

Secret Status: Top Secret () Secret () Restrict () Public ()

RKNanoD Software Development Kit Guide

(Version 1.3)

Product R&D Dept.III
Rockchip Electronics Co.,Ltd

www.rock-chips.com

Status: <input type="checkbox"/> development <input checked="" type="checkbox"/> public <input type="checkbox"/> modify	File Tag:	Software development kit guide	
	Version:	1.3	
	Author:	Ma Longchang	
	Data:	2016-5-16	
	Verify:	Zheng Yongzhi	2016-4-15

Version History

Version	Author	Date	Remarks
1.0	Ma Longchang	2016-4-15	Original
1.1	Ma Longchang	2016-5-16	Update
1.2	Ma Longchang	2016-08-03	Update
1.3	Ma Longchang	2016-10-08	Update

目录

1	简述	5
1.1	SDK 概述	5
1.2	设计目标	5
2	支持的特性	6
3	目录结构	7
4	SDK 软件配置	9
4.1	调试配置	9
4.2	存储设备选择配置	9
4.3	应用模块选择	10
5	SDK 双核交互	12
5.1	双核结构	12
5.2	Mailbox 简介	13
5.3	Mailbox 的使用	13
5.4	双核交互	14
5.4.1	解码控制交互命令	14
5.4.2	文件数据流交互命令	19
6	如何增加新的解码库	26
6.1	添加常规格式音频解码	26
6.2	添加用户支持的音频解码	33
7	工程编译	40
8	生成固件与烧写固件	42
8.1	EMMC Flash 固件生成工具	42
8.2	SPI NOR Flash 固件生成工具	43
8.3	烧写固件	44
8.3.1	单机烧写工具	44
8.3.2	工厂量产烧写工具	45
8.3.3	生成本地升级固件	46

9	资源打包方法.....	48
9.1	图片资源打包.....	48
9.2	字符串资源打包.....	49
10	BT 的配置.....	52
11.1	硬件介绍.....	52
11.2	软件配置.....	54
11.3	操作说明.....	56

1 简述

1.1 SDK 概述

RKNanoD MP3 SDK 是一个基于 RKNanoD 双核 Cortex-M3 芯片的多媒体音频软件开发包。该 SDK 包含了音频解码，编码，图片，FM，电子书等功能。支持 24bit HiFi 无损格式音频解码以及 16bit/24bit 高质量录音功能。

1.2 设计目标

RKNanoD MP3 SDK 是为了解决广大 HiFi 音频播放的应用需求，提供一个开发周期短、可靠性高、品质优良的产品解决方案。

2 支持的特性

RKNanoD SDK 主要是用作为 HIFI Mp3 播放器使用。目前支持的音频格式有：MP3、WAV、AAC、OGG、FLAC、APE、HIFI FLAC、HIFI APE、HIFI ALAC、视频播放器（AVI）、FM 收音机、WAV 编码（录音）、MP3 编码（录音）、电子书（txt）等。详见特征见下表。

特性	详细特征	备注
显示	64K Pixel Color LCD	Max to 320x240 dot(160×128/320×240)
USB	FS/HS Device 2.0	
USB 协议	MSC	
存储	EMMC Flash	Max to 256GB
	SPI Flash	
	SD Card	Max to 256GB
音频解码器	16 bits	MP3 / WAV / APE / FLAC / AAC / OGG
	24 bits	MP3 / WAV / AAC / OGG HIFI APE / HIFI FLAC / HIFI ALAC
EQ	Effect Sound	
媒体库	Title/Artists / Albums / Genres	Max to 8192 Songs
ID3	ID3V1 / ID3V2.3 / ID3V2.4 / Album art	Album art only support ID3JPEG
视频	JPEG / BMP / Video	File need convert to AVI
收音机	FM Radio	China: 87.0MHz – 108.0MHz Japan: 76.0MHz – 108.0MHz
录音	WAV Encode	8KHz~192KHz, 16bit/24bit
	MP3 Encode	8KHz~48KHz, 16bit
电子书	*.txt file / Book Mark	

3 目录结构

RKNanoD MP3 SDK 目录结构分为：

Common: 包括 OS、驱动、Fat32 文件系统、音频、视频解码库等底层开发包。

SDK_160_128: 资源、脚本、工程目录、固件存放以及应用相关等。

下面表格中将简单描述一下各个不同目录的用途。

目录名	子目录	子目录	详细描述
Common	BBsystem		B Core 解码相关的接口文件
	BootLoader		启动代码，用来 JTAG 调试
	Codec	Audio	音频解码器相关的解码库与 API 接口文件
		Image	图片解码相关的解码库与 API 接口文件
		Radio	FM 相关控制的接口文件
		UsbControl	USB 控制相关 API 接口文件
		Video	视频解码相关的解码库与 API 接口文件
	Display		与显示相关的所有接口文件
	Driver		各个不同外设驱动接口文件
	FileSys		Fat32 文件系统相关的接口文件
	FwUpdate		执行本地固件升级相关的接口文件
	Include		全局相关的头文件
	Plug		
	SortFileInfo		媒体库相关的排序接口文件
	System	Debug	调试接口
		FileSeek	文件搜索相关
		ModuleOverlay	模块调度相关
		Os	系统相关
		SysService	外设检测等系统服务相关代码

SDK_160_128	Build		编译工程目录
	Development		固件生成与烧写相关的工具
	Firmware		用于存放生成的下载到设备的固件
	Resource		存放图片与字符串生成工具及生成后文件
	Scatter		存放设备内存组织的脚本文件
	UI	Browser	浏览设备中目录结构的应用接口文件
		ChargeWin	充电相关的 UI 接口文件
		Dialog	存放用来在 LCD 上显示对话框的接口文件
		MainMenu	选择主应用模块的接口文件
		Medialib	媒体库相关的接口文件
		MusicWin	音乐播放器相关的应用 UI 接口文件
		PicWin	图片摸相关的应用 UI 接口文件
		RadioWin	FM 应用 UI 接口文件
		RecordWin	录音应用 UI 接口文件
		SetMenu	系统设置应用 UI 接口文件
		TextWin	电子书应用 UI 接口文件
		USB	USB 应用相关的 UI 接口文件
		VideoWin	视频播放器应用相关的 UI 接口

4 SDK 软件配置

RKNanoD MP3 SDK 支持可配置的软件应用。在“SDK_160_128/sysconfig.h”中详细列出了软件可控制的软件配置选项。用户可根据自身开发产品需求增减选择所需的功能应用。详细描述如下：

4.1 调试配置

```
/*
*-----*
*-----*           Debug Config
*-----*
*/
//debug compile switch.
#define _LOG_DEBUG_

#ifndef _LOG_DEBUG_
#define _UART_DEBUG_
//#define _BB_DEBUG_
//#define _FILE_DEBUG_
#endif

//#define USB_SERIAL_DEBUG
#ifndef USB_SERIAL_DEBUG
#define DEBUG_UART_PORT 1
#else
#define DEBUG_UART_PORT 0
#endif
```

Debug 调试配置，默认使用 uart 串口调试，串口调试中默认使用 uart 1 作为调试口。如上图，定义 DEBUG_UART_PORT 为 1。

其中需要注意：如果想要显示解码库中的 log，需要打开_BB_DEBUG_ 宏，这样解码相关的 log 将会打印出来。

4.2 存储设备选择配置

如下图，根据设备硬件的情况选择 Flash 配置。

1. 如果使用 EMMC Flash，打开_EMNC_宏定义。如果使用 SPI NOR Flash 打开_SPINOR_ 宏定义，二者只能选择其一。目前不支持 NAND Flash。SD 卡的支持可根据用户产品定义选择支持与否。
2. 如果选择了使用 EMMC Flash，可选择定义固件存放的位置，默认存放在 EMMC Flash 中，

不容许修改。用户可选择定义资源文件的存放位置，默认存放在固件中。

3. 如果选择了使用 SPI NOR Flash，用户可选择固件存放的位置，默认存放在 SPI Nor Flash 中，不容许修改。需要根据 SPI NOR Flash 的存储容量大小，用户可选择定义资源文件的存放位置，如果 SPI NOR Flash 容量比较大（大于 8M）可选择将资源文件与固件打包放在 Flash 中，否则只能放在文件系统中（SD 卡中）。
4. 此处 SDCARD_PORT 的定义根据用户自身硬件设计的不同选择不同的定义，如果硬件设计中将 SDCARD 的部分与 EMMC 电路共用，需要定义 SDCARD_PORT 为 1，否则使用默认值 0。

```
-----  
* Memory Device Option  
* Select which memory device are used  
*-----  
*/  
//#define _SPINOR_           //SPI NOR Flash Support  
#define _EMMC_               //EMMC Flash Support  
#define _SDCARD_              //SD CARD Support  
  
#ifdef _EMMC_                //eMMC boot  
#define FW_IN_DEV            3 //Firmware Stored in: 1:nandflash 2:sipnor 3:emmc 4:sd card  
#define RES_IN_DEV            1 //Resources Stored in: 1 FW 2: filesystem  
#define SDCARD_PORT            0 //SDCard Port Select: 0 SDIO 1: eMMC  
#endif  
  
#ifdef _SPINOR_              //SPI nor boot  
#define FW_IN_DEV            2 //Firmware Stored in: 1:nandflash 2:sipnor 3:emmc 4:sd card  
#define RES_IN_DEV            2 //Resources Stored in: 1 FW 2: filesystem; if SPI flash is large enough to stor  
#define SDCARD_PORT            0 //SDCard Port Select: 0 SDIO 1: eMMC  
#endif  
  
#ifdef _SDCARD_              //SD Card boot  
//#define FW_IN_DEV            4 //Firmware Stored in: 1:nandflash 2:sipnor 3:emmc 4:sd card  
//#define RES_IN_DEV            1 //Resources Stored in: 1 FW 2: filesystem  
#endif  
  
//#define DISK_VOLUME  
//#define _MULT_DISK_  
#define ENABLE_MBR
```

4.3 应用模块选择

如下图，定义了 NanoD SDK 中支持的应用模块。用户可根据自身需求现在打开或关闭。

```
/*
*-----*
*-----*
*----- Application Modules Option -----
*-----*
#define _MUSIC_
#define _RADIO_
#define _RECORD_
#define _BROWSER_
#define _SYSSET_
#define _VIDEO_
#define _PICTURE_
#define _EBOOK_
#define _USB_

#ifndef _SPINOR_
#define _MEDIA_MODULE_
#endif

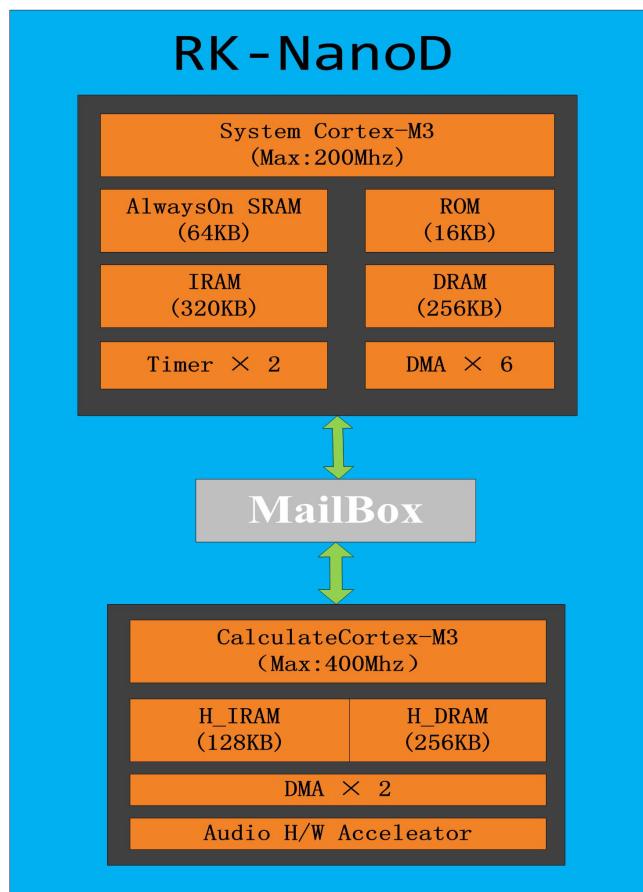
#define _WATCH_DOG_
```

5 SDK 双核交互

RKNanoD 是一款基于 ARM cortex-M3 的双核处理器。其中 system core (A 核) 为主处理器，芯片上电后首先启动系统核，加载系统软件运行；calculate core (B 核) 为从处理器，由主处理器通过软件进行启动和关闭。在 RKNanoD MP3 SDK 中，system core 用以运行系统管理、调度以及应用软件，calculate core 作为从处理器进行音频解码、编码等算法运算处理。A、B 核之间通过 mailbox 进行命令交互。RKNanoD 为 A、B 核分配了可独立支配的使用内存，同时也支持 A、B 核相互访问相应的内存、外设等接口。

5.1 双核结构

RK-NanoD 的双核架构如下面的框图。该框图简单描述了 A 核作为系统核，B 核作为运算核，以及 A-B 核之间通过 Mailbox 作为桥梁通讯的过程。

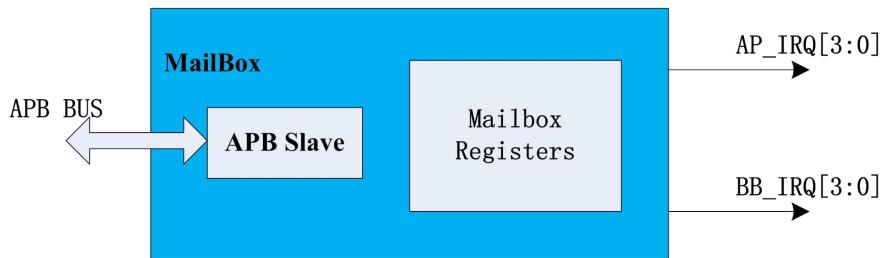


5.2 Mailbox 简介

Mailbox 是一个简单的 APB 外设模块，它通过写命令操作产生一个中断在双核 CPU 之间进行通讯。Mailbox 的寄存器可以被两个 CPU 通过 APB 接口访问。

Mailbox 有以下特点：

- 支持双核访问
- 支持 4 个通道，每一个通道包括了一个命令字，一个数组字寄存器和一位标识中断的标志位
- System core 的 4 个中断，用以接收 calculate core 发出的请求
- Calculate core 中有 4 个中断，用以接收 system core 发出的请求



Mailbox 内部模块图

5.3 Mailbox 的使用

RKNanoD 中有一个 mailbox，该 mailbox 有 4 个通道，每个通道都有独立可设的中断，每个通道的定义如下代码所定义的：

```
#define MAILBOX_INT_0 (uint32)(1 << 0)
#define MAILBOX_INT_1 (uint32)(1 << 1)
#define MAILBOX_INT_2 (uint32)(1 << 2)
#define MAILBOX_INT_3 (uint32)(1 << 3)

#define MAILBOX_CHANNEL_0 (0)
#define MAILBOX_CHANNEL_1 (1)
#define MAILBOX_CHANNEL_2 (2)
#define MAILBOX_CHANNEL_3 (3)

#define MAILBOX_ID_0 (0)
```

下面表格中详述了 Mailbox 每个通道的使用用途：

通道号	用途	备注
0	B 核启动以及 A 核调整频率通知 B 核时使用的通道	
1	音量解码中，A 核与 B 核命令与数据交互时使用的通道	
2	文件操作时，A 核与 B 核命令与数据操作交互时使用的通道	
3	调试 B 核代码时使用的通道。主要是用来 debug 使用。	B 核的打印输出是将打印的内容通过该通道发送到 A 核，A 核托管来打印输出的。

接口说明：

接口	说明
MailBoxWriteB2ACmd	B 核向 A 核写命令
MailBoxWriteB2AData	B 核向 A 核写数据
MailBoxWriteA2BCmd	A 核向 B 核写命令
MailBoxWriteA2BData	A 核向 B 核写数据
MailBoxReadB2ACmd	B 核读取 A 核的命令
MailBoxReadB2AData	B 核读取 A 核的数据
MailBoxReadA2BCmd	A 核读取 B 核的数据
MailBoxReadA2BData	A 核读取 B 核的数据

注意：Mailbox 的命令发送必须与数据一起发送，不能只发送命令或数据中的一种。

5.4 双核交互

本章节主要描述 A-B 核通过 Mailbox 相互交互的过程，将以音频解码为例，剖析 A-B 核通过 mailbox 相互通信，将解码命令与解码数据传递，完成整个解码的控制与交互。

5.4.1 解码控制交互命令

- 解码交互命令

首先看一下 Audio_main.h 中定义的音频解码使用到的解码命令。

```
typedef enum _MEDIA_MSGBOX_DECODE_CMD
{
    MEDIA_MSGBOX_CMD_DECODE_NULL,           /*请求音频codec打开操作*/
    MEDIA_MSGBOX_CMD_DEC_OPEN,              /*解析音频文件头信息出错, 不支持该文件*/
    MEDIA_MSGBOX_CMD_DEC_OPEN_CMPL,         /*解析音频文件头信息成功*/

    MEDIA_MSGBOX_CMD_DECODE,                /*请求解码操作*/
    MEDIA_MSGBOX_CMD_DECODE_CMPL,           /*完成一次解码操作*/
    MEDIA_MSGBOX_CMD_DECODE_ERR,            /*解码出错*/

    MEDIA_MSGBOX_CMD_DECODE_GETBUFFER,       /*请求解码数据*/
    MEDIA_MSGBOX_CMD_DECODE_GETBUFFER_CMPL, /*解码数据获取完成*/

    MEDIA_MSGBOX_CMD_DECODE_GETTIME,         /*请求获取当前解码时间*/
    MEDIA_MSGBOX_CMD_DECODE_GETTIME_CMPL,   /*当前时间获取完成*/

    MEDIA_MSGBOX_CMD_DECODE_SEEK,            /*指定位置解码*/
    MEDIA_MSGBOX_CMD_DECODE_SEEK_CMPL,      /*指定位置完成*/

    MEDIA_MSGBOX_CMD_DECODE_CLOSE,           /*请求关闭解码器*/
    MEDIA_MSGBOX_CMD_DECODE_CLOSE_CMPL,     /*解码器关闭完成*/

    MEDIA_MSGBOX_CMD_FLAC_SEEKFAST,          /*请求FLAC快速定位*/
    MEDIA_MSGBOX_CMD_FLAC_SEEKFAST_CMPL,    /*FLAC快速定位完成*/

    MEDIA_MSGBOX_CMD_FLAC_GETSEEK_INFO,      /*请求FLAC快速定位信息*/
    MEDIA_MSGBOX_CMD_FLAC_SEEKFAST_INFO_CMPL,/*FLAC快速定位信息完成*/

    MEDIA_MSGBOX_CMD_DECODE_NUM
};
```

解码命令

具体命令的解释如下表：

命令 (MEDIA_MSGBOX_CMD_*)	说明	方向 (命令发送的方向 A→B 或 B→A)
DECODE_NULL	空命令	命令字不能为“0”
DEC_OPEN	打开解码器	A → B
DEC_OPEN_ERR	打开解码失败	B → A
DEC_OPEN_CMPL	打开解码成功	B → A
DECODE	请求 B 核解码	A → B
DECODE_CMPL	解码完成	B → A
DECODE_ERR	解码出错	B → A
DECODE_GETBUFFER	请求 B 核解码数据	A → B
DECODE_GETBUFFER_CMPL	解码数据获取完成	B → A
DECODE_GETTIME	请求获取当前解码时间	A → B
DECODE_GETTIME_CMPL	当前时间获取完成	B → A
DECODE_SEEK	指定位置解码	A → B
DECODE_SEEK_CMPL	指定位置完成	B → A

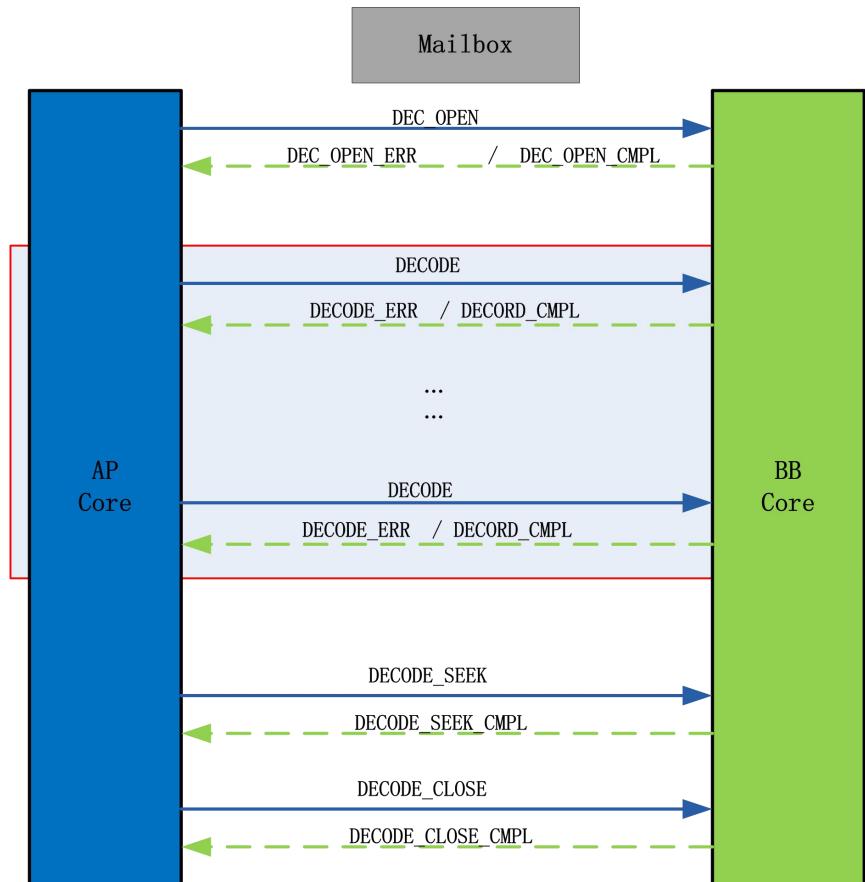
DECODE_CLOSE	请求解码关闭	A → B
DECODE_CLOSE_CMPL	解码关闭完成	B → A

A-B 核的解码命令交互

如上表所述，A 核向 B 核通过 Mailbox 发送解码相关的各种请求命令，B 核负责解码运算完成之后给出结果，同时将结果和状态通过 Mailbox 发回给 A 核，这样一问一答的交互方式就是通过 Mailbox 为桥梁完成。

- 解码交互过程

详细的 A-B 核交互过程如下图所示：



音频解码时 A-B 核通过 mailbox 的实际交互过程

注意：

从上图中可以看到，在 RKNanoD MP3 SDK 中，没有获取时间的请求与回复，也没有解码 buffer 数据的请求与回复，这是因为获取时间的操作和解码后的数据 buffer 获得，均放在了

DECODE 的命令交互中，每完成一次解码，B 核就将时间和解码的数据 buffer 存放在结构体中，同 mailbox 的 data 命令发送给 A 核。具体代码可以在 main2.c 中看到。如下：

```
00757:         case AUDIO_DECODE_DECODE:
00758:         {
00759:             //bb_printf1("AUDIO_DECODE_DECODE \n");
00760:             pcb.audio_decode_status = AUDIO_DECODE_IDLE;
00761:
00762:             if (GetMsg (MSG_AUDIO_DECODE_FILL_BUFFER))
00763:             {
00764:                 AudioFileInput2 (pRawFileCache);
00765:             }
00766:
00767:             if (1 != CodecDecode2 ())
00768:             {
00769:                 MailBoxWriteB2ACmd (MEDIA_MSGBOX_CMD_DECODE_ERR, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
00770:                 MailBoxWriteB2AData (0, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
00771:                 break;
00772:             }
00773:
00774: #ifdef MP3_DEC_INCLUDE2
00775:             if (CurrentCodec2 == CODEC_MP3_DEC)
00776:             {
00777:                 mp3_wait_synth();
00778:             }
00779: #endif
00780:
00781:             CodecGetTime2 (&gMediaBlockInfo.CurrentPlayTime);
00782:
00783:             CodecGetCaptureBuffer2 ((short*)&gMediaBlockInfo.Outptr, &gMediaBlockInfo.OutLength);
00784:
00785:             MailBoxWriteB2ACmd (MEDIA_MSGBOX_CMD_DECODE_CMPL, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
00786:             MailBoxWriteB2AData ((UINT32)&gMediaBlockInfo, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
00787:         }
```

B 核解码过程

- 解码交互的示例代码

打开解码器（A 核部分）：CodecOpen 中调用不同解码器的 SUBFN_CODEC_OPEN_DEC 参数。此处通过 mailbox 发送向 B 核发送 MEDIA_MSGBOX_CMD_DEC_OPEN 命令。

```
_ATTR_AUDIO_TEXT_
unsigned long CodecOpen (unsigned long ulCodec, unsigned long ulFlags)
{
    unsigned long ulRet;

    if (CurrentCodec == 0xff)
        return -1;

    // Pass the open request to the entry point for the codec.
    ulRet = (CodecPfn[CurrentCodec]) (SUBFN_CODEC_OPEN_DEC, 0, 0, 0);

    // Return the result to the caller.
    return (ulRet);
}
```

```
case SUBFN_CODEC_OPEN_DEC:
{
    DEBUG("gDecDone = %d ", gDecDone);

    gOpenDone = 0;
    MailBoxWriteA2BCmd(MEDIA_MSGBOX_CMD_DEC_OPEN, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
    MailBoxWriteA2BData(0, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
    timeout = 20000000;

    while (!gOpenDone)
    {
        WatchDogReload();
        BBDebug();
        // _WFI();
        DelayUs(1);

        if (--timeout == 0)
        {
            DEBUG("SUBFN_CODEC_OPEN_DEC: timeout!!!!");
            break;
        }
    }

    gOpenDone = 0;

    if (gError) //codec decode open error
        return 0;
    else
        return (1);
}
```

打开解码器（B 核部分）：mailbox 收到 A 核发送的命令后产生中断，B 核代码根据接收命令处理相应流程。

B 核 mailbox 中断服务程序：

```
_irq void MailBoxDecService()
{
    uint32 cmd;
    uint32 data;
    MailBoxClearA2BInt(MAILBOX_ID_0, MAILBOX_INT_1);

    cmd = MailBoxReadA2BCmd(MAILBOX_ID_0, MAILBOX_CHANNEL_1);
    data = MailBoxReadA2BData(MAILBOX_ID_0, MAILBOX_CHANNEL_1);

    switch (cmd)
    {
        case MEDIA_MSGBOX_CMD_DEC_OPEN:
            pcb.audio_decode_status = AUDIO_DECODE_OPEN;
            break;

        case MEDIA_MSGBOX_CMD_DECODE:
            pcb.audio_decode_status = AUDIO_DECODE_DECODE;
            break;

        case MEDIA_MSGBOX_CMD_DECODE_GETBUFFER:
            pcb.audio_decode_status = AUDIO_DECODE_GETBUFFER;
            break;

        case MEDIA_MSGBOX_CMD_DECODE_GETTIME:
            pcb.audio_decode_status = AUDIO_DECODE_GETTIME;
            break;

        case MEDIA_MSGBOX_CMD_DECODE_SEEK:
            pcb.audio_decode_status = AUDIO_DECODE_SEEK;
            pcb.audio_decode_param = data;
    }
}
```

B 核打开解码器出错处理：

```
case AUDIO_DECODE_OPEN:
{
    pcb.audio_decode_status = AUDIO_DECODE_IDLE;

    if (1 != CodecOpen2(0, CODEC_OPEN_DECODE))
    {
        bb_printf1("###AUDIO_DECODE_OPEN error!###");
        MailBoxWriteB2ACmd(MEDIA_MSGBOX_CMD_DEC_OPEN_ERR, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
        MailBoxWriteB2AData(0, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
        break;
    }

    {
        int retflag;
        CodecGetSampleRate2(&gMediaBlockInfo.SampleRate);
        CodecGetChannels2(&gMediaBlockInfo.Channel);
        retflag = CodecGetBitrate2(&gMediaBlockInfo.BitRate);
        if (retflag == 0)
        {
            //bb_printf1("BitRate = %d, Invalid bitrate number", gMediaBlockInfo.BitRate);
            bb_printf1("###AUDIO_DECODE_OPEN error!###");
            MailBoxWriteB2ACmd(MEDIA_MSGBOX_CMD_DEC_OPEN_ERR, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
            MailBoxWriteB2AData(0, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
            break;
        }

        if (gMediaBlockInfo.Channel > 2)
        {
            //bb_printf1("Channels = %d, > 2", gMediaBlockInfo.Channel);
            bb_printf1("### channels bigger than 2!###");
            MailBoxWriteB2ACmd(MEDIA_MSGBOX_CMD_DEC_OPEN_ERR, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
            MailBoxWriteB2AData(0, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
            break;
        }
    }
}
```

B 核打开解码器成功处理：

```
CodecGetCaptureBuffer2((short*)&gMediaBlockInfo.Outptr, &gMediaBlockInfo.OutLength);

MailBoxWriteB2ACmd(MEDIA_MSGBOX_CMD_DEC_OPEN_CMPL, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
MailBoxWriteB2AData((UINT32)&gMediaBlockInfo, MAILBOX_ID_0, MAILBOX_CHANNEL_1);
```

其他解码命令的交互类与此类似，这里不做赘述。

5.4.2 文件数据流交互命令

- 文件数据流操作命令

B 核在解码过程中，需要 A 核传输音频压缩数据。A 核从文件系统读取文件数据，通过命令交互方式将音频数据传送给 B 核，B 核在解码过程中对文件进行解析，根据解码需求向 A 核请求信的数据。在文件 file.h 中定义了 A-B 核之间对文件操作的交互命令，这些命令也是通过 mailbox 作为桥梁发送与传递的。

```
typedef enum _MEDIA_MSGBOX_FILE_CMD
{
    MEDIA_MSGBOX_CMD_FILE_NULL = 100,

    //MEDIA_MSGBOX_CMD_FILE_OPEN,
    MEDIA_MSGBOX_CMD_FILE_OPEN_CMPL,
    MEDIA_MSGBOX_CMD_FILE_OPEN_HANDSHK,

    //MEDIA_MSGBOX_CMD_FILE_CREATE,
    MEDIA_MSGBOX_CMD_FILE_CREATE_CMPL,
    MEDIA_MSGBOX_CMD_FILE_CREATE_HANDSHK,

    MEDIA_MSGBOX_CMD_FILE_SEEK,
    MEDIA_MSGBOX_CMD_FILE_SEEK_CMPL,

    MEDIA_MSGBOX_CMD_FILE_READ,
    MEDIA_MSGBOX_CMD_FILE_READ_CMPL,

    MEDIA_MSGBOX_CMD_FILE_WRITE,
    MEDIA_MSGBOX_CMD_FILE_WRITE_CMPL,

    MEDIA_MSGBOX_CMD_FILE_TELL,
    MEDIA_MSGBOX_CMD_FILE_TELL_CMPL,

    MEDIA_MSGBOX_CMD_FILE_GET_LENGTH,
    MEDIA_MSGBOX_CMD_FILE_GET_LENGTH_CMPL,

    MEDIA_MSGBOX_CMD_FILE_CLOSE,
    MEDIA_MSGBOX_CMD_FILE_CLOSE_CMPL,
    MEDIA_MSGBOX_CMD_FILE_CLOSE_HANDSHK,
} ? end MEDIA_MSGBOX_FILE_CMD ? MEDIA_MSGBOX_FILE_CMD;
```

文件命令

具体命令的解释如下表：

命令 (MEDIA_MSGBOX_CMD_*)	说明	方向 (命令发送的方向 A→B 或 B→A)
FILE_NULL	空命令	
FILE_OPEN_CMPL	文件打开完成	A → B
FILE_OPEN_HANDSHK	文件打开完成确认	B → A
FILE_CREATE_CMPL	文件创建完成	A → B
FILE_CREATE_HANDSHK	文件创建确认	B → A
FILE_SEEK	文件搜寻	B → A
FILE_SEEK_CMPL	文件搜寻完成	A → B
FILE_READ	请求文件读取	B → A
FILE_READ_CMPL	文件读取完成	A → B
FILE_WRITE	请求写入文件	B → A

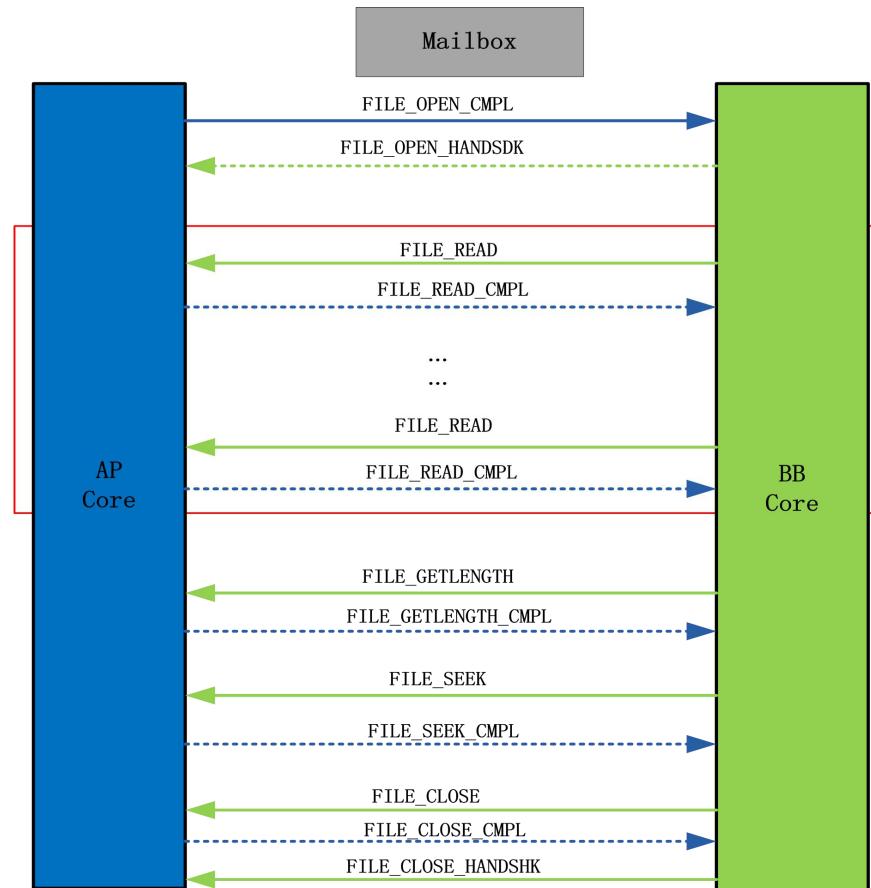
FILE_WRITE_CMPL	文件写入完成	A → B
FILE_TELL	文件当前位置请求	B → A
FILE_TELL_CMPL	文件当前位置完成	A → B
FILE_GETLENGTH	请求文件大小	B → A
FILE_GETLENGTH_CMPL	文件大小获取完成	A → B
FILE_CLOSE	请求文件关闭	B → A
FILE_CLOSE_CMPL	文件关闭完成	A → B
FILE_CLOSE_HANDSHK	文件关闭确认	B → A

A-B 核的文件操作的交互命令

一般文件操作都是 B 核向 A 核发送文件操作请求，A 核接收到 B 核操作请求后做响应处理，再将处理结果回复给 B 核。

- 文件数据流操作交互过程

A、B 核之间通过 mailbox 交互文件操作的过程详见下图：



- 文件操作 A、B 核交互示例代码

这里以打开文件、文件读取为例简单说明 A、B 通过 mailbox 交互的过程。

打开文件交互过程：

打开文件操作 (A 核部分): A 核先向 B 核发送文件打开的命令，以及文件打开后的文件句柄。

```
_ATTR_AUDIO_TEXT_
void AudioWaitBBStart(void)
{
    uint32 timeout = 5000000;

    gACKDone = 0;
    gFileHandle.codecType = CurrentCodec;
    MailBoxWriteA2BCmd(MEDIA_MSGBOX_CMD_FILE_OPEN_CMPL, MAILBOX_ID_0, MAILBOX_CHANNEL_2);
    MailBoxWriteA2BData((int)&gFileHandle, MAILBOX_ID_0, MAILBOX_CHANNEL_2);

    while (!gACKDone)
    {
        WatchDogReload();
        BBDebug();
        //__WFI();
        DelayUs(1);

        if (--timeout == 0)
        {
            DEBUG("AudioWaitBBStart: timeout!!!");
            break;
        }
    }

    gACKDone = 0;
} ? end AudioWaitBBStart ?
```

打开文件 (B 核部分): mailbox 收到 A 核发送的命令后产生中断，B 核代码根据接收命令处理相应流程。

B 核 mailbox 中断服务程序：

```
_irq void MailBoxFileService()
{
    MailBoxClearA2BInt(MAILBOX_ID_0, MAILBOX_INT_2);

    gCmd = MailBoxReadA2BCmd(MAILBOX_ID_0, MAILBOX_CHANNEL_2);
    gData = MailBoxReadA2BData(MAILBOX_ID_0, MAILBOX_CHANNEL_2);

    switch (gCmd)
    {
        case MEDIA_MSGBOX_CMD_FILE_OPEN_CMPL:
            gFileHandle = (FILE_HANDLE_t *)gData;
            pRawfileCache = (FILE *)gFileHandle->handle1;
            FileTotalSize = gFileHandle->filesize;
            CurFileOffset[(uint32)pRawfileCache] = gFileHandle->curfileoffset[0];
            CurrentCodec2 = gFileHandle->codecType;

#ifdef AAC_DEC_INCLUDE
            if (CurrentCodec2 == CODEC_AAC_DEC)
            {
                pAacFileHandleOffset = (FILE *)gFileHandle->handle2;
                CurFileOffset[(uint32)pAacFileHandleOffset] = gFileHandle->curfileoffset[1];
                pAacFileHandleSize = (FILE *)gFileHandle->handle3;
                CurFileOffset[(uint32)pAacFileHandleSize] = gFileHandle->curfileoffset[2];
            }
#endif
            gFileOpStatus = AUDIO_FILE_OPEN_CMPL;
            break;
    }
}
```

B 核打开解码器接收到打开文件命令后，开始进行解码器相关初始化的处理：

```
case AUDIO_FILE_OPEN_CMPL:
{
    gFileOpStatus = AUDIO_FILE_OPT_IDLE;
    AudioIntAndDmaInit2();
    AudioHWInit2();

    AudioCodecGetBufFersize2(CurrentCodec2, FS_44100Hz);
    if(CurrentCodec2 == CODEC_MP3_DEC)
    {
        AudioFileFuncInit2(pRawFileCache, HIFI_AUDIO_BUF_SIZE - 1024);
    }

#ifndef AAC_DEC_INCLUDE
    else if (CurrentCodec2 == CODEC_AAC_DEC)
    {
        ClearMsg(MSG_AUDIO_DECODE_FILL_BUFFER);
        //RKFfileFuncInit2();
        //HifiFileSeek(0, SEEK_SET, pRawFileCache);

        AudioFileMhFuncInit2(pRawFileCache, HIFI_AUDIO_BUF_SIZE - 1024);
        //AudioFileMhSeek2(0, SEEK_SET, pRawFileCache);
    }
#endif
#endif

#ifndef HIFI_ALAC_DECODE
    else if (CurrentCodec2 == CODEC_HIFI_ALAC_DEC)
    {
        ClearMsg(MSG_AUDIO_DECODE_FILL_BUFFER);
        //RKFfileFuncInit2();
        //HifiFileSeek(0, SEEK_SET, pRawFileCache);

        AudioFileMhFuncInit2(pRawFileCache, HIFI_AUDIO_BUF_SIZE - 1024);
        //AudioFileMhSeek2(0, SEEK_SET, pRawFileCache);
    }
#endif
#endif
    else
    {
        AudioFileFuncInit2(pRawFileCache, HIFI_AUDIO_BUF_SIZE - 1024);
    }

    MailBoxWriteB2ACmd(MEDIA_MSGBOX_CMD_FILE_OPEN_HANDSHK, MAILBOX_ID_0, MAILBOX_CHANNEL_2);
    MailBoxWriteB2AData(0, MAILBOX_ID_0, MAILBOX_CHANNEL_2);
}
break;
```

B 核初始化处理完成之后向 A 核发送 MEDIA_MSGBOX_CMD_FILE_OPEN_HANDSHK 握手命令，表示 B 核接收到文件打开的操作，并相应做好了准备工作。A 核接收到响应信号之后开始准备下一步动作。至此文件打开的 A、B 核交互操作完成。

文件读取交互过程：

文件读取（B 核部分）：这里 B 核解码过程中发现解码数据不够，会向 A 核发送文件读取命令。如下：

```

uint32 HifiFileRead(uint8 *pData, uint32 NumBytes, HANDLE Handle)
{
    uint32 timeout = 200000;

    gFileReadParam.pData = pData;
    gFileReadParam.NumBytes = NumBytes;
    gFileReadParam.handle = Handle;

    gCmdDone = 0;
    MailBoxWriteB2ACmd(MEDIA_MSGBOX_CMD_FILE_READ, MAILBOX_ID_0, MAILBOX_CHANNEL_2);
    MailBoxWriteB2AData((uint32)&gFileReadParam, MAILBOX_ID_0, MAILBOX_CHANNEL_2);

    if(gBufByPass)
    {
        timeout = 3000000;
        while (!gCmdDone)
        {
            //__WFI2();
            DelayUs2(1);
            if (--timeout == 0)
            {
                bb_printf("HifiFileRead: timeout!!!");
                break;
            }
        }
        CurFileOffset[gFileReadParam.handle] = gFileReadParam.NumBytes;
        gCmdDone = 0;
    }

    return gData; //return read data length
}
else
{
    return 0;
}
} ? end HifiFileRead ?

```

文件读取 (A 核部分): A 核接收到该命令后执行文件读取操作。在 mailbox 注册的文件操作的中断服务 AudioDecodingInputFileBuffer 中:

```

case MEDIA_MSGBOX_CMD_FILE_READ:
{
    uint32 ret;
    uint8 handle;
    uint8 *dataptr;
    uint32 num;
    gpFileReadParm = (FILE_READ_OP_t*)data;
    handle = gpFileReadParm->handle;
    dataptr = gpFileReadParm->pData;
    num = gpFileReadParm->NumBytes;
    ret = FileRead(gpFileReadParm->pData, num, handle);
    gpFileReadParm->NumBytes = FileInfo[gpFileReadParm->handle].offset;
    MailBoxWriteA2BCmd(MEDIA_MSGBOX_CMD_FILE_READ_CMPL, MAILBOX_ID_0, MAILBOX_CHANNEL_2);
    MailBoxWriteA2BData(ret, MAILBOX_ID_0, MAILBOX_CHANNEL_2);
    //dumpMemoryA(AudioFileBuffer, 32);
}
break;

```

读取操作执行完成之后, A 核会向 B 核发送 MEDIA_MSGBOX_CMD_FILE_READ_CMPL 命令和读取的数据长度。

文件读取 (B 核部分): 此时 B 核接收到 A 核命令和数据后, 将数据流写入自己的 buffer 中:

```
case MEDIA_MSGBOX_CMD_FILE_READ_CMPL:  
    gFileOpStatus = AUDIO_FILE_READ_CMPL;  
    if(gBufByPass == 0)  
    {  
        CurFileOffset[gFileReadParam.handle] = gFileReadParam.NumBytes;  
        AudioFileBufSize2[AudioFileWrBufID2] = gData;  
        AudioFileBufBusy2 = 0;  
        gBufByPass = 1;  
    }  
    gCmdDone = 1;  
    break;
```

6 如何增加新的解码库

目前 NanoD MP3 SDK 软件包中支持的解码库格式有 16bit 的 MP3、WAV、AAC、OGG、FLAC、APE 以及支持 24bit 的 HIFI APE、HIFI FLAG、HIFI ALAC 格式。实际应用中因为不同的客户还有其他音频格式可能需要支持，需要用户自己开发添加。本文档就是介绍怎样在 NanoD MP3 SDK 中快速的添加其他音频格式的解码库。

下文中分两中情况描述添加的过程。第一种情况，添加移植常规支持的音频格式。第二种情况，添加用户自定义开发的音频格式。

6.1 添加常规格式音频解码

添加常规格式音频解码，是指从已知的所支持的音频格式中添加需要支持的音频解码。此处以添加支持常见的 FLAC 音频格式为例，需要 SDK 修改地方如下：

- **Sysconfig.h** 中添加支持的格式宏定义。

```
#ifdef _MUSIC_
#define _MP3_DECODE_
#define _WAV_DECODE_
#define _AAC_DECODE_
#define _FLAC_DECODE_
#define _OGG_DECODE_
#define _HIFI_APE_DEC
#define _HIFI_FLAC_DEC
#define _HIFI_ALAC_DEC
```

- **BB_Core.c** 添加 B 核编译后生成的 flac 的解码库二进制文件。

```
#ifdef FLAC_DEC_INCLUDE
_ATTR_FLACDEC_BIN_TEXT_
const uint8 flac_code_bin[] =
{
    #include "flac_code.bin"
};

_ATTR_FLACDEC_BIN_DATA_
uint8 flac_data_bin[] =
{
    #include "flac_data.bin"
};
#endif
```

- **Audio_main.h** 中添加 flac 的代码段以及解码库段的定义，如下：

添加解码宏定义

```

#ifndef _FLAC_DECODE_
#define FLAC_DEC_INCLUDE
#endif

//-----
#define _ATTR_FLACDEC_TEXT_          _attribute_( (section("FlacDecCode")))
#define _ATTR_FLACDEC_DATA_         _attribute_( (section("FlacDecData")))
#define _ATTR_FLACDEC_BSS_          _attribute_( (section("FlacDecBss"), zero_init))

//-----
#define _ATTR_FLACDEC_BIN_TEXT_     _attribute_( (section("FlacDecBinCode")))
#define _ATTR_FLACDEC_BIN_DATA_    _attribute_( (section("FlacDecBinData")))
#define _ATTR_FLACDEC_BIN_BSS_     _attribute_( (section("FlacDecBinBss"), zero_init))

```

- 在 Common\Codec\Audio 文件夹下添加 flac 的解码库文件以及 A、B 核用到的解码文件。

名称	修改日期	类型	大小
AAC	2015/12/7 15:03	文件夹	
APE	2015/12/3 14:53	文件夹	
AudioControl	2016/2/26 11:16	文件夹	
Common	2016/2/26 11:17	文件夹	
flac	2015/12/3 15:25	文件夹	
HIFI	2015/4/27 15:24	文件夹	
ID3	2016/2/26 11:20	文件夹	
Include	2016/2/26 11:21	文件夹	
Library	2015/8/26 22:24	文件夹	
Mp3	2016/2/26 11:22	文件夹	
MP3_ENC	2015/9/17 15:40	文件夹	
Ogg	2015/12/3 15:46	文件夹	
RecordControl	2016/2/26 11:34	文件夹	
RkEQ	2015/8/25 16:36	文件夹	
ShuffleAll	2015/10/13 11:32	文件夹	
Spectrum	2015/4/21 18:59	文件夹	
Wav	2016/2/26 11:34	文件夹	
AudioConfig.h	2015/1/8 10:19	C/C++ Header	1 KB

Flac 文件夹中包含以下文件: 解码库 bin 文件以及 A、B 核解码交互接口文件。其中 pFLAC.c 中: A 核通过 mailbox 向 B 核发送消息, 等待 B 核真正解码处理。pFLAC2.c 中是真正的 flac 格式解码处理的功能接口。这里的 bin 文件是 flac 解码的库文件, 根据解码原始文件打包生成。

flac_code.bin	2015/12/3 11:51	BIN 文件	243 KB
flac_data.bin	2015/12/3 11:51	BIN 文件	112 KB
pFLAC.c	2016/2/25 15:47	C Source	12 KB
pFLAC2.c	2015/5/13 10:10	C Source	6 KB

- Global.h 中添加支持 flac 格式的后缀字符串 “FLA”

```

: _ATTR_SYS_DATA_ EXT char      AudioFileExtString[] = "MP1MP2MP3REVWAVAPEFLAACM4AOGGMP43GP";
: _ATTR_SYS_DATA_ EXT char      MusicFileExtString[] = "MP1MP2MP3REVWAVAPEFLAACM4AOGGMP43GP";

```

- Audio_globals.h 中添加一下 Flac 的 codec 类型:

```

enum {
#ifndef MP3_DEC_INCLUDE
    CODEC_MP3_DEC,
#endif

    CODEC_REV_DEC,

#ifndef AAC_DEC_INCLUDE
    CODEC_AAC_DEC,
#endif

#ifndef WAV_DEC_INCLUDE
    CODEC_WAV_DEC,
#endif

#ifndef APE_DEC_INCLUDE
    CODEC_APE_DEC,
#endif

#ifndef FLAC_DEC_INCLUDE
    CODEC_FLAC_DEC,
#endif
}

```

需要注意的是，A 核与 B 核的 codec ID 号必须保持一致。

添加 A 核的解码接口声明：

```

//pFLAC.C
extern unsigned long FLACDecFunction(unsigned long ulSubFn, unsigned long ulParam1,
                                         unsigned long ulParam2, unsigned long ulParam3);

```

添加 B 核的解码接口声明：

```

//pFLAC.C
extern unsigned long FLACDecFunction2(unsigned long ulSubFn, unsigned long ulParam1,
                                         unsigned long ulParam2, unsigned long ulParam3);

```

- pCodecs.c 中添加 A 核对应格式的解码入口：

```

ATTR_AUDIO_DATA_
static unsigned long (*CodecPFn[NUMCODECS])(unsigned long ulSubFn,
                                              unsigned long ulParam1,
                                              unsigned long ulParam2,
                                              unsigned long ulParam3) =
{
    #ifdef MP3_DEC_INCLUDE
    MP3Function,
    #else
    0,
    #endif

    #ifdef WMA_DEC_INCLUDE
    WMAFunction,
    #endif

    #ifdef AAC_DEC_INCLUDE
    AACDecFunction,
    #endif

    #ifdef WAV_DEC_INCLUDE
    PCMFuction,
    #endif

    #ifdef FLAC_DEC_INCLUDE
    FLACDecFunction,
    #endif

    #ifdef OGG_DEC_INCLUDE
    OGGDecFunction,
    #endif
}

```

- pCodecs2.c 中添加 B 核对应格式的解码入口：

```
#ifdef WAV_DEC_INCLUDE
#ifndef WAV_DEC_INCLUDE2
PCMFunction2,
#else
0,
#endif
#endif

#ifndef APE_DEC_INCLUDE
#ifndef APE_DEC_INCLUDE2
APEDecFunction2,
#else
0,
#endif
#endif

#ifndef FLAC_DEC_INCLUDE
#ifndef FLAC_DEC_INCLUDE2
FLACDecFunction2,
#else
0,
#endif
#endif

#ifndef OGG_DEC_INCLUDE
#ifndef OGG_DEC_INCLUDE2
OGGDecFunction2,
#else
0,
#endif
#endif
```

- **AudioControl.c** 中

AudioCodec 接口中添加 flac 格式类型的选择:

```
case 7:      //flac
    CurrentCodec = CODEC_FLAC_DEC;
    break;
```

AudioFileOpen 接口中添加 flac 格式的文件打开处理:

```
#ifdef FLAC_DEC_INCLUDE
{
    if (CurrentCodec == CODEC_FLAC_DEC)
    {
        ClearMsg(MSG_AUDIO_DECODE_FILL_BUFFER);
        RKFileFuncInit();
        FileSeek(0, SEEK_SET, (HANDLE)pRawFileCache);
    }

    if (CODEC_FLAC_DEC == CurrentCodec)
    {
        pFlacFileHandleBake = (FILE*)FileOpen(AudioFileInfo.Fdt.Name, AudioFileInfo.FindData.Clus,
                                                if ((uint32)pFlacFileHandleBake > MAX_OPEN_FILES)
                                                {
                                                    DEBUG("ERROR!!! pFlacFileHandleBake = %d, Open fail", pFlacFileHandleBake);
                                                    return ERROR;
                                                }
    }
}

#endif
```

AudioFileClose 接口中添加 flac 格式的文件关闭处理:

```
#ifdef FLAC_DEC_INCLUDE
{
    if (CurrentCodec == CODEC_FLAC_DEC)
    {
        RKFIOL_FClose((FILE*)pFlacFileHandleBake);
        pFlacFileHandleBake = (FILE*) - 1;
    }
}
```

AudioHWInit 接口中添加 flac 的启动 B 核初始化处理:

```
#ifdef FLAC_DEC_INCLUDE

    case CODEC_FLAC_DEC:
        ModuleOverlay(MODULE_ID_FLAC_DECODE, MODULE_OVERLAY_ALL);
    {
        //...
        StartBBSystem(MODULE_ID_FLAC_DECODE_BIN);

        //Others
        memset(&gFileHandle, 0, sizeof(gFileHandle));
        gFileHandle.handle1 = (unsigned char)pRawFileCache;
        gFileHandle.filesize = RKFIO_FLength(pRawFileCache);
        gFileHandle.curfileoffset[0] = FileInfo[gFileHandle.handle1].Offset;
        gFileHandle.handle2 = (unsigned char)pFlacFileHandleBake;
        gFileHandle.curfileoffset[1] = FileInfo[gFileHandle.handle2].Offset;
        AudioWaitBBStart();
    }
    break;
#endif
```

AudioHWDeinit 接口中添加 flac 格式的关闭 B 核的反初始化处理:

```
#ifdef FLAC_DEC_INCLUDE
    case CODEC_FLAC_DEC:
    {
        //...
        AudioWaitBBStop();
        ShutOffBBSystem();
    }
    break;
#endif
```

AudioFREQInit 接口中添加 Flac 格式的解码 A、B 核频率设置:

```
#ifdef FLAC_DEC_INCLUDE

    case (CODEC_FLAC_DEC):
    {
        DEBUG("ENTER FREQ_FLAC");
        FREQ_EnterModule(FREQ_FLAC);
        break;
    }

#endif
```

AudioFREQDeInit 接口中添加 flac 格式的模块卸载 A、B 核的频率:

```
#ifdef FLAC_DEC_INCLUDE
    case (CODEC_FLAC_DEC):
    {
        FREQ_ExitModule(FREQ_FLAC);
        break;
    }

#endif
```

- 这其中需要添加 FREQ_FLAC 的定义，在 PowerManager.h 中添加

```

typedef enum _FREQ_APP
{
    FREQ_IDLE = 0,
    FREQ_MIN,
    FREQ_INIT,
    FREQ_BLON,
    FREQ_AUDIO_INIT,
    FREQ_MP3,           //5
    FREQ_MP3H,
    FREQ_WAV,
    FREQ_AAC,          //10
    FREQ_AACL,
    FREQ_APE,
    FREQ_FLAC,         //15
    FREQ_OGG,
    FREQ_NOGG,
    FREQ_HOGG,
    FREQ_EHOGG,
    FREQ_HAPE,
    FREQ_HFLAC,
    FREQ_HALAC,
}

```

同时在 PowerManager.c 中添加 Flac 对应的频率配置表:

```

_ATTR_DRIVER_CODE_
FREQ_APP_TABLE g_CruAPPTabel[FREQ_APP_MAX] =
{
    // ID,      counter,      pll, sys_hclk, sys_stclk, sys_pclk, cal_hclk, cal_stclk
{FREQ_IDLE,        0,      0,      0,      0,      0,      0,      0},
{FREQ_MIN,         0, 12000000, 12000000, 12000000, 12000000, 0, 0},
{FREQ_INIT,        0, 96000000, 96000000, 96000000, 75000000, 0, 0},
{FREQ_BLON,        0, 24000000, 24000000, 24000000, 24000000, 0, 0},
{FREQ_AUDIO_INIT,  0, 48000000, 48000000, 48000000, 48000000, 48000000, 48000000},
{FREQ_MP3,         0, 24000000, 24000000, 24000000, 24000000, 24000000, 24000000},
{FREQ_MP3H,        0, 30000000, 30000000, 30000000, 30000000, 48000000, 48000000},
{FREQ_WMA,         0, 60000000, 60000000, 60000000, 60000000, 60000000, 60000000},
{FREQ_WMAH,        0, 96000000, 96000000, 96000000, 96000000, 96000000, 96000000},
{FREQ_WAV,         0, 88000000, 88000000, 88000000, 88000000, 88000000, 88000000},
{FREQ_AACL,        0, 18000000, 18000000, 18000000, 18000000, 36000000, 36000000},
{FREQ_AAC,         0, 80000000, 80000000, 80000000, 75000000, 80000000, 80000000},
{FREQ_FLAC,        0, 100000000, 100000000, 100000000, 75000000, 100000000, 100000000}, //43
{FREQ_SBC_ENCODING, 0, 40000000, 40000000, 40000000, 40000000, 0, 0},
{FREQ_GDS_ENCODING, 0, 40000000, 40000000, 40000000, 40000000, 40000000, 40000000}
}

```

- ModulInfoTab.h 文件中添加 flac 格式的一般处理代码模块以及解码模块的 ID:

```

MODULE_ID_WAV_DECODE,                                //43
MODULE_ID_AAC_DECODE,
MODULE_ID_FLAC_DECODE,

// B core
MODULE_ID_BB_CODE,
MODULE_ID_MP3_DECODE_BIN,
MODULE_ID_WAV_DECODE_BIN,
MODULE_ID_FLAC_DECODE_BIN,
MODULE_ID_AAC_DECODE_BIN,
MODULE_ID_APE_DECODE_BIN,

```

- ScatterLoader.c 中添加 flac 代码段与 flac 解码库的代码段:

加载段声明:

```
//FLAC
extern uint32 Load$$FLAC_DECODE_CODE$$Base;
extern uint32 Image$$FLAC_DECODE_CODE$$Base;
extern uint32 Image$$FLAC_DECODE_CODE$$Length;
extern uint32 Load$$FLAC_DECODE_DATA$$Base;
extern uint32 Image$$FLAC_DECODE_DATA$$RW$$Base;
extern uint32 Image$$FLAC_DECODE_DATA$$RW$$Length;
extern uint32 Image$$FLAC_DECODE_DATA$$ZI$$Base;
extern uint32 Image$$FLAC_DECODE_DATA$$ZI$$Length;

//FLAC
extern uint32 Load$$FLAC_DECODE_BIN_CODE$$Base;
extern uint32 Image$$FLAC_DECODE_BIN_CODE$$Base;
extern uint32 Image$$FLAC_DECODE_BIN_CODE$$Length;
extern uint32 Load$$FLAC_DECODE_BIN_DATA$$Base;
extern uint32 Image$$FLAC_DECODE_BIN_DATA$$RW$$Base;
extern uint32 Image$$FLAC_DECODE_BIN_DATA$$RW$$Length;
extern uint32 Image$$FLAC_DECODE_BIN_DATA$$ZI$$Base;
extern uint32 Image$$FLAC_DECODE_BIN_DATA$$ZI$$Length;
```

加载段内容：

```
//FLAC
{
    (uint32) (&Load$$FLAC_DECODE_CODE$$Base),
    (uint32) (&Image$$FLAC_DECODE_CODE$$Base),
    (uint32) (&Image$$FLAC_DECODE_CODE$$Length),
    (uint32) (&Load$$FLAC_DECODE_DATA$$Base),
    (uint32) (&Image$$FLAC_DECODE_DATA$$RW$$Base),
    (uint32) (&Image$$FLAC_DECODE_DATA$$RW$$Length),
    (uint32) (&Image$$FLAC_DECODE_DATA$$ZI$$Base),
    (uint32) (&Image$$FLAC_DECODE_DATA$$ZI$$Length),
},
//FLAC
{
    (uint32) (&Load$$FLAC_DECODE_BIN_CODE$$Base),
    (uint32) (&Image$$FLAC_DECODE_BIN_CODE$$Base),
    (uint32) (&Image$$FLAC_DECODE_BIN_CODE$$Length),
    (uint32) (&Load$$FLAC_DECODE_BIN_DATA$$Base),
    (uint32) (&Image$$FLAC_DECODE_BIN_DATA$$RW$$Base),
    (uint32) (&Image$$FLAC_DECODE_BIN_DATA$$RW$$Length),
    (uint32) (&Image$$FLAC_DECODE_BIN_DATA$$ZI$$Base),
    (uint32) (&Image$$FLAC_DECODE_BIN_DATA$$ZI$$Length),
},
```

- BuildAll.scr 脚本文件中添加 flac 代码段的定义：

```
;FlacDecode
FLAC_DECODE_CODE (AUDIO_DECODE_CODE_BASE) OVERLAY ;Flac Decode Code
{
    *(FlacDecCode)
    flac_stream_decoder.o(+RO)
    flac_format.o
    flac_tab.o
    flac_decode.o
}
FLAC_DECODE_DATA (AUDIO_DECODE_DATA_BASE) OVERLAY ;Flac Decode Data
{
    *(FlacDecData)
    *(FlacDecBss)
    flac_stream_decoder.o(+RW)
    flac_stream_decoder.o(+ZI)
}
FLAC_DECODE_CODE_END (ImageLimit(FLAC_DECODE_CODE)) OVERLAY {}
FLAC_DECODE_DATA_END (ImageLimit(FLAC_DECODE_DATA)) OVERLAY {}
ScatterAssert(ImageLimit(FLAC_DECODE_CODE) < (SYS_CODE_BASE + SYS_CODE_SIZE))
ScatterAssert(ImageLimit(FLAC_DECODE_DATA) < (SYS_DATA_BASE + SYS_DATA_SIZE))
```

添加 Flac 解码库代码段的定义：

```
;-----  
;Flac Decode Bin  
FLAC_DECODE_BIN_CODE (HAUDIO_DECODE_CODE_BASE) OVERLAY ;flac Decode Code  
{  
    *(FlacDecBinCode)  
}  
FLAC_DECODE_BIN_DATA (HAUDIO_DECODE_DATA_BASE) OVERLAY ;flac Decode Data  
{  
    *(FlacDecBinData)  
    *(FlacDecBinBss)  
}  
FLAC_DECODE_BIN_CODE_END (ImageLimit(FLAC_DECODE_BIN_CODE)) OVERLAY {}  
FLAC_DECODE_BIN_DATA_END (ImageLimit(FLAC_DECODE_BIN_DATA)) OVERLAY {}  
ScatterAssert(ImageLimit(FLAC_DECODE_BIN_CODE) < (HRAM_CODE_BASE + HRAM_CODE_SIZE))  
ScatterAssert(ImageLimit(FLAC_DECODE_BIN_DATA) < (HRAM_DATA_BASE + HRAM_DATA_SIZE))
```

至此，添加常规音频格式的工作就已经完成了。下文将介绍用户自己添加非常规格式音频解码的步骤。

6.2 添加用户支持的音频解码

因为产品的差异化需要，在开发中，客户有可能要添加自己支持的非常规的音频格式解码，以便使自己的产品在满足不同消费群体与消费对象中展现独特的一面。下面将介绍一下这方面的实现步骤。

同时，用户添加支持的非常规的音频格式的解码时，需严格按照本 RKNanoD 平台解码中解码流程中的接口要求，提供对应的 API，负责无法与 RKNanoD 平台的解码流程相兼容。具体以 mp3 为例，下图是 RKNanoD 解码器接口定义：

```
enum  
{  
    SUBFN_CODEC_GETNAME,  
    SUBFN_CODEC_GETARTIST,  
    SUBFN_CODEC_GETTITLE,  
    SUBFN_CODEC_GETBITRATE,  
    SUBFN_CODEC_GETSAMPLERATE,  
    SUBFN_CODEC_GETCHANNELS,  
    SUBFN_CODEC_GETLENGTH,  
    SUBFN_CODEC_GETTIME,  
    SUBFN_CODEC_OPEN_DEC,  
    SUBFN_CODEC_OPEN_ENC,  
    SUBFN_CODEC_GETBUFFER,  
    SUBFN_CODEC_SETBUFFER,  
    SUBFN_CODEC_DECODE,  
    SUBFN_CODEC_ENCODE,  
    SUBFN_CODEC_SEEK,  
    SUBFN_CODEC_CLOSE,  
    SUBFN_CODEC_ZOOM,  
    SUBFN_CODEC_GETBPS  
};
```

此为完整的解码过程中需要提供的 API 功能，实际中可能不需要这么多，比如 mp3 解码的接口：

```
_ATTR_MP3DEC_TEXT_
unsigned long MP3Function2(unsigned long ulIoctl, unsigned long ulParam1,
                           unsigned long ulParam2, unsigned long ulParam3)
{
    switch (ulIoctl)
    {
        case SUBFN_CODEC_OPEN_DEC:
        {
            //codec open processing
            //todo...
        }

        case SUBFN_CODEC_GETBUFFER:
        {
            //get decode buffer
            //todo...
            return 1;
        }

        case SUBFN_CODEC_DECODE:
        {
            //decoding ...
            //todo...
            return 1;
        }

        case SUBFN_CODEC_GETSAMPLERATE:
        {
            //get sample rate
            //todo...
            return(1);
        }

        case SUBFN_CODEC_GETCHANNELS:
        {
            //get channels
            //todo...
            return(1);
        }

        case SUBFN_CODEC_GETBITRATE:
        {
            //get bitrate
            //todo...
            return(1);
        }

        case SUBFN_CODEC_GETLENGTH:
        {
            //get total time
            //todo...
            return 1;
        }

        case SUBFN_CODEC_GETTIME:
        {
            //get currnet time
            //todo...
            return 1;
        }

        case SUBFN_CODEC_SEEK:
        {
            //seek processing
            //todo...
            return 1;
        }
    }
}
```

```

        case SUBFN_CODEC_CLOSE:
        {
            //close codec
            //todo...
            return 1;
        }
        case SUBFN_CODEC_GETBPS:
        {
            //get bps
            //todo...
            return 1;
        }

        default:
        {
            return 0;
        }
    } ? end switch ulIoctl ?

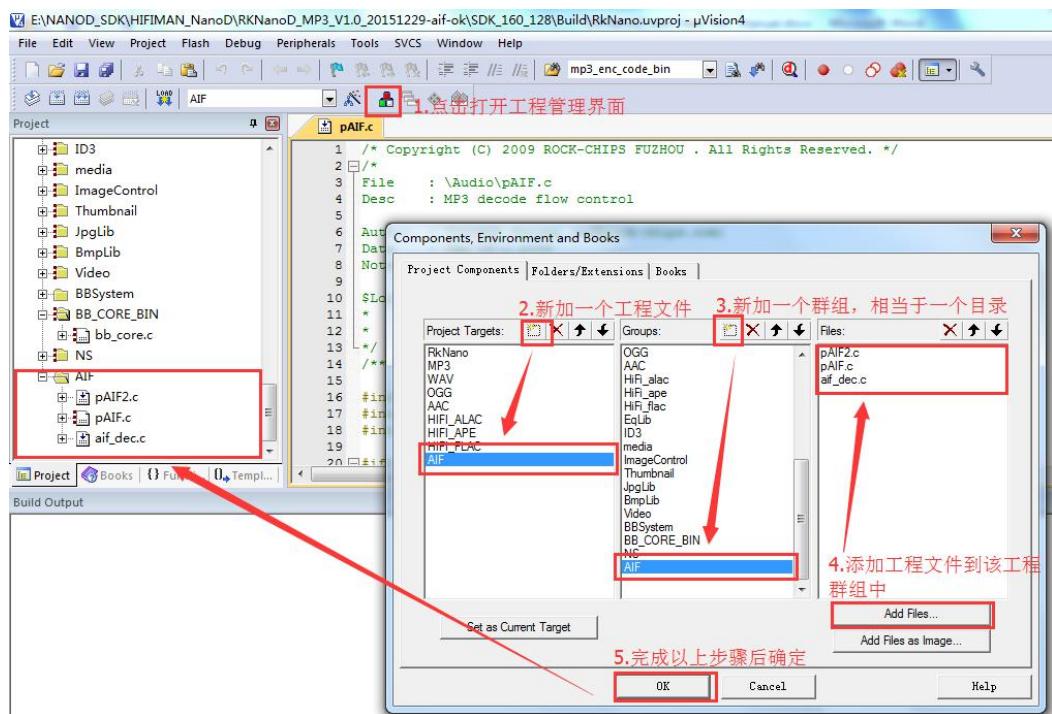
    return -1;
} ? end MP3Function2 ?

```

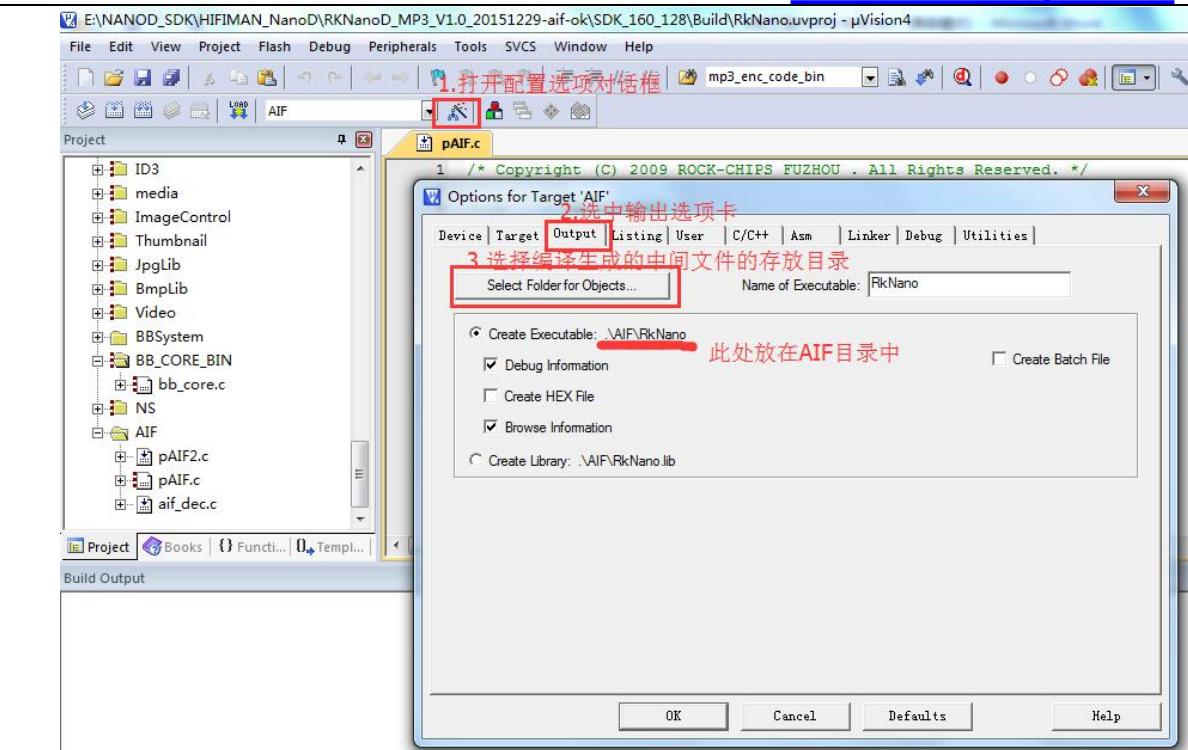
下文以添加 AIF 格式解码库工程为例，说明如何添加一个非常规的音频解码库工程，生成解码库。当然大部分工作同上一节介绍的，这里不再赘述，只陈述如何添加工程，如何配置工程，如何生成解码库等工作。

- 创建 B 核编译工程：

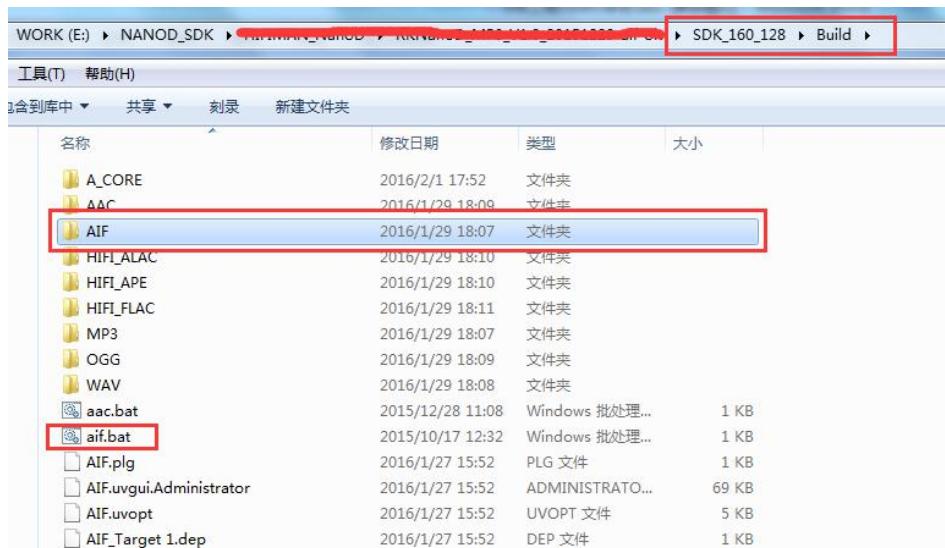
打开工程文件后，按如下图示操作，添加 aif 的 B 核编译工程。



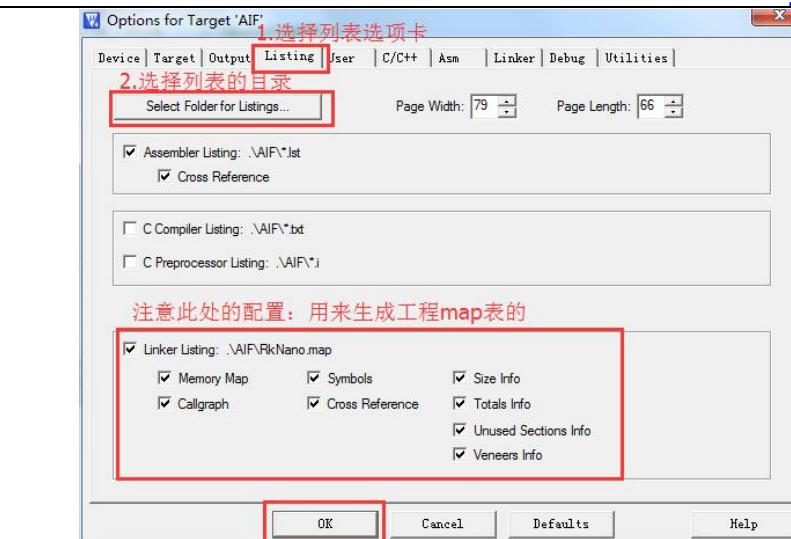
- 配置 AIF 解码工程：



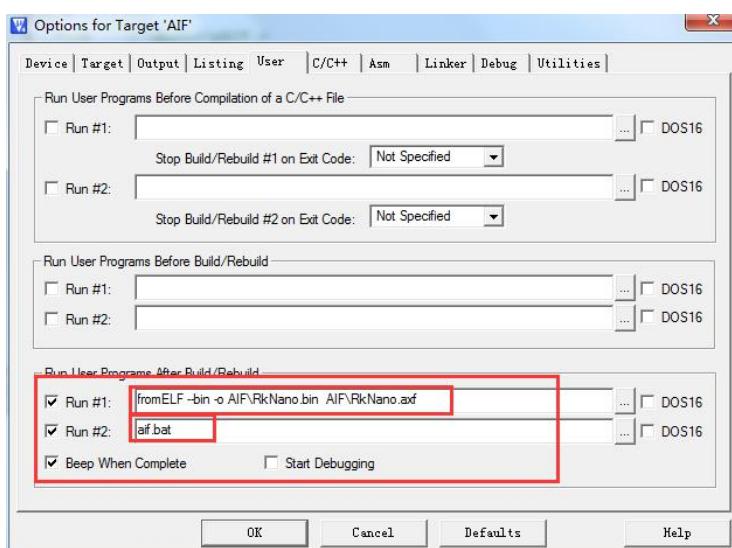
这里需要到\SDK_160_128\Build 目录下，新建一个输出接收目录，这里我们新建一个 AIF 的目录，该目录用来存放编译 AIF 解码库工程生成的中间文件。如下图



配置工程列表的配置：这里选择列表的目录时，就选择上面我们新创建的 AIF 的目录。



配置用户自定义命令，用以 B 核编译完成生成可执行的 bin 文件：



RUN#1 中填写:**fromELF --bin -o AIF\RkNano.bin AIF\RkNano.axf**

RUN#2 中填写：**aif.bat**

打开\SDK_160_128\Build 目录，创建 aif.bat，内容如下（注意路径的正确性）：

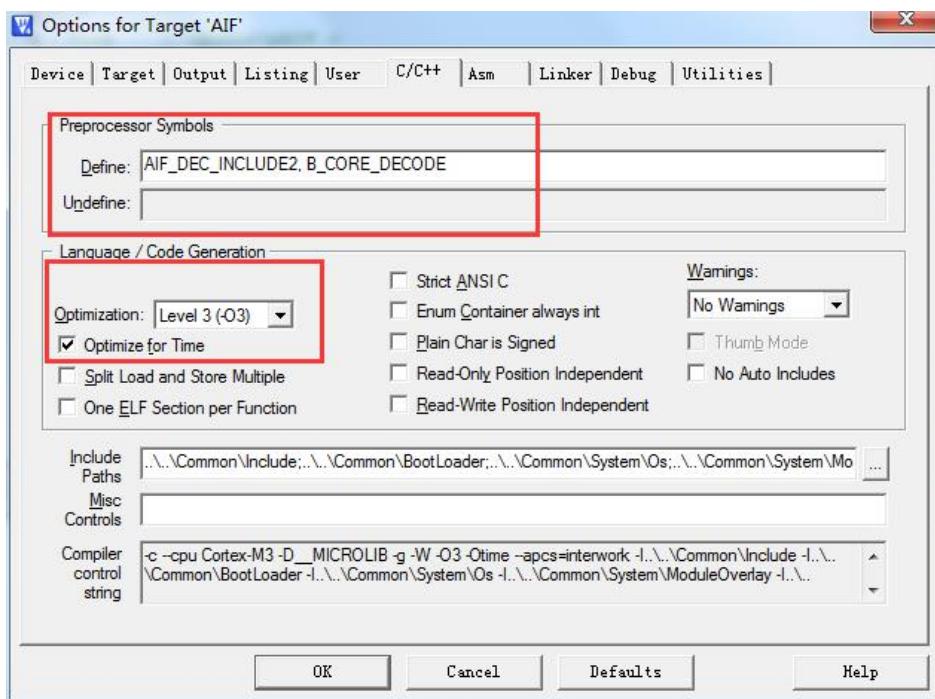
```
bin_generate.exe AIF\RkNano.bin\BB_SYS_CODE ..\..\Common\Codec\Audio\AIF\code.bin
bin_generate.exe AIF\RkNano.bin\BB_MAIN_STACK ..\..\Common\Codec\Audio\AIF\data.bin
```

该命令执行完成之后，会将 Build\AIF 目录下编译的转换为可执行 bin 文件，并拷贝到音频解码器指定目录 common\codec\audio\AIF\目录下。

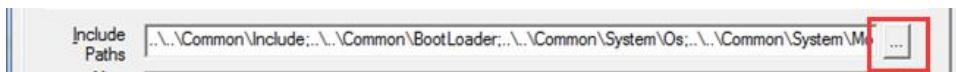
配置编译环境如下图所示：

这里需要预定一个编译宏 : AIF_DEC_INCLUDE2, B_CORE_DECODE, 不同解码库预定义的

编译宏不同。



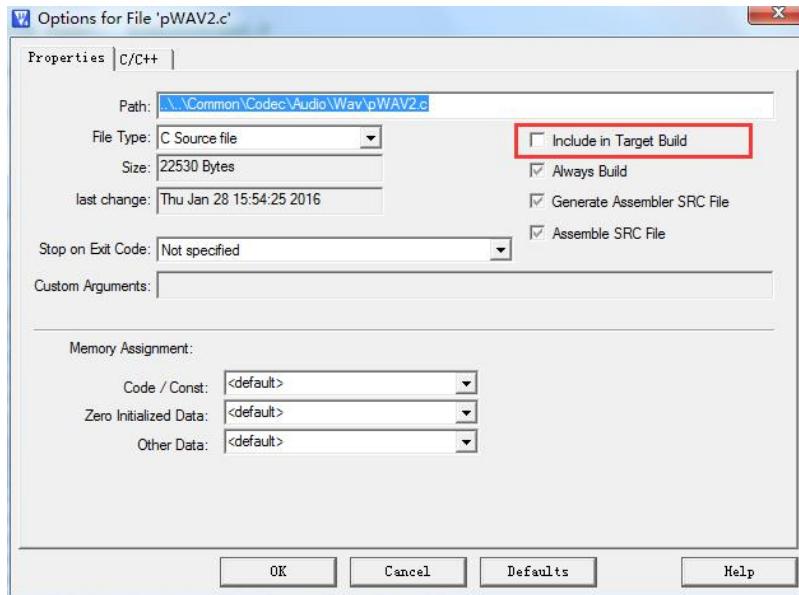
此处注意头文件路径的添加，不然会编译找不到解码相应的头文件或者定义。



B 核解码器工程创建完成之后，优先编译 B 核解码器工程，生成的二进制文件会更新到 A 核指定目录，此时由于该二进制代码变动，要求 A 核再重新编译才能使用。

注意事项：

1. 解码库工程添加中，小心包含其他解码库工程文件，如有包括，右键单击不需要参与编译的文件，取消下图中的对勾即可以排除。请确保，每个独立的解码库只编译该解码工程中的文件，BBsystem 是每个解码工程中均需包含编译的目录。



取消红框出的勾选，就可排除该文件参与编译。

如下图所示，参与编译的文件有蓝色向下箭头，不参与编译的文件显示为左侧红点，不带蓝色箭头。

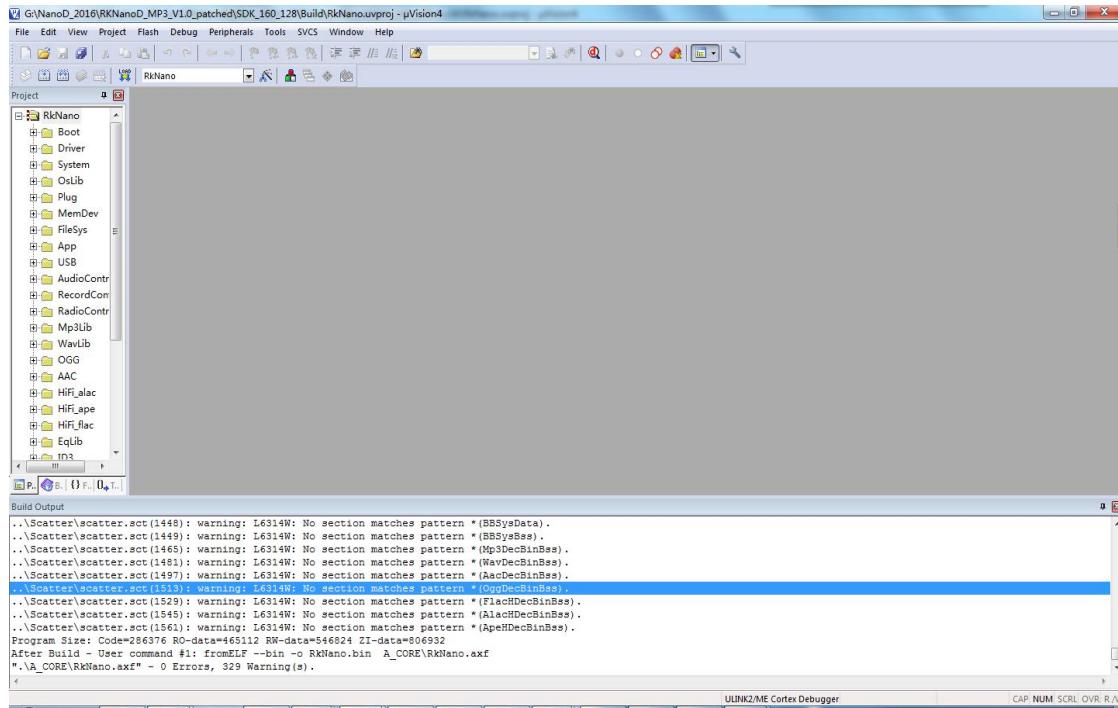


2. 解码库相关头文件路径的添加。
3. B 核编译解码库只编译 “***2.c” 及解码库算法文件的编译，如 pMp32.c, pMp3.c 不参加编译。同理，A 核工程编译时，pMp32.c 文件不参加编译。

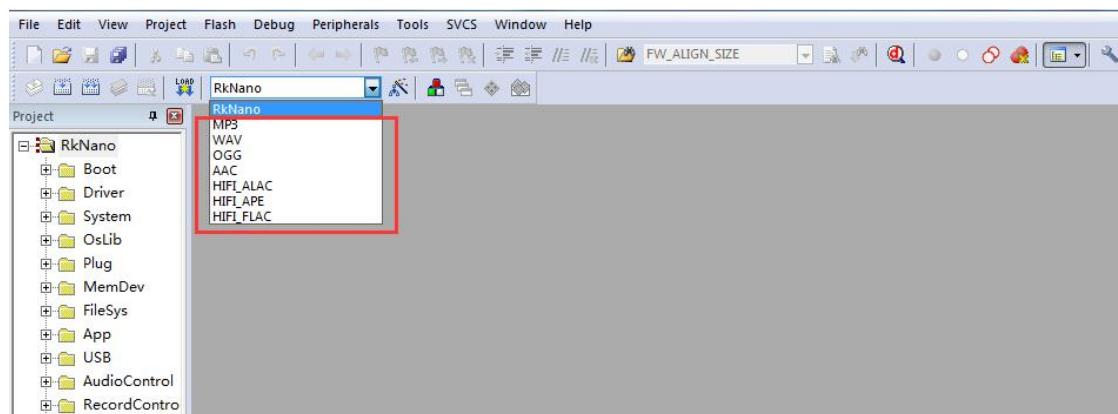
7 工程编译

RKNanoD SDK 使用 ARM 的 Keil μ Vision 4 （最好 4.72.10 及以上版本）编译开发。工程文件路径：SDK_160_128/Build/RkNano.uvproj。

打开工程编译软件，工程界面如下图：

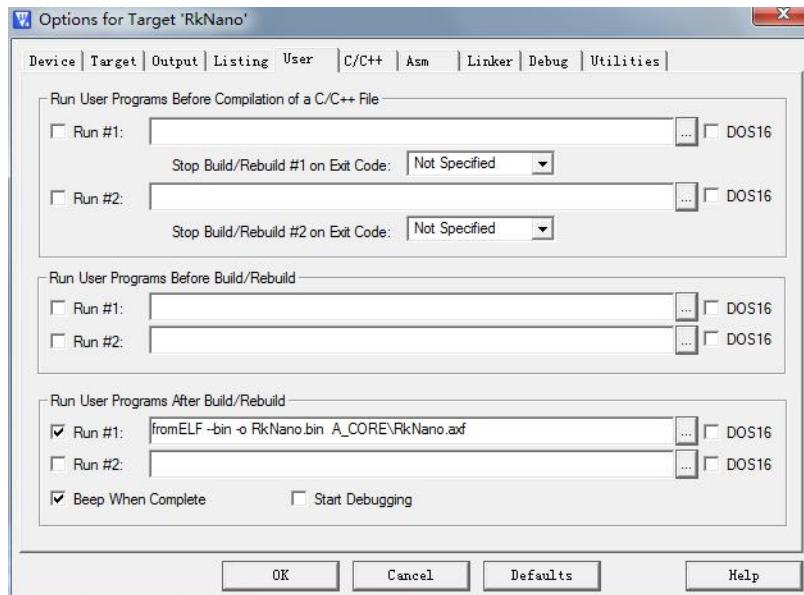


打开工程下拉列表，先编译 B 核解码库工程，如下图。



编译完成所有 B 核工程后，正确生成了解码库 bin 文件，再编译 RkNanod 工程，点击“build target”或者“rebuild All”按钮，编译整个工程，下面的输出窗口将会显示编译结果以及链接的结果，“scatter.sct”的警告信息对编译的结果没有影响，如果编译链接均成功，将会在 A_Core 文件

目录下生成 RkNano.axf 和 RkNano.bin。工程配置信息如下：



需要注意的是，首先需要全部编译 B 核工程，生成相应的运行于 B 核的 bin 文件之后，再编译 A 核工程；如果 B 核代码没有修改，可以直接编译 A 核工程。

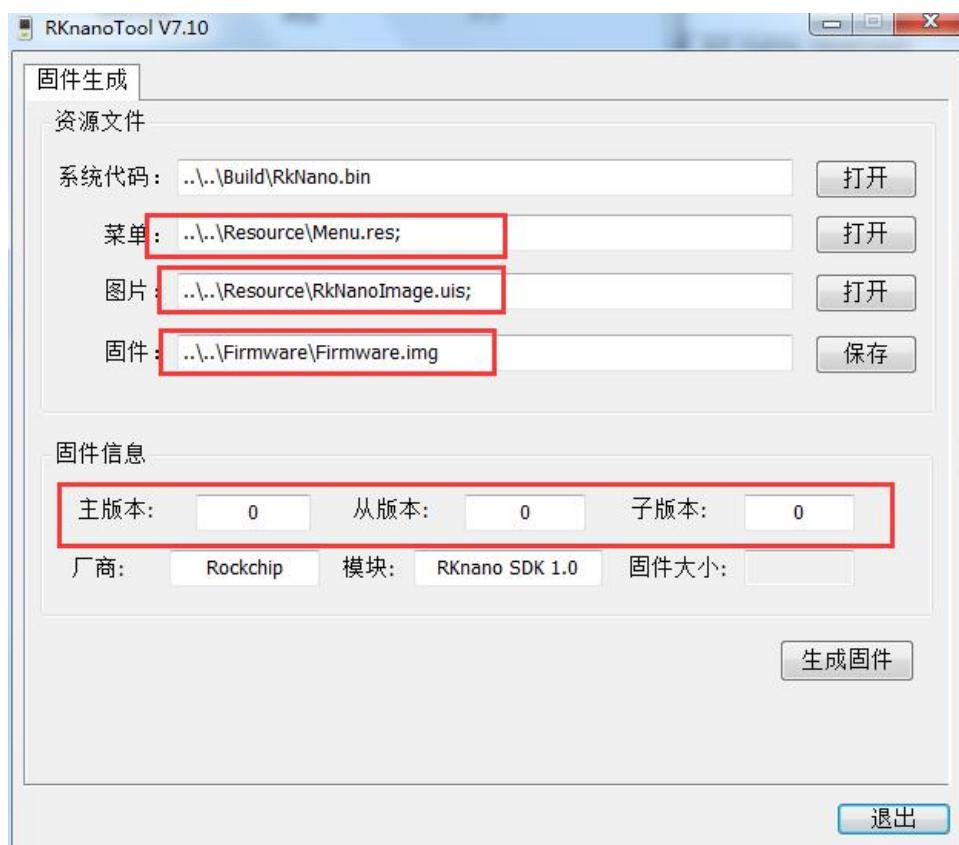
8 生成固件与烧写固件

RKNanoD MP3 SDK 中提供了固件的生成与烧写工具。软件工程编译链接后，会在 Build 目录中生成了 RkNano.bin 文件，该文件是可运行在 NanoD 芯片上的二进制可执行文件，固件则包括了可执行的二进制文件以及图片资源和字符串菜单资源。

8.1 EMMC Flash 固件生成工具

“SDK_160_128\Development\firmware_generate_eMMC” 目录中 RKnanoTool.exe 工具是用来生成 EMMC flash 下的固件工具。如下图：为 EMMC Flash 下的固件生成工具界面。

注意：固件信息中主版本号、从版本号均可以设置为 2 位有效数字。子版本号可以设置为 4 位有效数字。

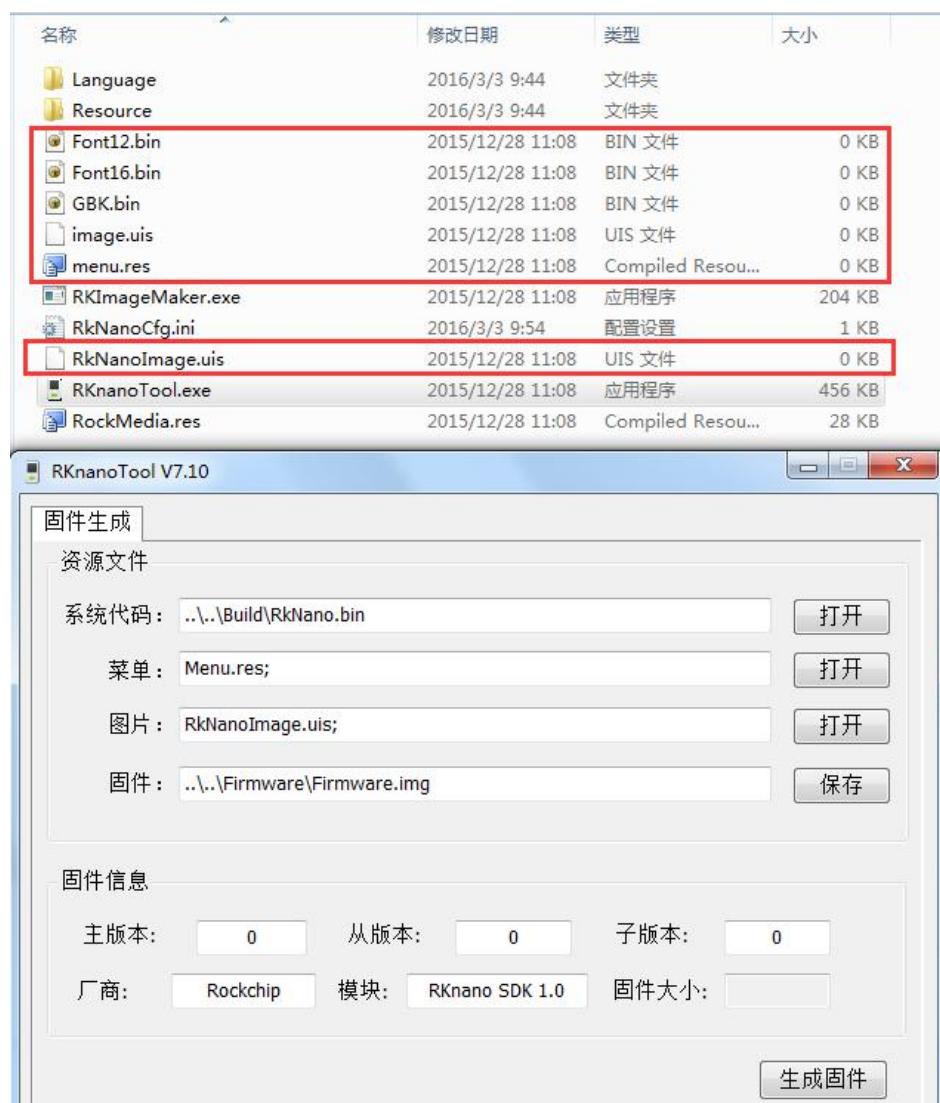


8.2 SPI NOR Flash 固件生成工具

“SDK_160_128\Development\firmware_generate_SPI” 目录中 RKnanoTool.exe 工具是用来生成 SPI Nor flash 下的固件工具。如下图：为 SPI Nor Flash 下的固件生成工具界面。

注意：

1. 其中固件信息中主版本号、从版本号均可以设置为 2 位有效数字。子版本号可以设置为 4 位有效数字。
2. 这里因为 SPI Nor Flash 的存储容量有限，固件与图片资源和菜单资源无法一起打包生成，但是生成工具又必须要这些文件，所以这里创建文件大小为 0 的空文件，以便能成功生成固件。

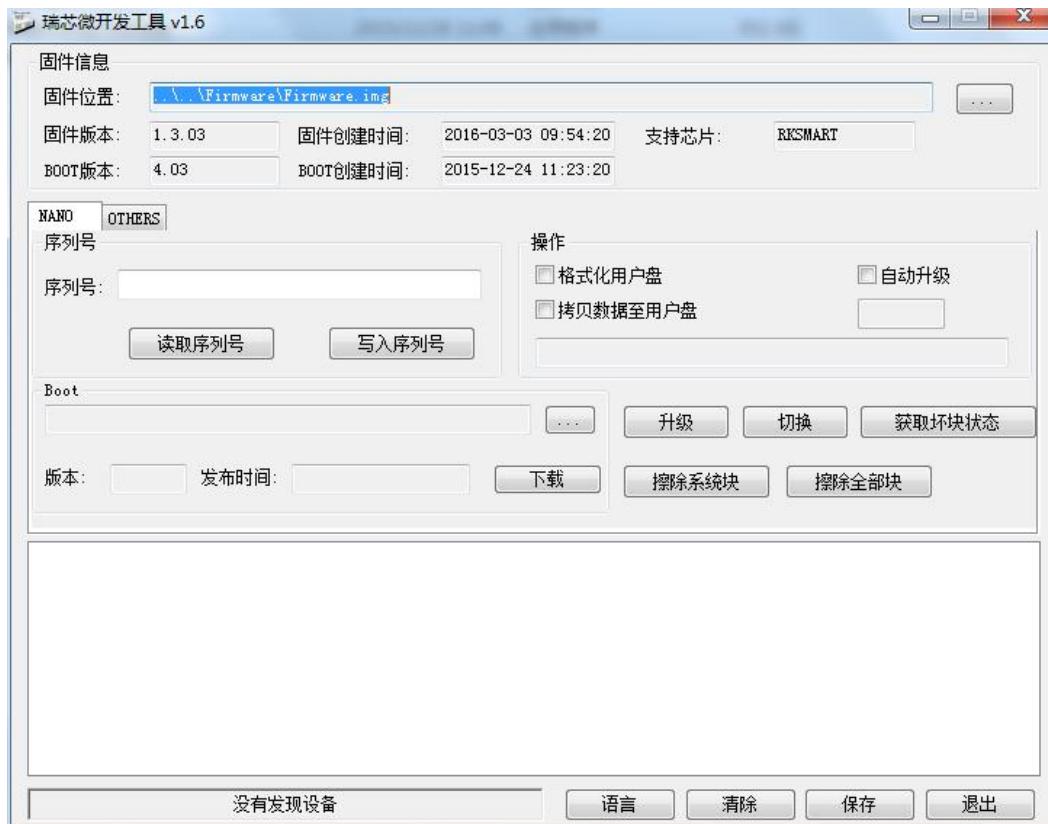


8.3 烧写固件

固件烧写工具根据使用用途的不同，分为两种。用户开发调试阶段的烧写工具，使用单机版的烧写工具，这种烧写工具只能一次识别、烧写一个设备。用户在开发完成后，在工厂批量生产时，需要使用工厂量产版的烧写工具，这种烧写工具可以一次识别、烧写 8 个设备。下面将分别做一下介绍。

8.3.1 单机烧写工具

下图 为 RKNanoD 的 单 机 固 件 烧 写 工 具 。 工 具 存 放 在“SDK_160_128\Development\firmware_upgrade” 中，RKDevelopTool.exe 打开界面如下：



固件位置：用来选择将要烧写到设备中的固件。用户可指定路径来选择不同的固件烧写。

默认固件路径存放在“..\..\Firmware\Firmware.img”

固件版本：生成工具界面中用户自定义的版本号，会在这里显示。

固件创建时间：显示最后一次固件生成的时间。

Boot 版本：显示最新的 boot 版本号。

Boot 创建日期: 显示 Boot 最后创建的日期。

序列号: 目前保留, NanoD 中未使用到。

切换: 用于切换升级模式, 在电脑连接 PC 识别到 MSC 设备时, 通过切换操作可以进入 MaskRom 升级模式。

升级: 下载更新固件。

擦除系统块: 擦除固件部分的存储块区域。

擦除全部块: 擦除所有的存储块区域。

8.3.2 工厂量产烧写工具

在目录 SDK_160_128\Development 中有一个目录 FactoryTool_v1.41, 该目录存放的是量产版的烧写工具。

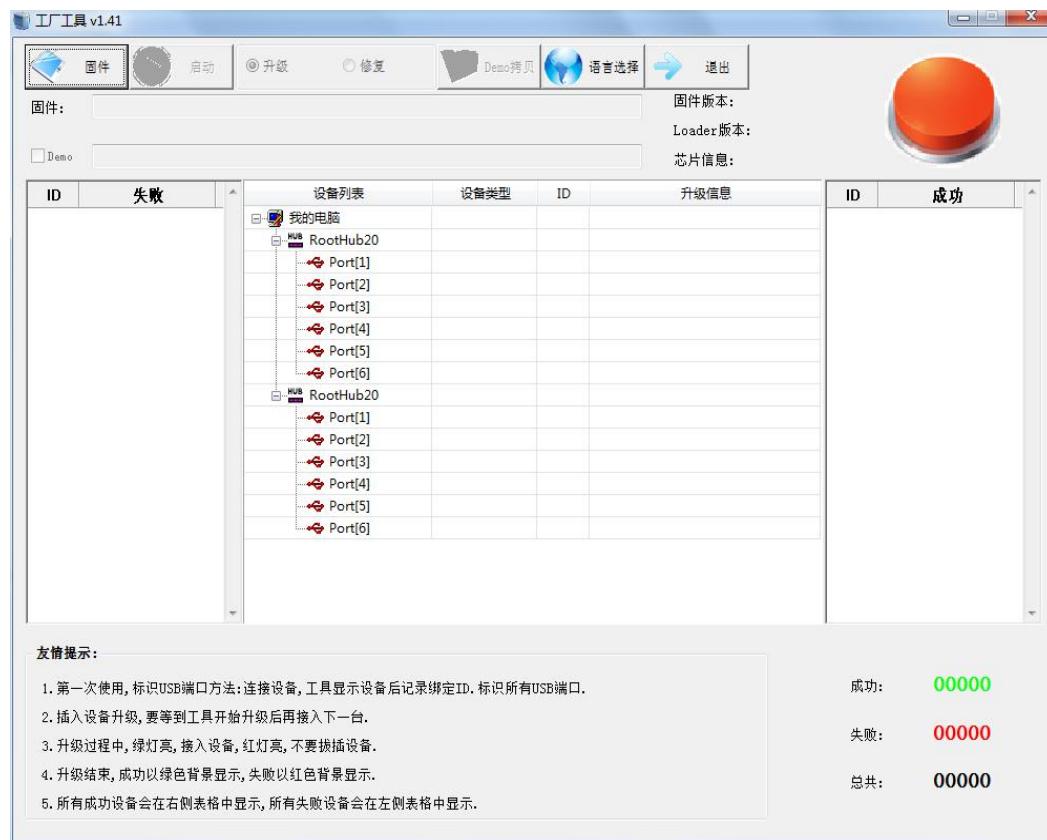
工具界面中重要功能按钮介绍如下:

固件: 用于指定加载需要烧写的固件。

启动: 当识别到设备插入, 会在设备列表中显示出来, 等所有设备都识别成功后, 选择启动去烧写固件。

注意:

- 量产版本的烧写工具需要根据 config.ini 来配置。详细的配置见该文件各个配置选项的说明。
- 如果用户的产品需要支持 SN (序列号) 功能, 需要根据用户自定义的 SN 定义规则生成 SN 工具。Rockchip 在 SDK_160_128\Development\SnDatTool 目录中提供了 SN 数据生成工具的一个 Demo VC 工程以及说明文档, 用户可根据自身需求做出修改, 编译该 VC 工程, 生成 SnDatTool.exe 工具, 使用该工具生成 sn.dat 数据文件, 将该文件放在 FactoryTool_v1.41 目录下, 配合量产工具使用即可。



使用 SN 数据生成工具的界面如下：



8.3.3 生成本地升级固件

有时因为修复了某方面的 bug，为了更方便的更新已发布出去的设备的固件，需要本地升级。而不是通过开发过程中的烧写工具或量产工具。这时候就要用到本地升级的固件。这里讲如何生成本地升级的固件。

生成新的固件以后，在..\\SDK_160_128\\Firmware 目录下，运行本地固件生成的批处理程序 GenUpdateFw.bat，调用 GenUFw.exe 生成本地升级的固件。此时的固件命名为 RKNANOFW.IMG。就可以将改固件发布出去，供用户升级。用户升级时，只需要将该 img 固件拷贝至 EMMC Flash 或 TF 卡（SPI FLASH）根目录即可，升级完成之后会重启，同时会删除该 img 文件。

注意：RKNANOFW.IMG 本地固件的命名不能更改，否则不能正确识别。

9 资源打包方法

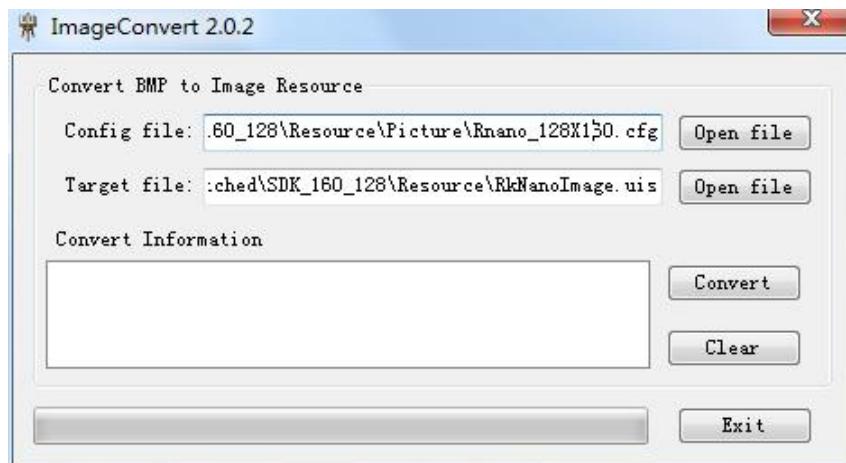
NanoD SDK 支持用户界面操作，这其中会显示图片与字符串。图片与字符串会作为固件的一部分打包进固件，保存在 flash 中。下面将分别介绍图片资源的打包与字符串资源的打包。

9.1 图片资源打包

图片资源存放在路径“..\SDK_160_128\Resource\Picture”目录下。所有的图片均为 24bit 位深像素点的 bmp 格式。注意图片命名格式，根据应用的不同，图片名称中需用括号标注图片的左上角起始坐标。

具体如：MasterName_SlaveName_(x, y).bmp。其中 MasterName 与 SlaveName 用来生成图片的 ID，会在程序中使用。x,y 为图片在 LCD 中起始左边。

图片资源打包工具使用 ImageConvert.exe, 打开工具后界面如下图所示：



Config file 选择 Picture 目录下的 Rnano_xxx.cfg 配置文件, 这里根据 LCD 显示屏的大小选择正确的配置文件。配置文件是组织需要打包的所有图片的列表，如图为打开 Rnano_128x160.cfg 的一部分的列表。

1	PowerOn0_(0,0).bmp
2	PowerOn1_(0,0).bmp
3	PowerOn2_(0,0).bmp
4	PowerOn3_(0,0).bmp
5	PowerOn4_(0,0).bmp
6	PowerOn5_(0,0).bmp
7	PowerOn6_(0,0).bmp
8	PowerOn7_(0,0).bmp
9	PowerOn8_(0,0).bmp
10	PowerOn9_(0,0).bmp
11	PowerOn10_(0,0).bmp
12	PowerOn11_(0,0).bmp
13	PowerOn12_(0,0).bmp
14	PowerOn13_(0,0).bmp

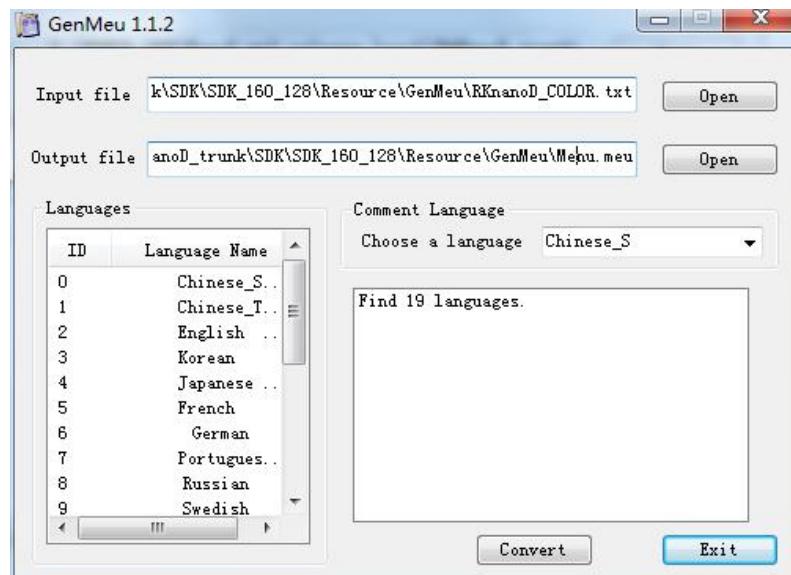
Target file 为需要生成的图片资源输出文件，名字可以有用户指定，后缀名为*.uis。

Convert: 开始转换的功能按钮。

图片资源转换工具从配置文件“Rnano_xxx.cfg”中依次获取图片名，当所有的图片完成打包后会生成文件“xxx.uis”这里默认使用“RKnanoImg.uis”和“ImageResourceID.h”。 “ImageResourceID.h”是用来保存图片的 ID 编号及 ID 名称。方便软件代码中使用 ID 编号来显示对于的图片。“RKnanoImg.uis”于固件一起打包存放在外部存储设备中。

9.2 字符串资源打包

字符串资源打包工具 GenMeu.exe 界面如下图所示：



其中，

Input file: 需要转换的 Unicode 文本文件。

Output file: 打包好的字符串数据文件。

Language: 显示语言名与 ID, 语言名与 ID 来自输入的文本文件。

Comment Language: 输出文件“MenuResourceID.h”中显示注释的语言

字符串是按照特定格式组织的 excel 文件。如下：

	A	B	C	D	E	F
1	/* MenuResourceID */	/* Chinese S */	/* Chinese T */	/* English */	/* Korean */	/* Japanese */
2	\$[X]MUSIC	\$[X]音乐播放	\$[X]音樂播放	\$[X]Music	\$[X]음악 재생	\$[X]音樂再生
3	\$[X]RADIO	\$[X]收音机	\$[X]收音機	\$[X]Radio	\$[X]FM	\$[X]FM
4	\$[X]RECORD	\$[X]录音	\$[X]錄音	\$[X]Record	\$[X]녹음	\$[X]錄音
5	\$[X]TEXT	\$[X]电子书	\$[X]電子書	\$[X]Text	\$[X]전자북 읽기	\$[X]オーディオブック読み
6	\$[X]EXPLORER	\$[X]资源管理器	\$[X]資源管理器	\$[X]Explorer	\$[X]파일 탐색	\$[X]エクスプローラー
7	\$[X]MAIN_SETTINGS	\$[X]设 置	\$[X]設 置	\$[X]Settings	\$[X]설정	\$[X]設定
8	\$[X]VIDEO	\$[X]视频播放	\$[X]視頻播放	\$[X]Video	\$[X]동영상 재생	\$[X]ビデオ再生
9	\$[X]PHOTO	\$[X]图片浏览	\$[X]圖片瀏覽	\$[X]Photo	\$[X]그림 보기	\$[X]フォトブラウズ
10	\$[X]FM_RADIO	\$[X]FM 收音机	\$[X]FM 收音機	\$[X]FM Radio	\$[X]FM 라디오	\$[X]FMラジオ
11	\$[X]GAME	\$[X]游 戏	\$[X]遊 戲	\$[X]Game	\$[X]게임	\$[X]ゲーム
12	\$[X]NOR	\$[X]NOR	\$[X]NOR	\$[X]NOR	\$[X]NOR	\$[X]NOR
13	\$[X]PLAYFX	\$[X]PlayFX	\$[X]PlayFX	\$[X]PlayFX	\$[X]PlayFX	\$[X]PlayFX
14	\$[X]ROCK	\$[X]ROCK	\$[X]ROCK	\$[X]ROCK	\$[X]ROCK	\$[X]ROCK
15	\$[X]POP	\$[X]POP	\$[X]POP	\$[X]POP	\$[X]POP	\$[X]POP
16	\$[X]CLASS	\$[X]CLASS	\$[X]CLASS	\$[X]CLASS	\$[X]CLASS	\$[X]CLASS
17	\$[X]BASS	\$[X]BASS	\$[X]BASS	\$[X]BASS	\$[X]BASS	\$[X]BASS
18	\$[X]JAZZ	\$[X]JAZZ	\$[X]JAZZ	\$[X]JAZZ	\$[X]JAZZ	\$[X]JAZZ
19	\$[X]USER	\$[X]USER	\$[X]USER	\$[X]USER	\$[X]USER	\$[X]USER
20	\$[X]UNKNOWN	\$[X]未知~	\$[X]Unknown~	\$[X]Unknown~	\$[X]Unknown~	\$[X]Unknown~
21	\$[X]NO_LYRIC_FILE	\$[X]没有歌词文件~	\$[X]沒有歌詞文件~	\$[X]No Lyric File~	\$[X]No Lyric File~	\$[X]No Lyric File~
22	\$[X]MANUAL	\$[X]手动	\$[X]手動	\$[X]Manual	\$[X]수동	\$[X]Manual
23	\$[X]RESET	\$[X]预设	\$[X]預設	\$[X]Reset	\$[X]자동 채널	\$[X]Preset
24	\$[X]MEM	\$[X]保存	\$[X]存儲	\$[X]Add Preset	\$[X]저장	\$[X]Mem
25	\$[X]DEL	\$[X]删除	\$[X]刪除	\$[X]Delete Preset	\$[X]삭제	\$[X]Del
26	\$[X]AUTO	\$[X]自动	\$[X]自動	\$[X]Auto Preset	\$[X]자동 선택	\$[X]Auto
27	\$[X]REC	\$[X]FM录音	\$[X]FM錄音	\$[X]FM Record	\$[X]FM 녹음	\$[X]FM記録
28	\$[X]PLAY_REC_DATA	\$[X]播放录音	\$[X]播放錄音	\$[X]Play Rec Data	\$[X]플레이 기록	\$[X]録音を再生
29	\$[X]DELETE_REC_DATA	\$[X]删除录音	\$[X]刪除錄音	\$[X]Delete Rec Data	\$[X]기록 삭제	\$[X]レコード中 [X]简 [X]中 [X]简
30	\$[X]DELETE_ALL	\$[X]全部删除	\$[X]全部刪除	\$[X]Delete All	\$[X]모두 삭제	\$[X]すべて削除

	A	B	C	D	E	F
109	\$[X]VOLUME_LIMIT_LINE2	\$[X]volume Level	\$[X]Volume Level	\$[X]Volume Level	\$[X]Volume Level	\$[X]Volume Level
110						
111	# [1]SETTINGS	# [1]设 置	# [1]設 置	# [1]Settings	# [1]설정	# [1]設定
112	###[1]SETTING_MUSIC	###[1]放音设置	###[1]放音設置	###[1]Music Settings	###[1]재생설정	###[1]再生設定
113	####[1]MUSIC_SHUFFLE	####[1]随机播放	####[1]隨機播放	###[1]Shuffle	###[1]Shuffle	###[1]Shuffle
114	####[0]MUSIC_SHUFFLE_OFF	####[0]关闭随机播放	####[0]關閉隨機播放	####[0]Shuffle Off	####[0]Shuffle Off	####[0]Shuffle Off
115	####[0]MUSIC_SHUFFLE_ON	####[0]开启随机播放	####[0]開啟隨機播放	####[0]Shuffle On	####[0]Shuffle On	####[0]Shuffle On
116	####[1]MUSIC_REPEAT_MODE	####[1]重复设置	####[1]重複設置	###[1]Repeat	###[1]반복모드	###[1]リピート設定
117	####[0]MUSIC_FOLDER_ONCE	####[0]关闭	####[0]Off	####[0]Off	####[0]Off	####[0]Off
118	####[0]MUSIC_FOLDER_REPEAT	####[0]目录重复	####[0]目錄重複	####[0]Repeat	####[0]폴더 반복 재생	####[0]目次リピート
119	####[0]MUSIC_REPEAT_ONE	####[0]单曲重复	####[0]單曲重複	####[0]Repeat 1 Song	####[0]한곡 반복 재생	####[0]シングルリピート
120	####[1]EQ_SELECT	####[1]EQ选择	###[1]EQ選擇	###[1]Equalizer	###[1]Equalizer	###[1]Equalizer
121	####[0]EQ_BASS	####[0]重低音	####[0]重低音	###[0]BASS	###[0]BASS	###[0]BASS
122	####[0]EQ_HEAVY	####[0]重金属	####[0]重金屬	###[0]Heavy	###[0]Heavy	###[0]Heavy
123	####[0]EQ_POP	####[0]流行	####[0]流行	###[0]Pop	###[0]Pop	###[0]Pop
124	####[0]EQ_JAZZ	####[0]爵士	####[0]爵士	###[0]Jazz	###[0]Jazz	###[0]Jazz
125	####[0]EQ_UNIQUE	####[0]独特	####[0]獨特	###[0]Unique	###[0]Unique	###[0]Unique
126	####[0]EQ_CUSTOM	####[0]用户自定	####[0]用戶自定	###[0]Custom	###[0]User EQ	###[0]User EQ
127	####[0]EQ_CUSTOM_EDIT	####[0] 编辑	###[0] Edit	###[0] Edit	###[0] Edit	###[0] Edit
128	####[0]EQ_NONE	####[0]正常	####[0]正常	###[0]Normal	###[0]Normal	###[0]Normal
129	###[1]SETTING_RADIO	###[1]收音设置	###[1]收音設置	###[1]FM Settings	###[1]라디오설정	###[1]ラジオ設定
130	###[1]RADIO_SCAN_SENSITIVITY	###[1]扫描敏感度	###[1]掃描敏感度	###[1]Scan Sensitivity	###[1]Scan Sensitivity	###[1]Scan Sensitivity
131	###[0]SCAN_SENSITIVITY_HIGH	###[0]高敏感度	###[0]高敏感度	###[0]High	###[0]High	###[0]High
132	###[0]SCAN_SENSITIVITY_LOW	###[0]低敏感度	###[0]低敏感度	###[0]Low	###[0]Low	###[0]Low
133	###[1]RADIO_STEREO_SWITCH	###[1]立体声开关	###[1]立體聲開關	###[1]Stereo Switch	###[1]스테레오 설정	###[1]ステレオ 中 [X]简 [X]中 [X]简
134	###[0]STEREO_SWITCH_OFF	###[0]关	###[0]關	###[0]Off	###[0]OFF	###[0]Off
135	###[0]STEREO_SWITCH_ON	###[0]开	###[0]開	###[0]On	###[0]ON	###[0]On

第一列字符串定义字符串的 ID 名称，其他列为多国语言的字符串。

第一行字符串用 “/@” 和 “@/” 是语言名称。

以 “[X]” 开始的字符串是用来在 LCD 上显示的字符串信息。使用 “##[1]” 或 “###[0]” 是设置菜单字符串。

“#” 的个数提示了设置菜单的级数关系，比如：

“#”：一级菜单

“##”：二级菜单

“###”：三级菜单

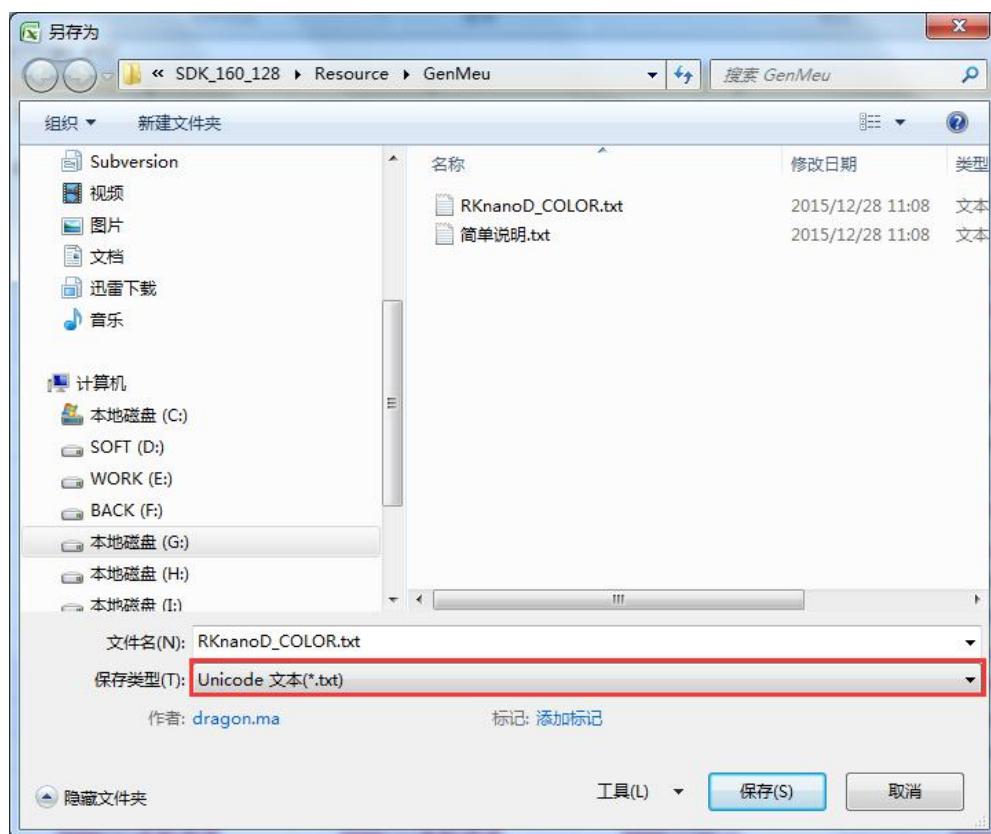
“[]” 中的数字表示了该菜单项下是否有子菜单：

1：有子菜单

0：没有子菜单

在使用 GenMenu.exe 工具之前，必须将 excel 文件转换为 Unicode 文件文件，方能使用。

选择 excel 开始菜单，选择 “另存为...” 保存类型选择 Unicode 文本选项，保存即可。如图：

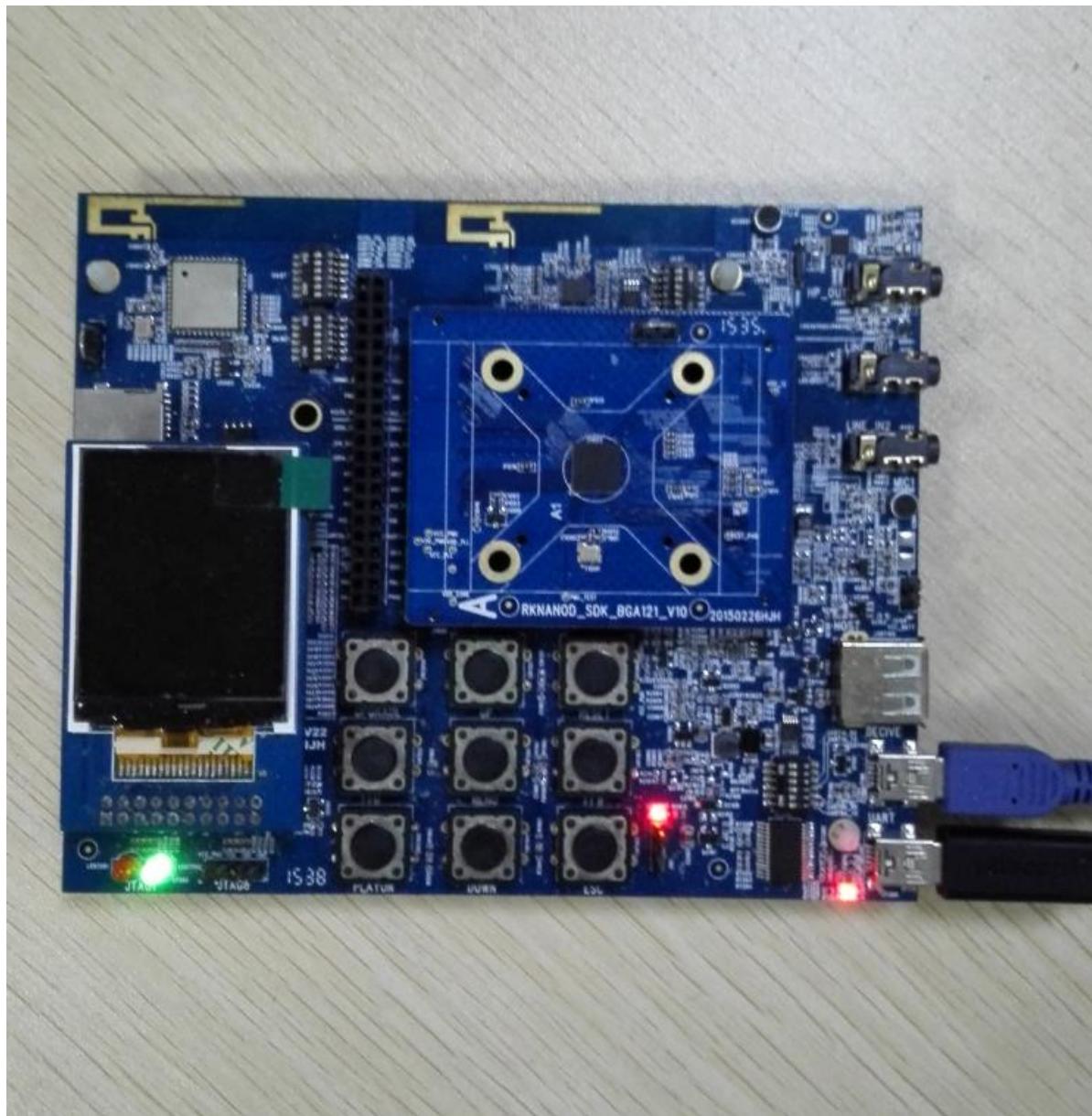


10 BT 的配置

10.1 硬件介绍

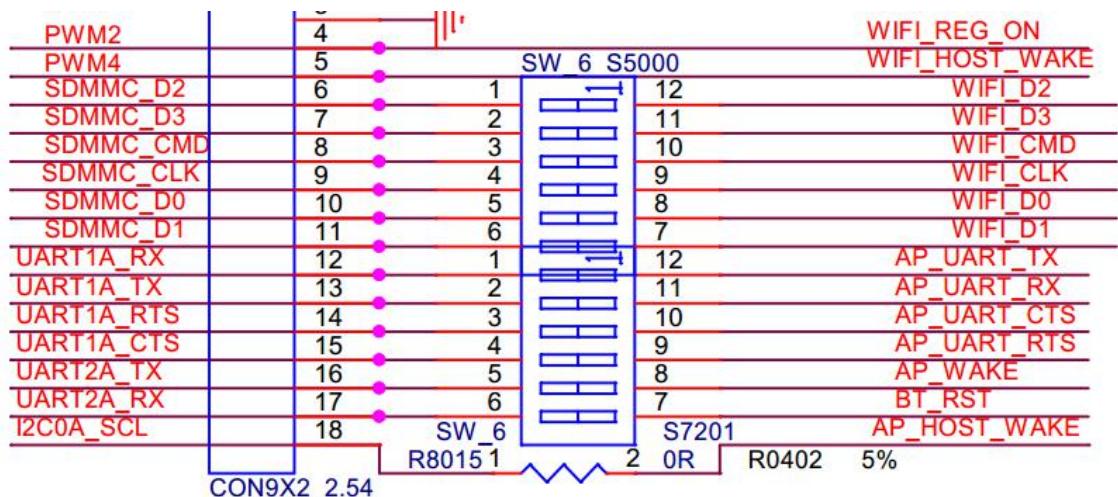
下图是使用 V22 底板(20150908HJH)，BGA 封装的顶板的 EVK 板。

BT 使用 Realtek 8761。

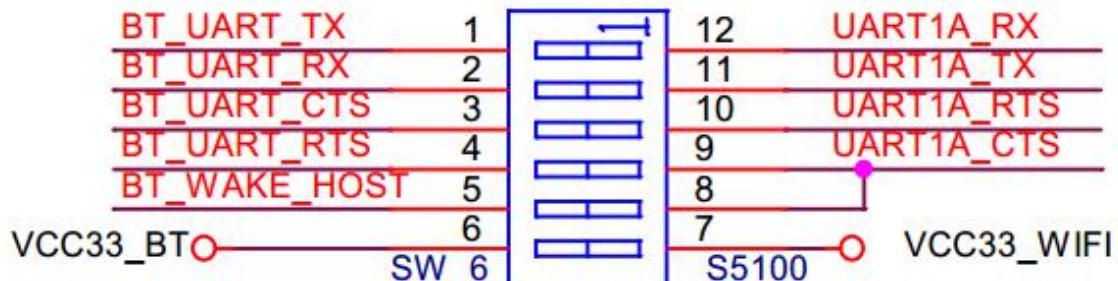


如果需要支持 BT 发射功能，EVK 板需要注意上图中的拨置开关的位置。

左上两个：全部拨置右边。

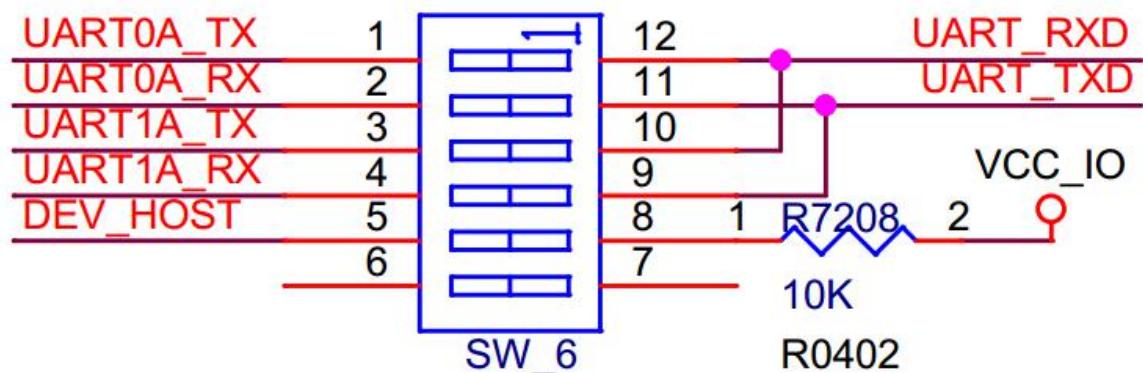


右上一个：编号 1, 2,6 的拨置右边，3,4,5 拨置左边。



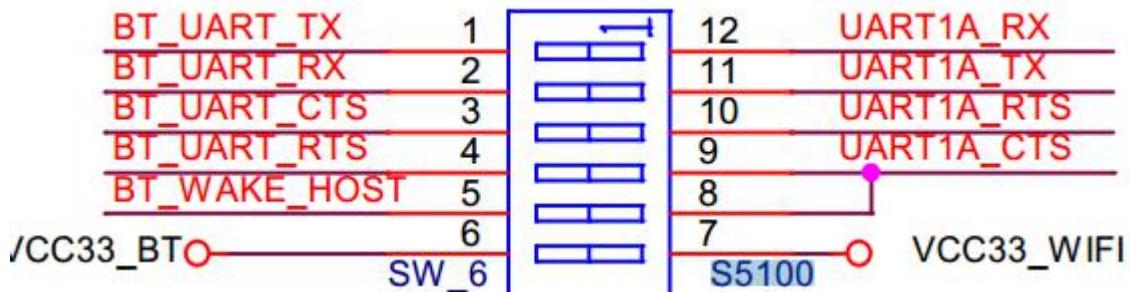
右下一个：1,2 拨置左边作为串口 0 输出打印，其他 4 个拨置左边。

S7200



注意：

此处需要提醒及注意的是，当使用 QFN 68PIN 顶板时，编号为 S5100 的拨置开关 1,2 需要拨置左边，作为普通 Uart1A 使用，不然会影响 FM 功能的正常使用。如下图：



QFN 68pin FM 使用 I2C 0B 通道。其他封装使用 I2C 1A 通道。



10.2 软件配置

如果需要使用到 BT 发射功能，需要打开 BT 的宏定义开关，在 sysconfig.h 中，可以看到#define _BLUETOOTH_ 宏定义。

```
//configer bluetooth function.
#define BT_CHIP_CC2564      0
#define BT_CHIP_CC2564B     1
#define BT_CHIP_RTL8761     2

#define BT_CHIP_CONFIG      BT_CHIP_RTL8761
//#define BT_CHIP_CONFIG      BT_CHIP_CC2564
#define BT_ENABLE_SET_MAC
#define BT_UART_INTERFACE_H4  1
#define BT_UART_INTERFACE_H5  2
#define BT_UART_INTERFACE_CONFIG BT_UART_INTERFACE_H5
```

上图中配置了蓝牙芯片相关的定义，目前 EVK 开发板使用 Realtek 8761 芯片，使用支持 H5 的 Uart 接口。

```
#define _A2DP_SOURCE_ // A2DP_SINK_ and _A2DP_SOURCE_ can't support at same project, choose one of them
#ifndef _A2DP_SOURCE_
#define AVRCP_
#define SBC_ENCODE_
#define BT_OFF_TIME_OUT      500//tick
#define BT_CONNECT_TIME_OUT  2000//tick
#define BT_LINK_SUPERVISION_TIMEOUT 20//second MAX is 40s
#define SSRC                  //for bluetooth function
#define PIN_CODE_WIN
#define NUMERIC_REQ_WIN
#define AVRCP_VERSION_1_4_
//#define BT_HOST_SNIFF
//#define HAVE_BLE
#endif
```

上图中配置了蓝牙发射支持的一些特性：如支持 avrcp，支持 SBC 编码，蓝牙关闭时的超时，蓝牙连接时的超时，SSRC（重采样），简单配对时的 PIN 码输入、PIN 码密钥确认、支持 v1.4 版本的 avrcp 等。

#define BT_VCC_ON_GPIO_CH	GPIO_CH2
#define BT_VCC_ON_GPIO_PIN	GPIOPortA_Pin2
#define BT_POWER_GPIO_CH	GPIO_CH0
#define BT_POWER_GPIO_PIN	GPIOPortB_Pin5
#define BT_HOST_RX_CH	GPIO_CH2
#define BT_HOST_RX_PIN	GPIOPortC_Pin1
#define BT_HOST_TX_CH	GPIO_CH2
#define BT_HOST_TX_PIN	GPIOPortC_Pin0
#define BT_HOST_CTS_CH	GPIO_CH2
#define BT_HOST_CTS_PIN	GPIOPortB_Pin7
#define BT_HOST_RTS_CH	GPIO_CH2
#define BT_HOST_RTS_PIN	GPIOPortB_Pin6
#define BT_HCI_UART_ID	UART_CH1
#define BT_UART_CH	UART_CH1_PA
#define BT_UART_INT_ID	INT_ID_UART1
#define BT_GPIO_INT_ID	INT_ID_GPIO0
#define BT_HCI_SERVER_INT_ID	INT_ID_UART5
#define BT_H5_TX_INT_ID	INT_ID_UART3
//#define BT_HOST_SNIFF	
#define BT_SBC_PROCESS_INT_ID	INT_ID_UART2

上图中定义了一些蓝牙芯片使用到的 GPIO 管脚定义。需要注意的是，BT 模组使用了 UART 1A 通道作为 HCI 层数据传输使用，如果需要串口打印，要避免使用 UART 1A 作为串口打印口，可改为使用 UART0A 作为打印口。BT 模块重定义了一些软中断 ID，比如 HCI Server 中断使用了 UART5 的中断。

10.3 操作说明

首次使用 EVK 板上的蓝牙发射功能时可能对操作不太熟悉，下文将对蓝牙发射功能，在 EVK 板上的操作做一简单说明。

首先进入 Mainmenu 界面的 Setting 界面，进入 Setting 功能列表，看到 Bluetooth Setting，这就是蓝牙发射功能的入口。代码中详见 SetBluetooth.c、SetBluetooth.h。进入蓝牙设置功能条目之后会出现三个功能条目项。BT ON、BT OFF、BT Information。

● BT ON

打开蓝牙。如果之前没有打开过蓝牙，以及保存过已配对的设备， 默认蓝牙打开以后会自动扫描周围打开的蓝牙设备，扫描之后会将扫描到的设备列出（扫描列表），设备名过长会自动滚动显示，上下键选择后， menu 键确认连接改设备，连接设备成功后，会进入已配对设备列表，此时已配对的设备名称前会出现耳机加蓝牙的图标。

手动扫描：

手动扫描可以使用 PLAYON 键。

删除已配对设备：

设备列表界面，选择一个已配对设备名称，按 FFD 键会弹出是否要删除已配对设备对话框。

显示已配对设备信息：

设备列表界面，选择一个已配对设备名称，按 FFW 键会弹出已配对设备的信息。ESC 键返回已配对设备列表。

断开已配对设备：

设备列表解决，按下 Menu 键弹出是否断开已连接设备的对话框，左右键选择 yes no。

连接过程中取消连接：

在设备列表界面，按下 Menu 键连接某个设备，在设备连接过程中又想取消连接，再次按 Menu 键。

● BT OFF

关闭蓝牙，如果蓝牙已配对连接，会断开连接，删除蓝牙线程。

- BT Information

显示本地 BT 设备的一些信息，本地设备名称、Mac 地址、A2DP\AVRCP 等版本号信息。

11 USB Host

NanoD SDK 支持 USB Host 功能。

硬件方面：

使用 USB Host 功能时，如使用 USB 供电，等系统起来以后，需将 V22 底板右下角的拨置开关 5、6 拨到左边，使能 USB Host 功能。此时插入 U 盘等具有 Host 功能的存储设备，即可识别。

软件方面：

Sysconfig.h 配置文件中打开 _USB_HOST_ 宏定义。