XR806 控制台命令方案

开发指南

版本号：1.0

发布时间：2020-12-14

版本历史

版本	日期	责任人	版本描述
1.0	2020-12-14	AWA 1680	创建文档。

目录

表格目录

1 前言

1.1 文档简介

本文档介绍了 XR806 平台上控制台命令方案的开发与使用方法。

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

2.1 背景说明

控制台命令（Console Command）方案是 XR806 SDK 中常用的测试及调试手段。通过串口命令的输入，可以让开发板实现用户需要的各种功能。开启控制台功能后，用户可通过串口输入命令字符串，控制台接收、解析命令，并调用系统接口执行对应操作，最后将处理结果的响应信息通过串口输出给用户。

例如，在 SDK 工程中完成 Console Command 功能配置并完成编译烧写后，可使用 mem 命令去获取内存地址的内容，命令操作如下所示：

```
$ mem r32 0x40040058 4          #Console 输入的命令，获取地址 0x40040000 的内容
40040058: 90000b10              #Console 响应的命令，返回地址 0x40040000 的内容
```

Console Command 方案可以便于用户进行高效的开发，本文主要介绍如何在工程里面进行控制台命令的添加和配置。

2.2 规格特性

Console Command 方案支持以下功能。

表 2-1 Console Command 方案功能特性表

规格类型	规格功能特性描述	备注
基本功能	支持字符串解析、命令匹配、命令参数提取	
扩展功能	支持命令类型扩展、命令类型删除	
其他功能	提供一套 SDK 命令库，支持常见模块的各类命令操作	

2.3 文件位置

以 SDK 包为根目录，Console Command 方案涉及到的主要文件位置如下。

表 2-2 Console Command 方案的文件位置

组件名	文件分类	文件位置
Console Command	控制台模块源码	./src/console/console.c
	控制台模块头文件	./include/console/console.h
	命令相关的源码	./src/project/common/cmd/
	命令相关的头文件	./src/project/common/cmd/cmd_defs.h; ./src/project/common/cmd/cmd_util.h ./src/project/common/cmd/cmd.h

组件名	文件分类	文件位置
	示例工程	./project/example/demo/hello_demo 等

说明

用户工程目录下的 command.c 主要用于 Console 命令管理，添加路径可参见 4.2.2 章节。

关键文件说明如下。

表 2-3 Console Command 方案的关键文件说明

文件名	文件说明
./src/console/console.c	Console 控制台机制和基本框架的实现
./src/project/common/cmd/cmd_util.c	通用的命令执行、命令解析相关代码实现
./src/project/common/cmd/cmd_defs.h	命令状态/事件响应类型定义
./src/project/common/cmd/cmd_<module>.c	模块 module 对应的命令代码实现
./src/project/common/cmd/cmd_util.h	SDK 中命令解析、执行等通用操作的头文件
./src/project/common/cmd/cmd.h	SDK 中常用模块的命令库头文件

说明

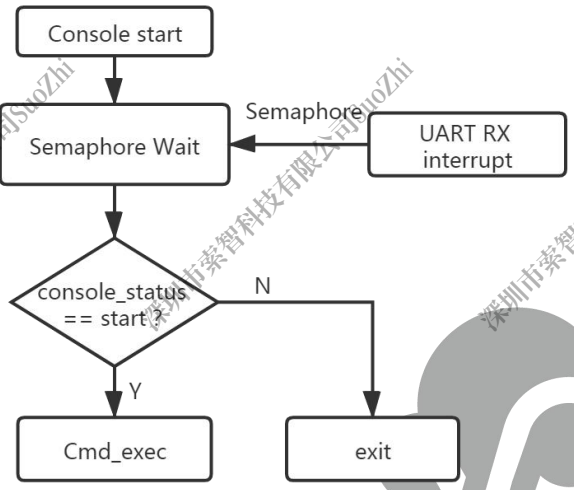
XR806 SDK 可在以下 GitHub 仓库获取：https://github.com/XradioTech/xr806_sdk.git

3 技术说明

3.1 命令解析

系统初始化控制台后，当用户通过串口输入命令，会触发 XR806 的 UART 接收中断。中断过程中会释放控制台等待的信号量并将输入数据传输给控制台，控制台线程将调用命令解析函数对数据进行解析。控制台执行解析命令流程如下图所示。

图 3-1 控制台执行流程



3.2 命令分级

在 XR806 SDK 中，控制台命令以分级的方式进行组织，命令格式如下：

```
<一级命令名> [<二级命令名> <三级命令名> .....] [命令参数]
```

每一条完整的命令均由若干级的命令名和可选的命令参数组成。不同级的命令名之间一般采用空格进行分割，命令执行时会逐级进行字符串匹配。

命令 A：“net sta config my_sta_ssid 123456789”

命令 B：“net sta enable”

命令 C：“net ap config my_ap_ssid 987654321”

命令 D：“net ap enable”

以上 4 个命令举例的说明如下：

1. “net”是一级命令名。
2. 一级命令“net”下有 2 个二级命令名：“sta”、“ap”。
3. 二级命令“net sta”下有 2 个三级命令名：“config”、“enable”。

4. 二级命令“net ap”下有 2 个三级命令名：“config”、“enable”。
5. 三级命令“net sta config”的命令参数是：“my_sta_ssid 123456789”。
6. 三级命令“net sta enable”无命令参数。
7. 三级命令“net ap config”的命令参数是：“my_ap_ssid 987654321”。
8. 三级命令“net ap enable”无命令参数。

每一级命令下均可包含若干不同的子命令，在代码中以子命令列表数组的方式实现。例如：

1. 一级命令列表为 g_main_cmds 数组，该数组中可包含 drv 子模块，sysinfo 子模块。
2. drv 子模块的命令列表为数组 g_drv_cmds（二级命令列表），sysinfo 子模块的命令列表为数组 g_sysinfo_cmds（二级命令列表）。drv 子模块中还可包含 UART 子模块和 WDG 子模块。
3. UART 子模块的命令列表为数组 g_uart_cmds（三级命令列表），WDG 子模块的命令列表为数组 g_wdg_cmds（三级命令列表）。

```
void main_cmd_exec(char *cmd) /* 一级命令执行 */
{
    cmd_main_exec(cmd, g_main_cmds, cmd_nitems(g_main_cmds));
}

static enum cmd_status cmd_drv_exec(char *cmd) /* 二级命令执行 */
{
    return cmd_exec(cmd, g_drv_cmds, cmd_nitems(g_drv_cmds));
}

enum cmd_status cmd_sysinfo_exec(char *cmd) /* 二级命令执行 */
{
    return cmd_exec(cmd, g_sysinfo_cmds, cmd_nitems(g_sysinfo_cmds));
}

enum cmd_status cmd_uart_exec(char *cmd) /* 三级命令执行 */
{
    return cmd_exec(cmd, g_uart_cmds, cmd_nitems(g_uart_cmds));
}

enum cmd_status cmd_wdg_exec(char *cmd) /* 三级命令执行 */
{
    return cmd_exec(cmd, g_wdg_cmds, cmd_nitems(g_wdg_cmds));
}

static const struct cmd_data g_main_cmds[] = { /* 一级命令列表(1) */
```

```
    { "drv",       cmd_drv_exec, CMD_DESC("driver modules")},
    { "sysinfo",cmd_sysinfo_exec, CMD_DESC("system info")},
};

static const struct cmd_data g_sysinfo_cmds[] = { /*  二级命令列表(1-1) */
    { "chip",       cmd_sysinfo_chip_exec, CMD_DESC("get chip arch version") },
    { "version", cmd_sysinfo_version_exec, CMD_DESC("get SDK version") },
};

static const struct cmd_data g_drv_cmds[] = { /*  二级命令列表(1-2) */
    { "uart",     cmd_uart_exec, CMD_DESC("uart module command sets")},
    { "wdg",       cmd_wdg_exec, CMD_DESC("watch_dog module command sets")},
};

static const struct cmd_data g_uart_cmds[] = { /*  三级命令列表(1-2-1) */
    { "tsf-start", cmd_uart_transfer_start_exec, CMD_DESC("uart transfer start")},
    { "tsf--stop",  cmd_uart_transfer_stop_exec, CMD_DESC("uart transfer stop")},
};

static const struct cmd_data g_wdg_cmds[] = { /*  三级命令列表(1-2-2) */
    { "start",          cmd_wdg_start_exec, CMD_DESC("wdg start")},
    { "stop",           cmd_wdg_stop_exec, CMD_DESC("wdg stop")},
};

enum cmd_status cmd_sysinfo_chip_exec(char *cmd)
{
    // sysinfo 模块具体功能实现
}

enum cmd_status cmd_sysinfo_version_exec(char *cmd)
{
    // sysinfo 模块具体功能实现
}

enum cmd_status cmd_uart_transfer_start_exec(char *cmd)
{
    // UART 模块具体功能实现
}

enum cmd_status cmd_uart_transfer_stop_exec(char *cmd)
{
```

```
    // UART 模块具体功能实现
}

enum cmd_status cmd_wdg_start_exec(char *cmd)
{
    // WDG 模块具体功能实现
}

enum cmd_status cmd_wdg_stop_exec(char *cmd)
{
    // WDG 模块具体功能实现
}
```

上述代码中，以向控制台输入“drv uart tsf-start 0 dma”命令为例，代码的执行流程如下：

1. “drv uart tsf-start 0 dma”命令会作为参数传入到 cmd_main_exec 函数中，执行 cmd_main_exec 的时候会匹配一级命令名。
2. 匹配到一级命令名“drv”后，执行 cmd_drv_exec 进行二级命令匹配和执行。
3. 匹配到二级命令名“uart”后，执行 cmd_uart_exec 进行三级命令匹配和执行。
4. 匹配到三级命令名“tsf-start”后，执行 cmd_uart_transfer_start_exec 完成 uart 的传输启动操作，且三级命令名空格后的子字符串“0 dma”会作为参数传入到 cmd_uart_transfer_start_exec 中。

说明

上述示例命令中的命令参数“0 dma”为可选项，依赖于用户对 cmd_uart_transfer_start_exec 函数的具体实现。

4 应用说明

4.1 应用简述

Console Command 方案已经内嵌到 XR806 SDK，应用步骤如下：

1. 进行 SDK 中 Console 功能配置，并配置 Console 控制台对应的串口信息，详细参见 4.2.1 章节。
2. 在工程中添加 Command 文件，包括 command.c 和 command.h，详细参见 4.2.2 章节。
3. 添加用户需要使用的命令，包括添加已有命令和用户自定义命令，详细参见 4.2.3 章节。

4.2 配置说明

本配置说明主要介绍用户在工程中添加 Console Command 的步骤。

4.2.1 Console 功能配置

4.2.1.1 Console功能使能

表 4-1 XR806 Console 功能使能

配置项	配置说明
系统的Console功能使能	设置说明： 此项配置用于在 SDK 中启用 Console 功能，即在 platform init 阶段进行 Console 功能的初始化。 设置位置： 工程目录下的 prj_config.h 文件，例如 hello_demo 工程中的 /project/demo/hello_demo/prj_config.h 文件。 设置方式： 在 prj_config.h 文件，添加或修改以下定义，其中 1 为进行 Console 功能初始化，0 为不进行 Console 功能初始化。 /* console enable/disable */ #define PRJCONF_CONSOLE_EN 1

4.2.1.2 Console任务栈大小配置

表 4-2 XR806 Console 任务栈大小配置

配置项	配置说明
Console 任务栈大小设置	设置说明： 在平台初始化时，由于 Console 的功能使能，系统会创建一个控制台的任务，控制台任务的栈大小会被 SDK 自动配置。默认的栈大小为 2KB，用户可根据实

配置项	配置说明
	际情况进行配置。 设置位置： 工程目录下的 prj_config.h 文件，例如 hello_demo 工程中的 /project/demo/hello_demo/prj_config.h 文件。 设置方式： 在 prj_config.h 文件，添加或修改以下定义： #define PRJCONF_CONSOLE_STACK_SIZE (2 * 1024)

4.2.1.3 Console串口使能

表 4-3 XR806 Console 串口使能

配置项	配置说明
Console 串口使能	设置说明： Console 功能依赖于串口通信，在 SDK 的项目配置中需要对串口进行使能。 设置位置： 工程目录下的 prj_config.h 文件，例如 hello_demo 工程中的 /project/demo/hello_demo/prj_config.h 文件。 设置方式： 在 prj_config.h 文件，添加或修改以下定义： #define PRJCONF_UART_EN 1

4.2.1.4 Console串口号配置

表 4-4 XR806 Console 串口号配置

配置项	配置说明
Console 串口号配置	设置说明： Console 功能依赖于串口通信，在 SDK 的项目配置中需要设置 Console 使用的串口号。Console 默认使用的串口号是 UART0，用户可根据实际情况进行配置。 设置位置： SDK/project/common/board 子目录下的 board_config.h 文件中 设置方式： 在 board_config.h 文件中，添加或修改用户需要使用的 UART ID： /* debug and console */ #define BOARD_MAIN_UART_ID UART0_ID

4.2.2 Command 文件添加

1) 向工程中添加或修改 command.c、command.h 文件，添加后，工程目录结构如下：

```
.
├──── command.c                    # 本工程的控制台命令入口和命令定义
├──── command.h
├──── gcc
│     ├──── defconfig              # 本工程的默认配置规则
│     └──── Makefile               # 本工程的 Makefile
├──── image
│     └──── xr806
├──── main.c                       # 本工程的入口
├──── prj_config.h                 # 本工程的功能选项配置
```

2) command.c 文件内容填充如下：

```
#include "common/cmd/cmd_util.h"
#include "common/cmd/cmd.h"
...
/* main commands */
static enum cmd_status cmd_main_help_exec(char *cmd);

static const struct cmd_data g_main_cmds[] = {
    { "help",     cmd_main_help_exec, CMD_DESC(CMD_HELP_DESC) },
    { "mem",  cmd_mem_exec, CMD_DESC("memory command") },
    { "thread",     cmd_thread_exec, CMD_DESC("thread information command") },
    ... /* Any addition/deletion by user */
};

static enum cmd_status cmd_main_help_exec(char *cmd)
{
    return cmd_help_exec(g_main_cmds, cmd_nitems(g_main_cmds), 8);
}

void main_cmd_exec(char *cmd)
{
    cmd_main_exec(cmd, g_main_cmds, cmd_nitems(g_main_cmds));
}
```

说明

控制台线程解析命令的入口为 main_cmd_exec 函数，因此 command.c 文件中必须要定义此函数。

4.2.3 命令修改添加

添加命令的方式有多种，可以选择添加 SDK 已提供的测试命令，也可以添加自定义命令。

4.2.3.1 命令列表的格式

SDK 中命令列表的格式定义有两种，分别为格式一和格式二。格式一对应的结构体和接口函数为：cmd_data、cmd_exec、cmd_help_exec；格式二对应的结构体和接口函数为：cmd2_data、cmd2_exec、cmd2_help_exec。两种格式的区别描述如下：

1）cmd_data 与 cmd2_data

两者均是用于定义命令列表的结构体类型（cmd2_data 中多了一个 name_len 的成员变量），主要区别在于命令名的匹配方式不同，如下：

表 4-5 cmd_data 与 cmd2_data 主要区别

结构体类型名	说明
cmd_data	以空格为间隔符，对命令名进行匹配
cmd2_data	以 name_len 长度对命令名进行匹配，命令名格式不限，由用户任意指定

两种结构体的成员说明如下：

表 4-6 cmd_data 结构体的成员说明

成员项	说明
char *name;	含义解释：此命令的名称（命令名） 使用说明：用于命令匹配
enum cmd_status (*exec)(char *);	含义解释：此命令对应的执行跳转接口 使用说明：N/A
char *desc;	含义解释：此命令的含义描述 使用说明：一般用于打印帮助信息

表 4-7 cmd2_data 结构体的成员说明

成员项	说明
char *name;	含义解释：此命令的名称（命令名）

成员项	说明
	使用说明：用于命令匹配
int name_len;	含义解释：命令名的长度 使用说明：单位：字节
enum cmd_status (*exec)(char *);	含义解释：此命令对应的执行跳转接口 使用说明：N/A
char *desc;	含义解释：此命令的含义描述 使用说明：一般用于打印帮助信息

2）cmd_exec 与 cmd2_exec

两者均是控制台方案中的命令匹配接口，功能均为：对输入的命令字符串进行解析，并通过命令名与命令列表数组中的元素进行匹配，匹配成功后，将命令名后面的字符串传入到对应的命令执行函数，并跳转到该函数执行。

两者的主要区别在于对输入的命令字符串解析策略不同（cmd_exec 接口与 cmd_data 型命令列表配套使用，cmd2_exec 接口与 cmd2_data 型命令列表配套使用），如下：

表 4-8 cmd_exec 与 cmd2_exec 主要区别

函数名	说明
cmd_exec	解析命令字符串时，内部会对输入字符串的第一个空格进行截断，空格前面的子字符串即为命令名，用于命令匹配，空格后面的内容作为参数传入下一级的跳转函数中
cmd2_exec	解析命令字符串时，直接从传入的参数中截取长度为 name_len 字节的子字符串作为命令名并进行命令匹配，匹配成功后，将命令名后面的所有字符内容全部作为参数传入下一级的跳转函数中。因此，有了 name_len 之后，命令名的格式可以是任意形式，内部截取命令名时不会以空格为标准，而是以 name_len 为标准。

3）cmd_help_exec 与 cmd2_help_exec

两者均为打印命令列表帮助信息的接口，区别在于所用的命令列表类型不同（cmd_help_exec 接口与 cmd_data 型命令列表匹配使用，cmd2_help_exec 接口与 cmd2_data 型命令列表匹配使用）。

表 4-9 cmd_help_exec 与 cmd2_help_exec 主要区别

函数名	说明
cmd_help_exec	打印输出 cmd_data 型命令列表中各命令的帮助信息
cmd2_help_exec	打印输出 cmd2_data 型命令列表中各命令的帮助信息

两种格式对应的串口输入要求如下：

格式一（cmd_data 型）：

```
command format: <command-name> <arg>...
```

格式二（cmd2_data 型）：

```
command2 format: <command-name>[<arg> ...]
```

如上述对比说明，cmd_data 和 cmd2_data 的作用主要是用于定义命令列表，两者的区别在于命令名的匹配方式不同：

对于格式一（cmd_data 型），以空格为间隔符，对命令名进行匹配；

对于格式二（cmd_data 型），以 name_len 长度对命令名进行匹配。

两种格式的命令列表写法与命令匹配过程均存在差异：

1）例如采用格式一，对 fs 命令下的 open 操作进行定义，如下：

```
static const struct cmd_data g_fs_cmds[] = {
    { "open",       cmd_fs_open_exec,       CMD_DESC("open file, fs open <file-path>, eg. fs open data/test.txt") },
};

enum cmd_status cmd_fs_exec(char *cmd)
{
    return cmd_exec(cmd, g_fs_cmds, cmd_nitems(g_fs_cmds));
}
```

命令列表的名称为 g_fs_cmds，列表成员采用 cmd_data 结构体类型，“open”表示命令名，cmd_fs_open_exec 为对应的命令执行函数，列表中不需要命令名长度信息。

进行命令匹配时，cmd_exec 默认以空格来分割。如若 cmd_fs_exec 中传入的命令为“open data/test.txt”，空格前面的“open”会用于命令匹配，空格后面的“data/test.txt”会作为参数传入到 cmd_fs_open_exec 中并跳转执行。

2）例如采用格式二，对 mem 命令下的“读”操作进行定义，如下：

```
static const struct cmd2_data g_mem_cmds[] = {
    { "r",1, cmd_mem_read_exec, CMD_DESC("r<bit_mode> <address> <length>, read data from memory, eg. mem r32 0x40040058 4") },
};

enum cmd_status cmd_mem_exec(char *cmd)
{
    return cmd2_exec(cmd, g_mem_cmds, cmd_nitems(g_mem_cmds));
}
```

命令列表的名称为 g_mem_cmds，列表成员采用 cmd2_data 结构体类型，“r”表示命令名，该命令名的长度（name_len）为 1。

进行命令匹配时，cmd2_exec 会依据命令名长度（name_len）对命令名进行截取。如若 cmd_mem_exec 中传入的命令为“r32 0x40040058 4”（“r”与“32”之间无空格隔开），由于列表中该命令名长度为 1，则“r”会作为命令名进行命令匹配，“r”后面的“32 0x40040058 4”会作为参数传入到 cmd_mem_read_exec 中并跳转执行。

说明

相比于 cmd_data 类型，cmd2_data 多了一个表示命令名称长度的成员 name_len，目的是为了满足特殊的命令处理：即命令名称为不规则格式，例如命令名由包含空格的字符串组成（如 cmd_wlan.c 文件中设置 bss 最大值命令的名称为：“bss max count ”，name_len 为 14）；例如命令名与参数之间没有空格（如 cmd_mem.c 中读内存操作的命令名称为“r”，name_len 为 1），通过 name_len 成员的加入可方便直接定位到命令参数的首地址。

实际使用中，多以 cmd_data 类型为主，串口输入时，命令名一般为一个简单名词或某个操作的缩写，后面紧跟此命令的参数，并使用空格隔开，此种命令的输入方式也更为常见，因此后文主要采用 cmd_data 类型命令格式进行介绍。

4.2.3.2 添加已有命令

XR806 SDK 中提供了常用的模块测试命令接口，用户可以将这些命令接口灵活地添加到自己的工程项目中，例如为了固件烧写方便，可以添加 upgrade 命令。在 command.c 文件中，往 g_main_cmds 数组中添加 cmd_upgrade_exec 命令，如下所示。

```
static const struct cmd_data g_main_cmds[] = {
...
    { "upgrade",cmd_upgrade_exec, CMD_DESC("upgrade command") },
...
};
```

说明

1. SDK 中已有的测试命令均包含在头文件 SDK/project/common/cmd/cmd.h 中，用户只需在 command.c 文件中引用此头文件即可。
2. 命令名和对应的跳转执行函数名可参考 SDK/project 目录下各工程子目录的 command.c 文件，以及用户需要使用模块的 cmd_<module>.c 与 cmd_<module>.h 文件。

4.2.3.3 添加自定义命令

命令的匹配数组包括 g_main_cmds（一级命令列表）和 g_<module>_cmds（各模块下的次级命令列表），无论用户扩展哪种类型的命令，只需完成以下步骤：

1. 在对应的数组中增加自定义的命令类型元素（若为次级命令类型，需要确认次级命令的执行是否最终包含在 g_main_cmds 数组中）；
2. 实现对应的跳转执行函数功能。

例如，为了测试某文件系统中文件的读写操作，可以自定义 g_fs_cmds 的命令操作集。

```
static const struct cmd_data g_fs_cmds[] = {
    { "open",       cmd_fs_open_exec,       CMD_DESC("open file, fs open <file-path>") },
    { "close",      cmd_fs_close_exec,      CMD_DESC("close file") },
    { "read",       cmd_fs_read_exec,       CMD_DESC("read file context") },
    { "write",      cmd_fs_write_exec,      CMD_DESC("write context to file, fs write <string>") },
};

enum cmd_status cmd_fs_exec(char *cmd)
{
    return cmd_exec(cmd, g_fs_cmds, cmd_nitems(g_fs_cmds));
}
```

然后将 cmd_fs_exec 添加到 g_main_cmds 数组中：

```
static const struct cmd_data g_main_cmds[] = {
...
    { "upgrade", cmd_upgrade_exec, CMD_DESC("upgrade command") },
    { "fs", cmd_fs_exec, CMD_DESC("fs command") },
...
};
```

最后依次实现 cmd_fs_open_exec、cmd_fs_close_exec、cmd_fs_read_exec、cmd_fs_write_exec 等接口即可。

4.2.3.4 删除命令

删除一级命令类型的指令，需要在一级命令列表（g_main_cmds）中删除对应的元素项；若需要删除模块下的次级命令，则只需在模块对应的次级命令列表（g_<module>_cmds）中删除该命令的元素项即可。

4.3 接口说明

XR806 Console Command 方案为用户提供一套命令操作接口，方便用户进行命令的自定义与扩展。其中用户可能会用到的关键函数接口如下表所示。

表 4-10 Console Command 方案接口简介

接口名	简要说明
命令匹配接口	包括 2 个接口函数，用于命令字符串的匹配。
命令参数提取接口	包括 1 个接口函数，用于命令字符串中参数的提取。
命令响应接口	包括 1 个接口函数，用于响应命令的执行情况。
其他接口	包括 3 个接口函数，用于主命令列表的解析执行、命令帮助信息的输出等。

4.3.1 命令匹配接口

4.3.1.1 cmd_exec

表 4-11 cmd_exec 接口函数说明

信息项	说明
原型	enum cmd_status cmd_exec(char *cmd, const struct cmd_data *cdata, int count);
功能	对输入的命令字符串进行解析，以空格为间隔符对命令名与命令列表数组中的元素进行匹配，匹配成功后，将命令名后面的字符串传入到对应的命令执行函数，并跳转到该函数执行。
参数	char *cmd 含义解释：命令字符串首地址 使用说明：本函数可能修改命令字符串的内容，所以，cmd 指向的内存空间必须是可读写的。 const struct cmd_data *cdata 含义解释：需要进行命令匹配的命令列表数组首地址 使用说明：数组元素的数据类型可参见“表 4-6 cmd_data 结构体的成员说明” int count 含义解释：命令列表数组中的元素个数 使用说明：N/A
返回值	CMD_STATUS_ACKED：无需输出命令响应 CMD_STATUS_OK：命令执行成功 CMD_STATUS_UNKNOWN_CMD：未知命令 CMD_STATUS_INVALID_ARG：无效参数 CMD_STATUS_FAIL：命令执行错误

4.3.1.2 cmd2_exec

表 4-12 cmd2_exec 接口函数说明

信息项	说明
原型	enum cmd_status cmd2_exec(char *cmd, const struct cmd2_data *cdata, int count);
功能	对输入的命令字符串进行解析，根据 struct cmd2_data::name_len 指定的长度，对命令名与命令列表数组中的元素进行匹配，匹配成功后，将命令名后面的字符串传入到对应的命令执行函数，并跳转到该函数执行。
参数	char *cmd 含义解释：命令字符串首地址 使用说明：本函数可能修改命令字符串的内容，所以，cmd 指向的内存空间必须是可读写的。 const struct cmd2_data *cdata 含义解释：需要进行命令匹配的命令列表数组首地址

信息项	说明
	使用说明：数组元素的数据类型可参见“表 4-7 cmd2_data 结构体的成员说明” int count 含义解释：命令列表数组中的元素个数 使用说明：N/A
返回值	CMD_STATUS_ACKED：无需输出命令响应 CMD_STATUS_OK：命令执行成功 CMD_STATUS_UNKNOWN_CMD：未知命令 CMD_STATUS_INVALID_ARG：无效参数 CMD_STATUS_FAIL：命令执行错误

4.3.2 命令参数提取接口

4.3.2.1 cmd_parse_argv

表 4-13 cmd_parse_argv 接口函数说明

信息项	说明
原型	int cmd_parse_argv(char *cmd, char *argv[], int size);
功能	解析并存储命令字符串中的参数到指定的指针数组中。 具体过程为：解析命令字符串 cmd 中的参数（以空格为参数间隔符），把解析到的各个参数作为子字符串，将其首地址存入数组 argv 中，输入的 size 为数组 argv 的长度，表示最大的可存储参数个数。 若 cmd 命令字符串中的参数个数大于 size，则也只在数组 argv 中存放 size 个参数。
参数	char *cmd 含义解释：命令字符串首地址 使用说明：本函数可能修改命令字符串的内容，所以，cmd 指向的内存空间必须是可读写的。 char *argv[] 含义解释：指针数组，用于存放各个参数字符串的首地址。 使用说明：N/A int size 含义解释：指针数组 argv 中可存放的元素个数 使用说明：N/A
返回值	从 cmd 命令字符串中解析到的参数个数。（若实际 cmd 命令字符串中的参数个数大于 argv 数组的长度，则返回的是数组 argv 的长度）

4.3.3 命令响应接口

4.3.3.1 cmd_write_respond

表 4-14 cmd_write_respond 接口函数说明

信息项	说明
原型	#define cmd_write_respond(status, fmt, arg...) \ cmd_write(CMD_CODE_TYEP_STATUS, (int)status, fmt, ##arg)
功能	是一个宏定义，用于向串口输出用户自定义的命令响应字符串，字符串格式为： <ACK> <status> <description> <ACK>：固定输出的命令响应标识字符串 <status>：status 对应的响应状态码 <description>：对应参数 fmt, arg...的标准格式化字符串
参数	int status 含义解释：响应状态码，用户可自定义任意数值，也可参考 SDK 中枚举类型 enum cmd_status 中的枚举成员进行赋值。参见“表 4-19 cmd_status 枚举类型成员说明”。 使用说明：N/A const char *fmt, ... 含义解释：可变参数（格式控制，输出列表），用于标准格式化输出。 使用说明：可参考 C 库 printf()函数的格式字符输出标准。
返回值	成功：返回向串口写入的字节数 失败：返回-1

4.3.4 其他接口

4.3.4.1 cmd_main_exec

表 4-15 cmd_main_exec 接口函数说明

信息项	说明
原型	enum cmd_status cmd_main_exec(char *cmd, const struct cmd_data *cdata, int count);
功能	命令执行的一级入口，内部会调用 cmd_exec 接口进行命令匹配、跳转执行，并依据命令执行的状态结果，向控制台输出标准响应，标准响应的格式可参见“4.4.1 命令的响应格式”章节。
参数	char *cmd 含义解释：命令字符串首地址 使用说明：本函数可能修改命令字符串的内容，所以，cmd 指向的内存空间必须是可读写的。 const struct cmd_data *cdata 含义解释：需要进行命令匹配的命令列表数组首地址 使用说明：一般为数组 g_main_cmds，数组元素的数据类型可参见“表 4-6

信息项	说明
	cmd_data 结构体的成员说明” int count 含义解释：命令列表数组中的元素个数 使用说明：N/A
返回值	CMD_STATUS_ACKED：无需输出命令响应 CMD_STATUS_OK：命令执行成功 CMD_STATUS_UNKNOWN_CMD：未知命令 CMD_STATUS_INVALID_ARG：无效参数 CMD_STATUS_FAIL：命令执行错误

4.3.4.2 cmd_help_exec

表 4-16 cmd_help_exec 接口函数说明

信息项	说明
原型	enum cmd_status cmd_help_exec(const struct cmd_data *cdata, int count, int align);
功能	打印输出 cmd_data 型命令列表中各命令的帮助信息
参数	const struct cmd_data *cdata 含义解释：需要输出帮助信息的命令列表数组首地址 使用说明：数组元素的数据类型可参见“表 4-6 cmd_data 结构体的成员说明” int count 含义解释：命令列表数组中的元素个数 使用说明：N/A int align 含义解释：输出帮助信息中各命令名所占的列数，主要用于信息显示对齐，SDK 中默认为 8 使用说明：N/A
返回值	CMD_STATUS_ACKED：无需输出命令响应 CMD_STATUS_OK：命令执行成功 CMD_STATUS_UNKNOWN_CMD：未知命令 CMD_STATUS_INVALID_ARG：无效参数 CMD_STATUS_FAIL：命令执行错误

4.3.4.3 cmd2_help_exec

表 4-17 cmd2_help_exec 接口函数说明

信息项	说明
原型	enum cmd_status cmd2_help_exec(const struct cmd2_data *cdata, int count, int align);
功能	打印输出 cmd2_data 型命令列表中各命令的帮助信息
参数	const struct cmd2_data *cdata 含义解释：需要输出帮助信息的命令列表数组首地址 使用说明：数组元素的数据类型可参见“表 4-7 cmd2_data 结构体的成员说明” int count 含义解释：命令列表数组中的元素个数 使用说明：N/A int align 含义解释：输出帮助信息中各命令名所占的列数，主要用于信息显示对齐，SDK 中默认为 8 使用说明：N/A
返回值	CMD_STATUS_ACKED：无需输出命令响应 CMD_STATUS_OK：命令执行成功 CMD_STATUS_UNKNOWN_CMD：未知命令 CMD_STATUS_INVALID_ARG：无效参数 CMD_STATUS_FAIL：命令执行错误

4.4 格式说明

4.4.1 命令的响应格式

Console Command 方案有以下标准响应格式定义。

表 4-18 Console Command 标准响应说明

分类	格式	说明
响应格式	<ACK> <status-code> <respond-description>或 <EVT> <event-code> <event-description>	<ACK>：状态响应标志 <status-code>：状态码，与 cmd_status 枚举成员值对应，含义描述参见表 4-19、表 4-20 所示 <respond-description>：状态描述，详见表 4-20 所示 <EVT>：事件响应标志（XR806 SDK 暂无使用） <event-code>：事件状态码（XR806 SDK 暂无使用） <event-description>：事件描述（XR806 SDK 暂无使用）

表 4-19 cmd_status 枚举类型成员说明

枚举成员项	说明
CMD_STATUS_ACKED = 100	含义解释：命令已响应 使用说明：用户若不需要控制台进行命令执行响应，则可将自定义的命令执行函数的返回值设为 CMD_STATUS_ACKED，此时在 cmd_main_exec 函数中不会进行统一的命令响应输出； 用户在自己需要的地方调用 cmd_write_respond 接口可以实现自定义格式的控制台响应输出，方便用户灵活使用，可参见“4.3.3 命令响应接口”章节。
CMD_STATUS_OK = 200	含义解释：命令执行成功 使用说明：N/A
CMD_STATUS_UNKNOWN_CMD = 400	含义解释：未知命令 使用说明：N/A
CMD_STATUS_INVALID_ARG = 401	含义解释：无效参数 使用说明：N/A
CMD_STATUS_FAIL = 402	含义解释：命令执行失败 使用说明：N/A

表 4-20 status-code 状态码含义及描述

状态码	描述与含义
100	表示命令已响应，无需输出<ACK>、<status-code>以及<respond-description>
200	表示命令执行正确，对应的<respond-description>字符串输出为“OK”
400	表示输入的命令未添加，无法执行，对应的<respond-description>字符串输出为“Unknown command”
401	表示输入命令参数无效，对应的<respond-description>字符串输出为“Invalid argument”
402	表示命令执行失败，对应的<respond-description>字符串输出为“Fail”

5 示例说明

本章节示例主要以 SDK 中 hello_demo 工程为例，分别介绍如何在工程中直接使用 SDK 已有的命令、以及在当前工程中添加用户自定的命令。

5.1 直接使用

5.1.1 示例简介

本工程示例以 hello_demo 工程为例，会使用 SDK 中现有的命令，进行 Flash 指定区域的读取、擦除、再读取等操作，并通过控制台响应各命令的执行状态与执行结果。

5.1.1.1 获取方式

hello_demo 示例有示例工程代码，位于 XR806 SDK 的/project/demo/hello_demo 目录，以下此示例工程简称为 hello 示例工程。

说明

XR806 SDK 可在以下 GitHub 仓库获取：https://github.com/XradioTech/xr806_sdk.git

---

5.1.1.2 准备工作

hello 示例工程的硬件准备有如下。

1. 评估板：运行示例工程代码。
2. 串口线：连接评估板的 UART0 插针，用于 console 控制台的输入输出。
3. PC 机：用于镜像烧录和 console 控制的输入输出。

hello 示例工程的软件准备，包括烧写工具、代码编译和烧写操作，请参见《XR806_SDK_快速入门指南》。

5.1.1.3 操作步骤

参照“4.2 配置说明”章节，在 SDK 中使用已有命令（对 Flash 的操作）的操作步骤如下：

1. 检查/确认 hello 工程目录下 prj_config.h 中串口使能、控制台使能情况。
2. 检查/确认工程目录下的 command.c、command.h 文件（hello 工程已默认添加）。
3. 在 command.c 文件的命令列表中添加 SDK 已有的 Flash 操作命令集。

```
static const struct cmd_data g_main_cmds[] = {
    { "mem", cmd_mem_exec, CMD_DESC("memory command") },
    { "heap", cmd_heap_exec, CMD_DESC("heap use information command") },
    { "thread",     cmd_thread_exec, CMD_DESC("thread information command") },
    { "upgrade",cmd_upgrade_exec, CMD_DESC("upgrade command") },
    { "reboot", cmd_reboot_exec, CMD_DESC("reboot command") },
...
    { "flash",    cmd_flash_exec, CMD_DESC("flash command") },
```

```
};
```

说明

cmd_flash_exec 跳转接口在 SDK/project/common/cmd/cmd_flash.c 文件中实现，内部通过对 Flash 操作命令集（g_flash_cmds 数组）的解析与命令匹配完成 Flash 模块的操作子命令执行函数的跳转，用户只需在 command.c 中#include "common/cmd/cmd.h"即可，其他模块也可以此类推。

完成编译烧写后，复位开发板，在控制台依次输入如下命令操作：

1）读取 Flash 物理地址为 0x100000 往后的 32 个字节数据：

```
flash read a=0x100000 s=32
```

2）擦除 Flash 物理地址为 0x100000 往后的 4Kb 区域：

```
flash erase s=4kb a=0x100000
```

3）再次读取 Flash 物理地址为 0x100000 往后的 32 个字节数据：

```
flash read a=0x100000 s=32
```

5.1.2 效果展示

在评估板中依次运行上述命令后，在控制台中会打印输出如下。

```
$flash read a=0x100000 s=32
<ACK> 200 OK
66 6c 61 73 68 20 72 65     61 64 20 61 3d 30 78 31     30 30 30 30 30 20 73 3d     33 32 0d 0a 0d 0a 0d 0a
$flash erase s=4kb a=0x100000
<ACK> 200 OK
$flash read a=0x100000 s=32
<ACK> 200 OK
ff ff ff ff ff ff ff ff     ff ff ff ff ff ff ff ff     ff ff ff ff ff ff ff ff     ff ff ff ff ff ff ff ff
```

可以看到，擦除前，Flash 指定区域中的数据为正常数据，擦除后则为全 f，同时控制台会响应“<ACK> 200 OK”表示当前命令执行成功，符合预期。

5.2 增加新的命令类型

5.2.1 示例简介

同样以 hello_demo 工程为例，用户可以有两种方式进行命令类型的增加，在 g_main_cmds 数组（位于 command.c 文件中）中增加或者在对应子模块的 g_<module>_cmds 数组（位于 cmd_<module>.c 文件中）中增加。g_main_cmds 数组为一级命令列表，可用于体现当前工程中各个模块的支持情况，g_<module>_cmds 数组则是对应的子模块命令列表，包含子模块的各个命令功能（为次级命令列表），两者的原理类似，都是通过将命令字符串进行解析，通过匹配命令名后进行跳转执行，用户可灵活选择。

假设用户需要在此工程中测试 Flash 擦/写指定大小区域（4K、64K）的操作耗时，则用户可以选择在 cmd_flash.c 文件的 g_flash_cmds 数组中添加新的命令类型。

本示例以此需求为例，向用户展示如何进行自定义操作命令，并完成测试。

5.2.1.1 获取方式

同“5.1.1.1 获取方式”章节，hello 示例工程的代码位于 XR806 SDK 的/project/demo/hello_demo 目录中。

说明

XR806 SDK 可在以下 GitHub 仓库获取：https://github.com/XradioTech/xr806_sdk.git

5.2.1.2 准备工作

hello 示例工程的硬件准备有如下。

4. 评估板：运行示例工程代码。
5. 串口线：连接评估板的 UART0 插针，用于 console 控制台的输入输出。
6. PC 机：用于镜像烧录和 console 控制的输入输出。

本示例工程的软件准备，包括烧写工具、代码编译和烧写操作，请参见《XR806_SDK_快速入门指南》。

5.2.1.3 操作步骤

参照“4.2 配置说明”章节，在 SDK 中使用已有命令（对 Flash 的操作）的操作步骤如下：

1．检查/确认 hello 工程目录下 prj_config.h 中串口使能、控制台使能情况。

2．检查/确认工程目录下的 command.c、command.h 文件（hello 工程已默认添加）。

3．在 command.c 文件的命令列表中添加 SDK 已有的 Flash 操作命令集。（前三步同“5.1.1.3 操作步骤”章节一致）

```
static const struct cmd_data g_main_cmds[] = {
    { "mem", cmd_mem_exec, CMD_DESC("memory command") },
    { "heap", cmd_heap_exec, CMD_DESC("heap use information command") },
    { "thread",    cmd_thread_exec, CMD_DESC("thread information command") },
    { "upgrade",cmd_upgrade_exec, CMD_DESC("upgrade command") },
    { "reboot", cmd_reboot_exec, CMD_DESC("reboot command") },
...
    { "flash",   cmd_flash_exec, CMD_DESC("flash command") },
};
```

4．在 cmd_flash.c 文件的 g_flash_cmds 数组中添加新的 Flash 操作命令，命令名为“ew_time”，该命令的跳转执行函数名称为“cmd_flash_ew_timetest_exec”如下所示。

```
static const struct cmd_data g_flash_cmds[] = {
    { "start",          cmd_flash_start_exec },
    { "stop",           cmd_flash_stop_exec },
```

```
    { "erase",          cmd_flash_erase_exec },
    { "write",          cmd_flash_write_exec },
    { "read",           cmd_flash_read_exec },
    { "overwrite",      cmd_flash_overwrite_exec },
#if (defined FLASH_XIP_OPT_WRITE) && (defined FLASH_XIP_OPT_ERASR)
    { "optimize",       cmd_flash_optimize_exec },
#endif
    { "set_rmod",       cmd_flash_set_rmode_exec },
    { "set_wmod",       cmd_flash_set_program_mode_exec },
    { "bench",          cmd_flash_bench_exec },
    { "press",          cmd_flash_press_exec },
    { "press_r",        cmd_flash_press_r_exec },
    { "enter_sip",      cmd_flash_enter_sip_exec },
    { "ew_time",        cmd_flash_ew_timetest_exec },
};
```

5．在 cmd_flash.c 文件中实现 cmd_flash_ew_timetest_exec 接口的功能，代码示例如下：

```
/* only test for flash erase and write time cost */
/* flash ew_time a=0x100000 s=4kb l=20 */
/* flash ew_time a=0x100000 s=64kb l=20 */
static enum cmd_status cmd_flash_ew_timetest_exec(char *cmd)
{
    uint32_t cnt;
    char size_str[8];
    int32_t size;
    uint32_t addr;
    FlashEraseMode size_type;
    uint8_t *wbuf;
    uint32_t loop;
    OS_Time_t time_start = 0, time_end = 0;
    uint32_t erase_time_all = 0, write_time_all = 0;

    /* get param */
    cnt = cmd_sscanf(cmd, "a=0x%x s=%7s l=%d", &addr, size_str, &loop);
    /* check param */
    if (cnt != 3) {
        CMD_ERR("invalid param number %d\n", cnt);
        return CMD_STATUS_INVALID_ARG;
    }

    if (cmd_strcmp(size_str, "chip") == 0) {
        size_type = FLASH_ERASE_CHIP;
        size = 0;
```

```
} else if (cmd_strcmp(size_str, "64kb") == 0) {
        size = 0x10000;
        size_type = FLASH_ERASE_64KB;
} else if (cmd_strcmp(size_str, "32kb") == 0) {
        size_type = FLASH_ERASE_32KB;
        size = 0x8000;
} else if (cmd_strcmp(size_str, "4kb") == 0) {
        size = 0x1000;
        size_type = FLASH_ERASE_4KB;
} else {
        CMD_ERR("invalid size %s\n", size_str);
        return CMD_STATUS_INVALID_ARG;
}

/* get write buf */
wbuf = cmd_malloc(size);
if (wbuf == NULL) {
        CMD_ERR("no memory\n");
        return CMD_STATUS_FAIL;
}
cmd_memset(wbuf, 0, size);

for (int i = 0; i < loop; i++) {
        /* get erase time */
        time_start = OS_GetTicks();
        if (HAL_Flash_Erase(MFLASH, size_type, addr, 1) != HAL_OK) {
                CMD_ERR("flash erase failed\n");
                return CMD_STATUS_FAIL;
        }
        time_end = OS_GetTicks();
        erase_time_all += ((time_end - time_start) / (OS_USEC_PER_MSEC / OS_TICK));

        /* get write time */
        time_start = OS_GetTicks();
        if (HAL_Flash_Write(MFLASH, addr, wbuf, size) != HAL_OK) {
                CMD_ERR("flash write failed\n");
        }
        time_end = OS_GetTicks();
        write_time_all += ((time_end - time_start) / (OS_USEC_PER_MSEC / OS_TICK));
}

CMD_SYSLOG("flash erase cost: %d ms\n", erase_time_all / loop);
CMD_SYSLOG("flash write cost: %d ms\n", write_time_all / loop);
```

```
    cmd_free(wbuf);
    return CMD_STATUS_ACKED;/* No standard response is required */
}
```

完成编译烧写后，复位开发板，在控制台依次输入如下命令操作：

1）测试擦写 Flash 4Kb 区域大小的平均耗时，测试次数为 20：

```
flash ew_time a=0x100000 s=4kb l=20
```

2）测试擦写 Flash 4Kb 区域大小的平均耗时，测试次数为 100：

```
flash ew_time a=0x100000 s=4kb l=100
```

3）测试擦写 Flash 64Kb 区域大小的平均耗时，测试次数为 20：

```
flash ew_time a=0x100000 s=64kb l=20
```

4）测试擦写 Flash 64Kb 区域大小的平均耗时，测试次数为 50：

```
flash ew_time a=0x100000 s=64kb l=50
```

5.2.2 效果展示

在评估板中依次运行上述命令后，在控制台中会打印输出如下。

```
...
$flash ew_time a=0x100000 s=4kb l=20
flash erase cost: 39 ms
flash write cost: 7 ms
$flash ew_time a=0x100000 s=4kb l=100
flash erase cost: 40 ms
flash write cost: 6 ms
$flash ew_time a=0x100000 s=64kb l=20
flash erase cost: 219 ms
flash write cost: 111 ms
$flash ew_time a=0x100000 s=64kb l=50
flash erase cost: 218 ms
flash write cost: 111 ms
```

可以看到自定义的 flash ew_time 命令执行正常，能完成 Flash 擦写时间测试功能，同时在命令功能函数中返回 CMD_STATUS_ACKED，控制台不会输出标准响应“<ACK> XXXX”等内容，均符合预期。

6 常见疑问

1. 能否在代码中直接将字符串常量作为 main_cmd_exec 等函数的参数进行命令执行？

   不可以，例如，在代码中进行如下使用是不正确的：

```
main_cmd_exec(“net sta enable”);
```

   因为控制台在进行解析字符串时，有可能会修改字符串的内容（如 cmd_parse_argv 接口源码中会对字符串的内容进行修改），上述代码中命令字符串为常量形式，常量的值是不可修改的，因此这种做法是不允许的。建议按照如下方式编码：

```
char buf[32];
strcpy(buf, “net sta enable”);
main_cmd_exec(buf);
```

附录 A： 术语表

表 A-1 术语表

C		
CMD	Command	指令
U		
UART	Universal Asynchronous Receiver/Transmitter	通用异步收发传输器

著作权声明

商标声明

免责声明