

无需从0开发

1天上手智能语音离在线方案



- 最完善的离在线语音上手开发指南
- 适配Windows、Linux两大开发环境
- 涵盖6大关键组件开发要点





平头哥芯片开放社区交流群
扫码关注获取更多信息



扫码注册平头哥 OCC 官网
观看各类视频及课程



阿里云开发者"藏经阁"
海量免费电子书下载

目录

智能语音方案介绍	4
1. 概述	4
2. 智能语音终端 SDK 架构特点	4
3. 语音领域组件	5
智能语音终端开发板介绍	8
1. CB5631 开发板介绍及上手指导	8
2. CB5654 开发板介绍及上手指导	10
智能语音 SDK 快速上手	13
1. Windows 开发环境下工具下载及调试安装	13
2. Linux 开发环境下工具下载及调试安装	18
3. 例程运行	23
智能语音应用开发指南	28
1. 概述	28
2. 核心组件介绍	29
3. 应用示例讲解	51
智能语音开发板适配指南	58
1. CB5631 适配指南	58
2. CB5654 适配指南	65
操作系统介绍及驱动开发指南	73
智能语音组件适配指南	74
1. 语音服务适配指南	74
2. 云服务适配指南	84
3. 语音算法适配指南	91
基本调试指南	95
1. 使用串口调试	95
2. 使用 GDB 调试	105
3. CPU Exception 分析及调试	117

I 智能语音方案介绍

1. 概述

针对智能语音应用场景，平头哥推出了以 AliOS Things 内核为基础的智能语音软件平台。该平台支持多款高性价比的语音 AI 芯片，提供了丰富的驱动模块及组件，包含拾音模块、本地唤醒算法模块、播放器模块及云端语音服务模块等，为客户快速实现产品落地提供了有力支撑。



2. 智能语音终端 SDK 架构特点



平头哥智能语音终端 SDK 有如下特点：

面向智能语音领域的软件框架

- 支撑语音应用的丰富组件：拾音、播放器、语音云服务等
- 易于适配的本地算法框架
- 完整的智能音箱解决方案应用 DEMO

极简开发

- 提供 CDK IDE 开发工具
- 提供 Shell 交互，支持内存踩踏、泄露、最大栈深度等各类侦测
- 提供包括低功耗框架、网络协议、蓝牙协议栈、虚拟文件系统、网络管理等各类模块化组件

高度优化的内核

- 内核支持 Idle Task 成本：Ram<1K, Rom<2k
- 支持 CSKY 加速指令对系统性能进行优化

全面的安全保护

- 提供系统和芯片级别安全保护
- 支持可信运行环境(TEE)
- 支持预置 ID2 根身份证和非对称密钥以及基于 ID2 的可信连接和服务

IoT 专属组件

- 空中固件升级（FOTA）
- 集成 AT 模组指令
- 快速 IoT 云端接入
- 支持多种物联网协议：Alink、MQTT、COAP、LWM2M
- 支持多种网络协议栈：
 - TCP/IP 协议栈（LwIP）
 - 套接字适配层(SAL)
 - 自组织网络协议(uMesh)
 - 支持 WIFI、Ethernet、NB-IoT、GPRS、Bluetooth 等通信硬件

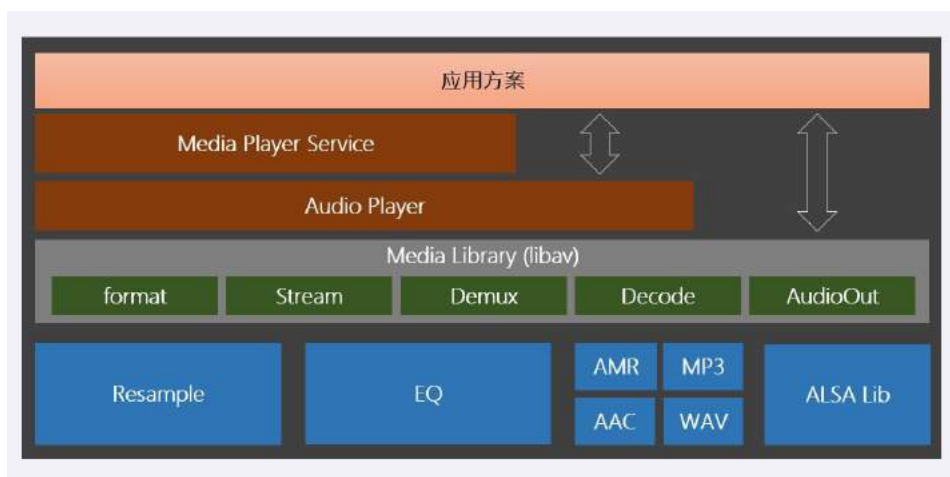
3. 语音领域组件

智能语音终端方案集成了针对语音应用领域的专用组件，提供了一系列便于用户使用的上

层接口，通过这些接口的组合，用户可以快速开发出符合产品定义的应用代码。本章将会介绍包括播放器、语音服务、云服务相关的功能架构，在《智能语音终端应用开发指南》一章将会详细介绍相关 API 的使用方法。

3.1 播放器服务

播放器支持 mp3, mp4, flac、wav、amr 等多种常见音频格式，可实现常用播放控制的功能，同时支持提示音、语音合成(TTS)、音乐播放的状态切换与维护等智能语音功能，方便上层应用开发。



其功能特点如下：

- 轻量、极简音频播放器
- MP3 解码: ROM < 35K, RAM < 20K
- 四层架构，按需裁剪&扩展
- 低时延，首播延时最低<20ms
- 编解码 DSP 指令加速，充分利用硬件资源
- 支持本地/核间(跨核)解码
- 支持 wav、mp3、m4a、amrnb、amrwb、flac、adts 等多种音频格式
- 支持 sd 卡、http(s)、fifo、mem 等多种取流方式
- 支持语音合成(TTS)播放
- 支持软件音量、音量曲线配置
- 支持音频信息及当前播放时间获取
- 支持重采样输出
- 支持音效(audio effector)、量化器(EQ)扩展及配置

其支持的播控操作如下：

- 支持多种类型播放、暂停、恢复、停止等操作

- 支持音乐播放后自动恢复
- 支持音量调节及渐入渐出效果
- 支持最低音量播放

3.2 语音服务

语音服务提供了拾音及本地算法相关功能，为智能语音应用的核心组件。用户使用该组件可以方便的获取包括本地唤醒、断句、回声消除等服务。

其功能特点如下：

- 支持多路麦克风及参考音采集
- 支持 16/24/32/48 KHz 的音频采样率
- 支持 KWS、VAD、AEC 等多种本地算法

3.3 云服务

语音云服务提供了针对智能语音应用的云端语音识别、语音合成、语义理解等服务接口，用户结合语音服务的相关事件即可快速实现符合产品定义的语音交互流程。



其功能特点如下：

- 支持云端 ASR、NLP、TTS 服务
- 支持唤醒音频云端二次确认
- 支持交互流程打断
- 已接入多家语音云服务平台

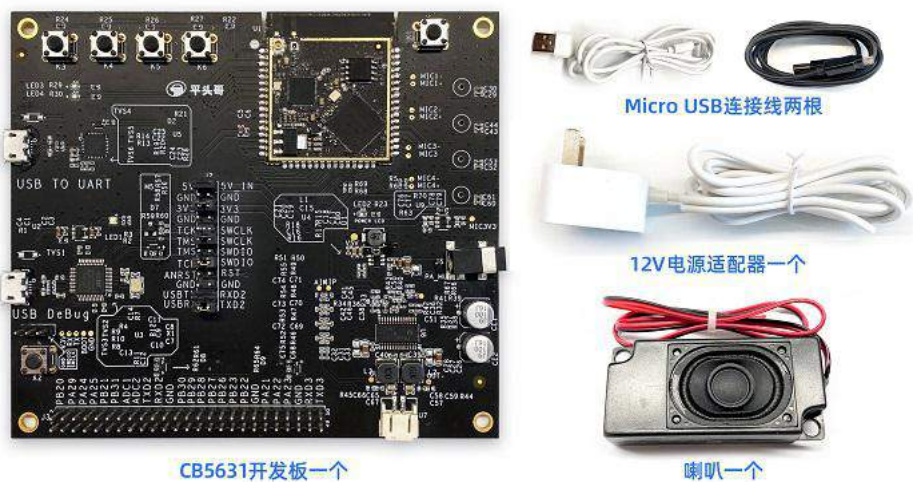
I 智能语音终端开发板介绍

1. CB5631 开发板介绍及上手指导

1.1 简介

CB5631 智能语音开发套件中包含：

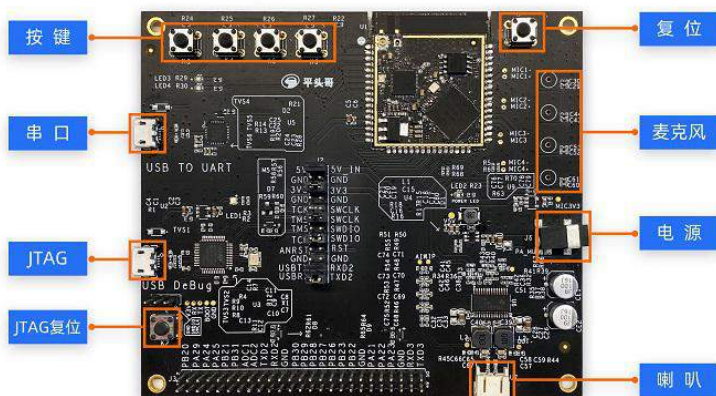
- CB5631 开发板一个
- 12V 电源适配器一个
- 喇叭一个
- Micro USB 连接线两根



1.2 开发板资源介绍

平头哥 CB5631 开发板采用 CH5631 多核处理器。其中包含两个玄铁 804 CPU 核，分别负责所有应用程序的运行和多媒体处理。还有一个玄铁 805 CPU 核，负责语音算法处理。

CB5631 开发板提供丰富的外设资源，使得开发者可以快速开发应用原型，CB5631 开发板资源如下图所示：



常用板载资源如下：

- MCU：CH5631，三核结构，主频分别是 CPU0：210MHz，CPU1：210MHz，CPU2：168MHz，片内叠封 16M byte SDRAM，片内集成 448K byte 高速 RAM，片内 eFUSE，可用于存储用户设备信息
- 片外 QSPI FLASH：GD25Q127C，16MB
- 外设
 - LED：1 个红色电源指示灯，1 个蓝色 JTAG 指示灯，1 个绿色可编程灯
 - 按键：1 个 JTAG 复位键，1 个系统复位键，4 个可编程通用按键
 - 常用接口：USB 转串口、喇叭接口、电源接口
 - 调试接口：USB 转 JTAG

1.3 引脚功能复用说明

PIN	GPIO/PWM	PINMUX1	PINMUX2	PINMUX3
PB20	GPIO/PWM1_O1	PDM_DAT2		
PA29	GPIO/PWM0_IO6	UART0_CTS	JTG_TRST	USI1_NSS
PA24	GPIO/PWM0_O9	EMMC_DAT0	I2S1_WSCLK	
PA25	GPIO/PWM0_O11	EMMC_DAT1	I2S1_SDA	
PB21	GPIO/PWM1_O3	PDM_DAT3		
PB31	GPIO/PWM1_IO10	USI2_NSS		
PB24(TXD2)	GPIO/PWM1_O9	UART2_TX		
PB25(RXD2)	GPIO/PWM1_O11	UART2_RX		
PB30	GPIO/PWM1_IO8	USI2_SD1		

PB29	GPIO/PWM1_IO6	USI2_SD0		
PB28	GPIO/PWM1_IO4	USI2_SCLK		
PB27	GPIO/PWM1_IO2	UART2_CTS		
PB26	GPIO/PWM1_IO0	UART2_RTS		
PB23	GPIO/PWM1_O7			
PB22	GPIO/PWM1_O5			
PA21	GPIO/PWM0_O3	USI0_SCLK		
PA22	GPIO/PWM0_O5	USI0_SD0		
PA23	GPIO/PWM0_O7	EMMC_CLK	I2S1_SCLK	
PB1(RXD3)	GPIO	I2S2_WSCLK	DMIC_CLK12	UART3_RX
PB0(TXD3)	GPIO	I2S2_SCLK	DMIC_CLK11	UART3_TX

注：引脚名称按照开发板上显示顺序排列

2. CB5654 开发板介绍及上手指导

2.1 简介

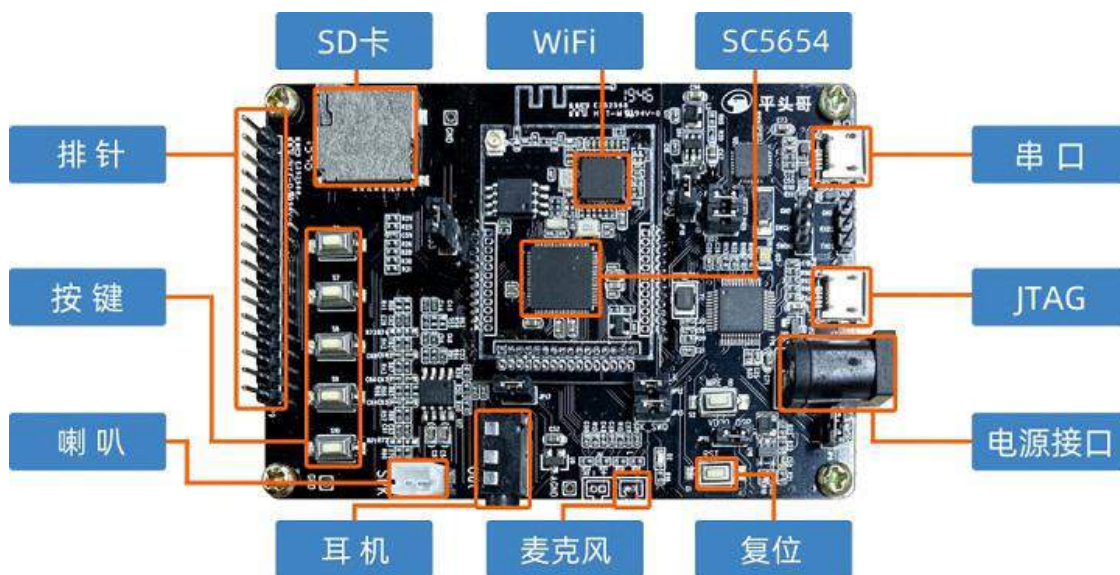
CB5654 智能语音开发套件中包含：

- CB5654 开发板一个
- 5V 电源适配器一个
- 喇叭一个
- Micro USB 连接线两根

2.2 开发板资源介绍

平头哥 CB5654 开发板采用 SC5654 双核异构处理器。其中一个核为玄铁 803 内核，负责应用处理，包括对接云端 ASR/NLP/TTS 服务、语音播报、网络通信、远程升级等。另一个核为 DSP 内核，负责本地语音识别算法。

CB5654 开发板提供丰富的外设资源，使得开发者可以快速开发应用原型，CB5654 开发板资源如下图所示：



常用板载资源如下：

- MCU：SC5654，双核架构，主频 160 MHz + 160 MHz，片内叠封 8M byte SDRAM，片内集成 240 KB 高速 RAM
- 片外 QSPI FLASH：M25Q64，8 MB
- 外设
- LED：1 个红色电源指示灯，1 个 RGB 三色可编程灯
 - 按键：1 个系统复位键，5 个可编程通用按键
 - 常用接口：USB 转串口、喇叭接口、电源接口
 - 调试接口：USB 转 JTAG

2.3 引脚功能复用说明

JP 引脚号	芯片引脚号	引脚名称	说明
1	VCC33	-	-
2	62	PB4	PCW_SCLK
3	61	PB5	PCW_WS
4	60	PB6	PCW_SDI
5	59	PA7	PCW_SDO
6	-	GND	-
7	53	PA2	ADIN2

8	7	PC13	I2C_SDA
9	6	PC12	I2C_SCL
10	-	GND	-
11	40	PD8	RXD3
12	41	PD7	TXD3
13	45	PD1	GPIO2_1/TDI
14	46	PD0	GPIO2_0/TCK
15	42	PD4	GPIO2_4/TRST
16	-	GND	-

注：JP9 引脚说明，由左至右，靠近 SD 卡的位 1 脚

I 智能语音 SDK 快速上手

1. Windows 开发环境下工具下载及调试安装

1.1 概述

Windows 开发环境采用剑池 CDK 开发工具。剑池 CDK 开发工具以极简开发为理念，是专为 IoT 应用开发打造的集成开发环境。它在不改变用户开发习惯的基础上，全面接入云端开发资源，结合图形化的 OSTRacer、Profiling 等调试分析工具，加速用户产品开发。

1.2 CDK 工具下载

- 登录[平头哥芯片开放社区](#)，进入栏目“技术部落->资源下载->工具->CDK 集成开发环境”，下载最新版本的剑池 CDK 集成开发环境安装包。下载完成后，点击 setup.exe，根据 CDK 安装向导提示完成安装。

注意：建议 CDK 不要安装在 C 盘，否则需要管理员权限运行

1.3 Windows 调试环境安装

安装串口驱动程序

- 登录[平头哥芯片开放社区](#)，进入栏目“技术部落->资源下载->工具->驱动工具”，下载并安装“CP210x_Windows10_Drivers”串口转 USB 驱动包（注：根据操作系统选择对应的版本）。

安装 Debug Server

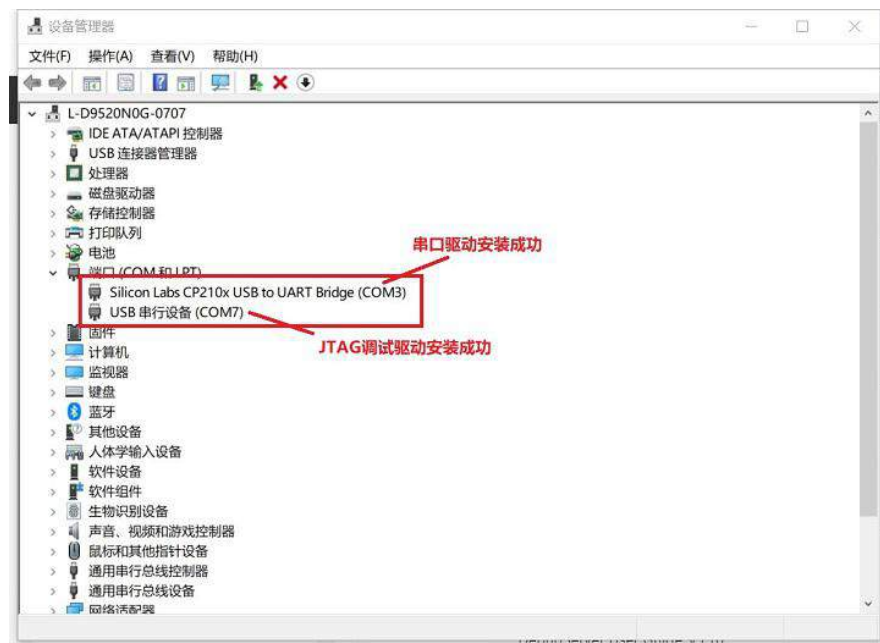
- 登录[平头哥芯片开放社区](#)，进入栏目“技术部落 -> 资源下载 -> 工具 -> Debug Server”，下载最新 CSKY-DebugServer-windows 版本。

注：下载或安装过程中有可能被防火墙拦截，注意将其添加到防火墙白名单中

- Debug Server 安装过程中，会同时安装 JTAG 调试驱动。

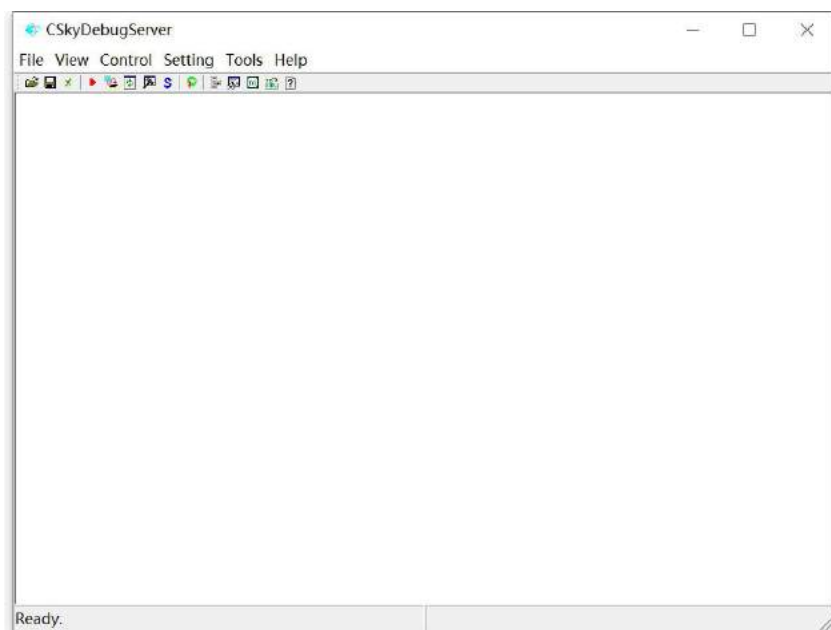
检查驱动状态


- 插入开发板调试串口和 JTAG 调试口后，打开计算机设备管理器，确认串口驱动以及 JTAG 调试驱动安装无误。

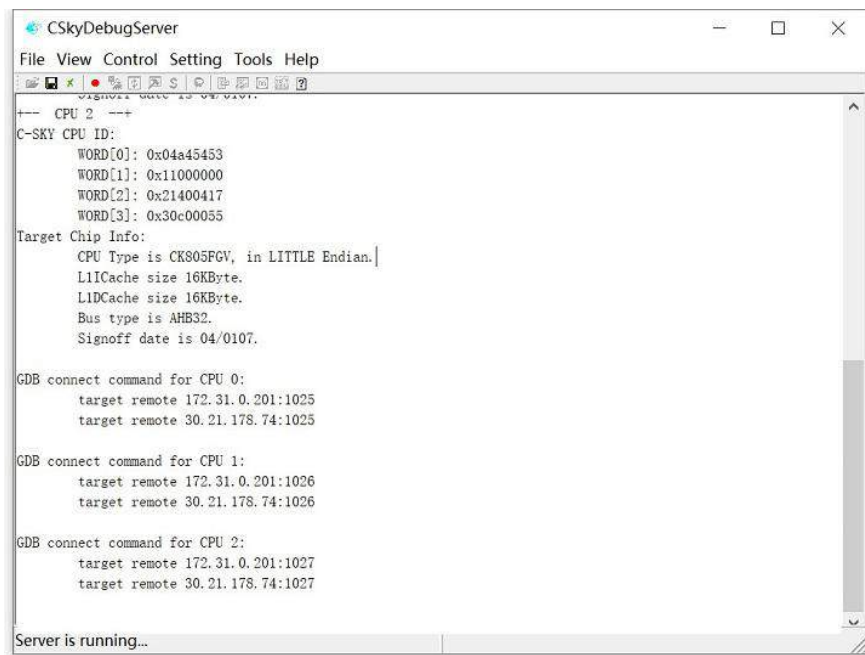


运行 Debug Server

- 烧录之前需要打开并运行 CSkyDebugServer。CSkyDebugServer 初始状态，左下角显示 Ready:



- 上电开发板，确保蓝色 JTAG 指示灯被点亮。点击 ，CSkyDebugServer 与设备连接成功：



DebugServer 处于上述的运行态时，可以烧录固件和对程序进行 GDB 调试。

1.4 应用开发

创建实例工程

本节以“智能语音终端解决方案”（适用于 CB5654 开发板）为例，介绍如何使用 CDK 进行应用开发。

- 运行 CDK，在首页点击新建工程：



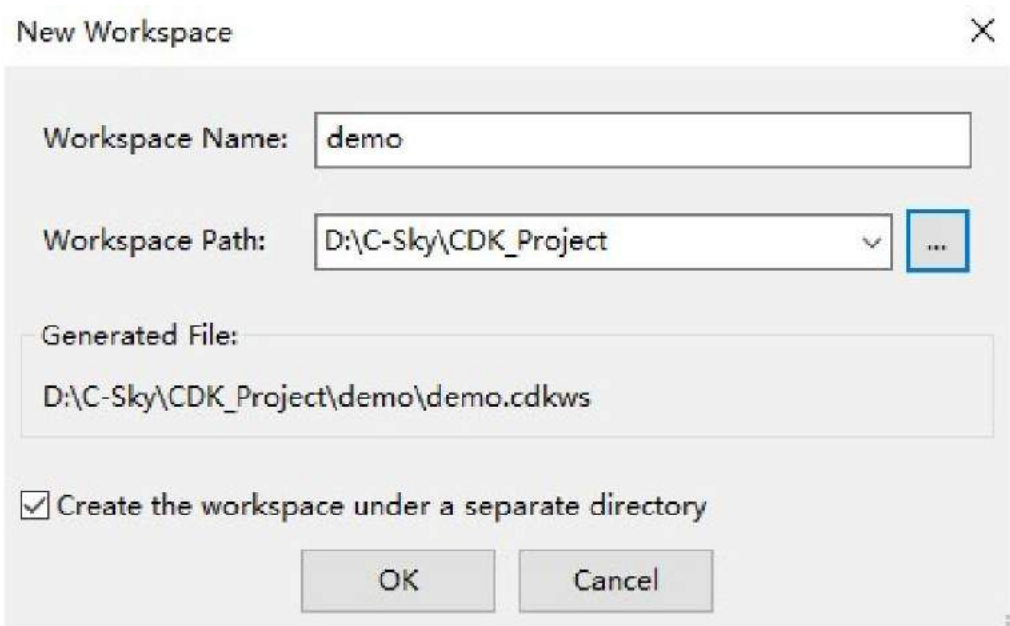
- 在搜索框内输入智能语音并按回车，在第一个搜索结果右边点击创建工程：



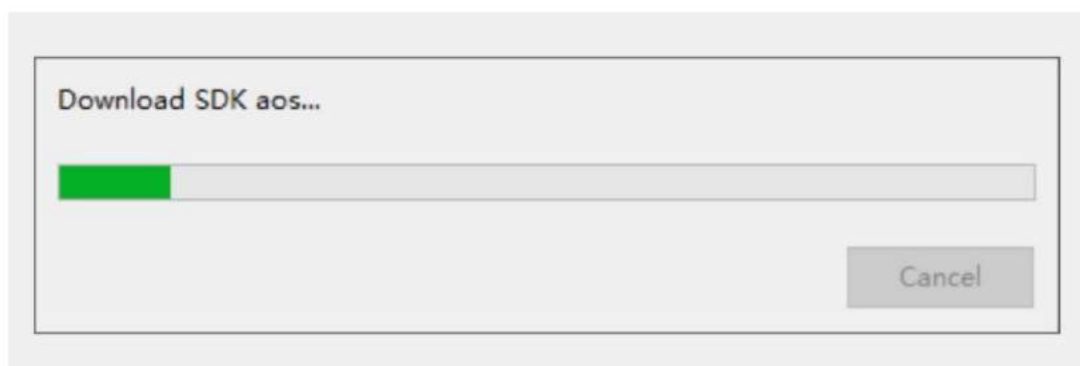
- 输入工程名称，点击下载方案：



- 输入工作空间名称和路径，点击 OK 创建工作空间

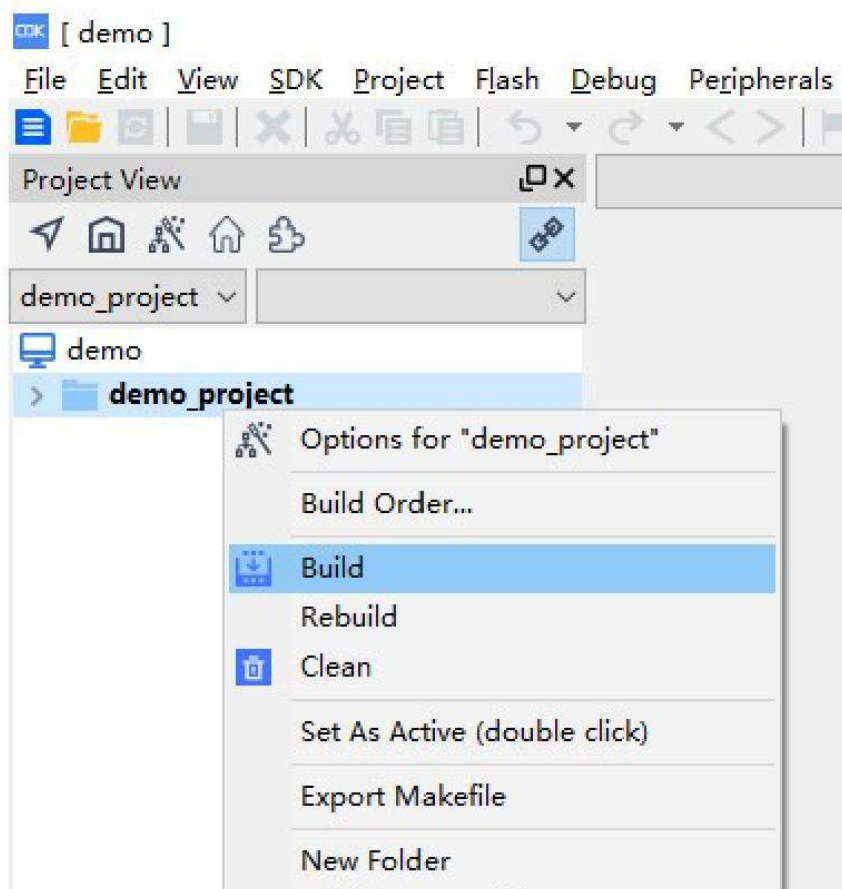


- 之后 CDK 会创建工程并自动下载工程所需依赖。



编译

- 下载成功后，可以在左边的导航栏中看到项目（本例中项目名称 demo_project）。
- 右击项目名称，选择 Build 开始编译。
- 编译时间大约需要 10 分钟。



- 编译成功后，在界面底部的输出窗口，会显示成功信息：

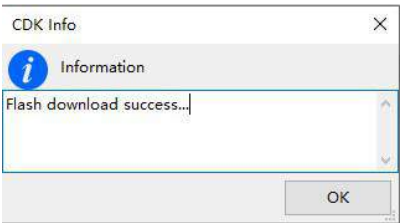
```
-----
bmtb, 0, 0, 0x08000000, 0x0001000, 0x08001000, bmtb
boot, 0, 0, 0x08001000, 0x00010000, 0x08011000, boot
tee, 0, 0, 0x08011000, 0x00010000, 0x08021000, tee
imtb, 0, 0, 0x08021000, 0x00002000, 0x08023000, imtb
prim, 1, 1, 0x08023000, 0x00300000, 0x08323000, prim
cpu1, 0, 1, 0x08323000, 0x00080000, 0x083a3000, cpu1
cpu2, 0, 1, 0x083a3000, 0x00200000, 0x085a3000, cpu2
kv, 0, 0, 0x08623000, 0x00004000, 0x08627000
lpm, 0, 0, 0x08627000, 0x00020000, 0x08647000, lpm
lfs, 0, 0, 0x08647000, 0x00040000, 0x08687000
misc, 0, 0, 0x08687000, 0x003c0000, 0x08a47000
kp, 0, 0, 0x08fff000, 0x00001000, 0x09000000, kp
boot, 41592 bytes
tee, 21688 bytes
prim, 1671812 bytes
cpu1, 414524 bytes
cpu2, 387330 bytes
lpm, 24484 bytes
kp, 576 bytes
bmtb, 192 bytes
imtb, 8192 bytes
-----
Done
====0 errors, 2 warnings, total time : 12m32s335ms=====
```

烧录

点击工具栏的  开始烧录：



烧录完成后，显示烧录成功窗口：



2. Linux 开发环境下工具下载及调试安装

2.1 编译环境配置

- 安装 python3 、python3-pip、scons

```
sudo apt-get install python3 python3-pip
sudo pip install scons (必须用 python3-pip 下载)
```

- 切换到 python3

```
# 增加配置项
sudo update-alternatives --install /usr/bin/python python /usr/bin/python2 100
$ sudo update-alternatives --install /usr/bin/python python /usr/bin/python3 100
# 切换配置
$ sudo update-alternatives --config python
There are 2 choices for the alternative python (providing /usr/bin/python).
Selection Path Priority Status
-----
* 0 /usr/bin/python2 100 auto mode
1 /usr/bin/python2 100 manual mode
2 /usr/bin/python3 100 manual mode
Press <enter> to keep the current choice[*], or type selection number:2
update-alternatives: using /usr/bin/python3 to provide /usr/bin/python (python)
in manual mode
```

- 卸载旧的编译工具 yoctools，之前如果没安装请跳过该步骤

```
$ sudo pip uninstall yoctools
```

- 安装编译工具

```
$ sudo pip install --no-binary=yoctools http://yoctools.oss-cn-beijing.aliyuncs.com/yoctools-1.0.60.1.tar.gz -i
https://mirrors.163.com/pypi/simple/
```

2.2 工具链安装

编译工具链采用 gcc 编译，第一次编译时，makefile 会自动下载编译工具链：

```
$ make
scons: Reading SConscript files ...
100.00% [#####] Speed:
9.260MB/S
Start install, wait half a minute please.
Congratulations!
```

工具链会被安装在当前用户的 `~/.thead` 目录下：

```
~/.thead$ ll
drwxr-xr-x 1 User user 4096 Apr 7 18:11 csky-abiv2-elf/
```

配置工具链路径到环境变量：

```
$echo "export PATH=$HOME/.thead/csky-abiv2-elf/bin:$PATH" >> ~/.bashrc
$source ~/.bashrc
```

2.3 Linux 调试环境安装

安装 Debug Server

- 登录[平头哥芯片开放社区](#)，进入栏目“技术部落 -> 资源下载 -> 工具 -> Debug Server”，下载最新 CSKY-DebugServer-Linux 版本，并解压
- 执行 `sudo sh ./CSKY-DebugServer-linux-*.sh -i`，开始安装
- 系统提示 “Do you agree to install the DebugServer[yes/no]”，输入 yes
- 系统提示设置安装路径 “Set full installing path:”，推荐安装到默认路径：直接回车
- 系统会提示 “This software will be installed to the default path: (/usr/bin/)?[yes/no/cancel]:”，输入 yes
- 安装成功后会提示：

```
Done !
You can use command “DebugServerConsole” to start DebugServerConsole!
(NOTE: The full path of ‘DebugServerConsole.elf’ is /usr/bin/C-Sky_DebugServer)
```

注意：安装过程中用户需要获取 sudo 权限。

Linux 虚拟机下安装 Debug Server

很多开发者习惯在 Windows 系统（或 Mac OS 系统）下安装 Linux 虚拟机来进行开发，常见的虚拟机有 WSL（Windows Subsystem for Linux），VMware，VirtualBox 等。然而在 Linux 虚拟机下，由于 Debug Server 没办法获取到 USB 设备信息，导致部分情况下 Debug Server 连接失败。

因此需要切换到 Windows 环境安装和使用 Debug Server。具体安装步骤，请参考《[Windows 调试环境安装](#)》。

运行 Debug Server

- 安装完成后，在任意目录下通过命令 `sudo DebugServerConsole`，来打开 DebugServer。打开后界面如下：


```

xie@xie-VirtualBox:~/workspace/test$ DebugServerConsole
+---+
| C-Sky Debugger Server (Build: Feb 19 2020) |
| User Layer Version : 5.8.18 |
| Target Layer version : 2.0 |
| Copyright (C) 2019 Hangzhou C-SKY Microsystems co.,ltd |
+---+
C-SKY: CKLink_Lite_V2, App_ver 2.16, Bit_ver null, Clock 2526.316KHz,
      2-wire, With DDC, Cache Flush On.
+-- CPU 0 --+
C-SKY CPU ID:
      WORD[0]: 0x04a11453
      WORD[1]: 0x10000000
      WORD[2]: 0x21400417
      WORD[3]: 0x30c00006
Target Chip Info:
      CPU Type is CK804FGT, in LITTLE Endian.
      L1ICache size 32KByte.
      Bus type is AHB-LITE32.
      Signoff date is 04/0107.
      Foundry is SMIC.
      Process is 55nm.

```

- DebugServer 连接成功如下:

```

***** DebuggerServer Commands List *****
singlestep/si
      execute single-step in the target
reset
      reset the target
pctrace
      show the PCFIFO(8 <= length <= 4096, default 8)
print/p
      print /x[d/f/o] *memory[$registers], eg p /x *0x20000000
      print target
set
      Set *memory[$registers]=value, eg. set $r0=0x1234
quit/q
      quit Debugger Server
help/h
      show help informations
CTRL+B ENTER
      switch input channel
*****
DebuggerServer$ Connection from 127.0.0.1 for CPU 0, at time 12:45.

```

2.4 应用开发

2.4.1 编译

以 CB5654 开发板为例, 进入开发板目录, 使用 make 命令开始编译:

```
$ cd solutions_cb5654/smart_speaker_cb5654
$ make clean; make
```

2.4.2 烧录

调试脚本

- 镜像的烧写通过 GDB 完成，需要先配置调试服务器的 IP 地址为本机地址。输入命令 ifconfig 获取本机地址（下例为：30.21.178.4）：

```
$ ifconfig
wifi0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 30.21.178.4  netmask 255.255.252.0  broadcast 30.21.179.255
    inet6   fe80::b031:5c7e:9a91:d8d2      prefixlen  64          scopeid
0x1<compat,link,site,host>
    ether 60:f2:62:77:45:d9  (Ethernet)
    RX packets 0  bytes 0 (0.0 B)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 0  bytes 0 (0.0 B)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- 在开发目录下（例如：solutions_cb5654/smart_speaker_cb5654）创建文件.gdbinit，设置 JTAG IP 地址为本机 IP 地址，端口号为 1025：

```
$ cat .gdbinit
target jtag jtag://30.21.178.4:1025
```

全部烧录

- 上电开发板，确保电源指示灯被点亮；运行 Debug Server 并保证与开发板连接成功
- 首次运行时，需要执行全部烧录命令，保证所有分区内容都被烧录：

```
$ make flashall
```

- 该命令会烧录 Bootloader 分区、算法分区、应用分区。
- **注意：**Bootloader 烧写若异常断电可能会导致设备无法引导。

应用固件烧录

- 大部分应用程序都运行应用分区上，修改应用程序代码后可以只烧写相应的应用程序固件，加快烧写速度。

```
$ make flash
```

烧录成功

- 烧录成功后，可以看到所有分区烧写进度都已至 100%。

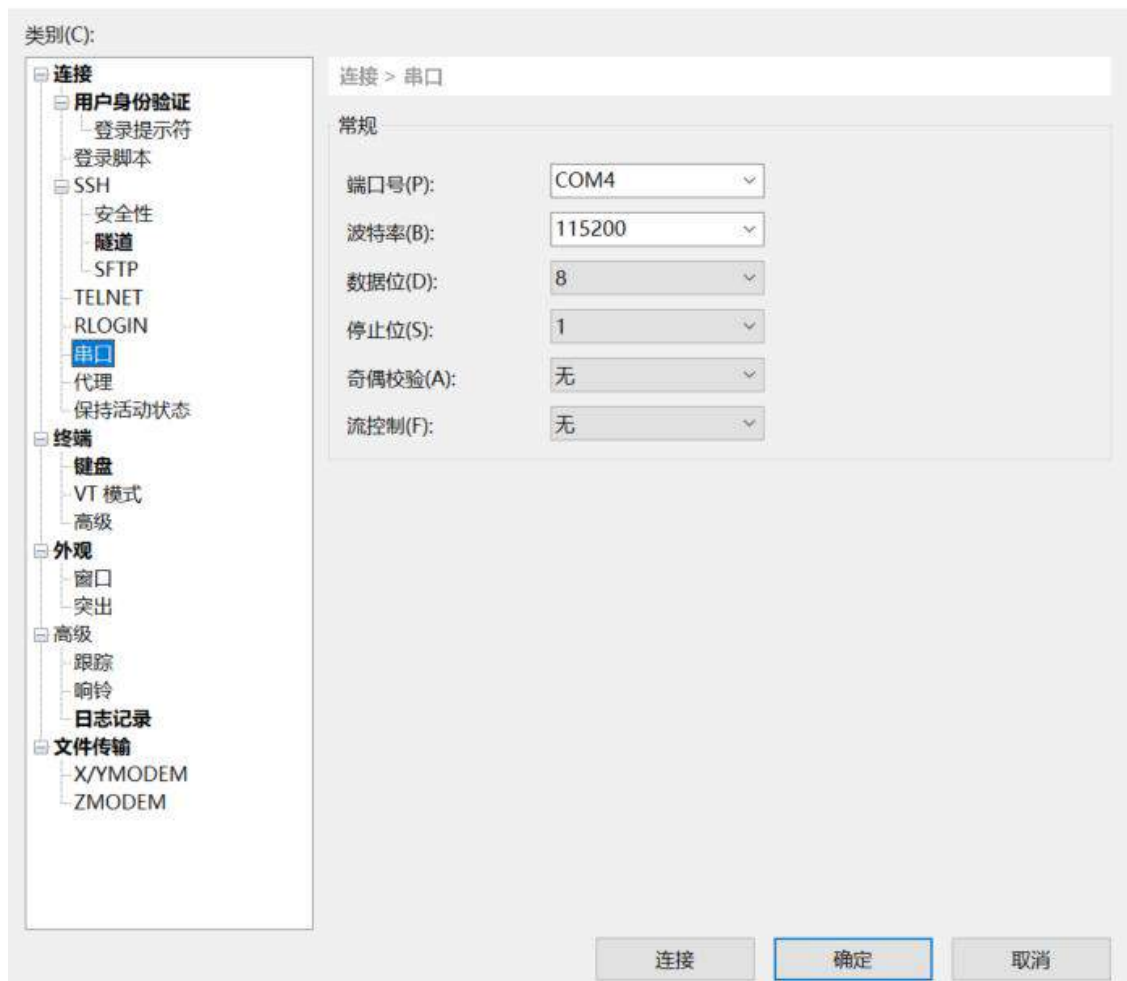
```
Program partition: bmtb          address: 0x8000000, size 192 byte
erasing...
program 08000000, 100%
Program partition: boot          address: 0x8001000, size 41592 byte
erasing...
program 08009000, 100%
Program partition: tee           address: 0x8011000, size 22048 byte
erasing...
program 08011000, 100%
Program partition: imtb          address: 0x8021000, size 8192 byte
erasing...
program 08021000, 100%
Program partition: prim          address: 0x8023000, size 1712796 byte
erasing...
program 081c3000, 100%
Program partition: cpu1          address: 0x8323000, size 233708 byte
erasing...
program 0835b000, 100%
Program partition: cpu2          address: 0x83a3000, size 388358 byte
erasing...
program 083fb000, 100%
Program partition: lpm           address: 0x85a7000, size 25508 byte
erasing...
program 085a7000, 100%
Program partition: lfs           address: 0x85c7000, size 143360 byte
erasing...
program 085e7000, 100%
Program partition: kp            address: 0x8ff000, size 576 byte
erasing...
program 08ff000, 100%
```

3. 例程运行

3.1 配置串口工具

Windows 串口工具

- 串口工具可以选择 [putty](#)/[SecureCRT](#)/[Xshell](#) 等常见 Windows 工具
- 以 Xshell 为例, 选择对应的端口号(安装完串口驱动后, 从设备管理器中可以查询到), 波特率设置为 115200, 数据位设为 8, 停止位设 1, 奇偶校验位和流控制位均设为 None:



Linux 串口工具

- Linux 下推荐使用 minicom 作为串口工具, 安装命令:

```
$sudo apt-get install minicom
```

- 启动 minicom 并配置参数:

```
$sudo minicom -s
//选择 'Serial port setup'
A - Serial Device      : /dev/ttyS0
E - Bps/Par/Bits       : 115200 8N1
F - Hardware Flow Control: No
```

3.2 开发板上电

- 按开发板上的复位按键，复位开发板，开发板正常上电 log 打印如下：

```
boot_v1.4

sdram ok
boot core...build: May 16 2020 14:29:18
BOOT[l] part_num:12
BOOT[E] otp cant find pk region
BOOT[l] verify [prim] ok
BOOT[l] load img & jump to [tee]

BOOT[l] all copy over..
BOOT[l] j m

BOOT[l] j 0x18000000

Tee v1.4.0 Initiaze done, Jun 22 2020 17:29:06
... 后续日志省略
```

3.3 配置 WiFi 密码

首次上电时，开发板没有配置 WiFi 用户名密码，无法连接无线路由器。用户可以通过串口命令行指令快速配置 WiFi 密码，步骤如下：

- 串口中输入指令分别设置 WiFi 参数(ssid, password)，其中 {ssid} 为 wifi 名称，{password} 为 wifi 密码，如果路由器没有密码，密码可随意填写
- **注意：所有命令都需要以换行符 \n 或\r\n 结尾**

```
kv set wifi_ssid {ssid}
kv set wifi_psk {password}
```

- 可以通过 kv get wifi_ssid/kv get wifi_psk 来获取设置的 ssid/密码，验证设置是否正确

```
>kv get wifi_ssid
kv get test
>kv get wifi_psk
kv get 123456
```

- 按开发板复位键，重启开发板
- 等待语音提示 正在启动，网络连接成功，并可以观察到串口打印输出 wifi 连接成功

```
[ 7.801532][1][netmgr ]start dhcp
[ 7.855701][1][netmgr ]IP: 192.168.1.103
```

也可以通过命令 `ifconfig` 检查网络连接状态，如果 Wi-Fi 连接失败，会显示 “WiFi Not connected”：

```
> ifconfig

wifi0   Link encap:WiFi  HWaddr 18:bc:5a:60:d7:f1
        inet addr:192.168.1.80
        GWaddr:192.168.1.1
        Mask:255.255.255.0
        DNS SERVER 0: 208.67.222.222
        WiFi Not connected
```

3.4 语音交互演示

开发板成功连网后，就可以进行语音交互演示了。

开发者可以通过唤醒词“宝拉宝拉”唤醒开发板，之后可以通过语音指令控制或询问开发板。

交互示例

```
人：宝拉宝拉
机：在
人：杭州天气
机：杭州今天阴转小雨，微风，15 摄氏度，出门别忘了得带伞哦！

人：宝拉宝拉
机：在
人：声音调大一点
机：好的
```


常用语音交互命令

控制命令：

- 声音大点/提高音量
- 声音小点/降低音量
- 停止播放
- 继续播放

交互命令：

- 讲个笑话
- 现在几点
- 天气预报/{城市}天气/明天天气/后天天气/下周天气

3.5 其他测试命令

更多测试命令，请参考解决方案文件夹下面的 README.md 文件。

I 智能语音应用开发指南

1. 概述

本章介绍智能语音终端 SDK 的软件开发方法。

名词解释

下面介绍本文中涉及的一些专有名词：

- PCM：脉冲编码调制，这里专指未经过编码的语音数据。直接从麦克风采集出来的语音即为 PCM 数据。
- KWS：关键词识别，识别特定的几个词语。在我们方案中，该关键词为“宝拉宝拉”，该过程在设备端实现。
- ASR：语音识别，将声音转化为文字的过程。该过程在云端完成。
- NLP：自然语言处理，将文字转化成语义的过程。该过程在云端完成。
- TTS：文本转语音，将文字转换成语音数据。该过程在云端完成。

SDK 目录介绍

下面是智能语音 SDK 的目录结构，表格中介绍了各个目录的功能。

文件/文件夹名称	内容描述
Boards	开发板配置信息，可以参考 智能语音开发板适配指南 了解如何对开发板进行配置
Chips	芯片驱动，包含了芯片驱动接口层（CSI）代码
Packages	SDK 开发包，包括了核心服务层和应用组件层
Makefile	用于编译的脚本
README.md	SDK 使用说明

app	解决方案代码
include	通用的头文件
package.yaml	SDK 配置文件
script	编译过程中用到的脚本
tools	利于开发调试的小工具

2. 核心组件介绍

本章介绍播放器组件、语音组件、云服务组件等核心组件。

- 播放器提供了语音播放功能，支持 PCM、WAV、MP3 等编码格式，可以实现内存音频、SD 卡音频、在线音频等多种音频流播放。支持音频的播放、停止、暂停、继续、音量控制等多种音频控制命令。通过适配对应的声卡播放通路，可以灵活得实现多种播放场景。
- 语音组件提供了语音唤醒及麦克风音频采集功能，可以将唤醒事件、VAD 事件、采集到的音频流等数据和事件实时传送给调用方，调用方结合云服务组件和播放器组件可以实现多种智能语音服务。
- 云服务组件提供了云端一体化的语音服务，封装了端侧和云侧的交互过程，开发者只需简单调用对应 API、处理回调函数，即可方便的获得云端 ASR/NLP 识别结果和 TTS 合成服务。
- 配网服务组件提供了配网框架及多种配网模块，封装了统一的应用接口，开发者无需关心配网实现，即可通过统一接口获取多种配网服务。
- 闹铃服务组件提供了闹铃服务，不论系统处于运行状态、低功耗状态还是待机休眠状态，都可准时产生闹铃回调。
- 按键服务组件提供 GPIO 高低电平、上升/下降沿等多种类型的按键扫描功能。开发者设置对应的按键参数、回调函数，按键服务会自动扫描并触发对应按键事件。

2.1 播放器服务

2.1.1 功能介绍

播放器服务组件支持播放、停止、音量等通用控制功能外，还支持常用的最小音量控制、音乐打断恢复及音乐渐变切换功能。支持 PCM、WAV、MP3 等编码格式。

应用示例中实现了如下播放器服务功能：

- 通知音播放。
- 在线音频播放
- 云端合成 TTS 流播放
- SD 卡音频播放

播放器服务组件的主要 API 如下：

API	说明
audi_player_init	播放器初始化
audi_player_play	播放音频
audi_player_pause	暂停
audi_player_resume	继续
audi_player_stop	停止
audi_player_mute	静音
audi_player_vol_adjust	调节音量，相对增减方式
audi_player_vol_set	调节音量，直接设置音量值 0~100
audi_player_vol_gradual	渐变方式调整音量
audi_player_vol_get	获取当前音量
audi_player_set_minvol	设置最小音量
audi_player_get_state	获取播放器状态
audi_player_resume_music	若音乐播放被通知音打断，可用该函数恢复

2.1.2 代码示例

初始化

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

void app_player_init()
{
    /*创建任务*/
    utask_t *task_media = utask_new("task_media", 2 * 1024, QUEUE_MSG_COUNT,
    AOS_DEFAULT_APP_PRI);
    /*初始化*/
    ret = aui_player_init(task_media, media_evt);
}
```

事件回调函数

在 aui_player_init 初始化时传入了播放器服务回调函数，之后所有的播放器事件都会通过回调函数通知给用户。

为方便应用开发，播放器中支持了两种播放类型，通知类型和音乐类型。通知可以打断音乐的播放，通知结束后，可以设置音乐是否自动恢复播放。回调函数中有播放类型和事件ID，用户可根据需要在事件中增加应用功能。

播放器事件如下表：

ID	事件
AUI_PLAYER_EVENT_START	开始播放
AUI_PLAYER_EVENT_ERROR	播放出错
AUI_PLAYER_EVENT_FINISH	播放完成

播放器类型如下表：

TYPE	类型
MEDIA_MUSIC	音乐
MEDIA_SYSTEM	通知

```
#include <media.h>

static void media_evt(int type, aui_player_evtid_t evt_id)
{
    /*播放音乐的事件处理*/
    switch (evt_id) {
        case AUI_PLAYER_EVENT_START:
            break;
        case AUI_PLAYER_EVENT_ERROR:
            break;
        case AUI_PLAYER_EVENT_FINISH:
            break;
        default:
            break;
    }
}
```

播放网络音乐

```
aui_player_play(MEDIA_MUSIC, "http://test_url/AudioTest1.mp3/", 1);
```

播放 SD 卡中的音频

```
aui_player_play(MEDIA_MUSIC, "file:///fatfs/1.mp3", 1);
```

播放 FIFO 中音频

```
aui_player_play(MEDIA_MUSIC, "fifo://test1", 1);
```

调节音量

```
/* 音量降低 10*/
aui_player_vol_adjust(MEDIA_ALL, -10)
/* 音量增加 10*/
aui_player_vol_adjust(MEDIA_ALL, 10)
```



```
/*音量设置到50*/
aui_player_vol_set(MEDIA_ALL, 50);
```

2.2 语音服务

2.2.1 功能介绍

语音服务组件提供关键词识别和语音数据的处理控制。输入麦克风的语音数据经过回音消除、降噪和关键词识别处理后再输出到应用层使用。

语音服务组件的主要 API 如下：

API	说明
aui_mic_start	启动 MIC 语音服务
aui_mic_stop	停止 MIC 语音服务
aui_mic_set_param	配置 MIC 语音服务参数
aui_mic_set_wake_enable	使能关键词算法检测
aui_mic_set_active	设置麦克风音频来源
aui_mic_control	控制 MIC 语音服务
aui_mic_get_www_data	获取唤醒二次确认数据
aui_mic_get_state	获取 MIC 语音服务状态
aui_mic_rec_start	启动 mic 网络录制
aui_mic_rec_stop	结束 mic 网络录制
aui_mic_init_kw_map	初始化离线命令词映射表
aui_mic_get_kw	获取离线命令词

2.2.2 代码示例

初始化

语音服务在一个独立的任务中运行。需要先创建一个任务，然后把任务句柄和回调函数传入初始化函数。

```
static int app_mic_init(int wwwv_enable)
{
    int ret;
    static voice_adpator_param_t voice_param;
    static mic_param_t param;

    /* 注册麦克风驱动 */
    voice_mic_register();

    /* 创建语音服务线程 */
    utask_t *task_mic = utask_new("task_mic", 3 * 1024, 20, AOS_DEFAULT_APP_PRI);
    ret = aui_mic_start(task_mic, mic_evt_cb);

    memset(&param, 0, sizeof(param));

    param.channels      = 5; /* 麦克风通道数 */
    param.sample_bits   = 16; /* 比特数 */
    param.rate          = 16000; /* 采样率 */
    param.sentence_time_ms = 600;
    param.noack_time_ms  = 5000;
    param.max_time_ms    = 10000;
    param.nsmode         = 0; /* 无非线性处理 */
    param.aecmode        = 0; /* 无非线性处理 */
    param.vadmode        = 0; /* 使能VAD */
    param.vadswitch      = 1;
    param.vadfilter      = 2;
    param.vadkws_strategy = 0;
    param.vadthresh_kws.vad_thresh_sp = -0.6;
    param.vadthresh_kws.vad_thresh_ep = -0.6;
    param.vadthresh_asr.vad_thresh_sp = -0.6;
    param.vadthresh_asr.vad_thresh_ep = -0.6;
    param.wwwv_enable    = wwwv_enable; /* 二次唤醒使能配置 */

    voice_param.pcm      = "pcmC0";
```

```
voice_param.cts_ms      = 20;
voice_param.ipc_mode    = 1;
voice_param.ai_param    = &param;
voice_param.ai_param_len = sizeof(mic_param_t);
param.ext_param1        = &voice_param;

/* 配置语音服务参数 */
aui_mic_set_param(&param);

if (wwwv_enable) {
    /* 二次唤醒使能初始化 */
    app_aui_wwwv_init();
}

return ret;
}
```

事件回调函数

在 app_mic_init 初始化时传入了语音服务回调函数，之后使用语音与设备交互时，所有的语音事件都会通过回调函数通知给用户。

语音事件如下表：

ID	事件
MIC_EVENT_SESSION_START	唤醒事件
MIC_EVENT_PCM_DATA	接收到麦克风语音数据
MIC_EVENT_SESSION_STOP	断句
MIC_EVENT_VAD	指示正在进行语音交互
MIC_EVENT_KWS_DATA	唤醒词数据

下例程介绍了事件回调函数的典型实现：

- 检测到唤醒词后会首先收到唤醒事件 MIC_EVENT_SESSION_START，调用函数 `aiu_mic_control(MIC_CTRL_START_PCM)` 打开语音数据接收。
- 之后程序会持续接收到 MIC_EVENT_PCM_DATA 事件，获取到回音消除后的语音数据。将此音频通过云服务接口推送到云端，用于 ASR/NLP 识别。
- 当断句事件 MIC_EVENT_SESSION_STOP 发生时，调用函数 `aiu_mic_control(MIC_CTRL_STOP_PCM)` 停止语音数据接收。

```
#include <yoc/mic.h>

static void mic_evt_cb(int source, mic_event_id_t evt_id, void *data, int size)
{
    switch (evt_id) {
        case MIC_EVENT_SESSION_START:
            /* 关键词唤醒，开始和云服务交互，打开语音数据接收 */
            ret = app_aui_cloud_start(do_wav);
            aui_mic_control(MIC_CTRL_START_PCM);
            break;
        case MIC_EVENT_PCM_DATA:
            /* 回音消除后的语音数据，推送到云端 */
            app_aui_cloud_push_audio(data, size);
            break;
        case MIC_EVENT_VAD:
            /* 检测到有效声音数据，可用于低功耗唤醒 */
            break;
        case MIC_EVENT_SESSION_STOP:
            /* 断句，停止和云服务交互，关闭语音数据接收 */
            app_aui_cloud_stop(1);
            aui_mic_control(MIC_CTRL_STOP_PCM);
            break;
        case MIC_EVENT_KWS_DATA:
            /* 接收到唤醒词数据 */
            break;
        default:;
    }
}
```

使能关键词检测

使用 `au_i_mic_set_wake_enable` 开启和关闭关键词检测。例如使用按键开始语音交互时，就通过调用该函数关闭关键词检测。

```
if (1 == asr_en) {  
    /* 使能关键词检测 */  
    au_i_mic_set_wake_enable(1);  
} else {  
  
    /* 关闭关键词检测 */  
    au_i_mic_set_wake_enable(0);  
}
```

2.3 云服务

2.3.1 功能介绍

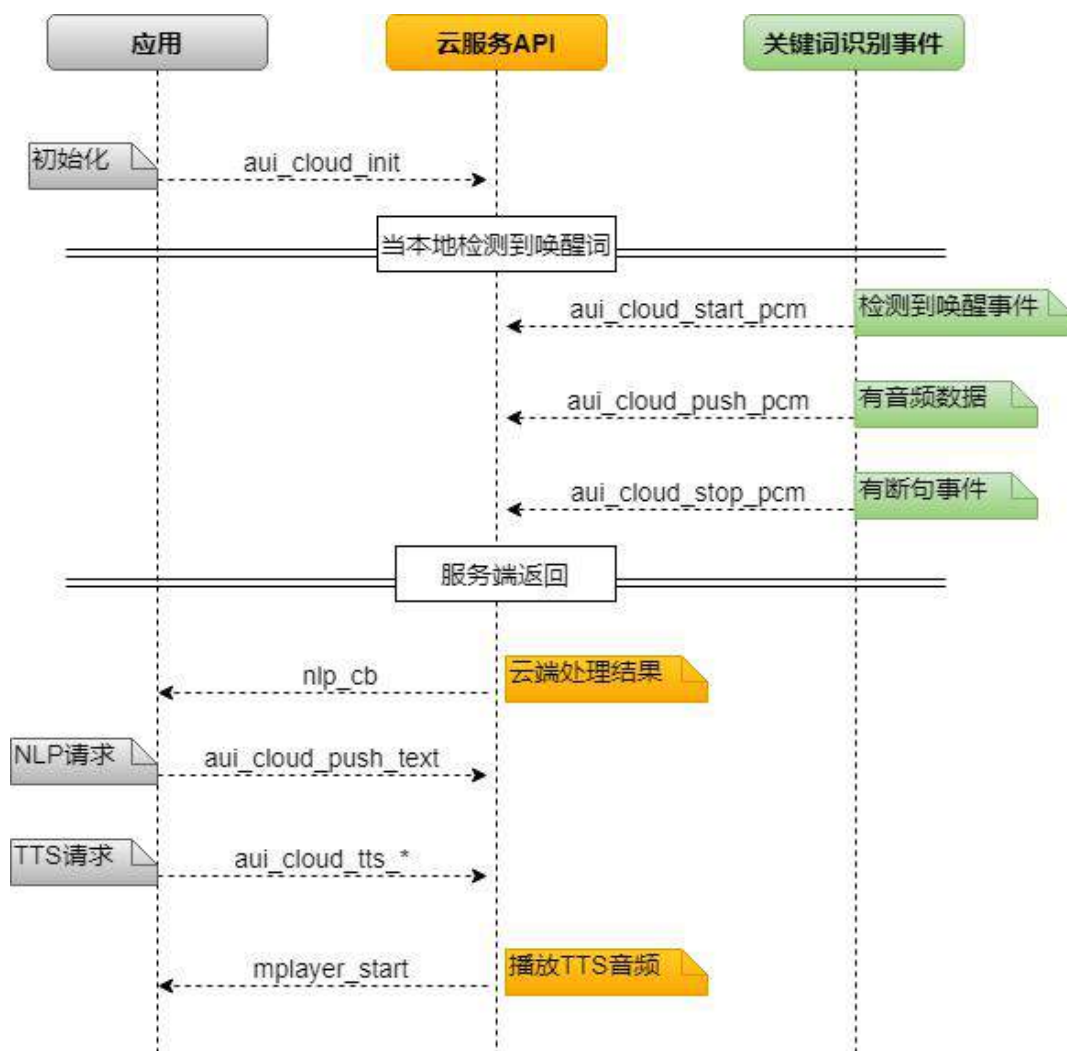
云服务组件提供应用与云端 ASR/NLP/TTS 服务交互的接口。调用对应服务 API 后，组件自动完成云端连接、鉴权、启动服务的过程，用户只需通过接口将需识别的音频或需合成的字符串传入，即可获得云端返回结果，设备端只需根据结果完成预定的应用行为。

云服务组件的主要 API 如下：

API	说明
<code>au_i_cloud_set_account</code>	设置 AUI 系统账号信息，在 <code>au_i_cloud_init</code> 之前调用
<code>au_i_cloud_init</code>	云服务初始化
<code>au_i_cloud_set_session_id</code>	设置云端会话 ID
<code>au_i_cloud_enable_wvv</code>	启动语音数据交互，在 <code>au_i_cloud_start_pcm</code> 前调用
<code>au_i_cloud_start_pcm</code>	启动语音数据交互
<code>au_i_cloud_push_pcm</code>	推送语音数据
<code>au_i_cloud_stop_pcm</code>	结束语音数据推送

aiui_cloud_force_stop	强制停止上云流程
aiui_cloud_init_wvv	使能云端唤醒二次确认流程
aiui_cloud_push_wvv_data	推送唤醒二次确认音频数据到云端
aiui_cloud_push_text	文本内容推送到云端进行 NLP 处理
aiui_cloud_start_tts	启动 TTS 语音合成服务
aiui_cloud_req_tts	向云端发送文本信息，请求 TTS 音频数据
aiui_cloud_stop_tts	停止 TTS 语音合成服务
aiui_textcmd_matchinit	向 AI 引擎输入文本数据，要求返回文本的 TTS 转换后的语音结果
aiui_textcmd_matchadd	控制命令设备端预解析
aiui_textcmd_find	输入字符串，返回对应的控制字符串
aiui_textcmd_matchnlp	预处理命令与 AUI NLP 的绑定

API 调用流程图



2.3.2 代码示例

初始化

程序初始化时，需使用初始化函数 `aui_cloud_init` 初始化云服务，并设定对应参数。

```

#include <yoc/aui_cloud.h>

static aui_t      g_aui_handler;
int app_aui_nlp_init()
{
    /* 添加账号 */
    cJSON *js_account_info = NULL;

```

```

cJSON_AddStringToObject(js_account_info, "device_uuid", device_uuid);
cJSON_AddStringToObject(js_account_info, "asr_app_key", asr_app_key);
cJSON_AddStringToObject(js_account_info, "asr_token", asr_token);
cJSON_AddStringToObject(js_account_info, "asr_url", asr_url);
cJSON_AddStringToObject(js_account_info, "tts_app_key", tts_app_key);
cJSON_AddStringToObject(js_account_info, "tts_token", tts_token);
cJSON_AddStringToObject(js_account_info, "tts_url", tts_url);

au_i_config_t cfg;
cfg.per          = "aixia";
cfg.vol          = 100;      /* 音量 0~100 */
cfg.spd          = 0;       /* -500 ~ 500*/
cfg.pit          = 0;       /* 音调*/
cfg.fmt          = 2;       /* 编码格式, 1: PCM 2: MP3 */
cfg.srate        = 16000;   /* 采样率, 16000 */
cfg.tts_cache_path = NULL;   /* TTS 内部缓存路径, NULL: 关闭缓存功能 */
cfg.cloud_vad     = 1;      /* 云端 VAD 功能使能, 0: 关闭; 1: 打开 */
cfg.js_account    = s_account_info;
cfg.nlp_cb        = au_i_nlp_cb;
g_au_i_handler.config = cfg;

au_i_asr_register_mit(&g_au_i_handler);
au_i_tts_register_mit(&g_au_i_handler);

ret = au_i_cloud_init(&g_au_i_handler);
au_i_nlp_process_add(&g_au_i_nlp_process, au_i_nlp_proc_mit);
}

```

语音数据推送

在进行 ASR 识别时，需推送语音数据到云端。语音数据的推送流程主要在语音服务回调中处理。启动、推送、停止分别在语音服务事件 MIC_EVENT_SESSION_START、MIC_EVENT_PCM_DATA、MIC_EVENT_SESSION_STOP 中控制。具体代码已经在 2.2.2 代码示例的“事件回调函数”一节有过介绍。

事件回调函数

云端下发的 ASR 和 NLP 结果通过 au_i_nlp_cb 返回给用户。

```
#include <yoc/au_i_coud.h>
```



```

/* 处理云端反馈的 ASR/NLP 数据，进行解析处理 */
static void aui_nlp_cb(const char *json_text)
{
    /* 处理的主入口，具体处理见初始化注册的处理函数 */
    int ret = aui_nlp_process_run(&g_aui_nlp_process, json_text);
    switch (ret) {
        case AUI_CMD_PROC_ERROR:
            /* 没听清楚 */
            ...
            break;
        case AUI_CMD_PROC_NOMATCH:
            /* 不懂 */
            ...
            break;
        case AUI_CMD_PROC_MATCH_NOACTION:
            /* 不懂 */
            ...
            break;
        case AUI_CMD_PROC_NET_ABNORMAL:
            /* 网络问题 */
            ...
            break;
        default:;
    }
}

```

请求 TTS 服务

应用解析云端下发数据得到 TTS 文本之后，需要向云端请求 TTS 播放服务。代码段如下：

```

/* TTS 回调函数 */
static void aui_tts_stat_cb(aui_tts_state_e stat)
{
    switch(stat) {
        case AUI_TTS_INIT:
            /* TTS 初始化状态 */
            ...
            break;
        case AUI_TTS_PLAYING:
            /* TTS 正在播放 */
            ...

```

```
        break;
    case AUI_TTS_FINISH:
        /* TTS 播放完成 */
        ...
        break;
    case AUI_TTS_ERROR:
        /* TTS 播放失败 */
        ...
        break;
    }
}

int app_aui_cloud_tts_run(const char *text, int wait_last)
{
    /* 注册回调函数 */
    aui_cloud_set_tts_status_listener(&g_aui_handler, aui_tts_stat_cb);
    /* 请求TTS 播放服务, 其中text 是TTS 文本 */
    return aui_cloud_req_tts(&g_aui_handler, text, NULL);
}
```

2.4 配网服务

配网服务由配网框架和配网模块组成，配网模块完成具体的配网功能，如一键配网、设备热点配网等，而配网框架封装了配网模块，为用户提供了统一的配网调用入口，简化了编程过程。

2.4.1 配网模块介绍

本服务提供设备的 WiFi 网络连接配置功能。支持两种配网方式，设备热点配网方式及阿里云生活物联网平台 SDK 配网方式。

设备热点网页配网

进入配网模式后，设备会辐射出一个 WiFi 热点。手机连接该热点后，会自动弹出配网页面，用户填写需连接的路由器 SSID 和密码信息。信息提交后，设备端会收到 SSID 和密码并连接路由器。如连接成功，设备会语音播报并广播配网成功信息，手机端监测到成功信息并弹出提示。如超时未成功，设备会语音播报配网超时。

生活物联网平台 SDK 配网

生活物联网平台 SDK 为阿里云生活物联网平台提供的设备端 SDK，包含 WiFi 配网、云端连接及设备控制功能。配合生活物联网平台手机端 SDK，可以形成多样化的网络解决方案。

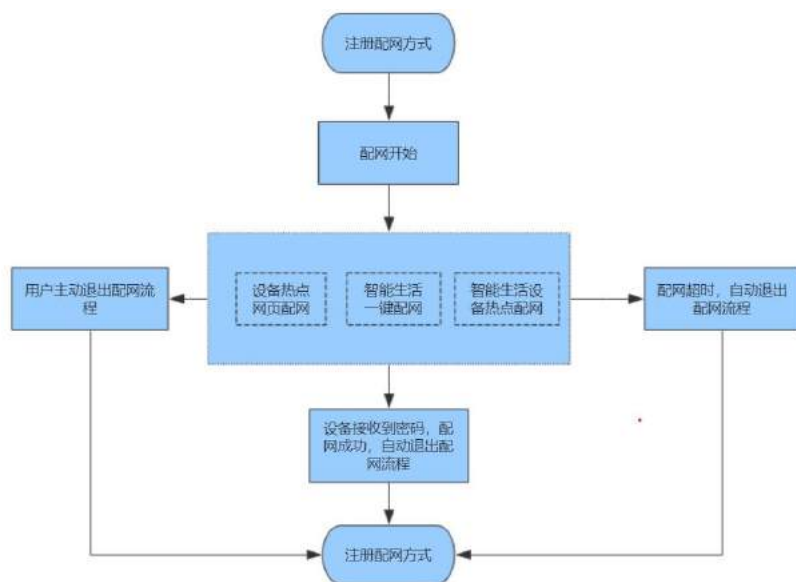
生活物联网平台 SDK 的 WiFi 配网方式支持一键配网和设备热点配网两种方式，可互为补充，其操作特点如下：

- 一键配网：手机无需切换路由，用户在 APP 上输入所连路由器 SSID 和密码，手机进入一键配网模式后，会不停发送包含 SSID 和密码的 802.11 射频加密报文。设备截获报文并解密后，即可连接路由器。此配网过程十分方便迅速，但可能存在兼容性问题。
- 设备热点配网：与设备热点网页配网原理类似，不过手机端需使用 APP 发送 SSID 和密码。设备接收到 SSID 和密码后自动连接路由器。此配网方式兼容性好，适合作为备用方案。

2.4.2 配网框架介绍

配网框架提供了配网模块的注册、配网的启动、停止等接口，为底层不同的配网模块提供了统一的接口函数。

在配网流程启动前，我们需要先注册配网方法，该动作接口需要适配层实现。启动完成后应用层将通过调用配网框架提供接口 `wifi_prov_start` 来进入配网状态。配网状态退出有三种方式，超时自动退出，收到配网结果自动退出，调用接口 `wifi_prov_stop` 主动退出。框架接口 `wifi_prov_start`，支持多个配网方法同时启动，如果几个方法没有互斥。退出配网状态时，所有启动的配网将一起退出。



配网服务组件的主要 API 如下：

API	说明
wifi_prov_start	启动配网服务
wifi_prov_stop	停止配网服务
wifi_prov_get_method_id	通过名称获取配网 ID

2.4.3 代码示例

初始化

程序初始化时，首先需要注册所需配网模块，可同时注册多种：

```
#include <softap_prov.h>
#include <wifi_provisioning.h>

/*注册设备热点网页配网服务，参数为 SSID 的前缀*/
wifi_prov_softap_register("YoC");

/*注册生活物联网平台配网服务*/
wifi_prov_sl_register();
```

启动配网

需要配网时，启动对应的配网模块，进入配网流程。

```
#include <softap_prov.h>
#include <wifi_provisioning.h>

/*启动配网服务，超时时间120 秒*/
wifi_prov_start(wifi_prov_get_method_id("softap"), wifi_pair_callback, 120);
```

事件回调函数

配网成功或失败都会调用该函数，函数的参数中 event 变量可以确认结果，如果由多种配网同时启动可以从 method_id 确认哪种配网，配网参数从 result 返回，可使用该参数去连接网络。

```
#include <wifi_provisioning.h>

static void wifi_pair_callback(uint32_t method_id, wifi_prov_event_t event,
    wifi_prov_result_t *result)
{
    if (event == WIFI_PROV_EVENT_TIMEOUT) {
        /*配网超时*/
        LOGD(TAG, "wifi pair timeout...");
    } else if (event == WIFI_PROV_EVENT_GOT_RESULT) {
        /*配网成功，获取配网参数*/
        LOGD(TAG, "wifi pair got passwd ssid=%s password=%s...", result->ssid, result->password);
    }
}
```

停止配网

调用该函数停止配网流程，若启动多种配网也会全部停止，退出配网状态。

```
#include <wifi_provisioning.h>

/*停止配网*/
wifi_prov_stop();
```

2.5 闹铃

2.5.1 功能介绍

闹铃服务组件提供设备闹铃提醒的功能。用户设置对应闹铃模式及回调函数后，闹铃到时，通过回调函数向用户抛出对应闹铃事件。

闹铃服务具有如下特点：

- 可设置“仅响一次”、“每天”、“每周”或“工作日”模式
- 支持最多 5 个闹铃
- 支持待机状态下唤醒
- 秒级精度

闹铃服务组件的主要 API 如下:

API	说明
clock_alarm_init	闹铃初始化处理
clock_alarm_set	闹铃设置，用于新增、修改或删除一个闹铃
clock_alarm_get	获取指定 id 的闹铃信息
clock_alarm_get_status	获取闹铃状态

2.5.2 代码示例

初始化

系统启动时，调用函数 clock_alarm_init 初始化闹铃服务，初始化函数会从系统 Flash 中获取数据，更新闹铃信息和状态。

```
#include <clock_alarm.h>
#include <rtc_alarm.h>

void main()
{
    ...
    clock_alarm_init(app_clock_alarm_cb);
    ...
}
```

事件回调函数

闹铃到时，系统会调用 app_clock_alarm_cb 函数，用户可在该函数中处理对应闹铃事件。若设置了多个闹铃，参数 clock_id 来指示哪个闹铃。

```
#include <cloc_alarm.h>
```

```
/* 处理闹铃到时提醒 */
static void app_clock_alarm_cb(uint8_t clock_id)
{
    LOGI(TAG, "clock_id %d alarm cb handle", clock_id);

    char url[] = "http://www.test.com/test.mp3"; //url 示例
    /* 播放闹铃音乐 */
```

```
    mplayer_start(SOURCE_CLOUD, MEDIA_SYSTEM, url, 0, 1);
}
```

闹铃设置

用户新增一个闹铃，需要设置闹铃的模式（即一次、工作日、每周、每天），输入具体的闹铃时分秒信息。

```
clock_alarm_config_t cli_time;

cli_time.period = CLOCK_ALARM_PERIOD_WORKDAY;
cli_time.hour = 7;
cli_time.min = 30;
cli_time.sec = 0;

/* 新增闹钟 */
id = clock_alarm_set(0, &cli_time);

/* 修改闹钟, clock_id 是被修改闹钟 id 号 */
id = clock_alarm_set(clock_id, &cli_time);

/* 删除闹钟, clock_id 是被删除闹钟 id 号 */
clock_alarm_set(clock_id, NULL);
```

2.6 按键服务

2.6.1 功能介绍

按键服务组件提供多种类型的按键扫描功能。用户可添加对应的按键类型、引脚号、触发阈值，启动后会周期性扫描按键引脚，当键值满足触发条件后，通过回调函数向用户抛出对应按键事件。

按键服务具有如下特点：

- 最多支持 10 个按键
- 支持 GPIO 高低电平按键
- 支持 GPIO 电平变化检测
- 支持单一按键和组合按键（2 个键组合），支持短按、长按

按键服务组件的主要 API 如下：

API	说明
button_srv_init	初始化服务
button_init	初始化单一按键
button_param_set	配置按键参数
button_combination_init	初始化组合按键

2.6.2 代码示例

初始化

按键服务的初始化和配置需要四个步骤：

- 定义单一按键表和组合按键表
- 创建 task 和一个消息队列，用于接收按键消息，并将用户处理函数注册到 task 中
- 初始化按键服务和按键表
- 根据需要配置按键参数（例如超时时间等）

按键表定义

```
/* 单一按键表 */
const static button_config_t button_table[] = {
    {APP_KEY_MUTE, (PRESS_UP_FLAG | PRESS_LONG_DOWN_FLAG), button_evt, NULL, BUTTON_TYPE_GPIO, "mute"},
    {APP_KEY_VOL_INC, (PRESS_UP_FLAG | PRESS_LONG_DOWN_FLAG), button_evt, NULL, BUTTON_TYPE_GPIO, "inc"},
    {APP_KEY_VOL_DEC, (PRESS_UP_FLAG | PRESS_LONG_DOWN_FLAG), button_evt, N
```



```

ULL, BUTTON_TYPE_GPIO, "dec"},
    {APP_KEY_STANDBY, (PRESS_UP_FLAG | PRESS_LONG_DOWN_FLAG), button_evt, N
ULL, BUTTON_TYPE_GPIO, "standby"},
    {0, 0, NULL, NULL},
};

/* 组合按键表 */
const static button_combinations_t bc_table[] = {
    {
        .pin_name[0] = "mute",
        .pin_name[1] = "inc",
        .evt_flag = PRESS_LONG_DOWN_FLAG,
        .pin_sum = 2,
        .tmout = 500,
        .cb = bc_evt,
        .priv = NULL,
        .name = "mute&inc_long"
    },
    {
        .pin_name[0] = "mute",
        .pin_name[1] = "dec",
        .evt_flag = PRESS_LONG_DOWN_FLAG,
        .pin_sum = 2,
        .tmout = 500,
        .cb = bc_evt,
        .priv = NULL,
        .name = "mute&dec_long"
    },
    ...
}

```

创建任务和消息队列

```

aos_task_t task;
static aos_queue_t s_queue;
static uint8_t s_q_buffer[sizeof(evt_data_t) * MESSAGE_NUM];

aos_queue_new(&s_queue, s_q_buffer, MESSAGE_NUM *
sizeof(evt_data_t), sizeof(evt_data_t));
aos_task_new_ext(&task, "b-press", button_task_thread, NULL, 4096,
AOS_DEFAULT_APP_PRI + 4);

```

初始化按键服务，并配置按键参数

```
button_task();
button_srv_init();
button_init(button_table);
button_param_t pb;
button_param_cur("mute", &pb);
pb.ld_tmout = 2000;
button_param_set("mute", &pb);
button_param_set("inc", &pb);
button_param_set("dec", &pb);

button_combination_init(bc_table);
```

事件回调函数

```
#include <yoc/adc_key_srv.h>
static void button_task_thread(void *arg)
{
    evt_data_t data;
    unsigned int len;

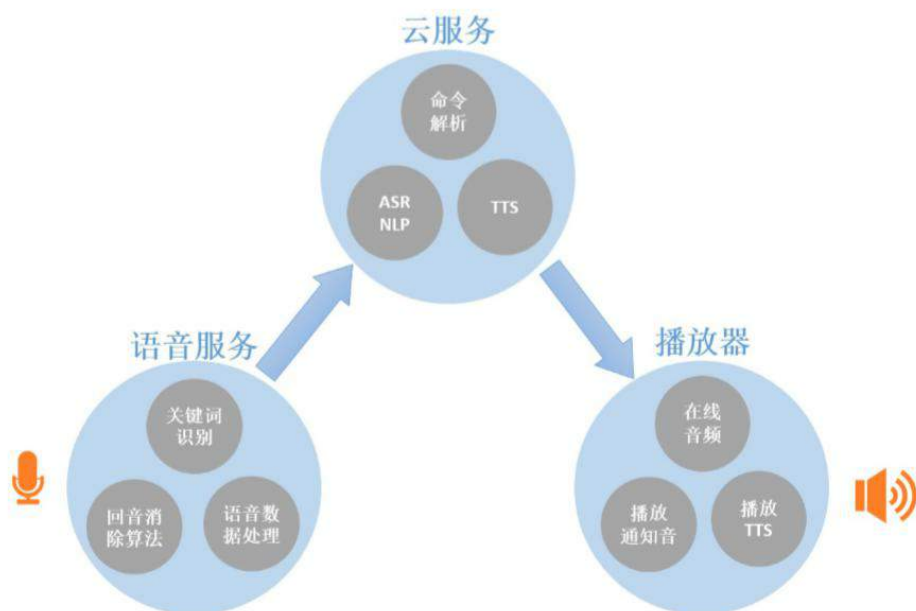
    while (1) {
        aos_queue_rcv(&s_queue, AOS_WAIT_FOREVER, &data, &len);

        if (strcmp(data.name, "mute") == 0) {
            if (data.event_id == BUTTON_PRESS_LONG_DOWN) {
                /* mute 长按 */
                ...
            } else if (data.event_id == BUTTON_PRESS_UP) {
                /* mute 短按 */
                ...
            }
        } else if (strcmp(data.name, "inc") == 0) {
            /* inc 按键 */
            ...
        } else if (strcmp(data.name, "dec") == 0) {
            /* dec 按键 */
            ...
        } else if (strcmp(data.name, "standby") == 0) {
            /* standby 按键 */
            ...
        } else if (data.event_id == BUTTON_COMBINATION) {
```

```
/* 组合按键 */
...
if (strcmp(data.name, "mute&inc_long") == 0) {
    /* mute 和 inc 组合长按 */
    ...
} else if (strcmp(data.name, "mute&dec_long") == 0) {
    /* mute 和 dec 组合长按 */
    ...
}
}
```

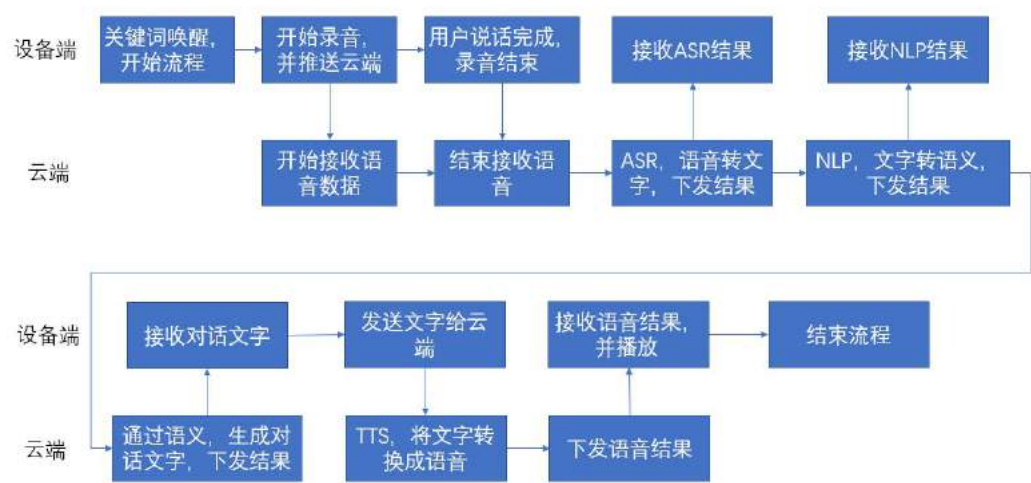
3. 应用示例讲解

上文中已经介绍了播放器、语音、云服务三个核心组件的功能和使用方法。三者间关系如下图，应用通过语音服务获取到音频输入，通过云服务推送到云端进行 ASR/NLP 识别，再将结果进行 TTS 合成，生成的音频通过播放器播放出来。



3.1 方案介绍

智能语音应用包含网络功能，音频播放服务、语音服务(关键词识别和数据交互)、云服务等。整个语音交互流程如下图。



通用交互流程

- 使用者说出“宝拉宝拉，今天天气怎么样”时，语音服务识别出“宝拉宝拉”这个关键词，产生唤醒事件。
- 语音唤醒事件中控制开始录音。
- 使用者继续说“今天天气怎么样”，语音数据回调中推送录音音频给云端。
- 使用者说完，产生语音断句事件，事件中停止录音和云端推送。
- 云端依次执行 ARS->NLP->TTS 三个服务：先将语音数据转化成文字“今天天气怎么样”，然后通过 NLP 算法，理解语义，得知是天气查询，最后通过云端技能接口获得有关的天气信息，再将天气信息的文字转成语音音频，推送给设备端。
- 设备端收到语音音频后调用播放器播放。

流程差异

当然并不是所有的语音交互都遵循这个流程，例如调整音量等命令类的语音交互，当使用者说出“声音小一点”时，设备端只需要获取到云端下发的 NLP 结果，判断为设备控制命令，就可以直接控制设备行为，而无需后续 TTS 流程。

同时还需认识到，使用不同的云端服务，调用的过程也会有区别。例如：

- 有些云服务器在进行 ASR/NLP 识别过程中不会单独下发 ASR 结果，而是直接下发最终的 NLP 结果。
- 不同云端 NLP 结果的封装方式差异很大。
- 有些云端不需要设备端参与 TTS 过程，会直接下发语音数据。

3.2 入口函数

应用的入口函数文件如下：

app/src/app_main.c

入口函数主要有，板级、系统级、音频驱动、网络、播放器、语音服务、云服务等模块的初始化。初始化完成之后，不同的服务和任务就在各自的线程中独立运行。

```
void main()
{
    board_base_init();
    yoc_base_init();

    LOGI(TAG, "Build:%s,%s",__DATE__, __TIME__);

    /* 系统事件处理 */

    sys_event_init();
    app_sys_init();

    /* 初始化LED 灯 */
    app_status_init();

    /* 音频数据采集 */
    board_audio_init();

    /* 初始化PWM LED 灯 */
#ifdef APP_PWM_EN && APP_PWM_EN
    app_pwm_led_init();
#endif

    /* 低功耗初始化 */
    app_lpm_init();

    /* 启动播放器 */
    mplayer_init(4 * 1024, media_evt, 20);

    /* 配置EQ */
    aui_player_sona_config(sona_aef_config, sona_aef_config_len);

    /* 启动麦克风服务 */
    app_mic_init(1);

    /* 开启功放 */
```

```
app_speaker_mute(0);

/* 网络初始化 */
wifi_mode_e mode = app_network_init();

if (mode != MODE_WIFI_TEST) {
    if (mode != MODE_WIFI_PAIRING &&
        app_sys_get_boot_reason() != BOOT_REASON_WIFI_CONFIG &&
        app_sys_get_boot_reason() != BOOT_REASON_WAKE_STANDBY) {
        local_audio_play(LOCAL_AUDIO_STARTING);
    }
}
#if defined(APP_FOTA_EN) && APP_FOTA_EN
    /* FOTA 升级初始化 */
    app_fota_init();
#endif

    if (g_fct_mode) {
        /* 产测初始化 */
        fct_case_init();
    }

    /* 交互系统初始化 */
    app_aui_nlp_init();
    app_text_cmd_init();
}

/* 按键初始化 */
app_button_init();

/* LED 状态初始化 */
app_set_led_state(LED_TURN_OFF);

/* 命令行测试命令 */
cli_reg_cmds();

return;
}
```

3.3 事件处理机制

系统中驻留一独立任务来处理事件，用户可通过订阅接口来获取系统事件。下面通过网络模块的实现代码来讲解该机制的使用方法。

代码路径

app/src/app_net.c

事件回调函数

```
static void user_local_event_cb(uint32_t event_id, const void *param, void
*context)
{
    if ((wifi_is_pairing() == 0) && wifi_network_initiated()) {
        network_normal_handle(event_id, param);
        network_reset_handle(event_id);
    } else {
        LOGE(TAG, "Critical network status callback %d", event_id);
    }
}
```

订阅系统事件

通过函数 app_net_init 进行网络的初始化并订阅网络事件，注册用户回调函数 user_local_event_cb。在程序中，当网络事件发生时，事件处理机制会调用 user_local_event_cb 函数进行网络事件的处理。

网络事件如下：

事件	说明
EVENT_NETMGR_GOT_IP	连接路由器成功
EVENT_NETMGR_NET_DISCON	与路由器断开连接

```
#include <aos/aos.h>
#include <yoc/netmgr.h>
#include <yoc/eventid.h>
#include <devices/wifi.h>
static wifi_mode_e app_net_init(void)
{
    char ssid[32 + 1] = {0};
    int ssid_len = sizeof(ssid);
    char psk[64 + 1] = {0};
    int psk_len = sizeof(psk);

    /* 系统事件订阅 */
    event_subscribe(EVENT_NETMGR_GOT_IP, user_local_event_cb, NULL);
    event_subscribe(EVENT_NETMGR_NET_DISCON, user_local_event_cb, NULL);

    /* 使用系统事件的定时器 */
    event_subscribe(EVENT_NTP_RETRY_TIMER, user_local_event_cb, NULL);
    event_subscribe(EVENT_NET_CHECK_TIMER, user_local_event_cb, NULL);
    event_subscribe(EVENT_NET_LPM_RECONNECT, user_local_event_cb, NULL);

    aos_kv_get("wifi_ssid", ssid, &ssid_len);
    aos_kv_get("wifi_psk", psk, &psk_len);

    if (strlen(ssid) == 0) {
        wifi_pair_start();
        return MODE_WIFI_PAIRING;
    } else {
        wifi_network_init(ssid, psk);
    }
    return MODE_WIFI_NORMAL;
}
```

自定义事件

除了系统事件以外，事件机制也允许用户自定义事件，我们以 EVENT_NTP_RETRY_TIMER 为例。

app_main.h 中统一管理所有用户自定义事件

```
#define EVENT_NTP_RETRY_TIMER      (EVENT_USER + 1)
#define EVENT_NET_CHECK_TIMER     (EVENT_USER + 2)
#define EVENT_NET_NTP_SUCCESS     (EVENT_USER + 3)
#define EVENT_NET_LPM_RECONNECT  (EVENT_USER + 4)
```


下面的代码是网络事件处理函数，回调中获取到 EVENT_NETMGR_GOT_IP 事件后，发送 EVENT_NTP_RETRY_TIMER 消息启动 NTP 对时，然后在回调的 EVENT_NTP_RETRY_TIMER 事件分支中调用 ntp_sync_time 进行对时，如果对时失败，调用 event_publish_delay 发送 EVENT_NTP_RETRY_TIMER 延时消息，一定时间后重新对时。

```
#include <yoc/netmgr.h>
#include <yoc/eventid.h>
static void network_normal_handle(uint32_t event_id, const void *param)
{
    switch (event_id) {
        case EVENT_NETMGR_GOT_IP: {
            /* 启动NTP对时 */
            event_publish(EVENT_NTP_RETRY_TIMER, NULL);
        } break;
        case EVENT_NETMGR_NET_DISCON: {
            LOGD(TAG, "Net down");
            /* 不主动语音提示异常，等有交互再提示 */
            internet_set_connected(0);
        } break;
        case EVENT_NTP_RETRY_TIMER:
            if (ntp_sync_time(NULL) == 0) {
#ifdef CONFIG_RTC_EN
                /* 网络对时成功,同步到RTC中 */
                rtc_from_system();
#endif
                if (wifi_internet_is_connected() == 0){
                    /* 同步到时间,确认网络成功,提示音和升级只在第一次启动 */
                    internet_set_connected(1);

                    app_status_update();
                    local_audio_play(LOCAL_AUDIO_NET_SUCC);
                    event_publish(EVENT_NET_NTP_SUCCESS, NULL);
                }
            } else {
                /* 同步时间失败重试 */
                event_publish_delay(EVENT_NTP_RETRY_TIMER, NULL, 6000);
            }
            break;
        default:
            break;
    }
}
```

智能语音开发板适配指南

本章将介绍开发板 CB5631 和 CB5654 的硬件适配方法。

1. CB5631 适配指南

1.1 目录结构

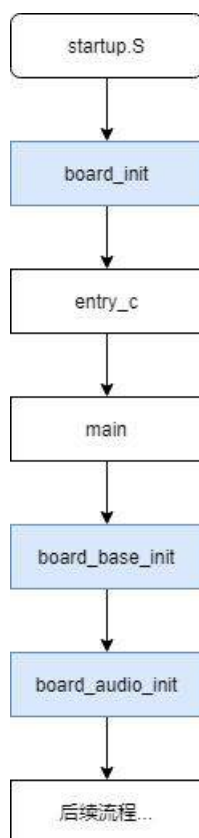
开发板适配代码位于 SDK 的 boards 目录下面，CB5631 的板级目录结构如下：

```
boards/csky/cb5631
├── README.md
├── bootimgs - 存放除了 prim 分区外的其他固件，包括 boot/cpu1/cpu2/lpm/tee 等固件
├── configs - 系统分区配置以及编译链接配置
├── include - 引脚配置定义和板级宏定义
├── package.yaml - 组件配置文件
└── src - 板级配置代码文件
```

1.2 初始化

1.2.1 初始化流程

应用分区的初始化流程如下：



startup.S 是系统的启动文件，系统启动时首先会跳转到该文件。在 startup.S 流程中，会调用函数 board_init() 进行板级引脚配置。接下来会通过 entry_c 进入 main 函数，在 main 函数里面调用 board_base_init() 和 board_audio_init()，进一步配置其他板级参数。板级配置完成后，才会开始后续的初始化流程。

1.2.2 引脚复用配置

芯片级的初始化和引脚的复用配置在 board_init 函数完成。board_init 代码路径如下：

```
boards/csky/cb5631/src/pinmux_init.c
```

代码示例

本示例配置了串口调试引脚，Wi-Fi/蓝牙通讯引脚。实际开发过程中，开发者可以根据需要灵活进行配置。

```
void board_pinmux_config(void)
{
    /* console */
    drv_pinmux_config(CONSOLE_TXD, CONSOLE_TXD_FUNC);
```

```
drv_pinmux_config(CONSOLE_RXD, CONSOLE_RXD_FUNC);
/* wifi sdio */
drv_pinmux_config(WIFI_SDIO_CMD, WIFI_SDIO_CMD_FUNC);
drv_pinmux_config(WIFI_SDIO_CLK, WIFI_SDIO_CLK_FUNC);
drv_pinmux_config(WIFI_SDIO_DAT0, WIFI_SDIO_DAT0_FUNC);
drv_pinmux_config(WIFI_SDIO_DAT1, WIFI_SDIO_DAT1_FUNC);
drv_pinmux_config(WIFI_SDIO_DAT2, WIFI_SDIO_DAT2_FUNC);
drv_pinmux_config(WIFI_SDIO_DAT3, WIFI_SDIO_DAT3_FUNC);

/* BT */
drv_pinmux_config(BT_UART_TXD, BT_UART_TXD_FUNC);
drv_pinmux_config(BT_UART_RXD, BT_UART_RXD_FUNC);
}

void board_init(void)
{
    board_pinmux_config();
}
```

1.2.3 其他板级初始化

其他板级的初始化包含两个接口，board_base_init 为板级小系统的初始化；board_audio_init 提供音频功能的初始化。

代码路径

boards/csky/cb5631/src/base_init.c

代码示例

板级小系统的初始化，主要包括：

- itcm 加载初始化，将部分启动代码加载到 itcm，可以加快启动速度；
- 注册串口驱动，CB5631 开发板使用两个串口；其他开发板可以根据真实情况进行注册；

注意：uart_csky_register 函数的参数是从零开始的 ID，实际的硬件串口号需减一

- spiflash 初始化
- i2c 初始化
- 分区表初始化

- cpu1/cpu2 初始化
- 配置 RTC 时钟

```
void board_base_init(void)
{
    itcm_code_load(); // itcm 加载初始化
    uart_csky_register(0); // 注册串口号 0
    uart_csky_register(1); // 注册串口号 1

    spiflash_csky_register(0); // spiflash 初始化
    iic_csky_register(0); // i2c 初始化

    console_init(CONSOLE_IDX, 115200, 512); // 串口初始化

    /* load partition */
    int ret = partition_init(); //分区表初始化
    if (ret <= 0) {
        LOGE(TAG, "partition init failed");
    } else {
        cpu1_init(); //cpu1 初始化
        cpu2_init(); //cpu2 初始化

        LOGI(TAG, "find %d partitions", ret);
    }

    /* RTC CLK setting */
    extern void drv_lclk_select_src(clk_src_e);
    /* 设置内部 RC 晶振，用于 RTC 时钟 */
    drv_lclk_select_src(ELS_CLK);
    drv_set_rtc_freq(CONFIG_RTC_CLK);
}
```

音频功能的初始化，主要是注册声卡驱动，该函数一般无需修改。

```
void board_audio_init()
{
    int id_list[5] = {0, 1, 4, EMPTY_INPUT_CHANNEL, EMPTY_INPUT_CHANNEL};
    snd_pangu_config_t config = {id_list, sizeof(id_list) / sizeof(int)};
    config.dac_db_max = -15;
    config.dac_db_min = -45;

    snd_card_pangu_register(&config);
}
```

1.3 参数配置

1.3.1 配置方法

本章主要介绍了引脚参数配置和其他配置。引脚需要根据芯片引脚定义和开发板原理图来进行配置，只有引脚配置完成后，才能保证上节介绍的引脚复用配置顺利运行。

以串口为例，配置方法如下：

- 通过硬件原理图可知，芯片的 PA19/PA20 引脚会连接到 USB 转串口芯片的 USB_TXD/USB_RXD 引脚，用于收发串口信号；所以需要将主芯片的 PA19/PA20 引脚配置为 UART1_TX 和 UART1_RX。
- 打开芯片引脚定义文件 pin_name.h，找到 PA19 和 PA20 引脚的功能复用，得知需要将两个引脚的功能分别定义为 PA19_UART1_TX 和 PA20_UART1_RX。

1.3.2 代码示例

代码路径：

```
boards/csky/cb5631/include/board_config.h
```

串口配置

```
/* console */
#define CONSOLE_TXD PA19
#define CONSOLE_RXD PA20
#define CONSOLE_TXD_FUNC PA19_UART1_TX
#define CONSOLE_RXD_FUNC PA20_UART1_RX
#define CONSOLE_IDX 1
```

Wi-Fi 配置

- WLAN_ENABLE_PIN: WiFi 芯片使能引脚
- WLAN_POWER_PIN : WiFi 芯片的供电开关引脚

```
/* wifi*/
#define WIFI_SDIO_CMD PA15
#define WIFI_SDIO_CMD_FUNC PA15_SDIO_CMD
#define WIFI_SDIO_CLK PA16
#define WIFI_SDIO_CLK_FUNC PA16_SDIO_CLK
#define WIFI_SDIO_DAT0 PA17
#define WIFI_SDIO_DAT0_FUNC PA17_SDIO_DAT0
#define WIFI_SDIO_DAT1 PA18
#define WIFI_SDIO_DAT1_FUNC PA18_SDIO_DAT1
```

```
#define WIFI_SDIO_DAT2 PA13
#define WIFI_SDIO_DAT2_FUNC PA13_SDIO_DAT2
#define WIFI_SDIO_DAT3 PA14
#define WIFI_SDIO_DAT3_FUNC PA14_SDIO_DAT3
#define WLAN_ENABLE_PIN PB2
#define WLAN_POWER_PIN 0xFFFFFFFF
```

BT 配置

- BT_DIS_PIN: BT 芯片使能引脚

```
/* BT */
#define BT_UART_IDX 0
#define BT_UART_TXD PA2
#define BT_UART_RXD PA0
#define BT_UART_TXD_FUNC PA2_UART0_TX
#define BT_UART_RXD_FUNC PA0_UART0_RX
#define BT_DIS_PIN PB4
```

按键配置

- 定义了四个 GPIO 按键对应的引脚

```
/* button */
#define APP_KEY_MUTE PA29
#define APP_KEY_VOL_INC PB20
#define APP_KEY_VOL_DEC PA24
#define APP_KEY_STANDBY PA25
```

功放配置

- 定义了功放（PA）静音的控制引脚

```
/* PA */
#define PANGU_PA_MUTE PA23
```

LED 配置

- 定义两个 LED 灯对应的引脚，以及灯的特性：低电平亮

```
/* LED */
#define LED0_PIN PB21
#define LED1_PIN PB31
```

```
#define LED_FLIP_FLAG 1 /* 低电平亮 */
```

QSPI Flash 配置

- 定义 QSPI Flash 的 ID

```
/* QSPI FLash */  
#define EXAMPLE_QSPI_IDX 0
```

RTC 配置

- 使能 RTC 时钟，并配置 RTC 时钟数值

```
/* RTC */  
#define CONFIG_RTC_EN 1  
#define CONFIG_RTC_CLK (24540) /* internal RC CLK: 24.540KHz */
```

GPIO 信息

- 开发板对通用 GPIO 的说明，方便测试验证，可选配

```
/* 可用 GPIO 列表 */  
#define USER_GPIO_LIST_STR \  
    "GPIO ID   Name\n \  
21      PA21 (IIC_SCLK)\n \  
22      PA22 (IIC_SD0)\n \  
23      PA23 (-PA MUTE)\n \  
24      PA24 (-KEY K5)\n \  
25      PA25 (-KEY K6)\n \  
29      PA29 (-KEY K4)\n \  
32      PB0  (UART3_TX)\n \  
33      PB1  (UART3_RX)\n \  
52      PB20 (-KEY K3)\n \  
53      PB21 (-LED0)\n \  
54      PB22 (PWM1_O5/PWM_LED_DEMO)\n \  
55      PB23 (PWM1_O7/LCD_RESET)\n \  
56      PB24 (PWM1_O9/UART2_TX)\n \  
57      PB25 (PWM1_O11/UART2_RX)\n \  
58      PB26 (PWM1_IO0/LCD_RS)\n \  
59      PB27 (PWM1_IO2/LCD_CS)\n \  
60      PB28 (PWM1_IO4/SPI_SCK/LCD_SCL)\n \  
61      PB29 (PWM1_IO6/SPI_MOSI/LCD_SDA)\n \  
62      PB30 (PWM1_IO8/SPI_MISO)\n \  

```



```
63      PB31 (-LED1)\n \n"
```

2. CB5654 适配指南

2.1 目录结构

SDK 中板级适配代码的目录结构如下：

```
boards/silan/cb5654
├── bootimgs - 引导固件
├── configs - 默认的系统分区及编译链接文件
├── dspalg_cxc - 士兰语音识别方案核间通讯接口
├── include - 包含引脚配置定义、OS 配置文件、LWIP 配置文件
├── audio - 音频初始化及 软 VAD 低功耗框架
├── base_init.c - 板级初始化
├── pinmux_init.c - 引脚复用配置
├── soc_lpm.c - 低功耗适配
├── board_lpm.c - 板级低功耗处理
└── package.yaml - 组件配置文件
```

2.2 初始化

2.2.1 引脚复用

设备引导时会先调用 board_init 函数，该函数中做芯片级的初始化和引脚的复用配置。

代码路径

```
boards/silan/cb5654/pinmux_init.c
```

代码示例

开发板中明确的引脚功能进行配置，例如 PA4、PA5 复用为 UART2，作为串口调试输出，PB0、PB1 复 用为 UART1 和 RTL8723 的蓝牙模块通讯。开发者需要根据实际的硬件连接进行复用的配置。

```
static void board_pinmux_config(void)
{
//console
drv_pinmux_config(PA4, PA4_UART2_TX);
drv_pinmux_config(PA5, PA5_UART2_RX);
// BT
```

```
drv_pinmux_config(PB0, PB0_UART1_TX);
drv_pinmux_config(PB1, PB1_UART1_RX);
//WiFi
drv_pinmux_config(PC2, PC2_SD_D0);
drv_pinmux_config(PC1, PC1_SD_D1);
drv_pinmux_config(PC6, PC6_SD_D2);
drv_pinmux_config(PC5, PC5_SD_D3);
drv_pinmux_config(PC3, PC3_SD_CLK);
drv_pinmux_config(PC4, PC4_SD_CMD_CMD);
drv_pinmux_config(PC0, PC0_SD_DET);
//SD card
drv_pinmux_config(PC9, PC9_SDIO_D0);
drv_pinmux_config(PC8, PC8_SDIO_D1);
drv_pinmux_config(PC13, PC13_SDIO_D2);
drv_pinmux_config(PC12, PC12_SDIO_D3);
drv_pinmux_config(PC10, PC10_SDIO_CLK);
drv_pinmux_config(PC11, PC11_SDIO_CMD);
drv_pinmux_config(PC7, PC7_SDIO_DET);
}
```

2.2.2 初始化接口

板级的初始化分别两个接口，board_base_init 为板级小系统的初始化；board_audio_init 提供音频功能的初始化。

代码路径

```
boards/silan/cb5654/base_init.c
```

代码示例

板级小系统的初始化，主要对可用的串口和 Flash 进行初始化，CB5654 开发板的三个串口都注册到串口驱动，若新的板子，串口被其他功能复用，根据情况删除注册。

注意：uart_csky_register 函数的参数是从零开始的 ID，实际的硬件串口号需减一

代码中宏 SOC_DSP_LDO_LEVEL 和宏 CONFIG_DMAC_DSP_ACQ 的功能，板级参数配置章节再说明

```
void board_base_init(void)
{
#ifdef SOC_DSP_LDO_LEVEL
extern void silan_dsp_ldo_config(int level);
extern void silan_soc_ldo_config(int level);
```

```

silan_dsp_ldo_config(SOC_DSP_LDO_LEVEL);
silan_soc_ldo_config(SOC_DSP_LDO_LEVEL);
#endif
uart_csky_register(0); /* UART1 */
uart_csky_register(1); /* UART2 */
uart_csky_register(2); /* UART3 */
spiflash_csky_register(0);
#ifdef CONFIG_DMACE_DSP_ACQ
sram_init();
#endif
}

```

音频功能的初始化，启动麦克风和参考音的采集，该函数一般无需修改，功能已经参数化，板级参数配置章节再说明

```

void board_audio_init()
{
#ifdef CONFIG_DMACE_DSP_ACQ
/* 参考音 增益，前端反馈，理论 (16)0dB 即可，但单端模补偿 6dB*/
voice_ref_init(24, 24); /* 数值单位 0.75dB 16 + 6/0.75 = 24 */
/* 麦克风 增益，boost (3)20dB 模拟增益(8)0dB，伪差分补偿 6dB，看信号还较小继续增加 12dB */
int mic_gain_val = 8 + (CONFIG_MIC_GAIN * 2 / 3);
voice_mic_init(3, mic_gain_val, mic_gain_val); /*数值单位 1.5dB 8 + 18/1.5 = 20*/
#endif
}

```

2.3 参数配置

2.3.1 硬件配置

代码路径

```
boards/silan/cb5654/include/board_config.h
```

代码示例

- SOC_DSP_LDO_LEVEL 宏定义芯片内部输出的 DSP 的供电电压
- CONSOLE_ID 定义调试串口输出的串口 ID，1 对应硬件的 UART2

```

/* 系统 */
//1:1.2V 2:1.0V 3:1.4V
// #define SOC_DSP_LDO_LEVEL 3

```

```
#define CONSOLE_ID 1
```

示例应用中使用一个 LED 灯，此处定义灯的引脚和参数

```
/* LED */
#define PIN_LED_R LED_PIN_NOT_SET
#define PIN_LED_G PD4
#define PIN_LED_B LED_PIN_NOT_SET
#define LED_FLIP_FLAG 1 /* 低电平亮 */
```

定义是否支持芯片内部 RTC，RTC 需要外部电路支持，若板子支持可开启改配置，示例应用会支持时间同步和闹铃功能

```
/* RTC */
#define CONFIG_RTC_EN 1
```

音频相关配置

- PIN_PA_EN 模拟功放对应的引脚号
- CONFIG_VOL_MAX 音量系统的最大值配置，下面参数已经配置为芯片的最佳参数，不建议修改
- CONFIG_LEFT_GAIN CONFIG_RIGHT_GAIN，左右声道 的音量配置，-1 表示该声道应用可调，该声道输出到扬声器，若固定一个值则表示该声道为参考音声道，该值不能大于 CONFIG_VOL_MAX
- CONFIG_MIC_GAIN 麦克风的增益配置 db 数，CONFIG_MIC_GAIN+20db 是总的增益数

```
/* 音频 */
#define PIN_PA_EN PD0
#define CONFIG_VOL_MAX (88)
#define CONFIG_LEFT_GAIN (-1) /* 左声道固定 -> PA */
#define CONFIG_RIGHT_GAIN (88) /* 右声道可调 -> REF */
#define CONFIG_MIC_GAIN (18) /* MIC 初始 20dB，该值在 20dB 基础上增加的 dB 数 */
```

WiFi 驱动配置

- WLAN_ENABLE_PIN WiFi 芯片使能引脚
- WLAN_POWER_PIN WiFi 芯片的供电开关引脚
- PIN_WL_WAKE_HOST WiFi 芯片的唤醒主控的引脚

```

/* WiFi */
#define WLAN_ENABLE_PIN    PC8
#define WLAN_POWER_PIN     PC0  /* 等于 0xffffffff 表示不支持 */

#define PIN_WL_WAKE_HOST PA6
#define PIN_WL_WAKE_HOST_GROUP LPM_DEV_MASK_GENERAL_GPIO2

```

ADC 按键配置

- PIN_ADC_KEY 按键的 ADC 引脚号
- KEY_ADC_VAL* ADC 各个按键对应的电压值
- KEY_AD_VAL_OFFSET 按键检测允许的误差范围

```

/* ADC 按键引脚 */
#define PIN_ADC_KEY    PA1

/* ADC 按键配置 */
#define KEY_ADC_VAL1 1751
#define KEY_ADC_VAL2 2311
#define KEY_ADC_VAL3 3051
#define KEY_ADC_VAL4 1376
#define KEY_ADC_VAL5 3587
#define KEY_AD_VAL_OFFSET 100  /* 按键值误差允许 */

#define VAD_ADC_VAL_MAX    KEY_ADC_VAL5 + KEY_AD_VAL_OFFSET /* 最大值 + 误差值 */
#define VAD_ADC_VAL_MIN    KEY_ADC_VAL4 - KEY_AD_VAL_OFFSET /* 最小值 - 误差值 */

```

GPIO 信息，开发板对通用 GPIO 的说明，方便测试验证，可选配

```

/* 可用 GPIO 列表 */
#define USER_GPIO_LIST_STR \
"ID  Name\n \
42  PD4(LED)\n \
46  PD0(PA MUTE)\n \
45  PD1\n \
38  PD7\n \
39  PD8\n \
29  PC12\n \
30  PC13\n \
2   PA2\n \
59  PB7\n \

```

```
60 PB6\n \
61 PB5\n \
62 PB4\n \
"
```

2.3.2 其他配置

配置项	说明
CONFIG_DMA C_DSP_ACQ	该配置需要在应用进行全局配置，用于切换算法 DSP 协议，示例默认使用 T-HEAD 算法协议无效定义该宏，切换到 SILAN 协议需要定义

2.4 低功耗

2.4.1 进入低功耗

应用低功耗流程需要进入某种低功耗状态时会调用该函数。

- 函数原型

```
void board_enter_lpm(pm_policy_t policy);
```

- 功能描述
控制设备进入指定的功耗模式
- 参数描述

IN/OUT	NAME	DESC
IN	policy	低功耗类型

pm_policy_t 枚举定义	
LPM_POLICY_NO_POWER_SAVE	正常运行状态
LPM_POLICY_LOW_POWER	进入软 VAD 低功耗状态
LPM_POLICY_DEEP_SLEEP	进入待机状态

- 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

2.4.2 退出低功耗

设备从低功耗状态唤醒前调用该函数，在该函数中适配外设的恢复流程。

- 函数原型

```
void board_leave_lpm(pm_policy_t policy);
```

- 功能描述

从低功耗状态唤醒进入正常运行状态。

- 参数描述

IN/OUT	NAME	DESC
IN	policy	低功耗类型

pm_policy_t 枚举定义	
LPM_POLICY_NO_POWER_SAVE	正常运行状态
LPM_POLICY_LOW_POWER	进入软 VAD 低功耗状态
LPM_POLICY_DEEP_SLEEP	进入待机状态

• 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

I 操作系统介绍及驱动开发指南

智能语音终端软件平台，采用了基于 AliOS Thing 为内核的 YoC 操作系统。关于 YoC 操作系统介绍、驱动开发指南以及核心模块说明，可以参考《[yocbook](#)》了解更多详细信息。

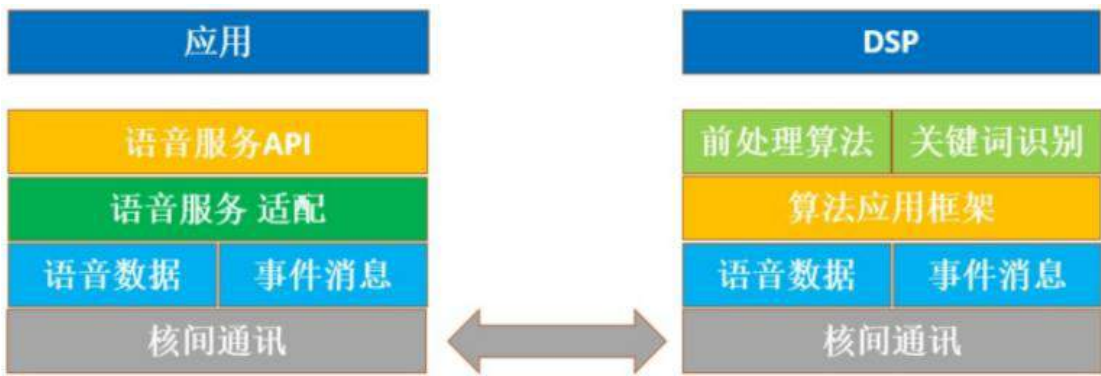
I 智能语音组件适配指南

1. 语音服务适配指南

1.1 概述

语音服务组件提供关键词识别和语音数据的处理控制。输入麦克风的语音数据经过回音消除降噪和关键词识别处理后再输出到应用层使用。YoC 在语音服务接口和算法实现之间增加了适配层，方便多种语音算法的接入，保持了应用代码的统一。

下图以 SC5654 芯片双核架构为例，适配层利用核间通讯，实现应用与 DSP 算法的数据交互。



1.2 适配接口

1.2.1 语音服务适配接口

语音服务适配接口注册在 mic_ops 结构体中，详细信息如下：

组件：mic

头文件：<yoc/mic.h>

语音服务的适配接口如下：

mic_ops 结构体中适配接口定义	
int (*init) (mic_t *mic, mic_event_t pcm_event);	初始化资源
int (*deinit) (mic_t *mic);	释放资源
int (*pcm_set_param) (mic_t *mic, void *param);	初始化语音算法参数
int (*kws_control) (mic_t *mic, int flag);	关键词识别使能控制
int (*kws_wake) (mic_t *mic, int flag);	发送命令让系统模拟一个唤醒事件
int (*pcm_data_control) (mic_t *mic, int flag);	开关控制 PCM 数据输出
int (*pcm_aec_control) (mic_t *mic, int flag);	开关控制算法回音消除

1.2.2 应用接口映射

应用通过调用应用层 API，操作对应语音服务适配接口，语音服务开发者只需实现适配接口定义。用户无需修改代码，即可迁移至对应语音服务上。

应用 API 与语音服务适配接口映射如下：

应用操作	应用 API	语音服务适配 API
启动语音服务	au_i_mit_start	init
关闭语音服务	au_i_mic_stop	deinit
初始化语音服务参数	au_i_mic_set_param	pcm_set_param
开关关键词唤醒	au_i_mic_set_wake_enable	kws_control
启停语音数据交互	au_i_mic_control(MIC_CTRL_START_PCM/MIC_CTRL_STOP_PCM)	pcm_data_control
强制关键词唤醒	au_i_mic_control(MIC_CTRL_START_SESSION)	kws_wake

1.3 接口说明

Init

• 函数原型

```
int (*init) (mic_t *mic, mic_event_t mic_event);
```

• 功能描述

语音服务初始化，在应用调用 aui_mic_start 时会调用该接口。若有私有数据需要保存，可通过函数 mic_set_privdata 将其保存在 mic->priv 私有成员指针中。设置后，其他适配函数就可以通过函数 mic_get_privdata 获取该指针。语音服务层通过应用注册的 mic_event 回调函数将语音事件传递给应用层。

• 参数描述

IN/OUT	NAME	DESC
IN	mic	语音服务对象
IN	mic_event	事件回调函数

• 相关定义

mic_event_t 结构体定义	
typedef void (*mic_event_t)(void *priv, mic_event_id_t evt_id, void *data, int size);	算法层产生的事件和数据都通过该回调传递到应用

mic_event_id_t 枚举定义	
MIC_EVENT_PCM_DATA,	音频数据
MIC_EVENT_SESSION_START,	开始对话
MIC_EVENT_SESSION_STOP,	停止对话

MIC_EVENT_VAD,	检测到声音
MIC_EVENT_VAD_DATA,	唤醒后命令词数据
MIC_EVENT_KWS_DATA	唤醒词数据

• 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

Deinit

• 函数原型

```
int (*deinit) (mic_t *mic);
```

• 功能描述

应用接口 `au_i_mic_stop` 执行时会调用该适配接口释放资源。

• 参数描述

IN/OUT	NAME	DESC
IN	mic	mic 对象

• 返回值

RETURN	DESC
0	成功

< 0	失败，错误码
-----	--------

kws_control

• 函数原型

```
int (*kws_control) (mic_t *mic, int flag);
```

- 功能描述
关键词识别使能控制。

• 参数描述

IN/OUT	NAME	DESC
IN	mic	mic 对象
IN	flag	0：关闭关键词识别算法功能；1：开启关键词识别算法功能

• 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

kws_wake

• 函数原型

```
int (*kws_wake) (mic_t *mic, int flag);
```

- 功能描述
强制算法发出一个模拟唤醒事件。

- 参数描述

IN/OUT	NAME	DESC
IN	mic	MIC 对象
IN	flag	传入唤醒词的 ID，唤醒事件的参数用此 ID 作为唤醒词 ID 传递给应用

- 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

pcm_data_control

- 函数原型

```
int (*pcm_data_control) (mic_t *mic, int flag);
```

- 功能描述

控制算法输出的音频流的开关。

- 参数描述

IN/OUT	NAME	DESC
IN	mic	mic 对象
IN	flag	0：停止数据传输；1：启动数据传输

- 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

pcm_aec_control

• 函数原型

```
int (*pcm_aec_control) (mic_t *mic, int flag);
```

• 功能描述

控制算法是否使能回音消除。

• 参数描述

IN/OUT	NAME	DESC
IN	mic	mic 对象
IN	flag	0: 关闭回音消除功能; 1: 开启回音消除功能

• 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

pcm_set_param

• 函数原型

```
int (*pcm_set_param) (mic_t *mic, void *param);
```


- 功能描述

初始化语音算法参数。

- 参数描述

IN/OUT	NAME	DESC
IN	mic	mic 对象
IN	param	MIC 的配置参数

mic_pcm_param 结构体定义	
int sample_bits;	采样精度 默认 16bit
int channels;	预留
int rate;	采样率 默认 16K
int nsmode;	去噪等级 0~3 非线性处理等级逐步加强，其他值无非线性处理
int acemode;	回音消除等级 0~3 非线性处理等级逐步加强，其他值无非线性处理
int vadmode;	VAD 等级 0~3 等级逐步加强
int sentence_time_ms;	有语音断句时间
int noack_time_ms;	无语音超时时间
int max_time_ms;	唤醒后总超时时间

• 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

mic_set_privdata

• 函数原型

```
int mic_set_privdata(void *priv);
```

• 功能描述

设置适配私有数据到 mic 对象。

• 参数描述

IN/OUT	NAME	DESC
IN	priv	适配的私有数据

• 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

mic_get_privdata

• 函数原型

```
void *mic_get_privdata(void);
```

- 功能描述

获取 mic 对象中的私有数据指针。

- 参数描述

无

- 返回值

RETURN	DESC
void*	私有数据指针

mic_ops_register

- 函数原型

```
int mic_ops_register(mic_ops_t *ops);
```

- 功能描述

注册适配层，一般增加一个适配需要实现一个新的函数封装该函数，提供一个简单的注册函数供应用使用。

- 参数描述

IN/OUT	NAME	DESC
IN	ops	适配层实现的

- 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

1.4 适配示例

适配完成所有接口函数后，定义适配接口结构体，完成函数注册。
示例如下：

```
/* 适配接口结构体 */
static mic_ops_t mic_adp_ops = {
    .init = mic_adaptor_init,
    .deinit = mic_adaptor_deinit,
    .kws_control = mic_adaptor_kws_control,
    .kws_wake = mic_adaptor_kws_wake,
    .pcm_data_control = mic_adaptor_pcm_data_control,
    .pcm_aec_control = mic_adaptor_pcm_aec_control,
    .pcm_set_param = mic_adaptor_set_param,
    .pcm_get_param = mic_adaptor_get_param,
};

/* 注册函数 */
void mic_thead_v1_register(void)
{
    mic_ops_register(&mic_adp_ops);
}
```

2. 云服务适配指南

2.1 概述

云服务组件提供应用与云端 ASR/NLP/TTS 服务交互的接口。调用对应服务 API 后，组件自动完成云端连接、鉴权、启动服务的过程，用户只需通过接口将需识别的音频或需合成的字符串传入，即可获得云端返回结果，设备端只需根据结果完成预定的应用行为。

为了减少用户的开发成本，YoC 定义了一套统一的适配接口，应用层可以用同样的代码在不同的云服务之间无缝切换。

2.2 适配接口

已适配的组件：aui_aliyunNls、aui_cloud

头文件：<yoc/aui_cloud.h>

云服务组件的主要 API 如下：

API	说明
au_i_cloud_init	云服务初始化
au_i_cloud_start_pcm	启动语音数据交互
au_i_cloud_push_pcm	推送语音数据
au_i_cloud_stop_pcm	结束语音数据推送
au_i_cloud_push_text	文本内容推送到云端进行 NLP 处理
au_i_cloud_start_tts	启动 TTS 语音合成服务
au_i_cloud_req_tts	向云端发送文本信息，请求 TTS 音频数据
au_i_cloud_stop_tts	停止 TTS 语音合成服务

2.3 接口说明

au_i_cloud_init

- 函数原型

```
int au_i_cloud_init(au_i_t *au_i);
```

- 功能描述

该函数用于初始化云服务。参数 au_i_t 结构包含的 config 成员用来指定语音合成的参数，包括发言人、音量、语速。适配云端请务必参考取值范围并进行转换，保证多平台切换是参数不做调整也能达到预期的效果，还有 nlp_cb 回调函数，供用户处理云端返回的信息，ASR 结果和 NLP 结果都使用同一个回调。

- 参数描述

IN/OUT	NAME	DESC
IN	au_i	云服务对象

aui_t 结构体定义	
aui_config_t config;	云端配置参数
void *context;	适配用户可以用来保存私有数据指针，其他函数可以通过 aui 参数获取获取 私有数据
aos_sem_t sem;	信号量，适配用户可选用
aui_config_t 结构体定义	
char *per;	发言人名称
int vol;	音量，取值 0-100，50 标准音量
int spd;	语速，-500~500，默认 0 标准语速
void (*nlp_cb)(const char *json_text);	云端返回文本数据的回调函数

• 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

aui_cloud_start_pcm

• 函数原型

```
int aui_cloud_start_pcm(aui_t *aui);
```

• 功能描述

启动语音数据交互，准备上传语音数据。

- 参数描述

IN/OUT	NAME	DESC
IN	aiui	云服务对象

- 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

aiui_cloud_push_pcm

- 函数原型

```
int aiui_cloud_push_pcm(aiui_t *aiui, void *data, size_t size);
```

- 功能描述

推送语音数据到云端进行识别。

- 参数描述

IN/OUT	NAME	DESC
IN	aiui	云服务对象
IN	data	需要上传的语音数据
IN	size	语音数据的字节数

aiui_cloud_stop_pcm

- 函数原型

```
int aiui_cloud_stop_pcm(aiui_t *aiui);
```

- 功能描述

结束语音数据推送，云端返回的结果通过调用 nlp_cb 回调函数通知应用进行处理。

- 参数描述

IN/OUT	NAME	DESC
IN	aui	云服务对象

- 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

aui_cloud_push_text

- 函数原型

```
int aui_cloud_push_text(aui_t *aui, char *text);
```

- 功能描述

文本内容推送到云端进行 NLP 处理。

- 参数描述

IN/OUT	NAME	DESC
IN	aui	云服务对象
IN	text	需要上传到云端的文本数据

- 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

aiui_cloud_start_tts

- 函数原型

```
int aiui_cloud_start_tts(aiui_t *aiui);
```

- 功能描述

启动 TTS 语音合成服务

- 参数描述

IN/OUT	NAME	DESC
IN	aiui	云服务对象

- 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

aiui_cloud_req_tts

- 函数原型

```
int aiui_cloud_req_tts(aiui_t *aiui, const char *player_fifo_name, const char *text, aiui_tts_cb stat_cb);
```

- 功能描述

向云端发送文本信息，请求 TTS 音频数据。要求异步实现，云端的语音数据可以直接写入

播放器的 nsfifo，然后调用 aui_player_play 来播放语音数据。nsfifo 的使用方法可参见组件 components_aliyunnls_mit_tts.c 中的实现。

• 参数描述

IN/OUT	NAME	DESC
IN	aui	云服务对象
IN	player_fifo_name	播放器 FIFO 格式的 url，必须以“fifo://”开头的字符串，例如 fifo://tts1
IN	text	请求 TTS 转换的文本内容
IN	stat_cb	TTS 请求时的状态回调函数

aui_tts_state_e 枚举定义	
AUI_TTS_INIT	正在初始化
AUI_TTS_CONTINUE	数据转化中
AUI_TTS_FINISH	转化完成
AUI_TTS_ERROR	转化错误

• 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

aui_cloud_stop_tts

• 函数原型

```
int aui_cloud_stop_tts(aui_t *aui);
```

• 功能描述

停止 TTS 语音合成服务

• 参数描述

IN/OUT	NAME	DESC
IN	aui	云服务对象

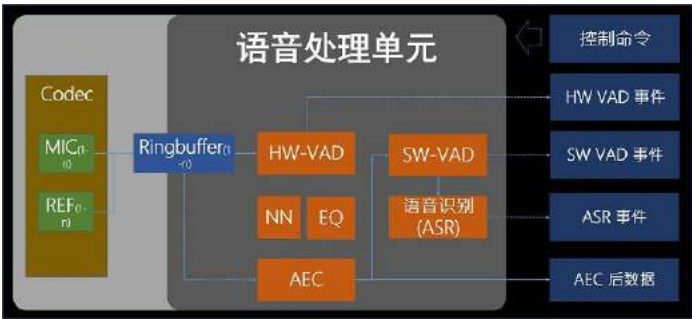
• 返回值

RETURN	DESC
0	成功
< 0	失败，错误码

3. 语音算法适配指南

智能语音 SDK 将算法实现与接口分离，设计出了一套层次化得调用框架，用户在此框架上可以方便得将自研算法移植进 DSP 中，利用 SDK 中原有得数据采集、播放、上云通道，以实现设备端算法得快速落地。

3.1 框架图



- AP（Application Processor）主要负责应用开发，CP（coprocessor）用来通过处理主 cpu 的一些工作负荷来使操作提速的辅助处理器，如语音算法。
- IPC（Inter-Process Communication，异构多核通信）
- voice data：数据内容格式为麦克风 m（m 路数据）和参考声 n（n 路数据）

3.2 特性

- 适用于多核 SoC，AP 核负责采集及搬送数据，CP 核负责离线语音识别及抛出各类事件
- 提供多种录音数据，方便对接云端语音处理及本地算法问题调试
- 提供 LPM 接口，方便低功耗管理
- 接入方式简单，最小只需实现数据采集及语音识别部分算法等接口

3.3 接口定义

本地算法初始化

```
voice_t *voice_ai_init(void *priv, voice_cts_ops_t *ops);
```

- 参数：
 - priv: 用户私有数据
 - ops: ai 算法实现
- 返回值：
 - 0: 成功 非 0: 失败

```
typedef struct __voice_cts_ops {  
    int (*init)(void *priv);  
    int (*deinit)(void *priv);  
    int (*aec)(void *priv, void *mic, void *ref, int ms, void *out); //aec 算法实现  
    int (*vad)(void *priv, void *mic, void *ref, int ms, void *out); //vad 算法实现  
    int (*kws)(void *priv, void *mic, void *ref, int ms, void *out); //kws 算法实现  
    int (*asr)(void *priv, void *vad_data, int ms); //asr 算法实现  
} voice_cts_ops_t;
```

初始化及去初始化

```
voice_t *voice_init(voice_evt_t cb, void *priv);  
void voice_deinit(voice_t *v);
```

- 参数:
 - cb: voice 事件
 - priv: 用户私有数
 - v: voice 句柄
- 返回值:
 - 0: 成功 非 0: 失败

```
typedef void (*voice_evt_t)(void *priv, voice_evt_id_t evt_id, void *data, int len);
typedef enum {
    VOICE_ASR_EVT,//asr 事件
    VOICE_SILENCE_EVT,//断句事件
    VOICE_DATA_EVT//回流数据到达事件
} voice_evt_id_t;
```

参数配置

```
int voice_config(voice_t *v, voice_param_t *p);
```

- 参数:
 - v: voice 句柄
 - p: pcm 参数
- 返回值:
 - 0: 成功 非 0: 失败

```
typedef struct {
    int cts_ms;//ai 算法每次数据大小，单位（ms）
    int ipc_mode;//与 ai 算法侧的通信方式，1: ipc
} voice_param_t;
```

拾音参数配置

```
int voice_add_mic(voice_t *v, voice_pcm_param_t *p);
int voice_add_ref(voice_t *v, voice_pcm_param_t *p);
```

初始化 ai 算法模块

- 参数:
 - v: voice 句柄
 - p: pcm 参数

- 返回值:
 - 0: 成功 非 0: 失败

```
typedef struct {  
    char        *pcm_name; //pcm 设备名  
    unsigned int  rate; //采样率  
    int          sample_bits; //采样位数  
    int          access; //是否为交错模式, 0: 非交错 1: 交错  
    int          channles; //通道总数  
    int          channles_sum; //通道总数  
    int          period_bytes; //pcm 周期数据量 ( 用户不必配置 )  
} voice_pcm_param_t;
```

启动（停止）本地算法

```
int voice_start(voice_t *v);  
int voice_stop(voice_t *v);
```

- 参数:
 - v: voice 句柄
- 返回值:
 - 0: 成功 非 0: 失败

数据回流控制

```
int voice_backflow_control(voice_t *v, voice_backflow_id_t id, int flag);
```

- 参数:
 - v: voice 句柄
 - id: 数据类型
 - flag: 0: 关闭回流, 1: 打开回流
- 返回值:
 - 0: 成功 非 0: 失败

```
typedef enum {  
    VOICE_MIC_DATA, //mic 数据  
    VOICE_REF_DATA, //ref 数据  
    VOICE_VAD_DATA, //vad 后数据  
    VOICE_AEC_DATA, //aec 后数据  
    VOCIE_BACKFLOW_DATA  
} voice_backflow_id_t;
```

I 基本调试指南

1. 使用串口调试

1.1 用内置串口命令调试

yoc 支持很多的串口命令，我们可以通过串口命令进行很多的调试操作

help

```
> help
help          : show commands
ping          : ping command.
ifconfig      : network config
date          : date command.
ps            : show tasks
free          : show memory info
sys           : sys comand
log           : log contrtol
iperf         : network performance test
kv            : kv tools
```

输入 help 命令，可以查看当前所有支持命令：

指令名称	指令简介
help	显示当前 YoC 系统所有支持的命令行命令
ps	查看当前系统的进程状态
free	查看系统实际使用内存情况
sys	系统命令及系统信息
date	获取系统当前的时间w设置系统当前时间
log	设置系统日志打印等级
kv	设置或读取 kv 分区的存储信息
ifconfig	获取系统当前网络状态

```

>ps
CPU USAGE: 640/10000
task                pri status      sp      stack size max used ratio left tick
%CPU
-----
dyn_me              60 pend    0x181d12fc    608    372  61%    0
  m
  0.0              61 ready  0x181d16a4   1376    216  15%    0
idle_task
98.1              9 pend    0x181d0398   2048    352  17%    0
DEFAULT-WORKQUEUE
0.0              5 pend    0x181d1c08   1152    512  44%    96
timer_task
0.0              60 sleep   0x181d0d84    552    340  61%    500
cpu_stats
0.0              32 pend    0x182a5144   2048    844  41%   1000
event_svr
0.0              32 pend    0x182a5a4c   1024    568  55%  60000
select
0.0              32 ready  0x182a608c   3072    852  27%   1000
cli
2.4              32 pend    0x182a6fd4   2048    400  19%  10000
uart_tas
k 0.0            32 pend    0x182a807c   4096    352   8%    0
lp
m
0.0              36 ready  0x182dcbd8   2048   1376  67%    97
tcpip_thread
0.0              32 pend    0x182dd690   4096    636  15%   1000
netmgr
0.0              36 pend    0x183a5d94   4096    424  10%    0
b-pres
s
0.0              45 sleep   0x1829db78   1024    340  33%  60000
fota-check
0.0              45 pend    0x1829e048   2048    868  42%  10000
fota
0.0

```

ps 命令可以打印出当前系统所有的线程状态，其中各项含义如下：

名称	含义
task	线程名称
pri	线程优先级 数值越大等级越低，最低优先级的线程是 idle 线程，优先级为 61
status	当前线程状态，下面详细描述
sp	栈指针地址
stack size	线程分配的栈大小
max used	线程使用到的栈空间峰值
ratio	线程使用到的栈空间峰值与栈大小的比值，这个值可以指导我们，当前栈空间的分配 是偏大还是偏小
left tick	被切出线程多久后会被再次调用到
%CPU	该线程占到的 cpu 负载，例如上面示例中，idle 线程占了 98.1% 说明 cpu 很空闲

其中有些信息，我们详细说明一下：

- 线程状态有 ready、pend、suspend、sleep、deleted
 - ready：表示当前线程已经等待被调度，系统的调度原则是：优先级不同 高优先级线程运行，优先级相同则各个线程时间片轮转运行。
 - pend：表示当前线程被挂起，挂起原因是线程在等待信号量、互斥锁、消息队列等，例如调用：aos_sem_wait，aos_mutex_lock 等接口，线程就会被挂起并置成 pend 状态。如果是信号量等待时间是 forever，则 left tick 的值为 0；如果有超时时间，则 left tick 的值就是超时 时间，单位为毫秒
 - suspend：表示当前线程被主动挂起，就是程序主动调用了 task_suspend 函数
 - sleep：表示当前线程被主动挂起，就是调用了 aos_sleep 等睡眠函数，left tick 的值即表示 睡眠的时间
 - deleted：当前线程已经被主动删除，也就是调用 krhino_task_del

- %CPU 状态只有在 k_config.h 文件中 RHINO_CONFIG_HW_COUNT 和 RHINO_CONFIG_TASK_SCHED_STATS 宏被设置 1 的时候才会出现。
- 第一行 CPU USAGE: 640/10000 表示，当前系统的整体负载，如上示例，系统的 CPU 占有率是 0.64%

free

```
> free
total used free peak
memory usage: 5652536 605316 5047220 1093576
```

free 命令可以使用输出当前系统的堆状态，其中：

- total 为 总的堆的大小
- used 为 系统使用的 堆大小
- free 为 系统空余的 堆大小
- peak 为 系统使用的 堆最大空间

单位为 byte

```
>free mem

----- all memory blocks
-----
g_kmm_head = 1829bfc8
ALL BLOCKS
address, stat  size    dye    caller  pre-stat  point
0x1829cb20  used      8  fefefefe  0x0      pre-used;
0x1829cb38  used    4128  fefefefe  0xbfffffff pre-used;
0x1829db68  used    1216  fefefefe  0x180190b6 pre-used;
0x1829e038  used    2240  fefefefe  0x180190b6 pre-used;
0x1829e908  used    4288  fefefefe  0x180190b6 pre-used;
0x1829f9d8  free     592  abababab  0x180aaa6d pre-used; free[ 0x0,
0x0]
0x1829fc38  used     40  fefefefe  0x180cb836 pre-free [0x1829f9d8];
0x1829fc70  used     40  fefefefe  0x180cb836 pre-used;
0x1829fca8  used   18436  fefefefe  0x1810448d pre-used;
0x182a44bc  used     40  fefefefe  0x180cb836 pre-used;
...
0x183a5ce0  used     16  fefefefe  0x1801d477 pre-used;
0x183a5d00  used     40  fefefefe  0x1801d477 pre-used;
0x183a5d38  used     12  fefefefe  0x1801a911 pre-used;
0x183a5d54  used     32  fefefefe  0x18010d40 pre-used;
0x183a5d84  used    4288  fefefefe  0x180190b6 pre-used;
```

```

0x183a6e54 free 4559244 abababab 0x18027fd9 pre-used; free[ 0x0,
0x0]
0x187ffff0 used sentinel fefefefe 0x0 pre-free [0x183a6e54];

----- all free memory blocks
-----
address, stat size dye caller pre-stat point
FL bitmap: 0x10f4b
SL bitmap 0x84
-> [0][2]
0x18349b88 free 8 abababab 0x1802a1b1 pre-used; free[ 0x0,
0x0]
-> [0][7]
0x182df2f8 free 28 abababab 0x0 pre-used; free[ 0x0,
0x0]
-> [0][25]

0x182df3c8 free 100 abababab 0x18010ea5 pre-used; free[ 0x0,
0x0]
...
0x182b5704 free 160204 abababab 0x1804fe55 pre-used; free[ 0x0,
0x0]
SL bitmap 0x4
-> [16][2]
0x183a6e54 free 4559244 abababab 0x18027fd9 pre-used; free[ 0x0,
0x0]

----- memory allocation statistic
-----
free | used | maxused
5047040 | 605496 | 1093576

----- alloc size statistic:-----
[2^02] bytes: 0 [[2^03] bytes: 1350 [[2^04] bytes: 398770 [[2^05] bytes:
29121 |
[2^06] bytes: 408344 [[2^07] bytes: 396962 [[2^08] bytes: 350 [[2^09]
bytes: 231 |
[2^10] bytes: 55 [[2^11] bytes: 38 [[2^12] bytes: 396677 [[2^13] bytes:
1410 |
[2^14] bytes: 14 [[2^15] bytes: 16 [[2^16] bytes: 0 [[2^17] bytes:
4 |
[2^18] bytes: 17 [[2^19] bytes: 0 [[2^20] bytes: 0 [[2^21] bytes:
0 |

```

```
[2^22] bytes:    0  |[2^23] bytes:    0  |[2^24] bytes:    0  |[2^25] bytes:
0  |
[2^26] bytes:    0  |[2^27] bytes:    0  |
```

`free mem` 命令可以打印出堆内各个 节点 的细节信息
整个打印信息被分成 4 个部分

- 第一部分为 系统所有 堆节点，包含了 节点的地址、大小、占用状态、调用 `malloc` 的程序地址等
- 第二部分为 当前系统 空置的 堆节点，信息与第一部分相同，只是单独列出了 `free` 的节点，可以观察系统的内存碎片情况
- 第三部分为 系统的总体堆内存使用情况，和 `free` 命令打印出的信息相同
- 第四部分为 堆节点的大小统计，与 2 的次方为单位进行划分

```
>free list
total used free peak
memory usage: 5652536 605316 5047220 1093576
0: caller=0xbffffffe, count= 1, total size=4128
1: caller=0x180190b6, count=25, total size=85696
2: caller=0x180aaa6c, count= 1, total size=592
3: caller=0x180cb836, count= 3, total size=120
4: caller=0x1810448c, count= 1, total size=18436
5: caller=0x18010a68, count=39, total size=1716
6: caller=0x18014548, count= 8, total size=580
7: caller=0x18054dda, count= 1, total size=1028
...
52: caller=0x18010d40, count= 2, total size=64
53: caller=0x1801d5b8, count= 3, total size=72
54: caller=0x1801d476, count= 6, total size=196
55: caller=0x1801d5ac, count= 3, total size=48092
56: caller=0x1801a910, count= 1, total size=12
57: caller=0x18027fd8, count= 1, total size=4559244
```

`free list` 是另一种形式的堆内存使用统计，统计了程序内各个 `malloc` 的调用并且还没有 `free` 的次数。

这个统计信息对于查找内存泄露非常有帮助。多次输出该命令，若 `count` 的值出现了增长，则可能有内存泄露的情况出现。

以上命令的 `caller` 信息，我们可以通过 在 `yoc.asm` 反汇编文件查找函数来确认具体的调用函数。

需要注意的是：free mem 和 free mem 只有在开启 CONFIG_DEBUG_MM 和 CONFIG_DEBUG 时才会出现，因为它需要占用一些内存空间用于存放这些调试信息

Sys

指令格式	指令简介
<code>sys kernel</code>	显示内核版本号
<code>sys os</code>	显示系统版本号
<code>sys sdk</code>	显示 sdk 版本号
<code>sys app</code>	显示 App 版本号
<code>sys id</code>	显示设备 id 号
<code>sys reboot</code>	重启

具体显示的信息如下：

其中 `sys app`、`sys id` 两个命令是在需要 fota 的时候才会使用到，一般是 occ 颁发出来的信息，不可更改，如果没有走过 fota 流程一般为空。其余的版本号信息，是代码宏定义的，可以修改。

date

`date` 命令是用于查询和设置当前系统时间，一般系统连上网络以后会定期调用 ntp，来和服务器同步时间，这个命令可以查询同步时间和设置系统时间

```
> date
TZ(08):Tue Aug 11 18:03:14 2020 1597168994
UTC:Tue Aug 11 10:03:14 2020 1597140194
date -s "2001-01-01 12:13:14"
> date -s "2020-08-11 18:15:38"
set date to: 2020-08-11 18:15:38
TZ(08):Wed Aug 12 02:15:38 2020 1597198538
UTC:Tue Aug 11 18:15:38 2020 1597169738
date -s "2001-01-01 12:13:14"
```

log

log 命令可以用于控制打印等级和打印的模块

```
> sys kernel
AOS-R-1.2.0
> sys os
YoC_V7.3
> sys sdk
Voice_SmartSpeaker_PanguRTL8723_V1.0.1
> sys app
1.0.0-20200314.1307-R-test
> sys id
e6d0d5480440000008376eb1e834a765
> date
TZ(08):Tue Aug 11 18:03:14 2020 1597168994
UTC:Tue Aug 11 10:03:14 2020 1597140194
date -s "2001-01-01 12:13:14"
> date -s "2020-08-11 18:15:38"
set date to: 2020-08-11 18:15:38
TZ(08):Wed Aug 12 02:15:38 2020 1597198538
UTC:Tue Aug 11 18:15:38 2020 1597169738
date -s "2001-01-01 12:13:14"
> log
Usage:
set level: log level 0~5
0:disable 1:F 2:E 3:W 4:I 5:D
add ignore tag: log ignore tag
clear ignore tag: log ignore
> log level 0
> log ignore fotalog tag ignore list:
fota
> log ignore RTC
log tag ignore list:
fota
RTC
>
```

log level num 用于控制打印等级

- 0: 关闭日志打印;
- 1: 打印 F 级别的日志;
- 2: 打印 E 级别及以上的日志;
- 3: 打印 W 级别及以上的日志;

- 4: 打印 I 级别及以上的日志;
- 5: 打印 D 级别及以上的日志, 也就是日志全开

`log ignore tag` 用于控制各个模块的打印

例如 `log ignore RTC` 表示关闭 RTC 模块的日志打印

需要注意的是: `log` 命令只能控制通过 LOG 模块打印出来的日志, 直接通过 `printf` 接口打印的日志不能被拦截。所以推荐用 LOG 模块去打印日志。

kv

kv 是一个小型的存储系统, 通过 key-value 的方式存储在 flash 中

```
> kv
Usage:
kv set key value
kv get key
kv setint key value
kv getint key
kv del key
>
```

`kv set key value` 是设置字符串类型的 value

`kv setint key value` 是设置整形的 value

例如:

```
kv set wifi_ssid my_ssid
kv set wifi_psk my_psk
```

如上两条命令是用于设置 wifi 的 ssid 和 psk, 重启后系统会去通过 kv 接口获取 flash 的 kv value 值, 从而进行联网。

Ifconfig

```
> ifconfig
wifi0 Link encap:WiFi HWaddr 18:bc:5a:60:d6:04
inet addr:192.168.43.167
GWaddr:192.168.43.1
Mask:255.255.255.0
DNS SERVER 0: 192.168.43.1
WiFi Connected to b0:e2:35:c0:c0:ac (on wifi0)
SSID: yocdemo
```

```
channel: 11
signal: -58 dBm
```

ifconfig 命令可以查看当前 网络连接的状态，其中：

- 第一部分是 本机的网络状态，包括本机 mac 地址，本机 IP，网关地址、掩码、DNS Server 地址
- 第二部分是 连接的路由器信息，包括 wifi 的名称，mac 地址，连接的信道、信号质量

1.2 创建自己的串口命令

上一节讲的都是系统内置的一些串口命令，加入自己想做创建一个串口命令用于调试，该怎么操作呢？yoc 中，串口命令代码模块为 cli，其代码头文件为 cli.h。我们需要包含这个头文件。

```
/*
 * Copyright (C) 2019-2020 Alibaba Group Holding Limited
 */
#include <string.h>
#include <aos/cli.h>
#define HELP_INFO \
"Usage:\n\tmycmd test\n"
static void cmd_mycmd_ctrl_func(char *wbuf, int wbuf_len, int argc, char **argv)
{
    int i;
    for (i = 0; i < argc; i++) {
        printf("argv %d: %s\n", i, argv[i]);
    }
    printf(HELP_INFO);
}
void cli_reg_cmd_my_cmd(void)
{
    static const struct cli_command cmd_info = {
        "my_cmd",
        "my_cmd test",
        cmd_mycmd_ctrl_func,
    };
    aos_cli_register_command(&cmd_info);
}
```

如上代码。

- 我们需要定义一个被 cli 回调的函数, 当串口输入这个命令时就会触发这个回调, `cmd_mycmd_ctrl_func`
- 我们需要定义一个命令字符串, 用于 cli 比较用于输入字符串来触发回调, `my_cmd`
- 我们需要定义一个 help 信息, 用于 串口输入 help 命令时打印出来, `my_cmd test`
- 当然最后我们在系统初始化时把这个命令注册到 cli 里面去, `cli_reg_cmd_my_cmd`

这样我们就可以拥有自己的串口调试命令了, 效果如下:

```
> my_cmd first cmd test
argv 0: my_cmd
argv 1: first
argv 2: cmd
argv 3: test
Usage:
mycmd test
```

2. 使用 GDB 调试

GDB 是 C/C++ 程序员的程序调试利器, 很多问题使用 GDB 来调试都可以大大提高效率。GDB 调试在查看变量、跟踪函数跳转流程、查看内存内容、查看线程栈等方面都非常方便。是深入理解程序运行细节最有效的方式之一。

同时, GDB 对于学习了解 C 语言代码、全局变量、栈、堆等内存区域的分布也有一定的帮助。

下面我们来介绍 CK GDB 在嵌入式芯片上的调试方法。

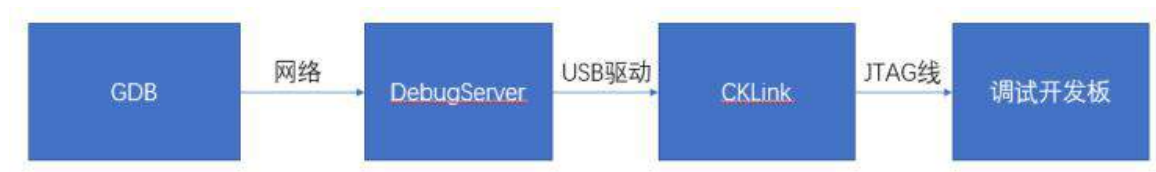
2.1 建立 GDB 连接

这一小节讲解一些嵌入式 GDB 调试使用的基础知识, 和在 PC 上直接使用 GDB 调试 PC 上的程序会有一些区别。

CK GDB 是运行在 PC 上的 GDB 程序, 通过仿真器和 JTAG 协议与开发板相连接, 可以调试基于 CK CPU 的芯片。其中 DebugServer 为作为连接 GDB 和 CKLink 仿真器的桥梁和翻译官, 一端通过网络与 GDB 连接, 另一端通过 USB 线与仿真器连接。

由于 GDB 与 DebugServer 通过网络通讯, 他们可运行在不同的 PC 上, 也可以运行在同一个 PC 上。

仿真器 CKLink 与开发板通过 20PIN 的 JTAG 排线连接。



CKLink

CKLink 实物如下图所示。可以通过[淘宝购买](#)。其使用方法可以查看：[CKLink 设备使用指南](#)。

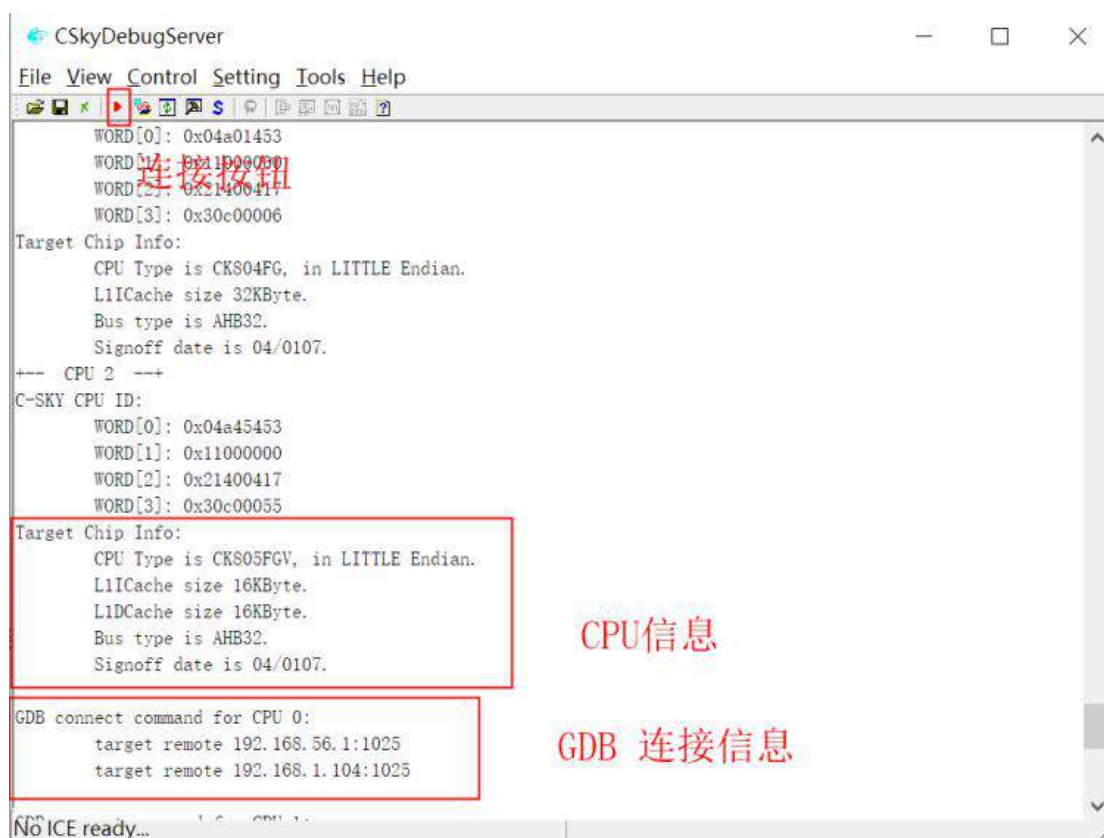
CKLink Lite及配件



DebugServer

DebugServer 有 windows 版本和 Linux 版本，下载和安装过程请参考：[Windows 调试环境安装](#)，[Linux 调试环境安装](#)。

以 windows 版本的 DebugServer 为例，安装完成以后，打开程序有如下界面：



点击连接按钮，如果连接成功会有 CPU 和 GDB 的信息打印，告知当前连接的 CPU 信息和开启的 GDB 服务信息。具体使用可以参考 OCC [资源下载](#) 页面下的文档《DebugServer User Guide_v5.10》。

2.2 启动 GDB 及配置

GDB 工具包含在整体的编译调试工具链里面，也可以通过 [OCC 下载](#)。GDB 的使用都需要通过命令行完成，通过在终端敲入命令来完成交互启动 GDB 通过如下命令进行：

```
csky-abiv2-elf-gdb xxx.elf
```

其中 xxx.elf 为当前板上运行的程序，它包含了所有的程序调试信息，如果缺少 elf 文件则无法进行调试。

启动 GDB 后输入如下命令连接 DebugServer。这条命令在 DebugServer 的界面会有打印，可以直接复制。

```
target remote [ip]:[port]
```

需要注意的是：运行 GDB 程序对应的 PC 需要能够通过网络访问 DebugServer 开启的对应的 IP 连上以后就可以通过 GDB 访问调试开发板上的芯片了。

.gdbinit 文件

.gdbinit 文件为 GDB 启动时默认运行的脚本文件，我们可以在.gdbinit 文件里面添加启动默认需要执行的命令，例如： `target remote [ip]:[port]`，那么在启动 GDB 的时候，会直接连接 DebugServer，提高调试效率。

2.3 常用 GDB 命令

这一小节介绍一些常用的 GDB 命令及使用方法。

加载程序

- 命令全名： `load`
- 简化： `lo`
- 说明：将 elf 文件 加载到 芯片中，这个命令对代码在 flash 运行的芯片无效。

举例：

```
(cskygdb) lo
Loading section .text, size 0x291a00 lma 0x18600000
section progress: 100.0%, total progress: 69.01%
Loading section .ram.code, size 0x228 lma 0x18891a00
section progress: 100.0%, total progress: 69.02%
Loading section .gcc_except_table, size 0x8f8 lma 0x18891c28
section progress: 100.0%, total progress: 69.08%
Loading section .rodata, size 0xeeac4 lma 0x18892520
section progress: 100.0%, total progress: 94.12%
Loading section .FSymTab, size 0x9c lma 0x18980fe4
section progress: 100.0%, total progress: 94.13%
Loading section .data, size 0x2e3c4 lma 0x18981400
section progress: 100.0%, total progress: 98.98%
Loading section ._itcm_code, size 0x9b70 lma 0x189af7c4
section progress: 100.0%, total progress: 100.00%
Start address 0x18600014, load size 3903412
Transfer rate: 238 KB/sec, 4003 bytes/write.
```

继续执行

- 命令全名： `c`ontinue`
- 简化： `c`
- 说明：继续执行被调试程序，直至下一个断点或程序结束。

举例：

```
(cskygdb)c
```

当点击 DebugServer 连接开发板并连上以后，程序会自动停止运行。等 GDB 挂进去以后，用 `c` 就可以继续运行程序。

当程序在运行的时候，GDB 直接挂入也会使程序停止运行，同样用 `c` 命令可以继续运行程序。

同样，当 `load` 完成后，也可以使用 `c` 运行程序

暂停运行

使用组件按键 `ctrl + c` 可以停止正在运行的程序。

停止运行程序就可以 进行各种命令操作，打印变量，打断点，查看栈信息，查看内存等。

当操作完成以后，使用 `c` 继续运行，或者使用 `n/s` 单步执行调试。

打印变量

- 命令全名： `print`
- 简化： `p`

打印变量可以打印各种形式

- 变量
- 变量地址
- 变量内容
- 函数
- 计算公式

举例：

```
(cskygdb)p g_tick_count
(cskygdb)p &g_tick_count
(cskygdb)p *g_tick_count
(cskygdb)p main
(cskygdb)p 3 * 5
```

可以指定打印格式

按照特定格式打印变量

- x 按十六进制格式显示变量。
- d 按十进制格式显示变量。
- o 按八进制格式显示变量。
- t 按二进制格式显示变量。
- c 按字符格式显示变量。

通过这个功能，还可以进行简单的各种进制转换

举例：

```
(cskygdb)p /x g_tick_count  
(cskygdb)p /x 1000  
(cskygdb)p /t 1000
```

需要注意的是：有些局部变量会有编译器优化掉，就无法查看这个变量了。

p 命令是万能的，可以 p 变量地址，可以 p 变量内容，可以 p 函数地址，基本上是一个符号，都可以 p 出来看看。

设置断点

- 命令全名： breakpoint
- 简化： b

设置断点可以让程序自动停止在你希望停止的地方，断点可以以下面多种方式设置

- 行号
- 函数名
- 文件名：行号
- 汇编地址

举例：

```
(cskygdb)b 88  
(cskygdb)b main  
(cskygdb)b main.c:88  
(cskygdb)b *0x18600010
```

硬件断点

嵌入式芯片一般都有硬件断点可以设置，它相对于普通断点的不同是，该断点信息保存在 cpu 调试寄存器里面，由 cpu 通过运行时的比较来实现断点功能，而普通断点则是通过修改该处代码的内容，替换成特定的汇编代码来实现断点功能的。

需要注意的是：硬件断点的设置会影响 cpu 的运行速度，但是对于一些微型的嵌入式芯片，代码放在 flash 这种无法写入，只能读取介质上时，就只能通过设置硬件断点才能实现断点功能，普通的断点设置将不会生效。

设置硬件断点通过另外一个命令设置，举例：

```
(cskygdb)hb main
```

设置内存断点

- 命令全名： `watchpoint`
- 简化 : `watch`

设置内存断点可以在内存的内容发生变化的时候 自动停止运行。可以通过设置变量、内存断点

举例：

```
(cskygdb)watch g_tick_count  
(cskygdb)watch *0x18600010
```

内存断点和硬件断点是相同的原理，只要是 cpu 运行导致的内存修改都会自动停止运行。内存断点和硬件断点都会占用 cpu 的调试断点数，每个芯片都由固定有限的个数可供设置，一般为 4 个或者 8 个等。

查看断点

- 命令全名： `info breakpoint`
- 简化 : `i b`

举例：

```
(cskygdb) i b  
Num Type Disp Enb Address What  
1 breakpoint keep y 0x18704f9c in main at vendor/tg6100n/aos/aos.c:110  
2 breakpoint keep y 0x1871ca9c in cpu_pwr_node_init_static at  
kernel/kernel/pwrmgmt/cpu_pwr_hal_lib.c:88
```

使能断点

- 命令全名： `enable`
- 简化 : `en`

举例：

```
(cskygdb)en 1
```

禁止断点

- 命令全名： `disable`
- 简化 : `dis`

举例：

```
(cskygdb)dis 1
```

查看栈信息

- 命令全名： `backtrace`
- 简化 : `bt`

例如：

```
(cskygdb) bt
#0 board_cpu_c_state_set (cpuCState=1, master=1)
at vendor/tg6100n/board/pwrmgmt_hal/board_cpu_pwr.c:103
#1 0x1871cb98 in cpu_pwr_c_state_set_ (
all_cores_need_sync=<optimized out>, master=<optimized out>,
cpu_c_state=CPU_CSTATE_C1,
p_cpu_node=0x189d2100 <cpu_pwr_node_core_0>)
at kernel/kernel/pwrmgmt/cpu_pwr_hal_lib.c:275
#2 _cpu_pwr_c_state_set (target_c_state=CPU_CSTATE_C1)
at kernel/kernel/pwrmgmt/cpu_pwr_hal_lib.c:495
#3 cpu_pwr_c_state_set (target_c_state=CPU_CSTATE_C1)
at kernel/kernel/pwrmgmt/cpu_pwr_hal_lib.c:524
#4 0x1871d20c in tickless_enter ()
at kernel/kernel/pwrmgmt/cpu_tickless.c:381
#5 0x1871ce74 in cpu_pwr_down ()
at kernel/kernel/pwrmgmt/cpu_pwr_lib.c:70
#6 0x187095a4 in idle_task (arg=<optimized out>)
at kernel/kernel/rhino/k_idle.c:48
#7 0x1870bf44 in krhino_task_info_get (task=<optimized out>,
idx=<optimized out>, info=0x8000000)
at kernel/kernel/rhino/k_task.c:1081
Backtrace stopped: frame did not save the PC
```


选择栈帧

- 命令全名: `frame`
- 简化 : `f`

举例:

```
(cskygdb) f 2
#2 _cpu_pwr_c_state_set (target_c_state=CPU_CSTATE_C1)
at kernel/kernel/pwrmgmt/cpu_pwr_hal_lib.c:495
495 ret = cpu_pwr_c_state_set_(p_cpu_node, target_c_state,
master, FALSE);
```

选择了栈帧就可以通过 `p` 命令查看该栈函数内的局部变量了。(函数内的局部变量是存放在栈空间中的)

单步执行

- 命令全名: `next`
- 简化 : `n`

举例:

```
(cskygdb) n
```

单步执行进入函数

- 命令全名: `step`
- 简化 : `s`

举例:

```
(cskygdb) s
```

单步执行（汇编）

- 命令全名: `nexti`
- 简化 : `ni`

举例:

```
(cskygdb) ni
```

单步执行进入函数（汇编）

- 命令全名: `stepi`
- 简化 : `si`

举例:

```
(cskygdb) si
```

相对于 `s` 的单步执行, `si` 的单步执行精确到了汇编级别, 每一个命令执行一条汇编指令。对于优化比较严重的函数, `s` 的按行 单步执行 流程往往会比较混乱, 按汇编的单步执行则会比较符合芯片底层的逻辑。当然使用 `si` 单步调试程序, 也需要程序员对于汇编指令有比较好的了解, 调试难度也比较大。但是对于嵌入式程序, 编译器必然会对程序进行各种优化, `s` 的单步调试往往不是很好的选择。

完成当前函数

- 命令全名: `finish`
- 简化 : `fin`

举例:

```
(cskygdb) fin
```

当想跳出该函数调试时, 使用该命令会相当方便。但是该命令有一个限制, 当在不会支持普通断点的设备上调试时 (代码放在 flash 上执行), 这个命令需要配合另一条命令才能生效

```
(cskygdb) set debug-in-rom
```

这条命令的意思是, 告诉 gdb 这个代码是放在 flash 上的, 需要使用硬件断点才能使用 `fin` 命令, 这条命令只需要执行一次。

设置变量

- 命令格式:

```
set [variable] = [value]
```

举例:

```
(cskygdb) set g_tick_count = 100
```

```
(cskygdb) set *0x186000010 = 0x10
```

在调试一些程序逻辑时，通过设置变量数值可以让程序走期望的流程，来方便调试。

查看内存

- 命令格式

```
x /[n][f][u] [address]
```

其中：

- *n* 表示显示内存长度，默认值为 1
- *f* 表示显示格式，如同上面打印变量定义
- *u* 表示每次读取的字节数，默认是 4bytes
 - *b* 表示单字节
 - *h* 表示双字节
 - *w* 表示四字节
 - *g* 表示八字节

举例：

CPU Exception: NO.2

```
r0: 0x00000014 r1: 0x18a70124 r2: 0x00001111 r3: 0x10020000
r4: 0x00000000 r5: 0x00000001 r6: 0x00000002 r7: 0x07070707
r8: 0x00000000 r9: 0x09090909 r10: 0x10101010 r11: 0x11111111
r12: 0x40000000 r13: 0x00000000 r14: 0x18b166a8 r15: 0x186d9c0a
r16: 0x16161616 r17: 0x47000000 r18: 0x3f800000 r19: 0x00000000
r20: 0xc0000000 r21: 0x40000000 r22: 0x00000000 r23: 0x00000000
r24: 0x40400000 r25: 0x12345678 r26: 0x12345678 r27: 0x12345678
r28: 0x12345678 r29: 0x12345678 r30: 0x12345678 r31: 0x12345678
vr0: 0x12345678 vr1: 0x00000000 vr2: 0x00000000 vr3: 0x00000000
vr4: 0x00000000 vr5: 0x00000000 vr6: 0x00000000 vr7: 0x00000000
vr8: 0x00000000 vr9: 0x00000000 vr10: 0x00000000 vr11: 0x00000000
vr12: 0x00000000 vr13: 0x00000000 vr14: 0x00000000 vr15: 0x00000000
vr16: 0x00000000 vr17: 0x00000000 vr18: 0x00000000 vr19: 0x00000000
vr20: 0x00000000 vr21: 0x00000000 vr22: 0x00000000 vr23: 0x00000000
vr24: 0x00000000 vr25: 0x00000000 vr26: 0x00000000 vr27: 0x00000000
vr28: 0x00000000 vr29: 0x00000000 vr30: 0x00000000 vr31: 0x00000000
vr32: 0x00000000 vr33: 0x00000000 vr34: 0x00000000 vr35: 0x00000000
vr36: 0x00000000 vr37: 0x00000000 vr38: 0x00000000 vr39: 0x00000000
vr40: 0x00000000 vr41: 0x00000000 vr42: 0x00000000 vr43: 0x00000000
vr44: 0x00000000 vr45: 0x00000000 vr46: 0x00000000 vr47: 0x00000000
vr48: 0x00000000 vr49: 0x00000000 vr50: 0x00000000 vr51: 0x00000000
```

```
vr52: 0x00000000 vr53: 0x00000000 vr54: 0x00000000 vr55: 0x00000000  
vr56: 0x00000000 vr57: 0x00000000 vr58: 0x00000000 vr59: 0x00000000  
vr60: 0x00000000 vr61: 0x00000000 vr62: 0x00000000 vr63: 0x00000000  
  
epsr: 0xe4000341  
epc : 0x186d9c12
```

这条命令对于调试 踩内存，栈溢出等大量内存变化的场景非常有帮助。

2.4 快速上手调试

接下来，你可以找一块开发板，按照下面步骤体验 GDB 调试过程：

- 如前面介绍，下载并安装 DebugServer
- GDB 连上 DebugServer
- `lo` //灌入编译好的 elf
- `b main` //打断点到 main 函数入口
- `c` // 运行程序
- 如果顺利，这时程序应该自动停在 main 函数入口
- `n` // 单步执行下一行程序，可以多执行几次
- 找几个全局变量，`p` 查看结果

大部分开发板上电都会自动会运行程序，连上 DebugServer 就会停止运行。

注意事项

- 调试的时候 elf 文件 一定要和运行程序对应上，不然没法调试，使用一个错误的 elf 文件调试程序，会出现各种乱七八糟的现象。而且同一份代码，不同的编译器，不同的主机编译出来的 elf 都可能不相同。所以保存好编译出来的 elf 相当重要。
- 对于一些代码运行在 flash 的芯片方案，GDB 调试的时候要注意转换，和在 ram 上 GDB 调试命令有一些不一样。
- `watch` 只能 `watch` 到 cpu 的 内存更改行为，如果是外设（DMA 等）运行导致的内存变化，不能被 `watch` 到
- CKLink 连接开发板可能存在各种问题连接不上，要仔细检查，包括：开发板是否上电，芯片是否上电，芯片是否在运行，JTAG 排线是否插反等等。

3. CPU Exception 分析及调试

3.1 CPU Exception 案例

在开发板运行过程中，有时会突然出现如下打印，进而程序停止运行，开发板也没有任何响应：

```
CPU Exception: NO.2
r0: 0x00000014 r1: 0x18a70124 r2: 0x00001111 r3: 0x10020000
r4: 0x00000000 r5: 0x00000001 r6: 0x00000002 r7: 0x07070707
r8: 0x00000000 r9: 0x09090909 r10: 0x10101010 r11: 0x11111111
r12: 0x40000000 r13: 0x00000000 r14: 0x18b166a8 r15: 0x186d9c0a
r16: 0x16161616 r17: 0x47000000 r18: 0x3f800000 r19: 0x00000000
r20: 0xc0000000 r21: 0x40000000 r22: 0x00000000 r23: 0x00000000
r24: 0x40400000 r25: 0x12345678 r26: 0x12345678 r27: 0x12345678
r28: 0x12345678 r29: 0x12345678 r30: 0x12345678 r31: 0x12345678
vr0: 0x12345678 vr1: 0x00000000 vr2: 0x00000000 vr3: 0x00000000
vr4: 0x00000000 vr5: 0x00000000 vr6: 0x00000000 vr7: 0x00000000
vr8: 0x00000000 vr9: 0x00000000 vr10: 0x00000000 vr11: 0x00000000
```

这段打印表明程序已经崩溃。接下来以它为例，来一步一步分析如何调试和解决。

3.2 基础知识介绍

3.2.1 关键寄存器说明

- pc：程序计数器，它是一个地址指针，指向了程序执行到的位置
- sp：栈指针，它是一个地址指针，指向了当前任务的栈顶部，它的下面存了这个任务的函数调用顺序和这些被调用函数里面的局部变量。在我们的 cpu core 框架里，它对应了 R14 寄存器
- lr：连接寄存器，它也是一个地址指针，指向子程序返回地址，也就是说当前程序执行返回后，执行的第一个指令就是 lr 寄存器指向的指令，在我们的 cpu core 框架里，它对对应了 R15 寄存器
- epc：异常保留程序计数器，它是一个地址指针，指向了异常时的程序位置，这个寄存器比较重要，出现异常后，我们就需要通过这个寄存器来恢复出现异常时候的程序位置。
- epsr：异常保留处理器状态寄存器，它是一个状态寄存器，保存了出异常前的系统状态。

这几个重要的寄存器都在上面的异常打印中打印出来了。

3.2.2 关键文件说明

- yoc.elf: 保存了程序的所有调试信息，GDB 调试时必须用到该文件，编译完程序后务必保留该文件。
- yoc.map: 保存了程序全局变量，静态变量，代码的存放位置及大小。
- yoc.asm: 反汇编文件，保存了程序的所有反汇编信息。这些文件都保存在每个 solutions 目录中。如果使用 CDK 开发，则位于项目的 Obj 目录中。

其中:

- yoc.map 文件必须在编译链接的时候通过编译选项生成，例如: CK 的工具链的编译选项为 ```-Wl,- ckmap='yoc.map'`
- yoc.asm 文件可以通过 elf 文件生成，具体命令为 `csky-abiv2-objdump -d yoc.elf > yoc.asm`

3.2.3 异常号说明

在 XT CPU 架构里，不同的 cpu 异常会有不同的异常号，我们往往需要通过异常号来判断可能出现的问题。

异常号	说明
0	重启异常
1	未对齐访问异常
2	访问错误异常
3	除以零异常
4	非法指令异常
5	特权违反异常
6	跟踪异常
7	断点异常，地址观测异常
8	不可恢复错误异常

这些异常中，出现最多的是 1、2 号异常，4、7 偶尔也会被触发，3 号异常比较好确认，其余基本不会出现。

3.3 基本分析过程

GDB 准备及连接

参考第二节：[2. 使用 GDB 调试](#)。

恢复现场

在 GDB 使用 set 命令 将异常的现场的通用寄存器和 PC 寄存器设置回 CPU 中，便可以看到崩溃异常的程序位置了。

```
(cskygdb)set $r0=0x00000014
(cskygdb)set $r1=0x18a70124
(cskygdb)set $r2=0x00001111
(cskygdb)set $r3=0x10020000
...
(cskygdb)set $r14=0x18b166a8
(cskygdb)set $r15=0x186d9c0a
...
(cskygdb)set $r30=0x12345678
(cskygdb)set $r31=0x12345678
(cskygdb)set $pc=$epc
```

不同的 CPU 通用寄存器的个数有可能不相同，一般有 16 个通用寄存器、32 个通用寄存器两种版本，我们只需要把通用寄存器，即 r 开头的寄存器，设置回 CPU 即可。

pc, r14, r15 三个寄存器是找回现场的关键寄存器，其中 r14, r15 分别是 sp 寄存器和 lr 寄存器，pc 寄存器需要设置成 epc。其余的通用寄存器是一些函数传参和函数内的局部变量。

设置完成以后，通过 bt 命令可以查看异常现场的栈：

```
(cskygdb) bt
#0 0x186d9c12 in board_yoc_init () at vendor/tg6100n/board/init.c:202
#1 0x186d9684 in sys_init_func () at vendor/tg6100n/aos/aos.c:102
#2 0x186dfc14 in krhino_task_info_get (task=<optimized out>, idx=<optimized out>, info=0x11)
at kernel/kernel/rhino/k_task.c:1081
```

```
Backtrace stopped: frame did not save the PC
```

从 bt 命令打印出来的栈信息，我们可以看到 异常点在 init.c 的 202 行上，位于 board_yoc_init 函数内。

到这里，对于一些比较简单的错误，基本能判断出了什么问题。

如果没法一眼看出问题点，那我们就需要通过异常号来对应找 BUG 了。

3.4 通过异常号找 BUG

程序崩溃后，异常打印的第一行就是 CPU 异常号。

```
CPU Exception: NO.2
```

如上，我们示例中的打印是 2 号异常。

2 号异常是最为常见的异常，1 号异常也较为常见。4 号、7 号一般是程序跑飞了，运行到了一个不是程序段的地方。3 号异常就是除法除零了，比较好确认。其余的异常基本不会出现，出现了大概率也是芯片问题或者某个驱动问题，不是应用程序问题。

CPU Exception: NO.1

一号异常是访问未对齐异常，一般是一个多字节的变量从一个没有对齐的地址赋值或者被赋值。

例如：

```
uint32_t temp;  
uint8_t data[12];  
temp = *((uint32_t*)&data[1]);
```

如上代码，一个 4 字节的变量 temp 从一个单字节的数组中取 4 个字节内容，这种代码就容易出现地址未对齐异常。这种操作在一些流数据的拆包组包过程比较常见，这个时候就需要谨慎小心了。

有些 CPU 可以开启不对齐访问设置，让 CPU 可以支持从不对齐的地址去取多字节，这样就不会出现一号异常。但是为了平台兼容性，我们还是尽量不要出现这样的代码。

CPU Exception: NO.2

二号异常是访问错误异常，一般是访问了一个不存在的地址空间。

例如：


```
uint32_t *temp;  
*temp = 1;
```

如上代码，`temp` 指针未初始化，如果直接给 `temp` 指针指向的地址赋值，有可能导致 2 号异常，因为 `temp` 指向的地址是个随机值，该地址可能并不存在，或者不可以被写入。2 号异常也是最经常出现的异常，例如常见的错误有：

- 内存访问越界
- 线程栈溢出
- 野指针赋值
- 重复释放指针（free）

请注意你代码里的 `memset`、`memcpy`、`malloc`、`free`、`strcpy` 等调用。

大部分 2 号异常和 1 号异常的问题，异常的时候都不是第一现场了，也就是说异常点之前就已经出问题了。

比如之前就出现了 `memcpy` 的内存访问越界，内存拷贝超出变量区域了。`memcpy` 的时候是不会异常的，只有当程序使用了这些被 `memcpy` 踩了内存时，才会出现一号或二号异常。

这个时候异常点已经不是那个坑的地方了，属于“前人埋坑，后人遭殃”型问题。

如果是一些很快就复现的问题，我们可以通过 GDB watch 命令，watch 那些被踩的内存或变量来快速的定位是哪段代码踩了内存。

如果是一些压测出现的问题，压测了 2 天，出了一个 2 号异常，恭喜你，碰到大坑了。类似这种，比较难复现的问题，watch 已经不现实了。

结合异常现场 GDB 查看变量、内存信息和 reiview 代码逻辑，倒推出内存踩踏点，是比较正确的途径。

再有，就是在可疑的代码中加 log 日志，增加压测的机器，构造缩短复现时间的 case 等一些技巧来加快 BUG 解决的速度。

CPU Exception: NO.4/NO.7

四号异常是指令非法，即这个地址上的内容并不是一条 CPU 机器指令，不能被执行。

七号异常是断点异常，也就是这个指令是断点指令，即 `**bkt*` 指令，这是调试指令，一般代码不会编译生成这种指令。

这两种异常大概率是 指针函数没有赋值就直接跳转了，或者是代码段被踩了

例如：

```
typedef void (*func_t)(void *argv);
func_t f;
void *priv = NULL;
if (f != NULL) {
    f(priv);
}
```

如上代码，`f` 是一个 函数指针，没有被赋值，是一个随机值。直接进行跳转，程序就肯定跑飞了。

这种异常，一般 `epc` 地址，都不在反汇编文件 `yoc.asm` 中。

CPU Exception: NO.3

3 号异常是除零异常，也是最简单、最直接的一种异常。

例如：

```
int a = 100;
int b = 0;
int c = a / b;
```

如上代码，`b` 变量位 0，除零就会出现 三号异常。

3.5 不用 GDB 找到异常点

有些时候无法使用 GDB 去查看异常点，或者搭环境不是很方便怎么办？

这个时候我们可以通过 反汇编文件和 `epc` 地址来查看，异常的函数。

打开 `yoc.asm` 反汇编文件，在文件内搜索 `epc` 地址，就可以找到对应的函数，只是找不到对应的行号。

例如：

```
186d9b14 <board_yoc_init>:186d9b14: 14d3 push r4-r6, r15
186d9b16: 1430 subi r14, r14, 64
186d9b18: e3fffc6 bsr 0x186d9aa4 // 186d9aa4 <speaker_init>
186d9b1c: 3001 movi r0, 1
186d9b1e: e3fe3221 bsr 0x1869ff60 // 1869ff60 <av_ao_diff_enable>
186d9b22: e3fe4ca9 bsr 0x186a3474 // 186a3474 <booab_init>
186d9b26: e3ffe7d bsr 0x186d9820 // 186d9820 <firmware_init>
...
186d9bfc: 1010 lrw r0, 0x188d1a50 // 186d9c3c
```

```
<board_yoc_init+0x128>
186d9bfe: e00c6aeb bsr 0x188671d4 // 188671d4 <printf>
186d9c02: ea231002 movih r3, 4098
186d9c06: ea021111 movi r2, 4369
186d9c0a: b340 st.w r2, (r3, 0x0)
186d9c0c: 1410 addi r14, r14, 64
186d9c0e: 1493 pop r4-r6, r15
186d9c12: 9821 ld.w r1, (r14, 0x4)
186d9c14: 07a4 br 0x186d9b5a // 186d9b5a
<board_yoc_init+0x46>
186d9c14: 188d19c0 .long 0x188d19c0
```

如上的汇编代码，根据异常的 epc 地址 0x186d9c12，我们可以确认异常的函数发生在 board_yoc_init 内。



平头哥芯片开放社区交流群
扫码关注获取更多信息



扫码注册平头哥 OCC 官网
观看各类视频及课程



阿里云开发者"藏经阁"
海量免费电子书下载