蜂鸟标准化离线方案
产品使用说明书

V1.2.0

2020 年 3 月

云知声智能科技股份有限公司

拥有完全自主知识产权的世界顶尖智能语音识别技术

重要声明

版权声明

商标声明

云知声智能科技股份有限公司的产品是云知声智能科技股份有限公司专有。在提及其他公司及其产品时将使用各自公司所拥有的商标，这种使用的目的仅限于引用。本文档可能涉及云知声智能科技股份有限公司的专利（或正在申请的专利）、商标、版权或其他知识产权，除非得到云知声智能科技股份有限公司的明确书面许可协议，本文档不授予使用这些专利（或正在申请的专利）、商标、版权或其他知识产权的任何许可协议。

不作保证声明

云知声智能科技股份有限公司不对此文档中的任何内容作任何明示或暗示的陈述或保证，而且不对特定目的的适销性及适用性或者任何间接、特殊或连带的损失承担任何责任。本手册内容若有变动，恕不另行通知。本手册例子中所用的公司、人名和数据若非特别声明，均属虚构。未得到云知声智能科技股份有限公司明确的书面许可，不得为任何目的、以任何形式或手段（电子的或机械的）复制或传播手册的任何部分。

保密声明

本文档（包括任何附件）包含的信息是保密信息。接收人了解其获得的本文档是保密的，除用于规定的目的外不得用于任何目的，也不得将本文档泄露给任何第三方。

本软件产品受最终用户许可协议（EULA）中所述条款和条件的约束，该协议位于产品文档和/或软件产品的联机文档中，使用本产品，表明您已阅读并接受了 EULA 的条款。

版本变更记录

编号	版本号	变更内容
1	v1.0.0	第一版正式版
2	v1.1.0	**更新 2.1 创建产品** - 导航栏变更，入口提示修改 **更新 2.2.2 配置终端能力** - 新增自定义命令词与答复部分，非语音交互播报定制与竞争词使用的规则介绍 - 新增开机音自定义介绍 - 新增主动退出配置介绍 - 调整配置介绍顺序
3	v1.1.1	文字描述细节修改
4	V1.2.0	**更新 2.2.2 配置终端能力** - 新增 AEC 开关功能 - 新增选择语音识别模型功能 **更新 3. 烧录使用** - 烧录工具 2.0.4 更新，更新指导说明 **更新 5.5 外设开发** - 调整 GPIO, I2C, ADC 的使用说明，新增 PWM 使用说明 **更新四、常见问题** - 调整 2. 烧录版本常见问题 **新增 5.6 watchdog 开发** **新增五、模型训练平台介绍**

目录

一、简介

1. 引言

云知声总部位于北京，在上海、深圳、厦门设有全资子公司。是一家专注物联网人工智能服务、拥有完全自主知识产权的世界顶尖的智能语音识别和语义理解技术的高新技术企业。自 2012 年由智能语音技术起家，云知声多年来不断拓展技术边界，技术能力不仅涵盖了感知、认知、交互等方面的人工智能语音语义技术，更是在突破了 AI 芯片设计技术后，一举打造出了自主研发、自主知识产权、自主可控的全栈式人工智能技术链条（深度学习算法+大数据+超级计算+芯片能力）。

以强大的计算平台为基础，云知声构建了“云端芯”生态体系，提供跨硬件平台、跨应用场景的软硬件一体化云端+本地智能的芯片终端解决方案。目前云知声是迄今为止行业内唯一实现芯片落地应用的公司、国内白色家电领域领先 AI 芯片供应商、国内首家推出医疗云服务并完成数百家医院系统测试的语音服务商、我国后装车机市场占有量行业第一、教育云社会化口语评测服务市场占有量第一。还在智慧生活（家居、车载、机器人等）和智慧服务（医疗、教育、司法等）等多个垂直场景均有落地应用。

2. 产品简介

云知声提供标准化软硬件一体离线语音解决方案，支持双麦阵列、前端降噪、语音唤醒、离线识别等。用户使用方案中的标准化硬件模组配合云知声设备平台与相关工具，快速定制语音产品，为空调、冰箱、灯具、开关等家居设备赋能，无需联网即可使其具备智能语音交互能力。

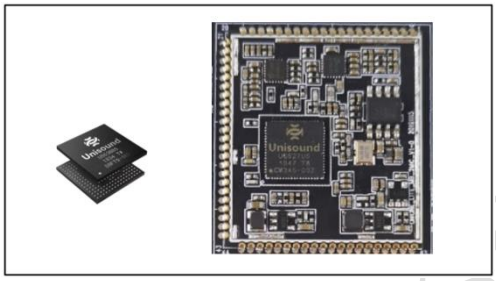
- **硬件：**基于云知声蜂鸟芯片为核心的标准硬件模块板，集成离线识别算法
- **设备平台：**可进行语音相关配置并生成软件版本的云知声智能设备平台
- **软件与源码：**可直接使用，也可修改代码进行二次编译的软件版本
- **文档与工具：**使用指南与技术开发文档，配套测试工具

二、产品说明

1. 硬件说明

蜂鸟芯片采用 CPU+DSP+NPU 三核架构，内置基于人工智能语音识别算法的 NN 硬件加速核，通过神经网络对音频信号进行训练学习，提高语音信号的识别能力。

蜂鸟芯片具有丰富的系统外设，包括 UART, I2C, SPI, PWM, RTC, Timer, ADC 等，支持 RC-5 和 NEC 红外协议，是一款低功耗高性能，主控级别的离线语音识别芯片。



蜂鸟芯片与标准核心板

蜂鸟芯片主要特性：

配置	规格
蜂鸟芯片	内存 1.5MB
	FLASH 8MB
mic 数量	2
AEC	1
VAD	有
TF 卡	调试
UART 通信接口	4 路
LED 接口	有
音频接口	PDM, I2S
控制接口	SPI, UART, I2C, GPIO, PWM 等

2. 软件功能

序号	功能	功能描述
1	**双 mic 降噪及 AEC**	• 2mic 线性麦克风阵列 • 支持回声消除、去混响及用户打断 • 支持家居场景 5m 的远讲
2	**声源定位**	可定位说话人方位 (±10°)
3	**语音唤醒**	• 高性能唤醒引擎 • 低功耗 • 支持带口音的普通话 • 低误唤醒率 (< 1 false in 48 hours)
4	**离线识别**	• 支持本地 100 条控制指令识别
5	**随时打断**	支持 AEC，可随时使用唤醒词打断进行中的对话，进行下一次对话。
6	**多轮对话**	一次唤醒连续对话，语音操作更加便捷自然。
7	**多种发音人音色**	• 提供标准女声、甜美女声、可爱女声、台湾女声、标准男声、女童声、男童声七种音色可选
8	**UART 主板对接**	云知声提供标准 UART 协议，也支持对接用户自有协议
9	**加密保护**	支持 secure boot，提供更加完善加密保障。
10	**智能设备平台**	云知声设备平台提供：**麦克间距配置，唤醒词自定义，命令词自定义，回复播报语自定义，发音人音色选择**等产品自定义配置

三、操作指南

1. 前期准备

1.1 账号创建

- 进入云知声开放平台：http://dev.hivoice.cn/
- 点击右上角的 **登陆/注册**，弹出如下窗口：

- 如无账号，请点击 **去注册**；
- 如已有云知声开发账号请忽略本步骤；

1.2 完善资料

注册成功后，为确保账号可用，请登陆云知声账号；
登陆后，点击右上角的 **完善资料**，输入相关必填信息。

1.3 开通权限

1. 联系云知声商务，申请开通云知声智能设备平台权限，获取离线标准开发板；
2. 开通权限后，产品控制台入口可见，在控制台-产品接入下。

2. 配置并生成产品版本

2.1 创建产品

（1）用户登陆云知声开放平台（http://dev.hivoice.cn/），点击导航栏中的 **控制台**，点击 **产品接入** 进入产品接入控制台。

（2）进入产品控制台后，点击 **创建产品**。

（3）云知声智能设备平台提供不同场景，用以定制不同语音产品形态，此处选择**智能家居（纯离线）**

（4）选择系统与芯片平台，选择 **蜂鸟芯片**

该页面点击芯片旁的**？**按钮，可以获取蜂鸟芯片产品手册。

（5）填写产品信息，包括名称与描述

（6） 确认所选项无问题，点击 **保存且下一步**，即初步创建产品成功

2.2 产品版本开发

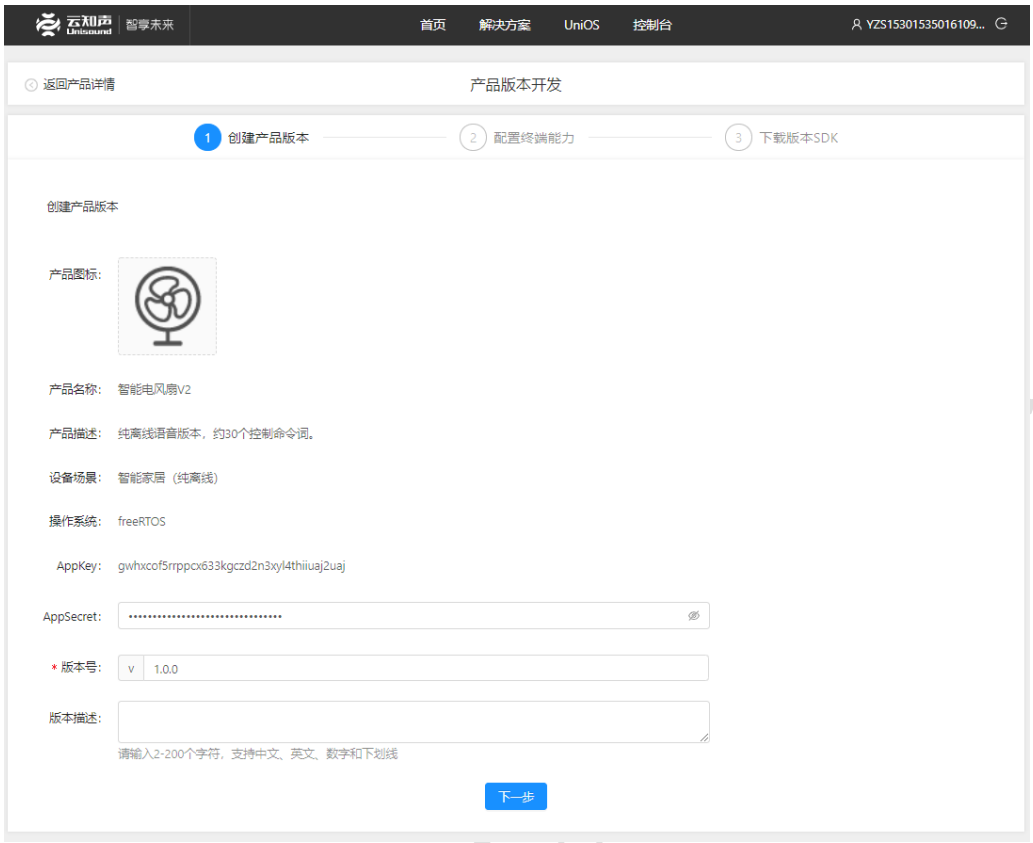
2.2.1 创建产品版本

（1）创建产品版本，填写版本号及版本信息，点击下一步

- 产品创建成功后，系统会自动为该产品分配 Appkey 与 AppSecret，此为该产品唯

一识别 ID 和密匙

- 一个产品可维护多个版本，可通过版本号与版本描述来区分



2.2.2 配置终端能力

进入终端能力配置，首先是语音产品基础能力表，介绍了一个语音类型产品应有的基本能力。

（1）前端信号处理相关配置

麦克风间距配置

- 麦克风配置默认为 2mic，默认麦克间距为 70mm
- 建议麦克间距为 60mm - 120mm，对超过产品支持的间距有输入限制；
- 如产品无特殊的麦克分布要求，**请使用默认推荐值**，以便采集的语音获得最优算法的降噪处理
- 点击 **问号**，可以看到麦克间距的具体定义

AEC（回声消除）配置

- 默认 AEC 配置打开，可将麦克风接收到的设备自噪声成分（设备自己播放的内容）消除掉。
- 如果关闭此项，设备将难以区分自噪声与外部声音，在音乐播放、TTS 播报等过程中将难以被唤醒词打断。

AEC 原理解释：

经过功放（PA）后的模拟信号，一路传给喇叭进行播放，另外一路需要经过 A/D 转换后进行保存。此时经过喇叭播放的信号会经过空气传播重新被 mic 所采集到，另外一路经过 A/D 后保存的信号则作为参考信号，供 AEC 算法使用。

（2） 自定义产品唤醒词，以及唤醒回复语

- 支持 3-5 个字的唤醒词自定义配置；
- 为保证语音产品有最佳唤醒体验，平台针对唤醒词的发音提供打分功能，并会提供具体指导信息；
- 评分不通过的唤醒词则不可使用；
- 完整唤醒词定义规则可点击界面上《唤醒词自定义规则》
- 唤醒回复最多支持 5 条，设置后将随机播报

（3）按照规则填写命令词与应答语

如下图所示，左部分为语法格式说明与规则介绍，右部分则为输入框，可以在页面内直接输入命令词与应答语的定义，如不满足语法格式要求平台则会进行实时提示。

语法格式说明与规则介绍

为方便产品构建，平台定义了一套语法格式，语法格式组成为：

action=命令词 1|命令词 2...@回复语

举例：**TempSet15**=设置十五度|十五度@已设为十五度

action 为一个控制指令的唯一标识，用户对着设备说出“设置十五度”“十五度”并被语义理解时，如已对接设备，语义理解模块会将 TempSet15 传给设备。

命令词 为想要定义的语音话术，用户必须按照定义的话术说出才有效。如用户可以使用“设置十五度”“十五度”来实现同一个设置温度 15 度的控制。

回复语 为针对该条控制指令的设备回复播报。

action、命令词、回复语均由用户定义。等号前的 action 为唯一值；等号后的命令词可输入多个，以 | 符号分割；@后为回复语，每个控制指令可以定义对应的回复。

点击语法格式后的 **问号**，可看到语法格式要求：

- action 由英文、下划线“_”和数字组成，必须英文开头，不区分大小写，15 个字符内
- 命令词最多支持 100 条，每条限 2 - 10 个字符，仅支持中文
- 一个 action 仅支持一条回复语，回复语总字数不得超过 500 个字符，支持中文、数字、逗号、句号、问号

点击语法句式后的 **问号**，可以看到针对回复播报特殊读音的标签使用说明：

- **<py>**：需要对单个汉字的发音进行纠正的场合。
  注：拼音声调范围为 1 - 5，1 - 4 对应一声到四声，5 对应轻声。
  例：已调<py>tiao2</py>至中<py>zhong1</py>风档
  播报为：已调(tiao2)至中(zhong1)风档
- **<value>**：需要将数字按照数值读法播报
  例：已设为<value>15</value>度
  播报为：已设为十五度
- **<code>**：需要将数字按照数字串逐位播报
  例：已设为<code>15</code>度
  播报为：已设为一五度

开发者注意：

1. 产品默认支持音量大小调节，音量调节控制默认句式如下：

- **volumeUpUni=增大音量**
- **volumeDownUni=减小音量**

如需自定义音量的控制话术与应答播报，可在语法文件内添加以上控制句式，只需保持=号前的 action 不变，修改=后的内容即可。

举例如下：

volumeUpUni=大声一点|调大声点@已为您调高音量

volumeDownUni=减小音量@好的，音量已减小

2. 产品内置音量等级三档控制逻辑，默认不开启。如需使用该逻辑，可在定义离线命令词与播报语处，添加词条后生效：

- 最大音量 **volumeMaxUni**
- 中等音量 **volumeMidUni**
- 最小音量 **volumeMinUni**

如希望添加最大音量调节，可参考下方举例添加句式：

volumeMaxUni=最大音量|音量调到最大@已调至最大音量

3. 用户可以在离线命令词自定义部分，指定特定的 action，定义非语音交互的应答语播报，用于非正常语音交互的情景。

开发者需从指定文件夹获取此部分定制的语音音频文件，并根据产品定义修改源码。可参见文内 5.3 播报逻辑开发。

格式如下：

Special_Replies@回复语 1|回复语 2|回复语 3

如警报播报：以智能烧水壶为例，产品内传感器检测到缺水状态，自动语音警报；
如极限播报：空调指令“升高温度”，正常播报为“温度已升高”；当温度到上限时，播报“温度已调至最高”。
举例如下：
Special_Replies@内胆干烧|水壶缺水|没水啦|主人请加水

4. 用户可以在离线命令词自定义部分，指定特定的竞争词（可理解为无效命令词），防止一定概率的误识别。
格式如下：

Special_Gabages=命令词 1|命令词 2|命令词 3|...

以空调产品为例，并没有高于 32 度的温控指令，用户说出“三十六度”时，不希望空调识别到“十六度”；
以空调产品为例，没有自动风的指令，当用户说说出“自动风”时，不希望空调识别到“自然风”（如果阈值设置较低）。
举例如下：
举例：Special_Gabages=三十六度|四十六度|五十六度|六十六度

（4）选择语音识别模型

蜂鸟芯片方案支持对声学模型进行选择，默认为云知声针对蜂鸟芯片定制的经过算法优化的通用声学模型，即“系统默认”。

如当前系统默认的声学模型性能无法满足用户需求时（注：此处用户需求，指经过语音性能量化测试的明确指标），可以使用云知声模型训练平台，采集特定离线命令词的语音训练数据并进行模型定制。可点击选择框右侧文字跳转至模型训练平台。

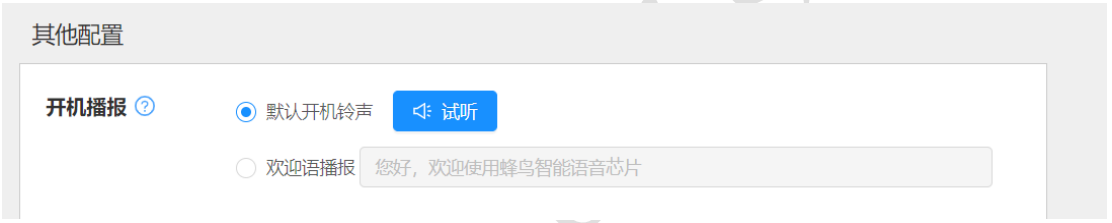
平台使用的详细介绍，请见：五、模型训练平台介绍

（4）针对产品定位，选择适合产品形态的发音人

- 活泼可爱的萱萱和 KiYo，童真童趣的糖糖和天天，温柔可亲的玲玲，庄重严肃的小雯玉与小峰，七种音色可供选择；
- 可调节音量与语速；
- 提供在线试听，方便用户进行决策；

（5）其它配置

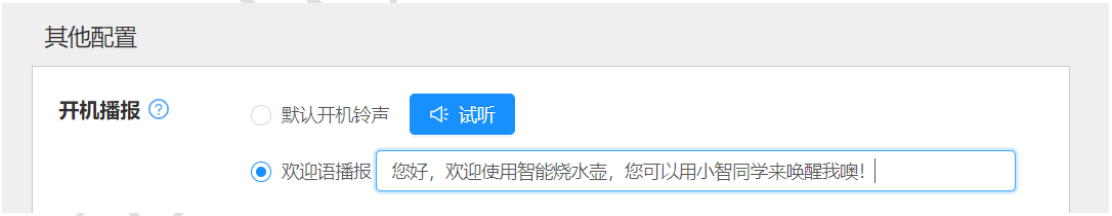
- **开机音配置**



产品默认为开机铃声，可在页面点击试听。

如需进行开机欢迎语的自定义，也可以选择自定义内容，产品则会在开机后自动进行播报。

欢迎语播报自定义限中文，可输入 30 个字符。



- **识别状态退出相关配置**

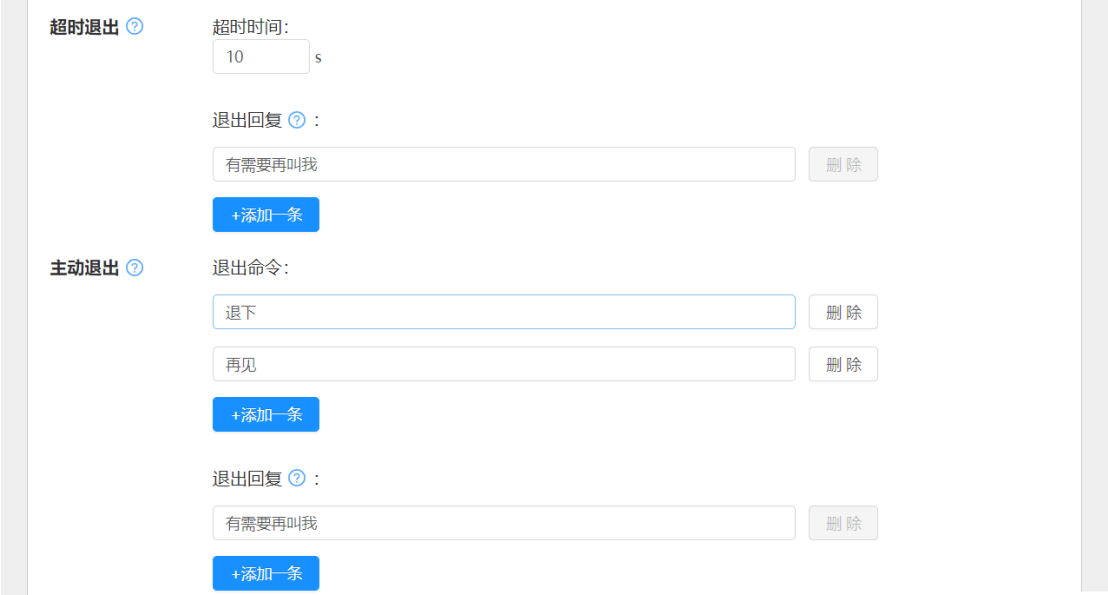
语音产品为保证用户体验，唤醒后一段时间内无语音输入产品会主动退出识别状态，如需输入命令词，则需再次唤醒。

此段等待用户输入语音命令的有效时间，即为“唤醒超时退出时间”。

- 为确保产品有较好的用户体验，支持 5s – 15s 的超时退出时间设置，默认为 10 秒
- 超时退出回复最多可设置 2 条

当用户想要结束对话时，也可以主动要求退出识别状态。

- 默认退出命令为“退下”“再见”，可根据产品定义修改，最多可设置 3 条
- 主动退出回复最多可设置 2 条



如配置完成，则可点击下一步，进入版本发布页面。

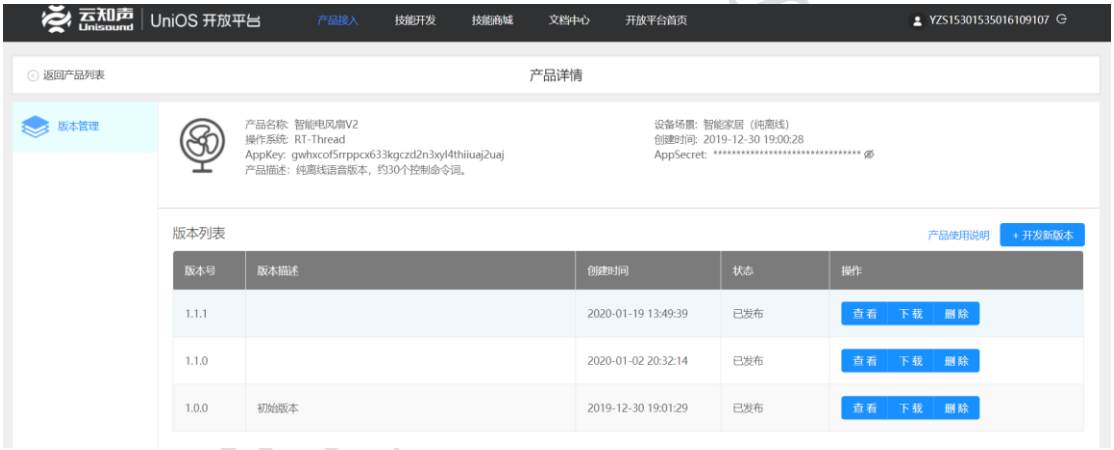
2.2.3 下载版本

（1）产品配置开发完成，进入发布页面后，可点击 **立即发布**

（2）产品发布需要一定时间，可以有两种方法获取到发布成功的产品

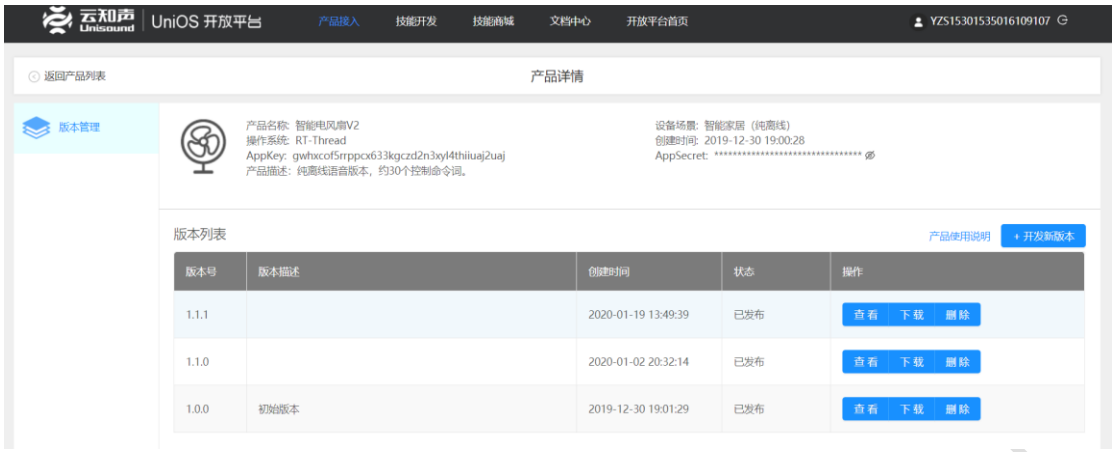
- 发布界面下，版本发布成功后，会有下载提示

- 也可以直接返回产品详情，或关闭页面一段时间再进入智能设备平台的产品详情，页面中的版本列表处点击下载



2.3 产品版本维护

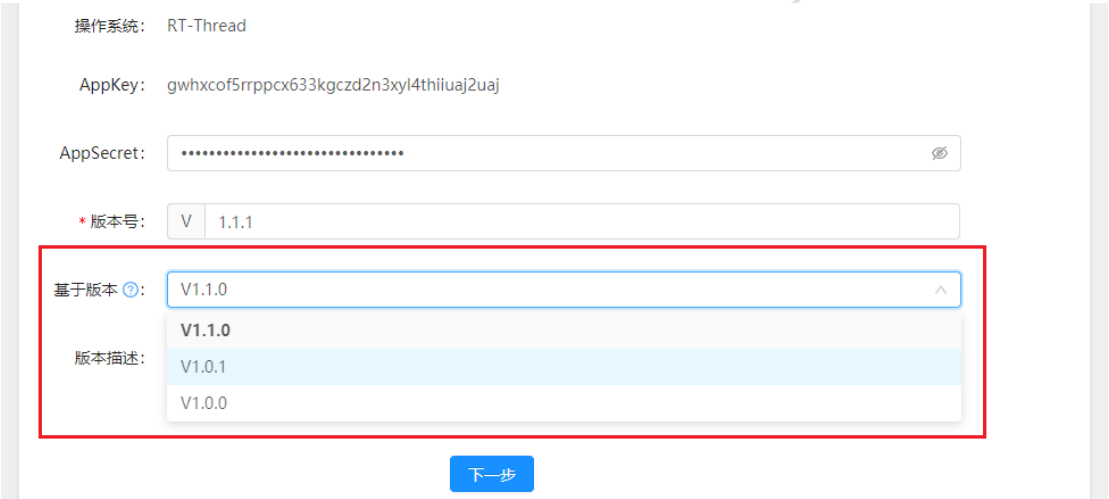
产品详情页面可以查看已创建的产品，可以查看每个版本的配置信息，也可以对已发布成功的版本进行下载或者删除操作



4.3.1 新版本迭代

同一个产品如有新需求的修改或迭代，可在产品详情页面点击 **+开发新版本**，后续的操作步骤同创建产品版本时的流程一致。

提供选择任一历史版本进行迭代开发的选项，默认为基于最新修改的版本，也可根据需求进行其它版本的选择。



3. 烧录使用

3.1 文件介绍

- **产品的软件烧录包下载成功后，解压文件，可以看到如下方文件目录**

- 如想直接使用，体验语音功能，请打开 image_demo 文件夹，右键单击烧录工具

UniOneDownloadTool.exe，“以管理员身份运行”打开

3.2 烧录指南

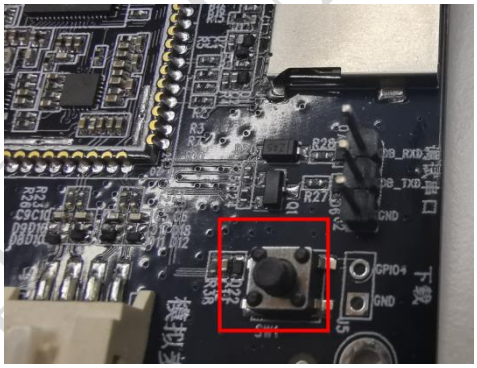
开始烧录之前，需要准备：

- Micro USB 连接线，用于供电
- UART 转接头与串口线，用于烧录

3.2.1 进入烧录模式

找到调试串口附近的短接按钮，按住不放，同时使用 USB 连接线给开发板上电，即接入电源。可直接接电脑，或者任意 5V 电源适配器（如手机充电器）。

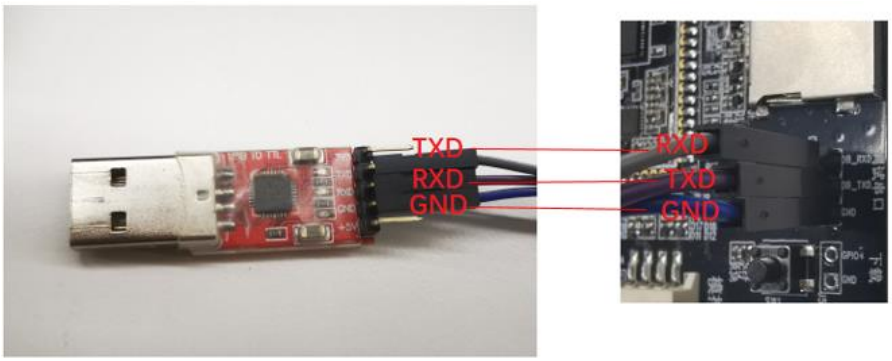
上电完毕后松开烧录按钮，此时开发板进入烧录模式。



烧录按钮示意图

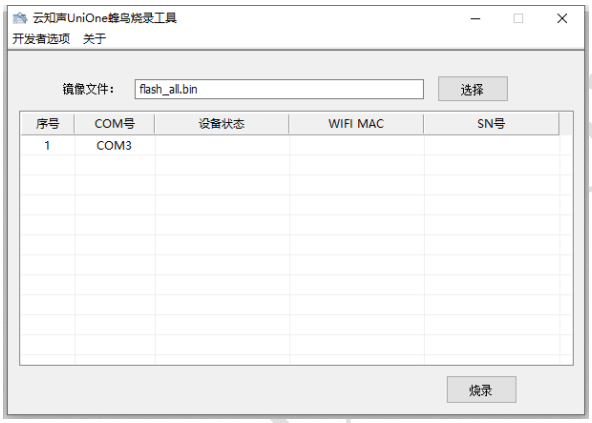
3.2.2 接入 UART

开发板上调试串口有：GND，RXD，TXD 三路，而 UART 转接头上同样 GND，RXD，TXD 是我们需要使用的。当串口线与 UART 转接头接入时，RXD TXD 需交叉对接，即 GND-GND，RXD-TXD，TXD-RXD，如下图所示：



UART 口反接示意图

打开烧录工具 UniOneDownloadTool.exe，串口线连接上 UART 转接头与开发板后，转接头插入电脑，烧录工具会如下图显示：



烧录工具检测到 COM3

如未显示则表示未识别到 COM 口，请检查：

① 串口线是否接好

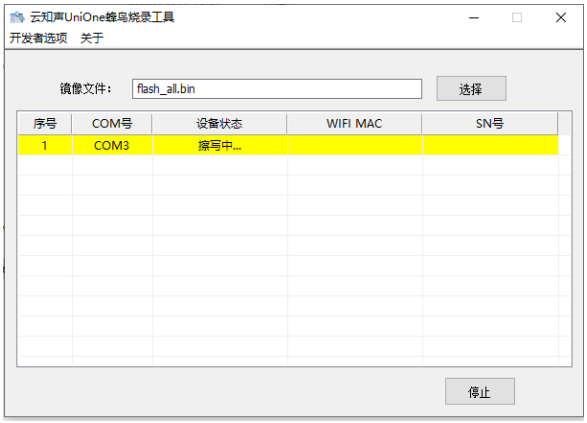
② 串口线是否反接

③ 是否安装 UART 对应驱动。

3.2.3 烧录版本

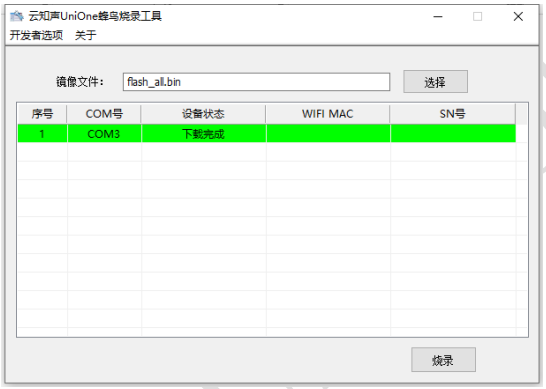
正常烧录版本，直接点击烧录按键，对应 COM 号链接的设备即开始烧录，最多支持 12 台设备同时烧录。

烧录中的设备处于黄色选中状态，且设备状态实时更新烧录进展。



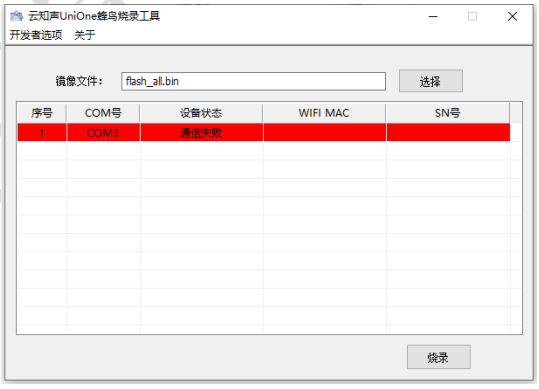
工具烧录中显示

烧录完成，烧录成功的设备将为绿色显示，如下图：



设备烧录成功

而烧录失败设备将处于红色显示，如下图：



设备烧录失败

如烧录失败，请检查：

① 开发板是否供电

② 开发板串口是否接好

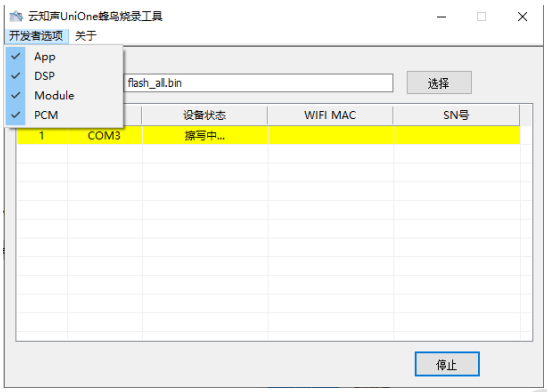
③ 串口线是否按照要求反接，确认后请回到步骤 1 重新短接上电，重新进入烧录模式。

④ COM 口是否超过了 COM16，如果超过，请参考四、常见问题：烧录版本常见问题

3.2.4 开发者选项（非开发者不必关注）

3.2.4.1 烧录部分 BIN 文件

只选择烧录部分文件情况下，可以点击开发者选项，取消勾选某些镜像文件（默认为全选），也可以点击选择按钮，选择其它路径的 BIN 文件来进行烧录。



开发者选项示意图

3.2.4.2 更新部分 BIN 文件

此处以更新 app.bin 为例：

(1) 确保基础的整包 flash_all.bin 已经完整升级过一次，已生成 addtion\Image\ImageBin 目录；

(2) 将烧录工具的文件输入栏中的 flash_all.bin 清空，或者删除 flash_all.bin 后再打开工具（如删除请记得备份）；

(3) 将更新的 app.bin 替换 addtion\Image\ImageBin 下的 app.bin；

(4) 开发者选项中，只单选 APP，点击烧录即可。

此方法同样适用于 dsp.bin, module.bin, pcm.bin, config.bin 等文件。

开发者选项示意图

3.3 配置文件的使用

用户点击了烧录按键后，烧录工具会解压 flash_all.bin 为一个 addtion 的文件夹，里边包含所有的需要烧录的 bin 文件，其中一个名为 config.bin 的文件，支持修改来进行相关配置。

如需进行配置，请进行如下操作：

① 打开 config.bin 进行修改并保存，正常的文本编辑工具即可，如记事本、写字板，

推荐使用 Notepad++；

② 将烧录工具的文件输入栏中的 flash_all.bin 清空，或者删除 flash_all.bin 后再打开工具（如删除请记得备份）。

具体每个参数如何配置，请见下方介绍。

3.3.1 lasr 本地语音识别相关配置

（1）**lasr_wkp** 本地语音识别：唤醒

- "std_threshold"：唤醒阈值，未进行语音性能测试确定阈值前，不建议修改
- "lp_threshold"：二次唤醒阈值，未进行语音性能测试确定阈值前，不建议修改
- "sleep_std_threshold"：深度休眠唤醒阈值，未进行语音性能测试确定阈值前，不建议修改
- "sleep_lp_threshold"：深度休眠二次唤醒阈值，未进行语音性能测试确定阈值前，不建议修改

（2） **lasr_asr** 本地语音识别：识别

- "timeout"：识别模式退出带唤醒模式超时时间，单位为秒（也可平台进行配置）
- "std_threshold"：识别阈值，未进行语音性能测试确定阈值前，不建议修改
- "lp_threshold"：二次识别阈值，未进行语音性能测试确定阈值前，不建议修改

3.3.2 log 设置

- "enable"：默认关闭。0：关闭 log 输出；1：打开 log 输出
- "set_level"：默认为一般调试信息。可选择范围 0 ~ 6，值越大打印等级越高，log 输出越多。0：输出 error log，1：输出 warning log，2：输出一般调试信息，3：输出 debug 信息
- "arpt_enable"：默认关闭。0：关闭 ARPT 工具对应 log 输出；1：打卡 ARPT 工具对应 log 输出。该功能不受 log.enable 影响，一般打开 ARPT 信息时应禁用 log 输出，避免输出信息错乱

3.3.3 record 录音配置

- "enable"：默认关闭。0：关闭录音保存到 SD 卡功能；1：将录音保存到 SD
- "save_asr_only"：默认保存全部录音数据。0：录音保存 4 路全部数据；1：录音仅保存 SSP 处理后数据

3.3.4 hvad 硬件人声检测配置

该部分不建议进行修改。

- "enable"：默认开启 HAVD。0：关闭 HVAD 功能；1：打开 HVAD 功能
- "time_to_sleep"：进入 HVAD 低功耗模式超时时间，单位为秒
- "eage_det"：HVAD 检测行为阈值，不建议修改
- "energy_threshold"：HVAD 触发音量值，值越大唤醒所需音量越大

· "trig_num"：HVAD 触发持续度阈值，值越大唤醒越困难

4. 开发前准备

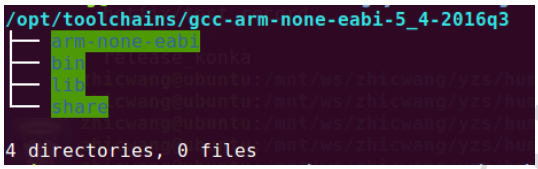
4.1 环境准备

（1）下载编译工具

推荐使用 Ubuntu 1604 / centos7 以上版本作为开发环境。

交叉编译工具下载地址：https://pan.baidu.com/s/15xPkCTDAn5PyFHQxlEVdxA 提取码：nah9

下载交叉编译工具并解压到开发环境中；



交叉编译工具层次图

（2）已下载的版本里除了可直接烧录使用的 image_demo 外，另外一个 ree_unione_lite_app 文件夹则为产品方案源码，如下图所示：

复制 rtt_unione_lite_app 文件夹至开发环境中；

（3）修改 rtconfig.py 中交叉工具链的路径为实际解压到的工具路径；

```
import os

# toolchains options
ARCH         ='arm'
CPU          ='arm9'
CROSS_TOOL   ='gcc'

if os.getenv('RTT_ROOT'):
    RTT_ROOT = os.getenv('RTT_ROOT')
else:
    RTT_ROOT = '../..'

if os.getenv('RTT_CC'):
    CROSS_TOOL = os.getenv('RTT_CC')

if  CROSS_TOOL == 'gcc':
    PLATFORM    = 'gcc'
    EXEC_PATH   = '/opt/toolchains/gcc-arm-none-eabi-5_4-2016q3/bin'
else:
    print('Please make sure your toolchains is GNU GCC!')
    exit(0)

if os.getenv('RTT_EXEC_PATH'):
   EXEC_PATH = os.getenv('RTT_EXEC_PATH')

```

（4）安装 scons 软件构造工具 sudo apt-get install scons

（5）安装 python 环境：sudo apt-get install python

（6）安装 sox 音频处理工具：sudo apt-get install sox

4.2 编译

(1) 执行编译

执行./build.sh，编译成功会看到如下信息：

```
LINK rtthread.elf
arm-none-eabi-objcopy -O binary rtthread.elf rtthread.bin
arm-none-eabi-size rtthread.elf
   text    data     bss     dec     hex filename
  98599     872   17880  117351   1ca67 rtthread.elf
scons: done building targets.
WRN:skip non-bin: ./pack.py
====== make debug bin done======
====== build done ==> output/ ======
```

(2) output 目录下可以看到编译生成的完整固件，使用章节 3 中的下载工具即可进行烧录使用：

release 对应正式量产版本，整体性能最佳；

debug 对应研发版本，实时输出调试 log；

```
rtt_unione_lite_app/output/
├── debug_flash_bin_creater
├── release_flash_bin_creater
├── debug_flash_all.bin
└── release_flash_all.bin
```

(3) output/debug_flash_bin_creater 或者 output/release_flash_bin_creater 目录下 app.bin 即对应当前源码编译出的应用固件，可参考章节 3 中的指导手册进行单独 app.bin 烧录，减少烧录时间，方便开发。

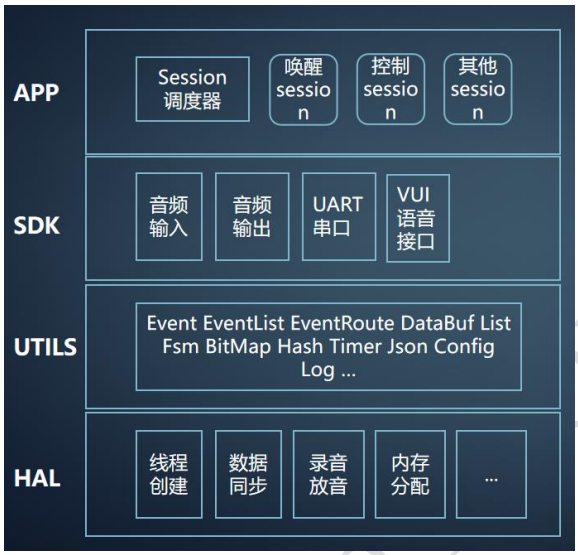
5. 二次开发

5.1 软件架构

开发工程中所涉及到的代码路径为 applications，层次划分如下图：

```
rtt_unione_lite_app/applications/
├── app
├── hal
├── sdk
├── utils
└── SConscript

4 directories, 1 file
```



5.2 交互方式开发

常见交互方式分为：

a) 一次唤醒，多次识别；

b) 一次唤醒，一次识别。

由于蜂鸟芯片定位为智能家居产品，为更贴近用户使用习惯，默认为一次唤醒多次识别。在源码中，是通过 app/inc/uni_session.h 中的 MULTI_DIALOGUE_MODE 宏进行控制：

```c
#ifdef __cplusplus
extern "C" {
#endif

#include "uni_iot.h"
#include "list_head.h"
#include "uni_event.h"
#include "uni_databuf.h"

#define MULTI_DIALOGUE_MODE          (1)  /* 1 multiply dialogue mode,
                                             0 single dialogue mode */

/**
 * Usage:   Callback function type used to determine if a event can be handled
 *          by the session in its current state.
 *          Implemented by session handler and called by session manager
 * Params:  event    event to be handled, cannot be NULL
 * Return:  Query result. Should be one of the following value:
 *          (1) E_OK:     event can be handled by this session
 *          (2) E_REJECT: event cannot be handled by this session
 *          (3) E_FAILED: something is wrong
 */
applications/app/inc/uni_session.h
```

下面具体介绍下两种交互方式的区别：

app/src/sessions 目录下的文件为各个场景业务逻辑的具体实现，以最常用的设备控

制场景为例，控制处理逻辑可以抽象为一个状态机，在代码中实现如下图：

```
static int _session_transition_init(void) {
  MicroFsmTransition session_transition[] = {
    {STATE_IDLE,          ID(VUI_APP_VOLUME_SETTING_EVENT), idle__vui_app_volume_setting},
    {STATE_IDLE,          ID(VUI_APP_SETTING_EVENT),        idle__vui_app_setting},
    {STATE_BROADCAST,     ID(VUI_APP_VOLUME_SETTING_EVENT), broadcast__vui_app_volume_setting},
    {STATE_BROADCAST,     ID(VUI_APP_SETTING_EVENT),        broadcast__vui_app_setting},
    {STATE_BROADCAST,     ID(AUDIO_PLAY_END_EVENT),         broadcast__audio_play_end},
    {STATE_BROADCAST,     ID(COMMON_STOP_EVENT),            broadcast__common_stop},
  };
  g_setting_session_transition = uni_malloc(sizeof(session_transition));
  if (NULL == g_setting_session_transition) {
    LOGE(SETTING_SESSION_TAG, "malloc failed !");
    return 0;
  }
  uni_memcpy(g_setting_session_transition, session_transition,
             sizeof(session_transition));
  return (sizeof(session_transition) / sizeof(session_transition[0]));
}
```

状态 事件 响应

代表设备控制过程中共有两个“状态”（空闲状态，处理完事件后的语音回复状态），每个“状态”在接收到指定“事件”时会做出对应的“响应”，在“响应”中实现具体操作。

以第一行{STATE_IDLE, ID(VUI_APP_VOLUME_SETTING_EVENT), _idle__vui_app_volume_setting}为例，代表设备在空闲状态下接收到一个本机音量调节事件，将通过调用_idle__vui_app_volume_setting 进行处理：

```
static Result _idle__vui_app_volume_setting(void *event) {
  Result rc;
  LOGT(SETTING_SESSION_TAG, "action called");
  rc = _volume_setting_process((cJSON *)(((Event *)event)->content.info), false);
  RecogLaunch(&g_wakeup_scene);
  StartTimer();
#if !MULTI_DIALOGUE_MODE
  _goto_wakeup();
#endif
  return rc;
}
```

若配置为一次唤醒，一次识别模式，执行完音量控制操作（_volume_setting_process）后，即主动调用_goto_wakeup 通知 wakeup session 使设备退回待唤醒状态。

再回到上述状态机的第五行{STATE_BROADCAST, ID(AUDIO_PLAY_END_EVENT), _broadcast__audio_play_end}，代表设备在执行完音量控制操作并进行相应结果的播报状态下收到播报结束的事件，将通过调用_broadcast__audio_play_end 进行处理：

```
static Result action_idle(void) {
  FsmSetState(g_setting_session->fsm, STATE_IDLE);
  return E_OK;
}

static Result _broadcast__audio_play_end(void *event) {
  LOGT(SETTING_SESSION_TAG, "action called");
#if MULTI_DIALOGUE_MODE
  RecogLaunch(&g_setting_scene);
  StartTimer();
#endif
  return action_idle();
}
```

若配置为一次唤醒，一次识别模式，因为前文_idle__vui_app_volume_setting 中已经主动通知设备退回待唤醒状态了，音量控制操作的结果也已经播报结束，代表本次设备控制的业务已经全部完成，故仅迁移当前状态机的状态为空闲，等待下一次已唤醒状态下的控制消息。

若配置为一次唤醒，多次识别模式，在音量控制操作的结果播报结束后，除了迁移当前状态机的状态为空闲外，还调用 RecogLaunch(&g_setting_scene)操作语音引擎进入了识别模式，此时无需唤醒即可再进行语音控制命令的识别。

5.3 播报逻辑开发

平台构建的本地播报文件存放在 applications/app/res/tone/目录下：

```
applications/app/res/tone/
├── 101.pcm
├── 102.pcm
├── 103.pcm
└── 104.pcm
```

（1）编号 100 以内命名的 pcm 为方案内部预留；

（2）编号 100+ 的 pcm 是根据平台的自定义答复配置生成的，对应关系可以从 application/app/res/tone/pcm_map.txt 中确认；

```
101.pcm 我在
102.pcm 我先走了
103.pcm 设备已打开
104.pcm 已经打开设备
105.pcm 设备已关闭
106.pcm 设备已停止
```

其中自定义命令词与答复语法格式前文已介绍，可见文档中的 2.2.2 配置终端能力。

（3）如果用户需要使用其它音频文件，请保证音频文件为 16000 采样率，16bit，单通道的 pcm 文件，同时建议编号从 200 开始，与平台配置生成的音频文件进行区分。

离线控制指令的解析可以在 applications/app/src/sessions/uni_setting_session.c 中修改：

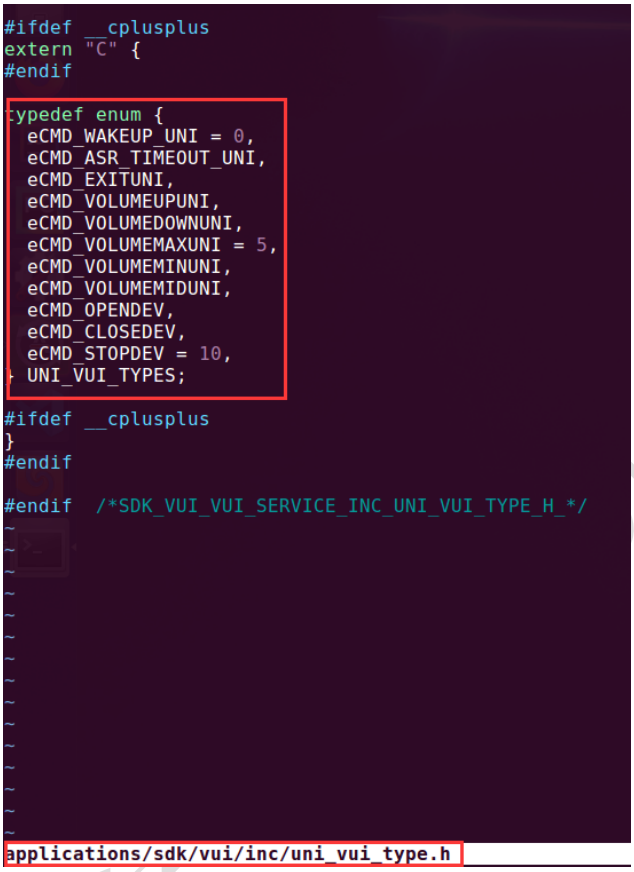
```
static Result _setting_central_control_process(cJSON *content) {
  Result ret = E_FAILED;
  char *deviceType = NULL;
  char *operands = NULL;
  char *text = NULL;
  char *pcm = NULL;
  if (NULL == content) {
    LOGW(SETTING_SESSION_TAG, "content not exist");
    return E_FAILED;
  }
  if (0 != JsonReadItemString(content,
                              "semantic.intent.operations[0].deviceType",
                              &deviceType) &&
      0 != JsonReadItemString(content,
                              "semantic.intent.operations[0].operands",
                              &operands)) {
    LOGW(SETTING_SESSION_TAG, "read deviceType or operands failed");
    goto L_END;
  }
  if (0 != JsonReadItemString(content,
                              "general.pcm",
                              &pcm)) {
    LOGW(SETTING_SESSION_TAG, "read pcm failed");
    goto L_END;
  }
  if (0 != JsonReadItemString(content,
                              "general.text",
                              &text)) {
    LOGW(SETTING_SESSION_TAG, "read text failed");
  }
  /*--- test code ---*/
  int cmd = -1;
  char fixed_pcm[] = "104";
  JsonReadItemInt(content, "semantic.intent.operations[0].cmd", &cmd);
  switch (cmd) {
    case eCMD_OPENDEV:
      ret = _central_control(content, fixed_pcm, text);
      goto L_END;
    default:
      break;
  }
  /*------*/
  ret = _central_control(content, pcm, text);
L_END:
```

收到“打开设备”指令时强制播放104.pcm

如红框中的示例代码，原接收到“打开设备”指令时是会随机播放 103 或者 104 号音频的，修改后的代码将通过 JsonReadItemInt(content, "semantic.intent.operations[0].cmd", &cmd) 获取此次控制指令的具体信息，当发现是“打开设备”时强制播放 104 号音频。（特殊的，如

“提高温度”指令，可以通过此方法添加针对边界温度的不同回复：已经达到最高温度的情况下再收到“提高温度”指令可回复“当前已是最高温度”）

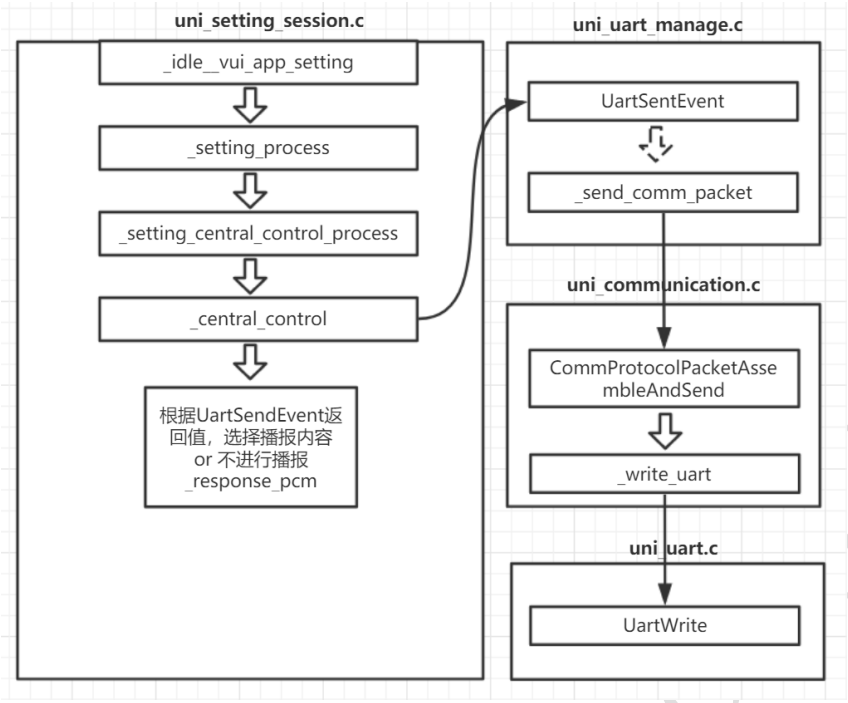
上例中“打开设备”对应的指令名 eCMD_OPENDEV 可以在 sdk/vui/inc/uni_vui_type.h 中确认：



5.4 串口协议开发

5.4.1 语音至串口调用流程

如 4.3.3 所述，离线控制指令的解析实现在 app/src/sessions/uni_setting_session.c 中，具体接口调用流程如下图：



5.4.2 通讯串口使用方式

默认通讯串口支持两种使用方式：

（1）异步方式，适合不需要 ack 机制的简单通讯场景，如：收到控制指令后即播报预期的处理结果，后台负责串口指令发送到主控板。这种方式可以简化主控板与语音板的协议实现，快速达到可落地状态。

（2）同步方式，适合需要 ack 机制的通讯场景，如：收到控制指令后即发送串口命令，且需要等到主控板的真正操作结果才进行相应播报。这种方式可以保证播报结果与实际结果完全一致。

在源码中通过修改 sdk/uart/src/uni_uart_manage.c 中的 UART_ASYNC 宏进行控制：

```
#if UNI_UART_MANAGE_DEBUG
#define UART_ASYNC          0
#define TAG                 "uart_manage"

typedef enum {
  UNI_SOUND = 0,
  UNI_CUSTOMER,
} CustomerType;

typedef struct {
  EventListHandle     event_list;
} UartManager;

typedef struct {
  CommCmd cmd;
  unsigned char payload[5];
} UartCmd;

applications/sdk/uart/src/uni_uart_manage.c
```

具体实现方法如下图，红框为异步方式，蓝色为同步方式：

```
Result UartSentEvent(void *content) {
  Event *event = NULL;
  EventContent event_content;
  uni_memset(&event_content, 0, sizeof(EventContent));
  event_content.info = cJSON_Duplicate((cJSON *)content, 1);
  event = EventCreate(EVENT_SEQUENCE_ID_DEFAULT, 0,
                      &event_content, _uart_event_free_handler);
  EventListAdd(g_uart_manager->event_list, event, EVENT_LIST_PRIORITY_MEDIUM);
  EventFree(event);
  return E_OK;
}
#else
Result UartSentEvent(void *content) {
  int cmd = -1;
  JsonReadItemInt((cJSON *)content, "semantic.intent.operations[0].cmd", &cmd);
  if (cmd != -1) {
    return _send_comm_packet(cmd);
  }
  return E_FAILED;
}
#endif

Result UartManageInit(void) {
  UartConfig config;
  g_uart_manager = uni_malloc(sizeof(UartManager));
  if (!g_uart_manager) {
    LOGE(TAG, "uart manage malloc failed.");
    return E_FAILED;
  }
  config.device = UNI_UART1;
  config.parity = UNI_PARITY_NONE;
  config.speed = UNI_B_9600;
  config.stop = UNI_ONE_STOP_BIT;
  config.data_bit = 8;
#if UART_ASYNC
  g_uart_manager->event_list = EventListCreate(_uart_event_process);
#endif
  UartInitialize(&config, CommProtocolReceiveUartData);
```

提供给app层的接口只负责把uart事件加入处理队列中

提供给app层的接口立即组装并发送串口协议包

创建后台uart事件处理队列

5.4.3 配置通讯参数

Uart 服务层代码所在路径为 sdk/uart/src/uni_uart_manage.c：

```
Result UartManageInit(void) {
  UartConfig config;
  g_uart_manager = uni_malloc(sizeof(UartManager));
  if (!g_uart_manager) {
    LOGE(TAG, "uart manage malloc failed.");
    return E_FAILED;
  }
  config.device = UNI_UART1;
  config.parity = UNI_PARITY_NONE;
  config.speed = UNI_B_9600;
  config.stop = UNI_ONE_STOP_BIT;
  config.data_bit = 8;
  g_uart_manager->event_list = EventListCreate(_uart_event_process);
  UartInitialize(&config, CommProtocolReceiveUartData);
  CommProtocolInit(UartWrite, _recv_comm_packet);
  return E_OK;
}

Result UartManageFinal(void) {
  CommProtocolFinal();
  UartFinalize();
  if (g_uart_manager) {
    if (g_uart_manager->event_list) {
      EventListDestroy(g_uart_manager->event_list);
    }
    uni_free(g_uart_manager);
applications/sdk/uart/src/uni_uart_manage.c [+]
```

如上图，默认使用 UART1，无校验，9600bps，1 停止位，8 数据位。可根据具体的需求进行修改，可选配置项在 utils/uart/inc/uni_uart.h 中：

```
/**
 * device type
 */
typedef enum {
  UNI_UART1,
  UNI_UART2,
  UNI_UART3,
} UartDeviceName;

/**
 * baud rate
 */
typedef enum {
  UNI_B_1200,
  UNI_B_2400,
  UNI_B_4800,
  UNI_B_9600,
  UNI_B_14400,
  UNI_B_19200,
  UNI_B_38400,
  UNI_B_57600,
  UNI_B_115200,
} UartSpeed;

/**
 *  parity check
 */
typedef enum {
  UNI_PARITY_ODD,
  UNI_PARITY_EVEN,
  UNI_PARITY_NONE,
  UNI_PARITY_MARK,
  UNI_PARITY_SPACE,
} UartParity;

/**
 * stop bit
 */
typedef enum {
  UNI_ONE_STOP_BIT,
  UNI_ONE_5_STOP_BIT,
  UNI_TWO_STOP_BIT,
} UartStop;

/**
```

5.4.4 异步方式

异步方式使用云知声标准串口协议：

```
/*-------------------------------------------------------------------------------*/
/*            layout of unisound communication app protocol                      */
/*-------------------------------------------------------------------------------*/
/*-1byte-|-1byte-|---2byte---|---2byte---|---2byte---|---2byte---|---N byte---*/
/*  SYNC |seq-num| customer  |  command  | checksum  |payload len|   payload  */
/*-------------------------------------------------------------------------------*/

/*--------------------------------ack frame--------------------------------------*/
/*  0xFF | seqNum|    0x0    |   0x0     |   crc16   |    0x0    |    NULL    */
/*-------------------------------------------------------------------------------*/
```

云知声标准协议数据格式如上图所示（详细协议定义请参照 utils/uart/doc/uart_protocol.pdf 文档）

首先配置 UART_ASYNC 宏为 1 以使用异步方式。

（1）sdk/uart/src/uni_uart_manage.c 源码中默认无需 payload，直接根据 cmd 数值填充到上述数据格式中进行发送。如识别到“设备停止”的语音（eCMD_STOPDEV，cmd 值为 0x0A），即可在通讯串口上接收到对应 command 为 0x0A 的数据协议包。

```
static Result _send_comm_packet(uni_u16 cmd) {
  int ret = -1;
  ret = CommProtocolPacketAssembleAndSend(UNI_CUSTOMER, cmd, NULL, 0, NULL);
  if (ret != 0) {
    return E_FAILED;
  }
  return E_OK;
}
```

无payload

sdk/vui/inc/uni_vui_type.h中定义的指令

无需ack

（2）如需添加 payload，亦或是重新组织语音指令对应串口 command 的数值，可灵活定义一个 map 表，示例如下（同 4.3.3，自定义的语音指令名可在 sdk/vui/inc/uni_vui_type.h 中确认）：

```
typedef struct {
  CommCmd cmd;
  unsigned char payload[5];
} UartCmd;

static UartCmd g_uart_cmd_map[] = {
  [eCMD_WAKEUP_UNI] = {0x02, {0, 0, 0, 0, 0}},
  [eCMD_EXITUNI]    = {0x07, {0, 1, 0, 0, 0}},
  [eCMD_OPENDEV]    = {0x0A, {0, 1, 2, 0, 0}},
  [eCMD_CLOSEDEV]   = {0x0A, {0, 1, 2, 3, 0}},
  [eCMD_STOPDEV]    = {0x0A, {0, 1, 2, 3, 4}},
};

static Result _send_comm_packet(uni_u16 cmd) {
  int ret = -1;
  ret = CommProtocolPacketAssembleAndSend(UNI_CUSTOMER,
                                          g_uart_cmd_map[cmd].cmd,
                                          g_uart_cmd_map[cmd].payload,
                                          sizeof(g_uart_cmd_map[cmd].payload),
                                          NULL);
  if (ret != 0) {
    return E_FAILED;
  }
  return E_OK;
}
```

1，定义转换后的数据结构，如定义需要转换cmd数值，且携带长度为5bytes的payload

2，针对需要的语音指令赋值map表：如赋值“停止设备”命令为0x0A，携带【0 1 2 3 4】 5 bytes payload

3，根据转换后的数据进行组包发送

5.4.5 同步方式

同步方式使用云知声标准串口协议：

```
/*-------------------------------------------------------------------------*/
/*            layout of unisound communication app protocol                */
/*-------------------------------------------------------------------------*/
/*-1byte-|-1byte-|---2byte---|---2byte---|---2byte---|---2byte---|---N byte---*/
/*  SYNC |seq-num| customer  |  command  | checksum  |payload len|   payload  */
/*-------------------------------------------------------------------------*/

/*-------------------------------ack frame---------------------------------*/
/*  0xFF | seqNum|    0x0    |   0x0     |   crc16   |    0x0    |    NULL    */
/*-------------------------------------------------------------------------*/
```

云知声标准协议数据格式如上图所示（详细协议定义请参照 utils/uart/doc/uart_protocol.pdf 文档）

首先配置 UART_ASYNC 宏为 0 以使用同步方式。

（1）配置 CommAttribute 属性，添加 ack 需求，以及等待 ack 的超时时间：

```
typedef struct {
  CommCmd cmd;
  unsigned char payload[5];
} UartCmd;

static UartCmd g_uart_cmd_map[] = {
  [eCMD_WAKEUP_UNI] = {0x02, {0, 0, 0, 0, 0}},
  [eCMD_EXITUNI]    = {0x07, {0, 1, 0, 0, 0}},
  [eCMD_OPENDEV]    = {0x0A, {0, 1, 2, 0, 0}},
  [eCMD_CLOSEDEV]   = {0x0A, {0, 1, 2, 3, 0}},
  [eCMD_STOPDEV]    = {0x0A, {0, 1, 2, 3, 4}},
};

static Result _send_comm_packet(uni_u16 cmd) {
  int ret = -1;
  unsigned char try_times = g_uart_cmd_map[cmd].try_times;
  CommAttribute attr;
  attr.need_acked = 1;
  attr.timeout_msec = 50;
  ret = CommProtocolPacketAssembleAndSend(UNI_CUSTOMER,
                                          g_uart_cmd_map[cmd].cmd,
                                          g_uart_cmd_map[cmd].payload,
                                          sizeof(g_uart_cmd_map[cmd].payload),
                                          &attr);

  if (ret != 0) {
    return E_FAILED;
  }
  return E_OK;
}
```

ack校验打开
50ms超时

根据 4.3.4.1 的接口调用图，查看 app/src/sessions/uni_setting_session.c 中语音指令解析处理部分的实现，根据返回值决定是否播报或者播报内容：

```
static Result _central_control(cJSON *content, char *pcm, char *reply) {
  if (E_OK != UartSentEvent((void *)content))
  {
    FsmSetState(g_setting_session->fsm, STATE_IDLE);
    LOGE(SETTING_SESSION_TAG, "uart send failed, don't response");
    return E_OK;
  }
  if (E_OK == _response_pcm(content, _get_random_pcm(pcm), reply)) {
    RecogLaunch(&g_wakeup_scene);
    FsmSetState(g_setting_session->fsm, STATE_BROADCAST);
    return E_HOLD;
  } else {
    FsmSetState(g_setting_session->fsm, STATE_IDLE);
    LOGE(SETTING_SESSION_TAG, "response pcm failed");
    return E_OK;
  }
}
```

串口发送失败即不播报操作结果

（2）如需添加重发机制，可在串口指令的 map 结构中定义“尝试次数”属性，只有在尝试次数全部失败后才会通知调用方，发送失败：

```
typedef struct {
  CommCmd cmd;
  unsigned char payload[5];
  unsigned char try_times;
} UartCmd;

static UartCmd g_uart_cmd_map[] = {
  [eCMD_WAKEUP_UNI] = {0x02, {0, 0, 0, 0, 0}, 1},
  [eCMD_EXITUNI]    = {0x07, {0, 1, 0, 0, 0}, 1},
  [eCMD_OPENDEV]    = {0x0A, {0, 1, 2, 0, 0}, 3},
  [eCMD_CLOSEDEV]   = {0x0A, {0, 1, 2, 3, 0}, 3},
  [eCMD_STOPDEV]    = {0x0A, {0, 1, 2, 3, 4}, 3},
};

static Result _send_comm_packet(uni_u16 cmd) {
  int ret = -1;
  unsigned char try_times = g_uart_cmd_map[cmd].try_times;
  CommAttribute attr;
  attr.need_acked = 1;
  attr.timeout_msec = 50;
  do {
    ret = CommProtocolPacketAssembleAndSend(UNI_CUSTOMER,
                                            g_uart_cmd_map[cmd].cmd,
                                            g_uart_cmd_map[cmd].payload,
                                            sizeof(g_uart_cmd_map[cmd].payload),
                                            &attr);
    if (ret == 0) {
      return E_OK;
    }
  } while (try_times--);
  return E_FAILED;
}
```

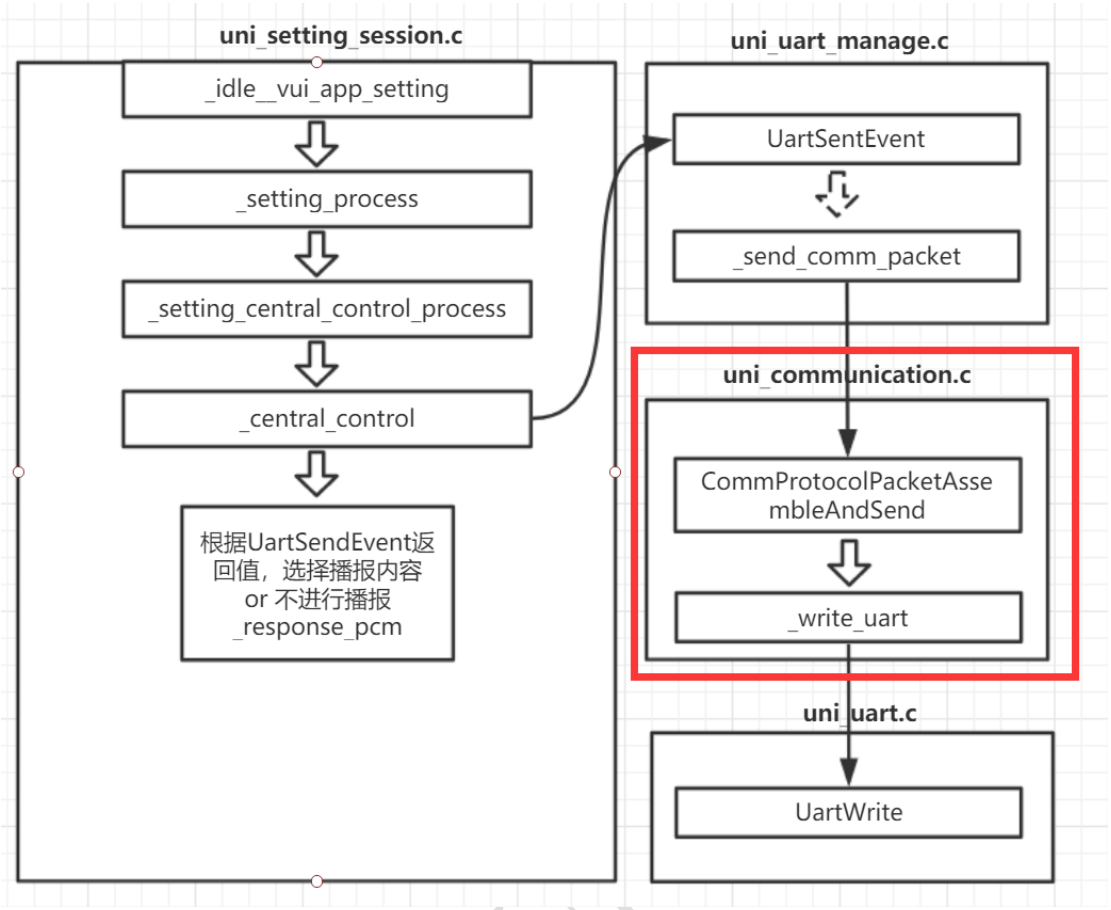
1，添加“尝试次数“属性

2，针对不同指令，赋值不同的尝试次数

3，根据指令的尝试次数进行出错重发

5.4.6 修改适配自有串口协议

当需要适配自有串口协议时，需要重新实现的是如下红框中的文件（uni_communication.c）：



以下图协议为例，介绍下需要实现的具体接口：

```
/*-------------------------------*/
/*   communication app protocol  */
/*-------------------------------*/
/*-1byte-|-1byte-|-5byte-|-1byte--*/
/*  SYNC  |  cmd  |payload|checksum*/
/*-------------------------------*/
```

（1）协议数据发送：

① 修改 uni_communication.h 中类型定义：

```
typedef unsigned char       CommSync;
typedef unsigned char       CommCmd;
typedef unsigned char       CommChecksum;
typedef int                 (*CommWriteHandler)(char *buf, int len);
```

② 修改 uni_communication.c 中串口协议结构：

```
typedef struct {
  CommSync      sync;       /* must be UNI_COMM_SYNC_VALUE */
  CommCmd       cmd;        /* command type, such as power on, power off etc */
  unsigned char payload[5]; /* the payload */
  CommChecksum  checksum;
} PACKED CommProtocolPacket;
```

③ （可选）修改提供给 uart 中间层的接口 CommProtocolPacketAssembleAndSend 并同步修改调用方：

```
int CommProtocolPacketAssembleAndSend(CommCmd cmd,
    如有需要，可根据协议定义精简入参定义 unsigned char *payload,
                                          CommPayloadLen payload_len) {
  int ret = 0;
  CommProtocolPacket *packet = uni_malloc(sizeof(CommProtocolPacket));
  if (NULL == packet) {
    LOGE(UART_COMM_TAG, "alloc memory failed");
    return -1;
  }
  _assmeble_packet(cmd, payload, payload_len, packet);
  ret = _write_uart(packet);
  uni_free(packet);
  return ret;
}
```

④ 协议组包实现：

```
static void _assmeble_packet(CommCmd cmd,
                             unsigned char *payload,
                             CommPayloadLen payload_len,
                             CommProtocolPacket *packet) {
  _sync_set(packet);
  _cmd_set(packet, cmd);
  _payload_set(packet, payload, payload_len);
  _checksum_calc(packet);
}
```

本例中协议可以抽象为 4 个步骤：

a 设置同步头：

```
static int _sync_set(CommProtocolPacket *packet) {
  packet->sync = UNI_COMM_SYNC_VALUE;
  return 0;
}
```

b 设置命令：

```
static int _cmd_set(CommProtocolPacket *packet, CommCmd cmd) {
  packet->cmd = cmd;
  return 0;
}
```

c 复制 payload：

```
static int _payload_set(CommProtocolPacket *packet,
                        unsigned char *buf, CommPayloadLen len) {
  if (NULL != packet && NULL != buf &&
      0 < len && len <= sizeof(packet->payload)) {
    memcpy(packet->payload, buf, len);
    return 0;
  }
  return -1;
}
```

d 计算 checksum 并附加到协议包中，本例中采用累加求和再取低八位的方式：

```
static int _checksum_calc(CommProtocolPacket *packet) {
  unsigned int i, checksum = 0;
  unsigned char *p = (unsigned char *)packet;
  for (i = 0; i < _packet_len_get(packet) - 1; i++) {
    checksum += *p++;
  }
  packet->checksum = (CommChecksum)(checksum & 0xFF);
  return 0;
}
```

⑤ 至此，已实现协议报文的主动发送。

（2）协议数据接受：

① 修改 uni_communication.c 中上报给上层的串口协议结构，如本例中的 sync，checksum 字段对上层无意义，直接剥离：

```
typedef struct {
  CommCmd        cmd; /* ctrl cmd such as power_on, power_off */
  char           payload[5];  /* parameter of command */
} PACKED CommPacket;
```

② 实现_protocol_buffer_generate_byte_by_byte 对接收到的数据进行协议解析，本例中，需要先收到 sync 头，然后按照协议定义的长度（7）进行整个协议包的组装，组装完成后调用_one_protocol_frame_process 进行解析：

```
static void _protocol_buffer_generate_byte_by_byte(char recv_c) {
  static int index = 0;
  /* check timestamp to reset status when physical error */
  if (_bytes_coming_speed_too_slow(index)) {
    LOGT(UART_COMM_TAG, "reset protocol buffer automatically[%d]", index);
    _reset_protocol_buffer_status(&index);
  }
  /* check protocol receive data length */
  if (_is_protocol_buffer_overflow(index)) {
    /* drop remain bytes of current frame */
    LOGW(UART_COMM_TAG, "recv invalid frame, buffer overflow");
    _reset_protocol_buffer_status(&index);
    return;
  }
  /* get frame header sync byte */
  if (LAYOUT_SYNC_IDX == index) {
    if (UNI_COMM_SYNC_VALUE == (unsigned char)recv_c) {
      g_comm_protocol_business.protocol_buffer[index++] = recv_c;
    } else {
      LOGW(UART_COMM_TAG, "nonstandord sync byte, please check");
    }
    return;
  }
  /* generate protocol */
  if (index < sizeof(CommProtocolPacket)) {
    g_comm_protocol_business.protocol_buffer[index++] = recv_c;
  }
  /* callback protocol buffer */
  if (sizeof(CommProtocolPacket) == index) {
    LOGT(UART_COMM_TAG, "assemble new frame, now callback");
    _one_protocol_frame_process(g_comm_protocol_business.protocol_buffer);
    _reset_protocol_buffer_status(&index);
  }
}
```

1，校错（可省略）

2，byte by byte 接受数据

③ 实现协议解析接口_one_protocol_frame_process（包括 checksum 校验接口等）：

```
static void _one_protocol_frame_process(char *protocol_buffer) {
  /* when application not register hook, ignore all*/
  if (NULL == g_comm_protocol_business.on_recv_frame) {
    LOGW(UART_COMM_TAG, "donot register recv_frame hook");
    return;
  }
  /* ack frame donnot notify application, ignore it now */
  if (_is_acked_packet(protocol_buffer)) {
    LOGT(UART_COMM_TAG, "recv ack frame");
    _set_acked_sync_flag();
    return;
  }
  /* disassemble protocol buffer */
  CommPacket *packet = _packet_disassemble(protocol_buffer);
  if (NULL == packet) {
    LOGW(UART_COMM_TAG, "disassemble packet failed");
    return;
  }
  /* notify application when not ack frame */
  g_comm_protocol_business.on_recv_frame(packet);
  uni_free(packet);
}
```

1，检测是否是ack

2，生成上报上层的数据包

3，通知上层

本例中是通过定义好的 ack cmd 数值（0xff）来检测是否为 ack 包的：

```
static uni_bool _is_acked_packet(char *protocol_buffer) {
  CommProtocolPacket *protocol_packet = (CommProtocolPacket *)protocol_buffer;
  return (protocol_packet->cmd == 0xFF);
}
```

上报报文的生成如下：

```
static CommPacket* _packet_disassemble(char *buf) {
  CommPacket *packet;
  CommProtocolPacket *protocol_packet = (CommProtocolPacket *)buf;
  if (!_checksum_valid(protocol_packet)) {
    LOGE(UART_COMM_TAG, "checksum failed");
    return NULL;
  }
  packet = uni_malloc(sizeof(CommPacket));
  if (NULL == packet) {
    LOGE(UART_COMM_TAG, "alloc memory failed");
    return NULL;
  }
  packet->cmd = protocol_packet->cmd;
  memcpy(packet->payload, _payload_get(protocol_packet),
         _payload_len_get(protocol_packet));
  return packet;
}
```

④ uni_uart_manage.c 中实现 uart 上报报文的处理：

on_recv_frame 是在 UartManageInit 时注册的：

```
Result UartManageInit(void) {
  UartConfig config;
  g_uart_manager = uni_malloc(sizeof(UartManager));
  if (!g_uart_manager) {
    LOGE(TAG, "uart manage malloc failed.");
    return E_FAILED;
  }
  uni_memset(g_uart_manager, 0, sizeof(UartManager));
  config.device = UNI_UART1;
  config.parity = UNI_PARITY_NONE;
  config.speed = UNI_B_9600;
  config.stop = UNI_ONE_STOP_BIT;
  config.data_bit = 8;
#if UART_ASYNC
  g_uart_manager->event_list = EventListCreate(_uart_event_process);
#endif
  UartInitialize(&config, CommProtocolReceiveUartData);
  CommProtocolInit(UartWrite, _recv_comm_packet);
  _set_send_timestamp(uni_get_clock_time_ms());
  g_uart_manager->heartbeat_timer = TimerStart(HEARTBEAT_MSEC,
                                               TIMER_TYPE_PERIODICAL,
                                               _heartbeat_timer_handle,
                                               NULL);

  return E_OK;
}
```

本例实现了主控板发生状态变更时主动通知语音板进行相应播报的逻辑：

```
static void _recv_comm_packet(CommPacket *packet) {
  LOGT(TAG, "recv one packet cmd:%02x payload: %02x %02x %02x %02x %02x",
       packet->cmd, packet->payload[0], packet->payload[1],
       packet->payload[2], packet->payload[3], packet->payload[4]);

  int cmd_size = sizeof(g_uart_cmd_map) / sizeof(g_uart_cmd_map[0]);
  for (int i = 0; i < cmd_size; i++) {
    if (g_uart_cmd_map[i].cmd == packet->cmd) {
      cJSON *nlu = NluGetLasrByIndex(i);
      if (NULL == nlu) {
        break;
      }
      cJSON *uart_broadcast = cJSON_CreateNumber(packet->payload[0]);
      cJSON_AddItemToObject(nlu, "uart_broadcast", uart_broadcast);
      EventContent content = {0};
      content.info = nlu;
      content.vui_session_id = EVENT_VUI_SESSION_ID_PRIVILEGE;
      Event *event = EventCreate(EVENT_SEQUENCE_ID_DEFAULT,
                                 ID(VUI_APP_UART_CTRL_EVENT),
                                 &content, _uart_event_free_handler);
      EventRouteProcess(event);
      EventFree(event);
      break;
    }
  }
}
```

1. 根据接收到的cmd，生成对应的意图（直接复用语音意图）

2. 发送uart ctrl 事件

其中 VUI_APP_UART_CTRL_EVENT 需要定义到 hal/inc/uni_types.h 的 UNI_EVENT_TYPE 枚举中：

```
  VUI_LOCAL_ASR_TIMEOUT_EVENT,
  VUI_CAPTURE_TIMEOUT_EVENT,
  VUI_APP_NOTE_EVENT,
  WATCH_DOG_FEED_EVENT,
  APP_TIMERUP_EXPIRE_EVENT,
  APP_MUSIC_TIMER_EXPIRE_EVENT,
  APP_GOTO_WAKEUP_EVENT,
  APP_BT_NOTIFY_EVENT,
  VUI_LOCAL_ASR_FAIL_EVENT,
  VUI_APP_BOOKMARK_EVENT,
  OTA_NOTIFY_EVENT,
  VUI_APP_FAV_LIST_EVENT,
  APP_RESET_NOTIFY_EVENT,
  APP_AUTOTEST_NOTIFY_EVENT,
  CLOUD_UNBOUND_EVENT,
  APP_SLEEP_MODE_EVENT,
  VUI_APP_UART_CTRL_EVENT
} UNI_EVENT_TYPE;添加uart ctrl事件到末尾
```

并在 app/src/uni_session_manage.c 中初始化 SessionManageInit 时注册上：

```
Result SessionManageInit(void) {
  uni_s32 events[] = {ID(AUDIO_PLAY_END_EVENT),
                      ID(AUDIO_MEDIA_PLAY_END_EVENT),
                      ID(SESSION_MGR_REVIEW_EVENT),
                      ID(TIMER_MGR_ALARM_TIMEUP_EVENT),
                      ID(NETWORK_STATUS_NOTIFY_EVENT),
                      ID(CLOUD_MP_EVENT),
                      ID(CLOUD_PUSH_MUSIC_LIST_EVENT),
                      ID(CLOUD_PULL_MUSIC_LIST_EVENT),
                      ID(VUI_APP_ALARM_EVENT),
                      ID(VUI_APP_CALENDAR_EVENT),
                      ID(VUI_APP_CHAT_EVENT),
                      ID(VUI_APP_MEDIA_PLAY_EVENT),
                      ID(VUI_APP_BT_MEDIA_PLAY_EVENT),
                      ID(VUI_APP_MUSIC_EVENT),
                      ID(VUI_APP_SETTING_EVENT),
                      ID(VUI_APP_SLEEP_EVENT),
                      ID(VUI_APP_TRANSLATION_EVENT),
                      ID(VUI_APP_VOLUME_SETTING_EVENT),
                      ID(VUI_APP_WAKEUP_EVENT),
                      ID(VUI_APP_MIC_MUTE_EVENT),
                      ID(VUI_APP_MIC_UNMUTE_EVENT),
                      ID(VUI_APP_STOCK_EVENT),
                      ID(VUI_APP_WEATHER_EVENT),
                      ID(VUI_CAPTURE_TIMEOUT_EVENT),
                      ID(VUI_APP_NOTE_EVENT),
                      ID(WATCH_DOG_FEED_EVENT),
                      ID(APP_TIMERUP_EXPIRE_EVENT),
                      ID(APP_MUSIC_TIMER_EXPIRE_EVENT),
                      ID(APP_GOTO_WAKEUP_EVENT),
                      ID(APP_BT_NOTIFY_EVENT),
                      ID(VUI_APP_BOOKMARK_EVENT),
                      ID(OTA_NOTIFY_EVENT),
                      ID(VUI_APP_FAV_LIST_EVENT),
                      ID(APP_RESET_NOTIFY_EVENT),
                      ID(APP_AUTOTEST_NOTIFY_EVENT),
                      ID(CLOUD_UNBOUND_EVENT),
                      ID(APP_SLEEP_MODE_EVENT),
                      ID(VUI_APP_UART_CTRL_EVENT)};
  list_init(&g_session_manager.session_list);
  g_session_manager.active_session = NULL;
  BbWrite(BB_KEY_EVENT_ID, INVALID_EVENT_ID_EVENT_INVALID);
  g_session_manager.event_list = EventListCreate(_event_callback);
  EventRouteSubscribe(_session_manage_schedule,
                      sizeof(events) / sizeof(events[0]), events);
  //g_session_manager.black_list = BlackListCreate(64);
  return E_OK;
```

接下来就是按照 4.3.2 中场景业务逻辑的实现说明进行 VUI_APP_UART_CTRL_EVENT 的处理，实现在“空闲”和“播报”状态下针对“中控板状态变更”事件的处理：

```
static int _session_transition_init(void) {
  MicroFsmTransition session_transition[] = {
    {STATE_IDLE,      ID(VUI_APP_VOLUME_SETTING_EVENT), _idle__vui_app_volume_setting},
    {STATE_IDLE,      ID(VUI_APP_SETTING_EVENT),        _idle__vui_app_setting},
    {STATE_IDLE,      ID(VUI_APP_UART_CTRL_EVENT),      _idle__vui_app_uart_ctrl},
    {STATE_BROADCAST, ID(VUI_APP_UART_CTRL_EVENT),      _broadcast__vui_app_uart_ctrl},
    {STATE_BROADCAST, ID(VUI_APP_VOLUME_SETTING_EVENT), _broadcast__vui_app_volume_setting},
    {STATE_BROADCAST, ID(VUI_APP_SETTING_EVENT),        _broadcast__vui_app_setting},
    {STATE_BROADCAST, ID(AUDIO_PLAY_END_EVENT),         _broadcast__audio_play_end},
    {STATE_BROADCAST, ID(COMMON_STOP_EVENT),            _broadcast__common_stop},
  };
  g_setting_session_transition = uni_malloc(sizeof(session_transition));
  if (NULL == g_setting_session_transition) {
    LOGE(SETTING_SESSION_TAG, "malloc failed !");
    return 0;
  }
```

```
static Result _do_uart_broadcast(void *event) {
  cJSON *root = (cJSON *)(((Event *)event)->content.info);
  int broadcast_status;
  char *pcm = NULL;
  int ret = JsonReadItemInt(root, "uart_broadcast", &broadcast_status);
  if (0 != ret) {
    LOGE(SETTING_SESSION_TAG, "read broadcast_status failed");
    FsmSetState(g_setting_session->fsm, STATE_IDLE);
    return E_OK;
  }
  if (0 != JsonReadItemString(root,
                              "general.pcm",
                              &pcm)) {
    LOGW(SETTING_SESSION_TAG, "read pcm failed");
    return E_OK;
  }
  _response_pcm(root, _get_random_pcm(pcm), NULL);
  uni_free(pcm);
  FsmSetState(g_setting_session->fsm, STATE_BROADCAST);
  return E_HOLD;
}

static Result _idle__vui_app_uart_ctrl(void *event) {
  return _do_uart_broadcast(event);
}

static Result _broadcast__vui_app_uart_ctrl(void *event) {
  _stop_player();
  return _do_uart_broadcast(event);
}
```

⑥ 至此，已实现协议报文的接受处理。

5.5 外设开发

5.5.1 API 使用说明

5.5.1.1 GPIO

(1) GPIO 函数介绍

具体实现详见 applications/hal/src /uni_hal_gpio.c

函数	void uni_hal_gpio_irq_init(void)
参数	void
描述	注册 GPIO 控制器的中断 handler，中断模式下，此 GPIO 引脚会默认为输入。
返回	无返回值

函数	int uni_hal_gpio_irq_cfg(int gpionum, gpio_irq_cb gpioN_irq_cb)
参数	gpionum: 需要配置的 GPIO number irq_cb gpioN_irq_cb: 当前配置的 GPIO 信号的中断回调函数，中断模式下，此 GPIO 引脚会默认为输入。
描述	注册 GPIO 信号的中断回调函数
返回	-1: GPIO 中断回调函数注册失败 0: GPIO 中断回调函数注册成功

函数	void uni_hal_gpio_set_value(int gpionum, int value)
参数	gpionum: 需要配置的 GPIO number

	value: GPIO 引脚要设置的电平，1=高电平， 0 =低电平。
描述	设置 GPIO 引脚的电平状态，引脚状态会默认配置为输出。
返回	无

函数	void uni_hal_gpio_get_value(int gpionum, int *value)
参数	gpionum: 需要配置的 GPIO number value: 当前 GPIO 引脚的电平状态返回值的指针。
描述	获取 GPIO 引脚的电平状态，1=高电平， 0 =低电平，引脚状态会默认配置为输入。
返回	无

函数	void uni_hal_gpio_pinmux_set(u32 gpio)
参数	gpio: 具体参考 uni_hal_gpio.h
描述	给需要使用的 gpio 配置 pinmux。
返回	无

(2) GPIO 使用范例

注：目前用户能够使用的 GPIO 如下：

GPIO1、GPIO2、GPIO3、GPIO4、GPIO7、GPIO8、GPIO11、GPIO14、GPIO15、GPIO16、GPIO17、GPIO18、GPIO19、GPIO20、GPIO21、GPIO22、GPIO23、GPIO26、GPIO29，

其中 GPIO3、GPIO17、GPIO18 不需要配置 pinmux，可以直接使用，其他管脚需要先配置 pinmux 再使用。**如果使用了其它的 GPIO，请联系我们。**

使用 gpio 的流程如下：

1. 使用的 GPIO 是否需要配置 pinmux，需要配置，跳到步骤 2，不需要跳到步骤 3；
2. 使用 uni_hal_gpio_pinmux_set 配置管脚 pinmux
3. 根据实际场景调用其他 api，实现相应功能

下面是具体 API 的使用方法举例：

举例 1：GPIO 引脚输出电平修改

a.电平拉低

```
uni_hal_gpio_set_value(3, 0); //GPIO3 引脚输出电平拉低
```

b.电平拉高

```
uni_hal_gpio_set_value(3, 1); //GPIO3 引脚输出电平拉高
```

举例 2：GPIO 引脚输入中断模式配置

```
void gpio_int_callback(void *arg){
        fLib_printf("gpio intterupt fired!\n");
}
uni_hal_gpio_irq_init(void); //初始化 GPIO 中断模式，注册中断 handler，默认高电平触发
uni_hal_gpio_irq_cfg(3,  gpio_int_callback);//注册 GPIO3 的中断回调函数，当 GPIO3 输入电平为高时，会触发中断，打印“gpio intterupt fired!”。
```

举例 3：GPIO 引脚输入状态获取

```
    int value;
```

```
uni_hal_gpio_get_value(3, &value)；//获取 GPIO3 引脚的输入电平状态。
```

举例 4：pinmux 使用

例如要使用 GPIO1 、GPIO2、 GPIO20 ：

```
u32 gpio = GPIO1_PCM_SYNC | GPIO2_I2S1_BCLK | GPIO20_SPI1_RX;
uni_hal_gpio_pinmux_set(gpio);
```

配置好 pinmux，就可以使用其他 api 控制相应 GPIO 了。

注意：从 uni_hal_gpio.h 定义可知有些 gpio 管脚有两个定义，这个要看实际电路上用的哪个脚，就配置哪个脚.比如硬件上用 GPIO1_PCM_SYNC 复用成 GPIO1，就用 GPIO1_PCM_SYNC 这个参数； 硬件上用 GPIO1_I2S1_LRCK 复用成 GPIO1，就用 GPIO1_I2S1_LRCK 这个参数.

5.5.1.2 I2C

(1) I2C 函数介绍

具体实现详见 applications/hal/src /uni_hal_i2c.c

注意：目前 I2C 在 DSP 已经使用，此接口目前是学习来用，如果有实际的使用需求，请联系开发人员

函数	int uni_i2c_init(i2c_clock clock)
参数	typedef enum { I2C_CLOCK_100K, I2C_CLOCK_400K }i2c_clock;
描述	i2c 初始化
返回	0：i2c 初始化成功 -1：i2c 初始化失败

函数	int uni_i2c_write(u8 slave_addr, u8 * buf, u32 length)
参数	u8 slave_addr: i2c 设备地址，一个 i2c 总线上可以挂多个设备 u8 * buf: 需要从 i2c 写的数据 U32 length：传输长度
描述	I2c 写数据
返回	0：传输成功 -1：传输失败

函数	int uni_i2c_read(u8 slave_addr, u8* buf, u32 length);
参数	u8 slave_addr: i2c 设备地址，一个 i2c 总线上可以挂多个设备 u8 * buf: 需要从 i2c 读出来的数据 U32 length：传输长度
描述	I2c 数据读取,如果读寄存器值,buf[0] is reg addr,否则 buf[0] is 0
返回	0：读取成功 -1：读取失败

(2) GPIO 使用范例

举例：用户使用的 codec 为 ES8218，需要通过 I2C 配置 codec。

Codec I2C 地址为 0x10

```
#define       ES8218_CHIP_ADR_BASE                    0x10
```

第一步，初始化 I2C

```
int ret;
u8 buff[2];
ret = uni_i2c_init(I2C_CLOCK_400K);
if(0 != ret){
   return ret;
}
```

第二步，写 I2C 数据

```
buff[0] = addr;
buff[1] = data;
ret = uni_i2c_write(ES8218_CHIP_ADR_BASE, buff, 2);
if(0 != ret){
   return ret;
}
return 0;
```

5.5.1.3 ADC

(1) ADC 函数介绍

具体实现详见 applications/hal/src /uni_hal_adc.c

函数	int uni_adc_init(uint8_t channel, uint8_t overlow1 ,uint16_t threshold1, uint8_t overlow2, uint16_t threshold2)
参数	uint8_t channel: adc 按键使用的 channel
描述	uint8_t overlow1 ,uint16_t threshold1, uint8_t overlow2, uint16_tthreshold2：设置 adc 电压检测范围。 overlow：1:超过 threshold 触发中断；0：低于 threshold 触发中断。
返回	0：adc 按键 初始化成功 -1：adc 按键 初始化失败

函数	int uni_adc_register_callback(adc_callback func)
参数	adc_callback func:adc 按键中断的回调函数
描述	adc 按键中断产生后，会调用用户注册的按键中断回调函数。
返回	0：adc 按键回调函数注册成功 -1：adc 按键回调函数注册失败

(2) ADC 使用范例

举例：用户通过电阻串联的方式检测不同的电压，从而检测按下的是哪个按键。所有的按键

按下的情况下，电压都会小于 2.6V。

（1）将 adc 的阈值（threshold）设置为 2.6V。

threshold 计算方式：

threshold = value / 1024 * 3.3V;

（2）定义 ADC 检测 callback 函数，参考函数如下：

```
int adc_callback_func(int value)
{
        //用户根据回调上来的当前电压值，做响应处理
}
```

（3）初始化 adc，并且将 adc 事件触发阈值设置为低于 2.6v 触发，参考代码如下：

```
int ret;
//配置检测低于 2.6v 的电压
ret = uni_adc_init(ADC_CHANNEL1, ADC_LOW, 800, ADC_LOW, 800);
if(0 != ret){
    return ret;
}
```

（4）用户注册自己的 adc 回调函数。当有低于设置阈值的电压产生后，会调用用户的回调函数，将阈值回调上来给用户，参考代码如下：

```
ret = uni_adc_register callback(adc_callback_func);
if(0 != ret){
    return ret;
}
return 0;
```

5.5.1.4 PWM

(1) PWM 函数介绍

具体实现详见 applications/hal/src /uni_hal_pwm.c

函数	int uni_hal_pwm_start(u8 pwm_num, u32 freq_hz, u32 duty_cycle)
参数	**pwm_num**: timer num from 1~8 **freq_hz**: frequency of output waveform, must be < pclk = 45.056M **duty_cycle**: should be within 100, i.e. if 50% duty cycle wanted, then duty_cycle = 50
描述	成功后，周期性输出相应占空比的波形
返回	0：pwm start 成功 -1：pwm start 失败
函数	void uni_hal_pwm_disable(u8 pwm_num)
参数	**pwm_num**: timer num from 1~8，
描述	Disable 一个已经工作的 pwm 通道
返回	void

(2) PWM 使用范例

举例：使用 pwm6 输出一个占空比 50%，freq：10k 的波形

uni_hal_pwm_start(6,10000,50);

如果想要动态的改成 60%占空比的波形，只需要再次调用此接口即可，即：
uni_hal_pwm_start(6,10000,60);

5.6 Watchdog 开发

在系统运行期间，出现程序指针错误，不在程序区，取出错误的程序指令等，都有可能会陷入死循环，程序的正常运行被打断，系统无法继续正常工作，导致整个系统的陷入停滞状态，发生不可预料的后果。

看门狗，又叫 watchdog，从本质上来说就是一个定时器电路，一般有一个输入和一个输出，其中输入叫做喂狗，输出一般连接到芯片的复位端。 看门狗的功能是定期的查看芯片内部的情况，一旦发生错误就向芯片发出重启信号。

注意：watchdog 应用代码已经开启，用户不需要再次使能 watchdog，用户可以用过接口了解 watchdog 的使用

具体实现详见 applications/hal/src /uni_hal_watchdog.c

函数	int uni_watchdog_start(u32 time)
参数	u32 time: time 时间之内，没有喂狗，系统重启。Time 的单位为 s
描述	打开 watchdog，并且设置 watchdog 重启时间
返回	0：watchdog 启动成功 -1：watchdog 启动失败

函数	int uni_watchdog_feed(void)
参数	无
描述	喂狗，需要在 watchdog 重启之前喂狗，否则系统重启
返回	0：watchdog 喂狗成功 -1：watchdog 喂狗失败

函数	int uni_watchdog_stop(void)
参数	无
描述	关闭 watchdog 功能
返回	0：watchdog 关闭成功 -1：watchdog 关闭失败

举例：使用看门狗，10s 没有喂狗重启

第一步，初始化看门狗为 10s 重启一次

```
int ret;
ret = uni_watchdog_start(10);
if(0 != ret){
    return ret;
}
return 0;
```

第二步，起一个喂狗的任务，每 9s 喂狗一次。

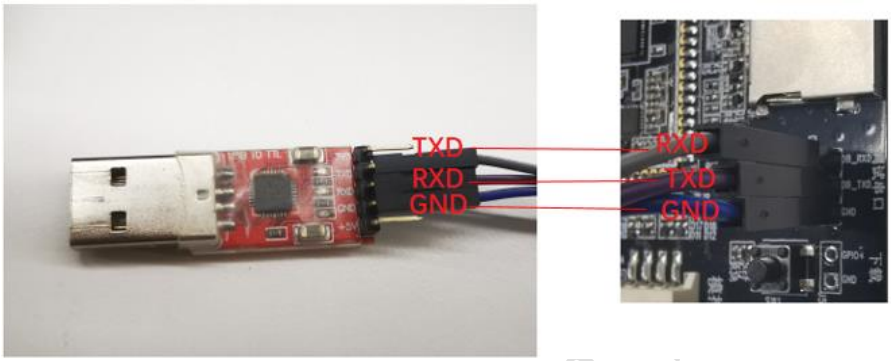
```
//喂狗
```

```
while(1){
    uni_sleep(9);
    uni_watchdog_feed();
}
```

5.7 调试验证

（1）下载研发版本后即可通过 log 信息确认设备运行状态。

调试串口为章节 3 中的下载串口，连接 PC 后使用常见串口工具（如 putty、xshell、secureCRT 等），配置波特率 (115200bps)、数据位 (8)、停止位 (1)、校验 (None)即可。

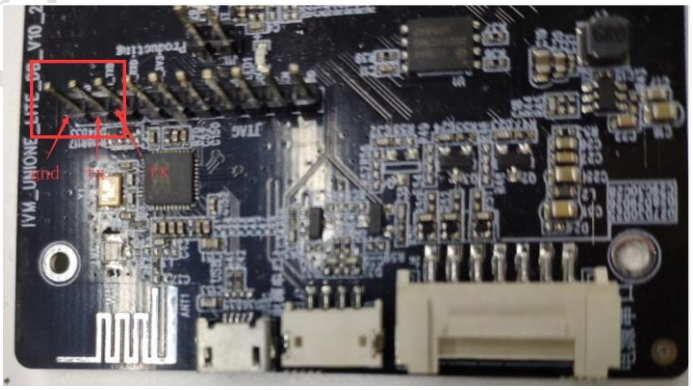


设备上电后，可在串口工具中观察到如下启动信息：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.0.2 build Jan  7 2020
 2006 - 2019 Copyright by rt-thread team
fLib_ConnectInt:handler=7ff4f065,IntNum=13
fLib_ConnectInt:handler=7ff4ef39,IntNum=18
fLib_ConnectInt:handler=7ff4b51d,IntNum=61

*|**||*|\*||**||**/'\   >>Unione Version : v0.2.0--2020010710
*\_//**|*\||**||**\_/   >>Build Data      : 2020010712
```

(2) 与主控板通讯的串口默认为 UART1，配置为波特率 (9600bps)、数据位 (8)、停止位 (1)、校验 (None)，直接与主控板 gnd，tx，rx 对接即可（无需交叉）：



调试时可以通过 uart 转 usb 转接器与 PC 进行连接，使用常见串口工具进行收发验证，这里推荐使用 XCOM 这款免费串口工具。

四、常见问题

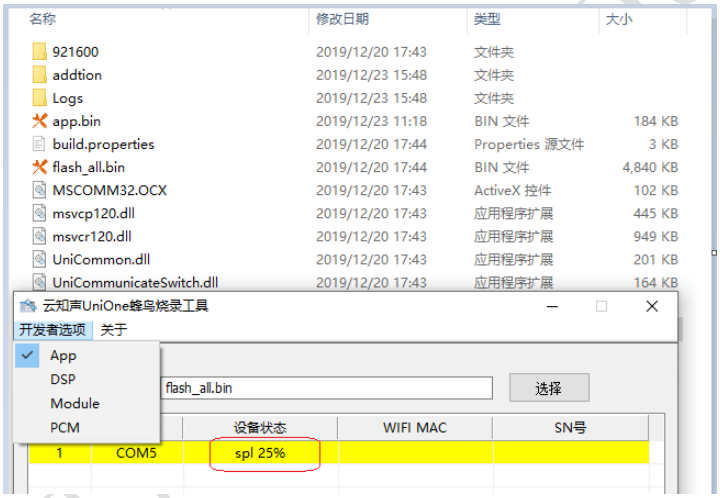
1. 平台使用常见问题

(1) 如需生成与命令词应答无关的本地播报音频，请联系 FAE 进行协助，后期会开放相关功能。

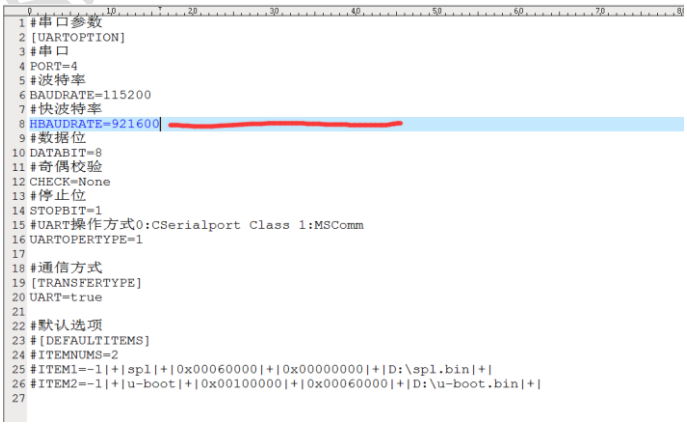
(2) 如遇到生成的本地播报音频中个别多音字发音错误的问题，请在平台进行多音字的标注，也可以选择同音词替换，并重新发布版本。

2. 烧录版本常见问题

(1) 烧写过程中卡住不动怎么办？



解决方案: 按下图修改配置文件 unionedebug.ini，将下载波特率降低为 921600，然后重启工具，再次尝试烧录，可能可以修复此问题。

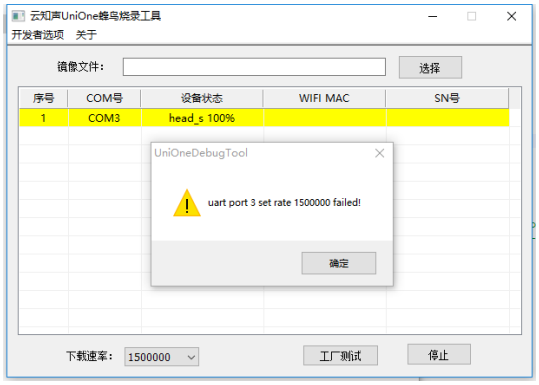


```
#串口参数
[UARTOPTION]
#串口
PORT=4
#波特率
BAUDRATE=115200
#快波特率
HBAUDRATE=921600
#数据位
DATABIT=8
#奇偶校验
CHECK=None
#停止位
STOPBIT=1
#UART操作方式0:CSerialport Class 1:MSComm
UARTOPERTYPE=1

#通信方式
[TRANSFERTYPE]
UART=true

#默认选项
#[DEFAULTITEMS]
#ITEMNUMS=2
#ITEM1=-1|+|spl|+|0x00060000|+|0x00000000|+|D:\spl.bin|+|
#ITEM2=-1|+|u-boot|+|0x00100000|+|0x00060000|+|D:\u-boot.bin|+|
```

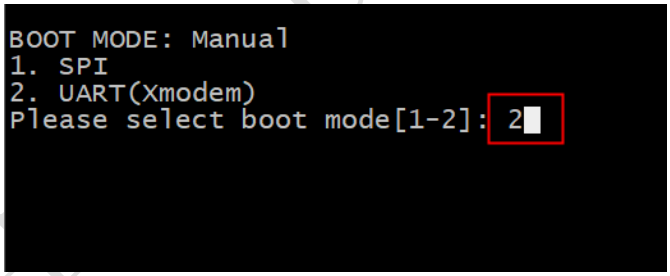
(2) 如出现如下图 set rate 1500000 failed 怎么办？



解决方案:从电脑 设备管理器中检查串口驱动版本，目前已知 Silicon Labs CP210x 10.x.x 版本的驱动会有此问题，请更新驱动到 6.x.x 版本即可。

(3) 遇到通讯失败怎么办？

解决方案：这种大概率是进入烧录模式失败引起，打开串口调试工具（如 SecureCRT），再次按照二.1 方法让板子上电，确保出现了开机选项，并且按 2 可以正常回显。



然后断开串口调试工具，点击烧录即可正常通讯进入烧录了。

3. 源码开发常见问题

(1) 可供二次开发的硬件资源有限（RAM<200KB，Flash<500KB），请预先评估二次开发的工作量，确认满足要求。以免因超出资源限制出现编译失败，烧录失败，启动失败等问题。

(2) 默认线程栈大小为 2KB，请避免使用大于 1KB 的栈空间，使用堆空间代替或直接使用静态内存（推荐），

(3) 根据语音性能测试指导手册，实际进行语音测试拿到测试结果，可以确定产品的阈值。确定好的阈值，可以在 sdk/vui/inc/uni_recog_common.h 中进行修改：

```
#define DEFAULT_WAKEUP_SCENE_STD_THRESHOLD      (-6.29)
#define DEFAULT_WAKEUP_SCENE_LOW_THRESHOLD      (-8.85)

#define DEEP_SLEEP_WAKEUP_SCENE_STD_THRESHOLD   (0.27)
#define DEEP_SLEEP_WAKEUP_SCENE_LOW_THRESHOLD   (-2.37)

#define DEFAULT_RECOGN_SCENE_STD_THRESHOLD      (-1.14)
#define DEFAULT_RECOGN_SCENE_LOW_THRESHOLD      (-2.42)
```

applications/sdk/vui/inc/uni_recog_common.h

(4) 方案中默认打开看门狗（watchdog）（代码实现为 app/src/sessions/uni_watchdog_session.c），从本质上来说就是一个定时器电路，一般有一个输入和一个输出，其中输入叫做喂狗，输出一般连接到芯片的复位端。看门狗的功能是定期的查看芯片内部的情况，一旦发生错误就向芯片发出重启信号。

```
  SessionManageRegister(g_watchdog_session->session);
  /* registe malibox */
  sys_hal_mailbox_register_callback(MAILBOX_WATCH_DOG_FEED,
                                    recv_watchdog_feed, NULL);
  /* start watchdog and enable restart */
  WatchDogStart(WATCH_DOG_RELOAD_SEC);
  fLib_WatchDog_SysResetEnable();
  return E_OK;
L_ERROR1:
  SessionDestroy(g_watchdog_session->session);
L_ERROR0:
  uni_free(g_watchdog_session);
  g_watchdog_session = NULL;
  return E_FAILED;
}
```

1，看门狗启动
2，使能看门狗触发系统重启机制

```
static int _session_transition_init() {
  MicroFsmTransition session_transition[] = {
    {STATE_IDLE, ID(WATCH_DOG_FEED_EVENT), _idle__watch_dog_feed_event},
  };
  g_watchdog_session_transition = uni_malloc(sizeof(session_transition));
  if (NULL == g_watchdog_session_transition) {
    LOGE(WATCH_DOG_SESSION_TAG, "malloc failed!");
    return -1;
  }
  uni_memcpy(g_watchdog_session_transition, session_transition,
             sizeof(session_transition));
  return (sizeof(session_transition) / sizeof(session_transition[0]));
}

static const char* _watchdog_session_state_2_str(uni_s32 state) {
  static const char *state_str[] = {
    [STATE_IDLE] = "STATE_IDLE",
  };
  if (state != STATE_IDLE) {
    return "N/A";
  }
  return state_str[state];
}

static Result _idle__watch_dog_feed_event(void *event) {
  LOGT(WATCH_DOG_SESSION_TAG, "action called.");
  fLib_WatchDog_ReStart();
  FsmSetState(g_watchdog_session->fsm, STATE_IDLE);
  return E_OK;
}
```

系统正常情况下会间歇性的收到喂狗消息：WATCH_DOG_FEED_EVENT

喂狗

开关看门狗功能可以通过 hal/inc/uni_iot.h 中的 UNI_WATCHDOG_DEBUG 宏进行控制，1 为打开，0 为关闭。

```
#define UNI_MEDIA_PLAYER_DEBUG          1
#define UNI_MP3_PLAYER_DEBUG            0
#define UNI_PCM_PLAYER_DEBUG            1
#define UNI_TTS_PLAYER_DEBUG            0
#define UNI_RECOGNIZ_DEBUG              1
#define UNI_SESSIONS_DEBUG              1
#define UNI_DNS_PARSE_DEBUG             0
#define UNI_NETWORK_DEBUG               0
#define UNI_WATCHDOG_DEBUG              1
#define UNI_UART_MANAGE_DEBUG           1
```

(5) 已经在版本 a 中进行了二次开发，开发后的版本为 a’，但是需求发生了变更，需要改动原有的唤醒词/命令词/播报等，使用新的离线命令词配置构建了版本 b，如何快速合并到版本 a’中：

① 替换 b 版本 rtt_unione_lite_app/applications/app/res/tone 目录至 a’版本中
② 替换 b 版本 rtt_unione_lite_app/scripts/cmd_content_total 文件至 a’版本中
③ 替换 b 版本 rtt_unione_lite_app/scripts/uni_cust_config.h 文件至 a’版本中
④ 替换 b 版本 rtt_unione_lite_app/flash_bin_creater/dsp.bin 文件至 a’版本中
⑤ 重新编译 a’版本

五、模型训练平台介绍

蜂鸟芯片方案支持通过云知声模型训练平台对声学模型进行定制。

当系统默认的声学模型性能无法满足用户需求时（注：此处用户需求，指经过语音性能量化测试的明确指标），可以使用模型训练平台，采集特定离线命令词的语音训练数据，进行模型快速定制，可有效提升语音识别准确率。

注：目前的模型训练平台仅支持蜂鸟芯片方案

目前模型训练平台（https://atp.hivoice.cn/）尚未正式在云知声开放平台中开放入口，仅在蜂鸟方案产品详情里，与配置终端能力里可进入。

产品详情入口：

配置终端能力入口：

正常的声学模型的流程为：

1. 新建模型 2. 创建录音任务 3. 收集录音文件 4. 选择录音文件进行模型训练 5. 使用模型

1. 新建模型

进入训练平台首页，点击 **+新建声学模型**

进入基础信息填写页面，需输入模型名称与模型描述。

注：一经确认无法修改

填写完毕后，点击**下一步**，进入训练模型页面。

由于新建模型尚未配置训练数据（即录音文件），页面显示为空，点击 **+添加录音**，进入添加录音页面。

如果该账号下从未采集过任何录音，则显示“暂无数据”。

需要首先采集录音，点击 **去新建录音任务**

2. 创建录音任务

录音任务的创建可以在创建模型过程中跳转进入，如 *1. 新建模型* 中所述。也可以在首页处点击**录音任务**，进入**录音任务**页面。

点击 **+新建录音任务**，进入录音任务的创建。

此处配置的录音任务，可通过录音 APP 获取进行采集。

进入新建录音任务页面，进行录音任务的配置。

3.1 录音任务基础信息

输入任务名称与任务描述，建议输入清晰明确的任务信息。

3.2 录音配置

语速配置：

默认仅勾选正常语速，如产品对于快语速或慢语速的唤醒与识别有要求，则可根据需求勾选。

详细语速要求可点击 ？ 号了解，目前系统定义的语速要求如下：

快语速	正常语速	慢语速
2 个字：≤700ms 3 个字：≤800ms 4 个字：≤900ms 5 个字：≤1000ms 6 个字：≤1100ms 7 个字：≤1200ms 8 个字：≤1400ms 9 个字：≤1600ms 10 个字：≤1800ms	2 个字：≥750ms 3 个字：≥850ms 4 个字：≥950ms 5 个字：≥1100ms 6 个字：≥1200ms 7 个字：≥1400ms 8 个字：≥1500ms 9 个字：≥1700ms 10 个字：≥1900ms	较少使用，暂不做要求

采样率：

目前仅支持 16000HZ。

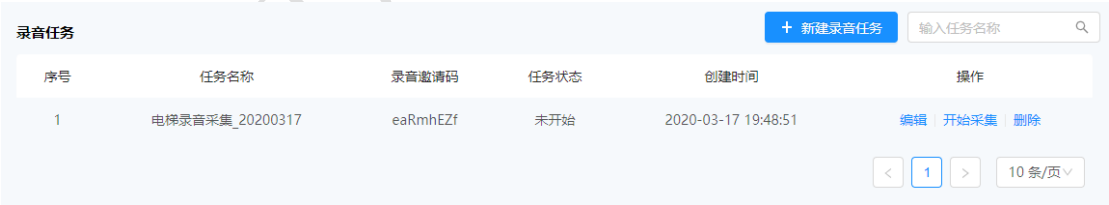
录音次数：

支持 1-3 次的配置，通常采集 3 次。

3.3 录音词条选择

当前训练平台仅支持导入云知声开放平台内创建的产品词条。支持选择唤醒词与命令词。选择产品与版本，完毕后点击 **确定**，即可发布录音任务。

点击确定后，跳转至录音任务列表页面



此时任务尚未开始，还可对创建的录音任务进行编辑。

如任务确认无误，可开始收集录音，则点击 **开始采集**。此时任务状态将切换为 进行中；此时任务只可查看，不可编辑。

开始采集后，录音邀请码生效，点击变蓝的邀请码，将一键复制邀请码与录音 APP 下载链接，目前仅支持安卓 APP。

对语音采集人发布任务时，只需粘贴，即可发送。

邀请文字如下：

录音邀请码：eaRmhEZf，通过链接：https://unios-udp.oss-cn-beijing.aliyuncs.com/unione/unisound_record_tool.apk 下载安装录音 APP，进入 APP 后跟随引导输入录音邀请码，即可进行录音采集，快来试试吧~

3. 收集录音文件

3.1 APP 使用

录音人可通过链接可下载 APP 工具，下载安装成功后，图标如下：

打开后，同意所有权限。

第一步，输入邀请码，点击**确认**。

第二步，如实填写个人信息，填写完成后，点击下一步。

第三步，进入录音任务显示页面，点击开始采集。

第四步，按照屏幕显示的要求开始录音，需确保处于安静环境。

保持安静，点击 **开始录音**，按照屏上语速要求读出显示的唤醒词/命令词，读完后点击 **停止录音**。

由于停止录音后会采集 1 秒钟的环境声音，因此**请始终保持安静**。

该词条录音完毕后，**播放录音** 按钮可用，可以重复收听已录制的语音。

如录音语速或能量（即音量大小）等不符合要求，会显示错误信息，提示重录。

如检测通过，则可以选择 **确定提交**，将该条语音上传云端。云端会进行二次检测。

点击 **确定提交** 后，如返回失败结果，需重新录制重新提交。通常为发音不清晰、环境太嘈杂、发音与词条不对应等情况。

提交成功，则表述云端检测通过，APP 将跳转下一条录音词条。

如所有词条录制完成，则跳转到完成页面。此时可以切换任务：即重新输入不同的任务邀请码；可切换录音人：即将录音所用手机交给其它录音人，并修改录音人信息。

注：

3.2 采集状态查看

录音任务发布者，可通过训练平台查看收集到的语音情况。

如已满足需求，则可点击 **结束采集**。

注：点击结束采集后，也可重新开启。

4. 选择录音文件进行模型训练

如录音文件采集完毕，可以选择想要的录音文件，开始对模型进行训练。

在训练模型页面中，点击 **添加录音**，选择已录音词条，点击 **确定**。支持添加多个任务中的不同录音词条文件。

添加录音完成后，可以选择按照男女性别 1:1 的比例选择要训练的录音文件，也可以直接全选。

由于示例为选取多个录音任务的文件，因此示例图中的训练词表内不同的录音任务采集数据分开展示（预计下期优化为单个词条仅显示总数）。

当选择完毕，训练数据已达要求，则可以点击 **开始训练** 。此时模型将切换为 **训练中** 状态。

注：1. 如训练数据里男生比女生多，则生成的模型对男生的识别率会比女生好；同理，如果正常语速数据大于快语速数据，那么正常语速更容易被识别；

2. 未达最低数据要求，则不可开始训练。

3. 为达优于系统默认模型的性能效果，训练数据建议不低于 50 人，下图人数仅为 demo 展示。

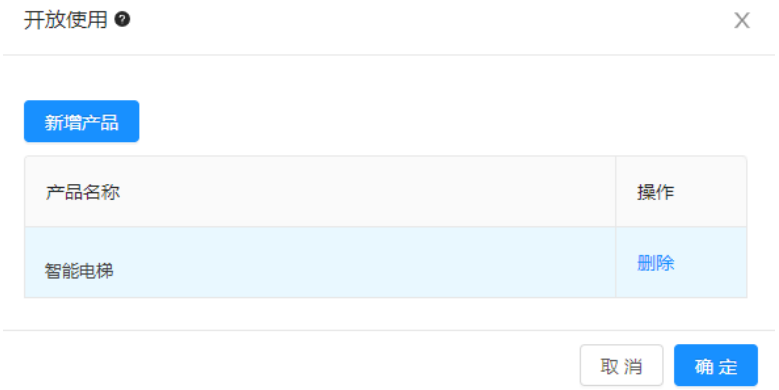
5. 使用模型

模型训练过程中与训练完成后，可点击 **查看**来查看该模型的词表。

由于示例为选取多个录音任务的文件，因此示例图中的训练词表内不同的录音任务采集数据分开展示（预计下期优化为单个词条仅显示总数）。

模型训练完成后，可以点击 **开放使用**，选择可以开放使用的产品。

开放使用弹框内，可点击 **新增产品** 对单个或多个产品进行开放使用。



确定后，在该产品中创建新版本时，配置终端能力，即可以选择使用该模型。

也可以选择模型 **下载**，通过替换产品对应版本里的本地模型后进行编译使用。

注：目前训练平台生成的模型仅适用蜂鸟芯片产品。