前 言

一、ZSN603 的诞生

在射频识别技术发展迅速的今天，射频识别技术在许多行业中得到了广泛的应用，如交通票务、智能家居、金融财务、医疗卫生等行业。在传统的射频识别技术中，往往需要繁琐的电路以及复杂的软件设计，而且不同类型的卡采用的是不同的协议，这样就造成了产品开发周期长、开发过程复杂、开发难度大等问题。

为了解决传统设计中的缺陷，广州致远微电子有限公司（http://www.zlgmcu.com/，后文简称 ZLG）研发设计了一种专用芯片：ZSN603。该芯片支持 ISO/IEC 14443 Type A/B 读卡器模式；集成了 Type B、Mifare UltraLight、Mifare S50/S70、SAM、CPU PLUS 卡的操作指令。用户只需要发送简单的命令，即可完成对卡片的读写。同时，其外部电路设计简单，可以快捷、有效的开发产品，缩短研发周期。

主控 MCU 与 ZSN603 之间有 I^2C、UART 两种可选的通信方式，在使用 UART 模式时，最少仅需要 2 根线即可使用 ZSN603 芯片提供的 RFID 功能。由此可见，使用该专用芯片可以极大的节省主控 MCU 的 I/O 资源。同时，因为 RFID 功能是由 ZSN603 系列芯片进行管理，这样也可以极大的减轻 MCU 的负担以及软件工程师的编程负担。

二、存在的问题

诚然，市面上已经有一些与 ZSN603 功能类似的专用芯片，但这些芯片实质很难快速应用到实际项目中。这是因为，一个好的产品，不仅仅是一系列硬件的堆叠，还需要优质软件的密切配合。而这正是市场所缺少的，使用某个芯片前往往需要花费大量的精力阅读数据手册，了解底层细节（寄存器），再针对特定的系统（自有系统、Linux、FreeRTOS……）编程，即使芯片厂商提供了一些 Demo 资料，由于可移植性的问题，往往也还是需要花费大量的时间移植、测试、验证。

为了便于用户设计与开发，ZLG 提供了相应的软件包，用户可以直接基于软件包开发应用程序，软件包与具体平台无关，用户可以方便的嵌入到自己的系统中，此外，ZLG 已经适配了 AWorks、AMetal、Linux 等常用平台，若用户在这些系统中开发应用程序，则不需要关心任何底层细节（比如 ZSN603 的通信协议），直接基于 API 编程即可。

实际开发中，要设计出优质的软件并非易事，还涉及到一些细节问题（如中断的处理等），因此，在软件包的基础上，还进一步提供了本编程指南，除了介绍各个 API 的功能和使用方法外，还详尽的介绍了一些编程中可能遇到的问题，以指导用户编程。

三、本书目的

本编程指南旨在为用户提供编程指导，书中列举了大量的程序范例，使用户可以尽可能充分的理解 ZSN603 的各种功能以及相应 API 的使用方法，快速上手，设计并开发出稳定可靠的应用程序。

四、面向对象

本书主要为使用 ZSN603 的软件工程师编写，也可作为了解 ZSN603 的阅读资料。此外，书中讲解了部分与 ZSN603 无关的跨平台通用接口，展示了一般专用芯片（模块）的软件设计方法，因而也可作为一般的软件读物，以了解一些编程方法。

目 录

第 1 章 ZSN603 简介 ……1
1.1 特点 ……1
1.2 引脚说明 ……1
1.3 通信模型 ……3
1.3.1 UART 模式 ……4
1.3.2 I2C 模式 ……4
第 2 章 ZSN603 协议简介 ……5
2.1 物理链路层 ……5
2.2 帧格式协议 ……6
第 3 章 ZSN603 通用驱动软件包 ……8
3.1 软件包的获取 ……8
3.2 软件包结构 ……8
3.3 软件包适配 ……10
3.3.1 类型适配 ……10
3.3.2 常量定义 ……11
3.3.3 工具函数定义 ……11
3.3.4 其他平台相关函数 ……12
3.4 ZSN603 设备相关函数 ……19
3.4.1 初始化函数使用 ……19
3.5 功能接口 ……22
3.5.1 设备控制类命令 ……22
3.5.2 Mifare S50/S70 卡类命令 ……23
3.5.3 ISO7816-3 类命令 ……25
3.5.4 ISO14443（PICC）卡类命令 ……26
3.5.5 PLUS CPU 卡类命令 ……28
3.6 典型应用范例 ……29
3.6.1 ZSN603 LED 控制测试 ……29
3.6.2 A 类卡激活测试 ……29
3.6.3 B 类卡激活测试 ……30
3.6.4 自动检测模式测试 ……31
3.7 多任务环境下的使用 ……32
第 4 章 在 AMetal 中使用 ZSN603 ……34
4.1 使用 ZSN603 通用软件包接口 ……34
4.1.1 实例初始化函数 ……34
4.1.2 配置 ……34
4.1.3 应用 ……36
第 5 章 在 AWorks 中使用 ZSN603 ……37
5.1 设备使能及配置 ……37
5.1.1 设备使能 ……37
5.1.2 设备配置 ……37
5.2 使用 ZSN603 通用软件包接口 ……41
第 6 章 在 Linux 中使用 ZSN603 ……43

6.1 使用 ZSN603 通用软件包接口 ······43
6.1.1 实例初始化函数 ······43
6.1.2 应用 ······45

第1章 ZSN603 简介

✍本章导读

ZSN603 是广州致远微电子有限公司开发的一款集成了卡操作指令的芯片，用户不需要进行编程，直接使用 I^2C/UART 方式按照一定的通信协议即可使用 RFID 功能完成对卡片的读写。其外部电路设计简单，可以使用该芯片快捷、高效的开发产品。

1.1 特点

- 宽电压工作范围 2.8V~3.6V；
- 符合 ISO/IEC 14443 TypeA/B、ISO7816-3 标准；
- 集成 TypeB、Mifare UltraLight、MifareS50/S70、SAM 、PLUS CPU 卡的操作命令；
- 提供 ISO14443-4 的半双工块传输协议接口，可方便支持符合 ISO14443-4A 的 CPU 卡及符合 ISO14443-4B 的 TypeB 卡片支持 ISO7816-3 接口标准；
- 支持串口方式进行指令操作；
- 支持 I^2C 接口命令操作；
- 支持外接 2 个读卡天线；
- 读卡距离可达 7cm（取决于天线设计）；
- 支持客户自行开发分体式天线板，且尺寸可任意定义；
- 工作温度符合工业级-40℃~+85℃要求。

1.2 引脚说明

ZSN603 共计 32 个引脚，引脚排列详见图 1.1，各引脚的说明如

表 1.1 所示。

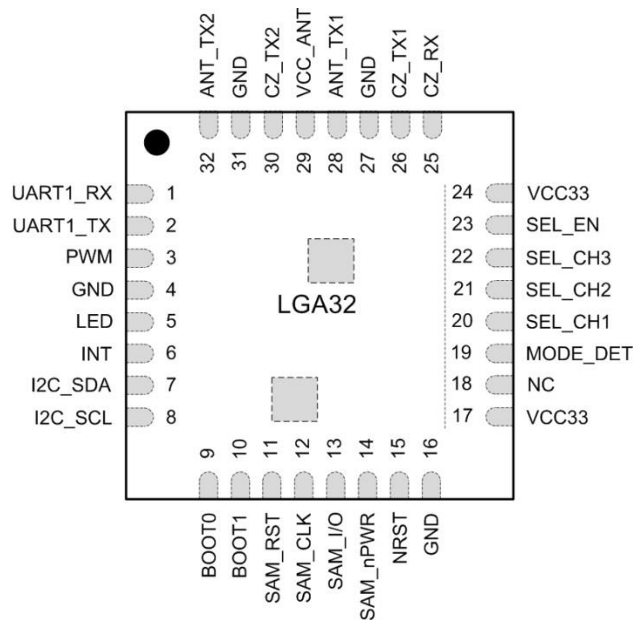

图 1.1　ZSN603 引脚分布图

表 1.1　ZSN603 芯片管脚说明

模块引脚号	主要功能	备注	引脚状态[1]	功能描述
1	UART1_RX		I	UART1 数据接收端
2	UART1_TX		O	UART1 数据发送端
3	PWM		O	PWM 输出端
4	GND		S	Ground
5	LED		O	LED 控制端
6	INT		I	外部中断输入
7	I2C_SDA		IO	I^2C 数据输入输出端，开漏结构，需上拉到供电电源
8	I2C_SCL		IO	I^2C 时钟输入输出端，开漏结构，需上拉到供电电源
9	BOOT0		-	固件升级引脚，预留测试点
10	BOOT1		-	固件升级引脚，预留测试点
11	SAM_RST		O	接触式 IC 卡控制的 RST 管脚
12	SAM_CLK		O	接触式 IC 卡控制的 CLK 管脚
13	SAM_I/O		IO	接触式 IC 卡控制的数据输入/输出管脚
14	SAM_nPWR		O	接触式 IC 卡控制的 VCC 控制管脚
15	NRST		I	芯片复位输入，低电平有效，需要连接上电复位电路
16	GND		S	Ground
17	VCC33		S	芯片供电电源输入，3.3V
18	NC		-	NC，禁止选用，需悬空处理
19	MODE_DET		I	通信模式检测
20	SEL_CH1		O	天线通道选择 1
21	SEL_CH2		O	天线通道选择 2
22	SEL_CH3		O	天线通道选择 3
23	SEL_EN		O	天线通道选择使能
24	VCC33		S	芯片供电电源输入，3.3V
25	CZ_RX		I	天线接收端
26	CZ_TX1		O	天线 1 发射端
27	GND		S	Ground
28	ANT_TX1		O	天线 1 测试端
29	VCC_ANT		S	天线供电电源，默认连接到 VCC33
30	CZ_TX2		O	天线 2 发射端
31	GND		S	Ground
32	ANT_TX2		O	天线 2 测试端

[1]　I = 输入; O = 输出; S = 电源。

1.3　通信模型

ZSN603 芯片支持两种不可同时使用的通信方式：UART 和 I^2C。外部与芯片通过这两种接口通信，必需按照规定的协议进行。本芯片以命令——响应方式工作，在系统中 ZSN603 芯片属于从属地位，不会主动发送数据（响应自动检测卡命令除外），通常主机首先发出命令，然后等待芯片响应。

1.3.1 UART 模式

在 UART 通信模式下，主控 MCU 可以通过 4 个 I/O 口完成对 ZSN603 的控制，具体的通信模型如图 1.2 所示。其中 NRST 引脚用于复位 ZSN603；INT 引脚用于通知主机事件产生（在 UART 模式下，仅用于在自动检测模式下通知用户有卡被检测到）；Tx 与 Rx 用于 UART 通信。图中 NRST 使用虚线进行表示，表明该连接是可选的。MODE_DET 引脚用于选择通信模式，在 UART 模式下 MODE_DET 引脚为高电平。

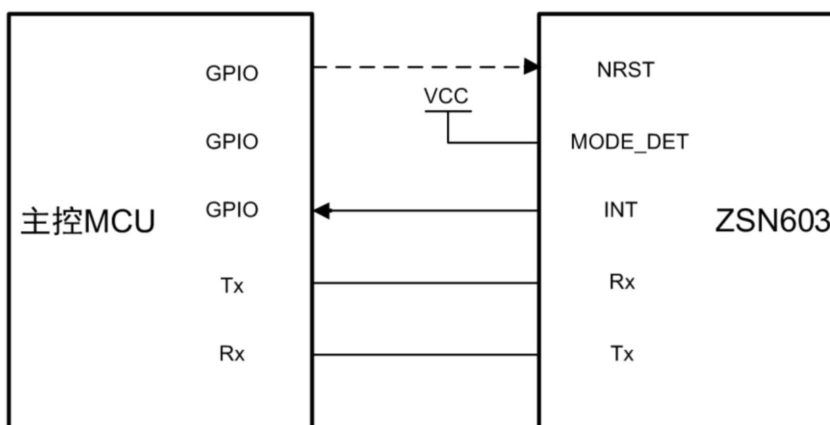

图 1.2　ZSN603 通信模型（UART 模式）

1.3.2 I2C 模式

在 I^2C 通信模式下，主控可以通过 4 个 I/O 口完成对 ZSN603 的控制，具体的通信模型如图 1.3 所示。其中 NRST 引脚用于复位 ZSN603；SDA 与 CLK 用于 I^2C 模式通信；INT 引脚为中断引脚，用于通知 I^2C 主机获取数据。图中 NRST 使用虚线进行表示，表明该连接是可选的。MODE_DET 引脚用于选择通信模式，在 I^2C 模式下 MODE_DET 引脚为低电平。

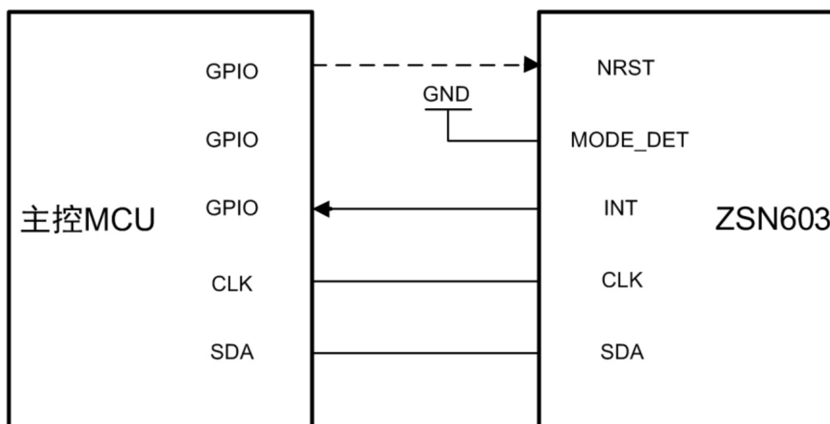

图 1.3　ZSN603 通信模型（I^2C 模式）

实际应用中，为了减少 ZSN603 对主控 MCU 的 I/O 占用，NRST 可以不由主控 MCU 控制，对于 NRST 引脚直接在外部设计一个 RC 复位电路即可。

第2章 ZSN603 协议简介

✍本章导读

ZSN603 系列芯片符合 ISO14443A、ISO14443B、ISO7816-3 标准，集成 TypeB、Mifare UltraLight、Mifare1 S50/S70、SAM 、PLUS CPU 卡的操作命令；提供 IS014443-4 半双工块传输协议接口，可方便支持符合 IS014443-4A 的 CPU 卡及符合 ISO14443-4B 的 TypeB 卡片；支持 UART、I^2C 两种通信接口；有配置参数的自动检测模式，检测到卡时可以通过中断引脚通知用户读取数据。与 ZSN603 通信，需严格遵守指定的帧格式进行，UART 与 I^2C 帧格式基本一致，只是在协议层上时略有不同而已（I^2C 模式具有数据链路层协议），本节将会重点介绍与 ZSN603 通信的相关协议。

实际上，ZLG 已经提供了通用的驱动软件包，用户大可不必深入了解相关通信协议，直接使用通用软件包提供的各个接口（第三章将详细介绍）即可，此外，对于使用 AMetal 或 AWorks 的用户，ZSN603 已经适配了标准接口，用户无需关心任何底层细节，仅需使用相关函数接口即可完成 ZSN603 的操作。本章的内容仅适合一些以学习为目的的学生或工程师，在实际应用中，为了将 ZSN603 快速应用到项目中，均不建议用户耗费大量的时间深入了解底层细节，而是直接使用接口操作 ZSN603 即可，这种情况下，建议用户跳过本章，直接阅读后续章节的内容。

2.1 物理链路层

物理链路层协议是 I^2C 通信接口必须符合的协议，若用户使用的为 UART 通信方式则可以直接忽略物理链路层，跳过该小节即可。

物理链路层是基于 I^2C 的有器件子地址模式，器件地址不固定，用户可根据指定命令进行修改，器件子地址为 2 字节。存储/数据交互空间为 542 字节，其中前 256 字节为保留使用，后 286 字节为命令帧/回应帧使用。其详细描述见表 2.1。

表 2.1 芯片存储空间分配

子地址	意义	备注
0x0000~0x00FF	保留	保留
0x0100	保留对齐使用	无特殊意义，任意读写
0x0101	主机控制/芯片状态	写入‘STATUS_EXECUTING’（0x8D） 将启动芯片执行地址 0x0104~0x021D 中的命令（命令在写入结束后才开始执行），写入其他值芯片无动作 读出是芯片当前的状态： STATUS_EXECUTING（0x8D） —— 命令还未执行 STATUS_BUSY（0x8C） —— 命令正在执行 STATUS_IDLE （0x8A） —— 芯片空闲 其他值 —— 执行结果
0x0102~0x0103	命令/回应帧长度	命令/回应帧的长度（小端模式）
0x0104~0x021D	命令/回应帧	命令/回应帧

注意：

1. 若一次性向包含‘0x0101’地址的连续存储空间写入数据，只有写入结束后，写入‘0x0101’地址的数据才起效。

2. ‘0x0101’地址的值为‘STATUS_EXECUTING’（0x8D）和‘STATUS_BUSY’（0x8C）时请勿向‘0x0101~0x021D’地址写入任何数据，因为此时‘0x0101~0x021D’地址空间是被芯片内部使用。向其写数据会造成不可估计的错误或异常情况。
3. ‘STATUS_IDLE’（0x8A）只有上电时才会自动出现。执行命令后亦不会自动恢复成‘STATUS_IDLE’状态，只会保持命令执行后的状态。若有需要，请在执行命令结束后将‘0x0101’地址的值改为‘STATUS_IDLE’。
4. 为了减少从机处理通信中断的次数，在命令执行期间，请勿频繁访问芯片的存储空间，即使是查询命令执行的状态。在命令执行期间，建议根据实际情况，2~10ms 查询一次，或者使用芯片中断输出脚（芯片命令执行完毕，中断脚会输出一个低电平，该电平持续到接收到本芯片的 SLA+W 或 SLA+R 为止）。
5. 只有命令帧格式有效的情况下，芯片才执行命令，命令帧格式无效的情况下只会产生状态，而不会产生中断。
6. 该层不适合于 UART 通信接口，UART 通信接口忽略该层。

芯片 I^2C 支持的最大速率为 400Kbps，命令执行完毕中断输出脚会产生一个低电平，该电平持续到芯片收到本芯片的 SLA+W 或 SLA+R 为止。可以通过检查该中断来判断命令是否执行完毕；也可以查询‘0x0101’地址的值来判断芯片执行的情况。命令执行完毕后可以读取‘0x0102~0x0103’的值来获取回应帧的长度，以便于确定需读取回应帧的字节数。

2.2 帧格式协议

无论是使用 UART 或者是 I^2C 方式与 ZSN603 进行通信，都遵循的是同样的帧格式。具体的帧格式如表 2.2 至表 2.5 所示

表 2.2 命令帧数据结构

地址 LocalAddr	卡槽索引 SlotIndex	包号 SMCSeq	命令类型 CmdClass	命令代码 CmdCode	信息长度 InfoLength
1 字节	1 字节	1 字节	1 字节	2 字节	2 字节
信息 Info					校验和 CheckSum
n 字节					2 字节

表 2.3 回应帧数据结构

地址 LocalAddr	卡槽索引 SlotIndex	包号 SMCSeq	命令类型 CmdClass	执行状态 Status	信息长度 InfoLength
1 字节	1 字节	1 字节	1 字节	2 字节	2 字节
信息 Info					校验和 CheckSum
n 字节					2 字节

特别注意事项：“命令码”、“信息长度”和“校验和”均以小端模式存放，即低字节在前。信息长度可以为 0，即没有信息。

表 2.4　命令帧数据结构说明

子地址	长度（字节）	说明	备注
LocalAddr	1	同 I^2C 地址模式相同，高 7 位为本机地址，低位为方向。	—
SlotIndex	1	IC 卡卡槽的索引编号（本芯片保留该字节，帧里面该字节填入 0x00 即可）。	—
SMCSeq	1	命令帧的包号。可以用来作为通信间的错误检查，从机接收到主机发来的信息，在应答信息中发出一个同样的“包号”信息，主机可以通过此信息检查是否发生的“包丢失”的错误。	可以为任意值
CmdClass	2	0x01：　设备控制类命令 0x02：　Mifare S50/S70 卡类命令(包括 USI14443-3A) 0x05：　ISO7816-3 类命令 0x06：　ISO14443（PICC）类命令 0x07：　PLUS CPU 卡类命令	不同的芯片，支持的命令类型是不一致的
CmdCode	2	命令代码	—
InfoLength	2	该帧所带信息的字节数	—
Info	InfoLength	数据信息	—
CheckSum	2	校验和，从地址字节开始到信息的最后字节的累加和取反	—

表 2.5　回应帧数据结构说明

字段	长度（字节）	说明	备注
LocalAddr	1	高 7 位同命令帧，低位为方向	—
SlotIndex	1	同命令帧	—
SMCSeq	1	同命令帧	—
CmdClass	1	同命令帧	—
Status	2	执行状态 0x00：命令执行成功 0x15：命令帧类型未找到错误 0x16：命令指令内容错误 0x17：命令参数无效 0x18：命令执行错误 others：射频卡操作失败	—
InfoLength	2	该帧所带信息的字节数	—
Info	InfoLength	数据信息	—
CheckSum	2	校验和，从地址字节开始到信息的最后字节的累加和取反	—

注：“命令码”、“信息长度”和“校验和”均以小端模式存放，即低字节在前。信息长度可以为 0，即没有信息。

由以上四个表可看出，命令帧格式与回应帧格式基本一致，只有“命令码”和“执行状态”之分。帧的最小长度为 10 字节，最长理论上可以到 65545 字节，实际上没有必要，设定 ZSN603 芯片帧的最大长度为 282 字节，信息的最大长度为 272 字节，就已完全满足处理。

ZSN603 总共有数十余条指令，具体参看《ZSN603 芯片用户指南》，本文将不再赘述。

第3章 ZSN603 通用驱动软件包

✍本章导读

为了便于用户快速将 ZSN603 应用到实际项目中，避免花费大量的时间去学习、了解 ZSN603 的内部细节。ZLG 特提供了 ZSN603 驱动软件包，使用户可以通过软件包提供的各个接口直接完成对 ZSN603 的操作，从便捷的使用 ZSN603 的 RFID 功能。本章将对如何使用该软件包进行详细的介绍。

3.1 软件包的获取

ZSN603 通用驱动软件包（为便于描述，后文将 ZSN603 通用驱动软件包简称为软件包）提供了一系列接口，实现了对 ZSN603 各种 RFID 指令的封装，使用户调用这些接口就可以快捷的使用 ZSN603 提供的各种功能，同时，这些接口与平台或硬件无关，是“通用”的，换句话说，可以在 Linux 中使用，也可以在 AWorks 中使用，还可以在 AMetal 中使用。

该软件包存于 ZSN603 资料集中，而 ZSN603 资料集从立功科技官方网站(http://www.zlgmcu.com/)下载（或者联系立功科技相关区域销售获取）。通用软件包的路径为：ZSN603 资料集/04.软件设计指南/02.通用驱动软件包。

通用驱动包中主要包含两个文件夹：driver 和 examples。driver 目录下存放了驱动相关的文件； examples 下存放了一些应用示例代码（基于驱动提供的功能接口编写的） 。 各文简要说明详见表 3.1。

表 3.1 软件包适配相关接口以及宏定义

序号	文件名	功能描述
driver	zsn603.h	各功能接口的声明，编写应用程序时查看
	zsn603.c	各功能接口的实现，用户一般无需关心具体内容
	zsn603_platform.h	与平台相关的接口声明、类型定义等
	zsn603_platform.c	平台相关接口的实现，不同平台实现不同
examples	demo_zsn603_entries.h	应用程序 demo 的入口函数声明
	demo_zsn603_led_test.c	LED 测试 Demo
	demo_zsn603_ picca_active _test.c	A 类卡激活 Demo
	demo_zsn603_ piccb_active _test.c	B 类卡激活 Demo
	demo_zsn603_auto_detect_test.c	自动检测模式 Demo

driver 目录下存放的文件是驱动的核心。其中，zsn603.h 和 zsn603.c 主要包含了 ZSN603 功能接口的声明和实现，应用程序主要基于 zsn603.h 文件中的接口编程；zsn603_platform.c 和 zsn603 _platform.h 文件中的内容与具体平台相关，不同平台实现可能不同，在 3.3 节中会详细介绍平台适配的方法，适配相关的内容即存于这两个文件中。

3.2 软件包结构

软件包的基本结构图详见图 3.1。

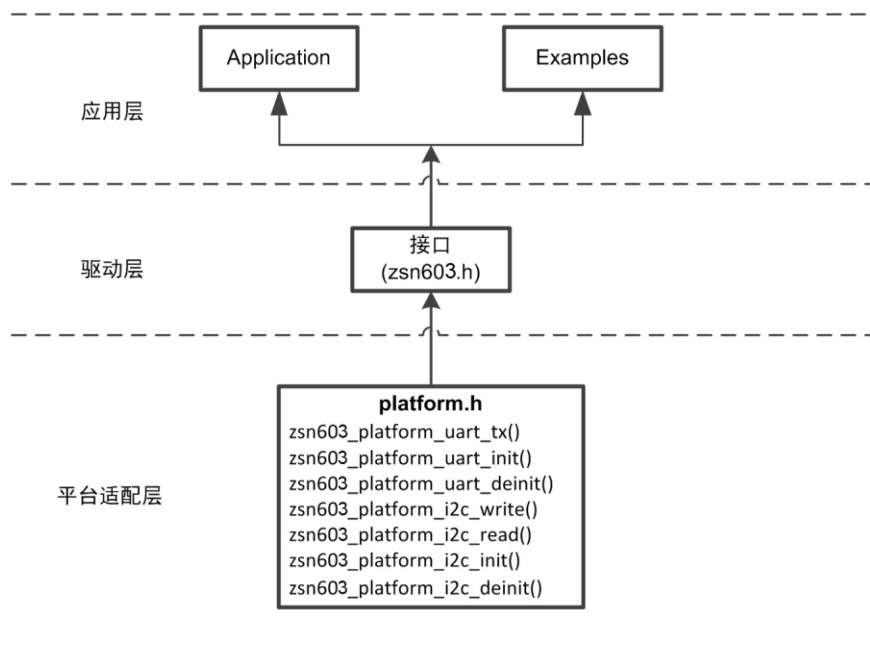

图 3.1　ZSN603 通信模型（I^2C 模式）

可以将软件包看作三层结构：应用层、驱动层和平台适配层。

- 应用层

应用层主要包括两部分：用户应用和 demo 程序。它们都基于 zsn603.h 文件中的功能接口编程（这些接口在 3.5 节中详细介绍）。后文会提到，在使用这些接口时，需要传递一个 handle（句柄）作为参数的值，基于 handle 编写的应用程序，可以很容易实现跨平台复用。但在不同平台中，获取 handle 的方式可能不尽相同。

需要注意的是，AMetal、AWorks、Linux、Windows 等平台已经完成了适配，因此，这些平台均已提供了获取 handle 的方法。 若用户在这些平台中使用 ZSN603， 则可以查看对应章节的内容，了解获取 handle 的具体方法（通常， 仅需调用一个函数即可） ， 获取到 handle 后，直接使用 3.5 节中介绍的接口操作 ZSN603 即可。基于此，若用户仅在这些平台中使用 ZSN603，则可以跳过 3.3 和 3.4 节的内容（这两节内容主要介绍了在用户特有平台中的适配方法，以及获取 handle 的方法，对平台适配和 handle 获取进行了原理性的介绍），直接从 3.5 节开始阅读。

- 驱动层

驱动层包含 zsn603.h 和 zsn603.c 文件， 它们实现了 ZSN603 核心功能的封装，其中部分与平台相关的功能，可能需要调用平台适配层的接口。

- 平台适配层

平台适配层位于最底层，其包含 zsn603_platform.h 和 zsn603_platform.c 文件，这部分代码的接口固定（需要被驱动层调用），但具体实现与平台相关。

ZSN603 通用驱动包的核心是完成了驱动层以及平台适配层的接口定义，在使用 ZSN603 通用驱动包之前，必须根据实际平台完成平台的适配工作。

虽然完成平台适配工作和基于 ZSN603 的应用编程可能是相同的人（ZSN603 通用驱动包的用户），但两者之间的侧重点不同：平台适配工作主要完成平台相关的适配，位于最底层，代码与平台息息相关；而应用编程位于最上层，代码与平台无关。因此，为了便于区分，在后文的叙述中，将完成平台适配工作的用户称之为“平台适配者”，而基于接口编写应用

程序的一般用户称之为“用户”。

3.3 软件包适配

通用软件包的存在，使得用户完全不需要关心 ZSN603 的内部细节，降低了用户使用 ZSN603 的难度，使用户可以更快的将 ZSN603 应用到实际项目中，主控 MCU 与 ZSN603 之间只需要通过 I^2C/UART 以及一些 GPIO 连接。由于在不同的软件环境（Linux、AWorks、AMetal、uc/OS、rt-thread 或用户使用的其它平台）中，I^2C 总线传输、UART 通信和 GPIO 的控制方法不尽相同，因此，在使用软件包提供的各个功能接口之前，必须根据具体平台完成对软件包的适配，实现相应的 I^2C 数据传输、UART 通信和相关功能接口。

适配工作对于部分 MCU 或平台可能较为繁琐（特别是需要了解 I^2C 的通信方法），实际上，本节讲解的适配方法主要是针对需要适配自有平台的用户而言。ZLG 已经对常用平台（AWorks、AMetal 和 Linux 等）进行了适配。若用户选用这些平台进行应用程序的开发，则可以跳过本小节，直接使用下一节中介绍的各个函数接口来操作 ZSN603，非常便捷。需要适配的内容可以通过 zsn603 _platform.h 文件获知，主要包含了 4 个部分：类型适配、常量定义、工具函数适配、其它平台相关函数。

3.3.1 类型适配

在 zsn603 _platform.h 文件中，主要使用 typedef 定义了以下几个类型：

- uint8_t ：8 位无符号整数类型
- uint16_t ：16 位无符号整数类型
- uint32_t ：32 位无符号整数类型
- zsn603 _platform_uart_devinfo_t ： 平台信息类型
- zsn603 _platform_uart_t ： 平台设备类型
- zsn603 _platform_i2c_devinfo_t ： 平台信息类型
- zsn603 _platform_i2c_t ： 平台设备类型

它们的默认定义如下：

```
typedef    unsigned char     uint8_t;
typedef    unsigned short    uint16_t;
typedef    unsigned int      uint32_t;
typedef    void    *    zsn603_platform_uart_devinfo_t;
typedef    void    *    zsn603_platform_uart_t;
typedef    void    *    zsn603_platform_i2c_devinfo_t;
typedef    void    *    zsn603_platform_i2c_t;
```

对于前 3 个数据类型，用户可以根据实际平台进行调整。若用户所处平台已经存在这些类型的定义，则可以删除 typedef 语句，直接包含相应头文件即可。

平台信息类型（zsn603_platform_uart_devinfo_t、zsn603_platform_i2c_devinfo_t）和平台设备类型（zsn603 _platform_uart_t、zsn603 _platform_i2c_t）可能用户暂时还不太理解，这些类型默认为 void *，在实际中，用户应该根据适配需要来完成这些类型的定义。对这两个类型的具体定义没有任何强制的要求，完全由适配者根据平台自行确定。简言之，在平台信息类型中，应该包含在该平台中使用 ZSN603 时，平台期望用户提供什么信息（比如引脚信息、中断号信息、I^2C 总线信息等等）；而平台设备类型中包含了上层用户无需关心，仅

用作适配者在适配 ZSN603 驱动时，需要保存的一些与 ZSN603 相关的平台内部数据（例如，当前 I^2C 的工作状态等等）。这两个类型的具体含义和用法在后文介绍平台初始化函数时，还会进一步解释。

3.3.2 常量定义

在 ZSN603 驱动中，使用了 NULL 表示空指针，因此，平台必须确保 NULL 被有效定义，在 zsn603_platform.h 文件中已经提供了默认的定义：

```
#ifndef NULL
#define NULL ((void *)0)
#endif
```

绝大部分情况下，NULL 都采用这种方式定义，因此，若无特殊情况，均无需对该部分内容进行修改。

3.3.3 工具函数定义

ZSN603 在软件包中存在一个超时机制，需要一个类似于二进制信号量的功能，不同的平台可能有不同的实现方法，因此需要平台根据实际情况实现一个二进制信号量的功能。该功能实现相关定义在 zsn603_platform.h 文件中，具体见表 3.2。

表 3.2 二进制信号量宏定义

序号	函数原型	功能简介
1	#define ZSN603_SEMB_DECL(semb)	信号量定义
2	#define ZSN603_SEMB_INIT(semb)	信号量初始化
3	#define ZSN603_SEMB_TAKE(semb, timeout)	信号量获取
4	#define ZSN603_SEMB_GIVE(semb)	信号量释放
5	#define ZSN603_SEMB_DEINIT(semb)	信号量删除

该接口用于软件包内部实现超时机制。例如，在 UART 发送命令帧之后，假设 ZSN603 未收到命令帧，若不使用超时，则软件包会死等 UART 回应帧；若使用超时机制，则软件包会在等待规定时间内接收数据，若在规定时间内无指定数据（回应帧）则返回失败。

信号量的相关宏定义保存在 zsn603_platform.h 文件内，主要包括：

```
#define ZSN603_SEMB_DECL(semb)              USER_SEMB_DECL(semb)
#define ZSN603_SEMB_INIT(semb)              USER_SEMB_ INIT (semb)
#define ZSN603_SEMB_TAKE(semb, timeout)     USER_SEMB_ TAKE (semb, timeout)
#define ZSN603_SEMB_GIVE(semb)              USER_SEMB_ GIVE (semb)
#define ZSN603_SEMB_DEINIT(semb)            USER_SEMB_ DEINT (semb)
```

- **ZSN603_SEMB_ DECL (semb)**

  定义一个二进制信号量，在编译时完成信号量所需的内存分配。

- **ZSN603_SEMB_INIT(semb)**

  信号量初始化。

- **ZSN603_SEMB_TAKE(semb， timeout)**

信号量获取，其意义为在规定超时时间内等待信号量释放，若在超时时间内获取到信号量，返回 ZSN603_SEMB_TAKE_SUCCESS（0）即可，若在超时时间内未收到信号量返回 ZSN603_SEMB_TAKE_TIMEOUT（1）即可。

- **ZSN603_SEMB_GIVE(semb)**

  信号量释放，用于释放正在等待的信号量。

- **ZSN603_SEMB_DEINIT(semb)**

  信号量的删除，当不再使用某信号量时，删除该信号量所占用内存，若用户使用的平台不支持该宏则不对该宏进行适配即可。

以上五个宏定义需根据用户使用平台不同进行适配，若用户在有操作系统的平台直接可用对应的二进制信号量操作函数对其进行适配即可。对于裸机，可使用简单的标志对其进行适配，也可实现相应的功能，范例程序详见程序清单 3.1。

程序清单 3.1　ZSN603 信号量相关宏定义适配范例

```
1    #define ZSN603_SEMB_DECL(semb)                volatile   int   semb
2    #define ZSN603_SEMB_INIT(semb)                semb   =   0
3    #define ZSN603_SEMB_TAKE(semb, timeout)       semb_wait(&semb, timeout)
4    #define ZSN603_SEMB_GIVE(semb)                semb   =   1
5    #define ZSN603_SEMB_DEINIT(semb)              /*  裸机不支持释放功能  */
6    unsigned char   semb_wait(int   *p_semb,   uint32_t   timeout)
7    {
8        unsigned int   time0 = user_systick_get();          /*  用户系统时钟获取  */
9        unsigned int   time1 = 0;
10       do{
11           time1 = user_systick_get() ;
12       }while(*p_semb == 0 && (time1 - time0) < timeout);   /*等待标志释放或者超时  */
13       if(*p_semb == 1){
14           *p_semb = 0;
15           return   ZSN603_SEMB_TAKE_SUCCESS;       /*  若等待信号量成功，则返回 success */
16       }else{
17           return   ZSN603_SEMB_TAKE_TIMEOUT;
18       }
19   }
```

注意，上述代码只是以裸机为例编写的一些示意性的代码，并没有实现其特定的功能，具体实现用户可以根据平台的不同，对其进行具体的适配。

3.3.4　其他平台相关函数

这类函数主要是平台相关的，且与 ZSN603 有密切的关联，是整个适配过程的关键。主要有三个函数，它们均在 zsn603_platform.h 文件中声明，接口原型详见表 3.3。

表 3.3　软件包适配相关接口

序号	函数原型	功能简介
1	int zsn603_platform_uart_tx(void *p_arg, uint8_t *p_data, uint32_t nbytes);	串口发送

续上表

序号	函数原型	功能简介
2	uint8_t zsn603_platform_uart_init(zsn603_platform_uart_t *p_dev, const zsn603_platform_uart_devinfo_t *p_info, zsn603_uart_funcs_t *p_funcs, void *p_arg);	串口平台相关初始化
3	uint8_t zsn603_platform_uart _deinit(zsn603_platform_uart_t *p_dev);	串口解初始化
4	int zsn603_platform_i2c_write(void *p_arg, uint8_t slv_addr, uint16_t sub_addr, uint8_t *p_data,uint32_t nbytes);	I^2C 总线写函数
5	int zsn603_platform_i2c_ read (void *p_arg, uint8_t slv_addr, uint16_t sub_addr, uint8_t *p_data,uint32_t nbytes);	I^2C 总线读函数
6	uint8_t zsn603_platform _i2c _init(zsn603_platform_i2c_t *p_dev, const zsn603_platform_i2c_devinfo_t *p_info, void(*int_pin_callback_function) (void *), void *p_arg);	I^2C 平台相关初始化
7	uint8_t zsn603_platform _i2c _deinit(zsn603_platform_i2c_t *p_dev);	I^2C 解初始化

由于用户所选择的通信方式不同则用户需要适配的函数接口也不同，若用户选择使用 UART 方式进行通信，则需额外配置序号 1~3 与 UART 通信相关函数适配即可；若用户选择使用 I^2C 方式进行通信，则需要进行序号 4~7 与 I^2C 通信相关接口适配即可。

1． 平台初始化函数

表 3.3 中序号 2、6 为平台相关初始化函数，用于用户平台完成一些包括引脚初始化、通讯方式初始化以及用户自定义相关内容初始化工作，例如 ZSN603 可能会使用到 INT 引脚、NRST 引脚、通信相关引脚等，则这些资源都必须在平台初始化函数中完成。对于该初始化函数，UART 与 I^2C 模式适配工作基本一致，只有细微的差别，用户若只会使用一种方式进行通信，则只需要关心所选通信方式的适配即可。具体平台初始化函数原型见程序清单 3.2 。

程序清单 3.2 平台初始化函数接口

```
1    /* UART 平台初始化函数 */
2    uint8_t  zsn603_platform _uart _init(zsn603_platform _uart _t              *p_dev,
3                                          const zsn603_platform _uart _devinfo_t  *p_info,
4                                          zsn603_uart_funcs_t                     *p_funcs,
5                                          void                                    *p_arg);
6
7    /* I2C 平台初始化函数 */
8    uint8_t  zsn603_platform _i2c _init(zsn603_platform _i2c _t                *p_dev,
9                                         const zsn603_platform _i2c _devinfo_t   *p_info,
10                                        void(*int_pin_callback_function)        (void *),
11                                        void                                    *p_arg);
```

该函数的返回值为 uint8_t 类型，主要用于返回初始化的结果，0 表示成功，非 0 的任

意值表示失败（平台初始化失败时，ZSN603_xx_init()也会随之失败）。参数总共有四个，其中 p_dev 分别为 zsn603_platform_uart_t 和 zsn603_platform_i2c_t 类型；为平台相关设备结构体；p_info 分别为 zsn603_platform_uart_devinfo_t 和 zsn603_platform_i2c_devinfo_t 类型，为平台相关设备信息结构体，p_funcs 为保存 UART 模式所需要设置中断回调函数的结构体，int_pin_callback_function 为 I^2C 模式需要设置的引脚中断回调函数；p_arg 为中断回调函数的参数，下面，分别对各个参数以及函数返回值作详细介绍。

- p_dev

p_dev 是指向平台设备的指针，在两个初始化函数中分别为 zsn603_platform _uart _t 和 zsn603_platform_i2c _t 类型，该类型中可以保存一些用户自定义平台以及通信相关的参数，如当前设备工作状态、当前设备从机地址等。用户需根据实际需求进行适配。具体定义例程见程序清单 3.3。

程序清单 3.3　平台相关设备类型定义

```
1    /* UART 相关平台设备结构体类型定义 */
2    typedef struct   user_platform_uart {
3          uint8_t   status;
4          uint8_t   local_address;
5          // 用户可自由添加或者删除
6    }user_platform_uart _t;
7    /* UART 相关平台设备结构体别名定义 */
8    typedef   user_platform_uart_t   zsn603_platform_uart _t;
9
10   /* I2C 相关平台设备结构体类型定义 */
11   typedef struct   user_platform_i2c {
12         uint8_t   status;
13         uint8_t   local_address;
14          // 用户可自由添加或者删除
15   }user_platform_i2c _t;
16   /* I2C 相关平台设备结构体别名定义 */
17   typedef   user_platform _i2c _t   zsn603_platform_ i2c_t;
```

在程序清单 3.3 中定义了 user_ platform_uart_ t 和 user_platform _i2c _t 两个结构体类型，这两个结构体中的成员默认为空，用户可自由添加。在定义完成之后第 8、17 行通过 typedef 将这两个类型的别名定义为 zsn603_platform_uart_t 与 zsn603_platform_i2c_t。这样无论用户自定义结构体的成员有多少，类型名为什么，软件包都可以获取该类型。

当然，用户也可以不使用该结构体，直接将结构体定义注释，并将第 8、17 行用以下代码进行简单替换，即将平台相关设备类型定义为 void * 类型即可。

```
typedef   void *      zsn603_ platform_uart_ t;            /*   UART 相关平台设备结构体别名定义 */
typedef   void *      zsn603_ platform_i2c_ t;             /*   I2C 相关平台设备结构体别名定义 */
```

ZSN603 驱动会负责定义 zsn603_platform_uart_t 与 zsn603_platform_i2c_t 类型的变量（开辟响应的内存空间），在调用平台相关函数时，将其地址作为各个函数的第一个参数传递，因此，适配者无需担心 p_dev 会指向一个无效的地址空间（比如 NULL），可以放心的使用该指针，访问其中的成员。

- p_info

p_info 在不同的通信方式平台相关初始化函数中分别为 zsn603_platform_uart_devinfo_t 和 zsn603_ platform_i2c_devinfo_t 类型，该类型与用户平台相关设备结构体一样，也属于用户自定义类型，用户需根据需求进行适配。该结构体中可以保存一些用户使用引脚、初始从机地址等不可修改的信息。具体定义例程见程序清单 3.4 。

程序清单 3.4　平台相关设备信息结构体类型定义

```
1    /* UART 相关平台设备信息结构体类型定义*/
2    typedef struct    user_platform_uart _info {
3        uint8_t     local_address;                        /*  初始 local_address */
4        uint32_t    baud_rate;                            /*  初始波特率  */
5        uint8_t     uart_com;                             /*  硬件端口号  */
6        // 用户可自由添加或者删除
7    } usert_platform_uar _info_t;
8    /* UART 相关平台设备信息结构体别名定义  */
9    typedef    user_platform _uart _info_t    zsn603_platform_uart_devinfo_t;
10
11   /*I2C 相关平台设备信息结构体类型定义*/
12   typedef struct    user_platform_i2c _info {
13       uint8_t     local_address;                        /*   初始 local_address */
14       uint32_t    i2c_speed;                            /*   I2C  速率  */
15       uint8_t     i2c_index;                            /*   硬件总线号  */
16       // 用户可自由添加或者删除
17   } user_platform_ i2c_info_t;
18   /* I2C 相关平台设备信息结构体别名定义  */
19   typedef    user_platform_i2c _info_t    zsn603_platform_uart_devinfo_t;
```

该平台相关设备结构体定义类似，区别在于平台设备中的成员在程序运行过程中可能会改变，但是设备信息内的成员原则上为 const 类型，用户不可在程序运行中不可对该结构体中成员进行修改。与设备结构体一样，用户也可以在适配中不使用该结构体，直接将该结构体的定义注释，并将程序清单 3.4 中第 9、19 行用以下代码进行替换即可。

```
typedef    void *      zsn603_platform_uart _devinfo_t;    /*   UART 相关平台设备信息结构体别名定义  */
typedef    void *      zsn603_platform_i2c _devinfo_t;     /*   I2C 相关平台设备信息结构体别名定义  */
```

- p_funcs 与 int_pin_callback_function

int_pin_callback_function 为 I^2C 平台初始化函数中第三个参数，该参数为一个函数指针类型，该参数为引脚中断函数，用于用户平台通知软件包有引脚中断产生；p_funs 为 UART 平台初始化函数中第三个参数，其类型为 zsn603_uart_funcs_t，该类型为软件包定义，用户不可进行修改，其类型定义如下：

```
typedef struct zsn603_uart_funcs {
    void (*zsn603_uart_rx)        (void *p_arg,    char    data);        /*  串口接收数据回调函数*/
    void (*zsn603_int_pin_cb)    (void *p_arg);                         /*  引脚中断引脚回调函数*/
}zsn603_uart_funcs_t;
```

由以上定义可得知，zsn603_uart_funcs_t 类型包含了 zsn603_uart_rx 与 zsn603_int_pin_cb 两个函数指针类型成员。zsn603_uart_rx 为 UART 接收回调函数，其作用为获取用户平台串口所接收到的数据；zsn603_int_pin_cb 与 int_pin_callback_function 一样，都是用于用户平台

通知软件包有引脚中断事件产生。

- p_arg

p_arg 在 UART 与 I²C 模式下，其作用都是作为传递进来的函数指针的第一个参数。

p_funcs 中 zsn603_int_pin_cb 与 int_pin_callback_function 一样，都是为引脚中断回调函数，用户在适配平台初始化过程中需设置该函数为中断引脚的中断回调函数，或直接在引脚中断函数中调用，该函数只有一个参数，且参数类型为 void *类型，该参数的值为平台初始化函数中传递的 p_arg，用户直接将平台初始化函数中的 p_arg 进行传递，无需做任何修改；

p_funcs 中 zsn603_uart_rx 成员为串口接收中断回调函数，用户在适配过程中需设置中断引脚为下降沿触发，并且将该函数设置为串口接收中断回调函数，或者直接在串口接收中断函数中调用该函数即可。该函数有两个参数，其中第一个参数类型为 void *类型，直接将平台初始化函数传递的 p_arg 参数进行传递即可，第二个参数为用户平台串口获取的数据。

对应裸机实现初始化函数的适配，一般采用直接在对应中断函数中调用回调函数，具体参考程序清单 3.5 与程序清单 3.6。

程序清单 3.5　ZSN603 串口平台相关初始化函数适配

```
1    void (*__g_zsn603_uart_rx)    (void    *p_arg,    char data);
2    void (*__g_zsn603_int_pin_cb)(void    *p_arg);
3    void *__gp_arg = NULL;
4
5    /*    用户串口接收中断函数  */
6    void user_uart_rx_handler()
7    {
8            char data;
9            data = user_uart_data_get();                          /*  用户获取串口接收数据  */
10           __g_zsn603_uart_rx(__gp_arg,    data);                /*  调用软件包的回调函数  */
11   }
12
13   /*    用户引脚中断函数  */
14   void user_gpio_int_handler()
15   {
16       __g_zsn603_int_pin_cb(__gp_arg);
17   }
18
19   /*    设备平台相关初始化函数    */
20   uint8_t    zsn603_platform_uart_init(zsn603_platform_uart_t             *p_dev,
21                                        const zsn603_platform_uart_devinfo_t    *p_info,
22                                        zsn603_uart_funcs_t                 *p_funcs,
23                                        void                                *p_arg)
24   {
25           __gp_arg = p_arg;
26           __g__zsn603_uart_rx = p_funcs->zsn603_uart_rx;
27           __g_zsn603_int_pin_cb = p_funcs-> zsn603_int_pin_cb;
28           user_gpio_cfg(nrst_pin, GPIO_OUTPUT);                 /*  配置 GPIO 为输出模式  */
```

```
29        user_gpio_set(nrst_pin, 0);                          /* GPIO 输出低电平*/
30        user_delay_ms(10);                                   /* 延时 1ms */
31        user_gpio_set(nrst_pin, 1);                          /* GPIO 输出高电平*/
32        user_delay_ms(50);                                   /* 延时 5ms*/
33        /* 其他用户自定义操作 */
34        return 0;
35    }
```

程序清单 3.6 ZSN603 I2C 平台相关初始化函数适配

```
1     void (*__g_zsn603_int_pin_cb)(void *p_arg);
2     void *__gp_arg = NULL;
3
4     /*  用户引脚中断函数 */
5     void user_gpio_int_handler()
6     {
7        __g_zsn603_int_pin_cb(__gp_arg);
8     }
9
10    /*  设备平台相关初始化函数  */
11    uint8_t   zsn603_platform_i2c_init(zsn603_platform_i2c_t              *p_dev,
12                                const zsn603_platform_i2c_devinfo_t   *p_info,
13                                void(*int_pin_callback_function)       (void *),
14                                void                                   *p_arg)
15    {
16        __gp_arg = p_arg;
17        __g_zsn603_int_pin_cb =  int_pin_callback_function;
36        user_gpio_cfg(nrst_pin, GPIO_OUTPUT);                /* 配置 GPIO 为输出模式 */
37        user_gpio_set(nrst_pin, 0);                          /* GPIO 输出低电平*/
38        user_delay_ms(10);                                   /* 延时 1ms */
39        user_gpio_set(nrst_pin, 1);                          /* GPIO 输出高电平*/
40        user_delay_ms(50);                                   /* 延时 5ms*/
41        /* 其他用户自定义操作 */
18        return 0;
19    }
```

2. 平台相关解初始化适配

在上一小节中介绍了平台初始化函数的作用及其适配方法，在平台初始化函数中，打开了一些资源，比如 I^2C 总线、UART、INT 引脚中断等。当 ZSN603 不在被使用时，这些资源应该被释放或者关闭，例如可以福安比 I^2C 总线控制器、关闭引脚中断等。与平台相关初始化函数相对应，表 3.3 中序号 3、7 为平台相关解初始化函数，这些操作都应当在解初始化中完成，在平台初始化函数中打开、申请了什么资源，就应在平台解初始化函数中进行相应的关闭与释放。

平台解初始化函数会在后文介绍的 zsn603_xx_deinit()函数中被调用。平台解初始化函

数的原型为：:

```
uint8_t    zsn603_platform_uart_deinit(zsn603_platform_uart_t    *p_dev);
uint8_t    zsn603_platform_i2c_deinit(zsn603_platform_i2c_t      *p_dev);
```

该函数的返回值为 int 类型，主要用于返回解初始化的结果，0 表示成功，非 0 表示失败（平台解初始化失败时，zsn603_xx_deinit()也会随之失败）。该参数仅 1 个 p_dev 参数，其指定了需要解初始化的平台设备。

在对平台初始化函数参数 p_dev 介绍时提到类型 zsn603_xx_platform_t 是用户根据自身平台定义的结构体，用于保存一些需要的平台自身需要使用到的变量。基于此，在平台解初始化函数中，就需要根据 p_dev 中保存的数据进行相关资源的释放操作。

3. UART 通信相关适配

若用户使用 UART 与 ZSN603 进行通信，则需要对表 3.3 中序号 1 函数进行适配。该函数为串口传输函数，需要实现的功能为使用串口发送指定缓冲区内指定长度数据。

串口传输函数原型为：

```
int zsn603_platform_uart_tx(zsn603_platform_uart_t *p_dev, uint8_t *p_data, uint32_t nbytes);
```

其中 p_dev 为用户平台相关设备，p_data 为指定缓冲区的地址，nbytes 为需要发送的字节数。函数返回值为 int 类型，若该函数执行成功，则返回 0 即可，若执行失败，用户可以返回任意非 0 值即可。

4. I²C 通信相关适配

若用户使用 I²C 与 ZSN603 进行通信，则需要对表 3.3 中序号 4、5 函数（I²C 读写函数）进行适配。该函数实现比较简单，读取函数与写入函数的结构基本是一致的。其功能为在指定的从机地址的指定器件子地址中写入/读取指定字节数据。其函数原型为：

```
int    zsn603_platform_i2c_write(zsn603_platform_i2c_t     *p_dev,
                                 uint8_t                    slv_addr,
                                 uint16_t                   sub_addr,
                                 uint8_t                   *p_data,
                                 int32_t                    nbytes);
int    zsn603_platform_i2c_read(zsn603_platform_i2c_t      *p_dev,
                                 uint8_t                    slv_addr,
                                 uint16_t                   sub_addr,
                                 uint8_t                   *p_data,
                                 int32_t                    nbytes);
```

该函数的返回值为 int 类型，主要用于返回解初始化的结果，0 表示成功，非 0 表示失败。函数有 4 个参数：p_plfm、sub_addr、p_buf、nbytes。各个参数的简要介绍如表 3.4。

表 3.4 I²C 传输函数参数

名称	含义	详细描述
p_arg	平台相关参数	平台自身结构体对象，用于保存一些必要的状态数据
slv_addr	I²C 从机地址	I²C 设备从机地址

续上表

名称	含义	详细描述
sub_addr	寄存器地址	本次读/写的目标寄存器地址，数据应从该地址指定的寄存器开始写
p_data	数据缓存区	写入寄存器的 nbytes 数据从 p_buf 指向的缓存中获取。
nbytes	数据量（字节）	表示本次读/写的数据量

除了第一个参数外，其它参数都描述了 I2C 写操作相关的数据信息：slv_addr 表示了设备的从机地址；sub_addr 表示了地址信息；p_buf 表示了数据的存储位置；nbyte 表示的写入数据的字节数。具体平台在实现 I^2C 数据传输时，最重要的职责就是根据这些参数信息完成 I^2C 数据读写。

3.4 ZSN603 设备相关函数

3.4.1 初始化函数使用

初始化函数的作用是完成 ZSN603 芯片的初始化，在使用 ZSN603 前，用户必须先完成所选通信模式的平台相关函数的适配，然后直接调用对应的初始化函数完成对 ZSN603 的初始化。下面，从接口原型开始，详细介绍该函数的调用方法。

初始化函数的原型为：

```
/* ZSN603 设备（UART 通信模式）初始化 */
zsn603_handle_t    zsn603_uart_init(zsn603_uart_dev_t                  *p_dev,
                                    const zsn603_uart_devinfo_t        *p_devinfo)
/* ZSN603 设备（I2C 通信模式）初始化 */
zsn603_handle_t    zsn603_i2c_init(zsn603_i2c_dev_t                    *p_dev,
                                    const zsn603_i2c_devinfo_t         *p_devinfo)
```

由以上函数原型定义可知，无论用户选择何种通信模式，其函数原型基本一致。为便于描述，将初始化函数的介绍分为两大部分：参数和返回值。

1. 参数详解

由初始化函数的原型可知，总共有三个参数：p_dev、local_address 和 p_trans_arg。其中，p_dev 用于为设备运行分配必要的内存空间；local_address 为设备的从机地址；p_trans_arg 为用户适配传输函数的自定义参数。下面分别对各个参数作详细介绍。

◆ p_dev

p_dev 的主要作用是为设备运行分配必要的内存空间（用于软件包内部保存一些与 ZSN603 相关的状态数据等），p_dev 指针指向的即是一段内存空间的首地址。内存大小为 zsn603_dev_t 类型数据占用的大小，基于此，可以先使用 zsn603_dev_t 类型定义一个变量，即：

```
    zsn603_dev_t        dev;
```

则 dev 的地址（即&dev）即可作为 p_dev 的实参传递。

由于这里仅需要分配一段内存空间，因此，zsn603_dev_t 类型的具体定义并不需要关心，例如，该类型具体包含哪些数据成员等。

上面仅仅是从 p_dev 参数的设置形式上对其作了直观的描述，实际上，可以从“面向对象编程”的角度对其作更深入的理解。在这里，zsn603_dev_t 类型为 ZSN603 设备类型，

每个具体的 ZSN603 硬件设备都可以看作是一个对象，即该类型的具体实例。显然，在使用一个对象前，必须完成对象的定义，或者说实例化，对象需要占用一定的内存空间，在 C 中，具体的表现形式为使用某个类型定义相应的变量，即

```
zsn603_dev_t        dev;
```

若有多个 ZSN603 对象，例如，系统中使用了两片 ZSN603，则可以使用该类型定义多个对象，即：

```
zsn603_dev_t        dev0;
zsn603_dev_t        dev1;
```

当存在多个 ZSN603 对象时，每个 ZSN603 对象都需要使用初始化函数进行初始化。初始化哪个对象，就将其地址作为初始化函数的 p_dev 实参传递。

◆ p_devinfo

p_devinfo 为 ZSN603 设备信息，为 zsn603_uart_devinfo_t 或 const zsn603_i2c_devinfo_t 结构体类型，其类型定义如下：

```
/* ZSN603 设备结构体(UART 模式)  */
typedef struct zsn603_uart_devinfo {
    uint8_t                          local_addr;          // 初始设备地址
    zsn603_platform_uart_devinfo_t   platform_info;       // 平台相关设备信息
}zsn603_uart_devinfo_t;
/* ZSN603 设备信息结构体(I2C 模式)  */
typedef struct zsn603_i2c_devinfo {
    uint8_t                          local_addr;          // 设备地址
    zsn603_platform_i2c_devinfo_t    platform_info;       // 平台相关设备信息
}zsn603_i2c_devinfo_t;
```

其中 local_addr 为 ZSN603 的初始设备地址，其高 7 位表示的是 7 位 I^2C 从机地址，最后一位固定为 0，故其值只可能为 0xb2、0xb4 等最后一位为 0 的值，不可能为 0xb3、0xb5 等最后一位为 1 的值。芯片出厂是 local_addr 值默认为 0xB2（即 7 位从机地址为 0x59），用户可以通过修改从机地址命令来对 local_addr 进行修改，接收到修改成功回应帧或者 ZSN603 设备重新上电后，下次命令传输将以新的 local_addr 为准，且 local_addr 掉电不丢失；platform_info 为用户平台相关设备信息结构体，在上节已经详细介绍，在此就不在赘述。

2. 返回值

初始化函数的返回值为 ZSN603 实例句柄，其类型为：zsn603_handle_t，该类型的具体定义无需关心，用户唯一需要知道的是，软件包提供的各个功能接口中，均使用了一个 handle 参数，用以指定操作的 ZSN603 设备。用户在调用各个功能接口时，handle 参数设置的值即为初始化函数返回的 ZSN603 实例句柄。获取 ZSN603 实例句柄的语句为：

```
zsn603_handle_t handle   =   zsn603_init(&__g_dev ,   &__g_dev_info);
```

其中，__g_dev，local_address 等参数值的定义实际是在驱动适配时完成的（例如，local_address 赋值是在驱动适配时完成的），对于上层应用来讲，其仅需获取到一个实例句柄，并不需要关心驱动适配的具体细节。基于此，为了避免用户对驱动适配作过多的了解，在适配软件包时，可以直接对外提供一个实例初始化函数，用以完成一个 ZSN603 硬件设备的初始化，进而获取到相应的 handle，范例程序详见程序清单 3.7。

程序清单 3.7　ZSN603 实例初始化函数的实现范例（1）

```
1    static    zsn603_i2c_dev_t        __g_i2c_dev;                    //  定义 I2C 方式通信 ZSN603 设备
2    static const zsn603_i2c_devinfo_t   __g_i2c_info = {
3          0xb2,
4          NULL
5    };
6    zsn603_handle_t      zsn603_i2c_inst_init (void)
7    {
8          return zsn603_i2c_init(&__g_i2c_dev ,   &__g_i2c_info);
9    }
10
11   static    zsn603_uart_dev_t        __g_uart_dev;                  //  定义 UART 方式通信 ZSN603 设备
12   static const zsn603_uart_devinfo_t   __g_uart_info = {
13         0xb2,
14         NULL
15   };
16   zsn603_handle_t      zsn603_uart_inst_init (void)
17   {
18         return zsn603_uart_init(&__g_uart_dev ,   &__g_uart_info);
19   }
```

基于此，用户可以直接调用无参数的实例初始化函数，根据用户所需要的通信方式选择对应的初始化函数，完成 handle 的获取，即：

```
zsn603_handle_t      handle = zsn603_uart_inst_init ();                  // 若通信方式为 UART
zsn603 _handle_t     handle = zsn603_i2c_inst_init ();                   // 若通信方式为 I²C
```

在实际应用中，若使用了两片 ZSN603，则可以提供两个实例初始化函数，以此获取两个 handle，不同的 handle 即代表了不同的 ZSN603 设备，范例程序详见程序清单 3.8　。

程序清单 3.8　ZSN603 实例初始化函数的实现范例（2）

```
1    static    zsn603_uart_dev_t        __g_uart_dev_0;                // 定义 ZSN603 设备 0（UART 方式）
2    static const zsn603_uart_devinfo_t   __g_uart_info_0 = {
3          0xb2,                                             //local address  为  0xb2
4          NULL
5    };
6    zsn603_handle_t      zsn603_uart_0_inst_init (void)
7    {
8          return   zsn603_uart_init(&__g_uart_dev_0 , & __g_uart_info_0 );
9    }
10   static const zsn603_uart_devinfo_t   __g_uart_info_1 = {
```

```
11         0xb4,                                                    //local address 为 0xb4
12         NULL
13    };
14    static    zsn603_uart_dev_t       __g_uart_dev_1;               // 定义 ZSN603 设备 1（UART 方式）
15    zsn603_handle_t     zsn603_uart_1_inst_init (void)
16    {
17         return   zsn603_init(&__g_uart_dev_1 ,   &__g_uart_info_1);
18    }
```

上面的实现范例以使用两个 UART 方式通信的 ZSN603 设备为例，用户可以调用相应的实力初始化函数获取与 ZSN603 对应的 handle，即：

```
zsn603_handle_t   handle0   =   zsn603_uart_0_inst_init ();        // 定义 ZSN603 设备 0（UART 方式）
zsn603_handle_t   handle1   =   zsn603_uart_1_inst_init ();        // 定义 ZSN603 设备 1（UART 方式）
```

注意，上述代码均为示意性代码，并没有实现任何特定的功能，具体实现完全由驱动适配决定的，一般来讲，无论如何适配，在适配完成后，都应提供类似于实例初始化函数的接口，以便用户快速获取到 ZSN603 实例句柄，进而使用 ZSN603 提供的各种功能。

3.5 功能接口

ZSN603 芯片的应用命令共分为设备控制类命令、Mifare S50/S70 卡类命令、ISO7816-3 类命令、IS014443(PICC)卡类命令、PLUS CPU 卡类命令五种命令。本节将从这五类命令介绍 ZSN603 的接口。

3.5.1 设备控制类命令

设备控制类命令包含设备信息读取、IC 卡接口操作、密钥操作、天线操作、EEPROM 操作等，具体接口原型见表 3.5。

表 3.5 设备控制类命令接口

函数原型	功能简介
uint8_t zsn603_get_device_info(zsn603_handle_t handle, uint32_t buffer_len, uint8_t *p_rx_data, uint32_t *p_rx_data_count);	获取设备信息 将会返回固件版本号
uint8_t zsn603_config_icc_interface(zsn603_handle_t handle);	配置 IC 卡接口
uint8_t zsn603_close_icc_interface(zsn603_handle_t handle);	关闭 IC 卡接口
uint8_t zsn603_set_ios_type(zsn603_handle_t handle, uint8_t isotype);	设置 IC 卡协议
uint8_t zsn603_load_icc_key(zsn603_handle_t handle, uint8_t key_type, uint8_t key_block, uint8_t *p_key, uint32_t key_length);	装载 IC 卡密钥
uint8_t zsn603_set_icc_reg(zsn603_handle_t handle, uint8_t reg_addr, uint8_t reg_val);	设置 IC 卡接口的寄存器值

续上表

函数原型	功能简介
uint8_t zsn603_get_icc_reg(zsn603_handle_t handle, uint8_t reg_addr, uint8_t *p_val);	获取 IC 卡接口的寄存器值
uint8_t zsn603_set_baud_rate(zsn603_handle_t handle, uint8_t baudrate_flag);	设置串口波特率
uint8_t zsn603_set_ant_mode(zsn603_handle_t handle, uint8_t antmode_flag);	设置天线驱动模式
uint8_t zsn603_set_ant_channel(zsn603_handle_t handle, uint8_t ant_channel);	切换天线通道
uint8_t zsn603_set_slv_addr(zsn603_handle_t handle, uint8_t slv_addr);	设置设备从机地址
uint8_t zsn603_control_led(zsn603_handle_t handle, uint8_t control_led);	LED 灯控制
uint8_t zsn603_control_buzzer(zsn603_handle_t handle, uint8_t control_byte);	蜂鸣器控制
uint8_t zsn603_read_eeprom(zsn603_handle_t handle, uint8_t eeprom_addr, uint8_t nbytes, uint32_t buffer_len, uint8_t *p_buf);	读 EEPROM
uint8_t zsn603_write_eeprom(zsn603_handle_t handle, uint8_t eeprom_addr, uint8_t nbytes, uint8_t *p_buf);	写 EEPROM

3.5.2 Mifare S50/S70 卡类命令

Mifare S50/S70 卡类命令主要包含 Mifare 卡类的操作，具体接口原型见表 3.6。

表 3.6　Mifare S50/S70 卡类命令接口

函数原型	功能简介
uint8_t zsn603_mifare_request(zsn603_handle_t handle, uint8_t req_mode, uint32_t *p_atq);	Mifare 卡的请求操作
uint8_t zsn603_mifare_anticoll(zsn603_handle_t handle, uint8_t anticoll_level, uint8_t *p_know_uid, uint8_t nbit_cnt, uint8_t *p_uid, uint32_t *p_uid_cnt);	Mifare 卡的防碰撞操作

续上表

函数原型	功能简介
uint8_t zsn603_mifare_select(zsn603_handle_t handle, uint8_t anticoll_level, uint8_t *p_uid, uint8_t *p_sak);	Mifare 卡的选择操作
uint8_t zsn603_mifare_halt(zsn603_handle_t handle);	Mifare 卡挂起
uint8_t zsn603_eeprom_auth(zsn603_handle_t handle, uint8_t key_type, uint8_t *p_uid, uint8_t key_sec, uint8_t nblock);	该命令用芯片内部已存入的密钥与卡的密钥进行验证
uint8_t zsn603_key_auth(zsn603_handle_t handle, uint8_t key_type, uint8_t *p_uid, uint8_t *p_key, uint8_t key_len, uint8_t nblock);	使用输入密钥与卡的密钥进行验证
uint8_t zsn603_mifare_read(zsn603_handle_t handle, uint8_t nblock, uint32_t buffer_len, uint8_t *p_buf);	Mifare 卡进行读操作
uint8_t zsn603_mifare_write(zsn603_handle_t handle, uint8_t nblock, uint8_t *p_buf);	Mifare 卡进行写操作
uint8_t zsn603_ultralight_write(zsn603_handle_t handle, uint8_t nblock, uint8_t *p_buf);	UltraLight 卡进行写操作
uint8_t zsn603_mifare_value(zsn603_handle_t handle, uint8_t mode, uint8_t nblock, uint8_t ntransblk, uint32_t value);	Mifare 卡的值块进行加减操作并写入指定块
uint8_t zsn603_mifare_cmd_trans(zsn603_handle_t handle, uint8_t *p_tx_buf, uint8_t tx_nbytes, uint8_t *p_rx_buf, uint32_t *p_rx_nbytes);	读写器与卡片的数据交互
uint8_t zsn603_mifare_card_active(zsn603_handle_t handle, uint8_t req_mode, uint8_t *p_atq, uint8_t *p_saq, uint8_t *p_len, uint8_t *p_uid);	用于激活卡片,是请求、防碰撞和选择三条命令的组合。

续上表

函数原型	功能简介
uint8_t zsn603_card_reset(zsn603_handle_t handle, uint8_t time_ms);	卡片复位
uint8_t zsn603_auto_detect(zsn603_handle_t handle, zsn603_auto_detect_ctrl_t *p_ctrl);	卡片的自动检测
uint8_t zsn603_get_auto_detect(zsn603_handle_t handle, uint8_t ctrl_mode, zsn603_auto_detect_data_t *p_data);	读取自动检测数据
uint8_t zsn603_mifare_set_value(zsn603_handle_t handle, uint8_t block, int data);	设置值块的值
uint8_t zsn603_mifare_get_value(zsn603_handle_t handle, uint8_t block, int *p_value);	获取值块的值
uint8_t zsn603_mifare_exchange_block(zsn603_handle_t handle, uint8_t *p_data_buf, uint8_t len, uint8_t wtxm_crc, uint8_t fwi, uint8_t *p_rx_buf, uint32_t *p_len);	用于芯片向卡片发送任意长度组合的数据串

3.5.3 ISO7816-3 类命令

ISO7816-3 类命令主要包含接触式 IC 卡类的操作，具体接口原型见表 3.7。

表 3.7 ISO7816-3 类命令接口

函数原型	功能简介
uint8_t zsn603_cicc_deactivation(zsn603_handle_t handle);	关闭 IC 卡的电源
uint8_t zsn603_mifare_cmd_trans(zsn603_handle_t handle, uint8_t *p_tx_buf, uint8_t tx_nbytes, uint32_t buffer_len, uint8_t *p_rx_buf, uint32_t *p_rx_nbytes);	该命令根据接触式 IC 卡的复位信息，自动选择 T = 0 或 T = 1 传输协议
uint8_t zsn603_cicc_cold_reset(zsn603_handle_t handle, uint32_t buffer_len, uint8_t *p_rx_buf, uint32_t *p_rx_len);	接触式卡冷复位
uint8_t zsn603_cicc_warm_reset(zsn603_handle_t handle, uint32_t buffer_len, uint8_t *p_rx_buf, uint32_t *p_rx_len);	接触式卡热复位

续上表

函数原型	功能简介
uint8_t zsn603_cicc_tp0(zsn603_handle_t handle, uint8_t *p_tx_buf, uint32_t tx_bufsize, uint8_t *p_rx_buf, uint32_t *p_rx_len);	T = 0 传输协议
uint8_t zsn603_cicc_tp1(zsn603_handle_t handle, uint8_t *p_tx_buf, uint32_t tx_bufsize, uint8_t *p_rx_buf, uint32_t *p_rx_len);	T=1 传输协议

3.5.4 ISO14443（PICC）卡类命令

ISO14443（PICC）卡类命令包含支持 ISO14443 类协议的卡的操作，接口原型见表 3.8。

表 3.8 ISO14443（PICC）卡类命令接口

函数原型	功能简介
uint8_t zsn603_picca_request(zsn603_handle_t handle, uint8_t req_mode, uint32_t *p_atq);	A 型卡的请求操作
uint8_t zsn603_picca_anticoll(zsn603_handle_t handle, uint8_t anticoll_level, uint8_t *p_know_uid, uint8_t nbit_cnt, uint8_t *p_uid, uint32_t *p_uid_cnt);	A 型卡的防碰撞操作
uint8_t zsn603_picca_select(zsn603_handle_t handle, uint8_t anticoll_level, uint8_t *p_uid, uint8_t uid_cnt, uint8_t *p_sak);	A 型卡的选择
uint8_t zsn603_picca_reset(zsn603_handle_t handle, uint8_t time_ms);	A 卡复位
uint8_t zsn603_picca_halt(zsn603_handle_t handle);	A 型卡的挂起
uint8_t zsn603_picca_deselect(zsn603_handle_t handle);	将卡片置为挂起(HALT)状态
uint8_t zsn603_picca_rats(zsn603_handle_t handle, uint8_t cid, uint32_t buffer_len, uint8_t *p_ats_buf, uint32_t *p_rx_len);	RATS(request for answer to select)
uint8_t zsn603_picca_pps(zsn603_handle_t handle, uint8_t dsi_dri);	PPS

续上表

函数原型	功能简介
uint8_t zsn603_picca_active(zsn603_handle_t handle, uint8_t req_mode, uint16_t *p_atq, uint8_t *p_saq, uint8_t *p_len, uint8_t *p_uid)	A 卡激活
uint8_t zsn603_picca_tpcl(zsn603_handle_t handle, uint8_t *p_cos_buf, uint8_t cos_bufsize, uint32_t buffer_len, uint8_t *p_res_buf, uint32_t *p_rx_len);	T=CL 半双工分组传输协议
uint8_t zsn603_picca_exchange_block(zsn603_handle_t handle, uint8_t *p_data_buf, uint8_t len, uint8_t wtxm_crc, uint8_t fwi, uint32_t buffer_len, uint8_t *p_rx_buf, uint32_t *p_rx_len);	读写器与卡片的数据交互
uint8_t zsn603_piccb_active(zsn603_handle_t handle, uint8_t req_mode, uint8_t *p_info);	B 卡激活
uint8_t zsn603_piccb_reset(zsn603_handle_t handle, uint8_t time_ms);	B 卡复位
uint8_t zsn603_piccb_request(zsn603_handle_t handle, uint8_t req_mode, uint8_t slot_time, uint32_t buffer_len, uint8_t *p_uid);	B 卡请求
uint8_t zsn603_piccb_attrib(zsn603_handle_t handle, uint8_t *p_pupi, uint8_t cid, uint8_t protype);	B 型卡修改传输属性(卡选择)
uint8_t zsn603_piccb_halt(zsn603_handle_t handle, uint8_t *p_pupi);	B 卡挂起
uint8_t zsn603_piccb_getid(zsn603_handle_t handle, uint8_t req_mode, uint8_t *p_uid);	读取二代身份证 ID

3.5.5 PLUS CPU 卡类命令

PLUS CPU 卡类命令主要包含支持 PLUS CPU 卡的操作，具体接口原型见表 3.9。

表 3.9 ISO14443（PICC）卡类命令接口

函数原型	功能简介
uint8_t zsn603_plus_cpu_write_perso(zsn603_handle_t handle, uint16_t addr, uint8_t *p_data);	SL0 更新个人化数据
uint8_t zsn603_plus_cpu_commit_perso(zsn603_handle_t handle);	提交个人化数据
uint8_t zsn603_plus_cpu_first_auth(zsn603_handle_t handle, uint16_t addr, uint8_t *p_data);	首次密钥验证（密钥来自用户输入）
uint8_t zsn603_plus_cpu_first_auth_e2(zsn603_handle_t handle, uint16_t addr, uint8_t key_block);	首次密钥验证（密钥来自内存输入）
uint8_t zsn603_plus_cpu_follow_auth(zsn603_handle_t handle, uint16_t addr, uint8_t *p_data);	跟随密钥验证，密钥来自函数参数
uint8_t zsn603_plus_cpu_follow_auth_e2(zsn603_handle_t handle, uint16_t addr, uint8_t key_block);	跟随密钥验证，验证的密钥来自芯片内部 EEPROM
uint8_t zsn603_plus_cpu_sl3_reset_auth(zsn603_handle_t handle);	PLUS CPU 卡复位验证
uint8_t zsn603_plus_cpu_sl3_read(zsn603_handle_t handle, uint8_t read_mode, uint16_t start_addr, uint8_t block_num, uint8_t *p_rx_data, uint32_t *p_rx_lenght);	读取 SL3 的数据块
uint8_t zsn603_plus_cpu_sl3_write(zsn603_handle_t handle, uint8_t write_mode, uint16_t start_addr, uint8_t block_num, uint8_t *p_tx_data, uint8_t tx_lenght);	写 SL3 的数据块
uint8_t zsn603_plus_cpu_sl3_value_opr(zsn603_handle_t handle, uint8_t write_mode, uint16_t src_addr, uint16_t dst_addr, int data);	SL3 PLUS CPU 卡值操作

3.6 典型应用范例

为了使用户加深对各个接口的理解，下面针对一些典型应用编写了相应的应用程序范例，这些应用程序的入口函数原型均为：

```
void demo_zsn603_xxx_entry(zsn603_handle_t handle);
```

其中，函数名中的“xxx”与具体应用的功能相关，例如 A 卡读 ID 测试，其函数名可能为：demo_zsn603_picca_read_id_test_entry()。所有入口函数均有一个 zsn603_handle_t 类型的 handle 参数，该参数通过 ZSN603 的初始化函数得到，在不同平台中，获取的方式可能不同，在后续章节中，会分别介绍在 AMetal、AWorks、Linux 中使用 ZSN603 的方法，其中会提到 handle 的获取方法。只要获取到 handle，即可将其传入到应用入口函数中，以运行相应的应用程序。

3.6.1 ZSN603 LED 控制测试

为了判断 ZSN603 上电复位后，芯片是否工作正常，可以做一个简单的 LED 灯测试根据执行结果进行判断，范例程序详见程序清单 3.9。

程序清单 3.9 设备类 LED 控制范例程序

```
1    void demo_zsn603_led_test_entry (zsn603_handle_t handle)
2    {
3        unsigned char ret = 0;
4        ret = zsn603_control_led (handle,   ZSN603_CONTROL_LED_ON);
5        if(ret == 0){
6            printf("led on !\r\n");
7        }else{
8            printf(" led control beacuse error 0x%02x", ret);
9        }
10       ret = zsn603_control_led (handle,   ZSN603_CONTROL_LED_OFF);
11       if(ret == 0){
12           printf("led off !\r\n");
13       }else{
14           printf("led control beacuse error 0x%02x", ret);
15       }
16   }
```

该应用程序的入口函数为：demo_zsn603_led_test_entry()，在程序清单 3.6 中，程序调用 zsn603_control_led()函数进行 LED 灯控制，若执行成功，LED 将会点亮后熄灭，用户可根据此测试效果判断 ZSN603 是否正常工作。

3.6.2 A 类卡激活测试

为了判断 ZSN603 的 RFID 功能是否正常工作，实现一个简单的激活操作：在 ZSN603 上电复位后，将任意一张 A 卡放置于天线感应区，根据激活是否成功判断 ZSN603 工作是否正常。范例程序详见程序清单 3.10。

程序清单 3.10　A 类卡激活范例程序

```
1    void demo_zsn603_picca_active_test_entry (zsn603_handle_t handle)
2    {
3        unsigned char   atq[2] = {0};
4        unsigned char   saq = 0;
5        unsigned char   len = 0;
6        unsigned char   uid[10] = {0};
7        unsigned char   ret = 0;
8        //A 卡请求接口函数   请求模式为  0x26   IDLE
9        ret = zsn603_picca_active(handle, 0x26, (uint16_t   *)atq, &saq, &len, uid);
10       if(ret == 0){
11           unsigned char i = 0;
12           printf("ATQ   is : %02x   %02x\r\n", atq[0], atq[1]);
13           printf("SAQ   is : %02x \r\n", saq);
14           printf("UID is : ");
15           for(i = 0; i < len ; i ++ ){
16               printf("%02x ",   uid[i]);
17           }
18           printf("\r\n");
19       }else{
20           printf("active fail beacuse error 0x%02x", ret);
21       }
22   }
```

该应用程序为 A 类卡激活函数，其入口函数为：demo_zsn603_picca_active_test_entry()，在程序清单 3.10 中，程序调用 zsn603_picca_active()函数，函数返回值保存在 ret 中，若 ret 等于 0，则表示命令主机发送成功、从机并成功执行命令帧，该例程 打印了激活操作返回的相关信息（包括 ATQ、SAQ 以及 UID）；若 ret 等于任意非 0 值则会打印对应错误号，具体错误标识在 zsn603.h 头文件中被定义。

3.6.3 B 类卡激活测试

ZSN603 提供的 RFID 功能支持 B 卡操作，程序清单 3.11 实现了 B 类卡激活的范例。

程序清单 3.11　B 类卡激活范例程序

```
1    void demo_zsn603_piccb_active_test_entry (zsn603_handle_t handle)
2    {
3        unsigned char   info[12] = {0};
4        unsigned char   ret = 0;
5        /*  在使用 B 卡相关的函数前，需设置协议为 B 类卡  */
6        ret = zsn603_set_ios_type (handle, ZSN603_ICC_ISO_TYPE_B);
```

```
7         if(ret   !=   0 ){
8               am_kprintf("ios set fail beacuse %0x2", ret);
9               return;
10        }
11        //B 卡请求接口函数   请求模式为  0x00   IDLE
12        ret = zsn603_piccb_active(handle, 0x00, info);
13        if(ret == 0){
14              unsigned char i = 0;
15              printf("CARD INFO    is :");
16              for(i = 0; i < 12 ; i ++ ){
17                    am_kprintf("%02x ",    info[i]);
18             }
19              printf("\r\n");
20        }else{
21              printf("active fail beacuse error 0x%02x", ret);
22        }
23  }
```

该应用程序为 B 类卡激活函数，其入口函数为：demo_zsn603_piccb_active_test_entry()，在程序清单 3.11 中，程序调用 zsn603_piccb_active()函数，函数返回值保存在 ret 中，若 ret 等于 0，则表示命令主机发送成功、从机并成功执行命令帧，该例程 打印了激活操作返回的相关信息；若 ret 为任意非 0 值，则会打印 ret（即对应错误号），具体错误标识在 zsn603.h 头文件中被定义。

3.6.4 自动检测模式测试

ZSN603 提供卡片自动检测功能，程序清单 3.12 实现了卡自动检测的范例。

程序清单 3.12 在多任务环境中使用 ZSN603

```
1   static int a = 0;
2   //有卡检测到回调函数
3   void __card_input(void *p_arg){
4         a = 1;
5   }
6   void demo_zsn603_auto_detect_test_entry(zsn603_handle_t handle)
7   {
8         uint8_t ret;
9         zsn603_auto_detect_ctrl_t      auto_ctrl;
10        zsn603_auto_detect_data_t    auto_data;
11        //自动检测模式配置
12        auto_ctrl.ad_mode              =     ZSN603_AUTO_DETECT_CONTINUE |    //检测到卡后继续检测
```

```
13                                              ZSN603_AUTO_DETECT_INTERRUPT |   //检测到卡后发生中断
14                                              ZSN603_AUTO_DETECT_SEND;         //检测到卡后自动发送
15          auto_ctrl.tx_mode            = ZSN603_ANT_MODE_TX12;
16          auto_ctrl.req_code           = ZSN603_MIFARE_REQUEST_IDLE;
17          auto_ctrl.auth_mode          = ZSN603_AUTO_DETECT_KEY_AUTH;
18          auto_ctrl.key_type           = ZSN603_ICC_KEY_TYPE_A;
19          auto_ctrl.p_key              = data;
20          auto_ctrl.key_len            = 6;
21          auto_ctrl.block              = 4;
22          auto_ctrl.pfn_card_input = __card_input;
23          auto_ctrl.p_arg              = NULL;
24
25          ret = zsn603_auto_detect(handle, &auto_ctrl);
26          if(ret == 0){
27              printf("Entry auto detect card mode success!\r\n");
28          }else{
29              printf("Entry auto detect card mode fail beacuse error 0x%02x!\r\n", ret);
30              return ;
31          }
32          while(a == 0);
33          ret = zsn603_get_auto_detect(handle, 0, &auto_data);
34          a = 0;
35          if(ret == 0){
36              printf("Auto detect card success!\r\n");
37          }else{
38              printf("Auto detect card fail beacuse error 0x%2x!\r\n", ret);
39          }
40      }
```

该应用程序为自动检测模式测试，其入口函数为：demo_zsn603_auto_detect_test_entry ()，在程序清单 3.12 中，程序调用 zsn603_auto_detect()函数，使 ZSN603 进入自动检测模式，且自动检测模式的配置可以自由配置，若进入成功则会打印相应提示信息，值得注意的是，auto_ctrl 中 pfn_card_input 成员，该成员是一个函数指针，用以通知用户卡被检测到，该函数由用户实现，同时由于该函数是在发生中断时被调用，所以在该函数中不能做太多工作，只能作为一个通知作用。在检测到卡后，用户可以调用 zsn603_get_auto_detect()函数来获取自动读卡数据，并且可以配置 ZSN603 是否在继续进行检测。

3.7 多任务环境下的使用

ZSN603 软件包提供的接口仅仅是对 ZSN603 原生功能的封装，没有考虑多任务应用场合，即多个任务均访问了 ZSN603 的情况。因此，在多任务环境中，若多个任务同时使用到

了 ZSN603，则应在它们之间增加互斥机制。简单地，可以使用互斥信号量实现这一功能，示意代码详见程序清单 3.13。

程序清单 3.13　在多任务环境中使用 ZSN603

```
1    mutex_t    mutex_zsn603;   // 定义互斥量
2
3    void task1_entry (void)
4    {
5        while(1) {
6            mutex_take(&mutex_ zsn603);                                       // 获取信号量
7            zsn603_control_led (handle,   ZSN603_CONTROL_LED_ON);        // 关闭 LED 灯
8            mutex_give(&mutex_ zsn603);                                        // 释放信号量
9            // 其它操作
10       }
11   }
12   void task2_entry (void)
13   {
14       while(1) {
15           mutex_take(&mutex_ zsn603)                                       // 获取信号量
16           zsn603_control_led (handle,   ZSN603_CONTROL_LED_OFF);       // 打开 LED 灯
17           mutex_give(&mutex_ zsn603);                                        // 释放信号量
18           // 其它操作
19       }
20   }
21   void main ()
22   {
23       mutex_init(&mutex_ zsn603);                                             // 信号量初始化
24       // 创建并启动两个任务
25   }
```

值得注意的是，即使在裸机平台中，这个问题依然需要注意。虽然裸机平台没有多任务的概念，大部分程序在主循环中运行，但中断的产生可以打断主循环的运行，因此，不能在中断或主循环中同时访问 ZSN603，一般来讲，都不建议在中断中使用 ZSN603 相关的接口。

第4章 在 AMetal 中使用 ZSN603

✍本章导读

AMetal 平台已经完成了对 ZSN603 的适配，可以在初始化之后，使用 ZSN603 通用驱动软件包中提供的各个接口操作 ZSN603，使用 ZSN603 提供的相应的 RFID 功能，无需关心任何底层细节，使用户可以更加快捷的将 ZSN603 应用到实际项目中。

4.1 使用 ZSN603 通用软件包接口

在第三章中提到，AMetal 已经完成了对 ZSN603 通用软件包的适配，若用户选用 AMetal 平台，则无需自行适配，直接使用相应的功能接口操作 ZSN603 即可。

4.1.1 实例初始化函数

通过 3.5 节中的内容可知，所有的功能接口均需传入一个 handle 参数，用以指定操作的 ZSN603 对象，当使用 AMetal 时，用户可以根据自己所使用的通信方式直接通过 ZSN603 的实例初始化函数获得相应的 handle，其函数原型为：

```
zsn603_handle_t    am_zsn603_uart_inst_init (void);          // ZSN603 实例初始化函数（UART 模式）
zsn603_handle_t    am_zsn603_i2c_inst_init (void);           // ZSN603 实例初始化函数（I2C 模式）
```

UART 通信方式实例初始化函数调用形式如下：

```
zsn603_handle_t    handle    =    am_zsn603_uart_inst_init();
```

I^2C 通信方式实例初始化函数调用形式如下：

```
zsn603_handle_t    handle    =    am_zsn603_i2c_inst_init();
```

此处获取的 handle 即可用于 ZSN603 通用软件包提供的各个功能接口。

4.1.2 配置

通常情况下，实例初始化函数在模板工程中的 ZSN603 配置文件中实现，配置文件的默认名称为： am_hwconf_zsn603_uart.c 与 am_hwconf_zsn603_i2c.c，以使用 ZLG116 驱动 ZSN603 为例，文件的具体内容详见程序清单 4.1。

程序清单 4.1 am_hwconf_zsn603_uart.c 文件内容示意

```
1    #include "ametal.h"
2    #include "am_i2c.h"
3    #include "zsn603.h"
4    #include "zlg116_pin.h"
5    #include "am_zsn603.h"
6    #include "zsn603_platform.h"
7    #include "am_zlg116_inst_init.h"
8    static      zsn603_uart_dev_t            __g_zsn603_uart_dev;    // ZSN603(UART 模式) 设备实例
9    //   ZSN603(UART 模式) 设备信息
10   static const zsn603_uart_devinfo_t    __g_uart_info = {
11           0xb2,
12         {
13               -1,
14               PIOA_12,
```

```
15                am_zlg116_uart1_inst_init,
16                am_zlg116_uart1_inst_deinit,
17                9600,
18            }
19    };
20    //   ZSN603  实例初始化，获得 ZSN603 标准服务句柄(UART 模式) */
21    zsn603_handle_t    am_zsn603_uart_inst_init (void)
22    {
23        return zsn603_uart_init(&__g_zsn603_uart_dev, &__g_uart_info);
24    }
```

程序清单 4.2　am_hwconf_zsn603_i2c.c 文件内容示意

```
1     #include "ametal.h"
2     #include "am_i2c.h"
3     #include "zsn603.h"
4     #include "zlg116_pin.h"
5     #include "am_zsn603.h"
6     #include "zsn603_platform.h"
7     #include "am_zlg116_inst_init.h"
8     #include "am_hwconf_zsn603_i2c.h"
9     static    zsn603_i2c_dev_t     __g_zsn603_i2c_dev;              // ZSN603(I2C 模式)  设备实例
10    //   ZSN603(I2C 模式)  设备信息
11    static const zsn603_i2c_devinfo_t    __g_i2c_info = {
12        0xb2,
13        {
14            -1,
15            PIOA_12,
16            am_zlg116_i2c1_inst_init,
17            am_zlg116_i2c1_inst_deinit
18        }
19    };
20    //   ZSN603 实例初始化，获得 ZSN603 标准服务句柄(i2c 模式)
21    zsn603_handle_t    am_zsn603_i2c_inst_init (void)
22    {
23        return zsn603_i2c_init(&__g_zsn603_i2c_dev, &__g_i2c_info);
24    }
```

注：若在用户所使用的工程中不存在该配置文件，用户可以根据实际所需要的通信方式，将程序清单 4.1 或程序清单 4.2 中的内容原封不动的复制到一个新文件中（新文件可命名为 am_hwconf_zsn603.c），完成配置文件的添加。

之所以将这些内容放在配置文件中，主要是为了方便用户根据实际情况对文件中的部分配置信息作相应的修改。每一个可修改的地方视为一个配置项，对于 UART 用户可对其中的 6 个配置项可以进行修改，而 I^2C 模式则有 5 个配置项可以修改，各配置项的功能简介详见表 4.1 与表 4.1，这些配置项都可以根据实际情况修改。

表 4.1 ZSN603 UART 模式配置项

序号	对应程序清单 4.1 的行号	配置项	备注
1	12	Local address	0xb2（默认出厂 local address）
2	14	复位引脚	-1，默认使用硬件进行复位电路
3	15	工作模式	中断引脚
4	16	串口初始化函数	可修改此函数选择不同的硬件端口
5	17	串口解初始化函数	要与 3 配置项的端口项对应
6	18	串口初始波特率	9600（默认出厂波特率）

表 4.2 ZSN603 UART 模式配置项

序号	对应程序清单 4.2 的行号	配置项	备注
1	12	Local address	0xb2（默认出厂 local address）
2	14	复位引脚	-1，默认使用硬件进行复位电路
3	15	工作模式	中断引脚
4	16	串口初始化函数	可修改此函数选择不同的硬件端口
5	17	串口解初始化函数	要与 3 配置项的端口项对应

4.1.3 应用

当调用实例初始化函数获得 ZSN603 句柄时，即可使用该句柄操作 ZSN603，在 3.6 节中，根据各个接口编写了一个用于测试的简单应用程序，由于接口的通用性，当使用 AMetal 平台时，同样可以使用这段测试程序，范例程序详见程序清单 4.3。

程序清单 4.3 使用 ZSN603 通用测试程序进行测试

```
1   #include "ametal.h"
2   #include "zsn603.h"
3   #include "am_zsn603.h"
4   extern zsn603_handle_t    zsn603_uart_inst_init (void);
5   int am_main(void)
6   {
7          zsn603_handle_t handle = zsn603_uart_inst_init();
8          demo_zsn603_led_test_entry (handle);
9          while(1){
10         }
11  }
```

由此可见，通过实例初始化函数可以快捷的获取到 ZSN603 句柄，该句柄可直接用于通用软件包提供的各个功能接口。

第5章 在 AWorks 中使用 ZSN603

✍本章导读

AWorks 平台已经完成了对 ZSN603 的适配，可以在使能设备之后，使用 ZSN603 通用驱动软件包中提供的各个接口操作 ZSN603，使用其对应的 RFID 功能，无需关心任何底层细节，使用户可以更加快捷的将 ZSN603 应用到实际项目中。

5.1 设备使能及配置

在使用 ZSN603 之前，必须使能 ZSN603 硬件设备，并完成相关的配置。本文将以 MM32F103 为例，对 AWorks 平台下 ZSN603 的适配进行详细介绍。

5.1.1 设备使能

设备使能的方法为：确保在 aw_prj_params.h 文件中定义的 AW_DEV_I2C_ZSN603 宏或 AW_DEV_UART_ZSN603 宏处于有效状态，即未被注释。

```
#define    AW_DEV_I2C_ZSN603
#define    AW_DEV_UART_ZSN603
```

若 aw_prj_params.h 文件中没有定义该宏，则可以自行添加该宏的定义。通常情况下，若未定义该宏，表明用户所使用的模板工程没有添加 ZSN603 设备的默认配置，此时，用户还需添加相应的配置文件（将在下节进行详细介绍）。

5.1.2 设备配置

设备相关的配置集中在用户配置文件目录（user_config\awbl_hwconf_usrcfg\）下的 awbl_hwconf_i2c_zsn603.h 和 awbl_hwconf_uart_zsn603.h 文件中，文件的示意内容详见程序清单 5.1 与程序清单 5.2。

程序清单 5.1　awbl_hwconf_uart_zsn603.h 文件内容

```
1     #ifndef __AWBL_HWCONF_ZSN603_UART_H__
2     #define __AWBL_HWCONF_ZSN603_UART_H__
3
#include "mm32f103_reg_base.h"
4     #include "mm32f103_pin.h"
5     #include "mm32f103_inum.h"
6     #include "driver/gpio/awbl_mm32f103_gpio_private.h"
7     #include "awbl_zsn603_uart.h"
8     #include "zsn603.h"
9     #include "aw_prj_params.h"
10    #include "aw_serial.h"
11
12    #ifdef     AW_DEV_UART_ZSN603
13
14    // ZSN603 （UART 模式）设备信息
```

```
15  aw_local aw_const awbl_zsn603_uart_devinfo_t __g_zsn603_uart_devinfo = {
16      0xb2,                                                   // 初始 local address
17      9600,                                                   // 初始串口波特率
18      MM32F103_UART1_COMID,                                   // 串口硬件端口号
19      PIOC_5                                                  // 中断引脚号
20  };
21  // 设备实例内存静态分配
22  aw_local awbl_zsn603_uart_dev_t __g_zsn603_uart_dev;
23
24  #define AWBL_HWCONF_UART_ZSN603         \
25      {                                   \
26          AWBL_ZSN603_UART_NAME,          \
27          0,                              \
28          AWBL_BUSID_PLB,                 \
29          0,                              \
30          &__g_zsn603_uart_dev.super,     \
31          &__g_zsn603_uart_devinfo        \
32      },
33  #else
34  #define   AWBL_HWCONF_UART_ZSN603
35  #endif
36  #endif
```

程序清单 5.2　awbl_hwconf_i2c_zsn603.h 文件内容

```
1   #ifndef __AWBL_HWCONF_ZSN603_I2C_H__
2   #define __AWBL_HWCONF_ZSN603_I2C_H__
3   #include "mm32f103_pin.h"
4   #include "mm32f103_inum.h"
5   #include "driver/gpio/awbl_mm32f103_gpio_private.h"
6   #include "awbl_zsn603_i2c.h"
7   #include "zsn603.h"
8   #include "aw_prj_params.h"
9   #ifdef    AW_DEV_I2C_ZSN603
10  /* ZSN603 (I2C 模式) 设备信息 */
11  aw_local aw_const awbl_zsn603_i2c_devinfo_t __g_zsn603_i2c_devinfo = {
12          0xb2,
13          PIOA_15
14  };
15  aw_local awbl_zsn603_i2c_dev_t __g_zsn603_i2c_dev;    // ZSN603 (I2C mode) 设备实例内存静态分配
16
17  #define AWBL_HWCONF_I2C_ZSN603          \
```

```
18          {                                              \
19              AWBL_ZSN603_I2C_NAME,                      \
20              0,                                         \
21              AWBL_BUSID_I2C,                            \
22              MM32F103_I2C1_BUSID,                       \
23              &__g_zsn603_i2c_dev.super.super,           \
24              &__g_zsn603_i2c_devinfo                    \
25          },
26      #else
27      #define AWBL_HWCONF_I2C_ZSN603
28      #endif
29      #endif
```

特别地，若在用户获取的 SDK 中，不包含此文件，可以根据用户所需的通信方式，自行新建一个名为 awbl_hwconf_i2c_zsn603.h 或 awbl_hwconf_uart_zsn603.h 的文件，并将程序清单 5.2 或程序清单 5.2 中的内容作为新建文件的内容。

接下来，首先对这个文件的作用进行一个整体的分析，然后再对其中用户可能需要修改的内容（配置项）作详细介绍。

1. 文件作用整体分析

以 ZSN603 设备 UART 模式为例，该文件的主要作用是完成 ZSN603 的配置，并对外提供一个设备宏定义（AWBL_HWCONF_UART_ZSN603）。仅关注与定义该宏相关的语句，忽略其它代码，即：

```
#ifdef    AW_DEV_UART_ZSN603
// 其它信息...
#define   AWBL_HWCONF_UART_ZSN603                              \
    {                                                          \
          AWBL_ZSN603_UART_NAME,                               \
          // 其它信息...
    },
#else
#define   AWBL_HWCONF_UART_ZSN603
#endif
```

由此可见，该宏可能有两种定义，具体定义被 AW_DEV_UART_ZSN603 宏控制，仅当 AW_DEV_UART_ZSN603 被有效定义时，AWBL_HWCONF_UART_ZSN603 的定义才包含实际内容，否则，AWBL_HWCONF_UART_ZSN603 是一个内容为空的宏。这实际上也进一步展示了为什么 AW_DEV_UART_ZSN603 可以作为设备使能宏。

一个硬件设备要正常工作，必须将其对应的设备宏加入到 AWorks 指定的硬件设备列表中，硬件设备列表在 awbus_lite_hwconf_usrcfg.c 文件中定义，即一个名为：g_awbl_devhcf_list[] 的数组，该数组的每一个成员都描述了系统中的一个硬件设备。示例片段详见程序清单 5.3。

程序清单 5.3 硬件设备列表（awbus_lite_hwconf_usrcfg.c）

```
1       /* 硬件设备列表 */
2       aw_const struct awbl_devhcf g_awbl_devhcf_list[] = {
```

```
3          AWBL_HWCONF_MM32F103_NVIC                /*  nvic   */
4          AWBL_HWCONF_MM32F103_GPIO                /*  GPIO   */
5          /* ....其他硬件设备 */
6     };
```

在 g_awbl_devhcf_list[ ]数组中，每个元素都是以“AWBL_HWCONF_”作为前缀的一个宏，该宏本质上完成了一个设备描述的定义。例如，要使用 ZSN603，则应该将 ZSN603 对应的设备宏加入到硬件设备列表中，详见程序清单 5.4 。

程序清单 5.4　硬件设备列表（awbus_lite_hwconf_usrcfg.c）

```
1     /* 硬件设备列表 */
2     aw_const struct awbl_devhcf g_awbl_devhcf_list[] = {
3          AWBL_HWCONF_MM32F103_NVIC                /*  NVIC   */
4          AWBL_HWCONF_MM32F103_GPIO                /*  GPIO   */
5          AWBL_HWCONF_UART_ZSN603                  /*  ZSN603（UART mode）  */
6          AWBL_HWCONF_I2C_ZSN603                   /*  ZSN603（I2C mode）  */
7          /* ....其他硬件设备 */
8     };
```

通常情况下，若在系统工程中存在 ZSN603 的配置文件，则该宏默认已经加入到了硬件设备列表中，用户只需要用过使能宏 AW_DEV_I2C_ZSN603 或 AW_DEV_UART_ZSN603 宏控制设备是否使能即可。

2．　配置项详解

在使用 ZSN603 过程中，用户可能需要根据实际情况修改配置文件中的一些信息，例如例如 ZSN603 的 local_address、硬件端口号、中断引脚号等。在 UART 模式下，用户可以修改的地方有四个，可视为四个可配置项，具体见表 5.1；而在 I^2C 模式下，用户可以修改的地方有三个，可视为三个配置项，具体详见表 5.2。

表 5.1　ZSN603（UART 模式）配置项

序号	对应程序清单 5.1 的行号	配置项	默认值
1	16	设备地址（local_address）	0xb2
2	17	初始波特率	9600
3	18	硬件端口号	MM32F103_UART1_COMID
4	19	中断引脚	PIOC_5
5	27	设备单元号	0

表 5.2　ZSN603（I^2C 模式）配置项

序号	对应程序清单 5.2 的行号	配置项	默认值
1	12	设备地址（local_address）	0xb2
2	13	中断引脚	PIOA_15
3	21	所处总线	MM32F103_I2C1_BUSID
4	20	设备单元号	0

表 5.1 中序号 1、4 与表 5.2 中序号 1、2 的配置项是完全一致的，都是需要根据用户的硬件电路以及对应 ZSN603 设备的相关信息进行重新配置。

表 5.1 中序号 2、3 配置项为 UART 特有的配置项。波特率为用户配置的初始波特率，该值在 ZSN603 出厂设置为 9600，若用户对其进行修改，则在下次下载程序时，需对该值进行修改。硬件端口号为描述 ZSN603 使用的是哪个串口硬件端口，以便于主控 MCU 使用相应的接口进行串口通信。串口端口在 aw_prj_params.h 文件中定义。例如，在 MM32F103 硬件平台中，总共有 8 个串口：MM32F103_UART1_COMID~ MM32F103_UART8_COMID，其对应的串口索引为 COM0~COM7，定义详见程序清单 5.5。

程序清单 5.5 UART 索引定义

```
1   //  UART ID 分配
2   #define MM32F103_UART1_COMID          COM0
3   #define MM32F103_UART2_COMID          COM1
4   #define MM32F103_UART3_COMID          COM2
5   #define MM32F103_UART4_COMID          COM3
6   #define MM32F103_UART5_COMID          COM4
7   #define MM32F103_UART6_COMID          COM5
8   #define MM32F103_UART7_COMID          COM6
9   #define MM32F103_UART8_COMID          COM7
```

表 5.2 中的序号 3 配置项为 I^2C 模式特有的配置项。所处的总线配置是为了描述 ZSN603 挂在哪条 I^2C 总线上，以便主控 MCU 使用相应的 I^2C 总线与 ZSN603 通信。每条 I^2C 总线对应的编号在 aw_prj_params.h 文件中定义。例如，在 MM32F103 硬件平台中，默认有 2 条硬件 I^2C 总线：I^2C1、I^2C2，它们对应的总线编号为 0 和 1，定义详见程序清单 5.6。

程序清单 5.6 I2C 总线编号定义

```
1   //  I2C ID 分配
2   #define      MM32F103_I2C1_BUSID          0
3   #define      MM32F103_I2C2_BUSID          1
```

默认使用了 I^2C1 与 ZSN603 通信，若在硬件电路设计时，ZSN603 的通信接口连接至了 I^2C2，则可以将配置项修改为：MM32F103_I2C2_BUSID。

表 5.1 中序号 5 与表 5.2 中序号 4 为设备单元号。在 AWorks 中，设备单元号用以区分几个相同的硬件设备，例如，系统通过一条或多条 I^2C 总线挂载了(N+1)个 ZSN603，则单元号可能分别为 0 ~ N。若存在多个 ZSN603，则每个 ZSN603 都对应一个配置文件，各自享有一套独立的配置。一般来讲，若只连接了一个 ZSN603，则单元号固定为 0。

5.2 使用 ZSN603 通用软件包接口

在第三章中提到，AWorks 已经完成了对 ZSN603 通用软件包的适配，若用户选用 AWorks 平台，则无需自行适配，直接使用相应的功能接口操作 ZSN603 即可。

通过 0 节中的内容可知，所有的功能接口均需传入一个 handle 参数，用以指定操作的 ZSN603 对象，在使用 AWorks 时，可以直接通过 handle 获取接口获得，其函数原型为（保存在 awbl_zsn603_i2c.h 、awbl_zsn603_uart.h 文件）：

```
zsn603_handle_t    awbl_zsn603_i2c_handle_get(int    unit);              // 获取设备号为 unit 的 ZSN603(I2C)
zsn603_handle_t    awbl_zsn603_uart_handle_get(int    unit);             // 获取设备号为 unit 的 ZSN603 (UART)
```

在获取时，需要传入一个单元号，用以指定获取哪个 ZSN603 对应的 handle，该值与设备配置中“设备单元号”配置项相对应。一般情况下，只连接单个 ZSN603 时，单元号默认为 0。基于此，获取 handle 的语句即为：

```
zsn603_handle_t    handle = awbl_zsn603_i2c_handle_get(0);           // 获取设备号为 0 的 ZSN603(I2C)
zsn603_handle_t    handle = awbl_zsn603_uart_handle_get(0);          // 获取设备号为 0 的 ZSN603 (UART)
```

当调用实例初始化函数获得 ZSN603 句柄时，即可使用该句柄操作 ZSN603，在 0 节中，根据各个接口编写了一个用于测试的简单应用程序，由于接口的通用性，当使用 AWorks 平台时，同样可以使用这段测试程序，范例程序详见程序清单 5.7。

程序清单 5.7　使用 ZSN603（UART）测试程序进行测试

```
1    #include "aworks.h"
2    #include "zsn603.h"
3    #include "awbl_zsn603_uart.h"
4    int aw_main(void)
5    {
1        zsn603_handle_t    handle =awbl_zsn603_uart_handle_get(0);      // 获取 ZSN603 句柄（UART）
6        demo_zsn603_led_test_entry (handle);                            // 调用 LED 测试函数
7        while(1){
8        }
9    }
```

以上例程为 UART 方式，若要使用 I^{2}C 方式，直接将程序清单 5.7 中第 3、7 行代码进行修改即可，详见程序清单 5.8。

程序清单 5.8　使用 ZSN603（I^{2}C）测试程序进行测试

```
2    #include "aworks.h"
3    #include "zsn603.h"
4    #include "awbl_zsn603_i2c.h"
5    int aw_main(void)
6    {
7        zsn603_handle_t    handle    =    awbl_zsn603_i2c_handle_get(0);        // 获取 ZSN603 句柄（I2C）
8        demo_zsn603_led_test_entry (handle);                                      // 调用 LED 测试函数
9        while(1){
10       }
11   }
```

第6章 在 Linux 中使用 ZSN603

✍本章导读

Linux 平台已经完成了对 ZSN603 的适配，可以在使能设备之后，使用 ZSN603 通用驱动软件包中提供的各个接口操作 ZSN603，使用其对应的 RFID 功能，无需关心任何底层细节，使用户可以更加快捷的将 ZSN603 应用到实际项目中。

6.1 使用 ZSN603 通用软件包接口

在第三章中提到，Linux 已经完成了对 ZSN603 通用软件包的适配，若用户选用 Linux 平台，则无需自行适配，直接使用相应的功能接口操作 ZSN603 即可。

6.1.1 实例初始化函数

通过 0 节中的内容可知，所有的功能接口均需传入一个 handle 参数，用以指定操作的 ZSN603 对象。当使用 Linux 时，用户可以根据自己所使用的通信方式直接通过 ZSN603 的实例初始化函数获得相应的 handle。

1. 函数原型

Linux 下 ZSN603 实例化函数原型为（保存在 get_handle_t.h 文件中）：

```
zsn603_handle_t    linux_zsn603_uart_handle_get(int fd, int baudrate, char local_addr, int gpio, zsn603_uart_dev_t
*p_dev);             // ZSN603 实例初始化函数(UART 模式)
zsn603_handle_t    linux_zsn603_i2c_handle_get(int fd, char local_addr, int gpio, zsn603_i2c_dev_t *p_dev);
                     // ZSN603 实例初始化函数 (I2C)
```

该两种通信模式初始化的函数均定义在 get_handle_t.c 文件中，函数的内部主要做了两件事，一是对所用的通信接口（uart 和 i2c）进行初始化，二是对 ZSN603 的操作句柄进行相关内容初始化并返回。

2. 配置参数

在获取 handle 时，需要传入相关的配置参数，用以初始化并获取 ZSN603 对应的 handle。由于通信模式不同，其对应模式初始化的配置参数也是不同的。以下将对两种通信模式的参数配置进行说明。

ZSN603 模块的 uart 通信模式下的实例初始化函数参数，见表 6.1。

表 6.1 UART 模式的实例初始化函数参数

序号	函数参数	简要描述	默认值
1	fd	串口句柄（通过“open”串口设备文件获得）	—
2	baudrate	初始波特率	9600
3	localaddr	设备地址	0xb2
4	gpio	中断 GPIO 引脚序号	—
5	p_dev	zsn603_uart_dev_t 结构体指针	—

- fd

初始化参数 fd 为用户打开串口设备获取的句柄。Linux 的串口表现为设备文件。Linux 的串口设备文件命名一般为/dev/ttySn（n=0、1、2……），若是串口是 USB 扩展的，则串口设备文件命名多为/dev/ttyUSBn（n=0、1、2……）。串口设备的命名没有统一的标准，不同

的硬件平台对串口设备文件的命名也有所区别，所以用户需要根据自己所使用的平台打开串口设备以获取句柄。

- baudrate

串口设置波特率参数。在 Linux 平台下对 ZSN603 所支持的波特率分别有 9600、19200、38400、57600、115200、230400。在传入其他波特率值的情况下，串口将默认设置为 9600 波特率。

- localaddr

ZSN603 设备从机地址，用户可以直接以 16 进制形式写入，如模块出厂默认 localaddr 为“0xb2”。

- gpio

硬件中断引脚序号，当用户传入 ZSN603 模块中断引脚所接入的主机 gpio 引脚序号时，可以设置并启用其中断模式。gpio 引脚序号可以通过如下宏定义获得（该定义在 get_handle_t.h 文件中）。

```
/* Convert GPIO signal to GPIO pin number */
#define    GPIO_PIN(bank, gpio)    (32 * (bank) + (gpio))
```

当该参数设置为“-1”时，表示不设置不启用其中断模式。

- p_dev

zsn603_uart_dev_t 结构体指针，用户可以先在函数外定义一个 zsn603_uart_dev_t 结构体，再将该结构体指针传入。

ZSN603 模块的 i2c 通信模式下的实例初始化函数参数，见表 6.2。

表 6.2 I2C 接口的实例初始化函数参数

序号	函数参数	简要描述	默认值
1	fd	i2c 接口句柄（通过“open”i2c 设备文件获得）	—
3	localaddr	设备地址	0xb2
4	gpio	中断 GPIO 引脚序号	—
5	p_dev	zsn603_i2c_dev_t 结构体指针	—

- **fd**

初始化参数 fd 为用户打开 i2c 设备获取的句柄。用户需要根据自己所使用的平台的 i2c 设备以获取句柄。例如在 M335x 平台进行模块适配过程中，所打开的 i2c 设备文件为“/dev/i2c-3”。

- **localaddr**

ZSN603 设备从机地址，用户可以直接以 16 进制形式写入，如模块出厂默认 localaddr 为“0xb2”。

- **gpio**

硬件中断引脚序号，该参数在 I2C 模式下是必须要设置的。当用户传入 ZSN603 模块中断引脚所接入的主机 gpio 引脚序号时，可以设置并启用其中断模式。gpio 引脚序号同样可以通过如下宏定义获得（该定义在 get_handle_t.h 文件中）。

```
/* Convert GPIO signal to GPIO pin number */
#define    GPIO_PIN(bank, gpio)    (32 * (bank) + (gpio))
```

- **p_dev**

zsn603_uart_dev_t 结构体指针，用户可以先在函数外定义一个 zsn603_uart_dev_t 结构体，再将该结构体指针传入。

3. 函数调用示例

在 M335x 平台适配过程上，UART 通信方式实例初始化函数调用形式如下。

```
zsn603_uart_dev_t    p_dev;                                          /*  定义一个 zsn603_uart_dev_t 结构体   */
int    fd = open("/dev/ttyO1",O_RDWR);                                /*  获取串口设备句柄                    */
zsn603_handle_t    handle    =    linux_zsn603_uart_handle_get( fd , 9600 , 0xb2 , GPIO_PIN(1, 31) , &p_dev );
```

在 M335x 平台适配过程上，I^2C 通信方式实例初始化函数调用形式如下：

```
zsn603_i2c_dev_t    p_dev;                                           /*  定义一个 zsn603_uart_dev_t 结构体   */
int    fd = open("/dev/i2c-3" , O_RDWR);                              /*  获取 i2c 设备句柄                   */
zsn603_handle_t    handle    =    linux_zsn603_i2c_handle_get( fd , 0xb2 ,GPIO_PIN(1, 31) , &p_dev );
```

此处获取的 handle 即可用于 ZSN603 通用软件包提供的各个功能接口。

6.1.2 应用

在 Linux 中，调用实例初始化函数获得 ZSN603 句柄，即可使用该句柄操作 ZSN603，在 0 节中，根据各个接口编写了一个用于测试的简单应用程序，由于接口的通用性，当使用 Linux 平台时，同样可以使用这段测试程序，范例程序详见程序清单 6.1。

程序清单 6.1 Linux 平台使用 ZSN603 测试程序范例

```
 1 #include <unistd.h>
 2 #include "zsn603.h"
 3 #include "zsn603_linux_platform.h"
 4 #include "get_handle_t.h"
 5 #include "demo_zsn603_entries.h"
 6
 7 #define UART 1                                                  /* uart 接口测试使能（写 1 使能）        */
 8 #define I2C    0                                                /* i2c 接口测试使能（写 1 使能）         */
 9
10 int main(void)
11 {
12        int fd;
13 #if    UART
14        zsn603_uart_dev_t p_dev;                                 /*  定义一个 zsn603_uart_dev_t 结构体   */
15        fd = open("/dev/ttyO1",O_RDWR);                          /*  获取串口设备句柄                    */
16        if (fd < 0){
17              printf("fd fail\n");
18              return -1;
19        }
```

```
20      //获取 uart 模式下的 ZSN603 handle
21      zsn603_handle_t handle = linux_zsn603_uart_handle_get(fd, 9600, 0xb2, GPIO_PIN(1, 31), &p_dev);
22      printf("Get uart handle success!\n");
23 #endif
24 #if   I2C
25      zsn603_i2c_dev_t p_dev;                          /*  定义一个 zsn603_uart_dev_t 结构体   */
26      fd = open("/dev/i2c-3" , O_RDWR);                /*  获取 i2c 设备句柄                   */
27      if (fd < 0){
28          printf("fd fail\n");
29          return -1;
30      }
31      //获取 i2c 模式下 ZSN603 handle
32      zsn603_handle_t handle = linux_zsn603_i2c_handle_get(fd, 0xb2, GPIO_PIN(1, 31), &p_dev);
33      printf("Get i2c handle success!\n");
34 #endif
35      usleep(10000);//10ms
36
37      demo_zsn603_led_test_entry (handle);             /* LED 测试函数                        */
38      usleep(10000);
39
40 //   demo_zsn603_picca_active_test_entry(handle);     /*  激活 A 类卡测试函数                 */
41 //   usleep(10000);
42 //   demo_zsn603_piccb_active_test_entry(handle);     /*  激活 B 类卡测试函数                 */
43 //   usleep(10000);
44
45      demo_zsn603_auto_detect_test_entry(handle);      /*  自动检测卡函数                      */
46      usleep(10000);
47      return 0;
48 }
```

上述范例程序，综合了 ZSN603 模块 uart 和 i2c 两种通信方式的使用。在对该测试程序编译时，对宏“UART”或“I2C”定义“1”操作进行使能对应接口通信的使用，需注意的是，编译该程序测试时不可同时将“UART”、“I2C”宏定义为“1”。

由于两种通信模式的中断都采用了线程软中断机制实现，因此编译程序时应添加“-lpthread”参数进行编译，如下所示。

```
/* M335x  平台的编译链为 arm-none-linux-gnueabi-gcc */
arm-none-linux-gnueabi-gcc    *.c    -o    uart    -lpthread
```