XR806 SDK

快速入门指南

版本号：1.1

发布时间：2020-11-25

版本历史

版本	日期	责任人	版本描述
1.0	2020-10-12	AWA 0498	创建文档。
1.1	2020-11-25	AWA 0498	补充评估板硬件配置。

目录

图片目录

表格目录

1. 前言

1.1. 文档简介

本文档介绍了基于 XR806 SDK 进行开发的必备基础知识，包括开发环境搭建、SDK 编译和烧录、工程配置等。

1.2. 目标读者

使用 XR806 SDK 的开发人员。

1.3. 适用范围

此文档适用于 XR806 SDK，支持 XR806 系列芯片产品。

1.4. 文档约定

1.4.1. 标志说明

本文档采用各种醒目的标志来表示在操作过程中应该特别注意的地方，这些标志的含义如下：

标识	说明
警告	该标志后的说明应给予格外关注，如果不遵守，可能会导致人员受伤或死亡。
注意	提醒操作中应注意的事项。不当的操作可能会损坏器件，影响可靠性、降低性能等。
说明	为准确理解文中指令、正确实施操作而提供的补充或强调信息。
窍门	一些容易忽视的小功能、技巧。了解这些功能或技巧能帮助解决特定问题或者节省操作时间。

1.4.2. 地址与数据描述方法约定

本文档在描述地址、数据时遵循如下约定：

符号	例子	说明
0x	0x0200，0x79	地址或数据以 16 进制表示。
0b	0b010，0b00 000 111	数据采用二进制表示(寄存器描述除外)。
X	00X，XX1	数据描述中，X 代表 0 或 1。 例如，00X 代表 000 或 001；XX1 代表 001，011，101 或 111。

1.4.3. 数值单位约定

本文档在描述数据容量（如 NAND 容量）时，单位词头代表的是 1024 的倍数；描述频率、数据速率等时则代表的是 1000 的倍数。具体如下：

类型	符号	对应数值
数据容量（如 NAND 容量）	1 K	1024
	1 M	1 048 576
	1 G	1 073 741 824
频率，数据速率等	1 k	1000
	1 M	1 000 000
	1 G	1 000 000 000

2. 概述

XR806 是一颗高集成度无线应用 MCU，其集成了 ARMv8-M 内核、IEEE 802.11b/g/n Wi-Fi 子系统、BLE 5.0 子系统、电源管理系统、高级别的安全系统以及丰富的外设接口，具有优秀的射频性能、稳定性、可靠性和超低功耗。

芯之联软件开发平台，具有丰富的组件和灵活的应用框架，可满足用户对 Wi-Fi、BLE、低功耗、安全等多方面的需求，可协助用户快速开发智能产品应用，包括物联网（IoT）、智能家居、云连接等。

3. 开发环境搭建

3.1. 硬件环境

以评估板为例介绍：XR806 评估板使用 USB 或锂电池供电，通过串口烧录固件，评估板自带 USB 转串口功能，使用时需要先安装 CP2102N 串口驱动。

图 3-1 XR806 评估板

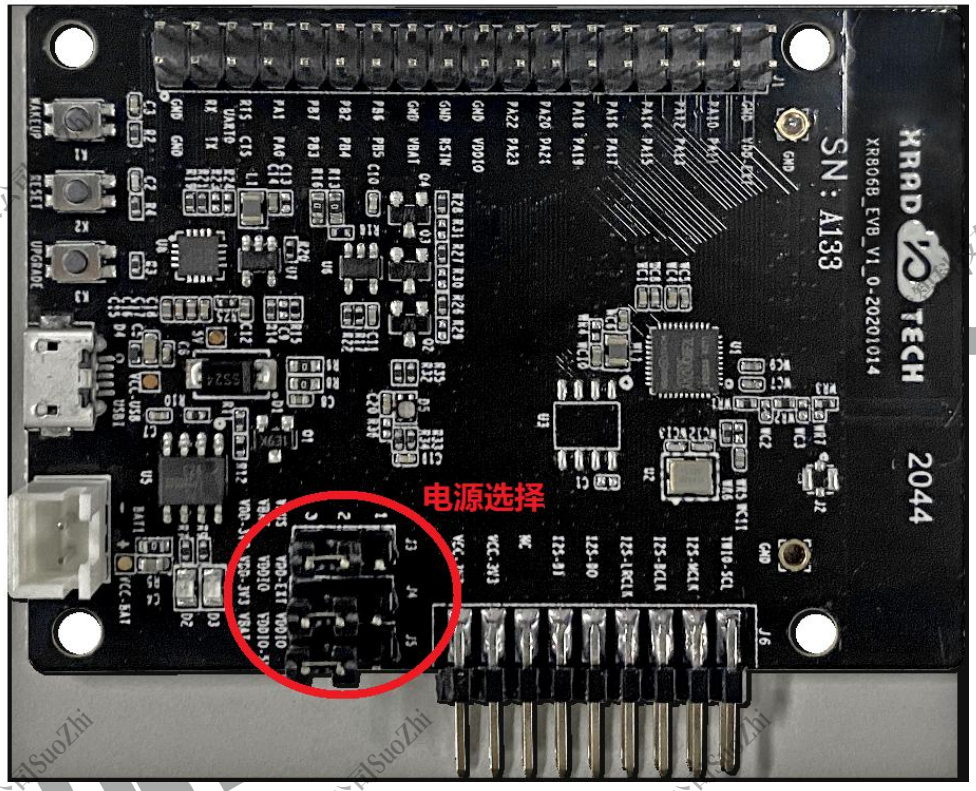


图 3-2 XR806 评估板电源选择

IC供电	供电方式	跳线帽选择
VBAT	VBUS(5V)	connect J3的1,2脚
	VDD-3V3	connect J3的2,3脚
VDDIO	VDD-EXT	connect J4的1,2脚
	VDD-3V3	connect J4的2,3脚
VDDIO-5V	VDDIO	connect J5的1,2脚
	VBAT	connect J5的2,3脚

在使用时，通常设置 VDD-EXT 输出 3.3V，并给 VDDIO 供电，VDDIO-5V 也采用 3.3V 供电，此时可通过 VBAT 测试整个系统（包括使用 VDD-EXT 和 VDDIO 供电的外设）的功耗。使用电池供电时，J3 跳 1 和 2，J4 跳 1 和 2，J5 跳 1 和 2 脚。

3.2. 获取 SDK

XRADIO SDK 托管于 GitHub，可通过以下命令获取 SDK 代码。

```
# 配置 git 在代码检出时不进行换行符转换（可选项）
$ git config --global core.autocrlf false

# 克隆仓库
$ git clone https://github.com/XradioTech/xr806_sdk.git
```

注意

Windows 某些 git 工具默认配置为代码检出时自动将文件换行符转换为 dos 格式，将会导致 SDK 中的 shell 脚本运行失败，需在克隆仓库前先将本地 git 配置为代码检出时不进行换行符转换。

3.3. 获取 Toolchain

XRADIO SDK 支持以下两种开发环境：

- Windows 环境：Cygwin + GCC
- Linux 环境：Ubuntu14.2 以上 + GCC

XRADIO SDK 使用的 Toolchain: gcc-arm-none-eabi-8-2019-q3-update

- Windows 版本

https://developer.arm.com/-/media/Files/downloads/gnu-rm/8-2019q3/RC1.1/gcc-arm-none-eabi-8-2019-q3-update-win32.zip?revision=2f0fd855-d015-423c-9c76-c953ae7e730b?product=GNU%20Arm%20Embedded%20Toolchain,ZIP,,Windows,8-2019-q3-update

- Linux 版本

https://developer.arm.com/-/media/Files/downloads/gnu-rm/8-2019q3/RC1.1/gcc-arm-none-eabi-8-2019-q3-update-linux.tar.bz2?revision=c34d758a-be0c-476e-a2de-af8c6e16a8a2?product=GNU%20Arm%20Embedded%20Toolchain,64-bit,,Linux,8-2019-q3-update

3.4. Linux Toolchain 安装与配置

1. 下载 Linux 版本的 Toolchain 压缩包“gcc-arm-none-eabi-8-2019-q3-update-linux.tar.bz2”，并保存至“~/tools”目录下（若“~/tools”目录不存在，则需先创建）。
2. 进入控制台终端，将 Toolchain 压缩包解压。

解压方法如下：

```
# 切换到 Toolchain 压缩包所在目录，例如为~/tools 目录
$ cd ~/tools

# 解压
$ tar -jxf gcc-arm-none-eabi-8-2019-q3-update-linux.tar.bz2
```

以上操作完成 Linux 环境下 Toolchain 的安装，且 Toolchain 安装目录与“[sdk]/gcc.mk”中的“CC_DIR”变量一致（[sdk]表示 SDK 根目录）。

```
CC_DIR = ~/tools/gcc-arm-none-eabi-8-2019-q3-update/bin
```

如果将 Toolchain 安装在其他目录，则需修改“[sdk]/gcc.mk”中的“CC_DIR”变量，使之指向实际的 Toolchain 安装目录。

3.5. Windows Toolchain 安装与配置

3.5.1. Cygwin 开发环境安装

1. 从以下链接下载安装文件“setup-x86_64.exe”或者“setup-x86.exe”：http://cygwin.com/install.html
2. 运行“setup-x86_64.exe”或者“setup-x86.exe”进行安装。

注意

安装路径不能包含中文字符或者空格。

3. 选择需要安装的工具包。必须安装以下工具：binutils、ncurses、libncurses-devel、intl、gcc-core、make、sed、awk、git、xz、python、unzip（可以使用 https://mirrors.aliyun.com/cygwin/ 进行库的下载）。
4. 安装完成后会在桌面产生“Cygwin”图标，双击图标即可进入 Cygwin 环境。XR806 SDK

3.5.2. Windows Toolchain 安装与配置

1. 下载 Windows 版本的 Toolchain 压缩包“gcc-arm-none-eabi-8-2019-q3-update-win32.zip”到目录“~/tools”下并解压。

说明

“~”表示 Cygwin 环境的当前用户主目录；如果 Cygwin 安装在“C:\”，则当前用户主目录“~”对应的路径为“C:\cygwin\home\[user-name]”（[user-name]为实际用户名）

以上操作完成 Windows 环境下 Toolchain 的安装，且 Toolchain 安装目录与“[sdk]/gcc.mk”中的“CC_DIR”变量一致（[sdk]表示 SDK 根目录）

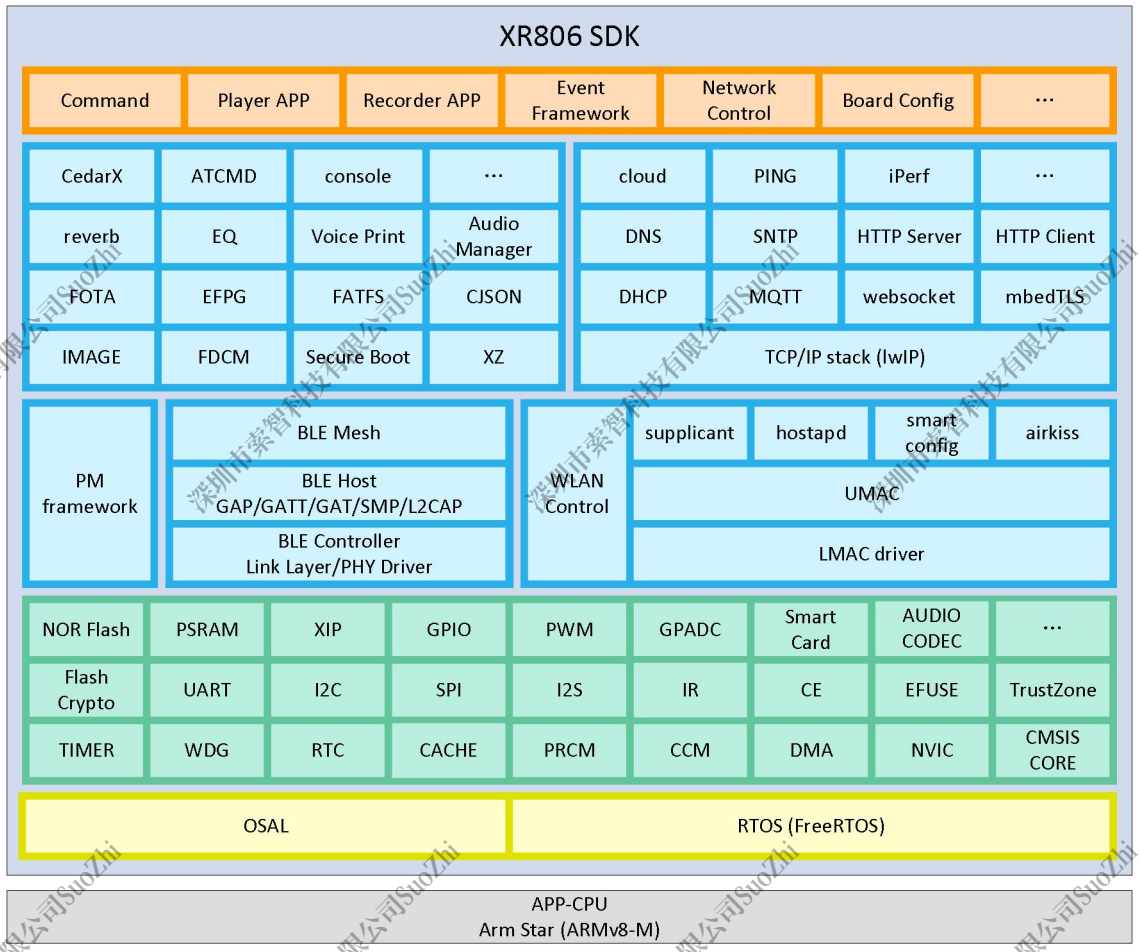
```
CC_DIR = ~/tools/gcc-arm-none-eabi-8-2019-q3-update/bin
```

如果将 Toolchain 安装在其他目录，则需修改“[sdk]/gcc.mk”中的“CC_DIR”变量，使之指向实际的 Toolchain 安装目录。

4. SDK 说明

4.1. 框架简述

图 4-1 XR806 SDK 框图



4.2. 目录结构

目录结构如下，其中 out 目录在编译时产生。

```
.
├── bin                    # bin 文件目录，存放预置 bin 文件
├── chip.mk
├── config.mk
├── gcc.mk
├── Kconfig
├── Makefile
├── include                # 头文件目录，存放模块对外头文件
└── lib                    # 库文件目录，存放预置库文件和编译生成的库文件
```

```
    ├──── libaac.a
    ├──── libadt.a
    └──── ......
├──── out                          # 编译生成文件存放目录，存放编译生成的 bin 和 image
├──── project                      # 工程总目录
    ├──── bootloader               # bootloader 工程
    ├──── common                   # 工程公用代码
    ├──── demo                     # 演示工程总目录，该目录下每个目录对应一个工程
    │   ├──── hello_demo
    │   ├──── wlan_demo
    │   └──── ......
    ├──── example                  # 示例工程总目录，该目录下每个目录对应一个工程
    │   ├──── uart
    │   ├──── wlan
    │   └──── ......
    ├──── image_cfg
    │   └──── image.cfg            # 默认镜像配置文件
    ├──── linker_script
    │   └──── gcc
    │       ├──── appos.ld         # 工程默认链接脚本
    │       └──── bootloader.ld    # bootloader 链接脚本
    ├──── project.mk
    ├──── Kconfig
    └──── ......
├──── src
    ├──── driver
    │   ├──── chip                 # 芯片外设驱动
    │   └──── component            # 扩展外设驱动
    ├──── image                    # image 模块
    ├──── kernel                   # 内核
    ├──── ota                      # OTA 模块
    └──── ......
├──── ......
└──── tools                        # 镜像打包及烧录等工具
```

4.3. 构建系统和配置文件

XRADIO SDK 的代码编译和镜像创建采用 Kconfig 和 Makefile 进行管理，Kconfig 和关键 Makefile 和配置文件的说明见表 4-1。除特殊说明外，下文所有路径描述均为相对于 SDK 根目录的相对路径。

表 4-1 SDK 关键 Makefile 和配置文件

文件路径	功能描述
Kconfig	1) 分布在各处，如顶层 Kconfig、prject/Kconfig、src/pm/Kconfig。 2) 用户可加入自己的配置，格式说明见 tools/config/Kconfig-language.rst。
MakeFile	顶层 Makefile，用于执行编译命令。
gcc.mk	1) 定义通用编译规则，例如交叉编译工具链、编译选项、链接选项等。 2) 一般情况下，用户只需要修改交叉编译工具链的路径定义“CC_DIR”。
config.mk	定义 SDK 使用的配置信息，不允许用户修改。
chip.mk	定义芯片配置选项，不允许用户修改。
src/Makefile	定义 SDK 模块，每个模块编译后均生成对应的库文件。
src/lib.mk	定义模块通用编译规则。
src/[module]/Makefile	1) 模块编译的 Makefile，用于指定模块的库文件名、源文件等。 2) [module]对应具体的模块路径，例如 image 模块的 Makefile 路径是“src/image/Makefile”。
project/project.mk	定义工程的通用编译规则，一般无需修改。
project/[prj]/gcc/defconfig	1) 定义特定工程的配置选项，可将此配置 copy 为顶层的.config，作为整个 SDK 的配置。 2) [prj]对应具体的工程路径，例如“hello_demo”工程对应的路径为“demo/hello_demo/”。
project/[prj]/gcc/Makefile	特定工程的 Makefile，用于指定工程源文件、库、链接脚本、镜像配置文件等。
project/common/prj_conf_opt.h	1) 定义工程功能选项的默认配置（即针对所有工程的默认配置）。每个功能选项对应一个宏定义，例如主线程栈大小（PRJCONF_MAIN_THREAD_STACK_SIZE）等。 2) 该文件中定义的宏，“project/”目录下的所有源文件均可见。 3) 一般情况下，用户不需要修改此文件。如果特定工程需要改变某功能选项的值，可在“project/[prj]/prj_config.h”中进行重定义，从而覆盖默认值。
project/[prj]/prj_config.h	1) 定义特定工程的功能选项配置值。在此文件中定义的功能选项值将会覆盖“project/common/prj_conf_opt.h”中定义的默认值。 2) [prj]对应具体的工程路径，例如“hello_demo”工程对应的路径为“demo/hello_demo/”。 3) 每个工程都必须实现本工程的“prj_config.h”文件。 4) 该文件中定义的宏，“project/”目录下的所有源文件均可见。

一般情况下，用户只需要修改以下文件来实现工程配置定义：

- project/[prj]/gcc/Makefile（用于指定工程源文件、库、链接脚本、镜像配置文件等）
- project/[prj]/gcc/defconfig（可用于覆盖顶层.config 中的默认配置）
- project/[prj]/prj_config.h（覆盖“project/common/prj_conf_opt.h”中的默认配置）

4.4. 代码编译和镜像创建

代码编译前需要在“gcc.mk”中设置正确的 GCC 交叉编译工具链路径，例如：

```
CC_DIR = ~/tools/gcc-arm-none-eabi-8-2019-q3-update/bin
```

所有代码编译和镜像创建命令均需在 Cygwin 环境或者 Linux 终端执行，在顶层目录编译。

1. 先 copy 一份配置作为默认配置。例如：

```
cp project/demo/hello_demo/gcc/defconfig .config
```

或

```
make PRJ=demo/hello_demo defconfig
```

代码编译和镜像创建命令的说明见表 4-2。

表 4-2 代码编译和镜像创建命令

命令	命令说明
make menuconfig	执行 SDK 基于图形界面的配置（如配置工程名，选择芯片型号、高频晶振等配置），该配置对所有工程生效。如果需要更改 SDK 基础配置，可再次执行此命令。 正确执行该命令后，将在 SDK 根目录下生成对应的“.config”文件。
make config	执行 SDK 基于命令行的配置（根据提示进行工程名，芯片型号、高频晶振等配置），该配置对所有工程生效。如果需要更改 SDK 基础配置，可再次执行此命令。 正确执行该命令后，将在 SDK 根目录下生成对应“.config”文件。 与 menuconfig 类似，但配置步骤较多，不建议使用。
make config_clean	删除“make config”命令生成的文件“.config”。
make oldconfig	使用上次的配置。
make PRJ=[prj_dir] defconfig	在顶层使用，需要带工程名，如 make PRJ=demo/hello_demo defconfig；作用和 cp project/demo/hello_demo/gcc/defconfig .config 一致。
make lib	编译“src”目录下的模块，生成库文件并复制到“lib”目录。
make lib_clean	删除“make lib”命令在“src”目录下生成的中间编译文件。
make lib_install_clean	删除“make lib”命令在“lib”目录下生成的库文件。
make	编译工程代码，生成可执行文件（位于“project/[prj]/gcc/”目录）。
make clean	删除“make”命令生成的所有文件。
make image	创建镜像文件（位于“out/”目录）。
make image_clean	删除“make image”命令生成的所有文件。
make build	相当于执行命令“make lib && make && make image”。
make build_clean	相当于执行命令“make image_clean clean lib_clean lib_install_clean”。
make objdump	生成反汇编文件，用于代码分析和调试，位于“out/”目录。
make size	make size显示工程 ELF 文件的 size 信息，如 text 段、data 段、bss 段大小等。

4.5. 应用示例

对“hello_demo”工程进行代码编译和镜像创建的常规过程，举例如下：

```
# 复制默认配置文件到顶层目录（不切换工程可不要此步骤）
$ cp project/demo/hello_demo/gcc/defconfig .config 或 make PRJ=demo/hello_demo defconfig

# 检查 SDK 基础配置，如工程名、芯片型号、高频晶振、板级配置是否正确
$ make menuconfig

# 清理，切换工程时需要
$ make build_clean

# 编译代码并生成镜像文件，生成的镜像文件为“out/xr_system.img”
$ make build(建议使用 make build -j 加速编译)
```

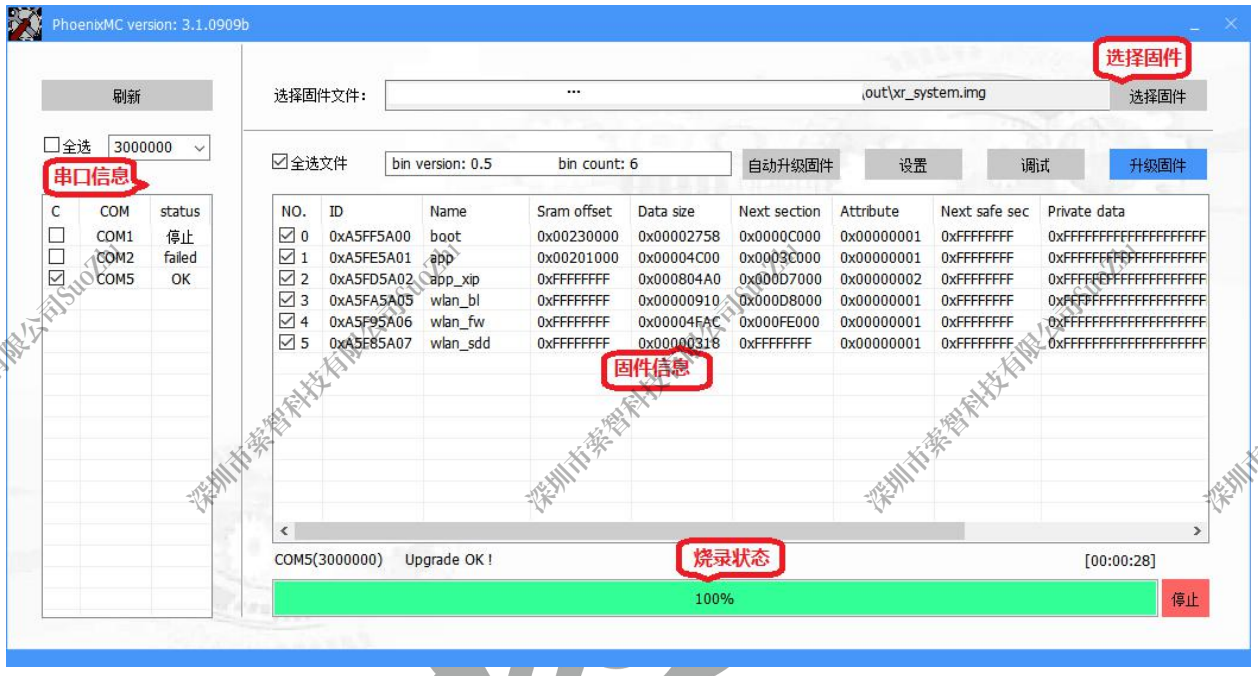
编译 bootloader 的过程如下：

```
# 复制默认配置文件到顶层目录
$ cp project/bootloader/gcc/defconfig .config 或 make PRJ=bootloader defconfig

# 检查 SDK 基础配置，如工程名、芯片型号、高频晶振、板级配置是否正确
$ make menuconfig

# 清理，切换工程时需要
$ make build_clean

# 编译代码并生成镜像文件，默认晶振是 40M，生成的镜像文件为
#“bin/xradio_v3/boot/xr806/boot_40M.bin”
$ make build(建议使用 make build -j 加速编译)
```

5. 镜像烧录

5.1. 镜像烧录

图 5-1 烧录工具界面



Windows 版本的烧录工具（GUI 版本）位于“tools/phoenixMC.exe”，Linux 版本的烧录工具（命令行版本，无 GUI 版本）位于“tools/phoenixMC”。本文只对 Windows 版本的 GUI 烧录工具使用进行简要介绍，详细说明可参考《XR806_PhoenixMC 工具_使用指南》。

PhoenixMC 的主要功能是通过串口将指定的合法固件文件（即前文提到的镜像文件，扩展名为“.img”）烧录到目标设备中，具体步骤如下：

1. 串口连线：将开发板上的 UART0 通过串口线连接到 PC，并进入升级模式（进入升级模式的方法见 5.2 节说明）。
2. 串口设置：点击左上角的“刷新”按钮可刷新已连接串口设备列表，勾选开发板对应的 COM 口。串口波特率最大支持 3000000，波特率越高，烧录速度越快。如果高波特率下容易出现烧录失败，可检查串口线、串口驱动是否稳定支持该波特率；或者降低波特率进行尝试。为了避免烧录速度过慢，建议波特率选择大于等于 921600。
3. 固件选择：点击“选择固件”按钮选择需要烧录的固件文件（扩展名为“.img”），固件信息栏会显示出当前固件的详细信息。另外，通过拖拽方式将固件直接拖入工具界面也可以达到同样的效果。
4. 启动烧录：点击“升级固件”按钮启动固件烧录。烧录状态栏显示当前选定串口对应设备的烧录进度和状态。当烧录成功时，进度条会达到 100%的进度并显示为绿色；当烧录失败时，进度条显示为红色并报告错误。
5. 复位设备：固件烧录成功后，开启 PC 串口工具并连接到 UART0，硬件复位开发板，将看到以下打印输出。

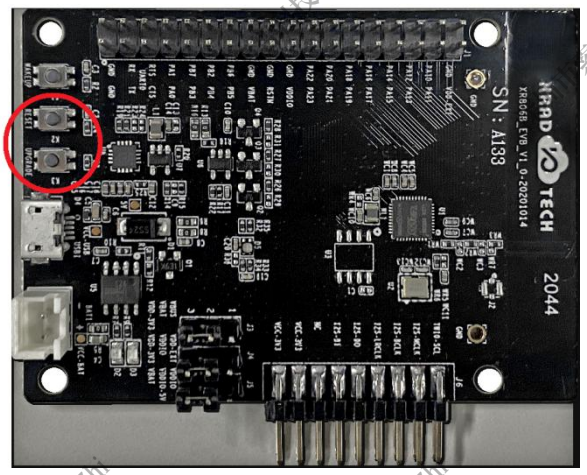
```
use default flash chip mJedec 0x0
[FD I]: mode: 0x2, freq: 48000000Hz, drv: 0
```

```
[FD I]: jedec: 0x0, suse default flash chip mJedec 0x0
[FD I]: mode: 0x4, freq: 48000000Hz, drv: 0
[FD I]: jedec: 0x0, suspend_support: 1
mode select:e

wlan information==================================================
firmware:
      version : R0-XR_C07.08.52.65_01.94 Sep 25 2020 20:43:25-Y01.94
      buffer    : 8
driver:
      version : XR_V02.05
mac address:
      in use            : 58:c1:74:cf:ee:6e
      in use            : 58:c1:74:cf:ee:6f
==================================================================

wlan mode:a

platform information ============================================
XR806 SDK V0.1 Oct 13 2020 14:16:47

heap space [0x217654, 0x24bc00), size 214444

cpu    clock 160000000 Hz
HF     clock    40000000 Hz

sdk option:
      XIP                : enable
      INT LF OSC         : enable
      SIP flash          : enable

mac address:
      efuse              : 00:00:00:00:00:00
      in use             : 58:c1:74:cf:ee:6e
==================================================================
```

5.2. 进入升级模式

1. 未烧录过固件的设备（FLASH 上无有效内容）在上电后自动进入升级模式。
2. 烧录过固件，且能够正常启动并进入控制台的设备，通过在控制台输入“upgrade”命令使设备进入升级模式（前提是控制台支持“upgrade”命令）。PhoenixMC 工具在点击“升级固件”按钮后，会默认发送一条“upgrade”命令；如果设备满足前述条件，则不需要手动在控制台输入“upgrade”命令也可以启动固件烧录。
3. 通过将 STRAP IO PB02 置为低电平并 RESET 设备，使设备进入升级模式，进入后需释放 PB02 IO 使其处于上拉状态。以评估板为例，先按住 UPGRADE 键（作用是将 PB02 置为低电平），再按 RESET，松开 RESET，松开 UPGRADE，即进入升级模式，按键位置如下：

图 5-2 进入升级模式



4. 在上电过程中短接 FLASH 的 MISO 信号到 GND，使系统启动失败后自动进入升级模式（如果是内部 sip flash，只能采用方法 3）。

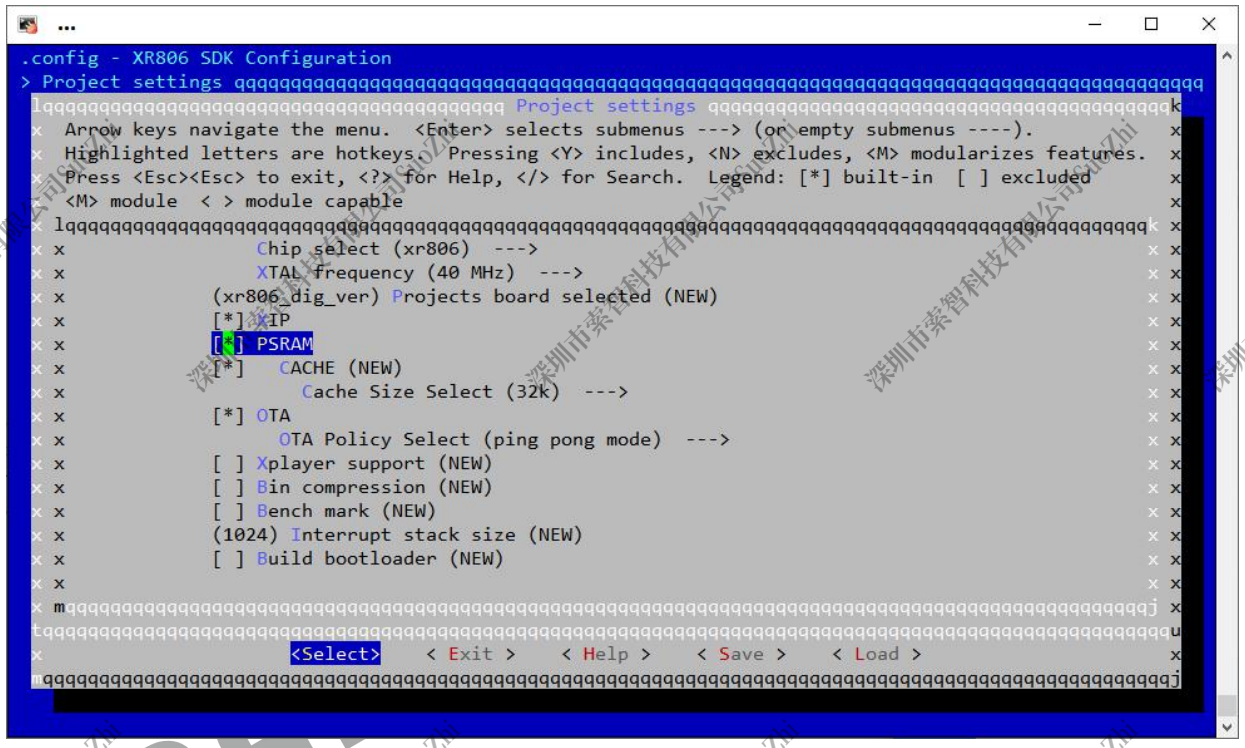
6. 工程配置

6.1. 配置选项

使用 make menuconfig 进入配置选择界面，已提供了 SDK 支持的大部分选项。

例如，要开启了 PSRAM 功能，选择如下：Project settings -> PSRAM。

图 6-1 配置选择界面



常用全局配置选项说明如表 6-1 所示。

表 6-1 常用全局配置选项

配置选项	配置说明
XIP	是否启用 XIP 功能。
PSRAM	是否启用 PSRAM（仅 XR806BM2I 支持 PSRAM）。
Cache Size Select	可选 8/16/32K；Cache 越大，性能越好，但可用 SRAM 会相应减少。
wlan station mode	是否支持 Wi-Fi station（不支持 WPS）模式。
wlan station mode with wps support	是否支持 Wi-Fi station（支持 WPS）模式。
wlan ap mode	是否支持 Wi-Fi softAP 模式。
wlan monitor mode	是否支持 Wi-Fi monitor 模式。
OTA 和 OTA Policy	是否支持 OTA 功能，以及 OTA 策略，当前支持乒乓和压缩升级 2 种方式。
xplayer	是否支持音频播控功能。
Trace heap memory usage and error	是否开启堆调试功能。该功能仅用于堆问题的调试，如内存泄漏等；开启该

配置选项	配置说明
	功能后，会多消耗约 8KB 内存。
Debug CPU usage	查看 CPU 占用率。

6.2. 工程 Makefile

工程 Makefile 位于“project/[prj]/gcc/Makefile”，主要用于指定工程的源文件、链接脚本、镜像配置文件等，关键变量的定义说明如表 6-2 所示。

表 6-2 工程 Makefile 变量

变量名	变量说明
ROOT_PATH	SDK 根目录路径，该路径是相对于工程 Makefile 的路径，例如“project/demo/hello_demo/gcc/Makefile”中定义的 ROOT_PATH 为“../../../..”。
PROJECT	工程名字，默认定义为跟工程目录名字一致（一般无需修改），即： PROJECT := $(notdir $(shell cd .. && pwd))
PRJ_ROOT_PATH	工程根目录，使用变量“ROOT_PATH”进行定义。
SRCS, OBJS	定义工程源文件和目标文件。
PRJ_EXTRA_LIBS_PATH	定义额外的库文件搜索路径。默认库文件搜索路径为“lib/”。
PRJ_EXTRA_LIBS	1) 定义额外的库文件。 2) 如果工程需要使用 SDK 以外的库文件，可通过“PRJ_EXTRA_LIBS_PATH”和“PRJ_EXTRA_LIBS”变量来指定。例如，需要链接工程根目录下的“$(PRJ_ROOT_PATH)/foo/libbar.a”，则增加定义如下： PRJ_EXTRA_LIBS_PATH := -L$(PRJ_ROOT_PATH)/demo/foo PRJ_EXTRA_LIBS := -lbar
PRJ_EXTRA_INC_PATH	定义额外的头文件搜索路径。例如，需要指定工程根目录下的“foo”目录为头文件搜索路径，则增加定义如下： PRJ_EXTRA_INC_PATH := -I$(PRJ_ROOT_PATH)/demo/foo
PRJ_EXTRA_SYMBOLS	定义额外的宏，仅工程代码可见，一般不使用。
LINKER_SCRIPT	1) 工程默认链接脚本为“project/linker_script/gcc/appos.ld”，特定工程可重定义“LINKER_SCRIPT”变量来指定该工程使用的链接脚本。 2) “LINKER_SCRIPT”的路径定义是相对路径（相对于“$(PRJ_ROOT_PATH)/gcc/”）。 3) 例如，工程需要使用“project/[prj]/gcc/appos.ld”链接脚本，则增加定义如下： LINKER_SCRIPT := ./appos.ld 4) 编译过程中，会根据默认链接脚本或者工程指定的链接脚本，生成最终使用的链接脚本“project/[prj]/gcc/.project.ld”。 5) 链接产生的 ELF 文件、bin 文件、map 文件等位于“project/[prj]/gcc/”目录下。
IMAGE_CFG	1) 工程默认镜像配置文件为“project/image_cfg/image.cfg”，特定工程可

变量名	变量说明
	重定义“IMAGE_CFG”变量来指定该工程使用的镜像配置文件。 2) “IMAGE_CFG”的路径定义是相对路径（相对于“$(PRJ_ROOT_PATH)/image/[chip]/”）。 3) 例如，工程需要使用“project/[prj]/image/xr806/image.cfg”镜像配置文件，则增加定义：IMAGE_CFG := ./image.cfg 4) 镜像创建过程中，会根据默认镜像配置文件或者工程指定的镜像配置文件，生成最终使用的镜像配置文件“project/[prj]/image/[chip]/.image.cfg”。 5) 创建的镜像文件位于“project/[prj]/image/[chip]/”目录下。
IMAGE_NAME	1) 工程默认镜像名字为“xr_system.img”，特定工程可重定义“IMAGE_NAME”变量来指定镜像名字（不含扩展名，扩展名固定为“img”）。 2) 例如，工程需要指定镜像名字为“foobar.img”，则增加定义：IMAGE_NAME := foobar

6.3. 链接脚本

链接脚本（扩展名为“.ld”）主要用于描述如何将输入文件（目标文件、库文件）中的段（text、data、bss 等）映射到输出文件（ELF 文件）中，并控制输出文件的存储布局，输出文件的存储布局采用输出段（链接脚本中以“SECTIONS”标记）进行描述。

工程的默认链接脚本为“project/linker_script/gcc/appos.ld”，特定工程可通过重定义“project/[prj]/gcc/Makefile”中的“LINKER_SCRIPT”变量来指定该工程使用的链接脚本。

工程默认链接脚本关键的输出段说明如表 6-3 所示。

表 6-3 链接脚本输出段

输出段	说明
.xip	1) XIP 段（可读、可执行），位于 FlASH 空间中。 2) 用于存储可执行代码和只读数据。在中断服务程序中执行的代码和访问的数据不能置于该输出段。
.psram_text	1) PSRAM text 段（可读、可执行），位于 PSRAM 空间中（仅 XR806BM2I 支持）。 2) 用于存储可执行代码和只读数据。
.psram_data	1) PSRAM data 段（可读、可写），位于 PSRAM 空间中（仅 XR806BM2I 支持）。 2) 用于存储初始值非零的全局变量、静态变量等，一般是输入文件的 data 段。
.psram_bss	1) PSRAM bss 段（可读、可写），位于 PSRAM 空间中（仅 XR806BM2I 支持）。 2) 用于存储初始值为零的全局变量、静态变量等，一般是输入文件的 bss 段。
.text	1) SRAM text 段（可读、可执行），位于 SRAM 空间中。 2) 用于存储可执行代码和只读数据。
.data	1) SRAM data 段（可读、可写），位于 SRAM 空间中。

输出段	说明
	2) 用于存储初始值非零的全局变量、静态变量等，一般是输入文件的 data 段。
.bss	1) SRAM bss 段（可读、可写），位于 SRAM 空间中。 2) 用于存储初始值为零的全局变量、静态变量等，一般是输入文件的 bss 段。

链接脚本中.xip 输出段的描述举例：

```
SECTIONS
{
    .xip :
    {
        *libaac.a: (.text .text.* .rodata .rodata.*)
        *libchip.a:hal_rtc.o (.text .text.* .rodata .rodata.*)
        *(.xip_text* .xip_rodata*)
} > FLASH
}
```

上文中的脚本代码进行了以下设置：

- 将 libacc.a 库文件的 text 段和 rodata 段置于.xip 输出段
- 将 hal_rtc.o 目标文件（位于 libchip.a 库文件中）的 text 段和 rodata 段置于.xip 输出段
- 将代码中标记为.xip_text 或者.xip_rodata 属性的段置于.xip 输出段

通过在输出段中添加输入文件的描述，可以将（目标文件、库文件）中的段（text、data、bss 等）映射到输出段中。所有输出段均遵循相同的语法描述。链接脚本详细的语法可以参考工程默认链接脚本或者 GUN ld 文件的语法说明文档。

在进行链接脚本修改时，有以下注意事项：

1. 在中断服务程序中执行的代码和访问的数据不能置于 FLASH 的.xip 和.psram 输出段。
2. 不同存储区域的访问速度存在差异，实际方案可根据算力需求进行存储布局的调整。例如：将执行不频繁、对速度要求不高的代码放在.xip 输出段；将执行频繁且对速度要求高的代码（例如某些算法）放在 SRAM 的.text 输出段。存储区域的访问速度由高到低为：SRAM、PSARM（仅 XR806BM2I 支持）、Flash。
3. 链接器按链接脚本中输出段的描述顺序，将输入文件的段置于指定输出段中；没有指定到 XIP 和 PSRAM 相关输出段的内容最后会默认置于 SRAM 相关输出段中。

6.4. 镜像配置文件

镜像配置文件（扩展名为“.cfg”）采用 JSON 格式书写，主要用于描述如何将工程编译生成的二进制文件（app.bin、app_xip.bin 等）和预置的二进制文件（wlan_bl.bin、wlan_fw.bin 等）打包成镜像文件（扩展名为“.img”）。

工程的默认镜像配置文件为“project/image_cfg/image.cfg”，特定工程可通过重定义“project/[prj]/gcc/Makefile”中的“IMAGE_CFG”变量来指定该工程使用的镜像配置文件。

表 6-4 中有关于“IMAGE_CFG”变量的详细说明。

镜像文件由“tools/mkimage”工具生成。通过烧录工具“tools/phoenixMC.exe”可将镜像文件烧录到开发板的 FLASH 中运行。

图 6-1 镜像配置文件

```
{
    "magic" : "AWIH",
    "version" : "0.5",
    "OTA" : {"addr": "1024K", "size": "4K"},
    "image" : {"max_size": "1020K"},
    "section" :
    [
        {"id": "0xa5ff5a00", "bin": "boot_40M.bin", "cert": "null", "max_len": "16K", "sram_offs": "0x00230000", "ep": "0x00230101", "attr": "0x01"},
        {"id": "0xa5fe5a01", "bin": "app.bin", "cert": "null", "max_len": "64K", "sram_offs": "0x00201000", "ep": "0x00201101", "attr": "0x1"},
        {"id": "0xa5fd5a02", "bin": "app_xip.bin", "cert": "null", "max_len": "600K", "sram_offs": "0xffffffff", "ep": "0xffffffff", "attr": "0x2"},
        {"id": "0xa5fa5a05", "bin": "wlan_bl.bin", "cert": "null", "max_len": "4K", "sram_offs": "0xffffffff", "ep": "0xffffffff", "attr": "0x1"},
        {"id": "0xa5f95a06", "bin": "wlan_fw.bin", "cert": "null", "max_len": "48K", "sram_offs": "0xffffffff", "ep": "0xffffffff", "attr": "0x1"},
        {"id": "0xa5f85a07", "bin": "wlan_sdd_40M.bin", "cert": "null", "max_len": "1K", "sram_offs": "0xffffffff", "ep": "0xffffffff", "attr": "0x1"},
        {}
    ]
}
```

镜像配置文件内容举例如图 6-1 所示，各字段的简要说明如表 6-4 所示。详细格式说明可参考文档 SDK 的说明文档：project/image_cfg/readme.md。

表 6-4 镜像配置文件字段

字段		说明
magic		区段（section）文件魔数，位于区段头部的标识，固定为 AWIH。
version		版本号。
OTA		用于指定 OTA 参数区域在 FLASH 中的其实地址和大小。
image		用于指定当前镜像大小的最大值。
section		区段文件列表。
section	id	区段的标识，SDK 定义的 ID 可参考文件“include/image/image.h”。
section	bin	区段存储的二进制文件名称，该文件内容将被打包存储为区段数据。
section	cert	区段对应的 Secure Boot 签名证书，若没证书则填“null”。
section	max_len	区段的最大 Size，以 KB 为单位，超过会报错，不足会留出空间。
section	sram_offs	区段加载到 SRAM 中的地址偏移，以 B 为单位；0xFFFFFFFF 表示无效值。
section	ep	区段的 ENTRY POINT；0xFFFFFFFF 表示无效值，使用时会被忽略。
section	attr	区段属性。其中 0x1 表示普通可执行文件，0x2 表示 XIP 文件。

6.5. 新工程创建

6.5.1. 创建工程目录

新工程的创建可参考“project”目录下的已有工程。首先需要给工程起个有意义的名字，并以该名字作为工程目录。工程名字建议和项目相关，例如“storyteller”（故事机工程）；不推荐采用不具备标识性的名字，例如“demo1”、“audio2”等。

假设需要创建新工程“foo”（实际工程不建议采用这种命名），操作如下：

以“hello_demo”为模板，创建新工程“foo”（假设当前工作目录为 SDK 根目录）

```
$ cp -r project/demo/hello_demo project/demo/foo
```

6.5.2. 创建工程板级配置文件

由于不同项目方案的 PCB 板设计都不一样，一般情况下，每个项目方案都需要创建自己的板级配置文件。板级配置文件的创建可参考“project/common/board/”目录下的 XRADIO 开发板配置文件。

假设需要在新工程“foo”创建板级配置“bar_evb”，操作如下：

```
#以“project/common/board/xr806_dig_ver”为模板，创建板级配置“project/common/board/bar_evb”
#  （假设当前工作目录为 SDK 根目录）
$ cp -r project/common/board/xr806_dig_ver project/common/board/bar_evb
```

执行以上操作后，创建“project/common/board/bar_evb”目录，目录下包含“board_config.c”和“board_config.h”两个文件（所有的板级配置目录都必须包含这两个文件）。

由于从“project/common/board/xr806_dig_ver”复制的板级配置文件只适用于 XRADIO 开发板，所以，必须根据实际的 PCB 设计修改板级配置文件。

“board_config.c”和“board_config.h”共同定义了板级相关的配置信息，例如 pinmux 配置、CPU 频率、AHB/APB 总线频率、UART 波特率等。最关键的是要修改“board_config.c”中的 pinmux 配置，使之与 PCB 设计相符。

图 6-2 工程 pinmux 配置示例

```
static const GPIO_PinMuxParam g_pinmux_uart0[] = {
    { GPIO_PORT_B, GPIO_PIN_0,  { GPIOB_P0_F2_UART0_TX,   GPIO_DRIVING_LEVEL_1, GPIO_PULL_UP } }, /* TX */
    { GPIO_PORT_B, GPIO_PIN_1,  { GPIOB_P1_F2_UART0_RX,   GPIO_DRIVING_LEVEL_1, GPIO_PULL_UP } }, /* RX */
    //{ GPIO_PORT_B, GPIO_PIN_14,  { GPIOB_P14_F5_UART0_CTS,  GPIO_DRIVING_LEVEL_1, GPIO_PULL_UP } }, /* CTS */
    //{ GPIO_PORT_B, GPIO_PIN_15,  { GPIOB_P15_F5_UART0_RTS,  GPIO_DRIVING_LEVEL_1, GPIO_PULL_UP } }, /* RTS */
};

static const GPIO_PinMuxParam g_pinmux_uart1[] = {
    { GPIO_PORT_B, GPIO_PIN_14, { GPIOB_P14_F2_UART1_TX,  GPIO_DRIVING_LEVEL_1, GPIO_PULL_UP } }, /* TX */
    { GPIO_PORT_B, GPIO_PIN_15, { GPIOB_P15_F2_UART1_RX,  GPIO_DRIVING_LEVEL_1, GPIO_PULL_UP } }, /* RX */
};

static const GPIO_PinMuxParam g_pinmux_uart2[] = {
    { GPIO_PORT_B, GPIO_PIN_14, { GPIOB_P14_F3_UART2_TX,  GPIO_DRIVING_LEVEL_1, GPIO_PULL_UP } }, /* TX */
    { GPIO_PORT_B, GPIO_PIN_15, { GPIOB_P15_F3_UART2_RX,  GPIO_DRIVING_LEVEL_1, GPIO_PULL_UP } }, /* RX */
    { GPIO_PORT_A, GPIO_PIN_20, { GPIOA_P20_F2_UART2_CTS, GPIO_DRIVING_LEVEL_1, GPIO_PULL_DOWN } }, /* CTS */
    { GPIO_PORT_A, GPIO_PIN_19, { GPIOA_P19_F2_UART2_RTS, GPIO_DRIVING_LEVEL_1, GPIO_PULL_DOWN } }, /* RTS */
};
```

```
static HAL_Status board_get_pinmux_info(uint32_t major, uint32_t minor, uint32_t param,
                                        struct board_pinmux_info *info[])
{
    HAL_Status ret = HAL_OK;

    switch (major) {
    case HAL_DEV_MAJOR_UART:
        if (minor == UART0_ID) {
            info[0].pinmux = g_pinmux_uart0;
            info[0].count = HAL_ARRAY_SIZE(g_pinmux_uart0);
        } else if (minor == UART1_ID) {
            info[0].pinmux = g_pinmux_uart1;
            info[0].count = HAL_ARRAY_SIZE(g_pinmux_uart1);
        } else if (minor == UART2_ID) {
            info[0].pinmux = g_pinmux_uart2;
            info[0].count = HAL_ARRAY_SIZE(g_pinmux_uart2);
        } else {
            ret = HAL_INVALID;
        }
        break;
```

图 3-2 展示了“project/common/board/xr806_dig_ver/board_config.c”中 uart 相关的 pinmux 配置，主要包含两部分代码的实现：

1. pinmux 配置数组，图中包含了 uart0、uart1、uart2 三个 pinmux 配置数组。
2. “board_get_pinmux_info()”函数中对 pinmux 配置数组的获取代码。

所有的 pinmux 配置信息均需包含以上两部分的代码实现。

修改工程“board_config.c”的注意事项包括：

1. 所有的 pinmux 配置信息不能冲突，任何一个 pin 脚只能配置一种功能。例如，在 uart0 pinmux 配置数组中将 PB01 设置为 GPIOB_P1_F2_UART0_RX，就不能同时在 PWM pinmux 配置数组中将 PB01 设置为 GPIOB_P1_F4_PWM5_ECT5。
2. 板级配置信息（pinmux 配置等）必须与 PCB 设计相符，否则会导致系统启动异常或功能不可用。
3. 删除无用的 pinmux 配置信息以节省代码空间（为了示范不同外设的 pinmux 配置方法，“project/common/board/xr806_dig_ver/board_config.c”定义了较全的外设 pinmux 配置，实际使用不会这么多）。例如，PCB 只设计一个 UART0，则 UART1、UART2 相关的配置信息都应删除。
4. 创建或者删除 pinmux 配置数组的同时，务必修改“board_get_pinmux_info()”函数的相关代码，否则会导致编译错误。

6.5.3. 创建工程链接脚本

工程默认链接脚本为“project/linker_script/gcc/appos.ld”。新建工程一般需要实现本工程的链接脚本，相关说明可参考 6.2 节“LINKER_SCRIPT”变量说明和 6.3 节。

建议新工程创建本工程的链接脚本路径为：“project/[prj]/gcc/appos.ld”，具体实现可参考以下两个文件：

1. 工程默认链接脚本：“project/linker_script/gcc/appos.ld”
2. 其他工程编译过程中产生的实际链接脚本：“project/[prj]/gcc/.project.ld”

6.5.4. 创建工程镜像配置文件

工程默认镜像配置文件为“project/image_cfg/image.cfg”。新建工程一般需要实现本工程的镜像配置文件，相关说明可参考 6.2 节“IMAGE_CFG”变量说明和 3.4 节。

建议新工程创建本工程的镜像配置文件路径为：“project/[prj]/image/[chip]/image.cfg”，具体实现可参考以下两个文件：

1. 工程默认镜像配置文件：“project/image_cfg/image.cfg”
2. 其他工程创建镜像过程中产生的实际镜像配置文件：“project/[prj]/image/[chip]/.image.cfg”

6.5.5. 修改工程 defconfig

工程需要的配置选项在“project/[prj]/gcc/defconfig”中定义。具体工程需要根据项目方案需求，开启或禁用相关配置选项。例如，以下配置开启了 xplayer、XIP、OTA（OTA 需要开启 WLAN 功能）功能（某项配置默认开启，需要关闭时，通过“# CONFIG_PSRAM is not set”关闭）。

```
CONFIG_PROJECT="demo/foo"
CONFIG_BOARD="bar_evb"
CONFIG_XIP=y
# CONFIG_PSRAM is not set
CONFIG_XPLAYER=y
CONFIG_WLAN=y
```

注意

重点确认工程名字和板级配置是否正确，若不配置，会使用默认配置，默认配置与客户定义的不一致会导致非预期的结果。

6.5.6. 修改工程 Makefile

工程 Makefile 位于“project/[prj]/gcc/Makefile”，主要用于指定工程的源文件、链接脚本、镜像配置文件等，详细说明可参考 3.2 节。

一般需要修改的项包括：ROOT_PATH、PRJ_ROOT_PATH、SRCS、LINKER_SCRIPT、IMAGE_CFG。

假设“project/demo/foo”工程的文件目录结构（假设芯片选择为 XR806）如下：

```
project/demo/foo/
├──── command.c
├──── command.h
├──── gcc
│     ├──── appos.ld
│     ├──── defconfig
│     └──── Makefile
├──── image
│     └──── xr806
│           └──── image.cfg
├──── main.c
└──── prj_config.h
```

“project/demo/foo/gcc/Makefile”中需要修改的变量如下：

```
# 定义 SDK 根目录
ROOT_PATH := ../../../../..

# 定义工程根目录
PRJ_ROOT_PATH := $(ROOT_PATH)/project/$(PROJECT)

# 定义工程源文件，需要根据具体工程的源代码结构进行源代码指定
SRCS := $(basename $(foreach dir,$(DIRS),$(wildcard $(dir)/*.[csS])))

# 定义链接脚本
LINKER_SCRIPT := ./appos.ld

# 定义镜像配置文件
IMAGE_CFG := ./image.cfg
```

6.5.7. 修改工程 prj_config.h

工程“prj_config.h”定义本工程的功能选项配置值，可覆盖“project/common/prj_conf_opt.h”中的默认值。具体工程需要根据项目方案需求，重定义相关的功能选项。

所有可选的功能选项可参考“project/common/prj_conf_opt.h”中的注释描述。例如，以下代码将主线程栈设置为 2KB，MAC 地址从 EFUSE 获取，禁用看门狗超时复位功能：

```
/* main thread stack size */
#define PRJCONF_MAIN_THREAD_STACK_SIZE    (2 * 1024)

/* MAC address source */
#define PRJCONF_MAC_ADDR_SOURCE                SYSINFO_MAC_ADDR_EFUSE

/* watchdog enable/disable */
#define PRJCONF_WDG_EN                         0
```

6.5.8. 修改工程 command.c

工程“command.c”定义了本工程可用的控制台命令。具体工程可根据需求增加或删除命令，SDK 支持的所有命令在“project/common/cmd/”目录下实现，用户也可以自定义新命令。

6.5.9. 编译工程

工程编译方法可参考 4.3 节，具体编译示例可参考 4.4 节。

7. 常见问题

7.1. 烧录失败处理

可能是串口线波特率太高传输数据不稳定导致，检查是否安装了合适的串口驱动，或者更换不同型号的串口线，若仍不能解决尝试降低波特率，比如降低到 921600 或 115200。

7.2. SecureCRT 换行符处理

SecureCRT 需要做配置：“选项” -> “会话选项” -> “终端” -> “仿真” -> “模式”，在初始模式和当前模式中分别选中“新行模式”。

7.3. SecureCRT make menuconfig 显示乱码处理

- 若在 Linux 服务器编译，使用 SecureCRT 登录，make menuconfig 时显示乱码。

在 SecureCRT 中做如下配置，中文版如下：“选项” -> “会话选项” -> “终端” -> “仿真”，在终端中选择“Xterm”，并勾选“ANSI 颜色”。

英文版为：options -> terminal -> emulation -> xterm 勾选 ANSI color ，然后重新登录即可。

著作权声明

商标声明

免责声明