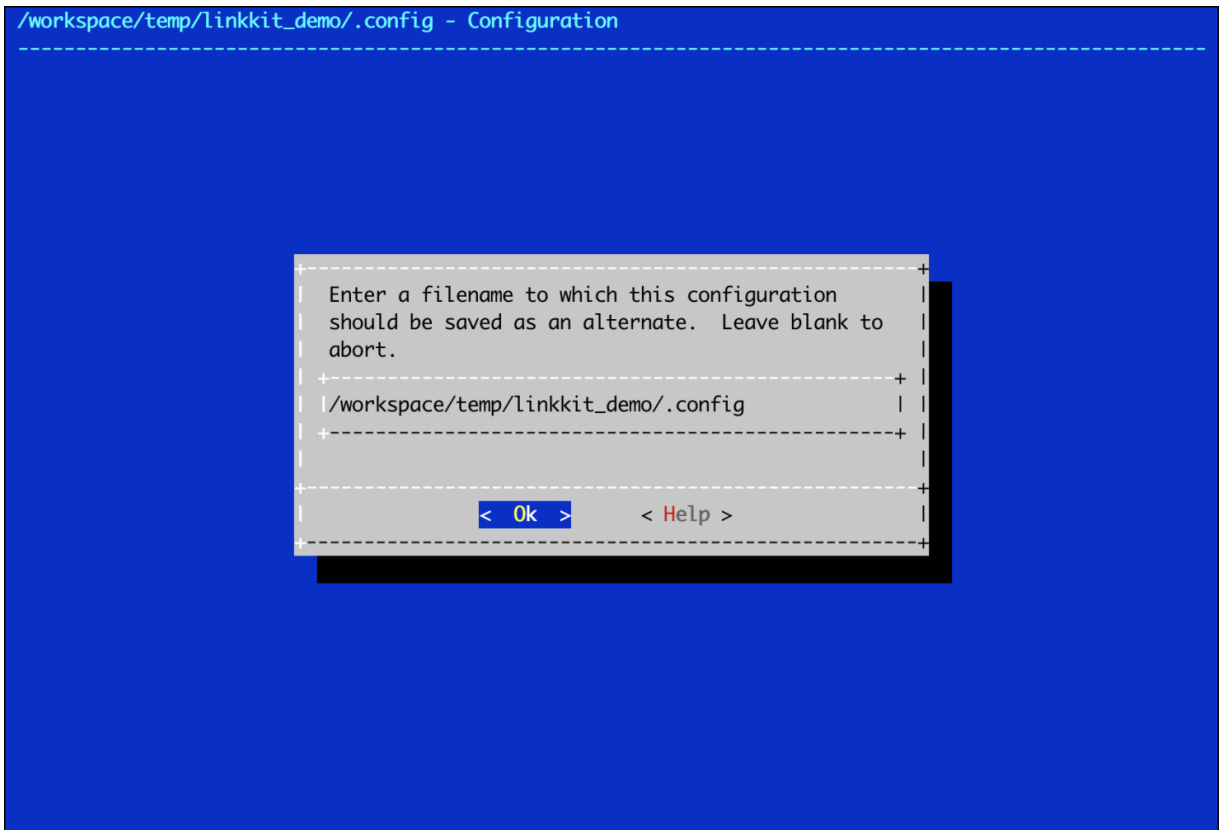
步骤3. 选择uLog Configuration目录，按enter进入



步骤4. 选择“Switch on Pop Out Log to Cloud”，按控制键使能（确保左侧方括号中有*）



步骤5. 选择目录下方的Save，按Enter保存配置。



步骤6. 多次按ecs按键退出组件化配置界面

步骤7. 重新编译固件，烧录到设备上

```
aos make clean && aos make
```

5.2.2. 开通远程调试运维服务

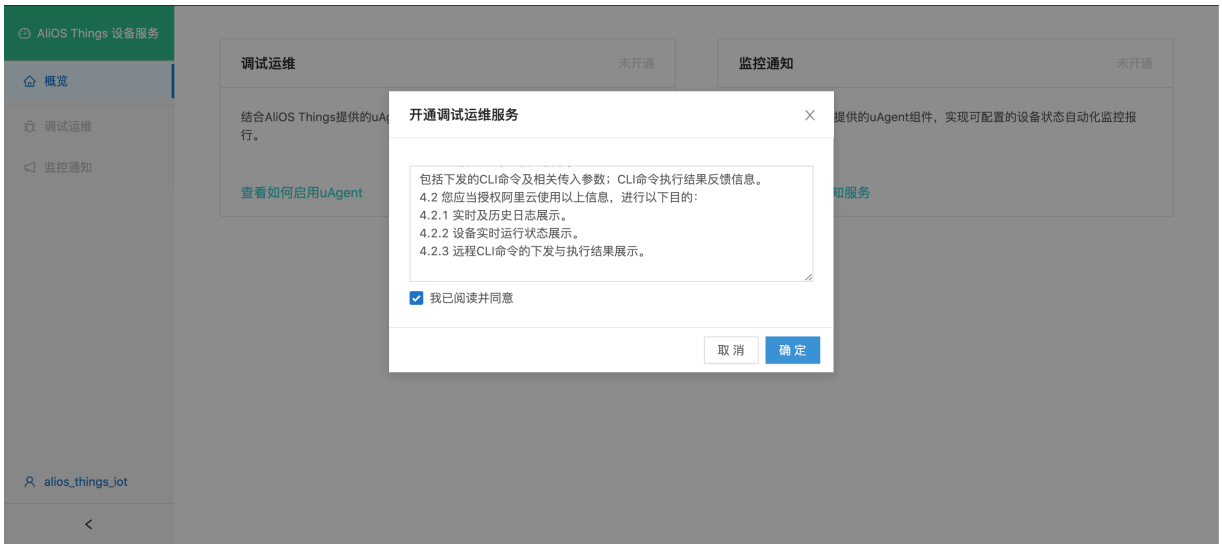
本文档讲述如何开通远程调试运维服务 和 如何开启对某个产品的远程调试运维功能

开通远程调试运维服务

步骤1. 访问 AliOS Things设备服务页面

步骤2. 在调试运维服务卡片上，点击“开通服务”按键

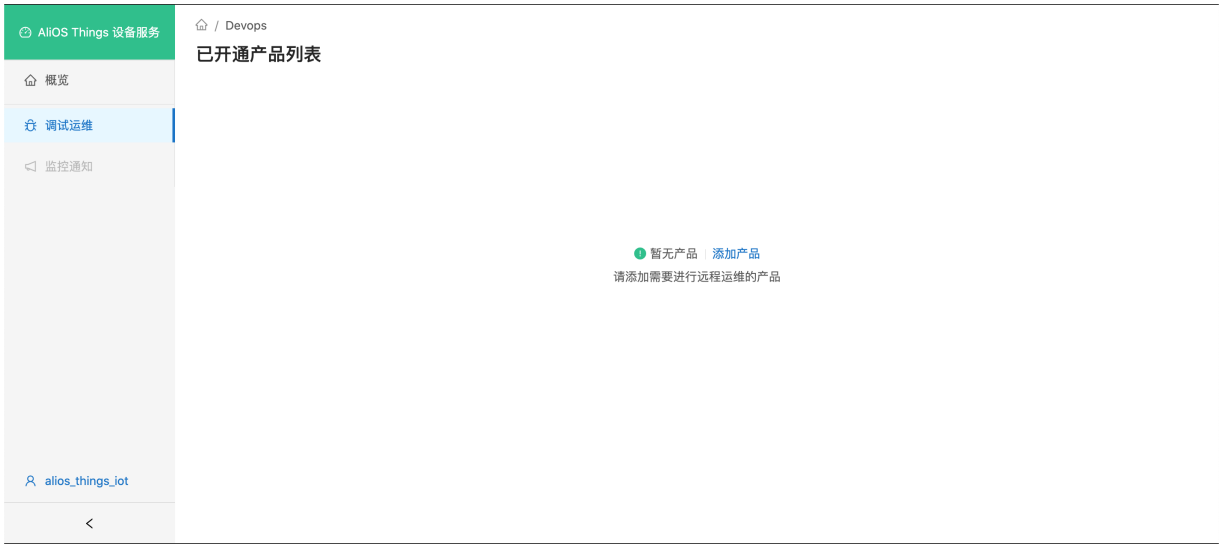
步骤3. 阅读服务协议，同意并确认开通服务



步骤4. 确保服务变成已开通状态

开启对某个产品的调试运维功能

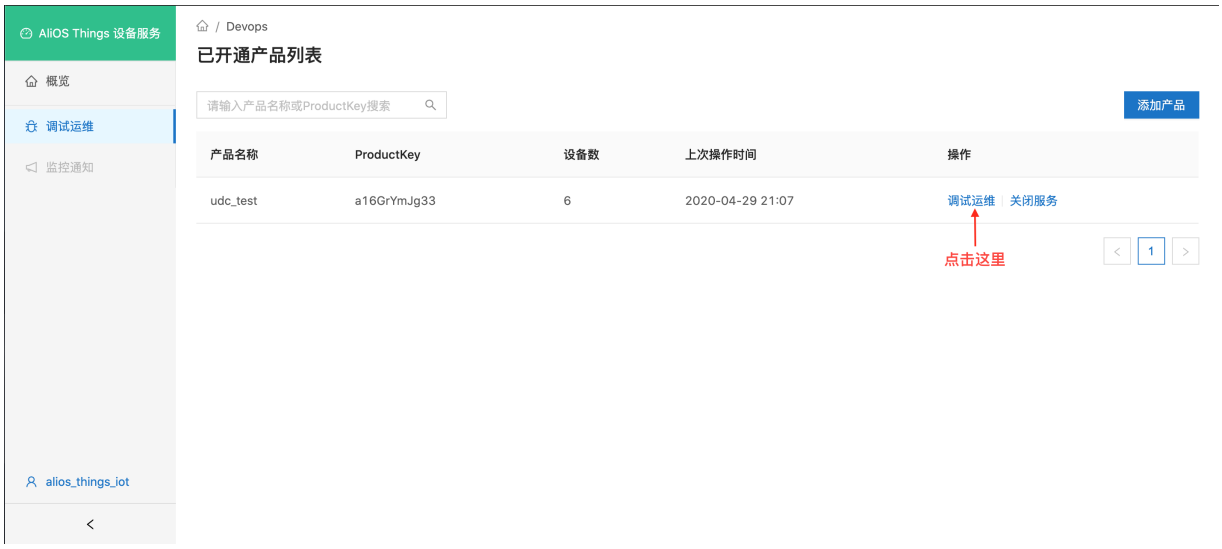
步骤1. 访问调试运维服务页面



步骤2. 开启对某个产品的调试运维

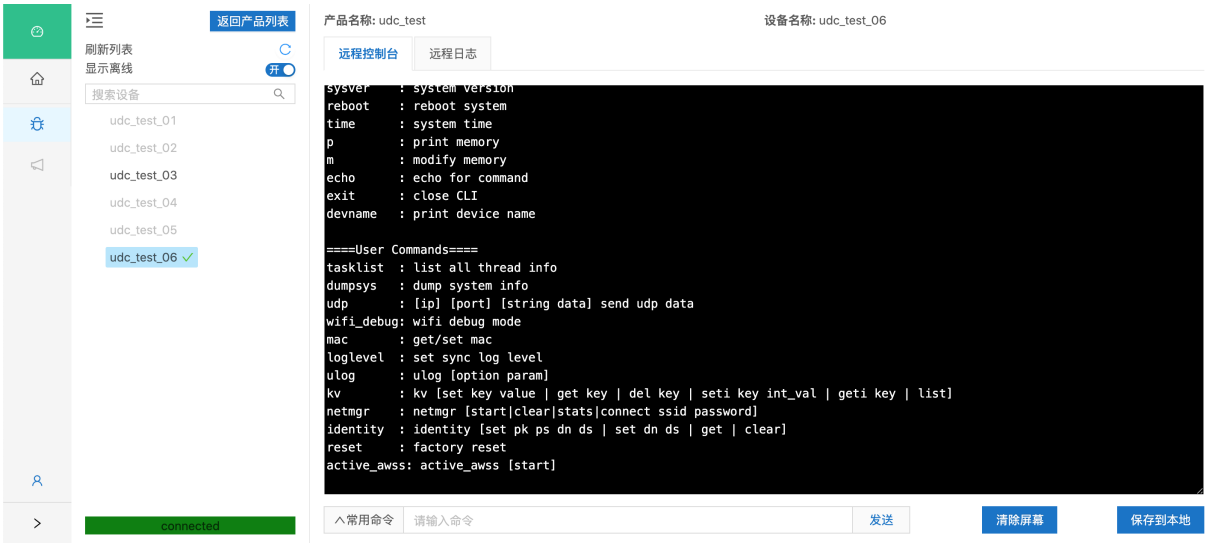
- 点击“添加产品”按钮；
- 在弹出的对话框中选择你想要远程调试运维的产品；
- 阅读服务协议并勾选同意；
- 点击确定按钮，即可完成对所选产品的调试运维功能的添加。

步骤3. 点击“调试运维”按键，进入产品的调试运维页面



步骤4. 在调试运维页面，点击选择你想操作的设备来打开其运维页面

步骤5. 在运维页面做具体的调试运维，如： 远程与设备做命令行交互远程查看分析设备日志



5.3. 操作指南

5.3.1. 远程与设备做命令行交互

本文档讲述如何通过调试运维控制台远程与设备做命令行交互。

背景信息

AliOS Things通过CLI模块为设备提供了命令行交互的能力。通过命令行交互，我们可以完成如：

- 查看分析设备状态
- 调整设置设备参数
- 操作控制设备行为

等调试运维操作。

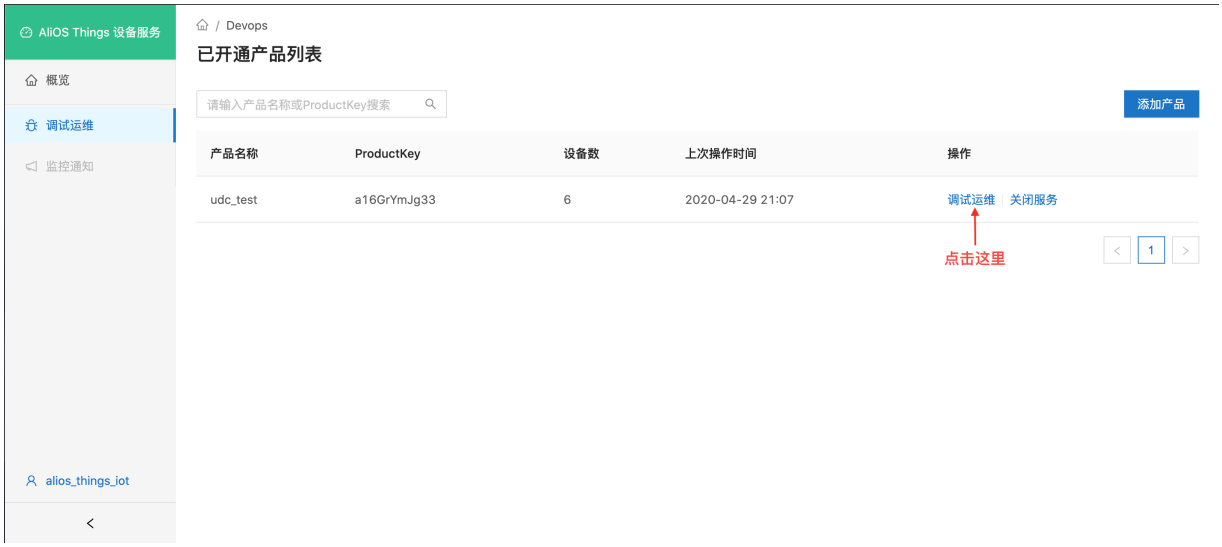
设备的命令行交互功能通常是通过串口透出的。开发者需要将设备的串口接到电脑上才能进行操作。通过串口交互通常会给开发者调试运维设备（尤其是运程设备）带来很多不便。为了解决这个问题，AliOS Things通过uAgent模块+运程调试运维控制台的配合，把设备的命令行交互的能力透出到了网页上。这种方式能极大地方便开发者远程与设备做命令行交互。

前置条件

- 请确保已开启设备的远程调试运维功能
- 请确保已 开通远程调试运维服务，并且已经开通设备所属产品的调试运维功能

远程命令行交互操作步骤

步骤1. 访问调试运维服务页面，点击需要运维产品的“调试运维”按钮，进入其调试运维页面



步骤2. 在左边的设备列表中，找到你要操作的设备，单击打开其操控页面

Tips: 如果设备数量较多，可以在设备列表上方的搜索框里输入设备名称，来快速找到你要操作的设备。

步骤3. 熟悉打开的设备远程控制台交互页面布局

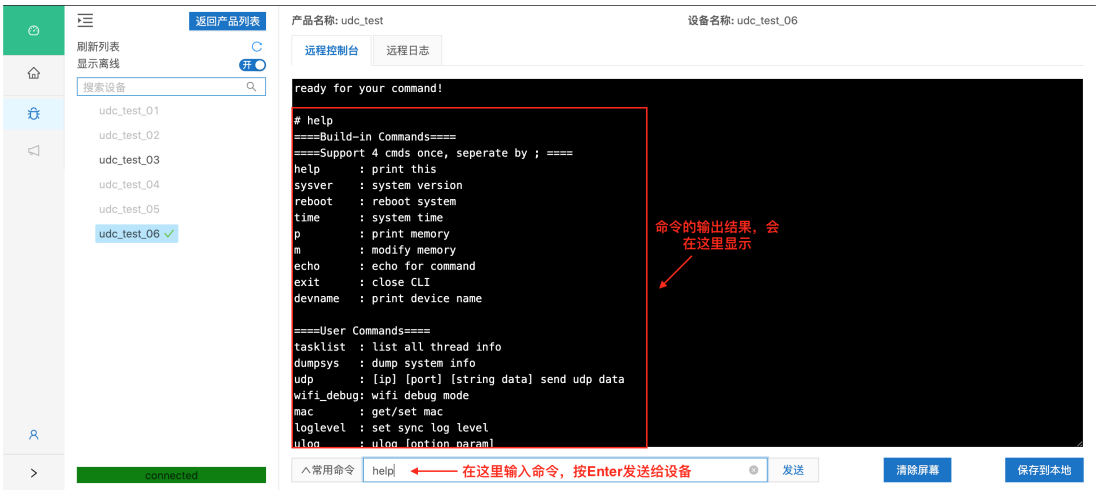
远程控制台交互页面提供与设备的命令行交互功能，其的布局如下图所示。页面主要包括三个功能区：

- 交互日志显示区：显示用户与设备所有的命令行交互日志
- 交互命令输入区：包括命令输入框 和 命令快捷输入框，提供用户输入命令并发给设备的功能
- 交互日志操作区：提供按钮来操作交互日志，包括“清除屏幕”按钮和“保存到本地按钮”。清除屏幕按钮能清空日志显示；保存到本地按钮能将交互日志下载为文件保存到本地，以方便用户做进一步分析。

步骤4. 输入命令与设备交互

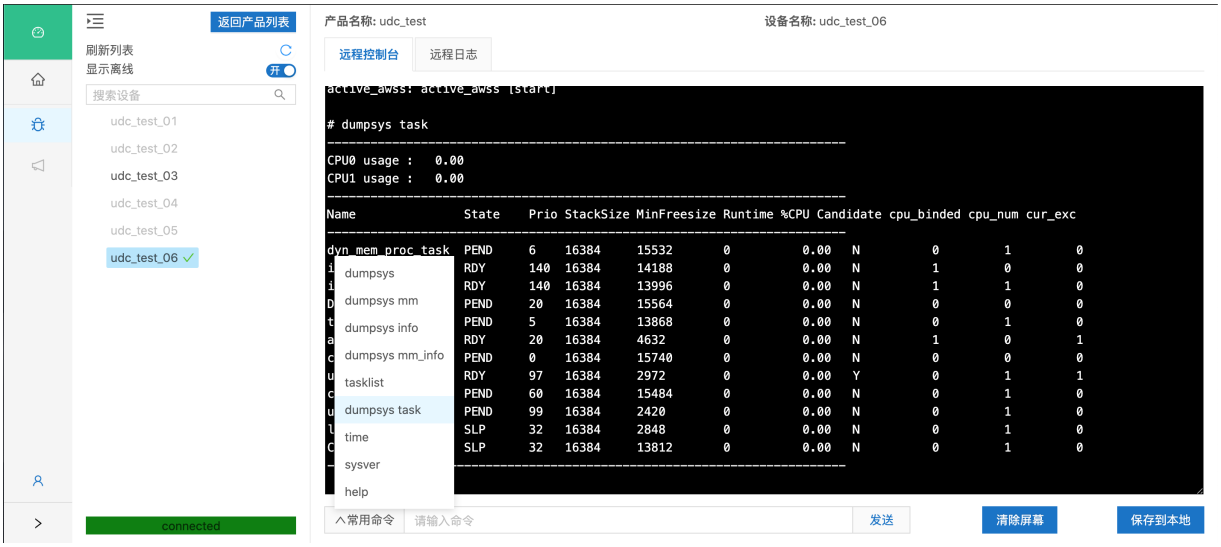
在命令输入框输入命令，按Enter（或者点击“发送”按钮）可以发送给设备。命令的输出结果，会显示到页面的交互日志显示区。

Tips: 在命令输入框，按上/下按键，可以快捷选择输入历史命令哦

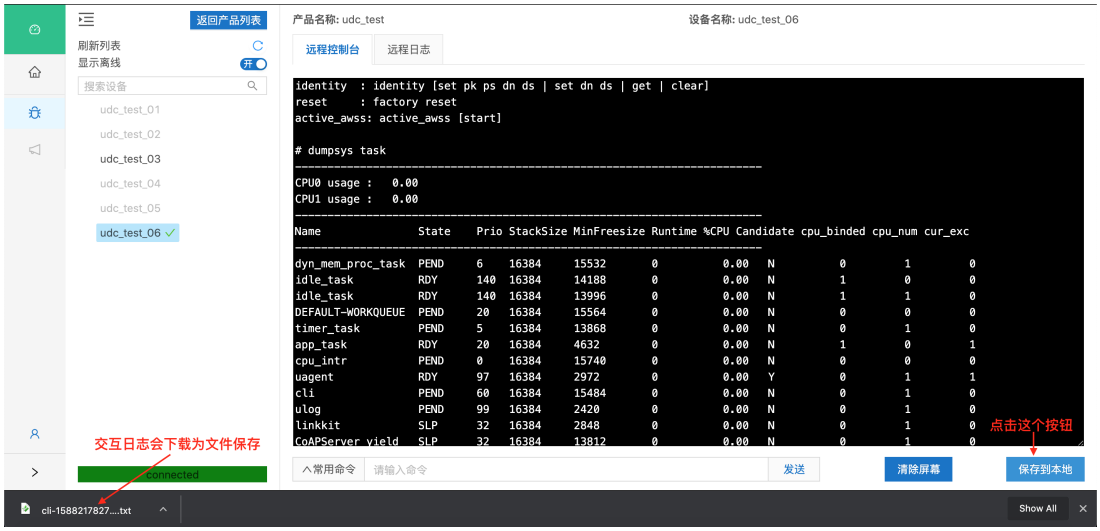


步骤5. 点击常用命令输入框，可以快捷输入频繁使用的命令

Tips：常用命令列表，会随着你输入的命令动态更新哦，永远显示您最频繁使用的命令



步骤6. 点击“保存到本地”按钮，可以将您与设备交互的所有日志，以文件形式下载保存到本地，以便做进一步分析



步骤7. 点击“清除屏幕”按键，可以清除交互日志显示去的所有日志

5.3.2. 远程查看分析设备日志

本文档讲述如何远程查看分析设备日志。

背景信息

通过日志来调试分析诊断设备的行为，是我们解决设备问题的基本方式。设备的日志通常是由串口输出的。查看设备日志需要把设备串口连接到开发调试的电脑上。对于已经规模生产，散步到各个地方的IoT设备来说，串口日志分析几乎不可能完成-开发者很难将海量设备的串口都实时监控起来。

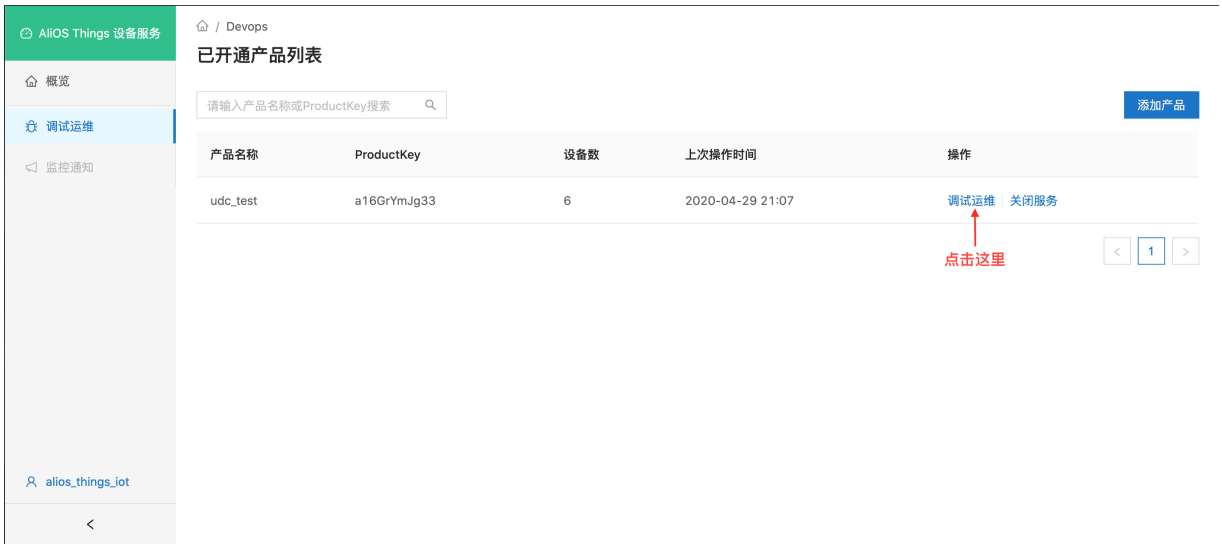
为了解决这种问题，AliOS Things通过uAgent模块+运程调试运维控制台的配合，把设备的日志按需上传到了云上，并提供网页帮助开发者分析查看日志。这种方式能方便开发运维人员大规模远程地抓取设备日志，快速高效地分析解决问题。

前置条件

- 请确保已开启设备的远程调试运维功能，并且在设备端使能了日志上传配置项
- 请确保已 开通远程调试运维服务，并且已经开通设备所属产品的调试运维功能

远程查看分析设备日志操作步骤

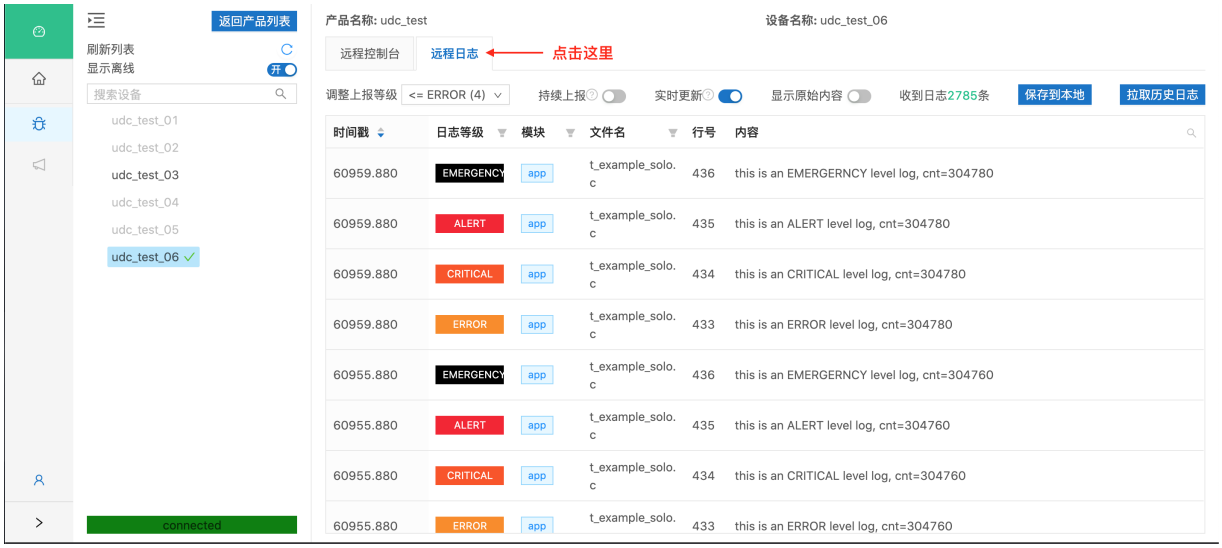
步骤1. 访问调试运维服务页面，点击需要运维产品的“调试运维”按钮，进入其调试运维页面



步骤2. 在左边的设备列表中，找到你要操作的设备，单击打开其操控页面

Tips: 如果设备数量较多，可以在设备列表上方的搜索框里输入设备名称，来快速找到你要操作的设备。

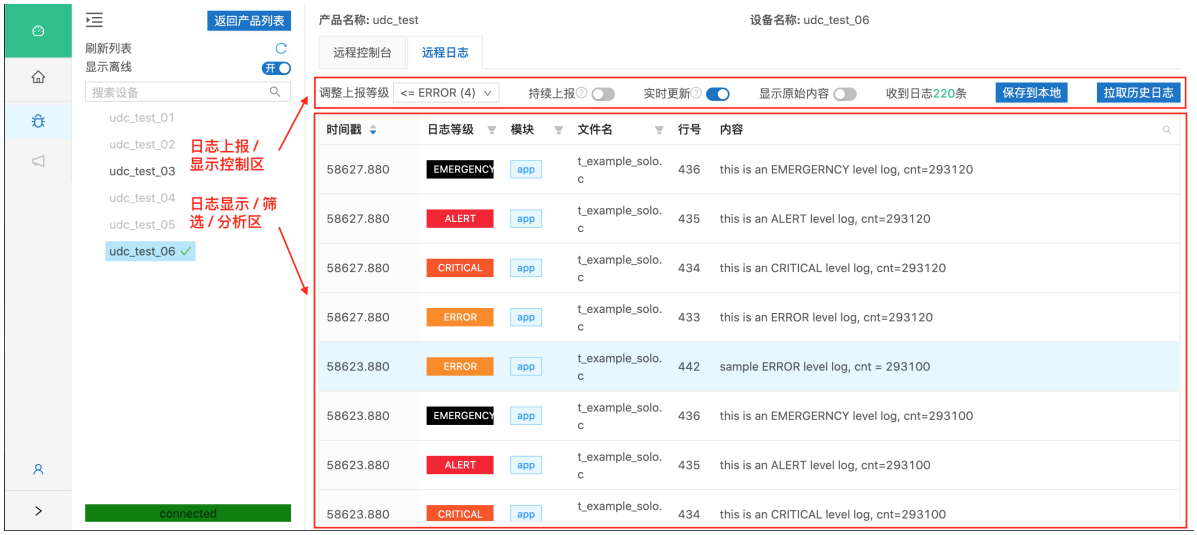
步骤3. 在操控页面中，点击“远程日志”选项卡，进入远程日志查看分析页面



步骤4. 熟悉远程日志查看分析页面

远程日志页面主要有两个功能区：

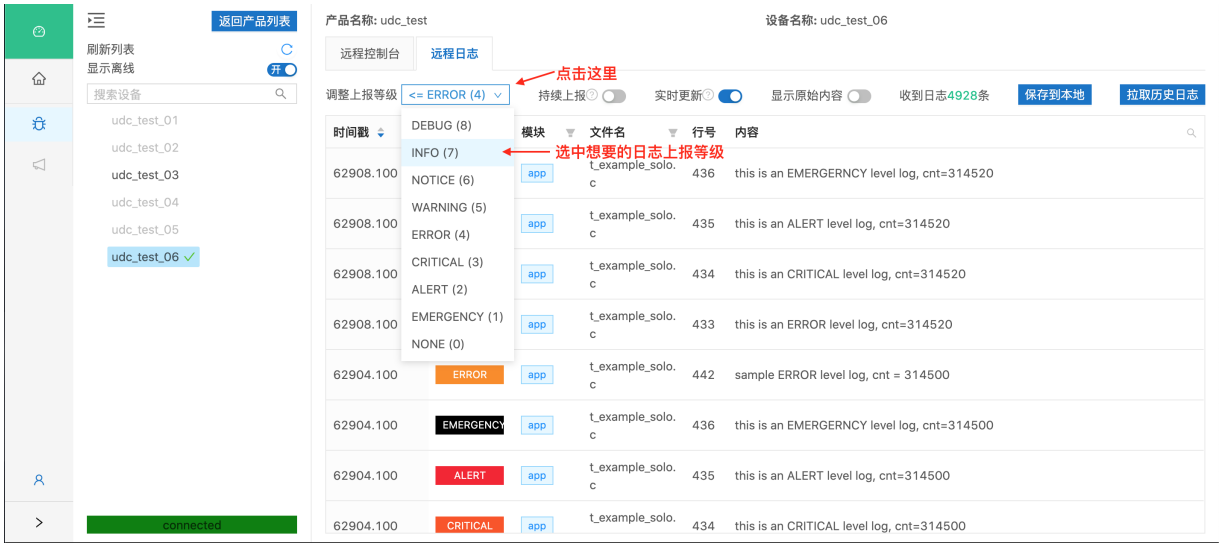
- 日志上报/显示控制区：控制设备的日上报行为；设置日志的显示方式；操作日志的拉取/下载/保存
- 日志显示/筛选/分析区：提供日志查看/筛选/分析等功能



步骤5. 在页面操控区，控制日志上报 / 显示/保存/拉取等行为

操作A. 调整日志上报等级

点击“调整上报等级”选择框，可以在弹出的下拉列表中选择期望的设备日志上报筛选等级。比如我们选中如下图所示的INFO等级，则设备会上报等级<=INFO（INFO/NOTICE/WARNING/ERROR/CRITICAL/ALERT/EMERGENCY）的所有日志。选择的上报等级数值越大，设备会上报越详细的日志。

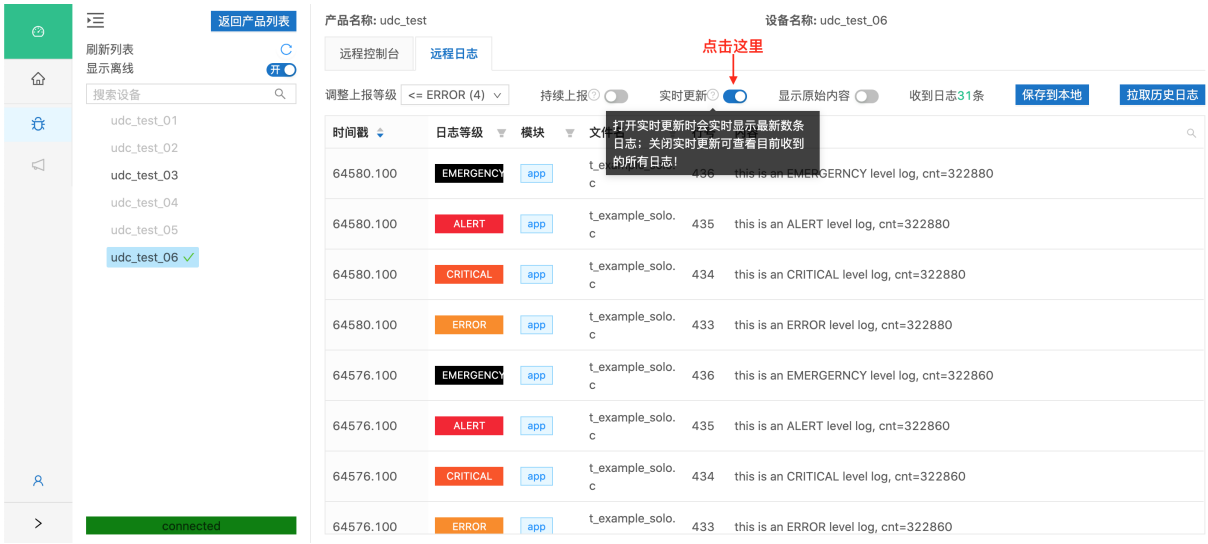


操作B. 控制日志持续上报行为

点击“持续上报”开关，可以控制设备是否持续上报日志到运维服务云后台。默认情况下，设备仅在远程日志页面打开时才会上传日志。在远程日志页面退出后，设备即会停止日志上传。如果需要设备持续地上传日志，以便将来拉取分析，请点击开启持续上报功能。开启持续上报后，请注意留意设备的消息流量消耗。

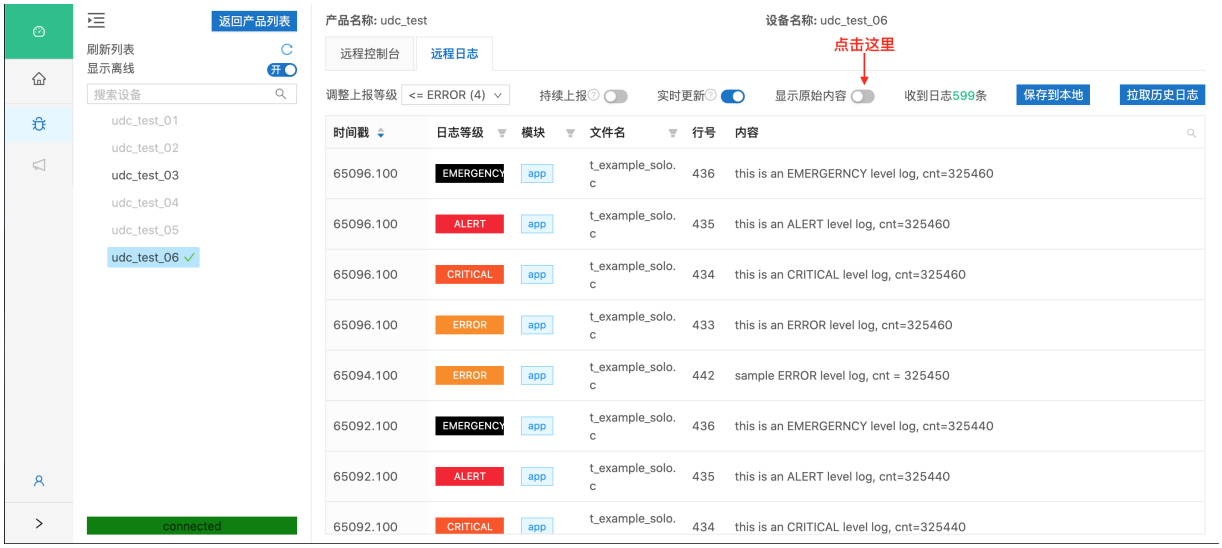
操作C. 控制日志实时更新显示行为

点击“实时更新”开关，可以控制日志的实时刷新行为。受限于浏览器渲染性能，在开启实时更新显示时，页面仅显示收到的最新数条日志，以保证显示的流畅性。如果用户需要查看分析目前收到的所有日志，可以关闭实时更新。关闭实时更新后，最新收到的日志将不会更新到页面。

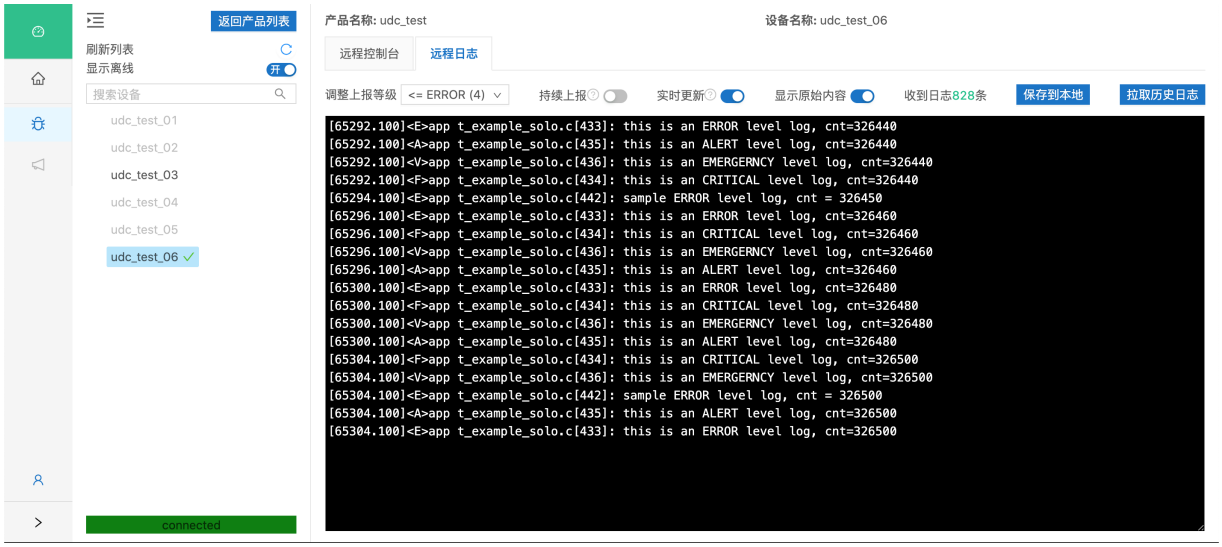


操作D. 控制日志显示方式

点击“显示原始内容”开关，可以切换日志的显示方式。默认情况下，页面会解析收到的日志，并以下图所示的易读的方式显示出来。

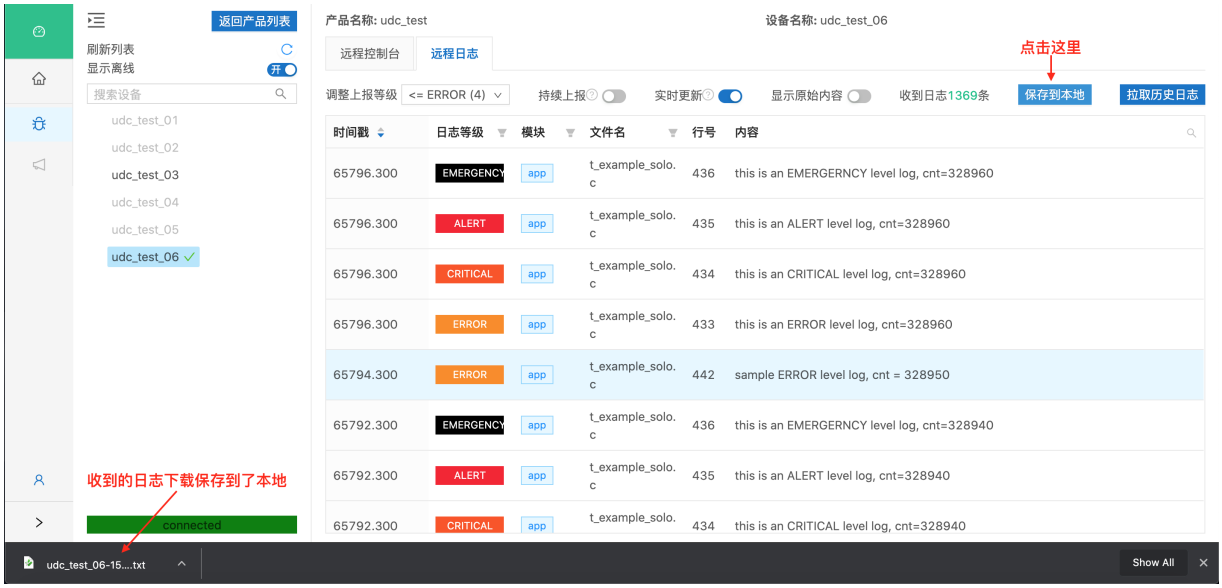


如果开发者更习惯阅读分析原始日志，可以打开“显示原始日志”开关，看到如下图所示的原始日志显示。



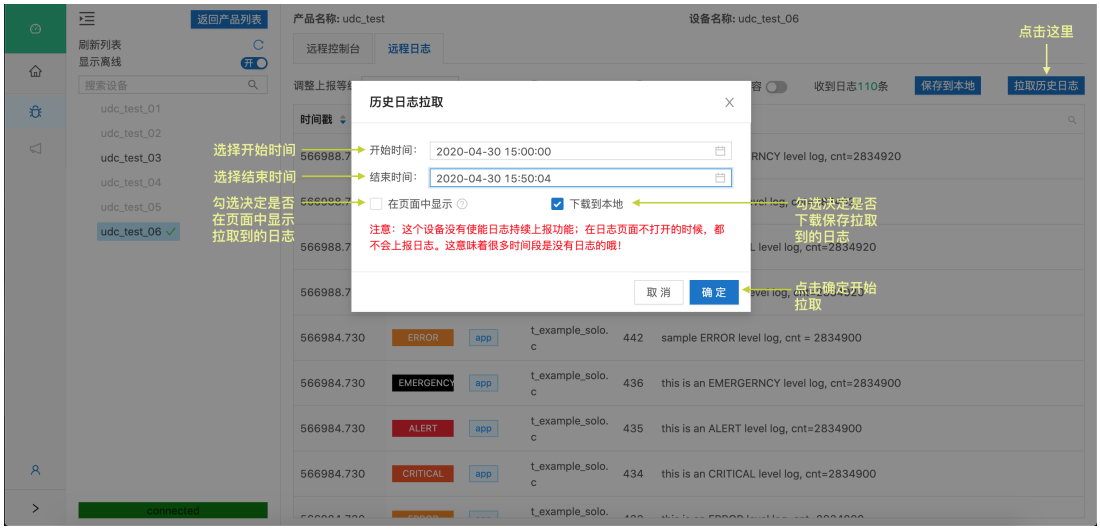
操作E. 下载保存收到的日志到本地

点击“保存到本地”按键，可以将页面目前收到的所有日志下载保存到本地，以便用其它工具做进一步分析。



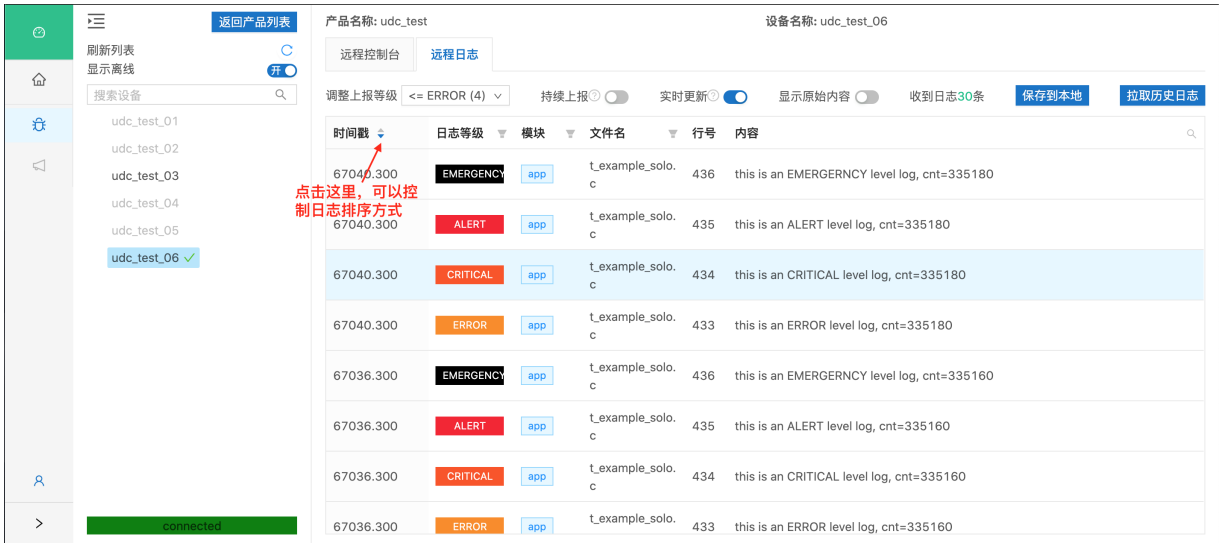
操作F. 拉取历史日志

点击“拉取历史日志”按键，可以拉取设备上传的历史日志。您可以拉取选定时间段的日志。对于拉取的日志，您可以选择在当前页面显示分析 或者 保存下载到本地做进一步分析。

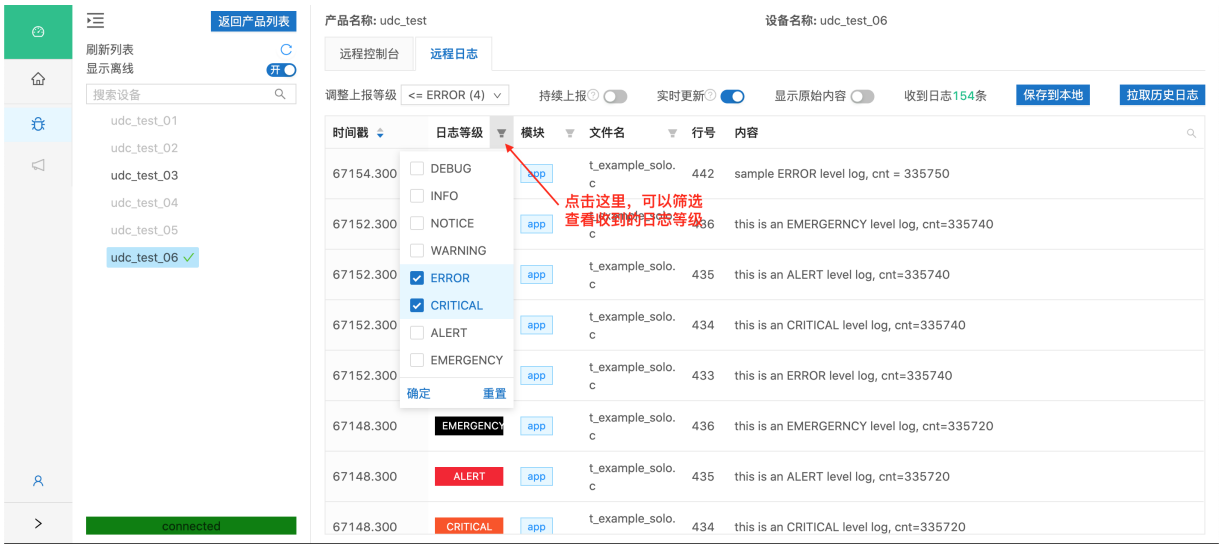


步骤6. 在页面日志显示分析区，查看筛选分析设备日志

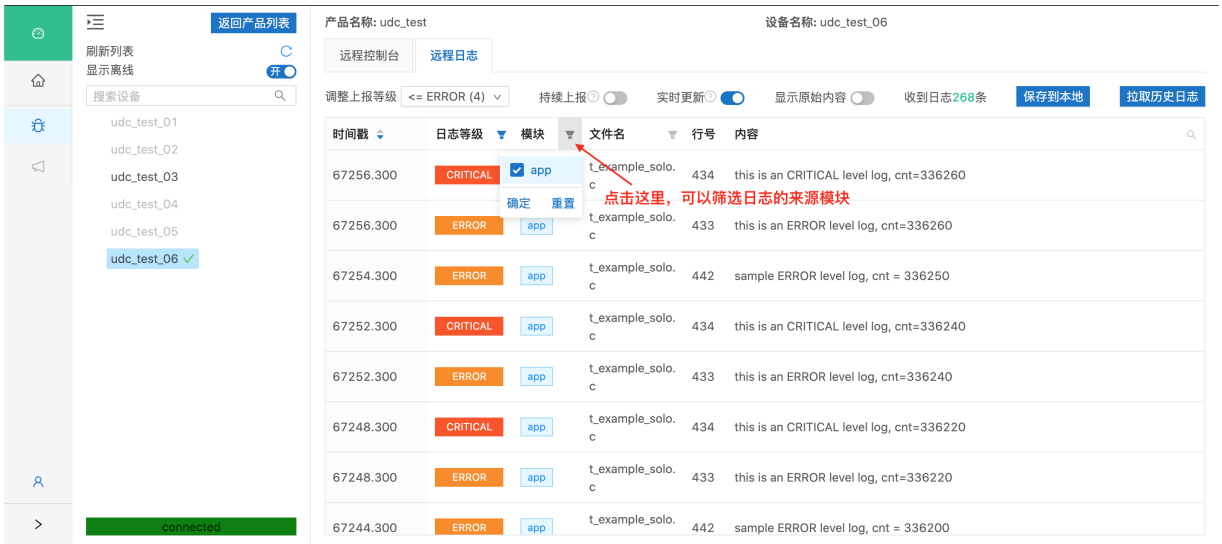
操作A. 控制日志排序方式



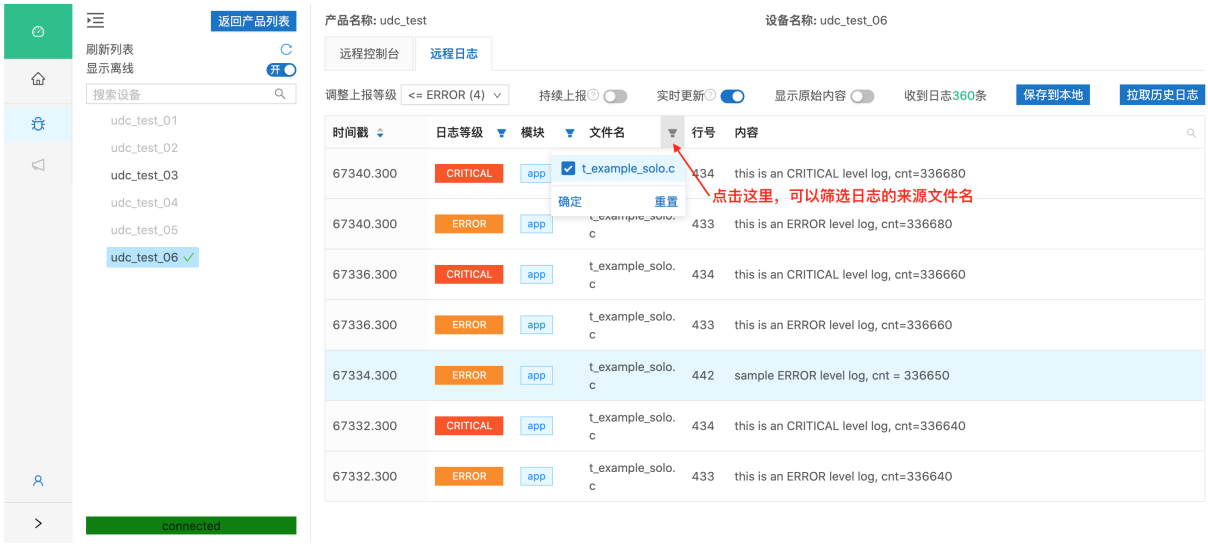
操作B. 筛选日志等级



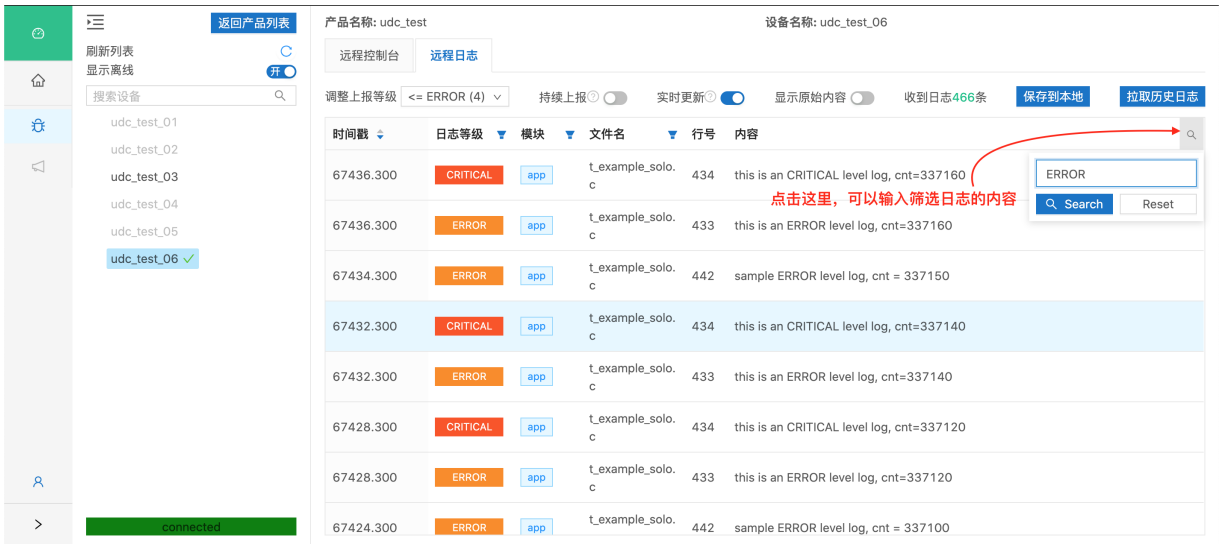
操作C. 筛选日志来源模块



操作D. 筛选日志来源文件名



操作E. 筛选日志内容



6.应用笔记

6.1. 维测模块及维测解析工具使用介绍

维测即异常处理，在不同体系架构文档中对“异常”的定义存在不同，“异常”即Exception，在MIPS与ARM中的把interrupt、trap也都归类为Exception。而本文中的“异常”指的是代码出现了致命错误，导致系统异常崩溃而无法运行下去的状况。产生的直接原因可能是执行非法指令、访问非法内存等，间接原因可能是内存申请失败、代码跑飞、内存被踩、callback函数未注册等等。

维测能解决什么问题

简单的说，维测的价值在于可以缩短bug定位时间。如果一个bug出现导致系统异常后，用户可以不用连仿真器、不用加打印、不用打开gdb单步调试的情况下，可以快速找到bug原因，或者帮助用户指出可能的异常点，进而修复节省开发时间。

举例说明：

1. 代码中访问了非法内存（比如：在不可写的地址处写了数据）导致系统崩溃，维测模块可以记录访问非法内存时的pc值，告诉用户挂在了哪一行；
2. 代码跑飞了(pc=0)，维测模块记录了函数调用的栈，并根据栈向上回溯可以找到A->B->C的函数调用过程；
3. 用户内存申请时malloc 失败，维测模块可以记录用户此时申请了多少内存导致了内存池不够、此时还有多少字节内存可以供申请、用户是在哪个任务中申请的内存、从系统启动开始内存的申请情况等信息，帮助用户查看是否有组件申请了过大内存但没有释放等内存泄漏的情况；
4. 已经有明显的踩内存现象，但是无法具体定位踩内存因，只知道一段内存被非法改写，复现现象不一致，定位相当耗时。维测模块可以提供接口，设置该段内存的属性为不可访问，从而制造memory访问异常，结合异常现场打印，快速定位踩内存的元凶；
5. 在一次长达数小时的压测中（如linkkit app、蓝牙配网等压测）出现了系统内存缓慢释放后内存耗尽、某个任务栈被踩等问题导致了系统奔溃，问题长时间压测才复现，维测模块可以在bug首次出现但系统还在正常运行的时候（内存第一次泄漏、任务栈第一次被踩等），主动触发异常告警，提醒用户系统存在隐患，并打印出异常现场信息供分析，不用等到数小时后才复现问题。

还有很多问题可以通过维测能力来帮助定位，更高级的维测功能也在持续开发中。

AliOS Things 维测组件提供的能力

AliOS-Things提供了2部分维测能力：

1. 维测模块 --- 组件名为**debug**，组件位于core/debug路径下，功能是设备端的异常接管和异常log打印输出。
2. 维测解析工具 --- 工具名为**core_dump**，工具位于components/utility/debug_tools路径下，是PC端基于Python的解析异常log的工具，生成更为详细的异常报告。

当系统异常后，可由AliOS-Things接管异常处理。维测组件会打印丰富的异常现场信息，协助用户定位问题。

1. AliOS Things维测模块提供的异常现场信息（设备端能力）

```
!!!!!!!!!! Exception !!!!!!!!!!
========== Regs info ==========      异常现场寄存器信息
PC    0x401111AA
PS    0x00060130
A0    0x801111CA
A1    0x3FFDC830
A2    0x00000017
A3    0x3FFC2C18
A4    0xFFFFFFFF
A5    0x00000287
```

```
A6    0x00000002
A7    0x00000044
A8    0x00000001
A9    0x12345678
A10   0x00000001
A11   0x3FFCACFC
A12   0x00000001
A13   0x3FFDC510
A14   0x3FFDC510
A15   0x00000001
SAR   0x00000001
EXCCAUSE 0x0000001D
EXCVADDR 0x12345678
========== Stack info ==========      异常现场栈信息
stack(0x3FFDC830): 0x03020100 0x00000044 0x00000002 0x00000000
stack(0x3FFDC840): 0xFFFFFFFF 0x00000287 0x00000002 0x00000044
stack(0x3FFDC850): 0x4008C1E4 0x3FFDC880 0x3FFC09C0 0x00000000
stack(0x3FFDC860): 0x8008D555 0x3FFCE680 0x3F4010C3 0x4008D50C
stack(0x3FFDC870): 0x00000000 0x3FFDC8A0 0x00000000 0x00000000
stack(0x3FFDC880): 0x00000000 0x00000000 0x00000000 0x00000000
stack(0x3FFDC890): 0x00000000 0x00000000 0x00000000 0x00000000
stack(0x3FFDC8A0): 0x00000000 0x00000000 0x3FFDC8AC 0x00000000
stack(0x3FFDC8B0): 0x00000000 0x00000000 0x00000000 0x00000000
stack(0x3FFDC8C0): 0x00000000 0x00000000 0x00000000 0x00000000
stack(0x3FFDC8D0): 0x00000000 0x00000000 0x00000000 0x00000000
stack(0x3FFDC8E0): 0x00000000 0x00000000 0x00000000 0x00000000
stack(0x3FFDC8F0): 0x00000000 0x00000000 0x00000000 0x00000000
stack(0x3FFDC900): 0xFEFEFEFE 0x40111C80 0x3FFDA8F0 0x00000320
stack(0x3FFDC910): 0x00000001 0x00000012 0x00000000 0x00000000
stack(0x3FFDC920): 0x00000000 0x00000000 0x00000000 0x00000000
========== Call stack ==========
======= Call stack Begin =======      栈回溯信息，可以得出函数调用过程
backtrace : 0x401111AA
backtrace : 0x401111C7
backtrace : 0x4008D521
backtrace : ^task entry^
======== Call stack End ========
========== Heap Info ==========      内存信息，可以看出内存申请了多少，还剩多少
-------------------------------------------------------------
[HEAP]| TotalSz   | FreeSz    | UsedSz    | MinFreeSz |
   | 0x0002D418 | 0x0001C868 | 0x00010BB0 | 0x0001A4D0 |
-------------------------------------------------------------
[POOL]| PoolSz    | FreeSz    | UsedSz    | BlkSz     |
   | 0x00002000 | 0x00001B60 | 0x000004A0 | 0x00000020 |
-------------------------------------------------------------
========== Task Info ==========      任务状态信息，可以看出任务栈是否过小
-------------------------------------------------------------------------------
TaskName        State   Prio     Stack     StackSize (MinFree)
-------------------------------------------------------------------------------
dyn_mem_proc_task   PEND    0x00000009 0x3FFC792C 0x00000400(0x000002C0)
idle_task        RDY     0x0000003D 0x3FFC7D50 0x00000400(0x0000029C)
DEFAULT-WORKQUEUE   PEND    0x00000014 0x3FFC8230 0x00000C00(0x00000ADC)
timer_task       PEND    0x00000005 0x3FFC6B40 0x00000C00(0x00000ACC)
esp_timer        PEND    0x00000006 0x3FFCAE80 0x00001000(0x00000E4C)
ipc0           PEND    0x00000002 0x3FFCC0B0 0x00000400(0x000002CC)
tcp/ip          PEND    0x00000007 0x3FFCED68 0x00000C00(0x000009E4)
eventTask        PEND    0x0000000A 0x3FFD0798 0x00001200(0x00000B5C)
```

```
wifi            PEND    0x00000004 0x3FFD2200 0x00000E00(0x000005EC)
main             RDY    0x00000020 0x3FFDA900 0x00002000(0x00001AD4)
cli             PEND    0x0000003C 0x3FFDCC40 0x00000800(0x0000056C)
========== Queue Info ==========     queue使用信息
-------------------------------------------------------
QueAddr   TotalSize  PeakNum   CurrNum   TaskWaiting
-------------------------------------------------------
======== Buf Queue Info ========     buf queue使用状态，可以看出哪个任务在bufqueue消息
-----------------------------------------------------------------
BufQueAddr TotalSize  PeakNum   CurrNum   MinFreeSz  TaskWaiting
-----------------------------------------------------------------
0x3FFC67C0 0x000001E0 0x00000000 0x00000000 0x000001E0 timer_task
0x3FFCEC78 0x00000040 0x00000000 0x00000000 0x00000040 tcp/ip
0x3FFD0188 0x00000600 0x00000000 0x00000000 0x00000600 eventTask
0x3FFD1B78 0x00000640 0x00000000 0x00000000 0x00000640 wifi
========== Sem Info ==========    信号量使用状态，可以看出哪个任务在等待信号量
---------------------------------------------
SemAddr   Count    PeakCount  TaskWaiting
---------------------------------------------
0x3FFC6798 0x00000000 0x00000000 dyn_mem_proc_task
0x3FFC8208 0x00000000 0x00000000 DEFAULT-WORKQUEUE
0x3FFCAE48 0x00000000 0x00000000 esp_timer
0x3FFCC040 0x00000000 0x00000000
0x3FFCC078 0x00000000 0x00000000 ipc0
0x3FFCE920 0x00000001 0x00000001
0x3FFCE958 0x00000000 0x00000000
0x3FFCE990 0x00000000 0x00000000
0x3FFCE9C8 0x00000001 0x00000001
0x3FFCEA00 0x00000000 0x00000001
0x3FFCEBC8 0x00000000 0x00000000
0x3FFCEC00 0x00000000 0x00000000 cli
0x3FFCFA18 0x00000000 0x00000000
0x3FFCFA50 0x00000001 0x00000001
0x3FFD21C8 0x00000000 0x00000001
0x3FFDA858 0x00000000 0x00000000
!!!!!!!!!! dump end  !!!!!!!!!!
```

2. 维测解析工具的使用（PC 端能力）

维测解析工具core dump在tools/debug_tools路径下，在路径下有详细的使用方法介绍README，这里简单说明使用方法：

```
python coredump.py log helloworld@esp32devkitc.elf
```

其中：log --- 上面系统异常时的串口打印输出，可以拷贝到一个文件中，文件名任意，这里取名为log.elf --- 此时系统对应的elf文件

维测解析工具输出如下（部分）：

1. 异常原因可能解释：

```
********** Alios Things Exception Core Dump Result **********
========== Show Exc Regs Info ==========
EXCCAUSE : 0x0000001D
EXCADDR : 0x12345678
A load/store referenced a page mapped with an attribute that does not permit
Potential reasons:
1. Access to Cache after it is turned off
2. Wild pointers
```

可见是访问了非法内存。

2. 栈回溯 backtrace，即可清晰看到发生异常的函数调用过程： app_entry -- > application_start ---
>test_panic

并且指出了函数代码的路径和行号。如图：

```
======= Call stack Begin =======
backtrace : 0x401111AA
test_panic at /home/yx170385/code/aos/app/example/helloworld/helloworld.c:20
backtrace : 0x401111C7
application_start at /home/yx170385/code/aos/app/example/helloworld/helloworld.c:37
backtrace : 0x4008D521
app_entry at /home/yx170385/code/aos/platform/mcu/esp32/bsp/entry.c:30
backtrace : ^task entry^
```

3. 指出异常发生在哪个任务中

```
========== Show Task Info ==========
Crash in task : main
```

为了方便用户更好的使用，维测工具也在不断的升级中。

当前主线版本支持维测功能的情况

从AliOS Things 2.0开始，维测功能上线。打开维测会使得ROM大小增加2K左右，RAM基本无变化。

在当前主线上，默认打开维测功能的板子有以下几款“精品芯片”计划的板子：

1	developerkit	是	是	
2	stm32f429	是	是	网关应用
3	esp32	是	是	智能语音应用
4	esp8266	是	是	
5	pca10040(nrf52832)	是	是	手环
6	amebaz_dev (RTL8710)	是	是	连接应用
7	mk3080 (RTL8710)	是	是	连接应用
8	uno-91h（RDA5981x）	是	是	连接应用
9	bk7231u	是	是	智能语音应用

打开维测功能的方法（出临时版本使用）

1. **在2.1及后续版本上打开维测的方法：**

当前的ARM和Xtensa架构的芯片已经完成基本维测功能的适配，覆盖了绝大多数的板子。维测功能默认是关闭的，如果需要打开，采用menuconfig配置的方法，步骤如下：

1. aos make menuconfig
2. 选择kernel --- Debug & Cpuusage Support
3. 将“Enable debug panic feature” 和 “Enable stack backtrace feature”选中，保存退出
4. aos make

请注意： 当前最新版本3.1上支持了多种编译形态，即：

aos make---- 默认版本即 release版本，有基本维测功能，异常后自动重启，不会卡住

aos make xxxx **BUILD_TYPE=inspect** ---- 为全维测版本，增加了异常后cli 接管、栈溢出检测等功能，并且不会重启，调试的时候请选择这个版本

1. **在rel_2.0 上打开维测的方法：**

编辑 kernel/rhino/debug/include/k_dftdbg_config.h将下面2个宏的值改为1重新编译

打开维测的宏后，编译出来的debug组件有3K多的字节，才是正常的，如图所示：

维测定位方法及API

在大多数情况下，使用维测组件及维测解析工具即可解决问题，维测接口api只在内部调试定位问题时使用，暂时无 aos 对外api，这里也总结了4个维测接口，用户可根据实际情况使用。

API 列表

debug_mm_overview()	内存堆信息状态显示
debug_task_overview()	任务状态显示
debug_backtrace_now()	调用栈回溯显示
debug_memory_access_err_check(addr, size, mode)	通过mpu监控一段内存，定位踩内存-栈溢出等场景使用

API 详情

debug_mm_overview()定义描述

描述	内存堆信息状态显示
入参	int *print_func -- 打印函数，可直接输入NULL
返回值	无

此函数会打印堆的相关统计，如下所示：

```
========== Heap Info ==========
------------------------------------------------------------
[HEAP]| TotalSz   | FreeSz    | UsedSz    | MinFreeSz |
      | 0x0004A838 | 0x00047E50 | 0x000029E8 | 0x00047E50 |
------------------------------------------------------------
[POOL]| PoolSz    | FreeSz    | UsedSz    | BlkSz     |
      | 0x00002000 | 0x00001E00 | 0x00000200 | 0x00000020 |
------------------------------------------------------------
```

上面统计分成两部分，HEAP与POOL。HEAP是总的统计，POOL是HEAP的一部分。

HEAP与POOL的区别是，当用户使用

```
aos_malloc(size)
```

来分配内存的时候，size若小于32字节（RHINO_CONFIG_MM_BLK_SIZE，在k_config.h中定义），malloc会在POOL上固定分配32字节内存，反之则在HEAP上分配用户定义size的内存。

HEAP中的内容含义：

- TotalSz，堆的总大小。
- FreeSz，当前堆的空闲大小。
- UsedSz，当前堆的使用量，即UsedSz = TotalSz - FreeSz。
- MinFreeSz，堆空闲历史最小值，即TotalSz - MinFreeSz 便是堆历史使用量峰值。

出异常时，可以利用该信息大致判断堆是否出现空闲内存不足的问题。

调用示例

用户代码中，通过debug_mm_overview(NULL)接口可以主动打印heap消耗情况。这里给出一个周期性打印heap消耗的方式：

```
void krhino_tick_hook(void)
{
  //添加下面的代码
  static int s_cnt;
  if ( s_cnt++ % RHINO_CONFIG_TICKS_PER_SECOND == 0 )
  {
    debug_mm_overview(NULL);
  }
}
```

使用krhino_tick_hook()钩子函数需要开启 `RHINO_CONFIG_USER_HOOK` 。

这样就可以让系统每秒都打印一次heap占用统计，来判断是否出现空闲内存不足、内存泄漏等问题

debug_task_overview()

定义描述

描述	任务状态信息显示
入参	无
返回值	无

此函数会打印堆的相关统计，示例如下所示：

```
-------------------------------------------------------------------------------
TaskName        State   Prio    Stack    StackSize (MinFree)
-------------------------------------------------------------------------------
dyn_mem_proc_task   PEND    0x00000006 0x200047A8 0x00000400(0x00000328)
idle_task       RDY     0x0000003D 0x200043F8 0x00000320(0x00000288)
DEFAULT-WORKQUEUE   PEND    0x00000014 0x200036D4 0x00000C00(0x00000B44)
timer_task      PEND    0x00000005 0x20003154 0x000004B0(0x000003B4)
aos-init        PEND    0x00000020 0x20000EE0 0x00001800(0x000014D8)
cli         PEND    0x0000003C 0x20008CB8 0x00000800(0x00000688)
```

上面打印出系统当前共有dyn_mem_proc_task、idle_task、default-workqueue、timer_task、aos_init、cli共6个任务每个任务的当前状态、优先级、任务栈以及栈的使用情况，若MinFree显示为0，则该任务很可能出现栈溢出情况，建议修改任务创建时的栈大小。

调用示例可将debug_task_overview()根据需要加在代码中，观察任务信息。若使能了cli，cli中也有类似的打印。

debug_backtrace_now()

定义描述

描述	调用栈过程显示
入参	无
返回值	无

调用示例

用户可主动调用该函数，特别是在导致系统异常的怀疑点处调用，当代码执行到该函数时，会打印出栈回溯信息，示例如下所示：

```
========== Call stack ==========
......
backtrace : 0x08009B2A
backtrace : 0x0800A06C
backtrace : ^task entry^
```

栈回溯结果可以从下向上关注，最底部为^task entry 表示异常发生在任务中，为 interrupt^表示异常发生在中断处理中。之后根据编译工具链提供的arm-none-eabi-addr2line工具，输入地址与编译出的elf文件，可以找到对应C代码的位置。 举例：

```
d:\git\aos\out\helloworld@developerkit\binary>arm-none-eabi-addr2line -e helloworld@developerkit.elf 0x0800A06C
D:\git\aos/kernel\rhino\core/k_idle.c:59
d:\git\aos\out\helloworld@developerkit\binary>arm-none-eabi-addr2line -e helloworld@developerkit.elf 0x08009B2A
D:\git\aos/kernel\rhino\core/k_tick.c:13
```

根据以上C代码信息，分析可能的错误。

debug_memory_access_err_check()

注意： 只使用于 ARM cortex M，并且硬件带有MPU的处理器

定义描述

描述	通过mpu监控一段内存，定位踩内存、栈溢出等场景使用

入参	addr_start：需要监控内存的起始地址
	addr_size: 需要监控内存的大小，最小为32字节，最大为2G（注意size需要被addr_start整除）
	mode：访问模式。0 -- 禁止访问；非0 -- 只读访问
返回值	无

调用示例

```
/*踩内存场景*/
extern void debug_memory_access_err_check(unsigned long addr_start, unsigned long addr_size, unsigned int mode);
/*监控0x20008000起始处，大小为0x400(1K字节)的一段内存，设置该内存不可访问
 若该段内存被访问，则直接触发异常*/
debug_memory_access_err_check(0x20008000, 0x400, 0);
```

适用场景：已经明确有明显的踩内存现象，但是无法具体定位踩内存根因，只知道一段内存被非法改写， 复现现象不一致，定位相当耗时。

使用方法： 将这段内存通过上面接口设置地址访问权限，一旦非法访问立即异常，通过解析工具可快速定位。

```
/*栈溢出场景*/
/*在kernel/rhino/k_stats.c 的krhino_stack_ovf_check 中可增加下面代码，定位栈溢出*/
void krhino_stack_ovf_check(void)
{
  cpu_stack_t *stack_start;
  uint8_t    i;
  stack_start = g_active_task[cpu_cur_get()]->task_stack_base;
  for (i = 0; i < RHINO_CONFIG_STK_CHK_WORDS; i++) {
    if (*stack_start++ != RHINO_TASK_STACK_OVF_MAGIC) {
      k_err_proc(RHINO_TASK_STACK_OVF);
    }
  }
  if ((cpu_stack_t *)(g_active_task[cpu_cur_get()]->task_stack) < stack_start) {
    k_err_proc(RHINO_TASK_STACK_OVF);
  }
 /*增加下面的代码*/
 /*set ready task stack_base(32Bytes) access mode for stack ovf*/
 extern void debug_memory_access_err_check(unsigned long addr_start, unsigned long addr_size, unsigned int mode);
 debug_memory_access_err_check((unsigned long)(g_preferred_ready_task[cpu_cur_get()]->task_stack_base), 0x20, 0);
}
```

适用场景： 怀疑任务栈溢出，栈底被破坏。

使用方法： 在任务切换的栈溢出检测中增加上述接口，将栈低开始的32字节通过mpu保护起来，一旦出现栈溢出立即异常，通过解析工具可快速定位。

利用mpu 实现栈溢出的原理可参见下面的ppt:利用MPU实时栈溢出检测方法.pdf

常用技巧

上面几种情况的信息显示，都可以在系统发生异常的时候打印出来，用户也可以主动触发异常，将出现错误时的现场打印出来。为调试时提供帮助。

主动触发异常方法：

以GCC下Cortex-M系列为例，可以通过下面的代码主动触发异常：

```
__asm__ __volatile__("udf 0":::"memory");
```

类似的，通过强行跑到0地址，也可以触发异常。

```
((void (*)())0)();
```

AliOS Things维测能力与竞品的对比

通过分析竞品（freertos、mbed os）的代码和实际测试竞品rtos在同样板子上的维测运行相结合的方法，得出了竞品维测能力的结论：

1. freertos 维测能力几乎为0，没有系统异常后的现场分析。
2. mbed os的panic处理方式如下，也只提供了通用regs，core regs保存，和部分出错信息打印。

6.2. 使用线上的开发板做开发调试

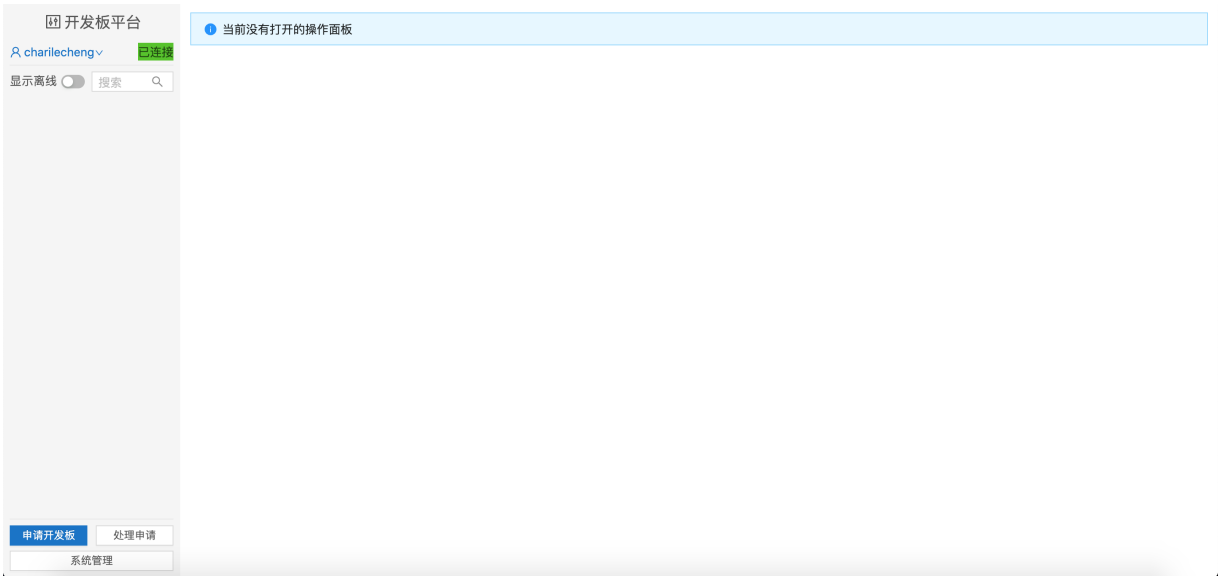
当您手上没有现成可用的开发板时，也可以使用线上的开发板来调试验证您的程序。

我们开发了开发板平台，来为开发者提供线上可用的开发板，以方便大家能随时随地的学习、开发和验证。

开通服务并申请开发板

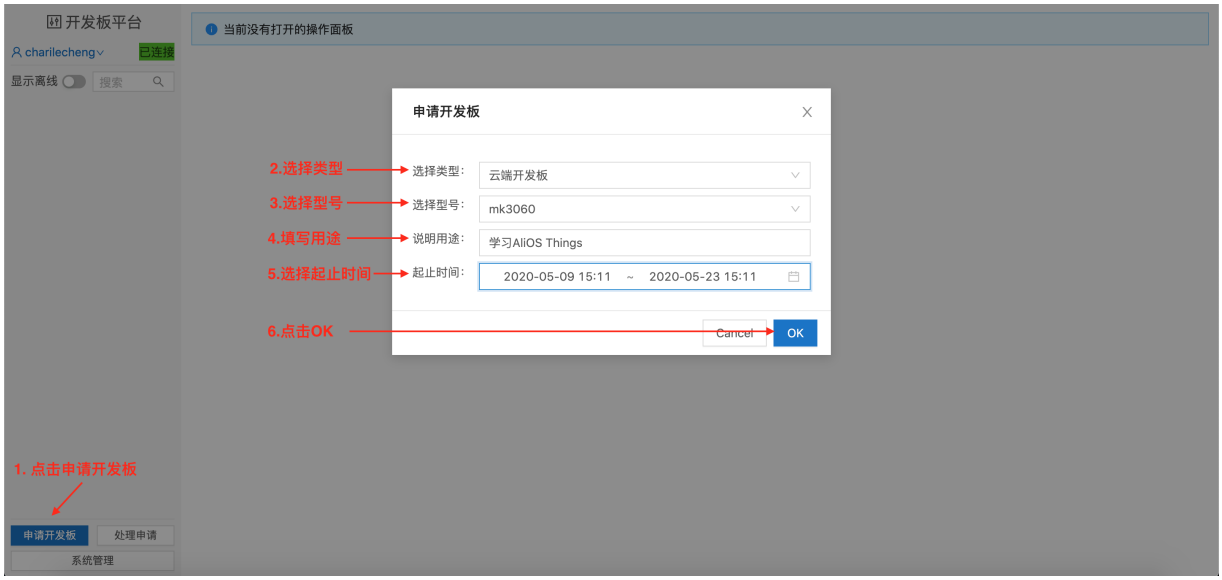
步骤1. 点击访问 开发板平台

NOTE：第一次登陆时，需要阅读并同意服务协议，并开通服务

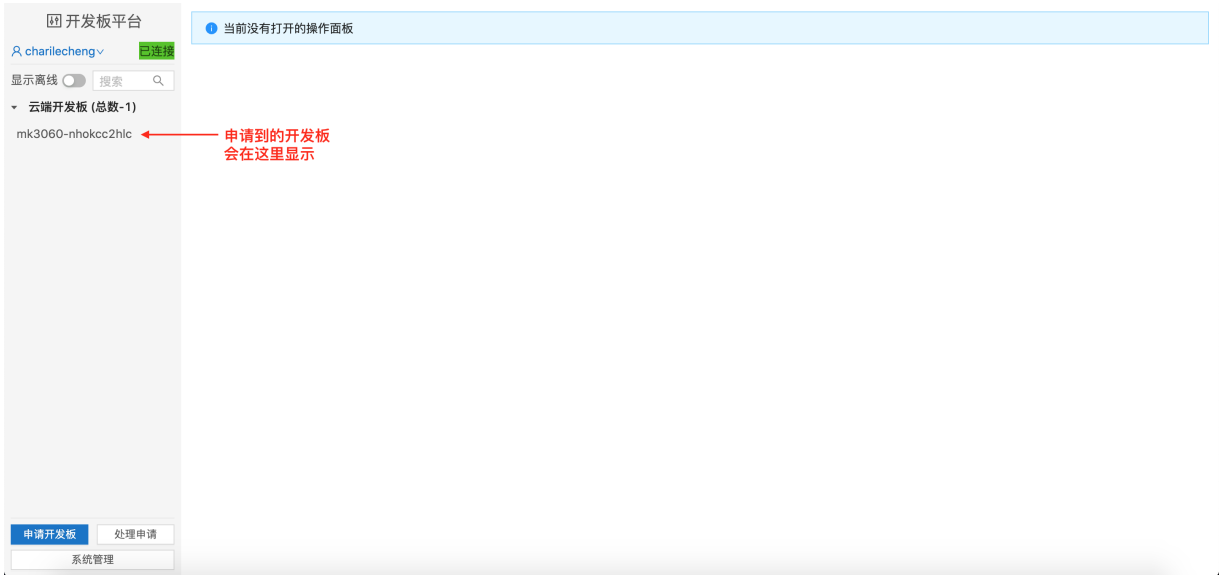


步骤2. 申请需要使用的开发板

按下图所示的步骤，申请您需要使用的开发板。



等待审批通过以后，申请到开发板会出现在左边的设备列表中。点击开发板就可以使用了。



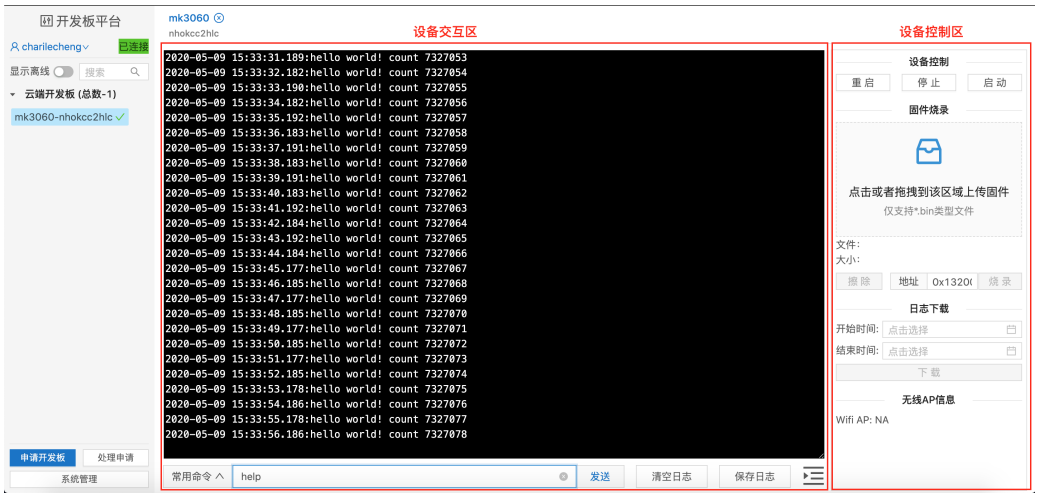
使用线上的开发板开发调试

步骤1. 点击设备，打开其操作面板

步骤2. 熟悉设备的操作面板

设备的操作面板包括左右两个分区，如下图所示：

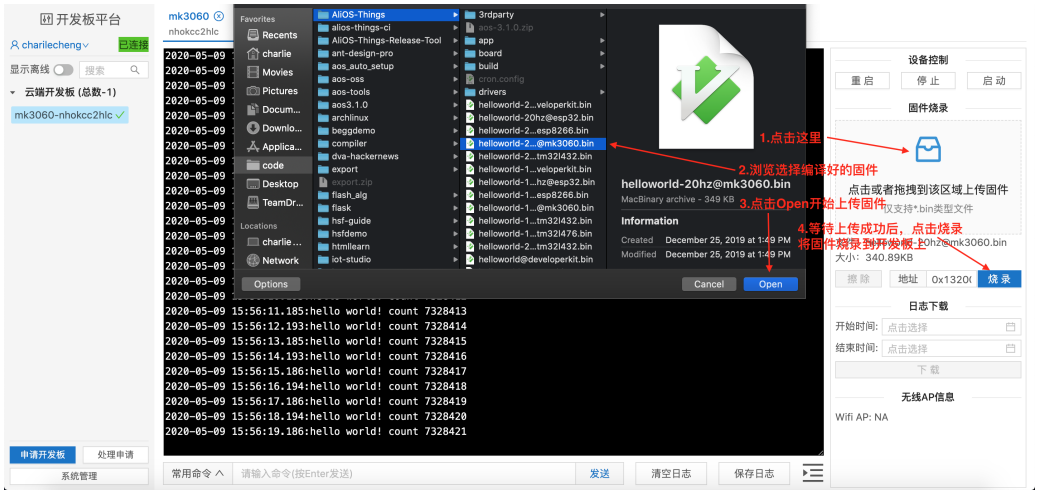
- 左边是设备交互区，负责：显示设备运行的日志；接受用户输入命令并发送给设备
- 右边是设备控制区，可以：控制设备启停；擦除烧录设备固件；拉取设备日志 和 显示额外的信息



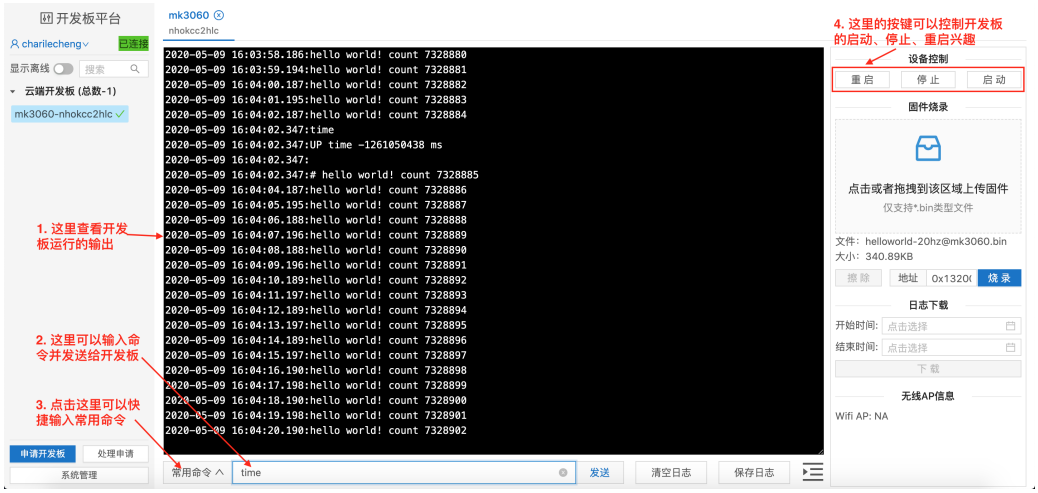
步骤3. 烧录您的固件到设备

Note: 请您事先编译好固件。可以参考前面的相关章节了解如何编译固件：使用命令行开发使用 AliOS Studio IDE 开发

按下图所示的步骤，可以快速地将您开发的固件烧录到开发板上。



步骤4. 与开发板交互



步骤5. 操作日志

使用限制

线上的开发板仅仅能提供有限的操作（串口交互，设备启停，固件烧录擦除等），主要用于学习验证。其使用限制包括：

- 不能外接额外的器件
- 不能与设备做物理交互（如按键交互）
- 不能观察物理的设备状态（如LED灯的闪烁情况）
- 不支持GDB调试

如果您是做实际的设备/项目开发，建议您还是将真实的开发板连到您自己的电脑上进行。

6.3. 使用 Docker 开发

本文推荐docker下进行日常开发和编译。 Docker基本命令参考：这里

Linux环境开发

适用于Linux环境开发者，有专门代码服务器或虚拟机

安装docker

```
$ sudo apt-get install docker-ce
```

获取docker镜像

```
$ docker pull registry.cn-hangzhou.aliyuncs.com/alios_things/rtos:latest
```

启动docker

为其命名alios-docker:

```
$ docker run -it --privileged --name alios-docker registry.cn-hangzhou.aliyuncs.com/alios_things/rtos /bin/bash
```

注意： 1）退出docker后，可用以下命令重新启动已有docker:

```
$ docker container start -ia alios-docker
```

2）镜像更新前，需要备份有用的代码，重新获取新镜像并启动

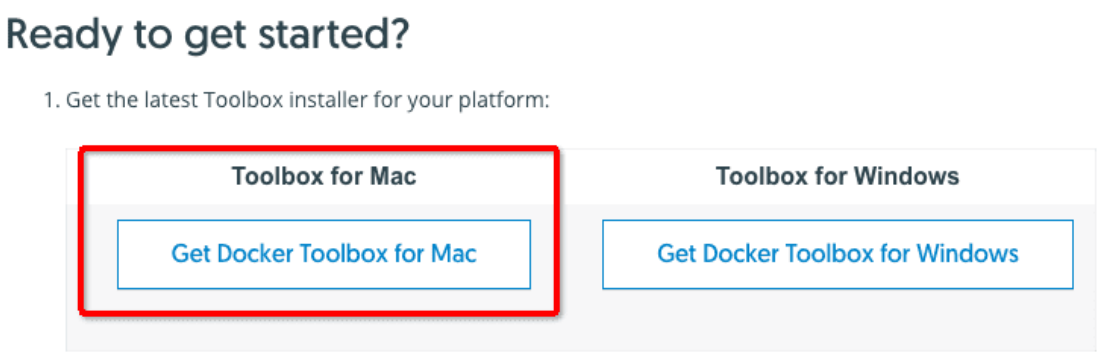
Mac环境开发

适用于无专门代码服务器或虚拟机，习惯mac本机开发

适用场景一：有linux经验，代码编辑与编译均可在docker环境下进行

下载mac环境下的docker工具包并安装

打开https://docs.docker.com/toolbox/overview/#whats-in-the-box



安装后打开Toolbox的DOCKER CLI， 实际上进入mac 的Terminal, 再继续下面的操作

获取docker镜像

```
$ docker pull registry.cn-hangzhou.aliyuncs.com/alios_things/rtos:latest
```

启动docker

为其命名alios-docker:

```
$ docker run -it --privileged --name alios-docker registry.cn-hangzhou.aliyuncs.com/alios_things/rtos bin/bash
```

获取代码

1. 组件化工具获取：按需选择适当组件获取需要的代码 本地zip文件，或 wget http链接(根据所选组件生成源码下载的链接) 获取
2. github获取：将获取全量代码

```
git clone https://github.com/alibaba/AliOS-Things.git -b <release_branch_name>
```

注意：

1. 退出docker后，可用以下命令重新启动已有docker:

   ```
   $ docker container start -ia alios-docker
   ```

2. 需要烧机时，配置docker USB设备，参考Windows下的USB设备配置，启动docker命令中无需-v参数

适用场景二：代码编辑和调试在mac，编译和烧录在docker下进行

与mac场景一比较，需要建立共享目录，完成代码在两种环境中的共享。

下载docker工具

同上

获取docker镜像

同上

获取代码

推荐组件化工具获取，将zip包解压到本机某个目录下， 如 /Users/xxx/alios

启动docker

为其命名alios-docker，并指定本机与docker的目录映射关系： 使用-v 参数 -v <本机代码所在目录名>:<docker中映射名>

```
$ docker run -it --privileged --name alios-docker -v /Users/xxx/alios:/workspace registry.cn-hangzhou.aliyuncs.com/alios_things/rtos bin/bash
```

至此，可以达到对/Users/xxx/alios中的代码进行本地编辑和调试，而编译时， 转入docker中的/workspace下，执行

```
# aos2.1.0以及后续版本
aos make <app>@<board> -c config && aos make
# aos2.1.0之前版本
aos make <app>@<board>
```

注意： 需要烧机时，配置docker USB设备，参考Windows下的USB设备配置， 启动docker命令中-v 参数稍有差别

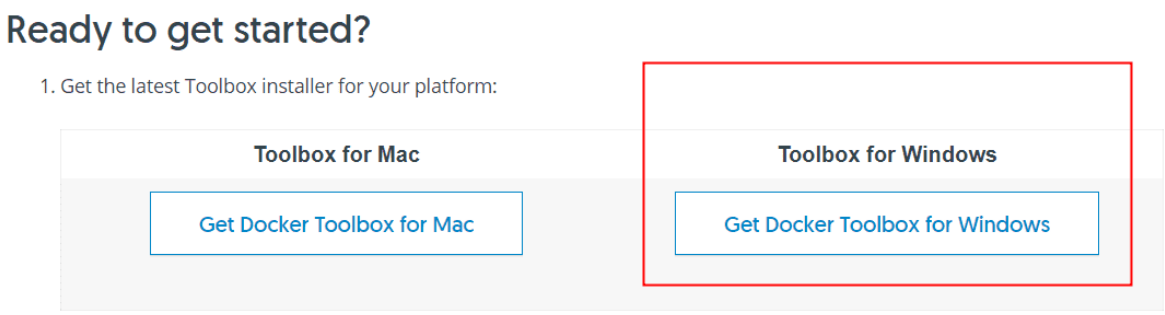
Windows环境开发

适用于无专门代码服务器或虚拟机， 习惯Windows本机开发

适用场景一：有linux经验，可工作于linux虚拟机

下载Windows环境下的docker工具包并安装

打开https://docs.docker.com/toolbox/overview/#whats-in-the-box下载windows下的docker工具



Toolbox将默认安装VirtualBox，之后打开Docker Quickstart Terminal进行下面的操作

获取docker 镜像

```
$ docker pull registry.cn-hangzhou.aliyuncs.com/alios_things/rtos:latest
```

启动docker

```
$ docker run -it --privileged --name alios-docker registry.cn-hangzhou.aliyuncs.com/alios_things/rtos bash
```

获取代码

1. 组件化工具获取：按需选择适当组件获取需要的代码 本地zip文件，或 wget http链接(根据所选组件生成源码下载的链接) 获取
2. github获取：将获取对应发布分支的代码 git clone https://github.com/alibaba/AliOS-Things.git -b <release_branch_name>

注意： 1）退出docker后，可用以下命令重新启动已有docker:

```
$ docker container start -ia alios-docker
```

2）需要烧录设备时， 参考Windows场景二中的USB设备配置，docker启动时，无需-v参数

适用场景二：代码编辑和调试在windows，编译和烧录在docker下进行

与场景一比较，需要建立共享目录，完成代码在两种环境中的共享。

下载docker工具

同上

获取docker镜像

同上

获取代码

推荐组件化工具获取，将zip包解压到本机某个目录下， 如d:\work

启动docker，带目录共享能力

方式一：创建共享目录的docker 执行如下脚本，按要求copy命令并执行后即可 set_share_folder.zip

例如：需要做 d:\work与 docker中的/workspace的目录共享，执行脚本后，命令如下： 1）脚本执行后，自动停在虚拟机终端：

```
$ sudo mkdir –parents /d/work
$ sudo mount –tvboxsf d/work /d/work/
$ exit
```

2）退出后回到windows命令行，启动AliOS docker:

```
$ docker run -it --privileged --name alios-docker -v /d/work:/workspace/ registry.cn-hangzhou.aliyuncs.com/alios_things/rtos bash
```

这样就启动了一个AliOS Things的docker, 之后可以在docker环境里进行编译开发.

注意： Windows下取消共享目录命令：

```
C:/Program Files/Oracle/VirtualBox/VBoxManage.exe sharedfolder remove default --name d/work
```

方式二：利用samba服务的docker

1）**创建存储volume**, 便于与docker下的目录共享：

```
$ docker volume create --name aos-vol
```

2）**启动docker**：为其命名alios-docker，并-v指定volume目录映射到docker中的/workspace

```
$ docker run -it --privileged -p445:445 --name alios-docker -v aos-vol:/workspace/ registry.cn-hangzhou.aliyuncs.com/alios_things/rtos bash
```

3）**安装samba并配置**, 此时已进入docker环境：

```
$ apt install samba
$ vim /etc/samba/smb.conf  #在文件最后添加以下内容
[alios]
  path = /workspace
  public = yes
  case sensitive = yes
  map archive = no
  only guest = yes
  writable = yes
  force user = aosuser
  force group = aosuser
$ groupadd -g 1000 aosuser        #添加分组，与上面指定的分组名保持一致
$ useradd -m -u 1000 -g 1000 aosuser  #添加用户，与上面指定的用户名保持一致
```

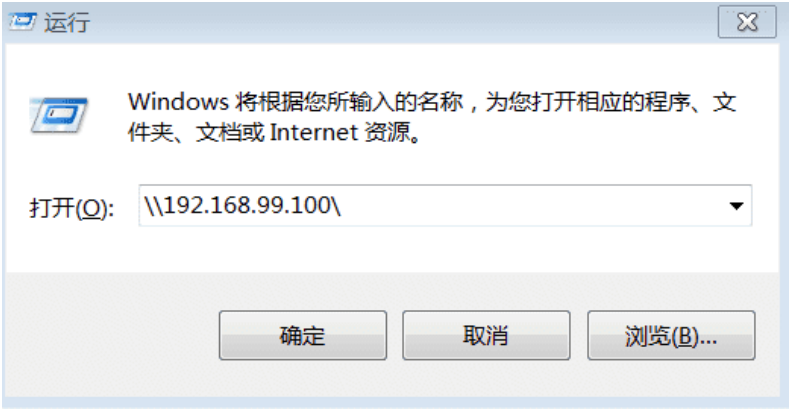
4）启动samba daemon:

```
$ /usr/sbin/smbd
```

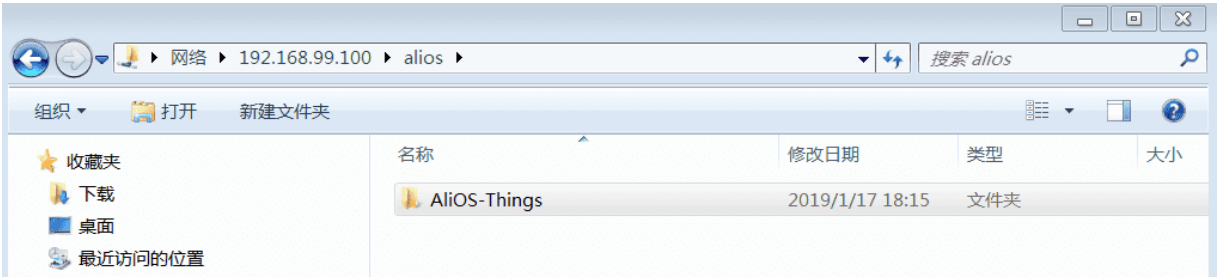
5）获取docker的虚拟IP： 回到Docker Quickstart Terminal里查看ip

```
$ docker-machine ip
192.168.99.100
```

6）配置网络连接：在windows下按win+R键，调出运行窗口，输入ip



此时，在windows下可访问docker里的内容， 亦可随意添加内容到docker目录里：



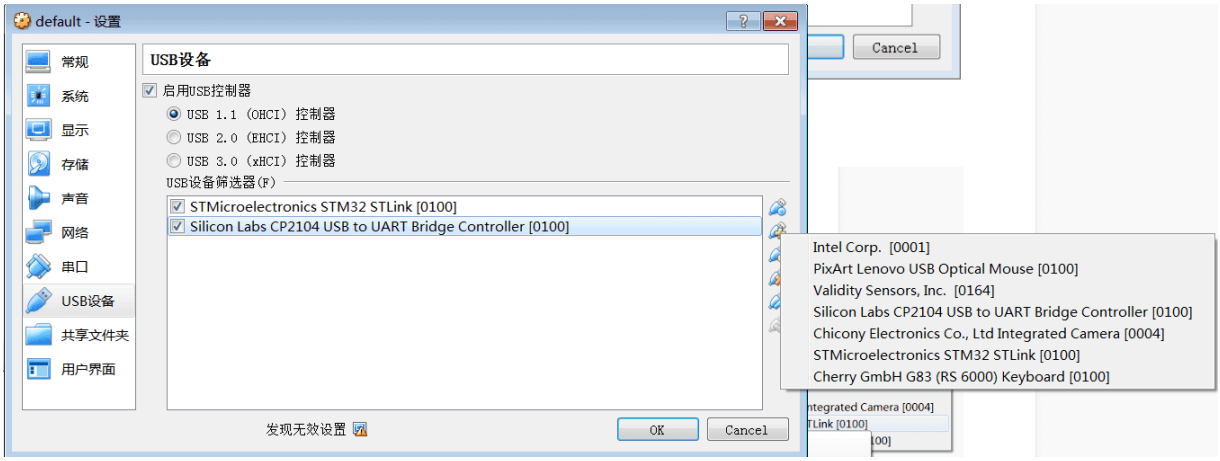
配置docker_usb设备

注：如果无烧录需求， 可忽略此步骤

1）打开Docker Quickstart Terminal （Mac时，可直接使用mac terminal），停止缺省虚拟机：

```
$ docker-machine stop default
```

2）打开VirtualBox的管理界面，设置->usb设备->点选“启用usb控制器”->添加usb设备->确认



3）重新运行缺省虚拟机：

```
$ docker-machine start default
```

或者打开一个新的Docker Quickstart Terminal

```
$ docker-machine ssh default
```

4）检测usb设备：连线后使用下面命令进行检测

```
$ dmesg | grep1 usb
.....
[  19.942282] usb 1-2: cp210x converter now attached to ttyUSB0
```

5）欲使用usb设备的功能，启动docker时需添加新的启动参数--privileged，而-v参数根据目录共享需要添加：

```
$ docker run -it --privileged --name alios-things -v /d/work:/workspace/ registry.cn-hangzhou.aliyuncs.com/alios_things/rtos bash
```

6）编译代码，使用aos命令烧写：比如烧写适配developerkit板子的helloworld应用的image：

```
$ aos upload helloworld@developerkit
.....
[INFO]: Firmware upload succeed!
```

7.FAQ

7.1. 生成IAR/KEIL工程常见问题

首先可以在board目录或者platform/mcu/目录下，在要开发的board目录下打开.mk文件，如果有如下内容，且有相关的文件，则已经支持生成keil/IAR工程，如果没有相关的代码和文件就需要按照生成keil/IAR工程的说明自己添加keil/IAR工程的支持：

```
ifeq ($(COMPILER), armcc)
$(NAME)_SOURCES += startup_stm32l496xx_keil.s
else ifeq ($(COMPILER), iar)
$(NAME)_SOURCES += startup_stm32l496xx_iar.s
else
$(NAME)_SOURCES += startup_stm32l496xx.s
endif
ifeq ($(COMPILER),armcc)
GLOBAL_LDFLAGS += -L --scatter=board/developerkit/STM32L496.sct
else ifeq ($(COMPILER),iar)
GLOBAL_LDFLAGS += --config board/developerkit/STM32L496.icf
else
```

Error:L6218E: Undefined symbol __Heap2Base (referred from soc_impl.o).

这个问题可以在 AliOS Things 的移植文档里找到解决方法。AliOS Things 移植文档文档里有如下说明，可以按照说明来配置内核的堆：

```
2.2.6.3 内核使用堆的配置
如果要使用内存申请功能，则需要打开RHINO_CONFIG_MM_TLF宏，来使能k_mm模块，并且配置对应的堆空间。
堆空间定义有三种方式：链接脚本定义、汇编定义、数组定义。推荐方式：链接脚本定义。
其基本原则是要预留一个内存空间作为堆使用，并将其交给g_mm_region管理。
```

这个问题是下边 soc_impl.c 文件里的代码造成的，可以根据移植文档，修改 soc_impl.c 文件和.sct文件来解决这个问题，也可以使用数组定义和汇编定义的方法修改，可以在参考 platform/mcu/stm32l4xx_cube/aos/soc_impl.c 的方法：

```
extern void *__HeapBase;
extern void *__HeapLimit;
extern void *__Heap2Base;
k_mm_region_t g_mm_region[] = {
{(uint8_t *)&__HeapBase, (uint32_t)0x8000},
{(uint8_t *)&__Heap2Base, (uint32_t)0x8000}
};
int g_region_num = sizeof(g_mm_region)/sizeof(k_mm_region_t);
```

其它的芯片也许会碰到相同的问题，可以使用前述的方法解决。

armcc: not found

需要添加keil的bin目录到环境变量里，具体请参考第一个 Keil MDK 工程 > 准备工作。

Fatal error: C3903U: Argument '/hardfp' not permitted for option 'apcs'.

keil没有注册，所以默认不支持浮点型编译，请注册keil。

7.2. aos-cube常见问题

aos-cube已经安装成功，但是还是无法运行aos指令（找不到aos）

这种情况请安装如下操作进行确认：

第一步：

可以试试**重启终端或者vscode**，然后再运行aos指令。如果还是不行的话，请参考第二步。

第二步：

1. 请确认aos-cube安装在哪个目录：

```
win10:/d/workspace/github/AliOS-Things$ python -m site
sys.path = [
  'D:\\workspace\\github\\AliOS-Things',
  'C:\\windows\\SYSTEM32\\python27.zip',
  'C:\\Python27\\DLLs',
  'C:\\Python27\\lib',
  'C:\\Python27\\lib\\plat-win',
  'C:\\Python27\\lib\\lib-tk',
  'C:\\Python27',
  'C:\\Users\\xxxx\\AppData\\Roaming\\Python\\Python27\\site-packages',
  'C:\\Python27\\lib\\site-packages',
]
USER_BASE: 'C:\\Users\\xxxx\\AppData\\Roaming\\Python' (exists)
USER_SITE: 'C:\\Users\\xxxx\\AppData\\Roaming\\Python\\Python27\\site-packages' (exists)
ENABLE_USER_SITE: True
```

其中 `USER_BASE` 就是python用户安装目录，查看该用户安装目录下的 `Scripts` 中是否存在aos指令，如果存在的话，把该目录（ `C:\Users\chenan.xxw\AppData\Roaming\Python\Scripts` ）添加到Path环境变量中即可。

如果还是有问题的话，请提问题到github issues，或者可以参考使用我们提供的docker镜像。

aos-cube升级到新版本无法正常工作

aos-cube升级到0.3.x版本，运行aos提示“from aos.**main** import main”失败，本地Python模块路径未更新，可尝试先卸载再安装：

```
# 首先卸载已安装的aos-cube
$ pip uninstall aos-cube
# 重新安装aos-cube
$ pip install aos-cube
```

linux系统上串口、usb没有权限

Linux 操作串口及 j-link 会有root权限要求，尤其在使用 AliOS Studio时候。可以复制60-openocd.rules(来自 Arduino github project)文件到 `/etc/udev/rules.d` 目录下，然后运行如下命令生效：

```
sudo udevadm control --reload-rules
```

或者你也可以手动对串口及 j-link 的权限做配置：

- `$ sudo usermod -a -G dialout $(whoami)` ,添加当前用户到 dialout 组，提供直接使用串口能力。

- `$lsubs` 找到 i-link 厂商ID。如： `Bus 002 Device 008: ID 1366:0105 SEGGER` ,厂商ID为1366，新建 `/etc/udev/rules.d/99-stlink-v2.rules` 文件，在文件里面添加规则：`SUBSYSTEM=="usb", ATTR{idVendor}=="1366", MODE="666", GROUP="plugdev"` 。

配置操作串口及 j-link权限后，重启系统生效。

ModuleNotFoundError: No module named 'constant'

```
Create Project Fail.Traceback (most recent call last):
File "/Users/chenyulun/.aos/python-venv/bin/aos",
line 7, in from aos.aos import main File "/Users/chenyulun/.aos/python-venv/lib/python3.6/site-packages/aos/aos.py",
line 26, in from .util import * File "/Users/chenyulun/.aos/python-venv/lib/python3.6/site-packages/aos/util.py",
line 10, in from constant import * ModuleNotFoundError: No module named 'constant'
```

aos-cube 0.3.11之前版本只支持python2.7，当前环境如果是python3的话，运行aos-cube就会导致冲突，请更新aos-cube。

No module named SCons.Script

```
aos-cube version: 0.2.60
[INFO]: Target: helloworld@developerkit
[INFO]: Currently in aos_sdk_path: '/home/ljh/work/AliOS/github_branch/AliOS-Things'
SCons import failed. Unable to find engine files in:
 /usr/local/bin/../engine
 /usr/local/bin/scons-local-3.0.1
 /usr/local/bin/scons-local
 /usr/local/lib/python2.7/dist-packages/lib/scons-3.0.1
 /usr/local/lib/scons-3.0.1
 /usr/lib/scons-3.0.1
 /usr/local/lib/scons-3.0.1
 /usr/local/lib/python2.7/dist-packages/lib/python2.7/site-packages/scons-3.0.1
 /usr/local/lib/python2.7/site-packages/scons-3.0.1
 /usr/lib/python2.7/site-packages/scons-3.0.1
 /usr/local/lib/python2.7/site-packages/scons-3.0.1
 /usr/lib/scons-3.0.1
 /usr/local/lib/python2.7/dist-packages/lib/scons
 /usr/local/lib/scons
 /usr/lib/scons
 /usr/local/lib/scons
 /usr/local/lib/python2.7/dist-packages/lib/python2.7/site-packages/scons
 /usr/local/lib/python2.7/site-packages/scons
 /usr/lib/python2.7/site-packages/scons
 /usr/local/lib/python2.7/site-packages/scons
 /usr/lib/scons
Traceback (most recent call last):
 File "/usr/local/bin/scons", line 192, in <module>
  import SCons.Script
ImportError: No module named SCons.Script
[AliOS-Things] ERROR: "s" returned error code 1.
[AliOS-Things] ERROR: Command "scons -f /home/ljh/work/AliOS/github_branch/AliOS-Things/ucube.py COMMAND=upload APPLICATION=helloworld BOARD=developerkit" in "/home/ljh/work/AliOS/github_branch/AliOS-Things"
```

请依次尝试以下方法：

- 更新scons： `pip install scons -U`
- 卸载scons然后重新安装： `pip uninstall scons` ； `pip install scons` 。
- 请使用虚拟环境安装aos-cube，参考链接：https://github.com/alibaba/AliOS-Things/wiki/AliOS-Things-Linux-Environment-Setup#安装python和aos-cube

Could not install packages due to an EnvironmentError: [Errno 13] Permission denied

ubuntu系统下，非root权限用户使用pip安装python包会提示没有权限问题：

```
/usr/local/lib/python2.7/dist-packages/pip/_vendor/requests/__init__.py:83: RequestsDependencyWarning: Old version of cryptography ([1, 2, 3]) may cause slowdown.
  warnings.warn(warning, RequestsDependencyWarning)
Requirement already satisfied: aos-cube in /usr/local/lib/python2.7/dist-packages (0.2.59)
Requirement already satisfied: requests in /usr/local/lib/python2.7/dist-packages (from aos-cube) (2.19.1)
Collecting scons (from aos-cube)
Collecting esptool (from aos-cube)
  Downloading https://files.pythonhosted.org/packages/51/9e/1abfb18922eeab54f0e38e7d0f113f18b285048a8d9b827f56927a848ee5/esptool-2.5.1.tar.gz (75kB)
    100% |████████████████████████████████| 81kB 58kB/s
Collecting pyserial (from aos-cube)
  Using cached https://files.pythonhosted.org/packages/0d/e4/2a744dd9e3be04a0c0907414e2a01a7c88bb3915cbe3c8cc06e209f59c30/pyserial-3.4-py2.py3-none-any.whl
Requirement already satisfied: idna<2.8,>=2.5 in /usr/local/lib/python2.7/dist-packages (from requests->aos-cube) (2.7)
Requirement already satisfied: urllib3<1.24,>=1.21.1 in /usr/local/lib/python2.7/dist-packages (from requests->aos-cube) (1.23)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /usr/local/lib/python2.7/dist-packages (from requests->aos-cube) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python2.7/dist-packages (from requests->aos-cube) (2018.8.13)
Collecting pyaes (from esptool->aos-cube)
Collecting ecdsa (from esptool->aos-cube)
  Using cached https://files.pythonhosted.org/packages/63/f4/73669d51825516ce8c43b816c0a6b64cd6eb71d08b99820c00792cb42222/ecdsa-0.13-py2.py3-none-any.whl
Building wheels for collected packages: esptool
  Running setup.py bdist_wheel for esptool ... done
  Stored in directory: /disk2/chenan.xxw/.cache/pip/wheels/4b/03/99/cb358131acc8e50b6fd724e39e050751464d85245c73de5ea0
Successfully built esptool
Installing collected packages: scons, pyserial, pyaes, ecdsa, esptool
Could not install packages due to an EnvironmentError: [Errno 13] Permission denied: '/usr/local/lib/python2.7/dist-packages/scons-3.0.1.dist-info'
Consider using the `--user` option or check the permissions.
```

请添加 `--user` 参数安装即可：

```
pip install --user aos-cube
```

fatal: unable to access 'https://gitee.com/alios-things/gcc-arm-none-eabi-osx.git/': Failed to connect to localhost port 8118: Connection refused

需要手动下载toolchain:

```
git clone https://gitee.com/alios-things/gcc-arm-none-eabi-osx.git gcc-arm-none-eabi
mv gcc-arm-none-eabi/main build/compiler/gcc-arm-none-eabi/OSX && rm -rf gcc-arm-none-eabi
```

UnicodeDecoderError：‘utf8’ codec can't decode byte xxx in position xxx

文件路径里包含中文等其它特殊字符；或者源代码文件是非ascii和utf8编码的，且包含中文等特殊字符。

需要将AliOS-Things的代码放在不包含特殊字符的路径下，并且创建工程目录时，也需要确保路径中不包含特殊字符；

对于非ascii编码和utf8编码的文件，需要使用vs code等编辑器将其转为utf8编码。同时需要更新文中包含的特殊字符，否则会显示乱码。

7.3. AliOS-Studio常见问题

Visual Studio Code is unable to watch for file changes in this large workspace

针对Linux系统，windows和mac不会出现这种情况。

该错误在 linux系统上比较常见，主要是因为linux系统最大可监听文件数有限制。linux系统默认系统可监听文件数为8192个， `AliOS-Things` 的源码比较大，文件数远远大于8192个，此时vscode无法监听所有的文件改动，导致 AliOS Studio 插件会工作不正常，报如下错误：

解决办法： 此时需要设置**linux系统最大可监听文件数**。

使用如下命令查看当前可监听文件数：

```
cat /proc/sys/fs/inotify/max_user_watches
```

编辑文件： `/etc/sysctl.conf` ，然后增加如下行：

```
fs.inotify.max_user_watches=524288
```

使用如下指令生效：

```
sudo sysctl -p
```

Arch Linux 用户请参考此链接.

更多细节请参考："Visual Studio Code is unable to watch for file changes in this large workspace" (error ENOSPC)。

Workspace is too large to watch for file changes

和上面的问题一样：Visual Studio Code is unable to watch for file changes in this large workspace