Voice Signal Process

Developer Guide

2017-08-05

杭州国芯科技股份有限公司

修订历史

版本	主要作者	版本描述	完成时间
1.0	任昂[renang@natioanlchip.com]	初稿	2017-08-08

目录

Developer Guide

1 概述

1.1 前言

VSP（语音信号处理）是专为 GX8010 芯片研发的语音信号处理框架。它运行在 MCU、DSP、NPU 和 CPU 上，主要完成待机和工作模式下的语音信号处理，比如降噪、去混响、回声消除、波束合成、特征提取、激活词识别等。除此之外，VSP 框架还实现了原始语音数据采集、系统的启动和初始化（BootLoader）、系统状态检测、系统功耗控制、LED 环效果、USB 声卡等功能。

VSP 在设计之初就兼顾考虑了语音信号处理的运算特点、硬件模块的特点、NPU 和 DSP 处理器的特点，并在很大程度上对算法实现的便利性、内存使用效率、处理器性能和系统功耗进行了充分的优化。

为了帮助算法工程师顺利的在 GX8010 实现、移植和优化语音处理算法，特编写此文档详细描述 VSP 框架的组成和工作原理，以及开发过程中所需要的相关信息。除了算法工程师之外，系统架构师、系统工程师、驱动工程师、应用工程师也可以参考该文档，以便在 VSP 框架的基础上实现附加功能。

本文首先描述 VSP 运行的硬件环境以及 VSP 的软件需求，然后给出主要的设计思想和实现原则，然后展开描述 VSP 软件的设计（异步消息、Context、流水线、工作模式、启动过程和软件部署），最后给出移植算法的参考方法，文末附带开发环境指南和 VSP 的 API 指南。

1.2 知识准备

为了充分的理解 VSP，并在 VSP 框架之上进行开发，并且让系统的稳定和高效的运转起来。一般来说，开发者需要具备以下知识、技能和相关经验：

- ✧ 语音信号处理的一般过程；
- ✧ 深度神经网络的工作原理，以及 GX8010 的 NPU 的开发流程；
- ✧ GX8010 语音信号处理子系统的硬件体系结构；
- ✧ C 语言的功底，开发者需要具备比较深厚的 C 语言功底，尤其要理解 C 语言的编译和链接的原理；
- ✧ Tensilica HiFi4 DSP 处理器的特点（尤其是矢量运算的特点），这对做算法优化尤其重要；
- ✧ Linux Kernel 驱动程序，VSP 框架最终都是依靠 Linux 驱动程序与上层应用进行数据交互的。如果开发者希望在 VSP 框架基础上添加附加功能，那么还需要了解 Linux Kernel 驱动程序的开发和调试；
- ✧ 嵌入系统（尤其是内存受限系统和前后台系统）的一些设计模式，比如多核的运算量分配策略、状态机、工作模式、消息派送、任务队列、静态内存分配等等；

本文将着重描述 VSP 框架所依赖的硬件子系统的相关信息，VSP 框架的组成和运行原理、算法开发环境的搭建、VSP SDK 的 API 等相关内容。其余知识请参考“参考文档”一节。

1.3 术语和缩略语

AEC	声学回声消除（Acoustic Echo Cancellation）
AGC	自动增益控制（Automatic Gain Control）
ASR	自动语音识别（Automatic Speech Recognition）
CPU	中央处理器（Central Process Unit） 可以执行复杂的算术和逻辑运算，用于运行操作系统、应用程序、网络协议、设备驱动、数据处理等功能。本文中，CPU 代表 GX8010 中集成的一颗 ARM Cortex A7 处理器。
DBF	数字波束合成（Digital Beam Forming）
De-Verb	去混响
DOA	波达方位（Direction Of Arrival） 指声音信号的到达方向（各个信号到达阵列参考阵元的方向角）
DSP	数字信号处理器（Digital Signal Processor） 可以高效的执行复杂的算术和逻辑运算，用于运行信号处理算法，比如 AEC、DOA、DBF、Denoise、De-Verb、声音特征提取等；本文中，DSP 代表 GX8010 中集成的一颗 Tensilica HiFi4 DSP 处理器核。
FFT	快速傅立叶变换（Fast Fourier Transform）
KWS	关键词检测（KeyWord Spotting）
LSP	链接脚本包（Link Script Package）
MAC	乘加器（Multiplier-ACcumulator）
MCU	微控制单元（Micro Control Unit）
NPU	神经网络处理器（Neutral-network Process Unit） 对矢量乘加、卷积、池化等运算进行优化，可以高效的运行神经网络模型。在本文中，NPU 代表 GX8010 集成的两颗神经网络处理单元。
PCM	脉冲编码调制（Pulse-code modulation）
PDM	脉冲持续时间调制（Pulse Duration Modulation）
PGA	可编程增益放大器（Programmable Gain Amplifier）
PWM	脉宽调制（Pulse Width Modulation）
SenseFlow	一个轻量级的开源 C++数据处理框架，可运行在 Linux、MacOSX、eCos 等操作系统上，实现特征提取、神经网络运算、与云端交互等功能。（https://github.com/nationalchip/senseflow）
TTS	文字转语音（Text-To-Speech）
VSP	语音信号处理（Voice Signal Process）

1.4 参考文档

1.4.1 DSP 相关文档

在开始开发 DSP 程序前，着重需要阅读以下文档：

- ✧ Xplorer 文档：当完成安装 Xplorer 后，用户可以在~/xtensa/XtDevTools/downloads/RF-2016.4/docs 找到以下文档：

1. 《Xtensa® System Software Reference Manual - Basic Runtime (XTOS) and HAL Reference Manual》（sys_sw_rm.pdf）
2. 《Xtensa® C and C++ Compiler User's Guide》（xtensa_xcc_compiler_ug.pdf）
3. 《HiFi 4 Audio Engine - User's Guide》（HiFi4_ug.pdf）
4. 《Xtensa® Linker Support Packages (LSPs) Reference Manual》（lsp_rm.pdf）

5. 《Xtensa® Software Development Toolkit User's Guide》（sw_dev_toolkit_ug.pdf）
6. 《Xtensa® C Application Programmer's Guide》（xtensa_capp_prog_guide.pdf）

✧ NatureDSP 库文档：当做算法定点化时，用户可采用 NatureDSP 的算法库充分利用 Tensilica HiFi4 DSP 的性能。

7. 《NatureDSP Signal for HiFi4 with VFPU - Digital Signal Processing Library Reference - Library Reference》

1.4.2 基础知识

✧ Linker Script：https://sourceware.org/binutils/docs/ld/Scripts.html

✧ KConfig：https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

✧ Makefile：https://www.gnu.org/software/make/manual/make.html

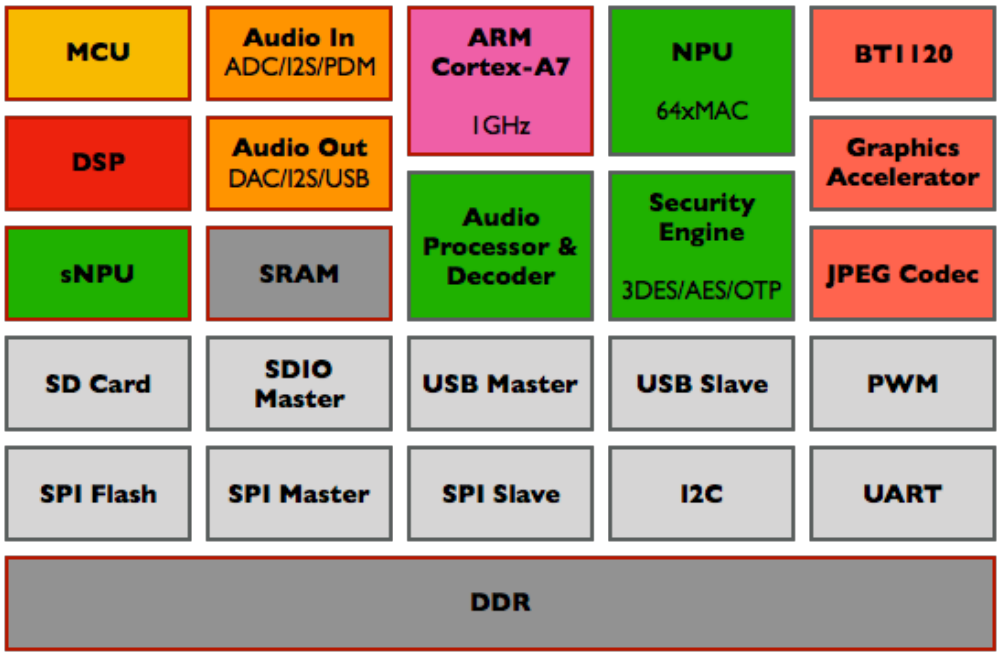
2 硬件子系统

2.1 概述

GX8010 芯片是杭州国芯科技股份有限公司为智能音箱、玩具机器人、语音助手等产品开发研制的高度集成的单芯片方案，它具有如下特点：

- ✧ 集成了 ADC，支持多达 8 路麦克风输入（也可支持数字麦克风和 PDM 输入）；
- ✧ 集成了一颗用于语音处理的 DSP，可运行自动增益控制、降噪、波束合成、寻向、回声消除、去混响、特征值提取等语音前处理算法；
- ✧ 集成了两颗用于语音识别的神经网络处理器（NPU），可运行 TensorFlow 模型，实现激活词识别、语音识别、语音合成等深度神经网络算法；其中一颗可被 VSP 和 SenseFlow 使用，另一颗只能被 SenseFlow 使用。
- ✧ 集成了音频解码器，支持 WAV/MP3/MP4/AAC/HE-AAC 等大部分音频格式；
- ✧ 集成了 ARM Cortex A7 处理器，带有 Neon 浮点处理单元，最高主频 1Ghz；
- ✧ 集成了 1Gb DDR3 内存和 768K SRAM（用于待机模式下的语音信号处理和唤醒词激活）；
- ✧ 集成了 SDIO、USB Host、USB Slave、SPI、I2C、I2S、BT1120 等扩展接口；
- ✧ 集成了动态功耗管理，和多级唤醒机制，实现低功耗待机；
- ✧ 可运行 Linux 和 RTOS 等主流嵌入式操作系统

其系统结构如下图：

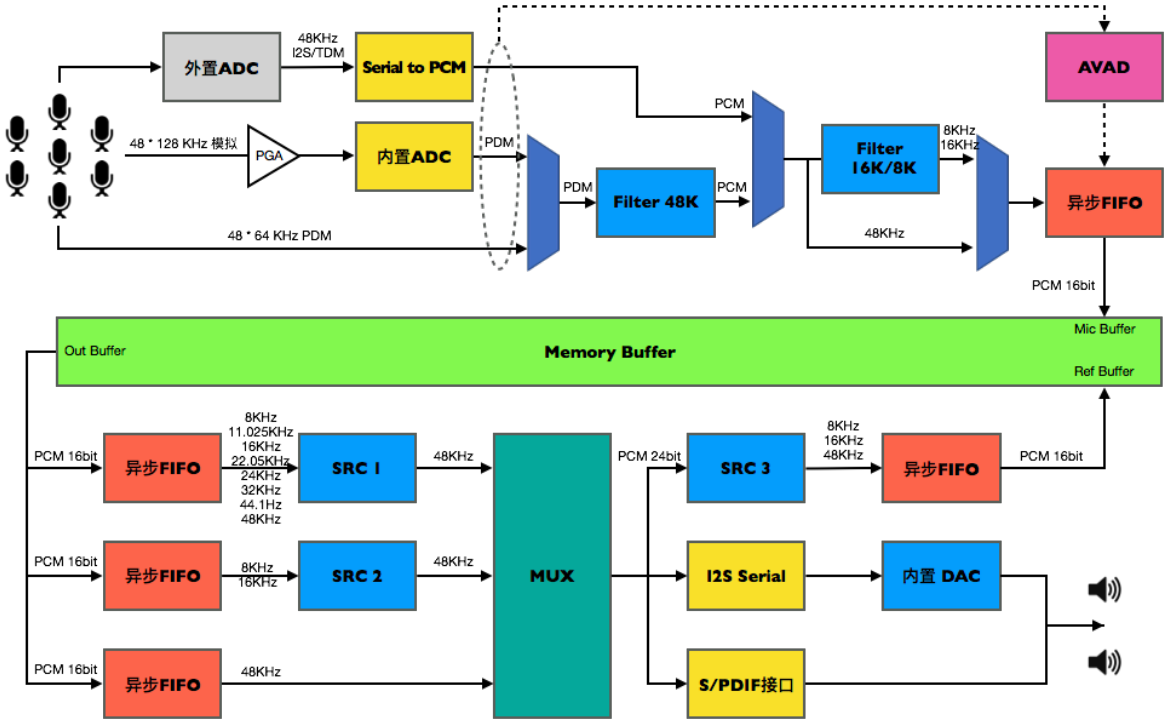


本文只会讨论与语音信号处理有关的硬件子系统，包括的硬件模块有：MCU、DSP、sNPU、Audio In、Audio Out、ARM、SRAM 和 DDR（图中红框的模块）。其他硬件子系统请参考相关文档。

2.2 音频采集子系统

2.2.1 *概述*

音频采集子系统负责采集环境音和参考音，并按一定规则存放在内存中。其基本结构如下图所示：



图中的上半部分为环境音的采集系统，其特点有：

- ✧ 可接收模拟麦克风、数字麦克风和 I2S/TDM 信号输入；
- ✧ 其中模拟麦克风输入通道上集成了可变增益的 PGA，可实现 0 到 50dB 的增益放大，调整步长为 2dB；
- ✧ 集成了可变增益的数字放大器，可实现 0 到 54dB 的增益放大，调整步长为 6dB；
- ✧ 可保存最多八个通道的麦克风数据到内存中，存储模式可支持乒乓 Buffer 模式或循环 Buffer 模式；
- ✧ 支持采样率自动转换，输出数据的采样率支持 8kHz、16kHz 和 48kHz，分辨率为 16bit；
- ✧ 支持按帧上报中断给 MCU，帧长度可调；
- ✧ 集成了 AVAD 功能，可根据能量阀值激活滤波器、PGA、PCM 写入等模块，实现低功耗待机。

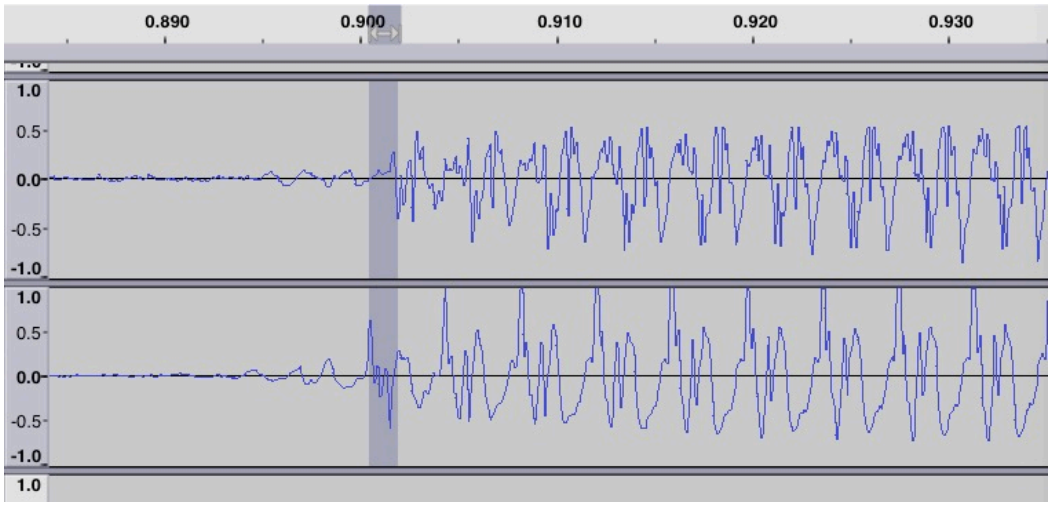
图中下半部分为参考音的采集系统，其特点有：

- ✧ 可接受内部参考音和外部参考音输入；其中内部参考音来自音频输出混音后的数据；外部参考音可来自模拟输入（比如放大器经衰减的信号）和 I2S 输入（比如外置音效处理器的输出信号）（图中未标出）；
- ✧ 集成了与环境音采集系统相同的采样率转换和写入系统，可支持最多 2 路的参考音输入；
- ✧ 参考音采集系统和音频输出系统可异步工作，即音乐播放结束后，参考音输入仍旧可以写入 0 数据，即保证了环境音和参考音同步存储；

各种环境音和参考音的配置如下表所示：

模式	环境音输入	麦克风增益	参考音输入
模拟麦克风	支持 1-8 路模拟麦克风	PGA： 0dB-50dB 数字： 12dB-54dB	支持 1-2 路数字参考音： ● 外部参考音（I2S 输入） ● 内部参考音
数字麦克风	支持 1-8 路数字麦克风（PDM 输入）	数字： 12dB-54dB	支持 1-2 路参考音： ● 外部参考音（I2S 输入，或模拟输入） ● 内部参考音
外部 ADC（I2S 或 TDM）	支持 1-8 路外部 ADC（I2S 输入）	数字： 0dbB-42dB	支持 1-2 路参考音： ● 外部参考音（I2S 输入，或模拟输入） ● 内部参考音

下图是实际采集的环境音（上面）和参考音（下面）数据。

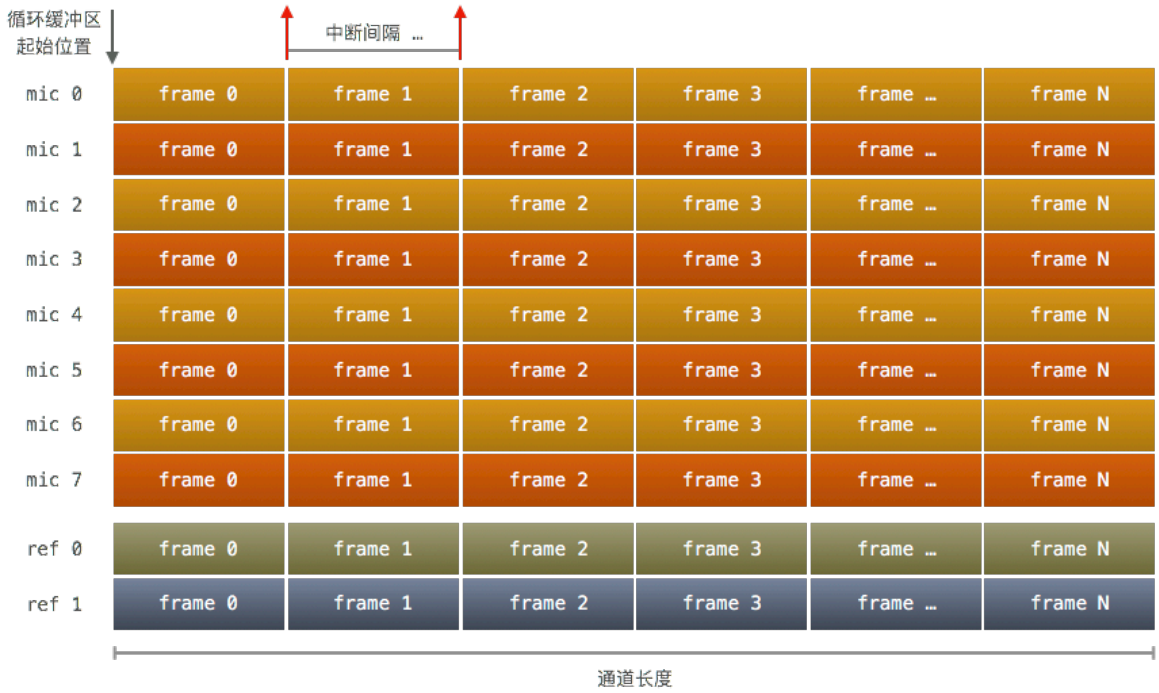


从图中可看出，参考音和环境音是同步存储的，并且参考音能略微超前于环境音存储到内存中[1]。 这个特性有利于 AEC 算法的实现，因为在处理每一帧麦克风数据的时候，它对应的参考音的数据已经存储在内存中了。

2.2.2 *存储规则*

音频采集系统能将环境音和参考音按通道和按帧有序的存储到循环缓冲区中，并按设定间隔触发中断给 MCU。如下图所示：

[1] 延迟时间包含两部分：1）扬声器和麦克风间距 25cm，声音传输需要 0.73 毫秒；2）麦克风输入滤波器造成的延迟。



存储的数据格式都是 16 位整形（带符号，也就是 Q15），采用了长 128 字节的乒乓 FIFO，所以对循环缓冲区有如下要求：

- ✧ 循环缓冲区的起始地址必须是 8 字节对齐；
- ✧ 每个通道的长度和中断间隔都必须是 128 字节的倍数；

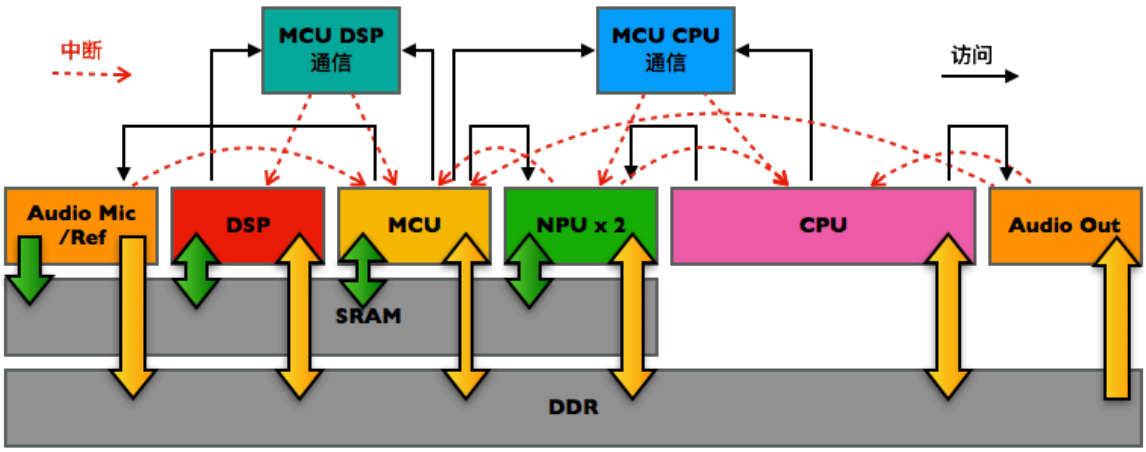
2.3 多核处理系统

2.3.1 概述

为了兼顾性能和功耗，GX8010 采用异构的多核处理系统。与语音处理相关的模块有：

- ✧ 音频采集子系统（Audio Mic 和 Ref）：特点见“音频采集子系统”。
- ✧ MCU：特点是功耗很低，工作频率也很低（最低可达 27M），系统上电后它便一直工作。它的角色是控制其他模块和处理它们发出的中断；由于 MCU 性能很低，它基本上不处理语音数据。
- ✧ DSP：特点是它拥有专门的硬件电路做信号处理。DSP 的运行频率为 400MHz，在 MCU 的控制下工作。MCU 负责加载 DSP 的 Firmware，向 DSP 发出处理请求。DSP 在没有处理请求的状况下可以进入低功耗模式（XWAIT_MODE）；
- ✧ NPU：特点是它拥有专门的硬件电路做神经网络运算。GX8010 配置有两颗 NPU，一颗的处理单元是 32x32MAC（简称为 sNPU），受 MCU 或 CPU 控制；另一颗的处理单元是 64x64MAC，受 CPU 控制。
- ✧ CPU：这是一颗 ARM Cortex A7 处理器，带有 Neon 7 浮点处理单元，运行频率可达 1GHz。MCU 可控制 CPU 的启动和复位，反之则不行。

这些模块的连接关系如下图所示：



图中绿色箭头表示待机状态下的内存访问，黄色箭头表示工作状态下的内存访问。除了 CPU 不能访问 SRAM 外，所有模块都可以直接访问 SRAM 和 DDR。除此之外：

- ✧ DSP 和 MCU 之间有一个通信模块，可以让 DSP 和 MCU 互发中断，以及传输 40 字节（10 个 32 位寄存器）的数据；通过这个模块，MCU 还可以复位和启动 DSP；
- ✧ CPU 和 MCU 之间也有一个通信模块，可以让 CPU 和 MCU 互发中断，以及传输 32 字节（8 个 32 位寄存器）的数据；通过这个模块，MCU 还可以复位和启动 CPU；
- ✧ DSP 和 CPU 之间没有直接的通信模块，无法直接通信，若要通信需要依靠 MCU 来完成；

2.3.2 *内存访问*

各模块访问内存的地址和方式有所差异，详见下表所示：

模块	SRAM	DDR
MCU 8K ICache 无 Dache	ICache： 0x20000000 - 0x20100000 直通： 0x40000000 ~0x40100000	ICache： 0x30000000 - 0x3FFFFFFF 直通： 0x50000000 - 0x5FFFFFFF
DSP 32K ICache 32K DCache	0x00000000 - 0x00100000	0x10000000 - 0x1FFFFFFF
其他模块	0x00000000 - 0x00100000	0x10000000 - 0x1FFFFFFF
CPU 32K ICache 64K DCache	禁止访问	0x10000000 - 0x1FFFFFFF

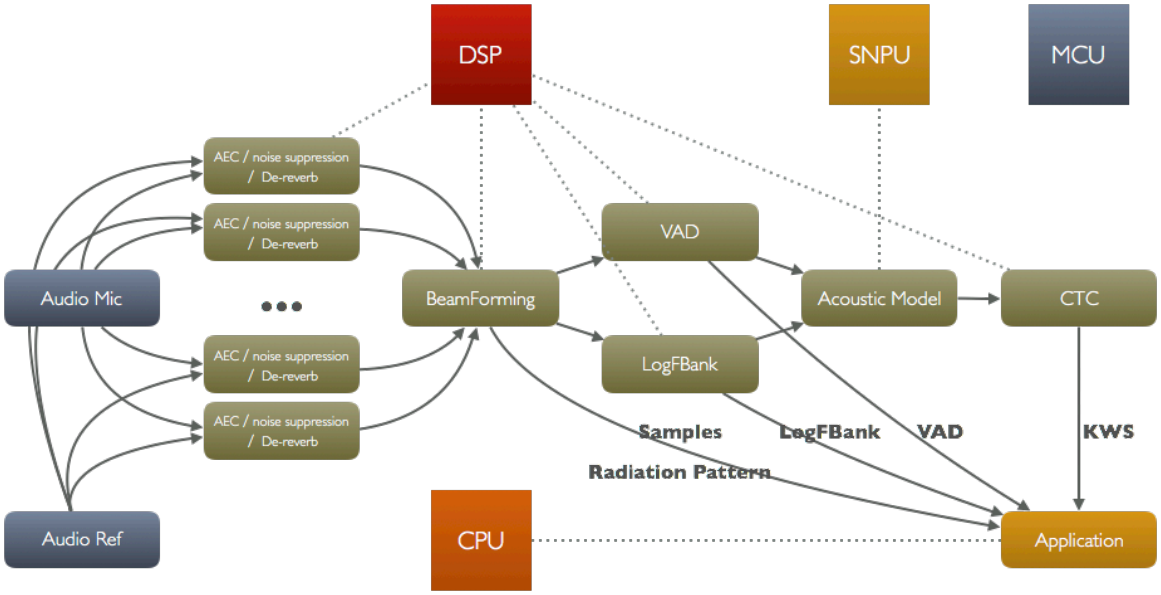
所有的模块（MUC、DSP、音频采集、CPU、NPU、……）的数据格式都是 Little Endian 的，共享数据不需要额外的 Endian 转换。

3 软件平台

3.1 需求概述

3.1.1 功能需求

VSP 的主要功能是做语音信号的处理。下图是一个语音信号前处理的流程示例：



环境音和参考音经 Audio Mic/Ref 采集后，做 AEC、降噪、去混响，然后做波束合成一路语音数据，然后做语音端点检测和特征提取（LogFBanks），然后做唤醒词识别，最后将波束合成后的语音数据、语音方位（Radiation Pattern）、语音特征值、VAD 状态和唤醒词（KWS）的识别状态发送给运行在 CPU 上的应用程序，应用程序可以：

- ✧ 利用 VAD 状态、唤醒词识别状态、语音方位信息控制 LED 环或向用户作出反馈；
- ✧ 利用语音特征值和 VAD 状态，用其他信号处理框架（比如 SenseFlow）做离线的语音识别；
- ✧ 利用波束合成之后的语音数据、VAD 状态，用云端的 ASR 服务做在线的语音识别。

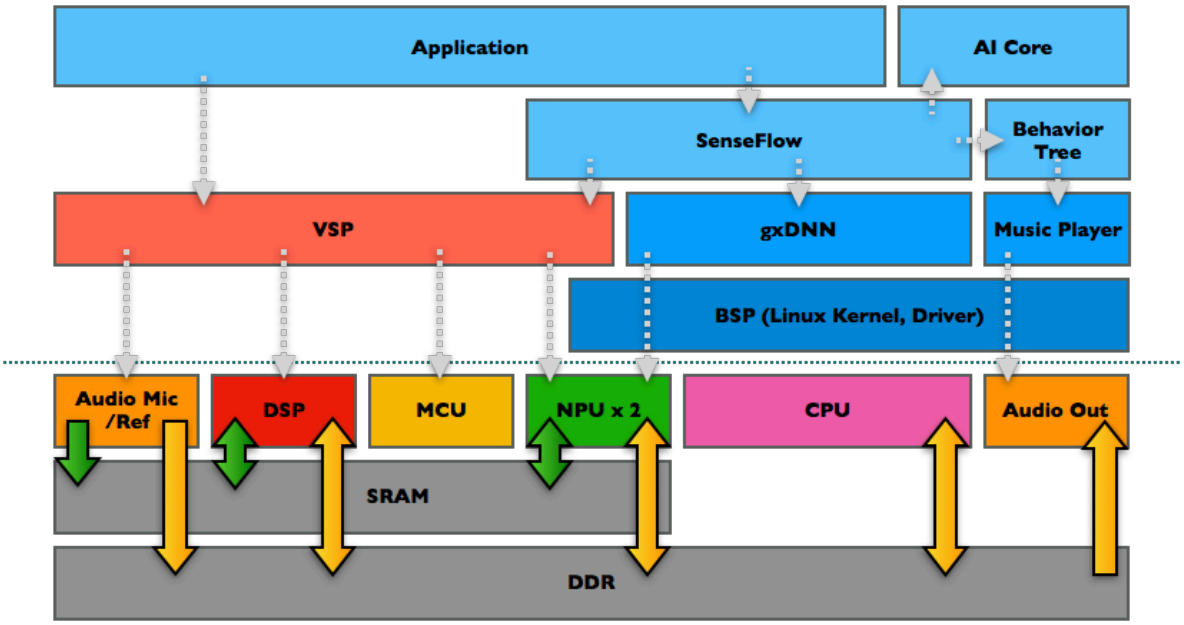
在这种语音信号处理过程中，DSP 负责 AEC、降噪、去混响、波束合成、DOA、特征值提取、VAD 检测、数据格式转换和 CTC 处理；sNPU 负责 KWS 模型的处理；MCU 控制所有模块按照一个既定的规则运转。

除此之外，由于 MCU 是整个系统最早运行的模块，VSP 还负责系统的初始化、CPU 的启动和待机控制、启动音乐播放等功能。另外，为了提高 LED 环的效果以及减轻 CPU 控制 LED 环的压力，VSP 也需要像 Ardiuno 那样控制 LED 环的运行。

另外，VSP 需要能独立工作，实现带算法的 USB 声卡功能。

3.1.2 系统需求

一般情况下，VSP 是与其他软件模块一起工作，构成一个系统来实现诸如智能音箱、玩具机器人、翻译机等功能的。它在整个软件系统中的位置如下图所示：



与 VSP 交互的软件模块有：

- ✧ 应用程序（Application）：应用程序主要实现智能系统的管理，比如配置 WiFi、按钮的响应、软件升级、系统待机等等。应用程序需要的 VSP 功能有：加载 DSP Firmware，控制 LED 环状态、获取语音通信的数据、切换 VSP 状态等；
- ✧ SenseFlow：SenseFlow 也是一个信号处理系统。与 VSP 不同的是，SenseFlow 运行在 CPU 上，用 C++实现，具有更大的灵活性，可以实现复杂的数据处理流程。在系统中，SenseFlow 通常用于离线的语音识别（ASR）、语音合成（TTS）、图像识别、与本地 AI 引擎交互、与云端交互等数据处理。SenseFlow 需要通过 VSP 获取离线语音识别所需要的数据，包括：原始的语音数据、语音方向、语音特征值、语音状态（VAD）等。
- ✧ 板级支持包（BSP）：这里的板级支持包指的是 CPU 上的板级支持包，包括 BootLoader、Linux 内核、驱动程序、基础的根文件系统。VSP 与 BSP 交互是通过驱动程序进行的。VSP 实现了一个 Linux 驱动程序，利用它与上层的应用程序和 MCU 交互。

3.1.3 *其他需求*

除了上述需求外，VSP 还需要满足以下需求：

- ✧ VSP 对 MCU 的性能要求低，关键路径的运算要足够快；
- ✧ VSP 的 MCU 要很健壮，这主要是因为 MCU 是不受其他模块控制的，如果崩溃，就只有依靠看门狗复位了；
- ✧ 在系统待机工作情况下，VSP 能在 768K 字节的 SRAM 上实现唤醒词激活功能，这里需要实现的算法有：降噪、波束合成、VAD、特征提取、KWS 模型、CTC 和 DOA；
- ✧ 知识产权保护。语音处理算法是合作伙伴的核心价值所在和重要关切点，VSP 要提供有效的算法保护方案；

3.2 设计思想

为了满足上述需求，结合实际的硬件平台和语音处理算法特点，提出以下 VSP 的主要设计思想和原则：

3.2.1 以多个帧为单元处理语音数据

以帧为单元处理语音数据具有以下优点和必要性：

- ✧ 许多运算，比如 FFT，FIR，都是需要积累一定量的数据方能进行的；
- ✧ 相比处理单个数据，批量处理数据更能发挥硬件的效率。比如批量处理数据可以充分使用 DSP 的 SIMD 指令；再比如从 64 位总线上读取数据，读取 8 个字节和读取 1 个字节的时间是一样的。

在 VSP 中，语音数据都是以单个或多个帧（1～4 帧）为单元进行处理的。帧的长度和每个单元内的帧数都可以指定。帧的长度由语音算法决定的，一般是 10ms 为一帧。而每个单元内的帧数一般由神经网络的模型决定的。

采取多个帧为单元批量处理数据的优点有：

- ✧ 大幅度减少模块之间交互（DSP 与 MCU 之间、MCU 与 CPU 之间）的开销，这个开销包括：发送和处理终端、清除和同步 Cache 等；
- ✧ 神经网络的输入特征往往由多个帧的特征量组成，采用多个帧为单元批量处理可以借由 DSP 来拼装输入特征，它的效率要比 MCU 高很多，同时降低了 NPU 驱动的复杂度和对内存的消耗。

当然这种方式也有缺点，最主要的缺点是：

- ✧ 增加了对循环缓存区的需求量。由于系统中存在五个独立的处理模块（Audio 采集、DSP、NPU、MCU、CPU），循环缓冲区至少需要保留 5 个以上单元的内存量方能保证系统的稳定运行。
- ✧ 增加了整体处理延时。这个延时是声音进入麦克风到 VSP 处理后交给 CPU 的延迟。最差情况下，延迟等于每个处理单元的时间长度乘以处理模块的数量。

所以要根据实际情况，根据多个因素来权衡选择每个单元的帧数。在 VSP 默认的配置中，在待机模式下：我们选择每个单元包含 3 个帧，帧长是 10ms，采样率是 16KHz，每个通道保留 8 个单元的内存，一共有 7 个环境音通道（待机模式下不需要采集参考音），所以我们需要保留的内存大小和最长处理延迟是：

$$\frac{16000\times10}{1000}\times2\times3\times8\times7 = 53760\text{（字节）}$$

$$10\times3\times5 = 150\text{（毫秒）}$$

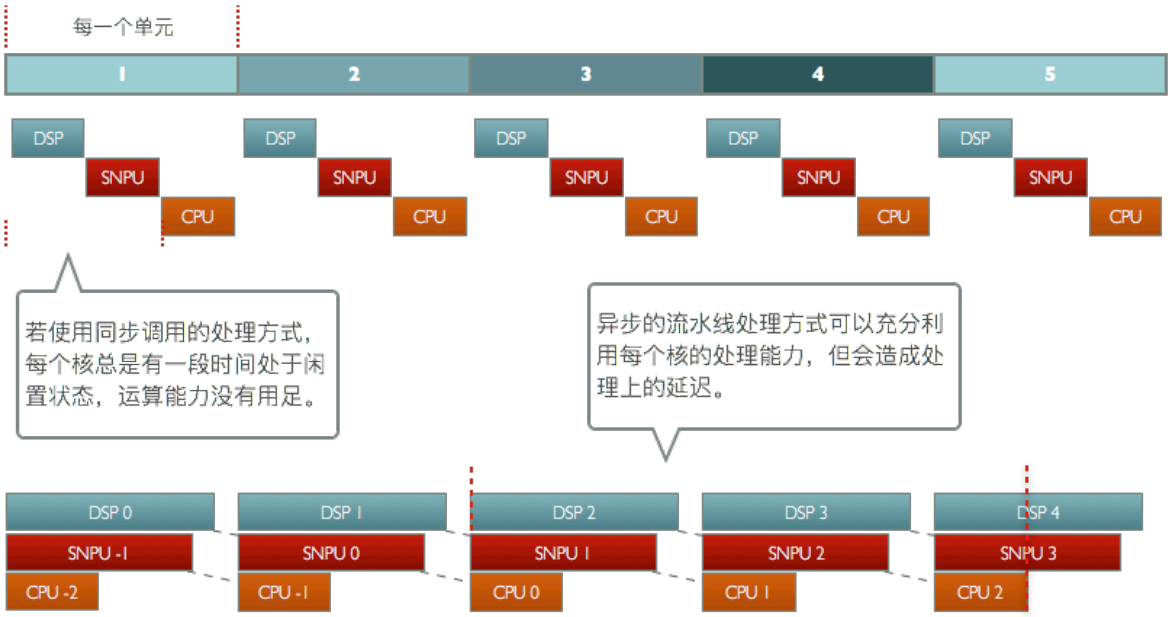
一般认为，0.15 秒的延迟对于语音应用是可以接受的。当用户说完一句话，半秒内就做出反馈，实际上是很快的速度。

在 VSP 中，用 Context（处理上下文）表示数据单元，详情可以参考“Context”一节。

3.2.2 采用消息驱动的异步流水线结构

在 GX8010 中，MCU 可以控制所有运算模块，反之则不然，所以在 VSP 中，只有 MCU 可以做运算的“调度者”。常用的运算调度有两种方式：

- ✧ 同步调用，就是 MCU 向各运算模块发起调用请求，并等待运算完成，然后调用下一个运算单元；
- ✧ 异步调用，就是 MCU 向运算单元发出请求，不等待运算完成；运算单元完成请求后，向 MCU 发出回复；当 MCU 收到某个运算单元的回复后，根据情况向下一个运算单元发出新的请求，……。如此周而复始；



由于功耗、内存和性能所限，MCU 没有使用 RTOS，而是采用了前后台系统。在没有操作系统提供的线程机制的情况下，如果采用同步调用方式，一个运算单元工作的时候，其他运算单元实际上处于待命状态，也就是运算能力没有用足。在这种情况下，对运算单元的性能要求就很高，不然就无法实现单元数据在规定的时间间隔内完成的要求。另一方面，多个运算单元之间采用同步消息互相调用，很容易造成死锁。

所以 VSP 只能采用异步调用机制，这是一种基于消息（和中断）的异步调用，语音数据存放在循环缓冲区中，运算请求和回复都不需要携带数据，而是只需携带指向数据的指针。在一个完整的运算处理过程中，起点在音频的采集系统，结束点在 CPU，其一般过程如下：

- ✧ 当音频采集系统完成一个单元的数据采集后，向 MCU 发出中断；
- ✧ MCU 收到音频采集系统的中断后，就组装个处理上下文（Context），然后向 DSP 发送处理请求；
- ✧ DSP 收到处理请求后，根据处理类型进行数据处理，处理完毕后向 MCU 发出回复；
- ✧ MCU 收到 DSP 的处理回复后，就组装个 sNPU 的任务，然后驱动 sNPU 进行运算；
- ✧ sNPU 完成运算后，向 MCU 发出中断；
- ✧ MCU 收到 sNPU 的中断后，向 DSP 发出一个做神经网络后处理的运算请求；
- ✧ DSP 收到请求后，进行运算，处理完毕后向 MCU 发出回复；
- ✧ MCU 收到 DSP 的处理回复后，向 CPU 发出处理数据的请求；
- ✧ CPU 收到请求后，便完成了整个 VSP 的处理流程，应用程序可以从队列中获取需要的数据。

实际的处理流程会因应用场景的不同而不同，这可以通过“工作模式”来统筹（详见“使用工作模式统筹调度资源”）。处理流程也可能中断或暂停，比如若没有检测到语音开始的信号，就没有必要做唤醒词激活；再比如 VAD 往往滞后，当收到语音开始的信号，要从历史的处理上下文中找出适合的数据做处理，所以要采取合适的内存管理策略。

- 关于消息格式，详见：“异步消息格式”一节。
- 关于流水线的具体设计，详见“处理流水线”一节。

3.2.3 *使用工作模式统筹调度资源*

语音处理的过程会因应用场景不同而不同，最典型的比如待机模式和工作模式的差异就如下表所列：

差异点	待机模式	工作模式
CPU	CPU 处于关闭状态	CPU 处于工作状态
DDR	DDR 处于自刷新状态，无法访问，所有数据都存放在 SRAM 中。	DDR 处于工作状态，数据都放在 DDR 中，这样才可以被 CPU 读取。
AEC	因为没有播放音乐，不需要做回声消除，因此也不需要采集参考音。	需要做回声消除，所以不管有没有播放音乐，都需要采集参考音。
VAD	为了节省功耗，要采用软硬结合的多重 VAD 检测流程。	为了提高响应速度和识别质量，要关闭硬件的 VAD 检测；
音频输出	不需要	需要
关键功能	识别唤醒词	识别唤醒词、采集音频和计算特征值供云端或本地的识别使用。

其他场景，比如系统启动过程根本不需要语音处理；比如音频配网过程中，需要以 48KHz 的采样率采集音频。随着产品功能的不断完善，场景会越来越多。如果在一个流水线中实现所有场景的需求，代码的复杂性会越来越大。不仅很难维护，内存的使用和处理单元的性能也很难优化。比如：流水线的每道处理过程都要检查场景类型，额外的开销就不可避免；再比如，系统启动时要播放一段启动音频，进入工作模式后，这段音频若不被释放，就会造成内存的浪费。

所以在 VSP 中，我们采用工作模式组织所有的流水线。具体设计，详见“工作模式”一节。

3.2.4 *以运算性能高和内存占用低为标准进行优化*

VSP 运行在内存受限和性能相对较低的环境中，所以各方面的决策要向运算性能高和内存占用低倾斜，具体而言，有如下原则：

3.2.4.1 原则 A：任务分配向性能高的运算单元倾斜：

当一个操作既可以在运算单元 A 实现，又可以在运算单元 B 中实现，如果 A 的性能比 B 的性能高，优先考虑用 A 来实现。

比如神经网络的输入特征值的拼接，MCU 和 DSP 都能实现，但是 DSP 的运算效率和访问内存的效率都比 MCU 高。这种情形优先采用 DSP 做这项工作，尽管 DSP 的负载可能已经很高了。

3.2.4.2 原则 B：去除与功能无关的代码

有些驱动代码原来采用了动态配置的方式。比如 Flash 驱动，集成了多种 Flash 的驱动，然后依靠 Probe，来确定使用何种驱动。这种情形下，应当裁剪调不会使用的 Flash 驱动。

一些 API 函数原本会检查输入参数的合法性，比如检查缓冲区是否有效，这些代码对于调试代码很有帮助，但是对于实际系统运行没有帮助，反而会增加处理时间和内存占用。

3.2.4.3 原则 C：最小内存搬运原则

每一次内存搬运，都会带来两个负面效果：

- ✧ 占用更多的内存；
- ✧ 消耗更多的总线带宽和 CPU 性能；

比如：音频采集系统已经将环境音和参考音存放在内存中，处理函数直接使用便可以了，而不必另行开辟内存存放之。因此，VSP 的很多驱动都只是简单封装一下寄存器的调用，而不是抽象出一个硬件模型，为的是节省内存和让出更多的处理能力给算法使用。

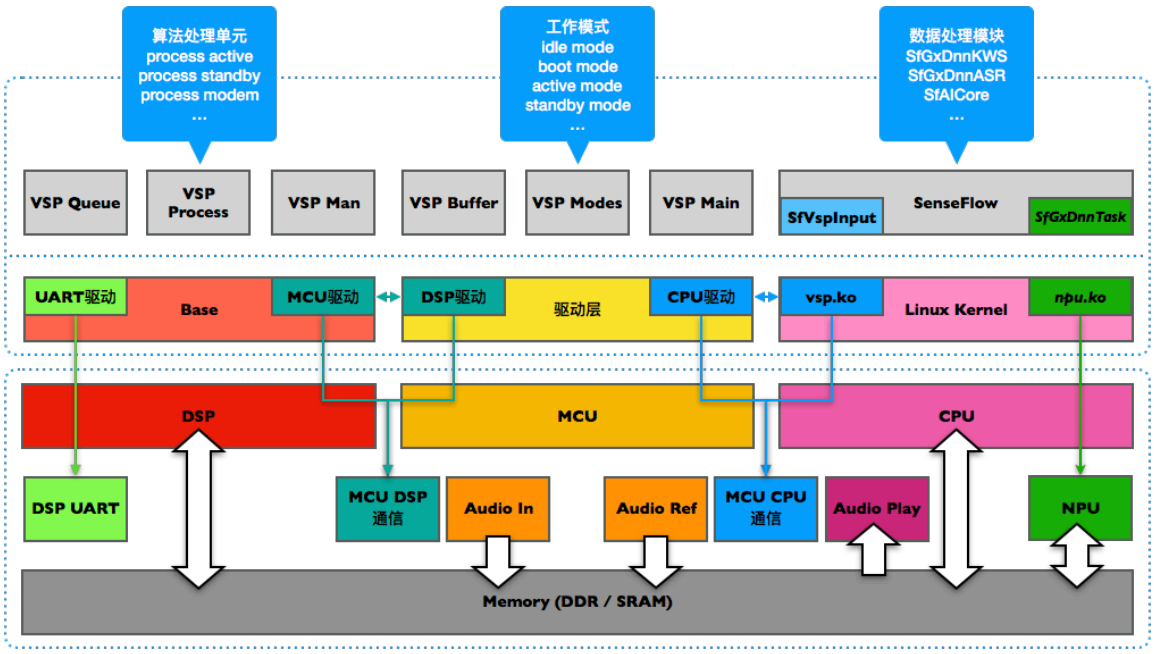
3.2.4.4 原则 D：在 build 过程中发现和解决错误

常规的软件开发有很多方法发现和解决错误，但这些方法都是需要耗费内存和运算能力的。而 VSP 所运行的环境恰恰没有那么多内存，MCU 的性能也较低。所以 VSP 的开发需要使用一些非常规的方式，来规避内存问题和其他问题。

比如 MCU 代码中去掉了 malloc 的支持，因为 heap 很小的情况下，malloc 很容易分配不到内存，并返回 NULL。而且 Heap 分配是一个很难决策的事情，必须通过大量测试才能认定 Heap 是否够用。去掉 malloc 支持后，所有的变量都用全局变量分配，很容易通过链接脚本了解 MCU 的内存占用情况。另外，通过工作模式的方式共享全局变量的使用空间，以这种方式来优化内存的使用。

3.3 软件构架

从纵向看，VSP 软件体系分为三个域：MCU、DSP 和 CPU。如下图所示：



这三个域的特点如下：

- DSP 域（dsp 目录）：DSP 的核心代码包括一个消息队列（VSP Queueu）和一组单元算法（VSP Process）。来自 MCU 的 Request 通过中断发送给 DSP，DSP 的中断处理程序把 Request 存放在消息队列中。然后 DSP 在主程序中逐一处理各 Request，并向 MCU 发送 Response。如果没有待处理的 Request，DSP 就进入睡眠，直到下一个 Request 的到来。采用这种结构的原因是 Request 的间隔是不均匀的，在 DSP 处理消息的过程中，可能会有多个 Request 到达。必须依靠消息队列让 Request 排队处理，才不至于让 DSP 错过 Request。
- MCU 域（mcu 目录）：MCU 的核心代码包括一组设备驱动、工作模式管理（VSP Modes）、音频采集缓冲区管理（VSP Buffer）。MCU 的代码采用前后台系统，由一组中断处理程序和主函数构成，这些程序被进一步用工作模式组织在一起。
- CPU 域（cpu 目录）：CPU 的核心代码包括一个 Linux 驱动程序和一个 SenseFlow 的组件。它提供了应用程序和 MCU 打交道的桥梁。

从横向看，这三个域的代码共享了一些公共的组件，包括：

- 一些数据结构的定义（include 目录）：这些数据包括异步消息结构（vsp_message.h）、Context（vsp_context.h）、Firmware 格式（vsp_firmware）、内核与应用的交互命令（vsp_ioctl.h）、LED 环的帧结构（vsp_led_frame.h），后续章节将会详细阐述它们；
- 一个 Kconfig 配置系统（scripts 目录，configs 目录）：一些涉及多个核的参数，比如关于 SRAM 的边界划分，都是靠配置系统完成的。
- 一个 Firmware 打包和加密工具（bin 目录）：将 DSP 和 MCU 程序打包成数据文件并加密，可供应用程序在适合的时机加载。

3.4 异步消息格式

MCU 与 CPU，以及 MCU 与 DSP 都是通过异步消息交换信息的，所有的异步消息都定义在 include /vsp_message.h 中。由于 DSP、MCU 和 CPU 的二进制文件可以分开部署，为了保证它们使用同一套协议解析消息内容，特定义一个消息版本宏：

```
#define VSP_MESSAGE_VERSION 0x20170628
```

当MCU握手DSP，CPU握手MCU时，会交换彼此的消息版本。如果不相等，就会停止握手。所以每次修改 include /vsp_message.h，都必须更改这个宏。

消息包含两部分：消息头和消息体。消息头定义如下：

```
typedef union {
    unsigned int value;
    struct {
        VSP_MSG_TYPE type:4;
        unsigned size:4;
        unsigned magic:8;
        unsigned param:16;
    };
} VSP_MSG_HEADER;
```

消息头各属性定义如下：

- ✧ 消息类型（type，4 位，必填）：表示消息的类型。经过精简和归并，16 个基础消息类型已足够了，可以利用 param 扩展出更多的消息类型；
- ✧ 消息体大小（size，4 位，必填）：表示消息头后面紧跟的消息体大小，以字（32 位）为单位。由于是可用于传输消息的寄存器最多只有 10 个，4 位大小也足够了；
- ✧ 魔数（magic，8 位，可选）：成对的消息，比如 PROCESS 和 PROCESS_DONE，可以用一个魔数来匹配；
- ✧ 参数（param，16 位，可选）：某些只需要传输少量数据的消息，可以用这 16 位传输参数，而不需要带有消息体。

VSP 的主要消息如下表所列：

消息类型	消息所带信息	说明
CPU 发送到 MCU（前缀 VMT_CM_）		
HELLO	vsp.ko 使用的消息版本	握手请求（任何模式）
LOAD_MCU	MCU 的 Firmware	加载 MCU 的 Firmware 的请求（仅 IDLE 模式）
LOAD_DSP	DSP 的 Firmware	加载 DSP 的 Firmware 的请求（仅 IDLE 模式）
SET_BUFFER	DDR 中分配的内存	设置工作模式所使用的 DDR 内存（仅 IDLE 模式）
SWITCH_MODE	工作模式（param 中）	切换工作模式请求（任何模式）

消息类型	消息所带信息	说明
START_STREAM	无	启动流传输（仅待机模式）
STOP_STREAM	无	关闭流传输（仅待机模式）
POWER_OFF	无	关闭 ARM 电源（仅待机模式）
PROCESS	处理上下文 处理类型（param）	处理一个单元数据（仅测试模式）
SET_LED_QUEUE	设置 LED 帧队列	设置 LED 帧队列（仅 IDLE 模式）
MCU 发送到 CPU（前缀 VMT_MC_）		
MCU_INFO	MCU 的相关配置信息	HELLO 的回复（任何模式）
LOAD_MCU_DONE	无	LOAD_MCU 的回复
LOAD_DSP_DONE	无	LOAD_DSP 的回复
SET_BUFFER_DONE	无	SET_BUFFER_DONE 的回复
SWITCH_MODE_DONE	当前工作模式（param 中）	SWITCH_MODE 的回复
PROCESS_DONE	处理上下文 处理类型（param）	VSP 完成一个单元的数据处理
LOG	LOG 缓冲区	将 MCU 或 DSP 的调试信息发送给 CPU
SET_LED_QUEUE_DONE	无	SET_LED_QUEUE 的回复
MCU 发给 DSP（前缀 VMT_MD_）		
HELLO	MCU 所使用的消息版本	握手请求
PROCESS	处理上下文 处理类型（param）	处理一个单元数据
HALT	DSP 进入 Halt 状态	MCU 发起这个请求强制 DSP 停止运行并进入 HALT，然后 MCU 可以重新加载 DSP 的 Firmware
DSP 发给 MCU（前缀 VMT_DM_）		
HELLO	DSP 所使用的消息版本	握手请求 / 回复
PROCESS_DONE	处理上下文 处理类型（param）	完成一个单元数据的处理
PROCESS_ERROR	处理上下文 处理类型（param）	无法完成一个单元数据的处理
LOG	LOG 缓冲区	将 DSP 的调试信息发送给 MCU

3.5 Context

VSP 使用 Context（处理上下文）来存放每个处理过程的输入、输出和中间过程数据，这个数据结构在 vsp_context.h 中定义。与消息类似，由于 DSP、MCU 和 CPU 的二进制文件可以分开部署，为了保证它们使用同一套协议解析 Context 的内容，这里也定义了一个 Context 的版本宏：

```
#define VSP_CONTEXT_VERSION 0x20170728
```

在 Context 传递过程中，处理 Context 的一方会检查 Context 的版本号，如果与己方的版本不同，就会拒绝处理 Context。

Context 有两个部分内容：VSP_CONTEXT_HEADER 和 VSP_CONTEXT。

VSP_CONTEXT_HEADER 包含一些不常变化的信息，其定义如下：

```
typedef struct {
    unsigned int        version;
    unsigned int        mic_num;
    unsigned int        ref_num;
    unsigned int        frame_num_per_context;
    unsigned int        frame_num_per_channel;
    unsigned int        frame_length;
    unsigned int        sample_rate;
    unsigned int        logfbanks_dim;
    unsigned int        logfbanks_group_num;
    unsigned int        far_field_pattern_num;
    unsigned int        out_buffer_size;
    short              *mic_buffer;
    unsigned int        mic_buffer_size;
    short              *ref_buffer;
    unsigned int        ref_buffer_size;
} VSP_CONTEXT_HEADER;
```

各字段的含义如下：

字段	说明
version	Context 的版本号。
mic_num	麦克风配置数量，一般第 0 路为中央麦克风，其他路麦克风按逆时针顺序排列；
ref_num	参考音配置数量；
frame_num_per_context	一个上下文包含的帧数（麦克风和参考音一致）
frame_num_per_channel	通道上包含的帧数（麦克风和参考音一致）
frame_length	帧长度
sample_rate	采样率
logfbanks_dim	特征值的维数

字段	说明
logfbanks_group_num	特征值组的数量，注意这里特征值组的数量与 frame_num_per_context 不一定是相同的，这取决于神经网络的输入维度；
far_field_pattern_num	远场的强度特征的维度
out_buffer_size	音频输出缓冲区的大小（字节）
mic_buffer	麦克风缓冲区的头指针
mic_buffer_size	麦克风缓冲区的大小（字节）
ref_buffer	参考音缓冲区的头指针
ref_buffer_size	参考音缓冲区的大小（字节）

VSP_CONTEXT 则包含每一个单元数据的内容，其定义如下：

```
typedef struct {
    VSP_CONTEXT_HEADER *ctx_header;
    unsigned            mic_mask:16;
    unsigned            ref_mask:16;
    unsigned int        frame_index;
    unsigned int        ctx_index;
    unsigned int        vad:8;
    unsigned int        kws:8;
    unsigned int        gain_current:8;
    unsigned int        gain_setting:8;
    int                 direction;
    unsigned int       *farfield_pattern;
    float              *logfbanks;
    short              *out_buffer;
} VSP_CONTEXT;
```

各字段的解释如下：

字段	说明
ctx_header	指向 Context 头的指针，注意这里使用的是设备地址，MCU 访问时要加上 0x40000000
mic_mask:16	当前数据的麦克风使能情况。在待机状态下，DSP 若要关闭某些麦克风的通路以减少功耗，可以设置这个值。
ref_mask:16	当前参考音的使能情况。
frame_index	帧序号。这是一个从零开始编号的 32 位序号。当序号等于 0 时，往往预示着模式的切换或采集模块的复位，处理单元应该复位内部的状态变量。
ctx_index	上下文序号。与帧序号相似，等于： frame_index/ frame_num_per_context

字段	说明
vad:8	VAD 状态，供 DSP 设置用；
kws:8	唤醒词状态，供 CTC 模块设置用；
gain_current:8	当前麦克风增益；
gain_setting:8	若 DSP 要调整麦克风增益，则需设置这一项；
direction	DOA 方位角；
farfield_pattern	指向存放远场声强的缓冲区，缓冲区大小由 far_field_pattern_num 定义；
logfbanks	指向存放特征值的缓冲区，缓冲区大小由 logfbanks_dim 和 logfbanks_group_num 定义
out_buffer	指向输出音频的缓冲区，缓冲区大小由 out_buffer_size 定义

为了减少运算开销，Context 没有采用类似 protobuf 的机制来封装数据，而是采用了非常原始和直接的方式存放数据。而 Context 中存放的内容与实际的算法实现和产品需求紧密相关。这一定程度上增加了 Context 与各模块的耦合性，提高了 VSP 的开发难度，各模块的开发者必须了解其他模块如何处理 Context 的，并且根据整体的需求添加或删减字段。

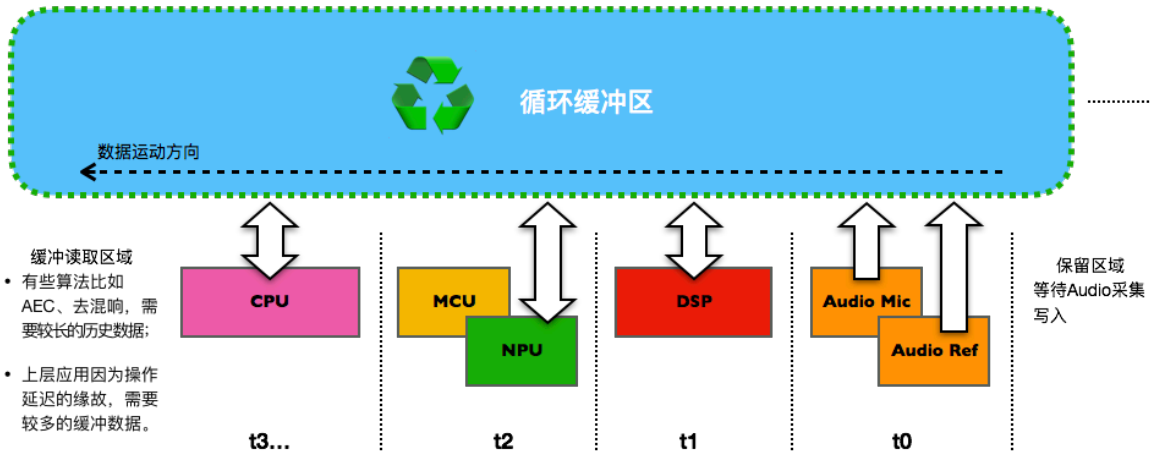
另外，Context 都存放在内存中，而 DSP 和 CPU 都是通过 Cache 访问内存的，所以在处理新的 Context 前要清除 Cache，而处理完 Context 后要将 Cache 写回到内存中。

3.6 处理流水线

VSP 有三根主要的流水线：

- ✧ 激活状态的语音处理流水线：起点在音频采集模块，终点在 CPU，功能是采集和处理语音数据，做唤醒词识别，并交给离线语音识别或在线语音服务器处理。这是最长的，参与者最多的流水线。
- ✧ 待机状态的语音处理流水线：起点在音频采集模块，终点在 MCU，功能是做唤醒词识别，并根据结果唤醒 CPU。
- ✧ 声波 Modem 的处理流水线：起点在音频采集模块，终点在 CPU，功能是将承载在声波中的信息解析出来。

这些流水线都是靠异步消息（PROCESS 和 PROCESS_DONE）和 Context 来实现的。其基本结构如下图所示：



流水线中包含以下主要模块：

- ✧ 循环缓冲区。循环缓冲区包含环境音的 Buffer，参考音的 Buffer 和 Context 的 Buffer。但是在流水线中流动的不是循环缓冲区的数据，而是指向循环缓冲区中所需处理的数据的指针。这些指针被进一步的封装，以消息的形式在流水线中流动。
- ✧ 处理单元。即参与数据处理的单元。

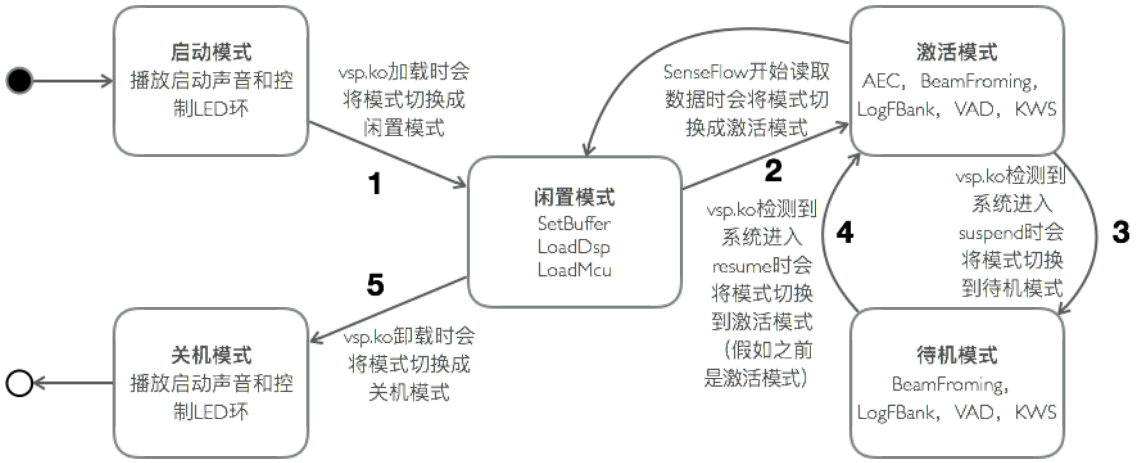
循环缓冲区在 vsp_buffer.c 中定义，有两个缓冲区：

- ✧ 处于 SRAM 的缓冲区，这片缓冲区只有环境音的 Buffer 和 Context 的 Buffer，缓冲区的大小比较小，只是留给待机的时候使用；
- ✧ 处于 DDR 的缓冲区，这片缓冲区有环境音 Buffer、参考音 Buffer 和 Context 的 Buffer，这片缓冲区相比 SRAM 的比较大，留给系统处于激活状态的时候使用。一些算法，比如 AEC、去混响，由于需要更多的历史数据，以及上层应用读取数据有阵发性，以及 DDR 中有充足的空间，所以缓冲区开的相对比较大些。

缓冲区的大小可通过 Kconfig 配置系统进行调整，最终这些配置项都保存在 MCU 中，其他模块，比如 CPU，是根据 MCU 中的配置项来分配缓冲区的。

3.7 工作模式

VSP 的流水线最终是靠工作模式组织在一起的。当前的工作模式决定了哪根流水线处于工作状态，或者哪条消息能被处理，哪条消息不能被处理。下图是系统工作模式的状态图：



各模式之间的变迁的说明如下：

- ✧ 启动进入启动模式：系统启动的时候直接进入启动模式，在这个模式下，MCU 会引导 CPU 启动 Linux 系统。在 Linux 启动过程中，MCU 会驱动扬声器播放启动音乐，会驱动 LED 环提示启动过程，会采集环境音数据做麦克风的校准；
- ✧ 启动模式到闲置模式：当 Linux 启动到加载 vsp.ko 时，vsp.ko 会通知 MCU 切换到闲置模式。闲置模式意味着所有的流水线和功能都停止工作，所以闲置模式就是用来设置资源的。比如加载 DSP Firmware，设置 DDR Buffer 都是在这个模式下进行的。
- ✧ 闲置模式到工作模式：当 SenseFlow 启动后，会通过 vsp.ko 将 VSP 切换到激活模式（Active Mode）。这时候激活状态的语音处理流水线就开始工作了；
- ✧ 工作模式到待机模式：当应用程序认为需要待机时，就会发起 Linux 的 suspend 操作，操作系统调用每个驱动程序的 suspend 操作，这时候 vsp.ko 就会通知 MCU 切换到待机模式。激活状态的流水线会被关闭，取而代之的是待机状态的流水线。当 CPU 运行到最后，需要关闭电源时，会调用待机模式的 POWER_OFF 消息实现关机。
- ✧ 待机模式到工作模式：当待机流水线识别到唤醒词后，MCU 会唤醒 CPU，然后 Linux 会调用 vsp.ko 的 resume 操作，vsp.ko 就会通知 MCU 切换到激活模式，然后待机状态的流水线停止工作，激活状态的流水线开始工作。
- ✧ 工作模式到闲置模式：当应用程序退出时，如果没有其他应用程序需要使用 vsp，则会进入闲置模式。
- ✧ 闲置模式到关机模式：当系统开始进入 shutdown 阶段时，vsp.ko 会被移除，在清理过程中，vsp.ko 会通知 MCU 切换到关机模式，这个模式里，会播放关机音乐和闪灯。最后，CPU 通过 POWER_OFF 消息通知 MCU 关闭所有电源。

工作模式的种类定义在 vsp_message.h 中，如下所示：

```
typedef enum {
    VSP_MODE_IDLE,
    VSP_MODE_BOOT,
    VSP_MODE_STANDBY,
    VSP_MODE_ACTIVE,
    VSP_MODE_MODEM,
    VSP_MODE_BYPASS,
    VSP_MODE_TEST,
} VSP_MODE_TYPE;
```

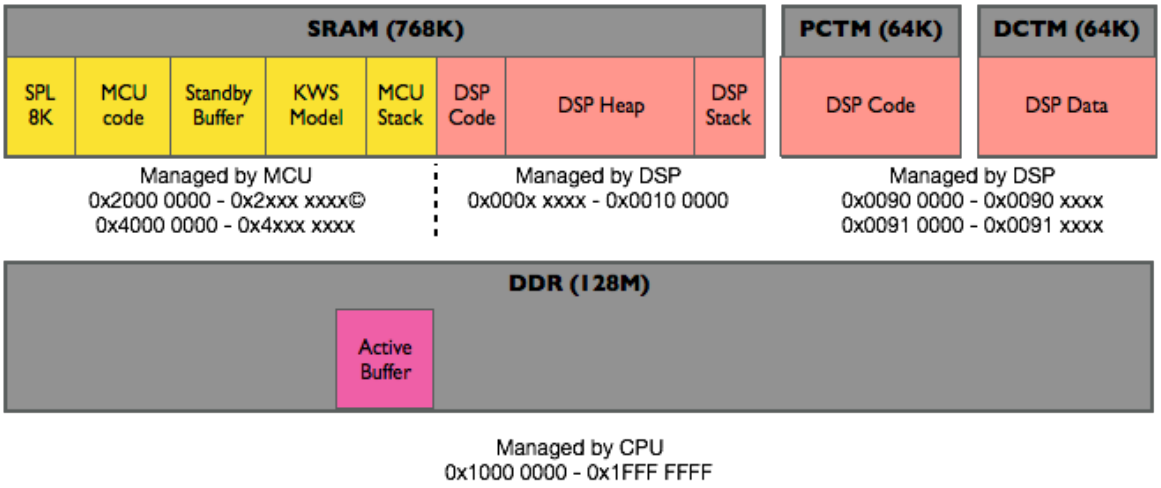
其中 MODEM 模式是用于声波通信传递 WiFi 的 SSID 和密码的，BYPASS 和 TEST 是用于调试的，BYPASS 可将环境音和参考音直接传递给 CPU，而不经过 DSP 的处理；而 TEST 模式是 CPU 将数据传递给 DSP 处理，再接收回来。

开发者也可以添加自己的工作模式，实现工作模式请参考 vsp_mode.h，要实现四个回调函数：

- ✧ VSP_MODE_INIT：初始化工作模式，返回 0 表示初始化成功，其他值表示初始化失败，系统会进入闲置模式；
- ✧ VSP_MODE_DONE：清理工作模式，没有返回值，必须成功清理；
- ✧ VSP_MODE_PROC：处理来自于 CPU 的消息，成功就返回 0，其他址表示处理失败；
- ✧ VSP_MODE_TICK：后台维护，一般不需要实现。

3.8 内存布局

VSP 的内存布局如下图所示：



所有的内存被分成三个区域，由 MCU、DSP 和 CPU 各自管理：

- ✧ MCU：处于 SRAM 的前半段，包含：8K 的 Bootloader、VSP 的 MCU 部分、待机模式的流水线所需要的循环缓冲区、唤醒词模型和 MCU 的堆栈；
- ✧ DSP：处于 SRAM 的后半段和 PCTM、DCTM。包含 DSP 的代码、数据、堆和栈；
- ✧ CPU：所有的 DDR 皆由 CPU 管理。所以激活模式的流水线需要的循环缓冲区虽然是 MCU 管理的，但却是由 CPU 从自己管理的 DDR 内存中分配的。

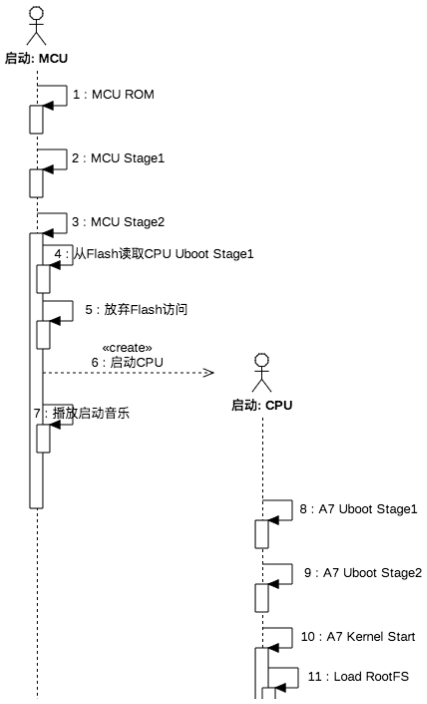
SRAM 的内存优化策略是尽可能的将 MCU 和 DSP 的边界向左移，留出尽可能多的内存供 DSP 的堆（Heap）使用。为此我们取消了 MCU 的堆，因为假如 MCU 存在堆，即内存分配通过 malloc 获得，我们必须通过一系列的测试才能知道多大的堆是合适的。一般来说这个测试值比实际使用的内存要大些，不然很容易出现内存分配失败而造成系统怠机。而取消了堆分配之后，所有内存都是通过全局变量分配，我们很容易的就可以通过链接脚本在链接过程中了解 MCU 真实的内存需求，从而进行更有针对性的优化。

3.9 启动过程

启动过程包含：系统启动阶段、VSP 加载过程和流水线启动过程三部分。

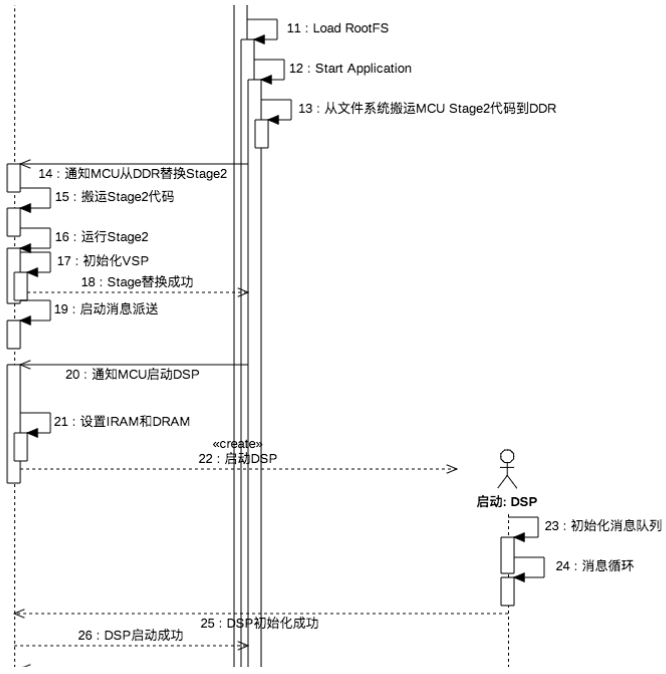
3.9.1 系统启动阶段

由于 MCU 是整个系统最早启动的部分，所以 VSP 承担了引导整个系统启动的任务。系统启动阶段的优化策略是尽可能早的让 CPU 启动起来，因为 CPU 的各方面性能都比 MCU 强，通过 CPU 加载会比 MCU 快得多。在 CPU 启动之前，MCU 便放弃了 Flash 的访问控制权。在 CPU 启动过程中，MCU 可以做一些 VSP 的准备工作，比如播放启动音乐，控制 LED 环亮灯，以及校准麦克风阵列。这个过程如下图所示：



3.9.2 *VSP 加载过程*

当 CPU 启动完毕后，便可以加载 VSP 的语音处理部分，如下图所示：



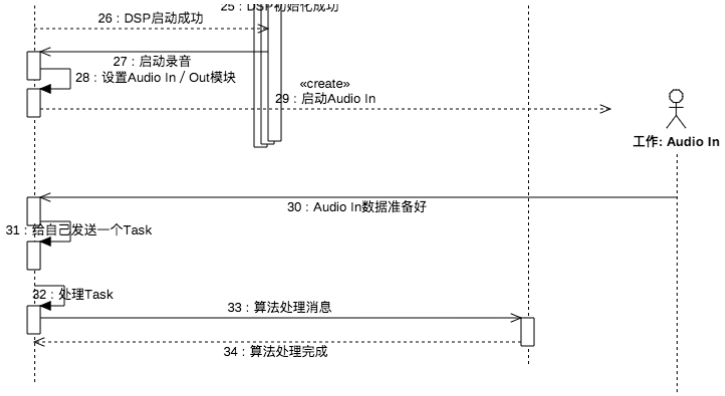
其过程是：

- ✧ 首先应用程序从文件系统中读取要加载的 MCU 和 DSP 的程序；
- ✧ 然后通过 vsp.ko 向 MCU 发出加载 MCU 的指令。MCU 于是切换到 Stage1，并且从内存中加载 MCU；
- ✧ 然后通过 vsp.ko 向新加载的 MCU 发出加载 DSP 的指令。

重新加载 MCU 的原因是：这时候加载的 MCU 包含语音处理流水线的代码和神经网络模型，而这些是启动过程所不需要的；另一方面语音处理过程中，也不需要启动音乐数据。

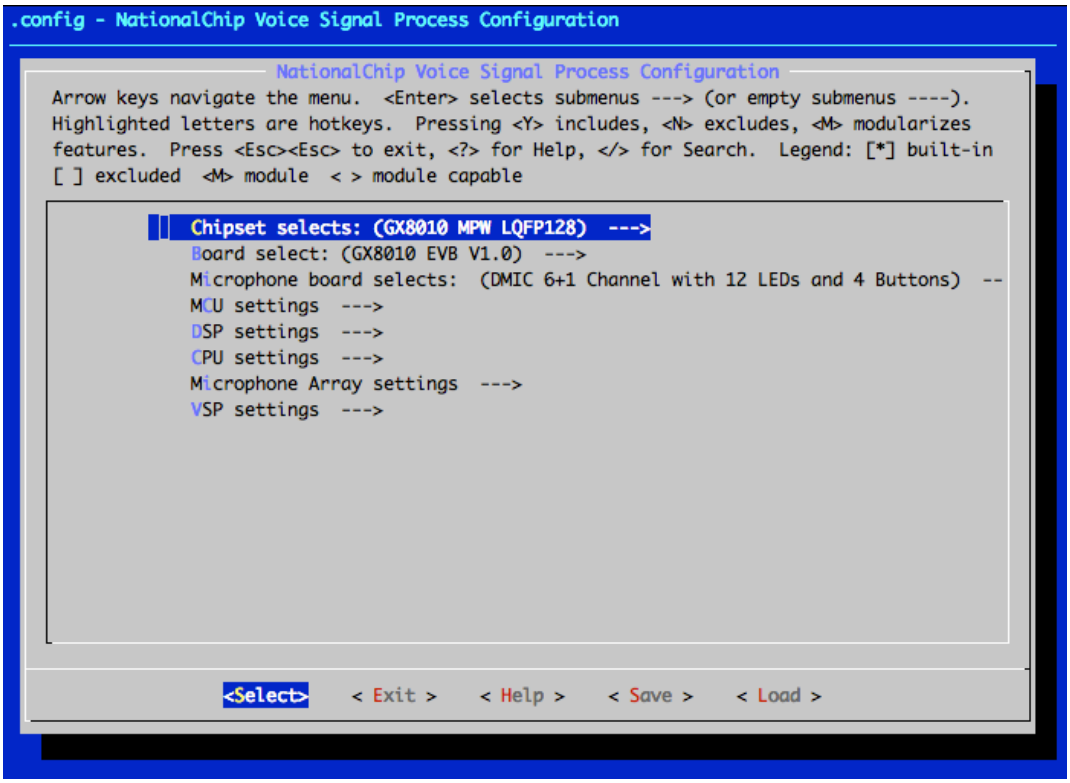
3.9.3 *启动流水线*

当所有的准备工作完成后，应用程序便可通过 SenseFlow 启动流水线，如下图所示：



3.10 配置系统

VSP 通过一套 Kconfig 系统来集中配置各模块的参数，如下图：



配置系统主要包含三部分：

- ✧ 硬件的选择，包括芯片、主板和麦克风板；
- ✧ 三个域的配置：MCU、DSP 和 CPU；

- VSP 的配置，包括麦克风阵列的配置、循环缓冲区的配置、LED 环的配置等等。

配置系统会生成两个文件：

- .config：用于 Makefile 系统
- autoconf.h：用于 C 代码

以下是几项关键的配置项：

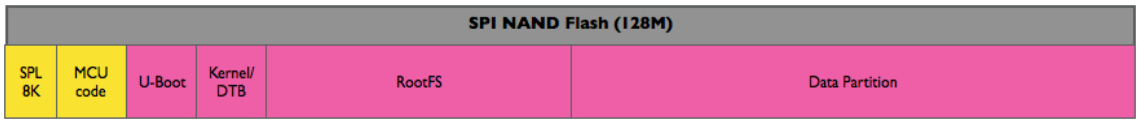
- DSP_SRAM_SIZE_KB：用于控制 DSP 占用 SRAM 的内存大小，也就是划分 MCU 和 DSP 在 SRAM 的边界。这个配置项会控制 MCU 和 DSP 的链接脚本；
- VSP_FRAME_NUM_PER_CONTEXT：用于控制每个 Context 中的帧数；
- VSP_SRAM_FRAME_NUM_PER_CHANNEL：用于控制待机模式下环境音循环缓冲区的大小；
- VSP_SRAM_CONTEXT_NUM：用于控制待机模式下 Context 循环缓冲区的大小；
- VSP_DDR_FRAME_NUM_PER_CHANNEL：用于控制激活模式下环境音和参考音缓冲区的大小；
- VSP_DDR_CONTEXT_NUM：用于控制激活模式下 Context 循环缓冲区的大小；

3.11 软件部署

3.11.1 Soc 模式

对于智能音箱、玩具机器人的产品推荐采用单颗 SPI NAND 部署所有的软件；而对于开发板，推荐采用 SPI NOR + SD Card 方式部署软件。

3.11.1.1 SPI NAND



如图，SPI NAND Flash 的前半段存放 VSP 的启动部分，后半段存放 CPU 的 Linux 系统。VSP 工作所需的 Binary 都存放在根文件系统。这种方式的优点是：

- 启动系统基本上是不需要更新的，而需要更新的部分集中在根文件系统中，这样就不必因为软件升级而影响 Bootloader，增加系统的稳定性；

3.11.2 SPI NOR + SD Card

如图，SPI NOR Flash 存放 VSP 的启动部分，Linux 系统都存放在 SD Card 上。这种方式的优点是：

- 不需要经常更新的启动系统，存放在 SPI Nor Flash 中，增加系统的稳定性；
- 我们很容易通过 U-Boot 的设置切换到网络启动 Linux 系统，非常适合开发阶段。
- 如果要更新离线工作的 Linux 系统，还可以通过取出 SD / TF 卡来完成。

3.11.3 USB 声卡或算法模块

对于 USB 声卡或算法模块，则可以根据是否需要捆绑算法 Binary，带有或不带有 SPI NOR Flash。

这部分的设计暂定。

3.12 算法移植

算法移植的一般过程是：

- ✧ 实现 Process（VSP_CONTEXT *context）函数；
- ✧ 在 PC 上模拟效果和做性能评估；
- ✧ 做算法定点化，并使用 NatureDSP 库优化关键路径的性能，直到性能满足要求。指标一般是对 10ms 数据，处理时间低于 10ms；
- ✧ 最后将算法集成到 VSP 中，并调整 MCU 代码；

3.12.1 参数设定

要设定的参数有：

- ✧ 采样率（SampleRate），只有 16K 和 8K 选择，一般选择为 16K；
- ✧ 帧长（FrameLength）：目前选择项只有 10ms 每帧；
- ✧ 每个 CONTEXT 的帧数量（FrameNumPerContext）；
- ✧ 每个通道上帧 Buffer 的数量（FrameNumPerChannel）；

为了保证每个 Context 处理时，帧数据都在连续的空间内，而不需要频繁判断边界，减少流水线被中断的损耗，对上述参数的一般要求是：

- ✧ FrameNumPerChannel 是 FrameNumPerContext 的整数倍；
- ✧ $\frac{SampleRate \times FrameLength}{1000}$×FrameNumPerChannel×2 必须是 128 的倍数；

3.12.2 性能评估

由于 GX8010 的 DSP 没有硬件 Profile 功能，一般情况下，性能评估需要在 PC 上完成。在 dsp/example/tb 下有一个性能评估项目，可以利用它做性能评估。

<缺性能评估结果图>

3.13 神经网络处理

暂未实现

3.14 知识产权保护

暂未实现

4 VSP 开发环境

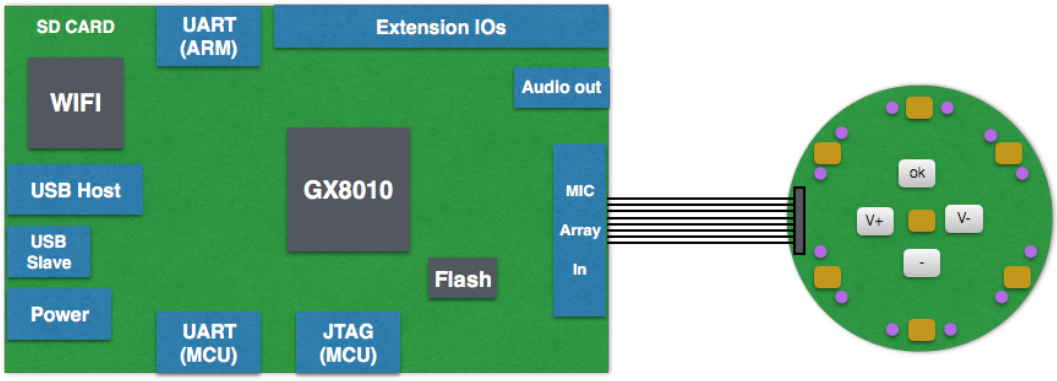
4.1 算法开发板

算法工程师可在 GX8010 EVB 评估板上开发算法，它具有如下的特点：

- ✧ 6+1 路圆形麦克风阵列，信噪比 > 65dB，灵敏度 > -26dBV @ 94dB 1KHz；
- ✧ 12 颗三色 LED 环，可调整亮度和色彩；
- ✧ 4 个按钮开关；
- ✧ 板载 802.11 b/g/n Wi-Fi 模块（SDIO 接口）和 3dBi 陶瓷天线，支持 AirKiss；
- ✧ USB Host 接口，可通过 USB Hub 连接更多的 USB 设备；
- ✧ USB Device 接口，可实现 UAC、UVC、MS、PTP 等 USB 设备；
- ✧ 8MB SPI NOR Flash（可替换成 128MB SPI NAND Flash）；
- ✧ TF 卡插槽；
- ✧ 2 路 USB UART 调试接口，连接 MCU 和 ARM；
- ✧ 40 脚扩展接口（兼容树莓派扩展板）；
- ✧ 尺寸与名片相仿（85mm x 56mm，双层 PCB）；
- ✧ 支持 USB 供电；

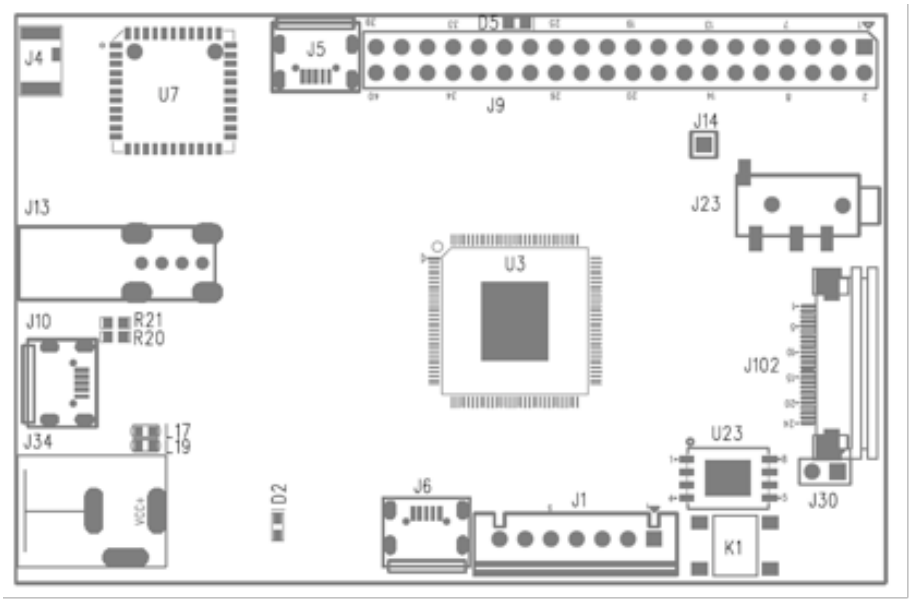
4.1.1 开发板框架图

GX8010 EVB 开发板由主板和麦克风板构成。主板上有主芯片、Flash、Wifi 三个主要器件，另外有 USB Master、USB Slave、两个 UART、Audio In 和 Audio Out，以及其它扩展接口。麦克风板由一个 6+1 的 PDM 麦克风阵列构成，每个麦克风同时配置了 2 个彩色 LED 灯，同时麦克风板上设置了 4 个按键。

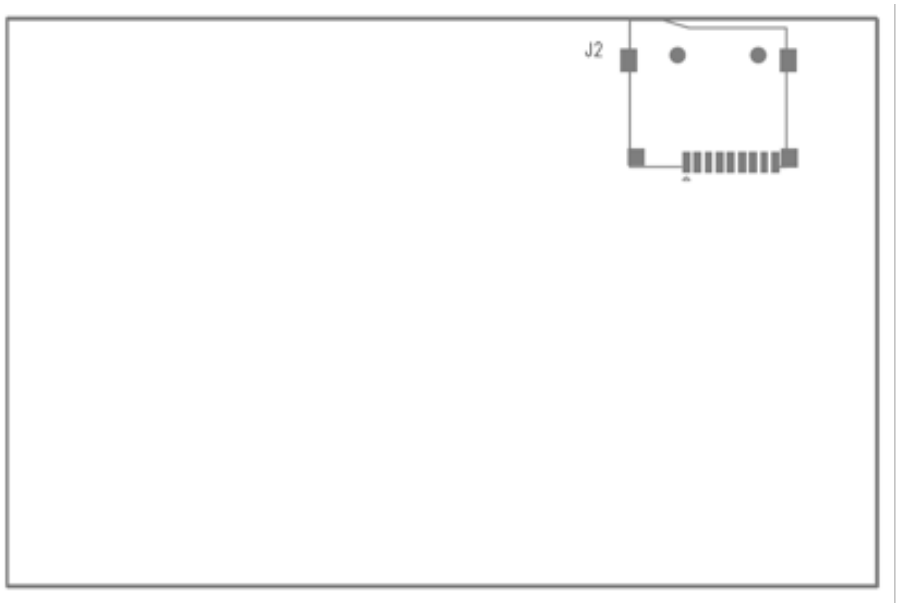


4.1.2 接口位置

正面：



反面



接口说明如下表：

位号	说明
J1	MCU JTAG 接口
J2	TF 卡插座
J4	Wi-Fi 陶瓷天线
J5	ARM 调试串口（板载 USB UART）
J6	MCU 调试串口（板载 USB UART）
J9	扩展接口（40 脚，兼容树莓派扩展板）

位号	说明
J10	USB Device
J13	USB Host
J14	PCM OUT 的 MCLK 的测试脚
J23	3.5mm 音频输出接口
J30	静音跳线
J34	DC 5V 输入
D2	电源输入指示
D5	功能指示
K1	复位按钮

4.1.3 *设置硬件环境*

如图设置开发环境：

所需设备如下：

- ✧ 安装 Ubuntu 14.04/16.04 的电脑；
- ✧ 开发板套件，包括评估板、USB 以太网转换器、电源转换器、TF 卡；
- ✧ 路由器（可选），推荐在一个局域网络里进行开发；
- ✧ 网线若干；

4.2 软件系统

4.2.1 *操作系统*

操作系统宜选择 Ubuntu 14.04 LTS 或 16.04 LTS，其他版本没有测试过。

如果选择安装 64 位版 Ubuntu，则需要安装 32 位支持包：

```
~$ sudo dpkg --add-architecture i386
~$ sudo apt-get update
~$ sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
libx11-6:i386 gtk2-engines:i386 lib32ncurses5 lib32z1 libxtst6:i386
libgtk2.0-0:i386 lib32ncurses5 libcanberra-gtk3-0:i386
```

4.2.2 *安装 MCU 工具链*

MCU 工具链文件有：

- ✧ csky-abiv2-elf-tools-i386-minilibc-20160308.tar.bz2：MCU 的工具链；
- ✧ boot 和 gxscpu.bin：MCU 的 Flash 下载工具；

假设 csky-abiv2-elf-tools-i386-minilibc-20160308.tar.bz2 存放在~/toolchain 目录下。

```
~$ cd /opt
~$ sudo tar xvf ~/toolchain/csky-abiv2-elf-tools-i386-minilibc-
20160308.tar.bz2
~$ mkdir -p ~/bin
~$ cp ~/toolchain/boot ~/bin
```

然后需要编辑.profile，将/opt/csky-abiv2-elf/bin 加入到 PATH 路径中。安装完毕后确认可以使用 csky-abiv2-elf-gcc 即可。

4.2.3 *安装 DSP 工具链*

DSP 的工具链文件有：

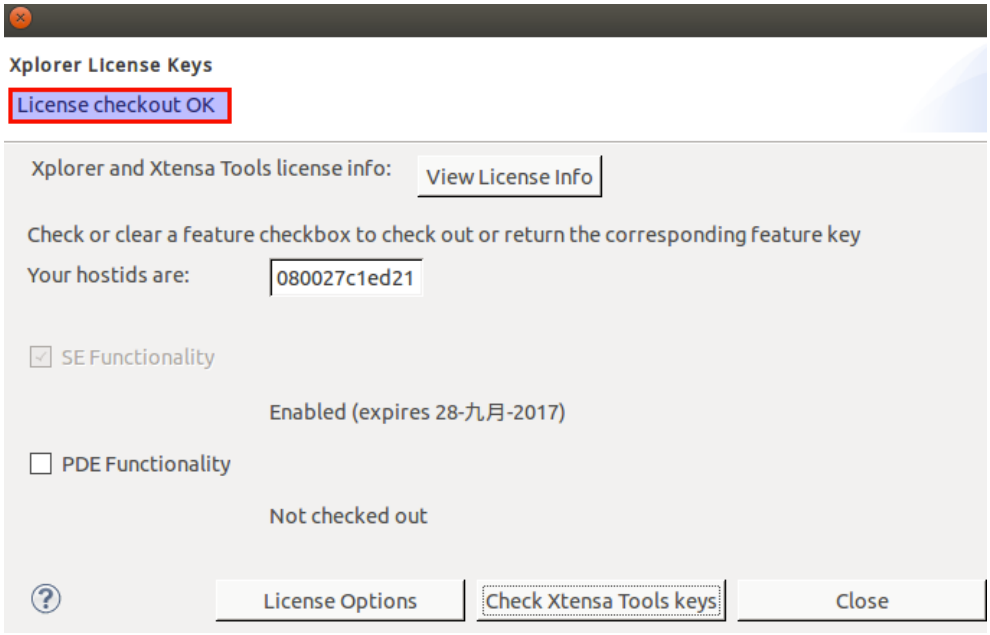
- ✧ Xplorer-6.0.4-linux-installer.bin：DSP 的开发工具；
- ✧ Hifi4_bd7_20161020_M_linux_redist.tgz：DSP 的 Core；

启动 Xplorer 安装程序：

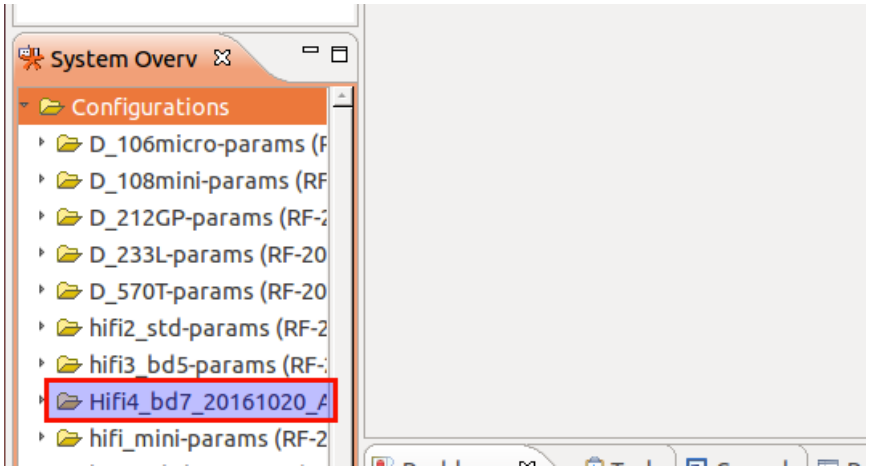
```
~$ chmod +x ~/toolchain/Xplorer-6.0.4-linux-installer.bin
~$ ~/toolchain/Xplorer-6.0.4-linux-installer.bin
```

请按默认提示安装，尤其是安装目录需要设定为~/xtensa。

Xplorer 安装完毕后需要设置 License，设置步骤是：点击菜单栏的 "Help" - "Xplorer License Keys"，在弹出的对话框上点击"Install Software Keys"，输入 License Address（请向 FAE 咨询），然后点击 Finish。



然后需要添加 Core，方法是：右键点击“System Overview”窗口中的“Configurations”，选择“Find and Install a Configuration Build”。在弹出的窗口中点击“Browser”并选择 configuration 文件（Hifi4_bd7_20161020_M_linux_redist.tgz），点击"OK"。



最后要将~/xtensa/XtDevTools/install/tools/RF-2016.4-linux/XtensaTools/bin/加入到 PATH 环境变量，安装完毕后可在命令行中测试一下：

```
~$ xt-xcc --xtensa-core=Hifi4_bd7_20161020_M -v

xt-xcc version 11.0.4
Thread model: single
```

✧ 安装 CPU 工具链

CPU 工具链可以使用 Ubuntu 自带的 ARM 工具链，安装方法如下：

```
~$ sudo apt-get install gcc-arm-linux-gnueabihf
```

4.2.4 *编译 VSP*

编译 VSP 的一般步骤是：

```
~$ cd ~/vsp
~$ make menuconfig
~$ make
```

编译后能产生以下文件：

- ✧ mcu.boot：bootloader，可通过 boot 命令下载到 Flash 中。
- ✧ mcu.bin：VSP 的 MCU 部分；
- ✧ dsp.bin：VSP 的 DSP 部分；
- ✧ vsp.ko：VSP 的 CPU 部分；

4.2.5 *安装调试环境*

调试环境包括 TFTP 服务器和 NFS 服务器，安装步骤如下：

- ✧ 安装 TFTP 服务器

```
~$ sudo apt-get install tftpd-hpa
~$ sudo vim /etc/default/tftpd-hpa
修改内容如下：
TFTP_USERNAME="tftp"
TFTP_ADDRESS="0.0.0.0:69"
TFTP_DIRECTORY="/opt/tftpboot"
TFTP_OPTIONS="-l -c -s"
~$ sudo service tftpd-hpa restart
```

在/opt/tftpboot 目录下存放 zImage 和 leo_robot.dtb。

- ✧ 安装 NFS 服务器

```
~$ sudo apt-get install nfs-kernel-server
~$ sudo vim /etc/exports
添加内容如下：
/opt/nfs *(rw,sync,no_all_squash,no_subtree_check,no_root_squash)
~$ sudo service nfs-kernel-server restart
```

在/opt/nfs 目录下存放根文件系统

✧ 从 TFTP 和 NFS 启动系统

打开 CPU 侧的终端，上电启动后，敲回车阻止 U-Boot 从 SD 卡中加载系统。然后修改 ipaddr 和 serverip 两个环境变量，然后运行 run bootnfs 启动系统。

5 附录

5.1 MCU API 说明

待定

5.2 DSP API 说明

待定

5.3 常见问题

待定